INTEGRACIÓN DE TÉCNICAS DE INTERACCIÓN NATURAL Y MULTITÁCTIL

Proyecto de grado

Integrantes

Vanessa Berazategui Silva

Matias Nassi Correa

Tutor

MSc. Ing. Christian Clark

MEDIALAB
Instituto de Computación
Facultad de Ingeniería
Universidad de la República
Montevideo - Uruguay

Mayo, 2017

Resumen

Interacción natural refiere a un conjunto de técnicas de Interacción Persona-Computadora que, al basarse en componentes naturales del usuario y su entorno, poseen la característica de ser invisibles (p.ej. interfaces basadas en gestos). Por otra parte, multitáctil refiere a aquellos sistemas capaces de ser manipulados mediante el contacto directo (p.ej. la pantalla de una tableta o un teléfono inteligente, una mesa inteligente, etc.).

ANIMuS (A Natural Interactive MUltitouch Surface), como se le llamó a este proyecto, pertenece al área Interacción Persona-Computadora y busca estudiar la integración de técnicas multitáctil e interacción natural, teniendo como objetivo utilizar ambas en simultáneo como principal vía de interacción del usuario con el sistema, aumentando una interfaz multitáctil con interacción natural. Para lograr dicho objetivo, se busca conformar un *framework* sobre el que se puedan desarrollar distintas aplicaciones que se beneficien de este tipo de interacción híbrida. Por lo tanto, la idea conceptual por detrás de este proyecto es obtener una superficie interactiva con la que el usuario pueda interactuar naturalmente, ya sea mediante gestos corporales y/o mediante el contacto directo.

La solución integra varios componentes, tanto de *hardware* como de *software*. Por un lado, se incluyen sensores *Microsoft Kinect* mediante los que se puede hacer un seguimiento de los movimientos corporales del usuario, también conocido como *body-tracking*. Mientras que por otro lado, se cuenta con sensores *Leap Motion* que permiten realizar un seguimiento más preciso de los movimientos de las manos y los dedos del usuario, comúnmente conocidos como *finger-tracking*. De esta forma, se pretende sacar el mayor provecho de las cualidades de cada uno de estos sensores. Adicionalmente, se integran múltiples componentes *software* como bibliotecas de visión por computadora para análisis de imágenes y mapas de profundidad, bibliotecas de aprendizaje automático para reconocimiento de gestos, entre otras.

Este trabajo se enmarca en otro de mayor alcance denominado *CUIN: Computación Ubicua e Interacción Natural*, llevado a cabo por docentes del Laboratorio de Medios del Instituto de Computación (InCo) de la Facultad de Ingeniería de la Universidad de la República en conjunto con investigadores del Centro de Investigación Básica en Psicología (CIBPsi). El proyecto tiene como objetivo investigar y analizar la incorporación de las técnicas anteriormente mencionadas a todos los componentes típicos de una oficina (paredes, mesas, etc.) para obtener una oficina futurista en la que la interacción sea completamente ubicua y lo suficientemente interactiva como para que pueda ser manipulada completamente de forma natural.

Palabras claves: interacción natural, interacción multitáctil, interacción persona-computadora, computación ubicua, *Microsoft Kinect, Leap Motion*, *body-tracking, finger-tracking*.

Dedicamos este trabajo a nuestras familias y amigos, quienes han sido un apoyo fundamental no sólo para este proyecto sino a lo largo de toda nuestra carrera universitaria.

Agradecemos también a Germán Hoffman por su apoyo durante etapas fundamentales del proyecto en lo que respecta al uso de la solución Mapinect, un *framework* de *video mapping* e interacción tangible utilizando sensores *Microsoft Kinect*. A pesar de que éste finalmente no fue utilizado como parte de la solución construida, el conocer más de este *framework* permitió indagar sobre ciertos problemas con los que la solución presentada también debió lidiar.

Por último, queremos agradecer a Klemen Istenič del Instituto de Visión por Computador y Robótica de la Universidad de Girona, por su amabilidad y ayuda en lo que respecta a la implementación de la característica multitáctil del sistema desarrollado. En este sentido, el haber contado con la investigación llevada a cabo como parte de su tesis de grado fue clave para lograr una solución acorde a las necesidades del presente trabajo.

Índice general

Capítulo 1: Introducción

La interacción basada en gestualidad multitáctil, también denominada 2D o bidimensional ha revolucionado la forma con la que los usuarios interactúan con los dispositivos [149]. Hoy en día, su uso se ha incrementado de forma exponencial al igual que la gestualidad 3D o tridimensional [155], también denominada "gestos libres", del inglés *touch free gesture*, y la interacción mediante lenguaje natural, del inglés *Natural Language Interaction (NLI*). Estas técnicas permiten que las personas se relacionen con los dispositivos de una manera más humana, flexible e intuitiva. De esta forma, se logra interactuar con elementos digitales de manera más semejante a la interacción llevada a cabo con los elementos físicos del mundo real, siendo innecesario contar con un elemento externo para lograr la interacción.

El presente proyecto busca estudiar la integración de las técnicas multitáctil e interacción natural con el fin de conformar un *framework* que permita desarrollar distintas aplicaciones que se beneficien de este tipo de interacción híbrida. Así, un usuario podría interactuar con una superficie de forma natural, ya sea mediante gestos corporales o el contacto directo resultado de aumentar una interfaz multitáctil con interacción natural.

En lo que resta del documento se describe el estudio realizado en lo que concierne a la combinación de estas dos formas de interacción, comenzando en este capítulo por describir el proyecto de forma conceptual, detallar los objetivos y la motivación por detrás del trabajo, así como también describir en términos generales los problemas específicos que se debieron resolver para cumplir con las metas propuestas.

1.1. Descripción general del proyecto

En esta sección se detalla la descripción general del proyecto, así como también, un diagrama conceptual presentando los diferentes elementos que lo conforman y cómo estos interactúan entre sí.

La solución a desarrollar surge originalmente como parte de la idea de construir una oficina del futuro, en la cual se puedan manipular todos los elementos que la conforman de forma natural. Este paradigma de interacción es comúnmente denominado computación ubicua, del inglés *ubiquitous computing*[1], que tal como se detalla en el *Capítulo 3*, consiste en integrar de la mayor y mejor forma posible la informática en entornos donde se encuentran las personas, sea una oficina, una casa, una tienda, etc., haciendo que los dispositivos de cómputo no se perciban como objetos en sí mismos sino que la inteligencia esté de cierta forma oculta y sea transparente para los usuarios [18]. Para lograr que los usuarios puedan manipular todos los elementos que conforman la oficina de forma natural y que la inteligencia de cada uno sea transparente, el presente proyecto debe conformar un *framework* que brinde los servicios necesarios para llevar a cabo aplicaciones que permitan interactuar con los elementos de la forma antes mencionada. Dentro de este contexto, el trabajo se centra particularmente en la interacción del usuario con los elementos presentes sobre una superficie plana como puede ser una mesa o una pared, conformando una de las múltiples piezas que componen el rompecabezas que da solución a la oficina del futuro.

El diagrama presentado en la *Figura 1* ilustra el contexto en el que está inmersa la solución a la problemática planteada, incluyendo una referencia a cada uno de los componentes que la conforman, y brindando a su vez una idea de cuál es la principal motivación. Es preciso mencionar que el diagrama es meramente conceptual, habiendo sido incluso generado previo a cualquier tipo de diseño y/o

[1] Otros nombres por los cuales se conoce a este paradigma, los cuales ilustran aún mejor su principal objetivo, son *Things That Think*, *Everyware* e Inteligencia Ambiental

implementación de la solución. Así, el único objetivo de este diagrama es permitir visualizar de mejor forma cómo estaría contextualizada la solución a desarrollar, cómo sería utilizada y qué componentes se deberían incluir con tal fin. A su vez, como se detalla en capítulos posteriores, algunas de las características fueron luego quitadas del alcance del proyecto debido a que, o bien no están fuertemente relacionadas con los objetivos o bien el tiempo y/o los recursos no fueron suficientes para incluirlas como parte de la solución final.

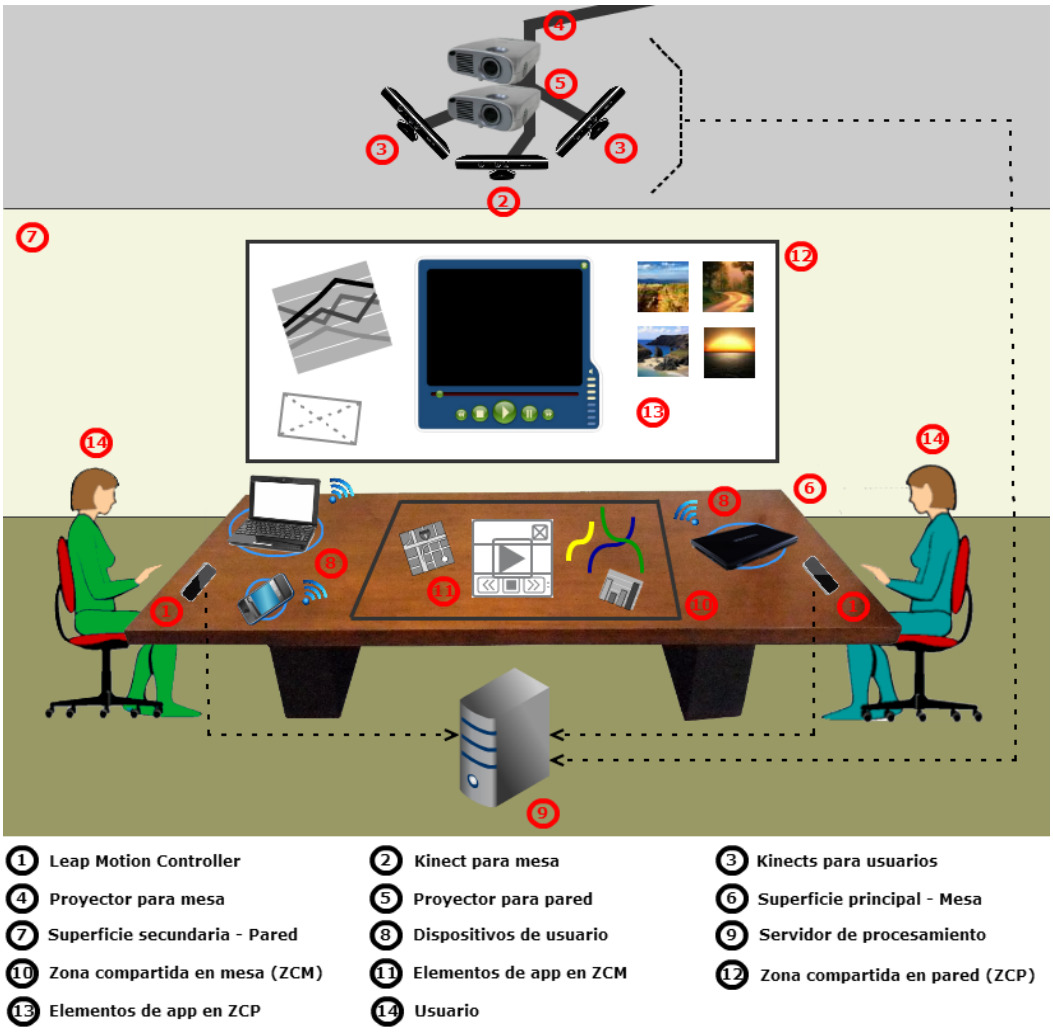


Figura 1: Diagrama conceptual de la solución propuesta

En la *Figura 1* se puede apreciar cómo es el espacio de interacción que forma parte de una posible aplicación que haga uso del *framework*. Por un lado, se observan múltiples usuarios compartiendo un espacio en el cual existen diferentes superficies de interacción (en este caso particular, una mesa y una pared), con las que interactúan de forma natural haciendo uso principalmente de sus manos y brazos. Por otro lado, se observan sensores de diferentes tipos. La utilización de estos diferentes tipos de sensores permite explotar las capacidades de cada uno, e integrarlos con el fin de paliar las carencias que poseen individualmente. Como se detallará en el *Capítulo 4*, este es un esquema fácilmente extensible en cuanto a cantidad de usuarios, sensores, proyectores, computadores y tamaño de la sala.

A continuación se detalla cada uno de los grupos de elementos presentes en el diagrama, presentando para cada uno de ellos la razón por la cual es utilizado, cómo lo utilizan los usuarios, si es que esto aplica, y finalmente si fue o no incluido en la solución final propuesta junto con los argumentos de esta decisión. De todas formas, algunas de las decisiones mencionadas en los siguientes párrafos serán ahondadas y presentadas con más detalle en el *Capítulo 4*.

1.1.1. Superficies de interacción: mesa y pared

Las superficies de interacción representan aquellas superficies o zonas en las que se ve reflejado el resultado de la interacción multitáctil o tridimensional que los usuarios realizan al utilizar el sistema. En el diagrama se incluye tanto una mesa como una pared, por lo que el usuario podría, por ejemplo, interactuar con los elementos presentes en la mesa, siendo ésta la zona de interacción más cercana, y usar la pared para interactuar con aquellos elementos más distantes a modo que sean fácilmente visibles. De todos modos, cabe mencionar que el alcance de este proyecto abarca únicamente la superficie horizontal (mesa) como superficie de interacción.

1.1.2. Interfaz gráfica e información visual: proyector

Debido a que ninguna de las superficies de interacción es un dispositivo capacitado para la visualización de contenido por sí sola, se utilizan proyectores para presentar la información visual al usuario en base a la información procesada como resultado de las acciones que éste realiza. En principio, la cantidad de proyectores requeridos es igual a la cantidad de superficies de interacción diferentes, por lo que, dado el alcance establecido para el proyecto, se hace uso de un único proyector utilizado para desplegar información sobre la superficie que oficia de mesa.

1.1.3. Sensores de interacción: *Microsoft Kinect* y *Leap Motion*

En lo que refiere específicamente a la forma de interacción de los usuarios con una aplicación que utilice el *framework*, como ya se ha mencionado, es posible que sea de forma multitáctil o tridimensional. Con este fin, son requeridos los sensores *Microsoft Kinect* y *Leap Motion*. En el *Capítulo 2* se presenta en detalle cada uno de estos sensores, incluyendo sus características físicas, su funcionamiento y qué provee cada uno de ellos en beneficio de la solución.

El sensor *Microsoft Kinect* ofrece una visión más amplia de la escena en comparación de lo que puede observar el sensor *Leap Motion*. Por ello, se lo ubica de forma que abarque toda la escena, o al menos la mayor cantidad de área posible. Mediante la información provista por este sensor se resuelve la detección de aquellos gestos de usuario a nivel de brazo y mano, es decir, aquellos que puedan ser reconocidos únicamente con información de las posiciones de los brazos y manos de los usuarios, no utilizándose para detectar gestos más precisos como los realizados con los dedos. Adicionalmente, este sensor también permite resolver la interacción multitáctil con la superficie de interacción.

Por otra parte, resolver el seguimiento esqueletal de la mano, también conocido como *hand skeletal tracking*[2] o *finger tracking* [125], de forma sencilla y con buenos resultados es una característica que no provee el sensor *Microsoft Kinect* para la versión utilizada. Con el fin de obtener este nivel de detalle de interacción se introduce el sensor *Leap Motion*, quien resuelve esta problemática de forma nativa. Cada sensor *Leap Motion* posibilita tanto el seguimiento esqueletal de las manos, como la detección de gestos tridimensionales realizados con ellas.

1.1.4. Dispositivos de usuario

Los dispositivos de usuario pueden ser computadoras personales, teléfonos celulares, agendas de notas, entre otros. Esto es, cualquier tipo de objeto con el que el usuario pueda contar y sea apoyado sobre

[2] *Skeletal tracking* refiere a algoritmos de reconocimiento que permiten el seguimiento del esqueleto, ya sea del esqueleto corporal completo como de ciertas partes del cuerpo que presentan un esqueleto en sí mismas. Un ejemplo de esto último son las manos, cuyo esqueleto está conformado por los huesos de los dedos, muñeca y en algunos casos parte del brazo.

una de las superficies de interacción permitiendo al usuario interactuar de forma completamente natural como si fueran un elemento más en la escena. Esta característica quedó finalmente fuera del alcance del proyecto debido a que requiere reconocer una gran variedad de objetos de distintas dimensiones y formas para posibilitar la interacción con ellos, lo cual cae dentro de un área que escapa de los objetivos principales del proyecto.

1.1.5. Procesamiento de información: servidor

El servidor es el computador encargado de ejecutar el componente que realizará el procesamiento de la información recolectada por los diferentes sensores de interacción que conforman la solución, incluyendo el procesamiento de los gestos que está realizando el usuario, los movimientos, el seguimiento esqueletal, y en general, cualquier cambio o evento que ocurra en la escena. De esta forma, el usuario percibirá que la inteligencia está efectivamente en los objetos con los que interactúa. Este último es uno de los pilares fundamentales en los que se basa la computación ubicua, presentada con más detalle en el *Capítulo 3*.

1.2. Objetivos

Se enumeran a continuación los objetivos a alcanzar para el presente proyecto:

1. Relevar el estado del arte en lo que respecta a las técnicas de interacción de tipo multitáctil así como también a las técnicas de interacción natural espacial, en particular, la interacción basada en gestos.
2. Desarrollar un *framework* que posibilite la construcción de aplicaciones que hagan uso de estas dos vías de interacción, ya sea de forma combinada o individual.
3. Realizar un prototipo de aplicación que utilice los servicios brindados por el *framework* generado para incorporar las características de interacción natural espacial y/o multitáctil a las posibilidades de interacción de los usuarios.
4. Elaborar un informe en el que se detallen todos los aspectos referentes a los puntos anteriores, incluyendo detalles técnicos de la solución, decisiones tomadas durante el transcurso del trabajo, etc.

1.3. Motivación

La interacción multitáctil hoy en día es una vía de interacción muy establecida, permitiendo interactuar con los dispositivos mediante manipulación directa, generalmente gracias a la inclusión de pantallas capacitivas [149]. Por otro lado, si bien la interacción natural gestual está establecida como forma de interacción [32], hoy en día está principalmente abocada a medios particulares como los videojuegos u otros medios ligados al entretenimiento [131], no habiéndose establecido aún como forma natural de interacción en aspectos cotidianos [32]. En este sentido, actualmente ambas vías de interacción conviven de forma aislada [49] en lo que respecta a los sistemas y/o dispositivos utilizados por la gran mayoría de los usuarios finales. Sin embargo, la integración de estos tipos de interacción es un área ampliamente estudiada, por lo que con seguridad ambos tipos de interacción convivan de forma más dependiente una de otra en el futuro cercano. Por esta razón, la principal motivación de este trabajo es lograr un *framework* que tome las ventajas de cada una de estas formas de interacción y las brinde al usuario de forma combinada como principal vía de interacción con sus aplicativos.

Otra de las motivaciones del proyecto es la posibilidad de integrar dos sensores de interacción diferentes como son el *Leap Motion* y el *Microsoft Kinect.* Si bien al día de hoy existe una gran cantidad de proyectos de investigación y soluciones ya establecidas abocadas al uso de cada uno de estos sensores de forma individual, no abundan de la misma forma proyectos que intenten integrar el uso de ambos y las

características que cada uno provee en una misma solución [49].

Finalmente, se busca lograr una solución de bajo costo, ya que actualmente la mayoría de las soluciones similares existentes incluyen superficies capacitivas que cuentan con inteligencia en sí mismas y tienen un costo muy alto, haciendo que el costo global de la solución aumente considerablemente. Adicionalmente, el hecho de no utilizar una superficie capacitiva como parte de la solución hace que sea una solución más genérica y flexible, permitiendo que cualquier superficie pueda ser, de cierta forma, dotada de capacidad de cómputo.

1.4. Problemas a resolver

Para lograr los objetivos del proyecto se debieron resolver una gran variedad de problemas, los cuales son presentados en detalle en el *Capítulo 4*. Algunos de los más relevantes son los siguientes:

1. Integración de múltiples sensores: como parte de la solución es necesario integrar y operar de forma conjunta tanto sensores *Microsoft Kinect* como sensores *Leap Motion*, obteniendo datos desde ambos y procesándolos para unificar la información recibida.
2. Múltiples sistemas de coordenadas: para que los datos de los distintos sensores puedan coexistir en una solución integrada, deben ser llevados a un sistema global de coordenadas. Por esto, es necesario resolver la unificación de los distintos sistemas de coordenadas que entran en juego en la solución en un único sistema de coordenadas global. A su vez, al ser naturalmente un sistema donde la información que se brinda al usuario es proyectada sobre una superficie, se debe resolver la proyección de esta información según la aplicación que esté haciendo uso del *framework* y las superficies que ésta utilice, siendo necesario también un sistema de coordenadas de proyección. Este problema se hace aún más complejo dado que no solo se debe mostrar información sobre la superficie sino que además el usuario interactúa de forma tangible sobre ésta, debiendo calibrar correctamente el mapeo de información desde el espacio tridimensional de interacción al espacio bidimensional de proyección o visualización de información. Finalmente, cada aplicación desarrollada sobre el *framework* contará con su propio sistema de coordenadas, por lo que es necesario transformar toda la información recopilada por los diferentes componentes del sistema para poder visualizarse correctamente en la aplicación.
3. Seguimiento esqueletal (*body tracking*): se debe determinar las posiciones de los usuarios y realizar en todo momento su seguimiento en la escena, así como también determinar qué gestos realiza cada uno de ellos.
4. Interacción multitáctil: se debe determinar las superficies de interacción y cuándo se efectúan eventos y/o gestos de tipo multitáctil. Al no contar con una superficie capacitiva, se deben utilizar algoritmos de visión por computadora con el objetivo de analizar la información provista por los diferentes sensores y reconocer este tipo de interacciones.
5. Reconocimiento de gestos: aunque el aprendizaje por computador no es un área directamente relacionada con los objetivos del proyecto, es deseable que se puedan reconocer diferentes tipos de gestos multitáctiles y/o tridimensionales. Para esto es necesario el reconocimiento de patrones posibilitado por este tipo de algoritmos de aprendizaje por computador.
6. Interferencia entre sensores: al incluir varios sensores que utilizan algoritmos basados en la emisión de rayos infrarrojos para obtener la información de los objetos en la escena, es natural que exista cierta interferencia entre ellos. Se debe solucionar este problema intentando evitar la interferencia entre los sensores con el fin de minimizar la pérdida de información.
7. Arquitectura distribuida y asincrónica: dadas las restricciones de los diferentes sensores a utilizar y con el objetivo de obtener una mejor performance, éstos deben ser gestionados y utilizados de forma distribuida. Para ello, se deben utilizar protocolos de comunicación adecuados y contar con vías de serialización de los datos obtenidos por los sensores para ser enviados de un nodo a otro.
8. Arquitectura extensible: la arquitectura de la solución resultante debe ser lo suficientemente

genérica y extensible como para que permita agregar nuevos módulos y características de forma sencilla. Se debe poder agregar de forma sencilla nuevos tipos de sensores de interacción, otros posibles gestos a reconocer, entre otros.

9. Definición de un *framework:* se debe desarrollar un *framework* sobre el cual se puedan crear aplicaciones haciendo uso de técnicas multitáctiles y de interacción natural gestual de forma combinada. Dicho *framework* debe exponer métodos y brindar servicios para la detección de eventos de ambos tipos.
10. Construcción del espacio de trabajo: para dar soporte a la solución es necesario un espacio de trabajo que permita utilizar satisfactoriamente el *framework*, teniendo en cuenta sus restricciones de espacio y *hardware*. El entorno construido debe contar al menos con una superficie de interacción y los distintos sensores de interacción, dispuestos de forma tal que posibiliten el correcto funcionamiento de la solución.
11. Desarrollo de aplicaciones: se debe desarrollar una aplicación que haga uso del *framework,* construído como forma de ilustrar todos los servicios y características que éste provee.

1.5. Organización del documento

En lo que respecta a la organización del resto del documento, se continúa con el *Capítulo 2*, donde se describen los componentes más relevantes vinculados tanto al *hardware* como al *software* que brindan soporte a la solución generada. Luego, en el *Capítulo 3* se detalla el estudio del estado del arte para el presente trabajo, así como también muchos de los conceptos y áreas de interés que abarca. En el *Capítulo 4* se detalla cada uno de los problemas a resolver y cómo éstos fueron resueltos para obtener la solución final. Posteriormente, en el *Capítulo 5* se plantea la solución desde el punto de vista del diseño y la arquitectura, describiendo las restricciones que ésta impone, así como también los detalles técnicos de la aplicación desarrollada sobre el *framework* construído. Luego, en el *Capítulo 6* se presenta la aplicación construida haciendo uso de los servicios provistos por el *framework*. Finalmente, en el *Capítulo 7*, se presentan las conclusiones del trabajo realizado, así como también las posibles mejoras que podrían realizarse.

Adicionalmente a los capítulos detallados, se incluye una sección *Glosario* en la cual se listan las definiciones de los conceptos más relevantes mencionados en el documento, una sección de *Referencias bibliográficas* en la que se listan todas las referencias bibliográficas del trabajo, una sección *Referencia de vídeos* donde se incluyen los vídeos generados para ilustrar las diferentes problemáticas desarrolladas, una sección *Índice de figuras* donde se incluyen las referencias a todas las figuras incluidas a lo largo del documento y una sección *Repositorio de código* donde se incluye el enlace público al repositorio con el código generado y se detalla su estructura. Por último, se incluye un conjunto de *Anexos* en los que se presenta información adicional a lo descrito en cada uno de los capítulos. Como parte de los anexos también se puede encontrar el manual de referencia del *framework* construido, incluyendo la guía de programador y la descripción de los servicios provistos, así como también los detalles de configuración del entorno de desarrollo. Adicionalmente a la documentación elaborada se adjunta una ficha en formato PDF con el diseño de arquitectura completo.

Capítulo 2: Soporte a la solución

En este capítulo se detallan los componentes más relevantes que dan soporte a la solución propuesta en lo que respecta a *hardware* y *software*. Para cada uno de los componentes se detalla el porqué de su utilización como parte de la solución, cuáles son sus características más relevantes y cómo es su funcionamiento.

2.1. *Hardware*

Como ya se mencionó en el capítulo previo, la solución propuesta para este trabajo se basa principalmente en el uso de dos sensores que permiten obtener información de lo que está ocurriendo en la escena en la cual se encuentran inmersos. Más específicamente, información relacionada a detección y el seguimiento de las acciones realizadas por los usuarios presentes en la escena. En esta sección se detallan las características más notorias de cada uno de estos sensores, describiendo en detalle cómo es su funcionamiento, los componentes que lo integran y qué características brindan al programador para construir aplicaciones.

2.1.1. Sensor *Microsoft Kinect*

Uno de los dos sensores centrales utilizados en la solución es el *Microsoft Kinect*[3], el cual permite reconocer los movimientos corporales realizados por un usuario dentro de una escena. Estos reconocimientos pueden ser usados a modo de comandos de forma tal que se pueda prescindir de la necesidad de contar con controles externos. Este pequeño sensor se hizo público originalmente en el año 2010 como accesorio adicional para las consolas de videojuegos *Xbox 360* de *Microsoft* y generó una revolución en el concepto del juego, ya que sin la necesidad de controles, siendo el propio cuerpo el encargado de enviar los comandos a lo largo del juego, el jugador se envuelve más en el rol brindando mejores experiencias de entretenimiento [81].

2.1.1.1. Descripción

El sensor *Microsoft Kinect* está compuesto por varias piezas a nivel de *hardware*: un sensor de profundidad, una cámara *RGB*, un motor de inclinación, un conjunto de micrófonos y un chip de control.

Figura 2: Componentes de *Microsoft Kinect:* 1- Sensor de profundidad, 2 - Cámara *RGB,* 3 - Motor de inclinación 4- Conjunto de micrófonos [83]

El sensor de profundidad está compuesto por un par cámara-proyector, donde el primer elemento del par es una cámara infrarroja con una resolución 640x480 píxeles y una velocidad/frecuencia de 30 *FPS*, mientras que el segundo elemento es un proyector de luz infrarroja. La cámara *RGB* es de tipo VGA con una

[3] El nombre *Kinect* viene inspirado por las palabras *kinetic* o cinético, que está relacionado al hecho de estar en movimiento y *connect* o conectado, por lo que relaciona los movimientos del usuario y de qué manera los conecta

resolución de 640x480 píxeles y una velocidad/frecuencia de 30 *FPS*. El conjunto de micrófonos está compuesto por cuatro micrófonos, donde tres de ellos están posicionados del lado derecho mientras que el restante está posicionado del lado izquierdo del sensor [83]. Aunque no fueron utilizados como parte de la solución, los micrófonos permiten captar conversaciones y comandos de voz emitidos por el usuario. El motor con el que cuenta el sensor es un motor de inclinación a un máximo de 30 grados que se ubica en su base, teniendo como objetivo brindar movilidad para posibilitar así el seguimiento de los usuarios.

Por otra parte, el sensor cuenta con el chip PS1080 creado por la compañía *PrimeSense*, el cual es el cerebro de *Microsoft Kinect* [82]. Este chip se encarga de todos los controles del sistema, como la proyección de luz infrarroja y la realización del procesamiento de la información recogida por cada cámara y cada sensor de audio. A su vez, usando el proyector de rayos infrarrojo también se soluciona el problema de la interferencia por luz ambiental ya que el sensor no está diseñado para registrar la luz visible. Esto hace que se minimice la cantidad de falsos positivos o lecturas erróneas, permitiendo reconocer objetos independientemente de la intensidad de la luz ambiente que existe en la escena. En la *Figura 3* se puede apreciar un diagrama que ilustra la interacción del procesador con los diferentes componentes y las diferentes interconexiones existentes. Mediante los componentes descritos, el chip PS1080 construye dos mapas, un mapa de profundidad y un mapa de color, que permiten representar la escena en la cual se encuentra ubicado el sensor.

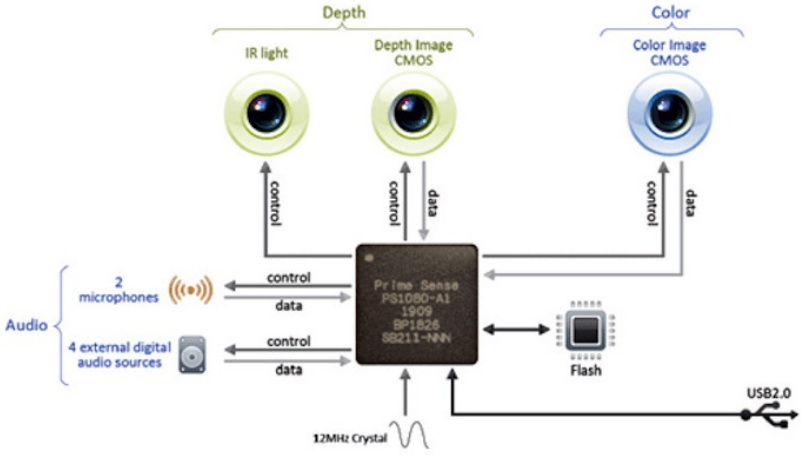


Figura 3: Chip *PS1080* de *PrimeSense* [82]

Para generar el mapa de profundidad, el proyector de luz infrarroja de *Microsoft Kinect* emite un patrón conocido de rayos infrarrojos invisibles. Este patrón es percibido por la cámara infrarroja tras rebotar contra los distintos objetos presentes en la escena. A causa de este rebote, el patrón sufre modificaciones y dado que tanto el patrón de rayos infrarrojos como la calibración de la cámara y el proyector son conocidos, se puede asociar cada punto emitido con el recibido y calcular las distorsiones existentes. A partir de estas distorsiones (en tamaño y ángulo) percibidas, el chip PS1080 calcula la distancia de los objetos y genera la imagen de profundidad de la escena. Esta técnica, en la cual se estudia la deformación que sufre un patrón de luz al ser intersecado por cualquier objeto, es denominada "luz estructurada" [21]. Por otro lado, la forma en que el sensor *Microsoft Kinect* genera el mapa de color es mediante el uso de su cámara *RGB*. De este modo entonces, mediante la construcción de estos dos mapas, quien procese esta información en el computador al cual se conecta el sensor puede reconocer diferentes elementos en la escena, así como también los cambios que en ella ocurren. Ambos mapas son semejantes a los que se pueden ver en la *Figura 5,* y una vez obtenidos son transferidos mediante un puerto USB 2.0 al computador.

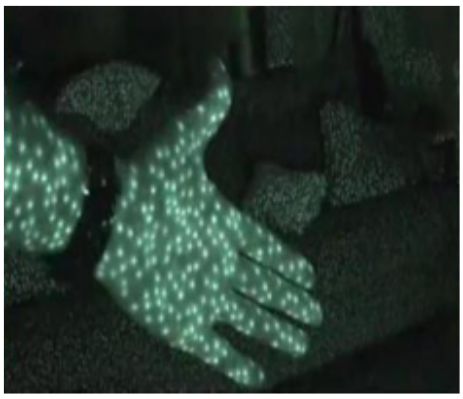


Figura 4: Patrón de rayos infrarrojos emitido por el sensor *Microsoft Kinect* [21]

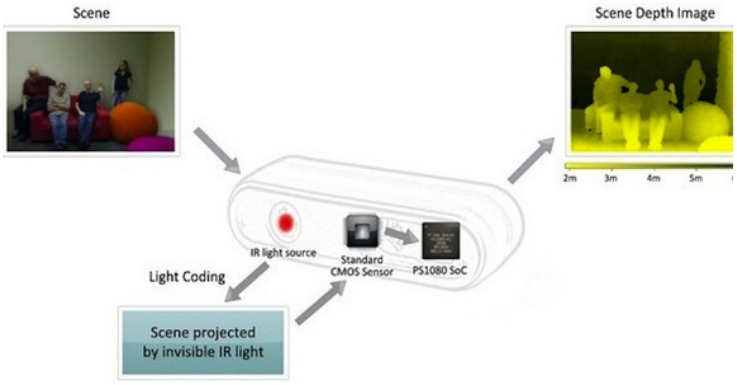


Figura 5: Mapas de color y profundidad de *Microsoft Kinect* [83]

2.1.1.2. Utilidad en la solución propuesta

Como ya se se mencionó, el sensor *Microsoft Kinect* comenzó utilizándose como un accesorio más para poder interactuar con la consola de videojuegos *Xbox360*. Hoy en día su utilización ha llegado mucho más lejos y muchas aplicaciones desarrolladas en todo el mundo están siendo innovadoras con el uso de este sensor [145]. Varias áreas que no sólo involucran el entretenimiento están utilizándolo de forma intensiva, creándose así aplicaciones que van desde lo que concierne a ayudar a niños con autismo, planteando nuevas formas de comunicación, hasta herramientas de apoyo para cirujanos en quirófanos con el fin de controlar robots de cirugía con gran precisión [144].

Microsoft Kinect es un sensor central para este proyecto dado que brinda soporte para resolver dos problemáticas bien distintas. Por un lado, provee la información necesaria para llevar a cabo la detección de interacción multitáctil sobre superficies que no poseen capacidad de cómputo por sí mismas. Por otro lado, provee mecanismos para la detección y el reconocimiento de usuarios, así como también el reconocimiento de ciertos gestos que realicen.

2.1.2. *Leap Motion*

El segundo sensor utilizado como soporte a la solución propuesta es el denominado *Leap Motion*, el cual permite realizar un seguimiento más preciso de las manos y dedos de los usuarios. Este sensor se hizo público a fines del año 2012 con el lanzamiento de unidades para uso específico por desarrolladores, las cuales, como se detalla más adelantes, poseen algunas restricciones con respecto a la posterior versión comercial. Dado el éxito y las repercusiones positivas que causó gracias a las nuevas posibilidades de interacción que brinda, la primer versión comercial dirigida al público general vio la luz a mediados del año 2013 [45].

2.1.2.1. Descripción

Leap Motion es un pequeño sensor periférico que está diseñado para ser colocado sobre una superficie plana, apoyado de forma tal que la parte donde se ubican los LEDs quede mirando hacia arriba. De todos modos, cabe mencionar que hoy en día está enfocado y optimizado para su uso en dispositivos de tipo *Head Mounted Display (HMD)*, como sensor de interacción para realidad virtual [60]. El sensor está básicamente orientado a la detección de manos y dedos de forma muy precisa, así como también a la detección de gestos realizados con éstos. La precisión que logra tener *Leap Motion* es de aproximadamente 0.01 milímetros y cuenta con un campo de visión (del inglés *field of view* o más comúnmente *FOV*) de 115 grados en forma de pirámide invertida con punto de apoyo en el centro del sensor.

"El objetivo es fundamentalmente transformar la manera en que las personas interactúan con las computadoras de la misma forma que el ratón lo hizo, lo que significa que la transformación afecta a todos, tanto desde el caso más básico de uso hasta los casos de uso más avanzados que se puedan imaginar para la tecnología informática"

-- Michael Buckwald, CEO de Leap Motion

Como se ilustra en la *Figura 6*, el sensor está conformado por dos cámaras y tres LEDs infrarrojos, quienes permiten realizar el seguimiento de la luz infrarroja con una longitud de onda de 850 nanómetros, valor que está por fuera del espectro visible al ojo humano. Dadas las características y ángulos de los *LEDs*, se tiene un espacio de interacción que, como se mencionó anteriormente, tiene la forma de una pirámide invertida desde el centro del sensor y que abarca una altura de aproximadamente 60 centímetros, valor limitado por las características de propagación de la luz *LED* [41].

Figura 6: Componentes *hardware* del sensor *Leap Motion* [41]

El sensor lee la información obtenida en su memoria local, realiza ciertos ajustes y luego la envía por USB al *software* de seguimiento de *Leap Motion* que procesa y analiza la información. Una vez que la información arriba al *software*, se realiza una serie de procesamientos matemáticos sobre los datos en crudo. Estos procesos comienzan con el filtrado de los objetos del fondo de la escena, como puede ser por ejemplo la cabeza del usuario, así como también la reducción de ruido por la iluminación del entorno, para luego continuar con el análisis de las imágenes y reconstruir una representación tridimensional de la escena. Posteriormente, se continúa con el procesamiento necesario para interpretar y extraer información sobre la posición de los dedos y manos, e inferir las posiciones de los objetos ocluidos (característica presente a partir del nuevo *SDK* v2 como se detalla en la *Sección 2.2.2*). Finalmente, los resultados son enviados como una serie de cuadros que pueden ser luego obtenidos por las aplicaciones mediante invocaciones provistas por el *SDK* del sensor [65].

2.1.2.4. Utilidad en la solución propuesta

En primer lugar, se debe mencionar que el espacio de observación que abarca el sensor *Leap Motion* es más pequeño, pero a la vez, de mucho mayor resolución que el provisto por el sensor *Microsoft Kinect*. Teniendo en cuenta esta diferencia, surge la idea de contar con una solución que tome las ventajas de cada uno de estos sensores y los integre de forma inteligente y lo más transparente posible para el usuario. De esta forma, el sensor *Microsoft Kinect* proporciona un espectro amplio de reconocimiento, haciendo posible el seguimiento de usuarios en base a un conjunto de puntos de interés de su esqueleto, sin incluir ningún tipo de reconocimiento a nivel de dedos. Complementariamente, *Leap Motion* proporciona lo que el primer sensor no brinda, es decir, el reconocimiento de los dedos de las manos y con ello una mayor precisión para los gestos efectuados con ellas. En otras palabras, lo que se propone para este proyecto es contar por un lado con sensores *Leap Motion* para lograr precisión con los gestos de usuario realizados con las manos y

dedos y por otro lado incluir sensores *Microsoft Kinect* para los demás componentes de interacción, como la interacción multitáctil, el reconocimiento y seguimiento de usuarios y aquellos gestos que no requieran demasiada precisión.

2.2. *Software*

Además de los sensores *hardware* que se presentaron en la subsección anterior, se utilizó un conjunto de bibliotecas de *software* para manejar la información que proporcionan estos sensores, así como también para dar soporte a las distintas problemáticas planteadas en el proyecto.

2.2.1. OpenNI-NiTE

En un principio se descartó el uso del *framework* nativo provisto por el sensor *Microsoft Kinect SDK* para la obtención de información ya que éste no permite portabilidad entre distintas plataformas. Buscando alternativas a esta situación se analizaron los *frameworks OpenNI* y *NiTE*, descritos en esta sección.

2.2.1.1. Descripción

OpenNI facilita la comunicación con sensores de audio, vídeo y profundidad de los dispositivos *hardware* compatibles como los sensores *Microsoft Kinect* o *ASUS Xtion*[4] [159], así como también la comunicación con módulos intermedios de percepción encargados de analizar y comprender los datos obtenidos de la escena mediante algoritmos de visión por computadora. Un módulo intermedio de percepción, también conocido como *middleware*, es un *software* que recibe datos visuales como una imagen, y retorna, por ejemplo, la posición de la palma de la mano detectada en dicha imagen, o bien identifica ciertos gestos predefinidos y genera eventos cuando éstos son detectados. *NiTE*, cuya sigla en inglés significa *Natural Interaction Technology for End user*, es el módulo intermedio de percepción más comúnmente utilizado en conjunto con *OpenNI*, alimentándose directamente de los datos proporcionados por este último. *NiTE* es la implementación de *PrimeSense* del motor de percepción, el cual incluye características de identificación de usuarios, seguimiento esqueletal y seguimiento de manos como se ilustra en la *Figura 7*, reconocimiento de varios gestos predefinidos, determinación de cuál de los usuarios tiene el control en cada momento, entre otros [95]. Un punto a destacar es que el *framework* provee únicamente las interfaces, delegando la lógica que implementa cada funcionalidad a los diferentes componentes, sean estos sensores *hardware* o módulos intermedios de percepción. Así, por ejemplo, *OpenNI* provee operaciones para obtener el mapa de profundidad de la escena en donde cada punto representa la distancia desde el sensor o los datos crudos de imagen *RGB* obtenidos desde las cámaras, pero la lógica de estas operaciones estará implementada en el controlador del sensor [108].

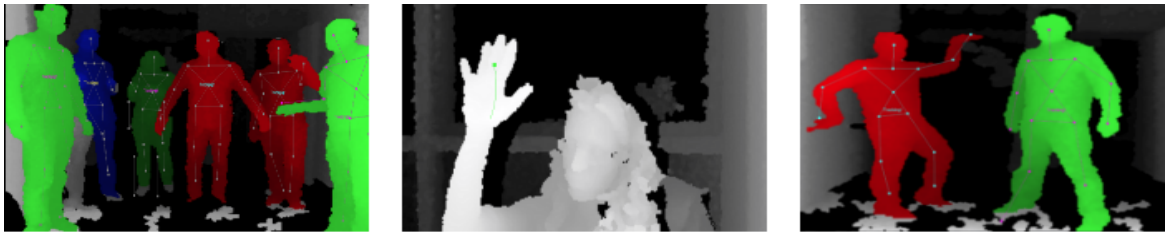


Figura 7: Seguimiento esqueletal y seguimiento de manos con *NiTE* [94]

La organización por detrás del desarrollo de estos *frameworks* fue inicialmente *PrimeSense*, una fundación conformada en noviembre del 2010 cuyo objetivo es promover y estandarizar la compatibilidad e interoperabilidad de la interacción natural mediante el uso de dispositivos y módulos intermedios

[4] Sensor de profundidad similar a Microsoft Kinect, también basado en el chip *PS1080* de *PrimeSense*

especializados. En este sentido, cabe mencionar que cualquier proveedor compatible con *OpenNI* puede registrar su propia implementación de las características soportadas por el *framework*.

OpenNI permite entonces interactuar básicamente con dos tipos de componentes, siempre y cuando sean compatibles con el *framework*. Por un lado, componentes *hardware* como por ejemplo sensores de profundidad y cámaras *RGB* que proveen datos crudos obtenidos a partir de la escena actual. Por otro lado, módulos intermedios como el ya mencionado *NiTE*, que son en definitiva el cerebro del sistema, encargándose de procesar y comprender los datos crudos para brindar información de mayor nivel de abstracción. En pocas palabras, los sensores observan el entorno y el módulo intermedio comprende la interacción del usuario con dicho entorno.

Es preciso mencionar que *OpenNI* y *NiTE* cambiaron la forma de exponer sus funcionalidades a fines del año 2012, como parte de la transición de la versión 1.5 a la versión 2.0. Este cambio de versión impactó considerablemente en el desarrollo de este proyecto, ya que, en su momento, parte del desarrollo se había implementado con la versión 1.5. Se debieron realizar las modificaciones necesarias para tener en cuenta esta nueva versión debido a que no mantuvo compatibilidad hacia atrás con las versiones anteriores. El cambio más sustancial que sufrió la arquitectura es que los componentes de los módulos intermedios como *NiTE* pasaron a estar por fuera de *OpenNI* en la estructura de capas [107]. En la versión 1.5 dichos componentes se debían registrar en *OpenNI*, haciendo que sus métodos queden abstraídos por su interfaz. En cambio, en la versión 2.0 los módulos intermedios pasan a estar en una capa por encima de *OpenNI*, quien le proporciona la información necesaria, y el propio módulo intermedio se encarga de comunicarse con las aplicaciones de usuario. Por esta razón, en esta nueva versión las aplicaciones de usuario se comunican directamente no sólo con *OpenNI* sino también con el módulo intermedio a utilizar como motor de percepción; en este caso, *NiTE*. En la *Figura 8* se puede observar la transformación que sufrió la arquitectura en la transición de la versión 1.5 a la versión 2.0, y cómo están dispuestas las diferentes capas en ambas versiones.

De todas formas, la nueva versión trae consigo varias ventajas como por ejemplo la simplificación del *framework* provisto, reduciendo notoriamente la curva de aprendizaje y brindando interfaces de comunicación más sencillas con los diferentes sensores. Mientras la versión anterior contaba con alrededor de cien clases y más de cien estructuras de datos y enumerados, la nueva versión consta solamente de doce clases y doce estructuras de datos y enumerados. Adicionalmente, al igual que cualquier otro módulo intermedio, *NiTE* queda desligado como un *framework* independiente que puede interactuar directamente con las aplicaciones de usuario [107].

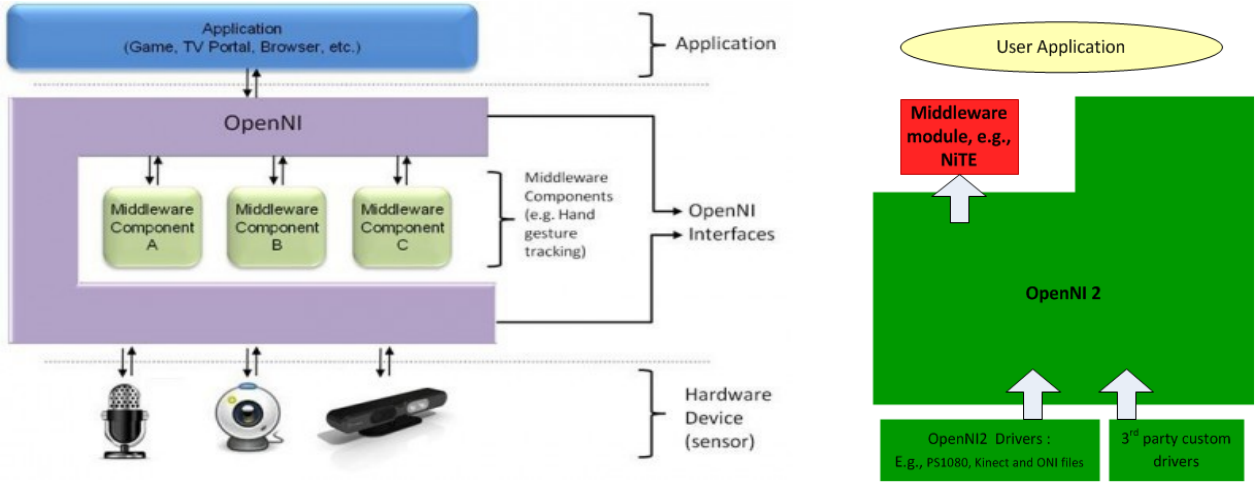


Figura 8: Arquitectura de *OpenNI 1.5* (izq.) [38] y *OpenNI 2.0* (der.) [107]

2.2.1.2. Sistema de coordenadas

El sensor *Microsoft Kinect*, a través de *OpenNI*, maneja un sistema de coordenadas cartesiano centrado en la cámara infrarroja, y reporta sus valores en milímetros. Este sistema de coordenadas se puede observar en la *Figura 9*, donde los ejes correspondientes a X y Z conforman un plano horizontal siendo el eje X paralelo al lado más largo del sensor, mientras que el Z es perpendicular a X. Los valores positivos para X son aquellos que se encuentran hacia la derecha del sensor, mientras que los valores positivos para Z son aquellos salientes al mismo. Por otro lado, el eje Y es perpendicular al sensor, considerando los valores positivos como aquellos por encima de él.

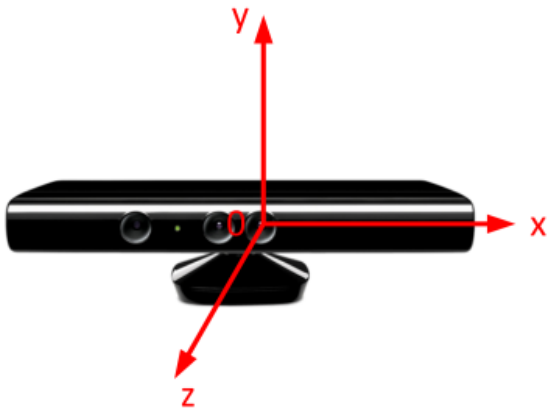


Figura 9: Sistema de coordenadas del sensor *Microsoft Kinect* [4]

2.2.1.3. *Seguimiento*

Toda la funcionalidad de alto nivel como la detección de usuario, el seguimiento de manos, el seguimiento esqueletal y la detección de gestos fueron removidas de *OpenNI* en su versión 2.0, requiriendo que estas funcionalidades sean implementadas y ofrecidas directamente por el módulo intermedio en uso. Por esta razón, *NiTE* es quien brinda los mecanismos para obtener dicha información de alto nivel. Proporciona un seguimiento esqueletal del cuerpo completo brindado información de un conjunto de puntos relevantes, que representan la cabeza, el cuello, los hombros, los codos, las manos, las rodillas, los pies, el centro del cuerpo y la cadera del usuario, como se puede observar en la *Figura 10*. Además de proporcionar un seguimiento a nivel general, *NiTE* también provee seguimiento específico a nivel de las manos del usuario. En este caso, el conjunto de puntos relevantes se reduce a los puntos correspondientes a los centros de las manos, obteniendo de este modo su posición en todo momento a medida que se mueven.

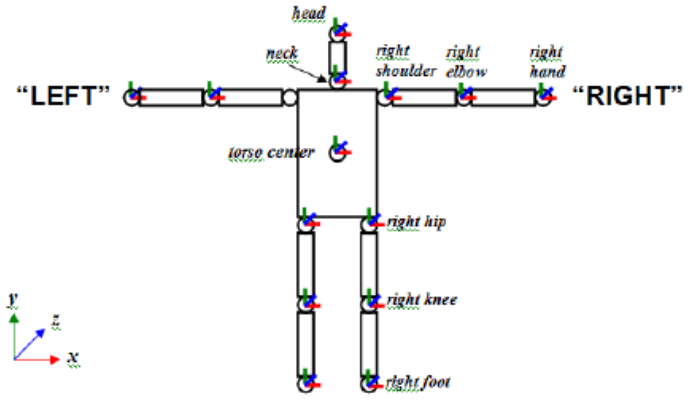
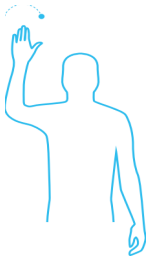


Figura 10: Puntos de interés de usuario [95]

Adicionalmente, *NiTE* reconoce un conjunto predefinido de gestos y poses realizadas por el usuario mediante movimientos de su cuerpo. Algunos de los gestos y poses reconocidas nativamente por *NiTE* se detallan en la *Figura 11*.



A. Levantar mano (del inglés *Raise hand*): movimiento consistente en levantar cualquiera de los brazos.

B. Clic (del inglés *Click*): movimiento consistente en mover la mano hacia adelante y hacia atrás como si se presionara un botón.

C. Saludar (del inglés *Wave*): movimiento consiste en mover la mano hacia los costados como si se estuviera saludando.

D. Pose Psi (del inglés *Psi Pose*): pose consistente en levantar los dos brazos con las palmas hacia el frente.

Figura 11: Algunos de los gestos predefinidos provistos por *NiTE* [94]

2.2.1.4. Motivo de uso

Por un lado, a diferencia del *framework* distribuido por *Microsoft*, tanto *OpenNI* como *NiTE* son multiplataforma, pudiendo ser utilizados en plataformas *Windows, Linux Ubuntu* o *Mac OS X*. A su vez, *OpenNI* se distribuye bajo licencia *Apache Licence* [3], lo cual permite disponer del código y distribuirlo de forma gratuita. *NiTE* es distribuido bajo licencia *NiTE License Terms*, lo que implica que puede ser utilizado sólo en conjunto con *hardware* autorizado por *PrimeSense*[5], pero su código no está disponible públicamente.

Un hecho crucial ocurrido fue que *PrimeSense* fue comprada por *Apple* en noviembre del 2013 [5], advirtiendo que los sitios oficiales de *OpenNI*, *NiTE* y *PrimeSense* serían dados de baja para abril del 2014 y pasarían a dejar de estar disponibles públicamente. De todas formas, gracias al licenciamiento de *OpenNI*, múltiples organizaciones que lo utilizan intensivamente como parte de sus productos preservaron la documentación y el código fuente. Dichas organizaciones han brindado repositorios de acceso público y han creado páginas específicas para el *framework* [100, 103] desde donde se pueden descargar las versiones de *OpenNI* para las distintas plataformas y la documentación correspondiente. Por otra parte, con *NiTE* sucede algo distinto, ya que, en este caso, previo a la compra de *PrimeSense* por parte de *Apple* el *framework* era distribuido mediante una licencia de uso privado, pero que lo hacía flexible para su utilización en conjunto a dispositivos autorizados. Esto implica que actualmente *NiTE* no esté disponible, habiendo desaparecido por completo del ámbito público. De toda formas, dado el grado de avance y desarrollo de la solución propuesta en el momento en que se dieron estos cambios, se decidió seguir utilizando la versión con la que se contaba de este *framework*, a pesar de que ya no cuenta con ningún tipo de soporte por parte de la compañía que le dio origen. Sin dudas esta situación fue un golpe duro para el proyecto, pero gracias a la continuidad que le ha dado la comunidad a *OpenNI* y a haber contado con una versión de *NiTE* previo a su disolución, se logró mantener gran parte del trabajo ya realizado.

[5] Aquellos sensores que cuentan con el procesador PS1080 fabricado por la compañía, entre los cuales se encuentra el sensor *Microsoft Kinect*

2.2.2. *Leap Motion SDK*

2.2.2.1. Descripción

Al igual que el sensor *Microsoft Kinect*, el sensor *Leap Motion* cuenta con un *framework* nativo denominado *Leap Motion SDK* [63]. Sin embargo, dicho *framework* no es tan restrictivo como el que posee nativamente el sensor *Microsoft Kinect*, ya que está disponible para una gran variedad de plataformas incluyendo *Windows*, *Mac*, y *Linux*, y un gran abanico de lenguajes de programación, como *C/C++*, *Java*, *C#* y *Python*. Se ha utilizado el *Leap Motion SDK* para la comunicación con el sensor *Leap Motion* ya que proporciona todos los mecanismos necesarios para acceder a los datos provistos por el sensor, así como también a un conjunto predefinido de gestos. Además, su desarrollo se expande continuamente mediante la liberación de nuevas versiones de forma ágil, lo cual implica que va en continuo crecimiento en cuanto a la funcionalidad brindada y estabilidad. Otro punto a considerar es que el sensor *Leap Motion* cuenta actualmente con una comunidad muy activa, debido a que múltiples desarrolladores se han mostrado interesados en desarrollar aplicaciones que saquen provecho del sensor [88].

En Agosto del 2014 se liberó la versión 2.1 del *Leap Motion SDK*, trayendo consigo un conjunto de cambios sustanciales en la forma de seguimiento además de otras nuevas características que las versiones anteriores no poseían y cuyo valor fue significativo para este proyecto. Por esta razón, luego de realizar múltiples pruebas para verificar su estabilidad y comprobar que el *framework* respondiera de manera correcta y robusta se decidió actualizar a la nueva versión. A su vez, mantuvo la compatibilidad con las versiones anteriores, por lo que todo el trabajo ya desarrollado con la versión anterior no tuvo que ser descartado ni debió sufrir alteraciones que implicaran un tiempo considerable de retrabajo.

En lo que respecta a las mejoras provistas por la nueva versión, una de las más relevantes es el uso de un nuevo modelo de seguimiento que brinda más información en lo que respecta a las manos y los dedos. La mano humana se modela de forma más real [71], permitiendo estimar las posiciones de todos los dedos y manos mediante la predicción de las posiciones para aquellos dedos y manos que no son totalmente visibles por el sensor. De esta forma, por cada mano detectada siempre se reconocerán cinco dedos, permitiendo que el seguimiento siga realizándose aunque los dedos no sean visibles totalmente, contrariamente a lo que sucedía en versiones anteriores donde el seguimiento tanto de los dedos como de las manos se perdía en estos casos. Esta técnica que permite que los dedos no desaparezcan gracias a estimaciones de su posición basadas en la posición de los demás dedos es conocida como predicción de seguimiento, del inglés *prediction tracking* y, como se detalla en secciones posteriores proporciona una mejor herramienta para la detección de gestos mediante técnicas de aprendizaje automático. En la *Figura 12* se puede visualizar la representación esquelética de la mano provista por el nuevo *SDK*.

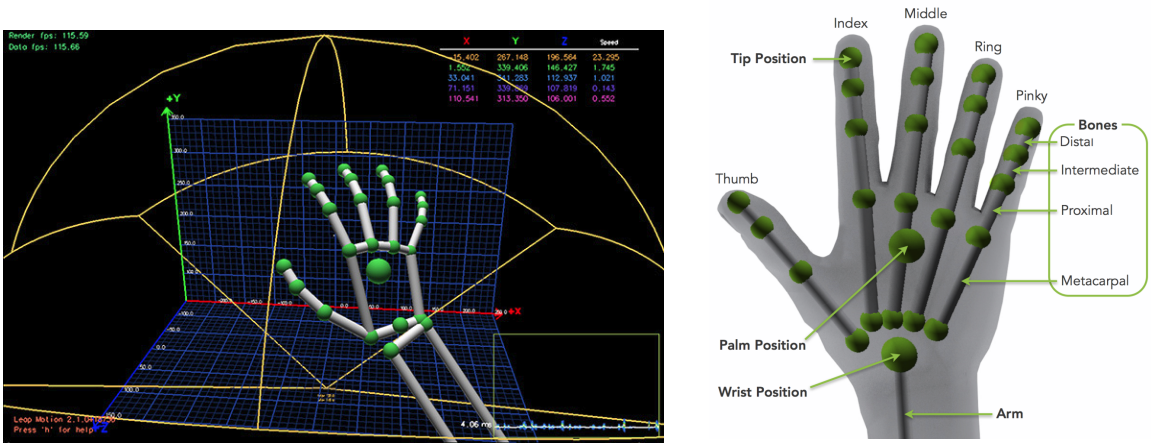


Figura 12: Representación esquelética de la mano provista por *Leap Motion SDK* v2.x [68]

Adicionalmente, esta representación esquelética también trae aparejada otros beneficios:

1. Identificación de manos: es posible identificar cuál es la mano derecha y cuál es la mano izquierda.
2. Información del antebrazo: se agrega información de seguimiento correspondiente al antebrazo del usuario.
3. Gestos: a los gestos ya proporcionados en las versiones anteriores del *Leap Motion SDK* se agrega la detección de nuevos gestos como *finger pinch* y *finger grab*, detallados en secciones posteriores.
4. Dedos: como ya se mencionó, se reporta siempre la existencia de cinco dedos junto con sus posiciones y el tipo de dedo, siendo incluso posible identificar cuáles de ellos están extendidos y cuáles no. Adicionalmente, es posible obtener la posición de cada uno de los puntos de interés o *joints* de cada dedo.

2.2.2.2. Sistema de coordenadas

El sensor *Leap Motion* maneja un sistema de coordenadas cartesiano con origen en el centro del sensor, tal como se puede ver en la *Figura 13*, y los valores son reportados en milímetros [107]. Los ejes correspondientes a X y Z conforman un plano horizontal, siendo el eje X paralelo al lado más largo del sensor y el eje Z perpendicular a éste. Los valores positivos para X son aquellos que se encuentran hacia la derecha del sensor, mientras que los valores positivos para Z son aquellos salientes al mismo. Por otro lado, el eje Y es perpendicular al sensor considerando los valores positivos como aquellos por encima de él.

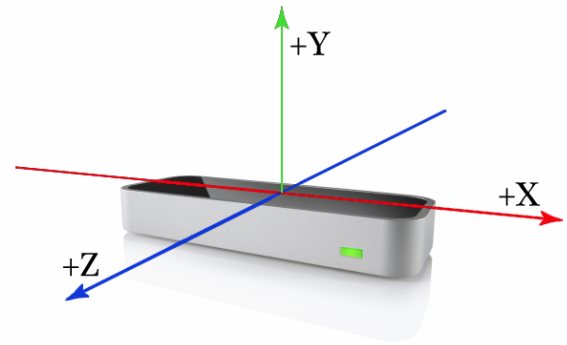
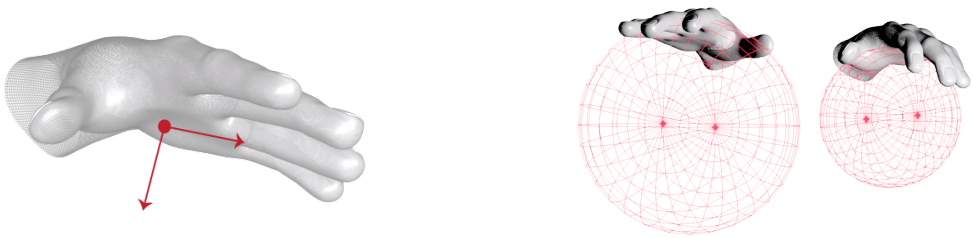


Figura 13: Sistema de coordenadas del sensor *Leap Motion* [62]

2.2.2.3. Seguimiento

Como se mencionó previamente, el sensor *Leap Motion* permite realizar el seguimiento de las manos, comúnmente conocido como *hand-tracking*, y el seguimiento de los dedos que la conforman, también conocido como *finger-tracking*. El sensor proporciona información sobre la posición, características y movimiento de las manos visibles en el campo de visión, junto con una lista de dedos, generando a su vez eventos para ciertos gestos realizados por el usuario. En esta subsección se detallan algunos aspectos referentes a los diferentes tipos de seguimiento brindados por el sensor, describiendo los atributos que provee para cada uno de ellos. En lo que respecta al seguimiento de manos, *Leap Motion* proporciona la posición del centro de la palma, la velocidad y dirección con la que ésta se desplaza, y el centro y radio de una esfera inversamente proporcional a la curvatura de la mano, es decir, que a mayor curvatura menor es el radio de la esfera y viceversa, alcanzando el valor máximo cuando la mano está completamente extendida. Estos atributos se representan en la *Figura 14*.



A. Vector Normal y Dirección de la palma B. Radio de la esfera

Figura 14: Atributos de la mano provistos por el sensor *Leap Motion* [67]

Por otro lado, en lo que refiere al seguimiento de dedos son abstraídos por el sensor como objetos apuntadores y poseen atributos de largo y ancho, así como también la posición de la punta, representados como círculos en la *Figura 15*, y vectores dirección y velocidad, representados como flechas en la misma figura.

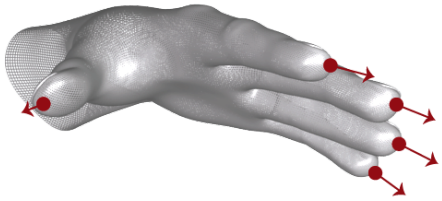
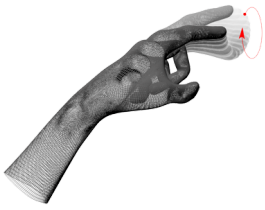


Figura 15: Vectores dirección y posición de los dedos provistos por *Leap Motion* [64]

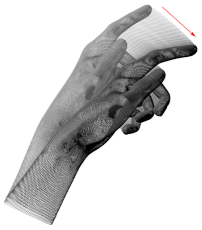
En cuanto a los gestos reconocidos por el sensor, se reconoce un conjunto predefinido de gestos mediante movimientos tanto de las manos y dedos que puedan encontrarse dentro del campo de visión. Algunos de ellos se ilustran en la *Figura 16*.



A. Círculo *(*del inglés *Circle):* movimiento continuo realizado con un dedo, semejante a la forma de un círculo, en sentido horario o antihorario.

B. Deslizar *(*del inglés *Swipe):* movimiento lineal y continuo en cualquier dirección realizado con un dedo.

C. Presión de tecla (del inglés *Key Tap):* movimiento equivalente a presionar una tecla realizado con un dedo.



D. Presión de pantalla *(*del inglés *Screen Tap):* movimiento equivalente a tocar una pantalla vertical realizado con un dedo.

Figura 16: Gestos predefinidos reconocidos por el sensor *Leap Motion* [66]

Adicionalmente a los gestos recientemente descritos, a partir en la versión 2.1 del *Leap Motion SDK* se incluyeron dos nuevos gestos, pellizcar y agarrar, del inglés *pinch* y *grab* respectivamente. Este es uno de

los puntos a destacar de la nueva versión ya que extiende el banco de gestos predefinidos provisto por el sensor.

2.2.3. *openFrameworks*

Otro de los *frameworks* utilizados como parte de la solución propuesta es *openFrameworks* (OF) [104]. Éste brinda múltiples facilidades en lo referente a los aspectos gráficos, actuando como motor gráfico para la generación de elementos gráficos bidimensionales o tridimensionales en pantalla, facilitando el agregado de menús, botones o cualquier otro componente visual que se quiera agregar en la aplicación a nivel de interfaz de usuario. Adicionalmente, actúa como biblioteca de propósito general, brindando la mayoría de las utilidades comúnmente utilizadas en aplicaciones interactivas como manejo de hilos, mutuo exclusión, manejo de red, y otras funcionalidades en forma de extensiones [105].

2.2.3.1. Descripción

openFrameworks es un conjunto de herramientas integradas, que proveen una gran variedad de operaciones de alto nivel para la manipulación de elementos gráficos. El *framework* es de código abierto, está desarrollado en *C++* y brinda una abstracción del funcionamiento de varias bibliotecas comúnmente utilizadas. Algunas de las más importantes son:

1. OpenGL, GLEW, GLUT y Cairo: utilizadas para el manejo de gráficos, ya sea 2D o 3D.
2. FMOD: utilizada para el manejo de entrada y salida de audio.
3. FreeType: utilizada para el manejo de fuentes.
4. FreeImage: utilizada para la manipulación de imágenes.

Otras posibles alternativas a utilizar como *frameworks* de programación de propósito general y *frameworks* gráficos son *Cinder* [14], *vvvv* [157], *Unity* [154], *Pure Data* [119] o *Processing* [115]. En particular, éste último es similar a *openFrameworks* en funcionalidad pero está orientado a desarrollo *Java*.

openFrameworks introduce una estructura predefinida que debe cumplir todo programa que haga uso del *framework*. Así, el programa debe estar compuesto necesariamente, y como mínimo, por tres métodos: *setup(), update()* y *draw()*, conformando de esta manera su cuerpo principal. Toda aplicación que utilice *openFrameworks* ejecuta inicialmente el método *setup()* y luego permanece en un ciclo ejecutando los métodos *update()* y *draw()* de forma secuencial e indefinida para cada nuevo cuadro a dibujar en pantalla, hasta su finalización. Adicionalmente, *openFrameworks* permite manejar eventos de forma sencilla para definir el comportamiento a seguir cuando se presiona cierta tecla del teclado, o un botón del ratón. De todas formas, cabe mencionar que los eventos anteriores no serán utilizados intensivamente como parte de la solución propuesta, ya que la interacción utilizada será completamente natural, sin requerir el uso de un teclado y un ratón.

2.2.3.2. Motivo de uso

Si bien existen un conjunto considerable de bibliotecas que proporcionan herramientas para la manipulación de elementos gráficos, se decidió utilizar *openFrameworks* debido a que se analizaron algunas características relevantes que llevaron a preferir su uso ante las demás. Adicionalmente, fue el *framework* de programación recomendado por el tutor del proyecto, quien tiene experiencia en su utilización. Algunos de los puntos claves para su utilización son:

1. Comunidad: posee una comunidad activa y una gran cantidad de foros de discusión, lo cual hace que *openFrameworks* provea una gran fuente de conocimiento para su fácil aprendizaje.

2. Lenguajes/Plataformas: es multiplataforma, estando actualmente soportado en *Windows, Linux, OSX, iOS* y *Android* y en varios *IDEs* como *CodeBlocks, XCode, Visual Studio y Eclipse*.
3. Arquitectura: es sencillamente extensible y al día de hoy ya cuenta con una gran cantidad de extensiones que se le pueden incorporar a demanda. Está fuertemente pensado para evitar perder tiempo con bibliotecas de bajo nivel como, por ejemplo, *OpenGL*, logrando una gran abstracción y permitiendo poner foco en la parte creativa de la aplicación.

Si bien los motivos anteriormente expuestos fueron lo que llevó a escoger a *openFrameworks* como *software* para manejo de gráficos, se pudo comprobar que obtener resultados visuales llamativos no es para nada sencillo. Es posible experimentar y gestionar de manera rápida formas primitivas, pero si se quiere un resultado visual de mayor calidad la tarea es más compleja. Para ello hay que tener un conocimiento más profundo de la herramienta y hacer uso de *addons*, *shaders* entre otros elementos para lograrlo, lo cual se escapa de los objetivos principales del proyecto.

2.2.4. OpenCV

Para todo lo referente al análisis de imágenes y aplicación de algoritmos de visión por computadora, se utilizó el *framework OpenCV*. En particular, como se detallará en siguientes capítulos, fue utilizado principalmente para implementar la característica multitáctil a ser provista por la solución.

2.2.4.1. Descripción

OpenCV, de su sigla en inglés *Open Source Computer Vision Library* [103], es un *framework* desarrollado originalmente por *Intel* que incluye cientos de algoritmos de visión por computadora y de propósito general. Pretende proporcionar un entorno de desarrollo fácil de utilizar y altamente eficiente gracias a que está desarrollado en *C/C++* optimizados aprovechando las capacidades que proveen los procesadores multinúcleo.

A pesar de contar con primitivas como binarización (transformación de una imagen *RGB* a una imagen en blanco y negro), filtrado y muchas otras funcionalidades referentes a la obtención de múltiples estadísticas de una imagen, *OpenCV* es principalmente un *framework* que implementa algoritmos para detección y reconocimiento de rostros, reconocimiento, segmentación y rastreo de objetos, seguimiento de objetos en movimiento, extracción de modelos 3D a partir de objetos, producción de nubes de puntos tridimensionales, búsqueda de imágenes similares a partir de una base de datos de imágenes, y una gran cantidad de funcionalidades adicionales, todas ellas basadas fuertemente en el análisis de imágenes.

2.2.4.2. Motivo de uso

Se decidió utilizar *OpenCV* debido a un conjunto de ventajas, algunas de las cuales se listan a continuación:

1. Licencia: consta de una licencia del tipo *BSD*, de su sigla en inglés *Berkeley Software Distribution*, que lo hace un *framework* gratuito para uso tanto académico como comercial, factor más que importante para el presente proyecto.
2. Lenguajes/Plataformas: dispone de interfaces para *C*, *C++, Python* y *Java* y es compatible con las plataformas *Windows, Linux, Android* y *Mac OS*, lo que lo hace un *framework* totalmente multiplataforma, otra de las características deseables como parte del proyecto.
3. Comunidad: cuenta con una gran comunidad activa de usuarios, siendo la solución más utilizada en lo que respecta al procesamiento de imágenes y uso de algoritmos de visión por computadora.

4. Aceleración por *GPU*: cuenta con soporte para la ejecución de varios de los algoritmos de procesamiento de imágenes provistos sobre *GPU*, lo que podría permitir mejorar considerablemente la performance obtenida.

2.2.5. *GRT*

Como se mencionó en el capítulo anterior, el objetivo primordial de este proyecto es lograr la integración de la interacción multitáctil con la interacción tridimensional, para lo cual es necesario el reconocimiento de gestos realizados mediante ambas vías de interacción. A pesar de que un análisis profundo sobre estos temas escapa del alcance del proyecto, fue necesario investigar y estudiar mecanismos que permitan el reconocimientos de gestos, como forma de complementar el banco de gestos nativos provistos por los diferentes sensores y bibliotecas utilizados, ya detallados en secciones previas. Para ello, se utilizó *GRT,* de su sigla en inglés *Gesture Recognition Toolkit* [33], una biblioteca que provee diferentes algoritmos de aprendizaje automático para posibilitar el reconocimiento de múltiples tipos de patrones, en particular, patrones en muestras de datos que representan gestos.

2.2.5.1. Descripción

GRT es un *framework* de aprendizaje automático creado por Nick Gillian del *Media Lab* del *MIT (Massachusetts Institute of Technology)*. Está completamente desarrollado en *C++,* es de código abierto y multiplataforma, estando disponible para *Windows, Linux* y *OS X*, y fue diseñado especialmente para la detección de gestos en tiempo real. *GRT* proporciona una variada colección de algoritmos de aprendizaje automático posibles de configurar, y a su vez, para entrenar, mediante el pasaje de datos proporcionados por el sensor de entrada que se esté utilizando, siendo compatible con cualquier tipo de sensor, lo que hace que sea sencillo de extender y adaptar a cualquier algoritmo de procesamiento que se realice sobre los datos proporcionados.

En cuanto a su organización, *GRT* está compuesto por varios módulos dispuestos de forma análoga a una tubería, del inglés *pipeline*, tal como muestra la *Figura 17.* Mediante esta estructura el *framework* permite encadenar múltiples algoritmos para crear un sistema de reconocimiento de gestos acorde a las necesidades particulares del usuario. La tubería consiste de dos fases diferentes con el fin de reconocer un gesto:

1. Fase de entrenamiento: a partir de ciertos datos de entrenamiento se obtiene el modelo entrenado que se utilizará para predecir nuevas muestras en tiempo real, así como también ciertas estadísticas como son la precisión o la matriz de confusión, la que permite visualizar la performance del algoritmo para cada posible clasificación. Ésta a su vez, permite reconocer de forma sencilla posibles falsos negativos y positivos que hayan ocurrido durante el proceso.
2. Fase de predicción: a partir de datos obtenidos en tiempo real por el sensor que se esté utilizando, se obtiene una etiqueta de predicción que representa la clasificación de cada dato particular en base a la información aprendida previamente.

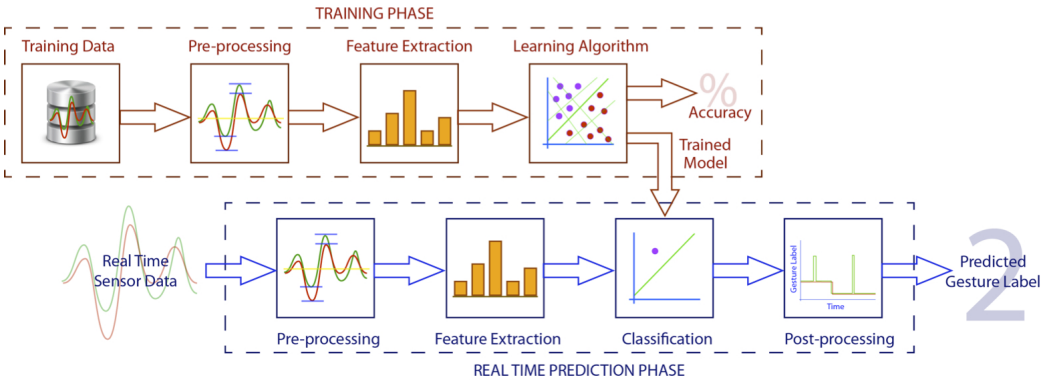


Figura 17: Diferentes fases en las que se utiliza la tubería de reconocimiento [33]

Los módulos que conforman la tubería, tal como se muestra en la *Figura 17*, son los siguientes:

1. Datos de entrenamiento*:* provee operaciones para generar y manipular el conjunto de datos de entrenamiento a ingresar como entrada a la tubería, así como también para testear la precisión del entrenamiento.
2. Preprocesamiento*:* permiten realizar cierto procesamiento a los datos antes de aplicar el algoritmo seleccionado.
3. Extractor de características: permite extraer, a partir de los datos introducidos como entrada, ciertos valores característicos de los gestos a reconocer con el fin de facilitar el reconocimiento.
4. Clasificación: provee operaciones que permiten aplicar el modelo aprendido a las muestras obtenidas en tiempo real.
5. Postprocesamiento: permite realizar cierto procesamiento a las muestras a predecir luego de realizar la clasificación mediante el modelo aprendido.

En términos generales, se requiere básicamente de seis pasos para crear un sistema de reconocimiento de gestos utilizando *GRT*:

1. Seleccionar el algoritmo a utilizar: elegir un algoritmo que sea útil para los gestos que se desee reconocer. La lista de algoritmos de aprendizaje automático provistos por el *framework* permite reconocer tanto poses estáticas como gestos dinámicos que dependen de aspectos referentes al tiempo de realización y al historial de movimiento.
2. Generar la tubería de reconocimiento de gestos: una vez que se selecciona el algoritmo adecuado, se debe crear un nuevo sistema de reconocimiento de gestos para entrenarlo y predecir futuros gestos en tiempo real. Este componente, una vez se cuente con la tubería entrenada, recibe datos y los pasa por la tubería para obtener el gesto predecido.
3. Grabar un conjunto de entrenamiento: antes de utilizar la tubería para predecir futuras muestras, ésta debe ser entrenada en base a ciertos datos de entrenamiento provistos. Este proceso de entrenamiento consiste en grabar determinada cantidad de veces los gestos que se quiere reconocer, asociando en cada caso el valor correspondiente a la salida que se quiera para cada uno, de forma que la tubería pueda generar el modelo que mejor se adapte a estos mapeos y ajustarse para predecir nuevas muestras.
4. Entrenar la tubería: una vez se cuenta con el conjunto de datos de entrenamiento, se debe entrenar la tubería, lo cual genera como salida el modelo que mejor se ajusta a los datos de entrenamiento y el que será utilizado posteriormente para predecir futuras muestras.
5. Probar la tubería: opcionalmente se puede probar la tubería para obtener su precisión para nuevos datos, realizando una pasada de predicción de prueba utilizando un nuevo conjunto de datos. Con el fin de no sesgar los resultados, éste no debe ser un subconjunto del conjunto de datos utilizado durante el entrenamiento.

6. Predecir nuevos datos: una vez que se cuenta con una tubería entrenada y se está conforme con la precisión obtenida en las pruebas, se la puede utilizar para predecir nuevos datos en tiempo real.

Como se mencionó anteriormente, es importante seleccionar un algoritmo de predicción acorde a los gestos que se deseen reconocer para obtener buenos resultados. *GRT* provee un arsenal bastante grande de algoritmos ya implementados, algunos de los cuales son los listados a continuación [36]:

1. *Min Dist:* algoritmo de reconocimiento simple y rápido, que posee bajo costo computacional y que es preferentemente bueno para el reconocimiento de gestos de tipo estático no temporales. Consiste en dividir el conjunto de entrenamiento en grupos de cierto tamaño y asociar a cada uno una clase que representa cierto tipo de gesto. Así, para predecir un nuevo dato se busca la clase que tiene distancia mínima del grupo al nuevo dato.
2. *Decision Trees:* algoritmo sencillo que funciona correctamente incluso en problemas complejos. Se basa en dividir el conjunto de entrenamiento según sus características en conjuntos que se denominan regiones. De esta forma, ante un nuevo dato, se determina según sus características a qué región pertenece. El algoritmo es particularmente útil para detectar gestos estáticos no temporales.
3. *Dynamic Time Warping (DTW):* algoritmo poderoso que funciona muy bien para el reconocimiento de gestos del tipo temporal, es decir, gestos consistentes en la sucesión de movimientos durante un intervalo de tiempo. A partir de los datos de entrenamiento este algoritmo define plantillas asociadas para cada gesto a reconocer. Luego, en tiempo real, busca a cuál de estas plantillas se parece más la muestra particular.
4. *K-Nearest Neighbor Classifier (KNN):* algoritmo sencillo que funciona bien para problemas de reconocimiento simples pero puede ser lento en la predicción de gestos en tiempo real cuando se posee un conjunto de entrenamiento considerable. Este algoritmo no aprende a partir de su conjunto de entrenamiento, sino que utiliza el mismo conjunto de entrenamiento para realizar la predicción determinando el conjunto de k "vecinos" más cercanos a la nueva muestra, motivo por el cual que puede ser lento en algunos casos.

Adicionalmente a las características anteriores, *GRT* provee reconocimiento y descartado automático de gestos inválidos, más conocido como *Automatic Gesture Spotting* [34]. Esta característica lo diferencia de otros *frameworks* similares, permitiendo reconocer gestos sin problemas desde un flujo de datos obtenido a partir de un sensor que puede además contener movimientos genéricos, descartando automáticamente aquellos que no sean lo suficientemente parecidos a ninguno de los gestos que se desea reconocer.

2.2.5.2. Motivo de uso

Se decidió utilizar *GRT* debido a que posee varias ventajas, algunas de las cuales se listan a continuación:

1. Licencia: se encuentra bajo licencia *MIT*, una licencia de *software* libre permisiva que permite la reutilización del código dentro de *software* propietario siempre y cuando todas las copias del mismo incluyan una referencia a sus términos y condiciones.
2. Reconocimiento de gestos: si bien la utilización de *OpenNI/NITE* y *Leap Motion SDK* permiten reconocer gestos nativos para cada uno de los sensores involucrados, éstos son acotados y no es posible extenderlos para nuevos gestos. Por otro lado, ninguno de estos *frameworks* permiten reconocer gestos multitáctiles sobre superficies no capacitivas. El uso de *GRT* permite reconocer nuevos gestos independientemente de tratarse de una superficie con capacidad de cómputo o no, simplemente obteniendo datos desde un sensor y pasándolo por la tubería de predicción para predecir si se corresponde a uno de los gestos objetivo o no.

3. Lenguajes/Plataformas: la versión precompilada del *framework* es provista para la plataforma *Mac* y es posible descargar el código fuente para generar la biblioteca manualmente en las demás plataformas.
4. Documentación: es muy completa gracias a la *wiki* construída especialmente para el *framework*. Cuenta con múltiples ejemplos de uso práctico así como también con toda la teoría necesaria para poder entender y aplicar los conceptos más relevantes relacionados a la disciplina de aprendizaje automático.
5. Comunidad: presenta una comunidad muy activa. En particular, como complemento a la *wiki* mencionada anteriormente, existe un foro mantenido por el propio desarrollador [35], quien brinda respuesta de forma muy ágil y detallada a las consultas planteadas.
6. Algoritmos: brinda un variado y extenso conjunto de algoritmos que proveen gran flexibilidad a la hora de definir los gestos a reconocer. Muchos de estos algoritmos permiten reconocer gestos estáticos mientras que otros permiten reconocer gestos temporales, haciendo que el espectro de posibles gestos a reconocer sea bastante considerable sin importar si son de tipo espacial o multitáctiles. Para esto, basta seleccionar un algoritmo que se adecúe al tipo de gesto a reconocer y proporcionar un conjunto de valores y parámetros acordes para efectuar el reconocimiento. Además, es posible cambiar fácilmente el algoritmo a utilizar, requiriendo muy poca modificación en el código gracias a que *GRT* es lo suficientemente flexible como para hacer prácticamente transparente el algoritmo específico utilizado para el reconocimiento.
7. Dispositivos de entrada: es posible utilizar el *framework* cualquiera sea el dispositivo utilizado para obtener los datos de entrada, ya sea un ratón, datos bidimensionales obtenidos de una mesa multitáctil, datos tridimensionales obtenidos de un sensor *Microsoft Kinect* o *Leap Motion,* o datos obtenidos del acelerómetro o giroscopio de un *smartphone,* ya que son lo único que requiere *GRT* con el fin de entrenar la tubería y predecir futuros datos.

2.2.6. Boost::Serialization

Como se mencionó en el *Capítulo 1* y como se abordará en secciones posteriores, la solución construida es naturalmente distribuida, por lo que van a haber sensores conectados a diferentes computadores, y a su vez, el servidor que lleva el registro de todo lo que sucede en la escena estará en otro computador. Por esta razón, para que el sistema completo se comporte como uno sólo haciendo que sus componentes cooperen entre sí, es necesario que los distintos computadores se comuniquen. Para esta colaboración, resulta vital que los datos naveguen a través de la red hacia el servidor para que éste pueda tener el control global y tome las acciones correspondientes según lo que está sucediendo en la escena en tiempo real. Para que la información viaje a través de la red donde está ejecutándose el *framework* debe ser serializada de cierta forma, siendo ésta una de las tantas funcionalidades provistas por *Boost*.

2.2.6.1. Descripción

Boost [140] es un conjunto de más de 80 bibliotecas de propósito general multiplataforma que extienden las provistas nativamente por el lenguaje *C++*, proveyendo soporte para múltiples aspectos no soportados en *C++* estándar como álgebra lineal, manejo de expresiones regulares, serialización de datos, procesamiento de cadenas de texto, entre muchos otras funcionalidades. La motivación principal por detrás de este conjunto unificado de bibliotecas está basada en la conocida expresión "no reinventar la rueda", debido a que las múltiples bibliotecas provistas están ampliamente probadas y son utilizadas intensivamente por la comunidad, reduciendo la cantidad de errores de las soluciones e incrementando la productividad. Esta cantidad tan variada de bibliotecas hace que sea un excelente complemento a utilizar en cualquier solución desarrollada en *C++*, centralizando la mayoría de las funcionalidades necesarias mediante una biblioteca común. La mayoría de las bibliotecas incluidas en *Boost* cuentan con licencia *Boost Software Licence* [12], posibilitando su uso tanto en soluciones de código abierto, gratuitas o comerciales, aunque aún existe una minoría de ellas que cuenta con su propia licencia.

Como se mencionó anteriormente, para intercambiar estructuras de datos entre varios computadores distribuidos es necesario serializarlas, definiendo el formato con el que serán enviadas por la red de forma que se puedan reconstruir en el otro extremo. Más específicamente, la serialización refiere a la deconstrucción reversible de una estructura de datos en una secuencia de *bytes*, que puede ser luego utilizada para reconstruir una estructura equivalente en un contexto diferente. En otras palabras, la serialización consiste en escribir ciertos datos y objetos en determinado medio, que puede ser un fichero, un *socket* u otro, y luego reconstruirlos en la memoria del mismo computador o uno diferente. Existen múltiples posibles enfoques para implementar la serialización de datos, como por ejemplo enviar las estructuras en memoria directamente en formato binario, definir una forma propia de codificar los datos como un *string*, o serializar los datos en formato *XML*. Cada una de estas alternativas presenta ciertos problemas y desafíos a considerar para realizar una correcta serialización, por lo cual se utilizó el módulo provisto por *Boost::Serialization* [11] con el fin de facilitar la tarea de serialización de datos. Algunas de las características y funcionalidades que brinda son las enumeradas a continuación:

1. Serializar tipos primitivos como booleanos y enteros es trivial nativamente, aunque este no es el caso de, por ejemplo, los punteros, para los que se debe serializar el objeto al que apunta y no el puntero en sí mismo para poder reconstruirlo correctamente. *Boost::Serialization* permite serializar de forma transparente este tipo de datos, lo cual de otra forma sería una tarea compleja. Además permite serializar de forma sencilla contenedores nativos de *C++* como listas, arreglos, mapas, etc., así como también jerarquías de objetos. De esta forma, es posible serializar de manera flexible tipos de datos definidos que contengan otros tipos dentro, clases base instanciadas estática o dinámicamente, entre otros.
2. La serialización resultante es multiplataforma y portable, pudiendo serializar y deserializar los datos independientemente de la plataforma, sea *Windows*, *Linux* u otro. Adicionalmente existe compatibilidad entre arquitecturas de 32 o 64 *bits* si la serialización es realizada en forma de ficheros *ASCII* o *XML*.
3. Provee varios formatos de serialización, siendo lo suficientemente flexible como para cambiar entre uno y otro sin modificar el código de serialización en sí mismo. Esto se logra gracias a abstraer el medio de serialización mediante una interfaz común que permite utilizar uno u otro formato de serialización de forma transparente. Algunos de los posibles formatos a utilizar son texto plano *ASCII*, ficheros en formato *XML* o directamente los datos binarios.

2.2.6.2. Motivo de uso

A pesar de que existen otras bibliotecas para serialización comúnmente utilizadas, como por ejemplo *Protocol Buffers* [116], se decidió utilizar *Boost::Serialization* debido a un conjunto de ventajas:

1. Licencia: su licencia es lo suficientemente flexible como para permitir utilizar las bibliotecas incluidas de forma gratuita tanto para usos comerciales como no comerciales.
2. Lenguajes/Plataformas: es multiplataforma y en cada nueva versión generada se mantiene su portabilidad en cuanto a compiladores y sistemas operativos. A diferencia de otras bibliotecas similares como *Protocol Buffers*, *Boost::Serialization* no requiere aprender un nuevo lenguaje específico de descripción de las estructuras para poder llevar a cabo la serialización. El hecho de contar con un lenguaje específico puede ser ventajoso en caso de requerir comunicarse con sistemas o componentes desarrolladas en lenguajes diferentes o que se encuentren en distintas plataformas ya que permite abstraer el mecanismo de serialización independientemente del lenguaje que se esté utilizando, aunque no es el caso de la solución desarrollada. En cambio, *Boost::Serialization* consiste simplemente del agregado de los métodos de serialización correspondientes para los componentes que se requieran serializar.

3. Comunidad: presenta una gran comunidad donde los desarrolladores participan activamente en listas de correo, foros, *chats*, etc., reportando problemas y brindando sugerencias con el fin de mantener la portabilidad y enriquecer el *framework*.
4. Documentación: la documentación es muy completa, presentando gran variedad de ejemplos así como también la teoría necesaria para comprenderlos.

2.2.7. Eigen

El cómputo de operaciones matriciales está presente en múltiples tipos de aplicaciones que van desde el procesamiento de imágenes mediante algoritmos de visión por computador hasta realidad aumentada, videojuegos y en general, cualquier aplicación interactiva basada en computación gráfica. Por esta razón, es que se hace necesario contar con una biblioteca que permita realizar estos cálculos de la forma lo más eficiente posible, ya que, en muchos casos, requieren gran potencia de cómputo dados los volúmenes de datos que se manejan en estos contextos. Esto se hace más crítico aún en sistemas en tiempo real, donde la performance es uno de los requerimientos más importantes.

2.2.7.1. Descripción

Eigen [25] es una biblioteca de álgebra lineal desarrollada en *C++* enfocada principalmente en brindar operaciones para la manipulación de matrices y vectores de forma eficiente. Adicionalmente a la versión original desarrollada específicamente para *C++*, hoy en día cuenta con adaptadores para múltiples lenguajes de programación ampliamente utilizados como *Java* y *Python*. *Eigen* está licenciada bajo los términos de la licencia *MPL2*, del inglés *Mozilla Public License v2* [87], lo cual hace que sea una solución de código abierto, que puede ser utilizada tanto como parte de *software* de código abierto como privativo. A su vez, es multiplataforma, pudiendo utilizarse con muchos de los compiladores *C++* existentes actualmente como *GCC* o *Microsoft VC*, y en la mayoría de las plataformas más conocidas como *Linux*, *Windows*, *OSX* e incluso plataformas *mobile* como *iOS*.

Cabe mencionar que existen múltiples bibliotecas que cuentan con muchas de estas características, pero ninguna de ellas las incluye todas, siendo *Eigen* la solución universal para el manejo de matrices en *C++*. De todas formas, esto depende de cuáles de estas funcionalidades se desea utilizar para poder sacar el mayor provecho de las mismas. Algunas de las características de la biblioteca son las enumeradas a continuación:

1. Performance: es una biblioteca rápida, optimizada tanto para matrices pequeñas como grandes, haciendo que la performance no difiera considerablemente en cada caso.
2. Facilidad de uso: utilizar *Eigen* es muy sencillo ya que no tiene ninguna dependencia más que la biblioteca estándar *C++* y está basada completamente en plantillas definidas en ficheros de cabecera, por lo que no es necesario ni siquiera compilarla previo a incluirla como parte de una solución. A su vez, provee un *framework* muy versátil que la hace muy fácil de utilizar.
3. Algoritmos: permite manejar cualquier tipo de matriz, sean estas de tamaño fijo o variable, esparsas o no, etc. y provee múltiples algoritmos de álgebra lineal así como también, un módulo de geometría que incluye manejo de transformaciones geométricas como traslación, rotación y escalado. Esta última es la característica más utilizada como parte de la solución propuesta a este trabajo.
4. Comunidad: la biblioteca es utilizada por varias compañías de renombre como *Google*, quien la utiliza para sus algoritmos de aprendizaje automático y visión por computador, y *Willow Garage*, quien la utiliza principalmente en dos de sus soluciones más conocidas como soporte matemático, *Point Cloud Library (PCL)* y *Robot Operating System (ROS)*.

2.2.7.2. Motivo de uso

Eigen fue utilizada principalmente para realizar las transformaciones geométricas necesarias de forma sencilla y eficiente en la solución propuesta. Adicionalmente, cabe mencionar que a pesar de que la algoritmia necesaria para la implementación de la característica multitáctil con sensores de profundidad requiere un manejo intensivo de matrices, en dicho caso no se utilizó esta biblioteca, sino los módulos matriciales provistos nativamente por la biblioteca *OpenCV*. La razón de esto es que, aunque *Eigen* es más performante que *OpenCV* en lo que respecta a operaciones matriciales, las conversiones entre los formatos utilizados por ambas bibliotecas genera cierto procesamiento extra que puede hacer que los resultados sean menos performantes en comparación con utilizar directamente las matrices provistas por *OpenCV*. Otros motivos más generales para la utilización de esta biblioteca son los siguientes:

1. Lenguajes/Plataformas: es compatible tanto con plataformas *Linux*, como *Windows*, *OSX* y *iOS*.
2. Documentación: provee una documentación extensa que describe cada una de sus funcionalidades en detalle, complementada por un foro comunitario activo.
3. Licencia: es de código abierto, lo cual permite utilizarla tanto en soluciones libres como propietarias.
4. Facilidad de uso: utilizar la biblioteca es muy fácil gracias a que provee un *framework* muy sencillo e intuitivo y su implementación está basada fuertemente en el uso de plantillas *C++*.

2.3. Resumen del capítulo

El presente proyecto hace uso de dos tipos diferentes de sensores, *Microsoft Kinect* y *Leap Motion*. Haciendo uso del sensor *Microsoft Kinect* es posible realizar el seguimiento del esqueleto de los usuarios y es utilizado además para posibilitar la interacción multitáctil mediante el análisis del mapa de profundidad de la escena ofrecido por el sensor. Por otra parte, en lo que respecta al sensor *Leap Motion*, se utilizan sus propiedades para registrar el seguimiento de las manos de los usuarios de la escena, así como también para la detección de los gestos realizados con ellas. Para analizar y gestionar los datos proporcionados por cada uno de estos sensores se hace uso de los *frameworks OpenNI* y *NiTE* para el caso del sensor *Microsoft Kinect y el Leap Motion SDK* para el sensor *Leap Motion.*

Por otra parte, se utiliza un conjunto de *frameworks* adicionales que dan soporte a distintas problemáticas. *openFrameworks* es utilizado para múltiples facilidades en lo que refiere a los aspectos gráficos, así como también para el manejo de hilos, mutuo exclusión y gestión de red. Con el fin de analizar el mapa de profundidad provisto por el sensor *Microsoft Kinect* en busca de interacciones multitáctiles se hace uso del *framework OpenCV*. Por otro lado, dado que la solución debe ser naturalmente distribuida resulta vital que los datos viajen a través de la red de forma eficiente, para lo cual se hace uso del *framework Boost::Serialization* con el objetivo de serializar la información intercambiada. A su vez, se utiliza el *framework GRT* para el reconocimiento de gestos no incluídos nativamente en los *frameworks* de control de los sensores, aplicándose tanto para gestos tridimensionales como multitáctiles. Por último, para realizar las transformaciones geométricas necesarias en este tipo de sistemas interactivos de forma sencilla y eficiente se utiliza principalmente el *framework Eigen*.

Capítulo 3: Estado del arte

En este capítulo se comienza haciendo un análisis de los conceptos más relevantes vinculados a las áreas de interés asociadas al proyecto, presentando definiciones, fundamentos y en algunos casos, el estudio de ciertos principios que actualmente hacen a la buena práctica de cada área en particular. De esta forma, se detallan conceptos y áreas fuertemente ligadas a este trabajo como Interacción Persona-Computadora, Interacción Natural, Interacción Multitáctil, entre otras.

Se comienza presentando el área principal en la que está centrado este proyecto, la Interacción Persona-Computadora, presentando algunos de los desafíos y motivaciones que hicieron que ésta surja como una disciplina. Se describen además ciertas técnicas y buenas prácticas que se suelen aplicar en lo que respecta a cualquier sistema interactivo que tenga como principal objetivo la interacción de la forma lo más natural posible con los usuarios finales. Luego, se presentan algunos conceptos que si bien son ramificaciones, especializaciones o subáreas del área genérica de la Interacción Persona-Computadora, es oportuno al menos dar una definición y brindar algunos ejemplos de utilización, ya que como se mencionó anteriormente, todas ellas están de cierta forma involucradas en este proyecto. Algunas de dichas áreas son la Computación Ubicua, la Realidad Aumentada y la Interacción Natural.

Habiendo descrito los conceptos básicos que dan fundamento a las áreas que abarca el presente proyecto, se presenta el estado del arte, detallando diversos proyectos y trabajos de investigación que están de cierta forma relacionados con las áreas de interés mencionadas. Por esta razón, se puede ver a estos trabajos como similares a la solución detallada en este informe, en el sentido de que abarcan las mismas áreas de investigación y en algunos casos, fueron una fuente de inspiración directa.

3.1. Interacción Persona-Computadora

La interacción tanto entre seres humanos como entre un ser humano y el entorno que lo rodea se da de forma natural [92]. Sin embargo, junto con el gran avance tecnológico y los nuevos medios, esta interacción se ha convertido en algo un poco más complejo debido a la intervención de una computadora. El análisis de este tipo de interacción es el principal interés de la disciplina Interacción Persona-Computadora o más comúnmente denominada *HCI*, del inglés *Human-Computer Interaction*. Aunque la situación clásica sería una persona usando un programa sentada frente a un computador, alrededor de la Interacción Persona-Computadora hay un amplio abanico de formas posibles de interacción, ya que los computadores pueden formar parte de un avión, monitores de ruta en coches, teléfonos móviles, libros electrónicos, sistemas de realidad virtual, etc. Los usuarios pueden formar parte de grupos u organizaciones, y por lo tanto, es necesario contar con sistemas distribuidos que permitan el trabajo cooperativo. Todas estas situaciones en las que un humano debe interactuar con un sistema computarizado deben ser manejadas por la disciplina de la Interacción Persona-Computadora.

En este escenario, es necesario utilizar técnicas de interacción que no presuman ningún conocimiento previo del usuario ni le atribuyan ninguna otra habilidad interactiva que no sea la que le permite relacionarse con otras personas o el entorno que lo rodea. Se busca entonces que las interacciones entre el hombre y las computadoras imiten a aquellas a las que los seres humanos están acostumbrados con el fin de hacer más agradable e intuitiva su utilización. A esta forma de interacción particular se la denomina Interacción Natural y la Interacción Persona-Computadora es la disciplina que la estudia. Por "interacción" se entiende a todos los intercambios que se dan entre la persona y la computadora, sean estos mediante mensajes, comandos u otros tipos de entradas y/o salidas soportadas. Una interacción eficiente está centrada en el usuario [39] y busca que éste entienda exactamente lo que se encuentra haciendo en todo momento, y a su vez, que sea consciente siempre del lugar en que se encuentra en las diferentes posibilidades de navegación. Todo esto sin que sea necesario pensar en cómo funciona o cómo encontrar la forma de realizar cierta tarea, sino, por el contrario, que pueda centrarse en cumplir las tareas

requeridas.

Cuando los seres humanos y las computadoras interactúan lo hacen a través de un medio o interfaz, un espacio que refleja las posibles funciones a realizar y el balance de poder y control. La interfaz está constituida por diferentes dispositivos, tanto físicos como lógicos, que permiten al usuario interactuar de manera precisa. En definitiva, la interfaz está conformada por aquellas partes de la computadora con las que el usuario entra en contacto, sea física (mediante diferentes tipos de dispositivos de interacción como teclados, ratones, superficies táctiles, sensores, etc.) o cognitivamente, comprendiendo la información que se le presenta. Desde el punto de vista del usuario, la interfaz es todo el sistema que compone la computadora. Es la parte que el usuario ve, oye, toca y con la que se comunica, por lo que una interfaz de usuario ineficiente origina problemas como la reducción de la productividad, el incremento del tiempo de aprendizaje o niveles de error inaceptables [39].

En este sentido, una interfaz que provea una experiencia intuitiva es aquella en la que el usuario simplemente se enfrenta a ella y comienza a utilizarla gracias a que es de fácil aprendizaje y provee las instrucciones, tutoriales y retroalimentación correctas [53]. Sin embargo, la interfaz es también un límite a la comunicación en muchos casos, ya que aquello que no sea posible de expresar a través de ella permanecerá fuera de la relación mutua [20]. En algunos casos, estos límites derivan del estado actual de los conocimientos de los usuarios acerca de cualquiera de las partes implicadas, donde la interfaz se convierte en una barrera debido a un pobre diseño y a la falta de instrucciones, tutoriales y retroalimentaciones. A pesar de que diseñar sistemas interactivos en donde las personas puedan aprender fácilmente su uso es bueno, es aún mejor diseñarlos de forma tal que las personas los utilicen sin siquiera percibir qué es lo que está pasando más allá de la interacción [53]. En definitiva, para ser intuitiva, una interfaz debe ser fácilmente asimilable, ayudando al usuario a construir un modelo mental del impacto que tiene cada una de las acciones realizadas. Esto hace que la interacción con la interfaz se transforme paulatinamente en un hábito y se logre reducir la cantidad de carga cognitiva necesaria para su utilización. La gran mayoría de los hábitos que se consideran intuitivos como caminar, andar en bicicleta, etc., se hacen intuitivos con el paso del tiempo, no de un momento para otro. Todos ellos comienzan siendo tareas complejas, las cuales, al seguir reglas entendibles, predecibles y consistentes, se tornan asimilables. Esta predecibilidad en conjunto con una buena retroalimentación es lo que convierte a estas actividades complejas en hábitos, y lo mismo aplica para los sistemas interactivos. De esta forma, en lugar de enfocarse en la interacción, los usuarios se enfocan en la experiencia que están teniendo y la tarea que están tratando de realizar [53]. El hecho de convertir acciones en hábitos es un objetivo a largo plazo, ya que el proceso requiere primero del aprendizaje y el entendimiento, que se irá transformando en hábito a lo largo de las semanas o meses de uso, y se tornará finalmente muy difícil de olvidar. Adicionalmente, hay que tener en cuenta que lo que es intuitivo para una persona puede no serlo para otra, dependiendo de su edad, experiencia y lenguaje, por lo que se debe comprender e investigar en detalle cuáles serán los usuarios finales de la aplicación en cuestión.

Los objetivos de la Interacción Persona-Computadora según Dan Diaper son desarrollar o mejorar la utilidad, efectividad, eficiencia, usabilidad y seguridad de sistemas que incluyen computadoras, entendiendo por "sistema" no solo al *hardware* y al *software* que la componen sino también a todo el entorno [129]. Según el autor, para hacer sistemas interactivos e intuitivos hace falta considerar los siguientes factores:

1. Factores psicológicos de los usuarios: los procesos cognitivos, la capacidad personal, el nivel de experiencia y la motivación.
2. Factores organizativos: el entrenamiento, el diseño del local de trabajo y su organización.
3. Factores del entorno: posibles ruidos, la ventilación/calefacción e iluminación, entre otros varios factores que repercuten en el entorno en que se encuentra el usuario.
4. Factores de salud y seguridad: estrés, dolores de cabeza, molestias visuales y desórdenes musculares.
5. Factores de confort: la comodidad de la silla en la que el usuario se encuentra sentado, si es que lo está, y el diseño del equipamiento utilizado.

6. Factores sobre interfaz de usuario: los dispositivos de entrada, las pantallas de salida, el diseño de los diálogos, el uso del color y los iconos utilizados, entre otros.

Se debe trasladar la comprensión de todos estos factores para desarrollar herramientas y técnicas que permitan a los diseñadores lograr sistemas computarizados acorde a las actividades para las que se aplicarán, y en base a esto, conseguir una interacción eficiente, efectiva y segura. Es muy importante comprender el hecho de que los usuarios no deben cambiar radicalmente su manera de ser y actuar, sino que por el contrario, los sistemas deben ser diseñados para satisfacer sus requerimientos. Para que un sistema interactivo cumpla sus objetivos tiene que ser usable y debido a la generalización del uso de los computadores, accesible a la mayor parte de la población humana. Por usabilidad se entiende a la medida en la que un producto puede ser usado por determinados usuarios para conseguir objetivos específicos con efectividad, eficiencia y satisfacción en un contexto de uso especificado, mientras que por accesibilidad se entiende a la flexibilidad para acomodarse a las necesidades de cada usuario y a sus preferencias y/o limitaciones [143]. En el *Anexo A* se presentan más detalles referidos a los principios de diseño para lograr una buena interacción.

Debido a todos estos factores, las referencias científicas relacionadas con la Ingeniería de *Software* en general y con Interacción Persona-Computadora en particular, sugieren la formación de equipos de desarrollo de sistemas compuestos por personas procedentes de disciplinas diversas, es decir, equipos multidisciplinarios. Tales equipos juegan un papel determinante en el desarrollo de *software* actual y futuro, ya que complementan el papel del ingeniero de *software* y del programador para producir sistemas que verdaderamente recojan las necesidades de los usuarios y su contexto, ofreciendo sistemas más agradables, más eficientes y, en definitiva, más fáciles de utilizar [39]. Algunas de estas disciplinas adicionales requeridas son:

1. La psicología cognitiva: se encarga de estudiar la percepción, la memoria, los modelos mentales y el modo en que las personas obtienen, perciben y procesan la información, puntos esenciales para conocer cómo utilizan una interfaz.
2. La ergonomía: estudia las características físicas de la interacción como el entorno físico, las pantallas, etc., permitiendo diseñar la solución acorde al contexto en el que se encuentra inmerso el sistema, haciendo que se incremente la eficiencia de la interacción y se simplifiquen las tareas.
3. El diseño: permite que el sistema sea visualmente agradable, siendo también esencial para lograr sistemas usables e incluso accesibles.

Con estos equipos multidisciplinarios surgen conceptos como el de diseño centrado en el usuario, del inglés *user-centered design*, que incorpora todos los conocimientos de las diferentes disciplinas para poder centrar el diseño de interacción en las necesidades y capacidades específicas del usuario. De esta forma, el usuario se convierte en una fuerte influencia para las decisiones tomadas en las distintas etapas del proceso, obteniendo una solución que brinda una mejor experiencia de uso. En el caso del presente trabajo, si bien la conformación de un grupo multidisciplinario sería lo ideal, es inviable, ya que quienes lo llevan a cabo son estudiantes únicamente del área de la Ingeniería de *Software*. Por ello, se debe paliar esta desventaja poniéndose de cierta forma en cada uno de los roles descritos en la mayor medida posible.

Con el diseño centrado en el usuario, surgen técnicas como la del prototipado rápido, del inglés *rapid prototyping*, que permiten obtener retroalimentación rica y constante por parte de los usuarios y consecuentemente amoldar la solución a sus necesidades. Estas técnicas se basan en la construcción de prototipos de diversa naturaleza que posibilitan obtener un primer acercamiento a cómo será el flujo de interacción final, evaluar la experiencia del usuario y obtener la mayor cantidad de oportunidades de mejora posibles [118]. En el *Anexo C* se puede encontrar un estudio más detallado sobre estas técnicas, incluyendo la descripción de algunos de los prototipos construidos como parte del proceso de desarrollo de este trabajo, ya que algunas de estas técnicas de prototipado rápido fueron utilizadas en los comienzos del proyecto para conceptualizar muchas de las características del sistema a desarrollar.

Por todo lo expuesto, el presente proyecto debe brindar los mecanismos necesarios para que se puedan desarrollar aplicaciones considerando estas buenas prácticas de la disciplina Interacción Persona-Computadora. Específicamente, posibilitar los mecanismos para una interacción centrada en el usuario, brindando retroalimentación acorde a cada interacción establecida. Esta interacción debe ser lo más parecida posible a la que los seres humanos están acostumbrados, sin que sea necesario pensar en cómo funciona el sistema sino que puedan centrarse en cumplir cada tarea. Por información más detallada sobre Interacción Persona-Computadora como disciplina, las disciplinas relacionadas y las dificultades de unificar todos los conocimientos en un objetivo común que se traduzca en un diseño de Interacción Persona-Computadora eficiente, referirse al *Anexo B*.

3.2. Computación Ubicua

La ubicuidad es la propiedad por la que una entidad se encuentra en todos los sitios al mismo tiempo. De este modo, la computación ubicua refiere a un tipo de interacción en la que tanto el procesamiento de la información por parte de los sistemas computaciones como las entradas y salidas que estos proveen están fuertemente integradas con las actividades y objetos cotidianos. De esta forma, se habilita a los usuarios a acceder a servicios de información acordes al contexto en el que se encuentren, donde sea y cuando sea, del inglés *at anywhere and anytime,* sin tener que centrarse en el computador sino solamente en la actividad que estos quieran realizar con el mínimo esfuerzo cognitivo y la mayor comodidad posible [152]. Si bien la idea conceptual por detrás de la computación ubicua surgió en la década de 1990, cuando Mark Weiser de *XEROX* acuñó el término por primera vez [145, 136], para ese entonces no existía la tecnología necesaria para llevar a cabo su desarrollo. Hoy en día, luego de varias décadas de progreso, estas ideas son productos viables y la computación ubicua se ha convertido en un término cada vez más común. Su evolución ha empezado a acelerarse recién en los últimos años debido al desarrollo de las tecnologías de la información y las comunicaciones, así como también al cumplimiento de la *Ley de Moore*[6], permitiendo que pequeños computadores de bajo costo se integren en una gran variedad de objetos cotidianos [72].

Otros avances que han impulsado el desarrollo de este paradigma son los recientes logros en los campos de los microsistemas y la nano-tecnología, permitiendo que pequeños microsensores sean embebidos para detectar una gran cantidad de parámetros del entorno. De esta forma, se ha posibilitado la construcción de sensores como las *RFID tags*, que son etiquetas de identificación basadas en radiofrecuencia, también conocidas genéricamente como *smart labels* [130]. También han habido avances considerables en el área de las tecnologías utilizables, del inglés *wearable technology*, que buscan construir dispositivos que puedan ser utilizados directamente sobre el cuerpo de las personas, abriendo un gran abanico en las formas de interacción persona-computadora [141]. Algunos ejemplos son relojes inteligentes capaces de llevar datos sobre la salud del usuario y enviarlos directamente a su médico personal, ropa elaborada con materiales especiales capaces de cambiar sus características cuando se estira o se dobla, o gafas que agregan información virtual a lo que el usuario está viendo.

Según describe Weiser, se considera que la computación ubicua establece el inicio de la tercer era de los sistemas de información [18]. La primer era estuvo dominada mayormente por los sistemas computacionales de gran escala que eran utilizados por una gran cantidad de usuarios necesariamente expertos. Luego, la segunda era, se caracterizó por el uso del modelo de un dispositivo de cómputo por usuario (computadora personal), produciéndose una relación más estrecha entre el hombre y la computadora. Posteriormente existe una etapa intermedia a ésta denominada "cómputo móvil" la que es considerada el paso previo a la era de la computación ubicua, distinguida por enfocarse en aspectos referentes a la movilidad tanto de los usuarios como de los dispositivos de cómputo y su interconexión

[6] La ley de Moore es una ley enunciada por Gordon E. Moore que expresa que cada dos años aproximadamente se duplica el número de transistores en un circuito integrado.

gracias al advenimiento de *Internet*. Finalmente, la última era es la de computación ubicua, donde múltiples computadoras sirven a una única persona para facilitar su trabajo y hacer el entorno más amigable. Lo mencionado anteriormente se puede ver resumido en la tabla presentada en la *Figura 18*, en la que se observa que la diferencia principal entre el cómputo móvil y el cómputo ubicuo es que en el primer caso los dispositivos brindan su servicios de forma independiente, mientras que en el segundo los dispositivos se integran de tal forma que el usuario se ve completamente inmerso en ellos.

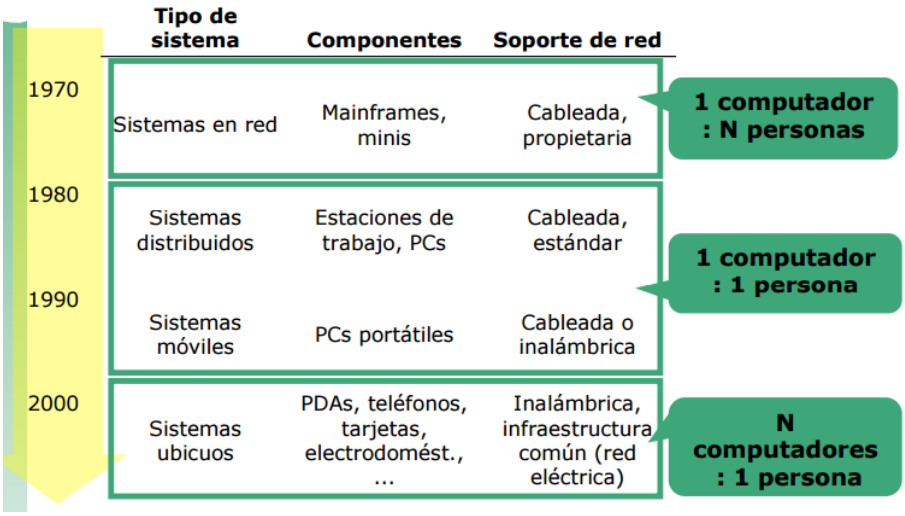


Figura 18: Evolución de la computación en alto nivel [137]

Desde un punto de vista tecnológico, la computación ubicua refiere a la posibilidad de conectar cada uno de los dispositivos del entorno entre sí mediante una red y a su vez, a *Internet*, lo cual se denomina comúnmente "*Internet* de la Cosas", del inglés *Internet of Things,* para poder así obtener información sobre cualquier objeto en cualquier momento y en cualquier lugar [51]. Este término ha surgido gracias a los avances en las redes inalámbricas (principalmente redes inalámbricas de corto alcance como *WiFi* o *Bluetooth*) y refiere a la interconexión de diferentes dispositivos de cómputo u objetos de uso cotidiano con el fin de enviar o recibir contenido remoto para sacar el mejor provecho del objeto en cuestión. Hoy en día, existe una gran variedad de objetos cotidianos conectados a *Internet*, desde teléfonos móviles, pasando por televisores y otros electrodomésticos, hasta lámparas, relojes y juguetes. En este escenario, no se interactúa con un único dispositivo como es usual sino con múltiples dispositivos de forma simultánea que controlan el entorno en el que se encuentra inmerso el usuario, quien generalmente no es consciente de su existencia. El procesamiento de la información y las capacidades de comunicación quedan integradas en objetos que no aparentan ser aparatos electrónicos, haciendo que las capacidades de la computación se vuelvan completamente ubicuas y universales.

Cabe mencionar que el paradigma de la computación ubicua también es conocido por otros nombres, aunque muchos de ellos tienen un enfoque ligeramente diferente, como puede ser inteligencia ambiental o *pervasive computing* [114]. Además, actualmente existen derivaciones de la computación ubicua como, por ejemplo, los sistemas sensibles al contexto que reaccionan ante la interacción de los usuarios en base al contexto en el que se ejecutan. Existen múltiples casos de este tipo de mecanismos en sistemas interactivos pero principalmente están presentes en los teléfonos celulares inteligentes de hoy en día, que se adaptan a la orientación de los elementos mostrados en pantalla según la posición en la que sea tomado el dispositivo, o se facilita la identificación de usuarios físicamente próximos con el fin de intercambiar información.

A modo de ejemplo, en la *Figura 19 y Figura 20* se presentan habitaciones basadas completamente en la computación ubicua, donde a pesar de ser tipos de habitación bien diferentes, en ambas existen múltiples unidades de procesamiento y diferentes dispositivos que se integran con el entorno, comunicándose entre ellos para controlar todo lo que sucede en la sala. Así, por ejemplo, si el detector de movimiento detecta que el usuario está acostado en el sillón puede enviar la orden de reducir la intensidad

de la luz al dispositivo de control de iluminación, o el sistema de notificación automática puede presentar información al usuario sobre la mesa como nuevos mails y/o noticias, que se pueden consultar y manipular en base a interacción gestual, sea en el aire o multitáctil, sin ningún intermediario. Además, la heladera puede encargar automáticamente suministros si detecta que están en falta y el paraguas puede contar con capacidades de comunicación de forma de advertir sobre posibles lluvias y tormentas, entre otros muchos posibles escenarios.

Figura 19: Habitación basada en comp. ubicua [13]

Figura 20: Oficina basada en comp. ubicua [6]

Actualmente existe una ciudad ubicua en construcción denominada *U-city*, que se encuentra en *New Songdo City* [153], una isla ubicada a 60 kilómetros al oeste de Seúl en Corea del Sur. La ciudad se basa en la idea de que todos los servicios provistos a los ciudadanos así como también las diferentes formas de infraestructura sean complementamente ubicuas, haciendo que los ciudadanos se comuniquen entre ellos, así como también con el entorno que los rodea, de la forma lo más transparente posible. Por más detalles sobre esta ciudad referirse al *Anexo D*.

En lo que respecta al presente trabajo, el sistema desarrollado consiste de un conjunto de servidores y sensores conectados entre sí mediante una red de área local (*LAN)*, controlando de forma simultánea el entorno donde se encuentra inmerso el usuario y aplicando la ubicuidad al ser éstos transparentes, ya que el usuario no es consciente de su existencia. A su vez, la superficie con la que interactúa el usuario es parte de la escena en la que se encuentra, por lo que está inmersa como parte de objetos cotidianos dentro de la sala.

3.3. Realidad aumentada

La realidad aumentada es una tecnología en creciente desarrollo y utilización desde hace ya varios años [122], que consiste básicamente en añadir información digital a los objetos reales al ser estos visualizados mediante una cámara o dispositivo similar. De esta forma, es posible aumentar la realidad que ve el usuario con elementos virtuales como videos, imágenes, sonidos o incluso haciendo uso de aromas y sensaciones [7] que agregan información adicional que de otra forma el usuario no podría detectar naturalmente. Esto permite crear experiencias que toman lo mejor de ambos mundos, el mundo real en el que vivimos y el mundo virtual generado computacionalmente.

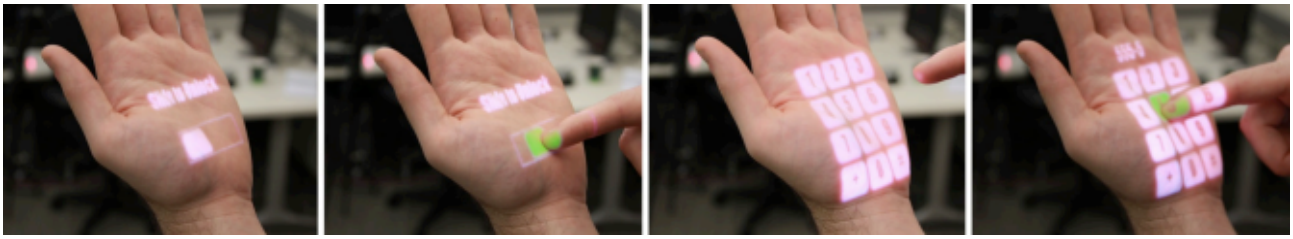


Figura 21: Realidad aumentada en superficies de interacción inusuales [52]

Una definición genérica de realidad aumentada sugiere que para que un sistema sea considerado

como tal debe cumplir tres requisitos básicos, independientemente de los dispositivos que se utilicen [52]:

1. Combinar objetos reales y virtuales en un entorno real.
2. Funcionar de forma interactiva en tiempo real.
3. Mantener los objetos reales y virtuales constantemente sincronizados. Por ejemplo, si el usuario está usando gafas de realidad aumentada y se desplaza, la visualización virtual debería mantenerse en el mismo lugar del espacio real.

Para lograr aumentar la realidad, en su versión más sencilla, es necesario una cámara que provea información del mundo real, un procesador que compute y procese dicha información, y una pantalla en la que se muestre la información procesada. Estos recursos necesarios existen en múltiples tipos y variantes. Por ejemplo, se puede aumentar la realidad en base a simples cámaras *RGB* que capten información de vídeo del entorno en el que se encuentra inmerso el usuario y algoritmos que permitan realizar el seguimiento de objetos. O de forma más compleja, complementado con cámaras de profundidad que permitan agregar información sobre distancia y profundidad de los objetos. A su vez, es posible utilizar diferentes tipos de pantallas en donde se muestre la información procesada, pudiendo ser la pantalla de un dispositivo de uso común como un *smartphone*, *tablet* o *laptop*, o de dispositivos más complejos como son pantallas montadas en la cabeza, del inglés *head-mounted displays (HMD)*, como por ejemplo *Microsoft HoloLens* [80], gafas de realidad aumentada como *Google Glass* [37], o sistemas basados en proyecciones como *Augmented Reality Sandbox* [141, 8].

En particular, *Google Glass*, presentado en la *Figura 22*, fue desarrollado por el laboratorio R&D de *Google* y consiste en un gafa especial que proyecta información del entorno en el que se encuentra el usuario en una pequeña pantalla que se encuentra muy cercana de su ojo derecho. Adicionalmente, incluye conexión a *Internet* vía *WiFi*, por lo que puede obtener y presentar información en tiempo real sobre el clima, diferentes indicaciones según dónde se encuentre el usuario, entre muchas otras posibilidades. Esto permite que una gran cantidad de información aparezca delante de los ojos del usuario superpuesta a las imágenes del mundo real que él mismo observa. Esta información es además actualizada para reflejar los movimientos de su cabeza y hacia qué dirección está observando. Por otra parte, *Augmented Reality Sandbox*, presentado en la *Figura 23*, simula y modela diferentes tipos de terreno y recursos naturales mediante el uso de arena y proyecciones, permitiendo visualizar información referente a la elevación y otras características del terreno.

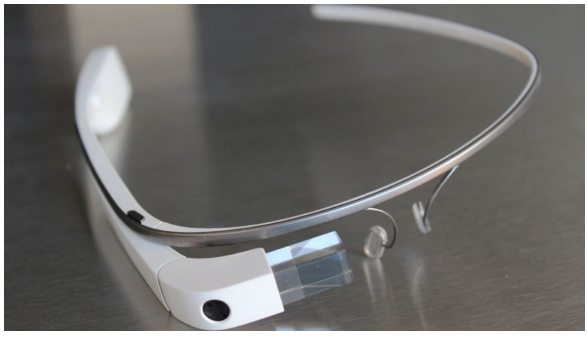


Figura 22: *Google Glass* [37]

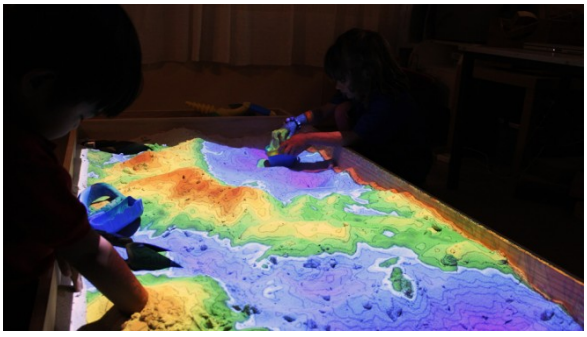


Figura 23: *Augmented Reality Sandbox* [147]

Finalmente, *Microsoft HoloLens*, presentado en la *Figura 24*, es un dispositivo de realidad aumentada desarrollado por *Microsoft* que permitir al usuario visualizar contenido virtual en tres dimensiones directamente sobre la habitación en la que se encuentra. Esto es gracias a que tiene embebido un dispositivo con tecnología similar a la del sensor *Microsoft Kinect* que le permite reconocer la estructura del entorno en el que se encuentra el usuario, y el cual posibilita además el reconocimiento de los movimientos y gestos que el usuario realiza con sus manos frente al dispositivo.

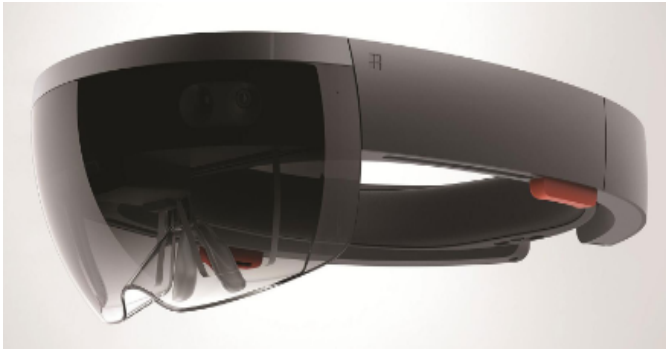


Figura 24: *Microsoft HoloLens* [80]

Adicionalmente, en la realidad aumentada puede ser necesario el uso de marcadores. Estos marcadores son usualmente similares a un código de barras con cierto patrón impreso, y permiten reconocer y realizar el seguimiento de objetos sobre los que se desea agregar información virtual [121]. Este caso es muy común en aplicaciones existentes hoy en día para *smartphones* y *tablets*, donde se escanea el entorno mediante la cámara del dispositivo en búsqueda de un marcador y una vez que se lo encuentra se incorpora la información virtual sobre él. Los posibles elementos virtuales agregados van desde un producto particular con el fin de promocionarlo, permitiendo que el usuario lo visualice en una representación tridimensional, hasta juegos con personajes controlados mediante el marcador, o incluso contenido estático que simplemente presenta información al usuario. De todas formas, con el avance de los algoritmos de reconocimiento de imágenes y patrones, y el avance en los algoritmos de visión por computador en general, hoy en día en muchos casos no es necesario el uso de marcadores para agregar información virtual de forma correcta. Dichos algoritmos son capaces de reconocer directamente objetos del mundo real (por su forma, tamaño, color u otra característica distintiva), así como también rostros y otros objetos sobre los que se suele mapear información virtual [126].

Figura 25: modelo 3D proyectado sobre marcador para realidad aumentada [122]

Por otro lado, existe un tipo de realidad aumentada basada en las capacidades del dispositivo de captar la posición del usuario, ya sea mediante *GPS* o diferentes algoritmos de geolocalización, ofreciendo diversos tipos de información dependiendo de dicha posición. De esta forma, el usuario puede por ejemplo, ser guiado dentro de una ciudad, obtener información de los edificios históricos cercanos al lugar en que se encuentre, o visualizar información sobre las constelaciones dependiendo de la zona del cielo al que dirija el dispositivo utilizado [126].

Como se ha mencionado a lo largo del documento, el presente proyecto busca utilizar una superficie del mundo real para proyectar un mundo virtual con el cual el usuario pueda interactuar mediante diferentes vías de interacción, aplicando entonces el concepto de realidad aumentada. En este caso los sensores *Microsoft Kinect* y *Leap Motion* proveen la información del mundo real que los computadores conectados a cada uno computan, la cual es luego procesada por el servidor antes de ser presentada al

usuario de forma visual sobre la superficie física de interacción, aumentándose con elementos virtuales interactivos.

3.4. Interacción natural

Como se mencionó en secciones anteriores, debido al creciente interés en la Interacción Persona-Computadora, es naturalmente necesario utilizar técnicas de interacción que no asuman ningún conocimiento previo del usuario ni le atribuyan ninguna otra habilidad que no sea la que le permite relacionarse con otros seres humanos y el mundo real. Las personas utilizan mayormente las manos para relacionarse con su entorno, para mover objetos que estén a su alcance e incluso para complementar otras interacciones como puede ser la verbal, por lo que un componente muy importante en lo que respecta a este tipo de interacción natural es la interacción gestual.

La interacción natural busca que las interacciones entre el hombre y las computadoras imiten a las existentes entre seres humanos, permitiendo inventar, diseñar y crear sistemas capaces de actuar recíprocamente con las personas de forma natural, respetando así la percepción y la forma natural de comunicación humana. En particular, se basa en construir interfaces adaptadas a las capacidades de las personas, de forma tal que permitan concentrar su atención en realizar la tarea en vez de lidiar con la forma de realizarla. De este modo, la computadora y los dispositivos que habilitan la interacción desaparecen en el ambiente y el entorno se convierte en la principal interfaz, haciendo que no se deba utilizar ningún elemento adicional para lograr una interacción satisfactoria e intuitiva. De este modo, habrán múltiples computadoras pero estarán más naturalmente integradas y serán menos visibles como parte de superficies, estructuras y objetos, permitiendo que los usuarios se concentren en el contenido. Lo anterior impulsa uno de los pilares fundamentales de la computación ubicua descrita en la sección anterior.

Actualmente, las personas son fuertemente dependientes de las computadoras para resolver tareas en casi cualquier ámbito, ya sea con el fin de procesar información, obtener entretenimiento o incluso para el control de sistemas en automóviles y casas inteligentes. Dada esta realidad, está clara la importancia de una buena interfaz con las computadoras. Sin embargo, a pesar de que la capacidad de procesamiento y prestaciones se ha incrementado considerablemente, muchas interfaces aún basan su funcionamiento en el uso del ratón y el teclado como en los años 70's. De todos modos, como se ilustra en la *Figura 26*, las interfaces basadas en ventanas, iconos y punteros, comúnmente conocidas como WIMP del inglés *Windows, Icons, Menus and Pointers*, están desapareciendo, dándole lugar a las interfaces basadas en nuevos mecanismos de interacción [50].

En esta transición, las interacciones basadas en gestos mediante interfaces gestuales sin cables están teniendo un foco realmente importante dentro del amplio espectro de interfaces de usuario naturales. Estas se aplican principalmente cuando se requiere completa libertad de movimiento, o generar una experiencia de inmersión, como ocurre por ejemplo en las habitaciones inteligentes o en los videojuegos. Las primeras soluciones brindadas para este tipo de interacción requerían que el usuario utilice grandes sensores, guantes u otro tipo de equipamiento considerado intrusivo. Sin embargo, esta necesidad ha ido desapareciendo a medida que los sensores, los microprocesadores y el *hardware* en general han incrementado su poder de cómputo y decrementado su costo y tamaño, complementado por nuevas técnicas de desarrollo de *software* que permiten captar, interpretar y responder a movimientos y gestos humanos [50].

Adicionalmente, durante esta transición ocurrió la invención de las pantallas multitáctiles, quienes también permitieron cambiar la forma de interactuar con las computadoras, posibilitando la manipulación directa de los elementos visualizados en la pantalla mediante los dedos o un lápiz óptico. Mediante esta interacción se obtiene una respuesta inmediata, más aún con el uso de retroalimentación háptica, del inglés *haptic feedback*, que permite brindar al usuario no solo una retroalimentación visual percibida por el sentido de la vista sino también una retroalimentación táctil percibida mediante el tacto. Ejemplo de esto

pueden ser vibraciones u otros estímulos que puedan ser percibidos físicamente.

Figura 26: Evolución de las interfaces de usuario [50]

Este tipo de interacción hizo que surgieran las interfaces multimodales [50], en las que se combinan varios métodos de entrada (típicamente manos, pies, cabeza y voz) y de salida para realizar cierta tarea. Esto extiende la interfaz gráfica de usuario con el fin de incrementar la usabilidad y accesibilidad. Un claro ejemplo de interacción multimodal es el automóvil, donde las manos, brazos, pies y piernas contribuyen conjuntamente para manejar el vehículo, y además, con la visión, oído y tacto, se monitorea el entorno y se actúa en consecuencia. Algunos de los métodos de entrada y salida utilizados en conjunto en los sistemas multimodales son el reconocimiento de la voz como entrada para recibir comandos y la síntesis de voz como salida para brindar retroalimentación al usuario, los gestos táctiles y el reconocimiento de movimientos utilizando dispositivos como acelerómetros y/o giroscopios. Actualmente no es extraño observar todos estos tipos de entradas y salidas en un dispositivo móvil como puede ser un *smartphone* o una *tablet*.

También es posible utilizar otro tipo de información diferente como entrada, como por ejemplo el estado emocional de los usuarios. De esta forma, si la magnitud del gesto detectado aumenta, el sistema pueda interpretar que el usuario se encuentra entusiasmado y responder acordemente. De forma contraria, si el sistema detecta que los gestos del usuario se tornan menos precisos, el sistema puede interpretar que el usuario se siente cansado y podría sugerirle un descanso. Finalmente, podría hacer uso de la posición del usuario o determinar si se encuentra sentado o parado para activar o desactivar el reconocimiento de ciertos tipos de gestos específicos.

En resumen, existe una gran variedad de técnicas de interacción natural, entre las que se destacan la interacción multitáctil, la interacción basada en gestos y la interacción basada en comandos por voz. Por más detalles sobre la evolución de estos paradigmas y los dispositivos de interacción referirse al *Anexo E.* Adicionalmente, en el *Anexo F* se incluye una investigación sobre los principios de diseño particulares para los sensores utilizados como parte de este proyecto, *Microsoft Kinect* y *Leap Motion,* detallando principios y buenas prácticas para el diseño de interacción con cada uno de ellos. En las subsecciones siguientes se detallan los aspectos más relevantes y algunas buenas prácticas de diseño para los tipos de interacción multitáctil e interacción basada en gestos, ya que son las técnicas de interacción principales utilizadas para llevar a cabo el presente proyecto.

3.4.1 Interacción multitáctil

Una superficie multitáctil es aquella que posee capacidad de cómputo propia para detectar cuándo alguien está interactuando con ella como son las comúnmente denominadas *touch-screens*. Si bien este tipo de superficies tiene una gran precisión en cuanto a la detección del área con la que se está interactuando, así como también la detección del momento exacto en el que ocurre la interacción, están construidas de un material especial y esto por lo general hace que sean costosas. A su vez, si se considera una superficie de gran porte como puede ser una pizarra electrónica o una mesa, la movilidad entre diferentes entornos es muy limitada, por lo que es también una solución poco portable. Algunos ejemplos de este tipo de superficies son la *FlatFrog Multitouch 3200* [28], *Figura 27*, la *Mega touchscreen de la*

Universidad de Groningen [123], *Figura 28*, o la *Samsung SUR40* [134], *Figura 29*.

Figura 27: *FlatFrog Multitouch* 3200 [29]

Figura 28: *Mega touchscreen University of Groningen* [124]

Figura 29: *Samsung* SUR40 [134]

Una alternativa para interactuar con superficies que no cuentan con capacidad de cómputo propia es mediante el uso de cámaras de profundidad que brinden la posibilidad de obtener el mapa de profundidad de la escena, como puede ser la provista por el sensor *Microsoft Kinec*t. Así, observando las diferencias entre un mapa de profundidad inicial tomado como base (captado previamente de forma tal que no sean visibles elementos de usuario sino únicamente la superficie de interacción) y un mapa de profundidad en tiempo real al momento de la interacción, se pueden reconocer los dedos y manos del usuario como aquellos píxeles que se encuentren más cercanos de la cámara con respecto a la imagen base, ya que esto indica que son elementos que se agregaron después con respecto al mapa de profundidad inicial. Las cámaras de profundidad reportan la distancia hacia la superficie más cercana de forma tal que la precisión disminuye a medida que la distancia entre la cámara y el objeto aumenta. Es por esto que los límites de resolución de las cámaras de profundidad, así como también la escasa visibilidad en caso de que exista algo que se interponga entre la cámara y la superficie, hacen que la precisión en la detección del instante en que ocurre la interacción no sea tan buena como aquellas implementaciones que usan superficies con capacidad de cómputo propia.

Por otro lado, como se detalla en los estudios sobre performance en superficies táctiles de *Microsoft Research* [40], el tiempo que pasa entre que se toma contacto con la superficie y ésta reacciona de alguna forma es denominado latencia. Generalmente para superficies táctiles en sí mismas la latencia está entre 1ms y 50ms, valores que permiten una percepción de interacción fluida para el ser humano. Pero en superficies táctiles implementadas por otros mecanismos como el de detección por profundidad, la latencia obtenida usualmente resulta ser más alta, en el entorno de los 100 ms o más, lo cual implica que en algunos casos se perciban efectos de retardo en la interacción. De todos modos, el impacto de estos retardos en la eficiencia de la interacción dependerán fuertemente de la aplicación que se esté utilizando, ya que quizás para selección de menús los retardos no sean perceptibles, pero sí, por ejemplo, para dibujar en un pizarrón. Sin embargo, usar cámaras de profundidad tiene ciertas ventajas destacables, algunas de las cuales son:

1. Superficie: la superficie de interacción no tiene que ser de material especial ni debe ser una

superficie plana, resultando en una solución más económica y mucho más genérica, flexible y portable.

2. Cámaras: mediante las cámaras de profundidad se puede hacer uso de información adicional inherente a la interacción, aprovechándose para enriquecer la aplicación final. Más específicamente, se puede obtener la posición del usuario o información sobre sus brazos y manos, determinando por ejemplo, si múltiples interacciones son del mismo usuario o realizadas con la misma mano. Se puede establecer de esta forma cuál fue el usuario que disparó una acción, y conocer a su vez qué es lo que está sucediendo por encima de la superficie.

El segundo punto en particular es la característica que más interesa para el presente proyecto, ya que requiere combinar interacciones del tipo multitáctil sobre superficies con interacciones por encima de ella, posiblemente basadas en gestos. Es por esto que la alternativa de utilizar el sensor *Microsoft Kinect* como dispositivo principal para implementar un sistema multitáctil que permita interactuar con una superficie mediante el tacto es la opción escogida. En el *Capítulo 4* se describen los detalles de la solución desarrollada con el objetivo de detectar interacciones multitáctiles mediante el uso de una cámara de profundidad.

3.4.2. Interacción gestual 2D y 3D

La interacción basada en gestualidad multitáctil, también denominada gestualidad 2D, ha revolucionado la forma de interacción de los usuarios con los dispositivos. Hoy en día, esto ha avanzado de forma exponencial, al igual que la gestualidad tridimensional, también denominada gestualidad 3D o *touch free gesture*, es decir, el reconocimiento de gestos sin necesidad de estar en contacto con una superficie. Estos gestos permiten a las personas relacionarse con los dispositivos de forma más humana e intuitiva. Como se mencionó en secciones anteriores, *Microsoft* con su sensor *Microsoft Kinect* fue uno de los primeros en incursionar en este tipo de interacción abocada principalmente al área de videojuegos. De todos modos, mediante por ejemplo el surgimiento del sensor *Kinect for Windows* y de televisores inteligentes hoy en día se está migrando este tipo de interacción más allá de los videojuegos [145].

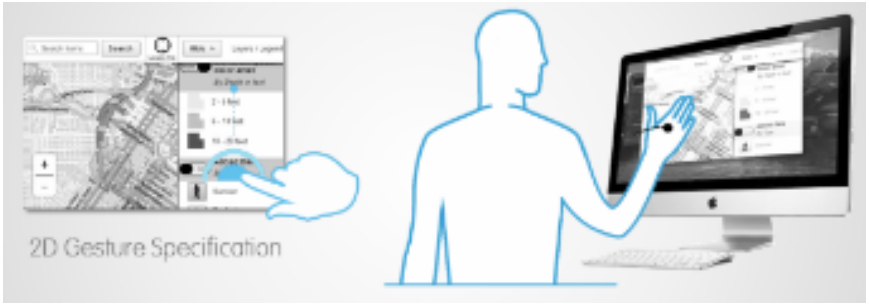


Figura 30: Gesto multitáctil (izq.), gesto espacial (der.) [93]

De esta manera, es posible interactuar con elementos digitales de forma semejante a la interacción realizada con elementos físicos del mundo real, ya que no se necesita de un elemento externo para lograr la interacción por parte del usuario. A su vez, la interacción gestual se vuelve muy útil para los casos en que acciones del tipo multitáctiles se vuelven impracticables. Adicionalmente al uso de gestos en tres dimensiones la interacción por comandos por voz permite también conformar interfaces muy ricas, de forma que, por ejemplo, la mano del usuario puede controlar cierto aspecto de la aplicación mientras que su voz controle otro.

A medida que estos tipos de interacciones fueron creciendo, aparecieron bibliotecas que definen patrones y ayudan a los diseñadores a definir buenas interfaces. Para el caso de la interacción multitáctil, un ejemplo de estos patrones es la guía realizada por *Lukew* [78], presentada en la *Figura 31,* donde se

detalla las interacciones multitáctiles más comúnmente utilizadas.

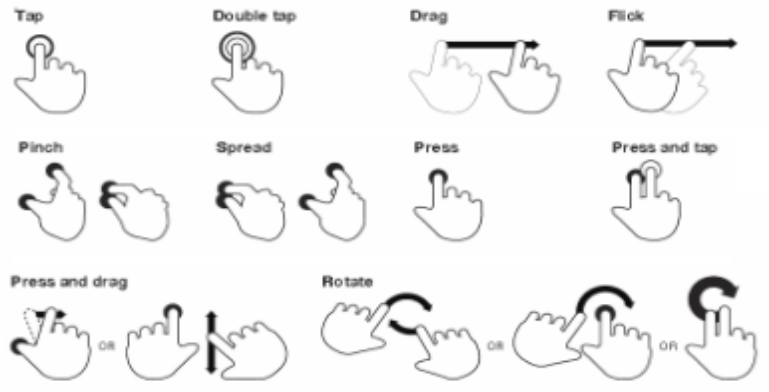


Figura 31: Interacciones multitáctiles más comúnmente utilizadas [78]

Por otro lado, algunos patrones para gestos en tres dimensiones comúnmente utilizados son los proporcionados por *Think Moto* [146], ilustrados en la *Figura 32*.

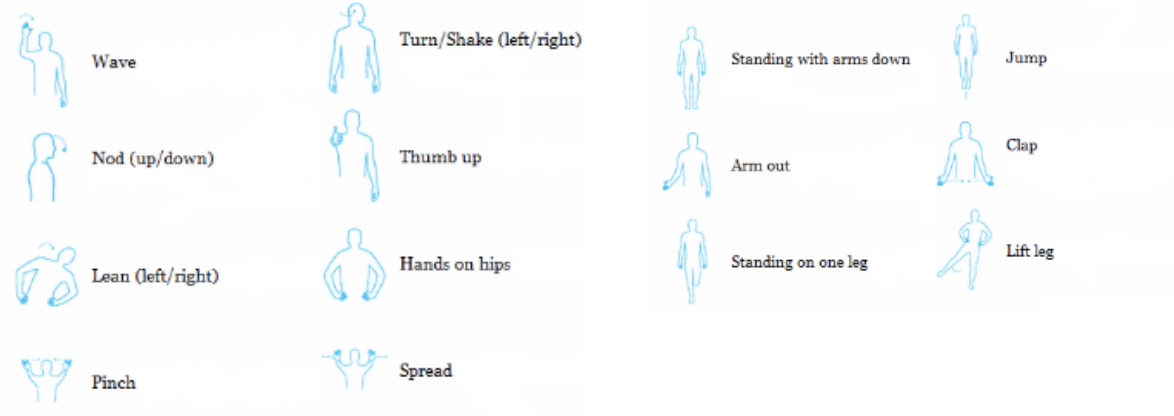


Figura 32: Interacciones en tres dimensiones [146]

Como parte del presente proyecto se definió un patrón de gestos propio, conformando así un banco de gestos consistente, que incluye tanto gestos multitáctiles como tridimensionales, así como también combinaciones de éstos para obtener gestos híbridos. En el *Anexo H* se pueden encontrar más detalles sobre el banco de gestos definido inicialmente, aunque es preciso mencionar que en la solución final desarrollada no se incluyeron la totalidad de los gestos definidos, sino sólo un subconjunto de ellos.

3.5. Soluciones existentes

Actualmente existe una gran cantidad de proyectos y trabajos de investigación que caen dentro de alguna o varias de las áreas previamente descritas en este capítulo. En esta sección se presentan algunos de ellos a modo de definir el estado del arte actual para las áreas de interés. En particular, se detallan aquellos que están más relacionados con el presente proyecto en cuanto al uso de una superficie física de interacción, y en la medida de lo posible, a la integración de las técnicas de interacción natural y multitáctil. profundizando en el *Anexo G* otros proyectos estudiados. De todos modos, cabe mencionar que hay múltiples trabajos de investigación que, aún cayendo dentro de las mismas áreas de interés, no están fuertemente ligados a este proyecto en cuanto a los objetivos y características generales.

3.5.1. *LightSpace*

LightSpace, Combining Multiple Depth Cameras and Projectors for Interactions On, Above, and Between Surfaces [74] es un proyecto elaborado por *Microsoft Research* en el año 2010. Consiste en una sala con varias cámaras de profundidad y proyectores a modo de experimentar las diferentes formas de interacción que se pueden establecer. La sala posee dimensiones de 3x2.5x2.5 metros de alto, ancho y largo respectivamente, y los proyectores y cámaras están organizados en conjunto como muestra la *Figura 33*, suspendidos a aproximadamente 3 metros del suelo. La ubicación de estos sensores garantiza que se cubra toda el área de la habitación, así como también que se superpongan las áreas que cada uno de ellos captura con el fin de reducir errores mediante redundancia de información. Tanto las cámaras como los proyectores están calibrados utilizando coordenadas de mundo y se requiere que ambos sean calibrados inicialmente mediante un proceso de calibración.

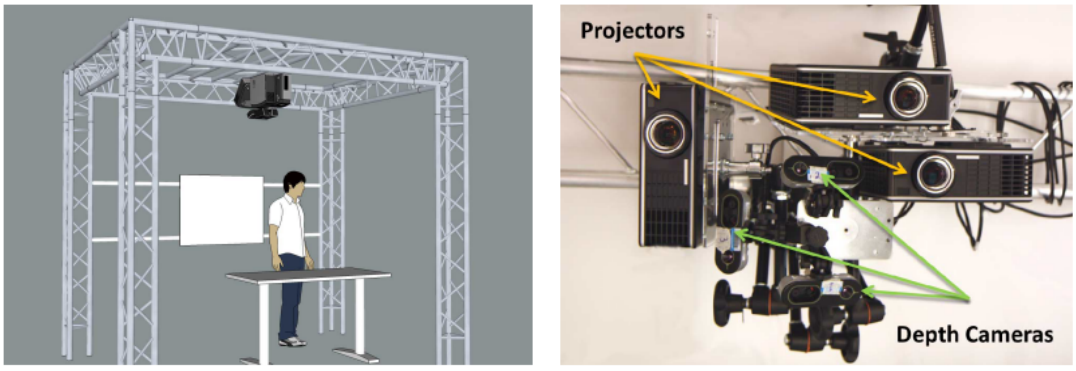


Figura 33: Representación de la sala de *LightSpace* [74]

Mediante los tres proyectores y las tres cámaras de profundidad, *LightSpace* permite transformar cualquier superficie en una superficie interactiva, brindando capacidad multitáctil y posibilitando la interacción gestual en el aire. Así, por ejemplo, un usuario podría tomar un objeto proyectado en una superficie y trasladarlo en su mano mientras camina para luego depositarlo en otra superficie. De esta forma, se rellena al vacío existente entre las diferentes superficies y se puede ver a todas ellas como conectadas, en el sentido de que es posible intercambiar elementos entre cualquiera de ellas. El uso combinado de varios proyectores y cámaras permite que se puedan proyectar elementos virtuales incluso sobre superficies que se mueven, como es el caso del usuario transportando un elemento en su mano. Adicionalmente, *LightSpace* permite transformar el tamaño de los objetos de modo que puedan ser proyectados en cualquier superficie. Más específicamente, la mano del usuario posee un área pequeña, por lo que al momento de la interacción *LightSpace* transforma el objeto en cuestión en una pequeña bola de color que representa al objeto, tal como ilustra la *Figura 34.*

Figura 34: Usuario transportando objeto seleccionado con su mano [74]

Adicionalmente, *LightSpace* permite conectar superficies mediante la selección de cada una de ellas. Para esto, el usuario puede seleccionar un objeto proyectado en una superficie y con la otra mano seleccionar otra superficie diferente, haciendo que el objeto se mueva de la superficie origen a la superficie destino. Estas superficies pueden ser dos superficies horizontales como mesas, superficies verticales como

paredes, o cualquier combinación de las anteriores. En la *Figura 35* se muestra un usuario transfiriendo un elemento desde una mesa hacia la pared con el objetivo de visualizarlo a una escala mayor.

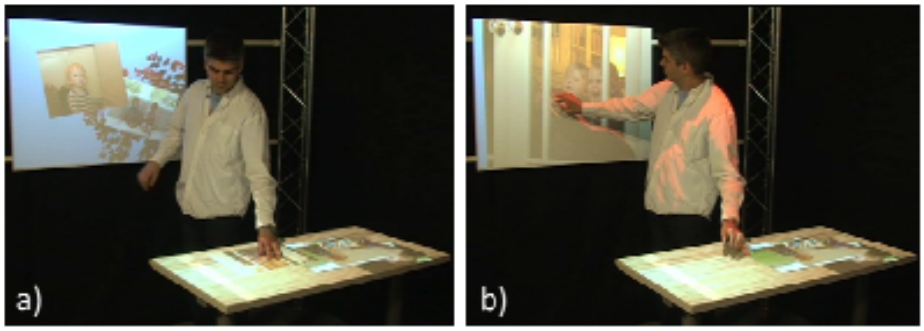


Figura 35: Usuario selecciona el objeto a transferir (izq.), usuario selecciona el destino del objeto (der.) [74]

Otras de las posibilidades que brinda *LightSpace* como forma de interacción son los menúes. Para cada menú existente se proyecta una marca en el piso. Cuando el usuario coloca su mano sobre la marca se puede interactuar con los elementos del menú moviendo la mano hacia arriba o hacia abajo, lo que hace que se desplieguen las diferentes opciones tal como se observa en la *Figura 36.*

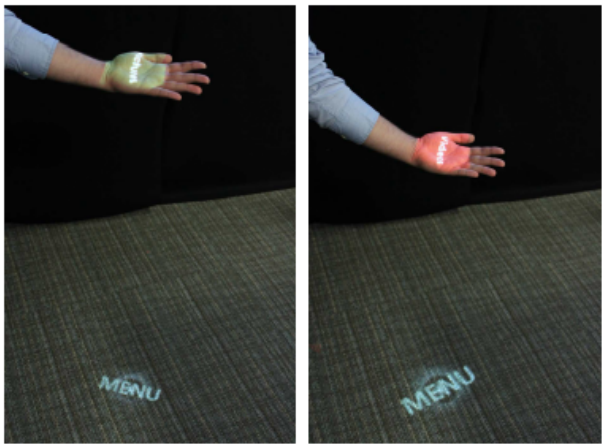


Figura 36: Interacción con menús [74]

En comparación con el presente proyecto, *LightSpace* comparte de cierta forma la organización de la estructura y los sensores que dan soporte a la solución, además del objetivo de contar con una o varias superficies interactivas de bajo costo con las que se pueda interactuar tanto mediante interacción gestual como multitáctil. Para esto, ambos proyectos pretenden crear un espacio que combine múltiples cámaras de profundidad, en particular sensores *Microsoft Kinect*, y al menos un proyector. En contraste a *LightSpace*, este trabajo no pretende incluir el traslado de objetos entre superficies ni la comunicación entre superficies de interacción.

3.5.2. *Low-Cost Efficient Interactive Whiteboard*

Low-Cost Efficient Interactive Whiteboard es un sistema multitáctil de bajo costo basado en cámaras de profundidad [75] desarrollado por el Grupo de Tratamiento de Imágenes de la Universidad Politécnica de Madrid en el año 2012. En particular, utiliza el sensor *Microsoft Kinect* como sensor de profundidad y el *framework OpenNI* para la obtención de datos desde el sensor. Como se puede apreciar en la *Figura 37*, el sistema consiste de una unidad de procesamiento, una superficie sobre la que se interactúa y una cámara de profundidad. La superficie puede ser una pantalla convencional conectada a la unidad de procesamiento, o bien cualquier otro tipo de superficie genérica, en cuyo caso se hace uso de un proyector

para mostrar la información sobre ella.

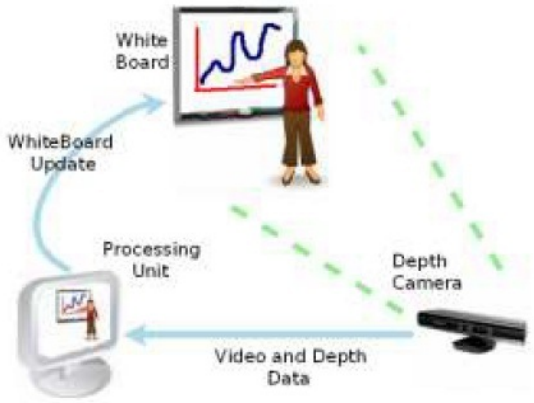


Figura 37: Arquitectura de sistema *Low-Cost Efficient Interactive Whiteboard* [75]

La solución brindada en este proyecto resulta particularmente interesante ya que para paliar la inferioridad de precisión respecto a otras alternativas se combinan dos técnicas de reconocimiento independientes, por un lado el reconocimiento en base a información de profundidad y por el otro el reconocimiento en base a información de vídeo o *RGB*. Para procesar tanto los datos de profundidad como los datos de vídeo se utilizan algoritmos de visión por computadora. En particular, para el procesamiento de los datos de vídeo se utiliza un algoritmo de seguimiento de características, del inglés *feature tracking*, denominado *Kanade–Lucas–Tomasi (KLT)* [76]. La combinación de ambas fuentes de datos se realiza de acuerdo a la posición relativa del usuario en la superficie. Si el usuario está lejos, se utiliza el mecanismo por profundidad, de lo contrario, el mecanismo por información de vídeo es más apropiado. A su vez, mientras que la información de la cámara de profundidad es utilizada para la detección de los gestos de usuario, la información de vídeo es oportuna para refinar los datos obtenidos. Como se muestra en la *Figura 38*, a grandes rasgos el funcionamiento del sistema consta de los siguiente pasos:

1. Se inicializa la posición de la pizarra en lo que respecta a información de profundidad y vídeo:
   a. Con el algoritmo *Gaussians Background Modeling* [10] se obtiene una estimación confiable del mapa de color y profundidad de la pizarra.
   b. Se reconoce el esqueleto entero del usuario mediante el *framework OpenNI*.
2. Se obtiene la información de profundidad y vídeo según corresponda para el correcto seguimiento de la mano del usuario.
3. De ser necesario se refinan o complementan los datos con el algoritmo de vídeo inicializado con los datos extraídos del mapa de profundidad.
4. Se abstrae el gesto y se procesa.
5. Se actualiza la pantalla.

El presente proyecto se correlaciona con *Low-Cost Efficient Interactive Whiteboard* sobre todo en la utilización del sensor *Microsoft Kinect* para el seguimiento esqueletal de los usuarios, el reconocimiento de gestos y la generación del mapa de profundidad de la escena. A su vez, coincide en la utilización del *framework OpenNI* para el análisis de los datos obtenidos por el sensor *Microsoft Kinect*. Por otro lado, al igual que *Low-Cost Efficient Interactive Whiteboard* este trabajo pretende utilizar una superficie genérica para la interacción multitáctil y el uso de un proyector para mostrar la información sobre ella, obteniendo una solución de bajo costo. Sin embargo, el hecho de paliar la inferioridad de precisión con la utilización de algoritmos más sofiticados de visión por computadora queda por fuera del alcance del presente proyecto.

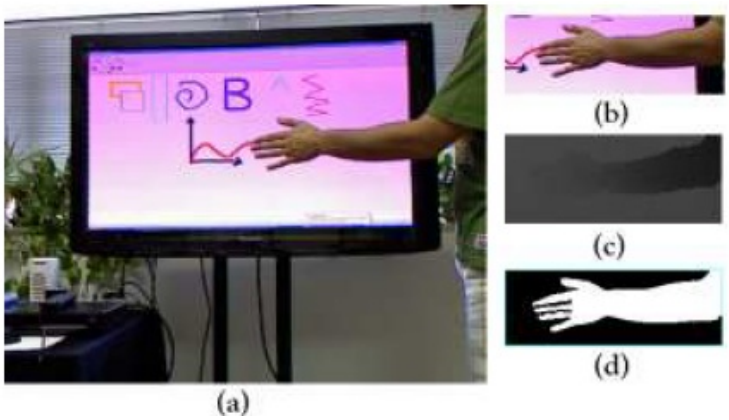


Figura 38: Modo de uso (izq.), detalle de información de video y profundidad (der.) [75]

3.5.3. *MisTable*

MisTable: Reach-through Personal Screens for Tabletops [85] es un proyecto desarrollado por estudiantes del grupo *Bristol Interaction and Graphics* de la Universidad de Bristol durante el año 2014. Este proyecto tiene como objetivo implementar un espacio colaborativo con el que se pueda interactuar, ya sea de forma multitáctil o gestual, permitiendo el trabajo colaborativo entre varios usuarios de forma simultánea. El sistema está construido entorno a una mesa plana horizontal, en la que se pueden proyectar imágenes y manipularlas. Adicionalmente, cuenta con pantallas personales de niebla en las que se proyecta información visual adicional con la que también se puede interactuar mediante el seguimiento de las manos del usuario gracias a un sistema basado en el sensor *Leap Motion*. La capa de niebla brinda al usuario una capa adicional de información visual, permitiendo personalizar la vista de cada uno de los usuarios en función de su identidad o preferencias, pero que puede también ser alcanzada y manipulada desde la superficie sólida. Estas pantallas personales se definen tanto como *see-through*, es decir que permiten visualizar los elementos que en ella se presentan así como también los que están por detrás en la superficie, como *reach-through,* es decir, que el usuario puede interactuar tanto con la pantalla personal como con la superficie sólida o el espacio sobre ella. En la *Figura 39* se puede observar las características antes descritas. En particular, se observan las diferentes superficies de presentación de contenido; por un lado la superficie sólida horizontal y por otro la superficie de niebla vertical.

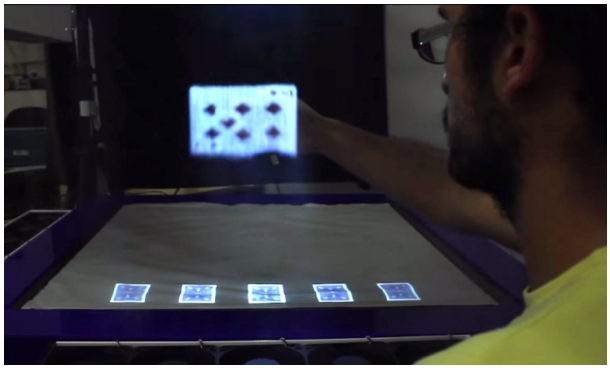


Figura 39: Manipulación directa sobre capa de niebla [85]

El sistema resultante extiende las características de los sistemas comunes basados en superficies de interacción, contando no con uno sino con tres espacios de interacción, cada uno con diferente dimensión e implicaciones colaborativas. La superficie sólida se convierte en un espacio compartido de interacción en dos dimensiones. Las pantallas personales mantienen información en dos dimensiones personal de cada usuario. Finalmente, el espacio por encima de la superficie permite interacción compartida en tres dimensiones. A su vez, cualquiera de los diferentes espacios de interacción están a la vista y al

alcance de los usuarios, pudiendo mover información de un espacio a otro. Así, por ejemplo, si la mano del usuario está aproximadamente dentro del plano de la niebla, todos los movimientos de los dedos son interpretados como interacciones con la pantalla personal. Si el usuario mueve su mano rápidamente desde la pantalla personal hacia la superficie sólida, el contenido seleccionado es trasladado a dicha superficie, quedando disponible para los demás usuarios tal como ilustra la *Figura 40*. De forma opuesta, se pueden mover elementos desde la superficie sólida hacia la pantalla personal.

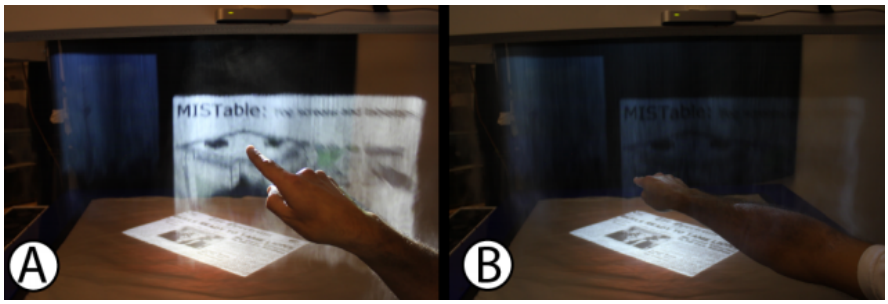


Figura 40: Interacción sobre pantalla de niebla (izq.), elemento enviado desde pantalla de niebla a superficie sólida (der.) [86]

Para proveer la interactividad, el proyecto se apoya en varios sistemas de seguimiento individuales. Se utiliza un sensor *Microsoft Kinect* colocado sobre la superficie sólida, permitiendo llevar el seguimiento de las manos de los usuarios y posibilitando el reconocimiento de gestos realizados sobre la superficie. Adicionalmente, un segundo sensor *Microsoft Kinect* es utilizado por encima de toda la estructura para realizar el seguimiento de la cabeza de los usuarios y mostrar imágenes con la perspectiva correcta para cada uno. Por otra parte, debido a que el sensor *Microsoft Kinect* no logra detectar de forma precisa las interacciones sobre las pantallas personales de niebla, se utilizan sensores *Leap Motion* ubicados en la parte superior de cada pantalla de niebla con el objetivo de hacer el correcto seguimiento de las manos y dedos de cada usuario en su pantalla personal.

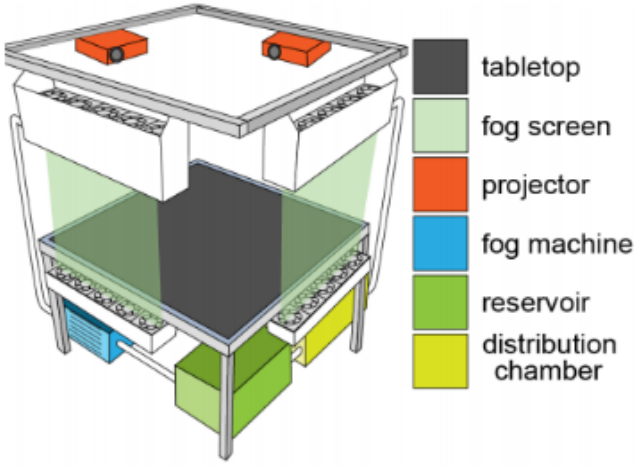


Figura 41: Elementos principales que conforman *MISTable* [86]

El sistema de distribución de niebla está conformado por una máquina de generación de niebla, un depósito y una cámara de distribución de niebla. Este sistema hace llegar la niebla a la parte superior de las pantallas personales, donde se encuentra una serie de ventiladores que hacen que ésta se mueva hacia abajo. La velocidad de estos ventiladores debe ser lo suficientemente pequeña como para que la niebla permanezca en forma de lámina y lo suficientemente rápida como para que se cubra todo el alto de la pantalla. Para asegurar esto, se colocan extractores por debajo de cada pantalla, quienes capturan la niebla enviada desde arriba y estabilizan la imagen desde abajo. Con el fin de mantener un flujo de niebla en forma de lámina, se incluye una capa de niebla de 15 mm de espesor, envuelta por dos capas de aire más finas de 70 mm de espesor. Esto permite proteger la capa intermedia de corrientes de aire externas, y en

consecuencia, mejora la calidad de los elementos mostrados en ella.

Finalmente, con el fin de alivianar el procesamiento necesario para hacer funcionar correctamente al sistema, éste se encuentra distribuido en dos equipos diferentes. Uno de ellos contiene uno de los sensores *Microsoft Kinect* y uno de los sensores *Leap Motion* (utilizados para uno de los usuarios), así como también todos los componentes necesarios para la distribución de niebla. Por otra parte, al segundo equipo se le conecta el otro sensor *Microsoft Kinect* y el otro sensor *Leap Motion* (utilizados para el otro usuario). Ambos equipos se comunican mediante una red de área local, enviando mensajes de seguimiento de la cabeza y dedos de cada usuario mediante el protocolo UDP.

MisTable es uno de los trabajos que más se asemeja al presente proyecto en el sentido de que pretende integrar en una única solución sensores *Microsoft Kinect* y *Leap Motion*. A su vez, establece una distribución arquitectónica similar, realizando la comunicación de los datos obtenidos por los distintos sensores y computadores mediante una red de área local. Finalmente, también brinda flexibilidad al permitir la interacción sobre cualquier tipo de superficie e implementa el reconocimiento de gestos tanto sobre la superficie (espaciales) como en la superficie (multitáctiles). Sin embargo, el presente proyecto no incluye superficies de interacción en niebla ni ningún tipo de interacción sobre éstas.

3.6. Resumen del capítulo

El presente proyecto se encuentra vinculado a un conjunto de áreas como son la interacción persona-computadora, computación ubicua, realidad aumentada, interacción natural, interacción multitáctil e interacción gestual espacial. En este capítulo se describe su relacionamiento, destacando que se deben tomar en cuenta los mecanismos necesarios para desarrollar aplicaciones considerando buenas prácticas conocidas, estudiadas y adoptadas para cada una de dichas áreas. Más específicamente, posibilitar los mecanismos para una interacción centrada en el usuario, lo más parecida posible a la que los seres humanos están acostumbrados, sin que sea necesario que piensen en cómo funciona sino que puedan centrarse en cumplir las tareas requeridas. Por otro lado, se presenta un estudio del estado del arte en cuanto a trabajos de investigación actualmente existentes que comparten ciertas características con el presente proyecto, ya sea en lo que respecta a los mecanismos de interacción incluídos o a las estructuras y dispositivos necesarios para dar soporte a la solución. En cada caso se analiza cómo se resolvieron los diferentes problemas presentados y cómo se amoldan a cada una de las áreas de interés.

Capítulo 4: Problemas a resolver

En este capítulo se presentan en detalle los problemas concretos que se debieron resolver como parte del presente trabajo, ya mencionados en el *Capítulo 1*, describiendo las soluciones encontradas para cada uno de ellos. La resolución a las distintas problemáticas puntuales que implicaron la toma de múltiples decisiones de diferente índole a lo largo del proyecto, y en conjunto conforman su resolución global, o al menos de gran parte de los desafíos que éste supuso. Por otra parte, en el *Anexo L* se describen algunas de las pruebas de concepto realizadas con el objetivo de mitigar ciertos riesgos referentes al uso de las diferentes tecnologías involucradas descritas en el *Capítulo 2*.

4.1. Integración de múltiples sensores

La integración de un sensor *Leap Motion* y de un sensor *Microsoft Kinect* en lo que respecta a poder obtener datos provenientes de ambos dentro de la misma solución es sencilla, ya que por medio del control de cada uno mediante la biblioteca correspondiente ambos conviven sin mayores problemas. La complejidad mayor yace en convertir los datos provistos por cada sensor a un mismo sistema de coordenadas. Una de las pruebas de concepto realizadas inicialmente permitió mitigar este riesgo en cierta medida, reduciendo el problema a encontrar la matriz de transformación adecuada que permita realizar dicha conversión, tal como se detalla en la *Sección 4.2*.

La solución final es un tanto más compleja que el caso planteado anteriormente que involucra un sensor de cada tipo, ya que cuenta con cuatro sensores en total; dos sensores *Microsoft Kinect* y dos sensores *Leap Motion*. Cada uno de los sensores *Leap Motion* son de uso exclusivo para los usuarios, mientras que uno de los sensores *Microsoft Kinect* es utilizado para captar la escena donde se encuentran inmersos todos los usuarios y el restante es utilizado para la resolución de la interacción multitáctil sobre la superficie. La configuración escogida para la distribución de los sensores en la conformación de la solución final consiste de un sensor *Microsoft Kinect* y un sensor *Leap Motion* por computador, por lo que se debe contar con dos computadores como mínimo, conectando un sensor de cada tipo de los antes mencionados a cada uno de ellos, tal como ilustra la *Figura 42*. Esta distribución fue escogida principalmente por dos razones. Por un lado, debido a la restricción inherente del sensor *Leap Motion* en lo que refiere a la imposibilidad de que dos sensores operen en un mismo computador [88]. Por otro lado, aprovechando el hecho de tener que utilizar dos computadores como mínimo debido a la restricción anterior, se distribuye también un sensor *Microsoft Kinect* por computador con el objetivo de distribuir la carga de procesamiento. Adicionalmente, como se detalla en la *Sección 4.8*, la arquitectura definida es lo suficientemente flexible como para extender la solución con más sensores de interacción.

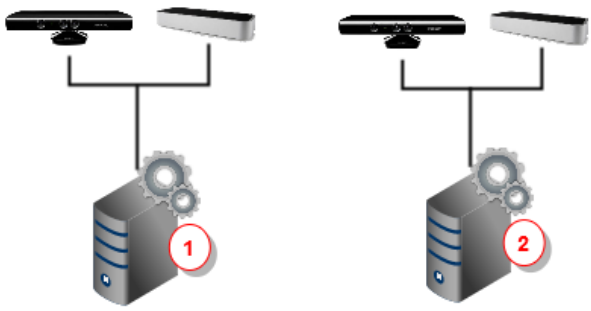


Figura 42: Distribución de sensores y computadores

Si bien la integración de los sensores antes mencionados en sí misma no es compleja, además del

problema referente a la unificación de sistemas de coordenadas, existe un inconveniente adicional referente al ruido que se genera debido al solapamiento de los rayos infrarrojos proyectados por los diferentes sensores, problema que se detalla en la *Sección 4.6* de este mismo capítulo.

4.2. Múltiples sistemas de coordenadas

Resulta vital que se pueda llevar a un mismo sistema de referencia lo que cada uno de los diferentes sensores capta, a modo de enriquecer y complementar la información recabada por cada uno de ellos. De esta forma, por ejemplo, se puede detectar la presencia de un usuario en la escena mediante el sensor *Microsoft Kinect*, identificando sus distintas articulaciones o *joints*, y por otro lado, mediante el sensor *Leap Motion* detectar el movimiento de sus dedos. Llevando esta información a un único sistema de referencia se pueden asociar los dedos detectados por el sensor *Leap Motion* a un cierto usuario detectado por el sensor *Microsoft Kinect*, logrando obtener mayor cantidad de información que la proporcionada de forma aislada por cada uno de los sensores.

Con este objetivo, se realizaron algunas pruebas de concepto, descritas con más detalle en el *Anexo L*, consistentes en unificar los sistemas de coordenadas de un sensor *Leap Motion* y un sensor *Microsoft Kinect*. Como se detalló en el *Capítulo 2*, ambos sensores utilizan un sistema de referencia cartesiano de mano derecha, en el que observando el sensor de frente, el eje X positivo apunta a la derecha, el eje Y positivo hacia arriba y el eje Z positivo en dirección al usuario. Según los resultados de estas pruebas, el problema se reduce a encontrar la transformación correcta que permita convertir los datos desde los sistemas de coordenadas de cada sensor particular al sistema de coordenadas global. Esta transformación puede ser especificada mediante el producto de las matrices que representan las transformaciones de rotación y traslación para cada sensor. Cabe observar que no es necesario realizar una transformación de escala dado que todos los sensores reportan sus valores en la misma unidad de medida (milímetros). De esta forma, para cada uno de los sensores se debe especificar las traslaciones y rotaciones necesarias para convertir el sistema de coordenadas local en el sistema global de coordenadas.

En la solución desarrollada se define que el sistema de coordenadas global coincide con el sistema de coordenadas local del sensor *Microsoft Kinect* que apunta directamente hacia la superficie, es decir, el responsable de brindar la información con la que se resuelve la interacción multitáctil. En la *Figura 43* se observa cada uno de los sistemas de coordenadas con los que se debe lidiar para la disposición de sensores utilizada. De esta forma, se deben aplicar las transformaciones correctas a los datos provistos por cada uno de los sensores para convertir su sistema de coordenadas local al sistema de coordenadas global definido. Así, las transformaciones necesarias para cada sensor son las siguientes:

1. Los valores reportados por el sensor *Microsoft Kinect K1* de la *Figura 43* son el caso más trivial, ya que no deben sufrir alteraciones debido a que el sistema de coordenadas local coincide con el sistema de coordenadas global definido.
2. En lo que respecta al sistema de coordenadas del sensor *Microsoft Kinect K2* (llámese *SCK2*) de la *Figura 43*, dado un punto expresado en coordenadas de *SCK2* se debe encontrar la transformación que permita obtener la posición de dicho punto en relación al sistema de coordenadas global. Esto se logra mediante la composición de dos transformaciones, que permiten hacer coincidir el sistema de coordenadas global con el sistema de coordenadas local *SCK2*. De esta forma, al aplicarse este conjunto de transformaciones a un punto genérico expresado en coordenadas locales de *SCK2* se obtienen las coordenadas del punto relativo al sistema de coordenadas global. Las transformaciones necesarias en este caso son una traslación en los ejes Y- y Z-, y una rotación sobre el eje X-[7].

[7] Se denota X+, Y+ y Z+ a los ejes positivos de un sistema de coordenadas cartesiano. Análogamente, X-, Y- y Z-

3. Por último, en lo que respecta a los sistemas de coordenadas de los sensores *Leap Motion L1 y L2* (llámense *SCL1* y *SCL2* respectivamente) de la *Figura 43*, también se requiere de dos transformaciones para obtener los valores de un punto genérico en el sistema de coordenadas global. Primero se debe realizar una traslación en los ejes Z+, Y+ y X (X- o X+ dependiendo de si se trata de *SCL1* o *SCL2*, respectivamente), y finalmente una rotación sobre el eje X-.

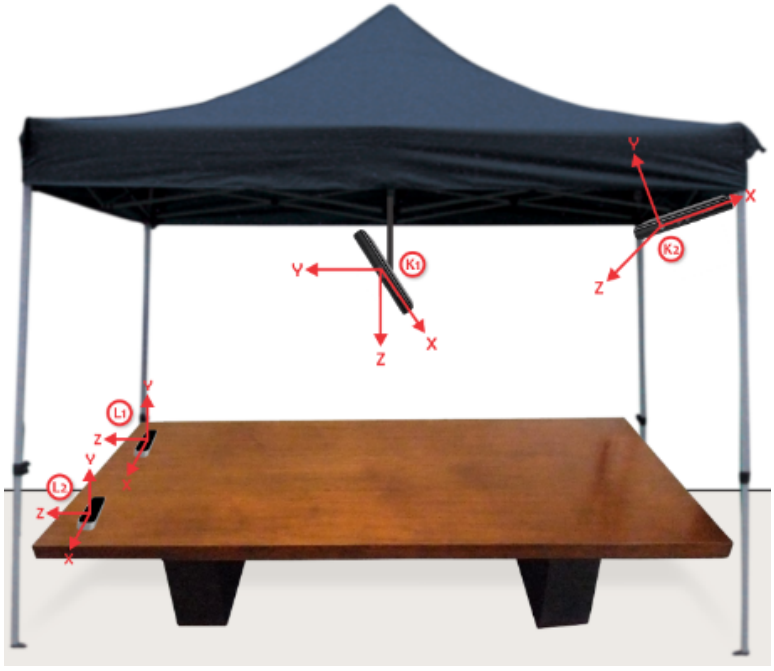


Figura 43: Sistema de coordenadas locales de los diferentes sensores

Como se mencionó anteriormente, las transformaciones son representadas por el producto de matrices, por lo que la transformación global va a estar dada por la multiplicación de las matrices de transformación de rotación y traslación correspondientes. En particular, para la solución propuesta, la transformación del sistema de coordenadas de cada sensor queda definida por la siguiente ecuación:

$$T\_Final = T\_Rotation * T\_Traslation$$

La matriz de traslación incluye las traslaciones en los tres ejes del sistema de coordenadas según corresponda. Por otra parte, en cuanto a la rotación, teniendo en cuenta que el signo de giro de las rotaciones está dado por la regla de la mano derecha[8], la transformación de rotación queda definida por la siguiente ecuación, en la cual primero se aplica la rotación en el eje Z, luego en el eje Y y por último en el eje X (importante considerar dada la característica no conmutativa de las rotaciones):

$$T\_Rotation = (rotationX * (rotationY * rotationZ))$$

De esta forma, para cada uno de los sensores se debe especificar la traslación en cada uno de sus ejes, teniendo en cuenta en este caso la magnitud a trasladar y el signo, ya que dependiendo de esto se trasladará en el mismo sentido del eje o hacia el lado contrario, así como también, el ángulo de rotación

se corresponden con los ejes negativos del sistema de coordenadas. Así, por ejemplo, una rotación en el eje X+ refiere a una rotación de cierto ángulo positivo entorno al eje X positivo (según la regla de la mano derecha)

[8] La Regla de la mano derecha establece que se debe extender el pulgar de la mano derecha en la dirección positiva del eje de rotación y luego curvar los dedos restantes. La dirección de estos últimos define la dirección positiva de la rotación.

para cada uno de sus ejes, teniendo en cuenta el ángulo a rotar y el sentido del giro según la regla de la mano derecha. Como se detalla en el *Capítulo 5*, cada una de las transformaciones para los diferentes sensores se define en un fichero de configuración de los diferentes parámetros del sistema. Cabe mencionar además, que para realizar todas las operaciones sobre matrices que permiten generar correctamente las transformaciones necesarias, se utiliza la biblioteca de álgebra lineal *Eigen* ya descrita en el *Capítulo 2*.

Las coordenadas globales obtenidas para los sensores *Microsoft Kinect* y *Leap Motion* correspondientes a la interacción gestual deben ser luego convertidas al sistema de coordenadas de escena, con el objetivo de mapear correctamente los gestos realizados con los diferentes elementos presentes en la escena para poder corresponder un gesto 3D con los diferentes elementos de la aplicación. Esto es realizado mediante una transformación adicional que permite escalar el rango del sistema de coordenadas global según un rango predefinido de la escena en la aplicación que está ejecutando sobre el *framework*. Por esto, es preciso tener en cuenta que un pequeño cambio en la posición de los diferentes sensores (lo cual resultó ser muy común dadas las características de la estructura en la que se encuentran instalados) puede repercutir negativamente en el reconocimiento de los usuarios en la escena y los gestos por ellos realizados, debido a que la transformación entre los diferentes sistemas de coordenadas puede no ser lo suficientemente precisa y generar así resultados inexactos.

Se debe tener en cuenta también que la pantalla sobre la cual se muestra información al usuario es una proyección sobre la propia superficie de interacción, por lo que las coordenadas globales obtenidas para el sensor *Microsoft Kinect* correspondientes a la interacción multitáctil deben ser luego convertidas al sistema de coordenadas de proyección con el objetivo de mapear correctamente las interacciones tangibles sobre los objetos virtuales que están siendo proyectados en la superficie. Esto requiere dos tareas esenciales para su correcto funcionamiento:

1. Definir una transformación que permita convertir las coordenadas de las interacciones multitáctil brindadas por el sensor *Microsoft Kinect*, tal como se detalló anteriormente en esta misma sección, en coordenadas de proyección o pantalla, es decir, un nuevo sistema de coordenadas bidimensional que permita representar el área de la superficie de interacción sobre la que se está proyectando la información, como si se tratase de una pantalla convencional. Este mapeo se realiza mediante un proceso previo de calibración de la proyección mediante el cual se obtiene la transformación que permite realizar la conversión del sistema de coordenadas global al sistema de coordenadas bidimensional de proyección. Con esta información es posible calcular la coordenada del área de proyección con la que se está estableciendo contacto mediante interacción multitáctil.
2. En base al punto de contacto en coordenadas de proyección obtenido, se debe inferir con qué objeto de la escena se pretende interactuar. Para esto se debe definir una transformación adicional que permita convertir una coordenada en el sistema de coordenadas de proyección (área de interacción) a una coordenada en el sistema de coordenadas de escena utilizado por la aplicación particular que está ejecutando sobre el *framework*. Cabe mencionar en este punto que aunque el sistema de coordenadas de escena es un sistema de coordenadas tridimensional, se utilizan solo las dos dimensiones dadas por el sistema de coordenadas de proyección para inferir con qué objeto se está interactuando. Una vez se identifica el objeto de la escena al que se corresponde la interacción, se procede a realizar la acción correspondiente, actualizando eventualmente su posición en el sistema de coordenadas de escena.

Los puntos anteriores se pueden ver ilustrados en la *Figura 44*, en la que se incluyen dos nuevos sistemas de coordenadas con respecto a la *Figura 43* presentada anteriormente. Por un lado, se incluye el sistema de coordenadas bidimensional de proyección o pantalla (P1) descrito en el punto 1. Por otro lado, se incluye el sistema de coordenadas tridimensional de escena (E1) utilizado por la aplicación, descrito en el punto 2.

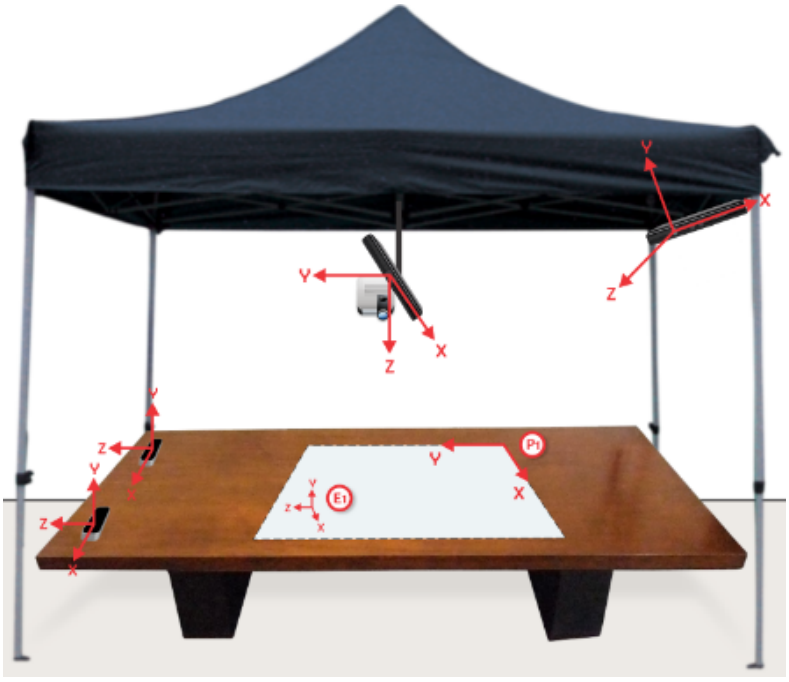


Figura 44: Sistema de coordenadas de proyección y sistema de coordenadas de escena

Para la calibración del Kinect K1 de la *Figura 43* con el proyector inicialmente se intentó utilizar el *addon* de *openframeworks* llamado *ofxReprojection* [101]. Utilizando este *addon* de forma nativa para la calibración no se logró obtener buenos resultados, lo cual era de suponer considerando que la extensión al día de hoy no tiene soporte y está discontinuado desde el año 2013. La herramienta debería permitir reconocer mediante la cámara *RGB* del sensor *Microsoft Kinect* un patrón conocido (por ejemplo, un patrón de tablero de ajedrez). Una vez reconocidos cierta cantidad de puntos de dicho patrón, se debe obtener los puntos correspondientes para el sensor de profundidad del *Microsoft Kinect*, de forma tal que se pueda generar una matriz de transformación que permita mapear cualquier punto sensor de profundidad del *Microsoft Kinect* a un punto en la pantalla. Es esta matriz la que *ofxReprojection* no genera correctamente y por lo tanto al aplicar el mapeo entre sistemas de coordenadas no se retornan los resultados adecuados.

También se intentaron utilizar otras herramientas para este fin, como por ejemplo *ofxProjectorKinectCalibration* [100] y *ofxKinectProjectorToolkit* [99], pero no se logró dejar funcional ninguna de ellas dado que ambas requieren el uso de *OpenNI* 1, cuando la solución propuesta para este trabajo utiliza *OpenNI* en su versión 2. Adicionalmente, se intentó utilizar *ofxCamaraLucida* [98], que requiere la librería *libfreenect* [73] para comunicarse y acceder a la información provista por el sensor *Microsoft Kinect*. Se intentó modificar el componente de acceso a datos de la herramienta para que utilizara los servicios de *OpenNI* en lugar de *libfreenect* para obtener información desde el sensor, pero se encontraron una gran cantidad de problemas de compatibilidad y se decidió descartar también esta opción como una posible alternativa.

Debido a las razones detalladas anteriormente, se desarrolló una herramienta a medida que permitiera realizar el mapeo correctamente, aprovechando las características de reconocimiento del patrón que ya proveé *ofxReprojection*. De esta forma, asumiendo que el sensor *Microsoft Kinect* y el proyector están dispuestos como se muestra en la *Figura 45*, el flujo de la herramienta de calibración desarrollada (denominada *CalibrationTool* y entregada como parte de este proyecto) es el siguiente:

Figura 45: Disposición del sensor *Microsoft Kinect* y el proyector

1. La configuración inicial es leída desde un fichero *XML*. Esta configuración incluye el conjunto de puntos que se desea obtener desde el tablero (pueden ser todos los puntos a reconocer en el tablero o un subconjunto de ellos), el tamaño del tablero y el modo de ejecución de la herramienta de calibración, que puede configurarse para hacer una calibración completa o para ejecutar en tiempo real como forma de testear una calibración previamente generada. En el fichero de configuración también puede indicarse si se desea espejar o invertir los ejes como parte del mapeo, lo cual es útil dependiendo de la posición del sensor *Microsoft Kinect* con respecto al sistema de coordenadas del proyector. La *Figura 46* ilustra un ejemplo de fichero de configuración inicial de la herramienta de calibración.

```
<calibration>
    <stage>3</stage> <!-- 0-COMPLETE_CALIBRATION, 1-ONLY_CAMERA_CALIBRATION, 2-TEST_POINT, 3-REALTIME -->
    <mirror>0</mirror>
    <gridRows>1</gridRows>
    <gridColumns>1</gridColumns>
    <nProjPoints>0</nProjPoints>
    <projPoints>
        <pointList>0,5,18,23</pointList>
    </projPoints>
    <nCamPoints>0</nCamPoints>
    <camPoints></camPoints>
</calibration>
```

Figura 46: Fichero *XML* de entrada a la herramienta de calibración

2. En base a esta configuración, la herramienta de calibración levanta una instancia de la herramienta externa *ofxReprojection* de forma que se pueda detectar el patrón de tablero de ajedrez, reutilizando así algunas de las características de esta herramienta. Una vez se reconoce el patrón, se cargan las coordenadas de pantalla del subconjunto de puntos especificados en el fichero de configuración, reconocidos por la cámara *RGB* del sensor *Microsoft Kinect*. En la *Figura 47* se puede apreciar la pantalla de control de *ofxReprojection* desde donde se puede definir la posición y el tamaño del tablero, visualizar los puntos reconocidos del patrón y el estado del proceso de reconocimiento.

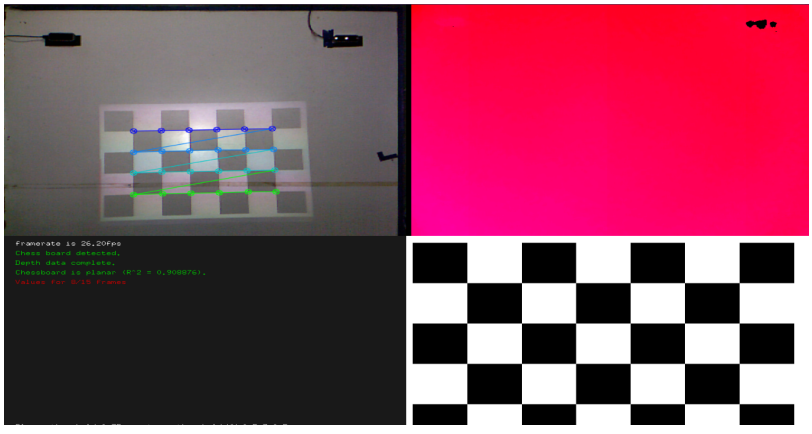


Figura 47: Pantalla de mando de *ofxRperojection*

Por otro lado, en la *Figura 48* se muestra el tablero a reconocer proyectado directamente sobre la superficie, según el tamaño y la posición definidas en la pantalla de mando de la herramienta.

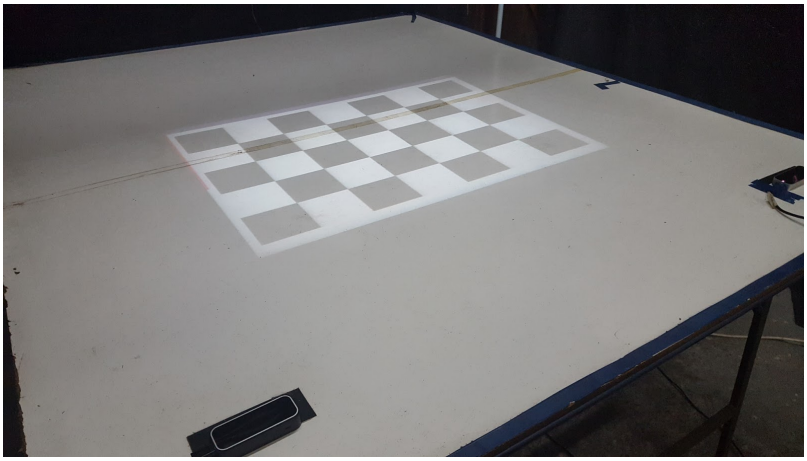


Figura 48: Patrón de tablero de ajedrez proyectado

3. Teniendo los puntos que se van a utilizar para la calibración en coordenadas de pantalla, la herramienta de calibración desarrollada muestra una nueva ventana mediante la cual se pueden obtener los valores en coordenadas de profundidad del sensor *Microsoft Kinect* para cada uno de los puntos en coordenadas de pantalla obtenidos en el punto anterior. La herramienta colorea el punto que se está calibrando actualmente para que se ingrese el mapeo en coordenadas de profundidad del sensor *Microsoft Kinect*, y recorre secuencialmente los diferentes puntos hasta que todos estén correctamente calibrados. El mapeo de calibración se puede ingresar de varias formas dependiendo del módulo de calibración que se esté utilizando. La herramienta soporta dos módulos de calibración. Por un lado, se incluye un módulo de calibración mediante seguimiento de manos que utiliza la posición de una de las manos del usuario para calibrar cada uno de los puntos seleccionados. Esto es útil para realizar una calibración sobre una pared vertical por ejemplo, tal como se puede apreciar en la imagen izquierda de la *Figura 49*. Por otro lado, se incluye un módulo de calibración táctil que utiliza el punto de contacto de un dedo con la superficie para la calibración de cada punto, tal como se muestra en la imagen derecha de la *Figura 49*. Este último módulo reutiliza la lógica del módulo de reconocimiento multitáctil del *framework* propuesto, implementado de la misma forma que se detallará en la *Sección 4.4*. Como se mencionó anteriormente, independientemente del módulo de calibración utilizado, el punto a calibrar cambia automáticamente cada cierto tiempo, finalizando el proceso una vez estén todos los puntos calibrados. La *Figura 49* también permite apreciar parte del proceso de calibración para una grilla formada por cuatro puntos.

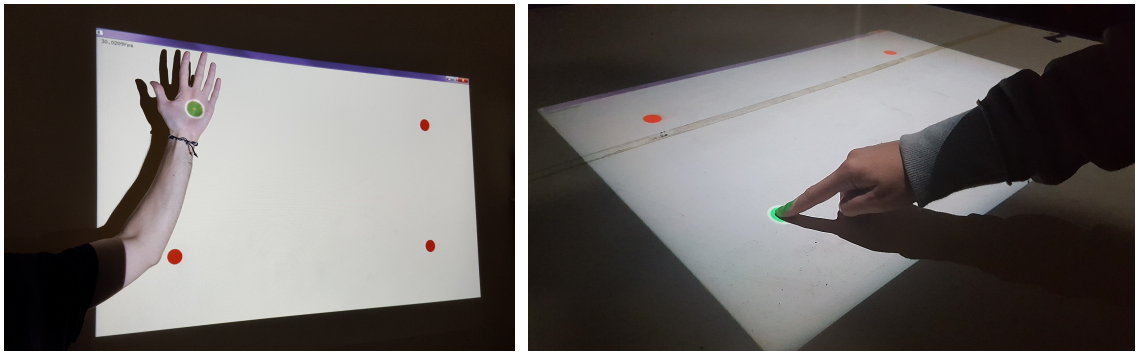


Figura 49: Módulos de calibración incluidos en la herramienta de calibración. Calibración con seguimiento de manos (izq.). Calibración con reconocimiento táctil (der.)

Durante el intervalo de tiempo en el que se está calibrando cada punto particular, se escribe un fichero de *log* con todos los valores obtenidos para dicho punto en coordenadas de profundidad del sensor *Microsoft Kinect*, ya sea utilizado con la calibración basada en seguimiento de manos o con la calibración basada en reconocimiento táctil. Este *log* queda almacenado en determinado directorio como parte de la salida de la herramienta de calibración y deberá ser analizado manualmente para obtener los valores en coordenadas de sensor para cada uno de los puntos de la grilla, obteniendo así el mapeo entre ambos sistemas de coordenadas para los puntos seleccionados. En la *Figura 50* se puede apreciar un ejemplo de *log* de calibración resultante del proceso.

```
---------------------------------------- 2017-04-11-21-32-55-307
[notice ] --- DATOS PARA EL PUNTO 0 ---
---------------------------------------- 2017-04-11-21-32-58-042
[notice ]  (-258,-307,0)
---------------------------------------- 2017-04-11-21-32-58-089
[notice ]  (-258,-307,0)
---------------------------------------- 2017-04-11-21-32-58-120
[notice ]  (-262,-309,0)
---------------------------------------- 2017-04-11-21-32-58-151
[notice ]  (-263,-310,0)
---------------------------------------- 2017-04-11-21-32-58-183
[notice ]  (-263,-306,0)

...

---------------------------------------- 2017-04-11-21-34-53-008
[notice ] --- DATOS PARA EL PUNTO 3 ---
---------------------------------------- 2017-04-11-21-34-53-039
[notice ]  (242,0,0)
---------------------------------------- 2017-04-11-21-34-53-070
[notice ]  (242,0,0)
---------------------------------------- 2017-04-11-21-34-53-102
[notice ]  (242,0,0)
---------------------------------------- 2017-04-11-21-34-53-133
[notice ]  (244,2,0)
---------------------------------------- 2017-04-11-21-34-53-164
[notice ]  (242,2,0)
```

Figura 50: Log de calibración resultante

Cabe mencionar además que se podría evitar realizar el análisis manual del *log* de calibración para obtener los valores concretos de cada punto en coordenadas de profundidad del sensor *Microsoft Kinect*, cargándose automáticamente de cierta forma luego de procesar cada punto (por ejemplo, promediando los valores obtenidos durante el intervalo de calibración de cada punto particular). Si bien de este modo todo el proceso de calibración se realizaría de forma completamente automática, esta característica fue dejada fuera del alcance de la herramienta de calibración.

4. Al finalizar la ejecución de la herramienta de calibración, adicionalmente al *log* de calibración descrito en el punto anterior se genera un fichero *XML* con una estructura similar al fichero pasado como entrada del proceso de calibración. A diferencia de la configuración inicial, el fichero *XML* resultante es parcialmente rellenado con la información que se obtuvo al momento de culminar el proceso de calibración, tal como se muestra en la *Figura 51*. Más específicamente, el fichero incluye la cantidad de puntos calibrados, los puntos concretos calibrados en coordenadas de pantalla y una plantilla con los puntos calibrados en coordenadas de profundidad del sensor *Microsoft Kinect*, cuyos valores deberán ser rellenados en base al análisis manual del *log* de calibración detallado en el punto anterior.

```
<calibration>
    <stage>3</stage>
    <mirror>0</mirror>
    <gridRows>1</gridRows>
    <gridColumns>1</gridColumns>
    <nProjPoints>4</nProjPoints>
    <projPoints>
        <pointList>0,5,18,23</pointList>
        <point>
            <x>0.151628852</x>
            <y>0.202455342</y>
        </point>
        <point>
            <x>0.836658180</x>
            <y>0.202455342</y>
        </point>
        <point>
            <x>0.151628852</x>
            <y>0.742857099</y>
        </point>
        <point>
            <x>0.836658180</x>
            <y>0.742857099</y>
        </point>
    </projPoints>
    <nCamPoints>4</nCamPoints>
    <camPoints>
        <point>
            <x>0</x>
            <y>0</y>
            <z>0</z>
        </point>
        <point>
            <x>0</x>
            <y>0</y>
            <z>0</z>
        </point>
        <point>
            <x>0</x>
            <y>0</y>
            <z>0</z>
        </point>
        <point>
            <x>0</x>
            <y>0</y>
            <z>0</z>
        </point>
        <point>
            <x>0</x>
            <y>0</y>
            <z>0</z>
        </point>
    </camPoints>
</calibration>
```

Figura 51: Plantilla *XML* resultante del proceso parcial de calibración

La *Figura 52* muestra el mismo fichero XML luego del análisis manual del log de calibración y el rellenado manual de los puntos de calibración en coordenadas de profundidad del sensor *Microsoft Kinect*. Este fichero final es el utilizado para almacenar el mapeo de calibración entre el sistema de coordenadas de pantalla y el sistema de coordenadas de profundidad del sensor *Microsoft Kinect*, a utilizar posteriormente según se requiera.

```
<calibration>
    <stage>3</stage>
    <mirror>0</mirror>
    <gridRows>1</gridRows>
    <gridColumns>1</gridColumns>
    <nProjPoints>4</nProjPoints>
    <projPoints>
        <pointList>0,5,18,23</pointList>
        <point>
            <x>0.151628852</x>
            <y>0.202455342</y>
        </point>
        <point>
            <x>0.836658180</x>
            <y>0.202455342</y>
        </point>
        <point>
            <x>0.151628852</x>
            <y>0.742857099</y>
        </point>
        <point>
            <x>0.836658180</x>
            <y>0.742857099</y>
        </point>
    </projPoints>
    <nCamPoints>4</nCamPoints>
    <camPoints>
        <point>
            <x>-262</x>
            <y>-307</y>
            <z>0</z>
        </point>
        <point>
            <x>255</x>
            <y>-290</y>
            <z>0</z>
        </point>
        <point>
            <x>-265</x>
            <y>-9</y>
            <z>0</z>
        </point>
        <point>
            <x>244</x>
            <y>1</y>
            <z>0</z>
        </point>
    </camPoints>
</calibration>
```

Figura 52: Fichero *XML* resultante del proceso completo de calibración

5. Por último, la propia herramienta de calibración permite ejecutar en modo tiempo real con el fin de testear el mapeo resultante. En este caso, se dibuja en todo momento sobre la proyección un círculo de color en la posición donde está la mano del usuario o el punto de

contacto con la superficie, dependiendo del módulo que se esté utilizando (módulo de seguimiento de manos o módulo táctil).

Contando con el mapeo parcial entre el sistema de coordenadas de proyección o pantalla y el sistema de coordenadas de profundidad del sensor *Microsoft Kinect* dado por la herramienta de calibración detallada anteriormente, solo resta definir la forma en que se obtiene el mapeo dado un punto genérico en coordenadas de profundidad del sensor *Microsoft Kinect*. En este sentido, tanto la herramienta de calibración en modo tiempo real como el *Core* del *framework* propuesto utilizan el mismo mecanismo de mapeo de un punto genérico tridimensional en coordenadas de profundidad del sensor *Microsoft Kinect* a un punto bidimensional en coordenadas de pantalla o proyección.

Este mecanismo utiliza el mapeo de los puntos discretos calibrados para interpolar los valores correctos para un punto genérico cuyo mapeo es desconocido [89]. Más específicamente, se divide la superficie en triángulos utilizando los puntos obtenidos durante la calibración, generando dos nuevos triángulos cada cuatro puntos de la grilla (mínima cantidad de puntos posibles durante la calibración). Como resultado se obtienen dos grillas de triángulos; una que incluye triángulos con vértices en coordenadas de pantalla o proyección, y otra con la misma estructura pero formada por triángulos con vértices en coordenadas de profundidad del sensor *Microsoft Kinect*. Así, dado un punto genérico en coordenadas de profundidad del sensor para el cual se desea obtener las correspondientes coordenadas de pantalla, se obtiene el triángulo en el cual cae el punto desde la grilla de triángulos en coordenadas de profundidad del sensor y se calculan las coordenadas baricéntricas utilizando los vértices de dicho triángulo. Una vez obtenidas las coordenadas baricéntricas del punto para el triángulo con vértices en coordenadas de profundidad del sensor, se aplican las mismas coordenadas baricéntricas al triángulo análogo pero considerando ahora los vértices en coordenadas de proyección, también obtenidas como parte de la calibración. Como resultado de este cálculo se obtienen las coordenadas de proyección del punto genérico que originalmente estaba en coordenadas de profundidad del sensor. En la *Figura 53* se puede apreciar el proceso de mapeo entre ambos sistemas de coordenadas basados en las coordenadas baricéntricas de los respectivos triángulos. En la figura, *K* es el sistema de coordenadas de profundidad del sensor *Microsoft Kinect* y *P* es el sistema de coordenadas de proyección.

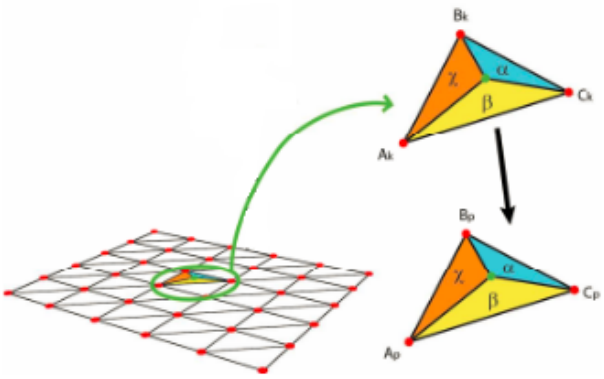


Figura 53: Grilla de triángulos en ambos sistemas de coordenadas

Para calcular las coordenadas baricéntricas se debe resolver el siguiente sistema de ecuaciones lineales, donde $p = (x, y)$ es el punto para el cual se quiere calcular las coordenadas dentro del triángulo dado por los vértices $v_1 = (x_1, y_1)$, $v_2 = (x_2, y_2)$ y $v_3 = (x_3, y_3)$, y λ_1, λ_2 y λ_3 son las coordenadas baricéntricas a calcular:

$$\begin{cases} x_1 * \lambda_1 + x_2 * \lambda_2 + x_3 * \lambda_3 = x \\ y_1 * \lambda_1 + y_1 * \lambda_2 + y_3 * \lambda_3 = y \\ \lambda_1 + \lambda_2 + \lambda_3 = 1 \end{cases}$$

El sistema de ecuaciones anterior se puede reescribir de forma matricial de la siguiente forma:

$$\begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Siendo M la matriz de coeficientes del sistema de ecuaciones anterior, podemos aplicar la regla de *Cramer* [19] para calcular las incógnitas λ_1, λ_2 y λ_3 que representan las coordenadas baricéntricas del punto $p=(x,y)$. Se comienza calculando D, el determinante de la matriz M:

$$\begin{vmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{vmatrix} = x_1 * y_2 + x_3 * y_1 + x_2 * y_3 - x_3 * y_2 - x_1 * y_3 - x_2 * y_1$$
$$= x_1 * (y_2 - y_3) + x_2 * (y_3 - y_1) + x_3 * (y_1 - y_2) = D$$

Luego se calculan los determinantes $D_{\lambda 1}$, $D_{\lambda 2}$ y $D_{\lambda 3}$ respectivamente, que se corresponden a los determinantes de las matrices resultantes de sustituir una fila de la matriz M según la regla de *Cramer*:

$$\begin{vmatrix} x & x_2 & x_3 \\ y & y_2 & y_3 \\ 1 & 1 & 1 \end{vmatrix} = x * y_2 + y * x_3 + x_2 * y_3 - x_3 * y_2 - x * y_3 - y * x_2$$
$$= x * (y_2 - y_3) + y * (x_3 - x_2) + x_2 * y_3 - x_3 * y_2 = D_{\lambda_1}$$

$$\begin{vmatrix} x_1 & x & x_3 \\ y_1 & y & y_3 \\ 1 & 1 & 1 \end{vmatrix} = x_1 * y + x_3 * y_1 + x * y_2 - y * x_3 - x_1 * y_2 - x * y_1$$
$$= x * (y_2 - y_1) + y * (x_1 - x_3) + x_3 * y_1 - x_1 * y_2 = D_{\lambda_2}$$

$$\begin{vmatrix} x_1 & x_2 & x \\ y_1 & y_2 & y \\ 1 & 1 & 1 \end{vmatrix} = x_1 * y_2 + x * y_1 + y * x_2 - x * y_2 - y * x_1 - x_2 * y_1$$
$$= x * (y_1 - y_2) + y * (x_2 - x_1) + x_1 * y_2 - x_2 * y_1 = D_{\lambda_3}$$

De esta forma, según la regla de *Cramer*, las incógnitas del sistema de ecuaciones lineales λ_1, λ_2 y λ_3 que representan las coordenadas baricéntricas del punto $p=(x,y)$, se obtienen de la siguiente forma:

$$\lambda_1 = \frac{D_{\lambda_1}}{D} = \frac{x * (y_2 - y_3) + y * (x_3 - x_2) + x_2 * y_3 - x_3 * y_2}{x_1 * (y_2 - y_3) + x_2 * (y_3 - y_1) + x_3 * (y_1 - y_2)}$$

$$\lambda_2 = \frac{D_{\lambda_2}}{D} = \frac{x * (y_2 - y_1) + y * (x_1 - x_3) + x_3 * y_1 - x_1 * y_2}{x_1 * (y_2 - y_3) + x_2 * (y_3 - y_1) + x_3 * (y_1 - y_2)}$$

$$\lambda_3 = \frac{D_{\lambda_3}}{D} = \frac{x * (y_1 - y_2) + y * (x_2 - x_1) + x_1 * y_2 - x_2 * y_1}{x_1 * (y_2 - y_3) + x_2 * (y_3 - y_1) + x_3 * (y_1 - y_2)}$$

Algunas consideraciones a tener en cuenta sobre el proceso de calibración desarrollado:

- Si bien es menos trabajoso realizar todo este proceso de calibración de forma completamente automática, es decir, proyectando un patrón de ajedrez, detectando cierta cantidad de puntos sobre la imagen de la cámara *RGB* del sensor y luego utilizando una transformación conocida entre la cámara *RGB* y la cámara de profundidad para obtener la transformación correcta entre ambos sistemas de coordenadas (lo que idealmente debería hacer *ofxReprojection* de forma nativa), en la práctica la calibración manual es una mejor alternativa [89]. Esto es debido a que los dedos humanos tienen cierto grosor, por lo que la diferencia entre el punto en la superficie y la posición de la punta del dedo obtenida es detectable. De esta forma, utilizando calibración manual el proceso de mapeo tiene en cuenta naturalmente este error en la detección del punto de contacto, a diferencia de la calibración automática donde los puntos obtenidos caen directamente sobre la superficie, sin tener en cuenta el grosor de los dedos.
- Si bien no se realizaron pruebas específicas para calcular la latencia en la interacción táctil incluyendo todos los cálculos para el correcto mapeo a coordenadas de proyección, se estima que se logra una latencia del orden de los 100 ms. Si bien esta latencia no es pequeña, se considera aceptable para el *framework* propuesto dado que no se espera utilizar los servicios del

framework para la construcción de aplicaciones en tiempo real como juegos u otros sistemas interactivos que requieran tiempos de respuesta cortos.

- Dado el proyector con el que se contó para este proyecto, existen ciertas restricciones que se deben cumplir para poder reconocer correctamente el patrón y realizar el proceso de calibración, que son por ejemplo que haya poca iluminación y que el proyector no esté demasiado distante de la superficie.

Finalmente, tal como se mencionó antes en esta misma sección, una vez se tienen las coordenadas de proyección del punto de contacto con la superficie gracias al proceso descrito previamente, se debe transformar el punto a coordenadas de escena en la aplicación que esté ejecutando sobre el *framework*. Esto es realizado mediante una operación provista por *openFrameworks* (ya detallado en el *Capítulo 2*), la cual permite obtener las coordenadas de escena dado un píxel o coordenada de pantalla. De esta forma, es posible establecer si el punto de contacto implica interacción con alguno de los elementos de la aplicación.

4.3. Seguimiento esqueletal

Es posible hacer el seguimiento de cada uno de los usuarios en la escena, lo que comúnmente se conoce como seguimiento esqueletal, mediante el sensor *Microsoft Kinect* y el *middleware NiTE*, ya detallado en el *Capítulo 2*. El rastreo del esqueleto realizado por *NiTE* consiste en procesar las imágenes de profundidad obtenidas por el sensor para detectar formas humanas e identificar las diferentes partes del cuerpo de cada usuario presente en la escena, siendo cada una representada por su articulación principal o *joint*, que no es más que un punto en el espacio expresado en milímetros con respecto a la posición del sensor *Microsoft Kinect* dentro de la escena. *NiTE* permite obtener la representación de un esqueleto formado por un total de 15 *joints*, ilustrados en la *Figura 54*. Mediante los *joints* de los usuarios es posible determinar dónde está ubicado cada uno en cada *frame* de tiempo dentro de la escena, conociendo así, sus movimientos durante el transcurso de la interacción.

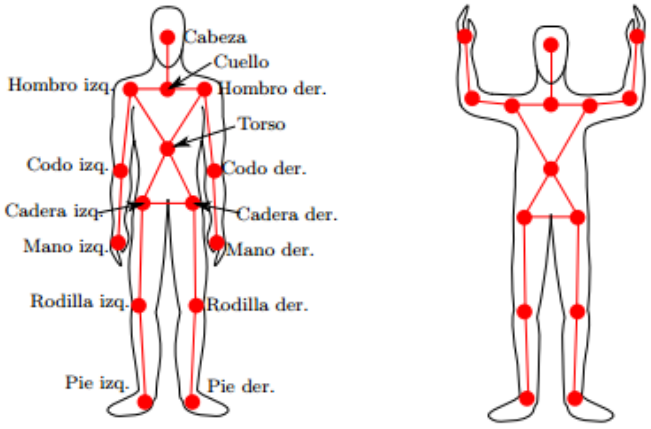


Figura 54: Puntos de interés del cuerpo de un usuario [50]

Para este trabajo los *joints* más importantes son aquellos que representan las manos de los usuarios, ya que son quienes permiten realizar el emparejamiento con la posición de la mano que brinda el sensor *Leap Motion*, con el fin de asociar dicha información adicional al usuario detectado. En definitiva, haciendo corresponder el centro de la mano provisto por el sensor *Microsoft Kinect* con el centro de la mano provisto por el sensor *Leap Motion* se puede establecer una correspondencia entre manos y cuerpo, aumentando la información que brinda un sensor con la que el otro sensor ofrece. Manteniendo esta correspondencia entre manos y cuerpo en todo momento, es posible además, por ejemplo, relacionar los gestos realizados mediante el sensor *Leap Motion* con un usuario previamente reconocido por el sensor *Microsoft Kinect*. Estos y otros aspectos referentes a cómo se realiza dicho mapeo se presentan con más

detalle en el *Capítulo 5*.

Es preciso mencionar también que fue necesario realizar diferentes pruebas con diferentes configuraciones con el fin de realizar un correcto seguimiento esqueletal. Inicialmente se estudiaron configuraciones en las que se contaba con un único sensor Microsoft Kinect dirigido de forma de abarcar tanto la superficie de interacción multitáctil como la zona de interacción gestual, por lo que un único sensor reconocía ambos tipos de interacción. Esta configuración fue luego descartada debido a que el procesamiento para ambas cosas, el análisis de mapas de profundidad con *OpenCV* y el seguimiento esqueletal con *NiTE* hacían que requiriera un alto nivel de procesamiento, obteniendo grandes retardos en la detección de la interacción multitáctil como del seguimiento de los esqueletos en cuestión. Se optó entonces por pasar a la configuración final detallada en las subsecciones anteriores, consistente de un sensor *Microsoft Kinect* específico para la interacción multitáctil y otro específico para la interacción gestual y el reconocimiento de los usuarios.

4.4. Interacción multitáctil

Como se detalló en el *Capítulo 3*, las superficies multitáctiles con capacidad de cómputo propia son precisas pero muy costosas y muy poco flexibles en cuanto a su portabilidad en diferentes entornos. Por esto, como ya fue mencionado, el objetivo para este trabajo es desarrollar una superficie multitáctil de bajo costo sin capacidad de cómputo propia, evitando las problemáticas planteadas anteriormente y logrando que la interacción multitáctil pueda realizarse sobre cualquier tipo de superficie, sea una pared, una mesa, una puerta, u otras. De esta forma, si bien la complejidad de la solución se incrementa, es más extensible y genérica en cuanto a las posibles superficies de interacción a utilizar. Mediante los mecanismos descritos en esta sección es posible dotar cualquier superficie de cierta inteligencia mediante la posibilidad de interacción multitáctil, seleccionando acordemente los valores de ciertos umbrales que habilitan el reconocimiento y el filtrado de la información no relevante.

Para llevar a cabo la necesidad planteada se investigaron diversas bibliotecas y *frameworks* ya existentes que pudieran ser utilizados como base para dar soporte a la resolución multitáctil para la solución a desarrollar. Estas se presentan con más detalle en el *Anexo I* junto a las restricciones que obligaron a descartar su utilización. Dadas estas restricciones y con el objetivo de tener un mayor control sobre las interacciones reconocidas, se optó por una implementación propia basada en los mapas de profundidad brindados por el sensor *Microsoft Kinect*. Con este fin, el sensor es dispuesto boca abajo apuntando en dirección a la superficie de interacción tal como se muestra en la *Figura 55*. Mediante esta disposición, la superficie de interacción queda limitada por la porción de mesa que abarca el campo de visión (del inglés *Field of View* o *FOV*) del sensor *Microsoft Kinect* como así también del proyector. Esto agrega de cierta forma, una restricción a la solución, ya que no se podrá interactuar sobre toda la extensión de la superficie de interacción sino solamente dentro de la porción de superficie visible por el sensor-proyector.

En lo que respecta al mecanismo concreto de detección de interacción multitáctil, una manera intuitiva de llevarlo a cabo mediante el uso de una cámara de profundidad es observando las diferencias entre un mapa de profundidad inicial tomado como base [156], es decir, un mapa captado previamente en donde no existen elementos de usuario en la escena sino simplemente la superficie de interacción, denominada *instantánea base*, y un mapa de profundidad en tiempo real en el momento en que se está dando la interacción, denominada *instantánea en tiempo real*. Así, por ejemplo, para detectar la presión de un dedo sobre una superficie, se debe comparar la imagen *instantánea en tiempo real* con la imagen de profundidad base almacenada inicialmente, es decir, la *instantánea base*. En esta comparativa, aquellos píxeles que se encuentren más cercanos de la cámara con respecto a la imagen de base obtenida serán los correspondientes a los dedos y manos del usuario, ya que esto indica que son elementos que se agregaron con respecto a la imagen *instantánea base*.

Figura 55: Disposición del sensor *Microsoft Kinect* para la resolución multitáctil

Así, conociendo la distancia o profundidad ($d_{surface}$) a la que se encuentra la superficie de interacción, todo píxel más cercano que ésta distancia será considerado o bien como parte de las manos o brazos del usuario, o bien como otros objetos que no están propiamente en contacto con la superficie en sí misma. Adicionalmente, es posible establecer umbrales que permitan filtrar aquellos píxeles que están o bien muy lejanos (d_{min}) o bien muy cercanos a la superficie (d_{max}). Estos píxeles no son considerados como puntos de contacto con la superficie y por lo tanto no son tenidos en cuenta como parte del objeto que está estableciendo contacto. A modo general entonces, se puede establecer la relación $d_{max} > d_{x,y} > d_{min}$ entre los diferentes umbrales definidos, ilustrada en la *Figura 56,* donde d_{max} *y* d_{min} definen los $d_{x,y}$ que efectivamente están en contacto con la superficie. El valor frontera d_{max} es elegido de forma tal que sea lo más similar posible a $d_{surface}$, considerando cierto margen con el fin de corregir posibles efectos de ruido introducido en el mapa de profundidad obtenido (lo ideal es minimizar $d_{surface} - d_{max}$).

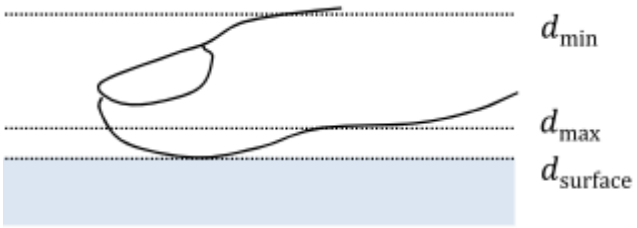


Figura 56: Definición de la zona de interacción mediante umbrales [156]

Por otra parte, en lo que respecta al valor frontera d_{min}, se debe tener en cuenta que la asignación de un valor muy grande o demasiado pequeño puede ocasionar cierta latencia en el momento en que se detecta la interacción. De todas formas, es preciso mencionar que mediante el uso de cámaras de profundidad no es posible detectar el momento exacto de la interacción, razón por la cual se asume cierta fisonomía de dedos y manos así como también, de su postura al momento de la interacción, agregando así ciertas restricciones adicionales a la solución. Por un lado, se supone que al momento de la interacción el dedo está en posición casi horizontal, mientras que por otro, se asume que tiene un grosor dado por un valor constante t. Así, se puede establecer que $d_{min} = d_{max} - t$ aproximadamente.

Como se mencionó previamente, toda la superficie abarcada por el campo de visión del sensor *Microsoft Kinect* y el proyector es considerado como área de interacción multitáctil. Si bien este es el objetivo, también se debe tener en cuenta que los usuarios posiblemente deseen apoyar objetos de uso común como teléfonos inteligentes, computadores portátiles, etc. A su vez, el usuario podría querer interactuar con estos elementos o incluso realizar algo tan natural como apoyar sus brazos sobre la superficie sin que sea considerado una interacción intencionada. Es por esto que en los mapas de profundidad obtenidos es preciso distinguir qué es considerado como parte de las manos de los usuarios y qué no, con el fin de reaccionar adecuadamente según corresponda. Esto se logra filtrando, de entre todos los objetos reconocidos como parte de la escena, aquellos cuyo tamaño sea mayor a cierto umbral previamente configurado como parte del fichero de configuración del sistema (no confundir con el umbral utilizado para reconocer los dedos), descrito con más detalle en el *Capítulo 5*. Este umbral es calculado en base a diversos ensayos previos consistentes en reconocer diferentes tipos de manos en diferentes posiciones, con el objetivo de obtener un tamaño óptimo que permita disminuir la cantidad de falsos positivos y negativos durante el reconocimiento. De este modo, recién luego que se detectan y filtran las áreas que representan potenciales manos de los usuarios se activa la algoritmia de interacción multitáctil detallada previamente, restringida sólo a dichas áreas.

4.5. Reconocimiento de gestos

Si bien los diferentes sensores utilizados tienen la capacidad de reconocer ciertos gestos de forma nativa tal como fue explicado en el *Capítulo 2*, es deseable que el *framework* que conforma la solución sea extensible en lo que respecta al reconocimiento de gestos, pudiendo agregar otros tipos de gestos diferentes a los provistos nativamente por los sensores de interacción. Es necesario entonces, contar con mecanismos que permitan implementar esta flexibilidad para el reconocimiento de nuevos gestos. Esto se logra gracias a la utilización del *framework GRT* (*Gesture Recognition Toolkit*), cuyas características ya fueron detalladas en el *Capítulo 2*, como base para el reconocimiento de gestos. Las características de reconocimiento nativas de este *framework* se complementan con componentes propios de la solución desarrollada con el objetivo de habilitar el agregado de gestos genéricos de forma lo más sencilla posible. Uno de estos componentes es la herramienta entregada conjunto al proyecto *GestureTrainers* que permite generar los datos necesarios para el agregado de nuevos gestos a reconocer mediante la configuración del *framework GRT* para cada uno de los sensores de la solución.

Para el agregado de nuevos gestos se debe generar inicialmente un fichero de entrenamiento. Este proceso implica realizar los diferentes gestos que se deseen reconocer, indicando a la aplicación de entrenamiento en cada caso a qué gesto se corresponde mediante un identificador prefijado para cada uno. Como resultado, se obtiene el fichero de entrenamiento que contiene los patrones que representan a los diferentes gestos definidos, que serán luego utilizados por el componente reconocedor para evaluar si un cierto flujo de datos generado en tiempo real en función de una interacción de usuario se corresponde o no con uno de los gestos entrenados. En este sentido, se puede decir que el sistema aprende a reconocer y calificar gestos futuros en base a la experiencia previa ganada durante la fase de entrenamiento, en la que un usuario experimentado indicó cómo clasificar cada patrón, lo cual en el área de aprendizaje automático se denomina comúnmente aprendizaje supervisado (del inglés *supervised learning*). Por más detalles sobre la funcionalidad de aprendizaje y reconocimiento del *framework GRT* referirse a la *Sección 2.2.5* del *Capítulo 2*.

Además de generar el fichero de entrenamiento, se debe configurar mediante un archivo de configuración del sistema, las características del reconocedor específico que utilizará cada sensor a incluir para reconocer gestos no nativos. Esto es así dado que según el tipo de gestos a reconocer es posible seleccionar un reconocedor óptimo para cada conjunto de gestos particular, así como también, el fichero de entrenamiento a utilizar en cada caso. Como se detallará en profundidad en el *Capítulo 5*, a este

reconocedor se le debe indicar el archivo de datos de pruebas utilizado para obtener métricas de la eficiencia del reconocimiento, el tipo de algoritmo utilizado para reconocer nuevas muestras (detallados en la *Sección 2.2.5* del *Capítulo 2*), y el mapeo entre el identificador asignado por el reconocedor al gesto en cuestión y el identificador de dicho gesto dentro del *framework* desarrollado.

4.6. Interferencia entre sensores

Como se mencionó en secciones anteriores, la solución final incluye dos sensores *Microsoft Kinect*, uno para dar soporte a la interacción multitáctil y otro para el reconocimiento de los usuarios en la escena y la interacción gestual. En el *Capítulo 2* se mencionó la forma en que el sensor *Microsoft Kinect* obtiene la información en la escena por medio de la proyección de un patrón conocido de rayos infrarrojos y el análisis de la deformación que éste sufre, captada tras rebotar con los diferentes elementos de la escena.

Esto implica que si más de un sensor proyecta patrones de rayos en la misma escena al mismo tiempo, un sensor puede captar patrones del otro sin ser capaz de distinguir su propio patrón, generando ambigüedades en las imágenes obtenidas. En definitiva, cuando se trata de hacer la correspondencia entre el patrón conocido y el proyectado, el sensor puede caer en áreas donde hay interferencia con rayos de patrones de otros sensores. Este problema se manifiesta en una interferencia que genera artefactos conocidos como "huecos negros" en los mapas de profundidad, que representan la falta de información debido a la interferencia entre los patrones de rayos. El sensor *Microsoft Kinect* no devuelve información cuando no tiene cierto grado de confianza entre la correlación de los patrones. Si la interferencia es mínima los huecos generados tienden a ser pequeños y aislados. En cambio si la interferencia aumenta los huecos tienden a ser más grandes, por lo que el problema se potencia a medida que hayan más sensores en la escena.

Para ejemplificar lo anterior, se presenta a continuación un conjunto de imágenes de profundidad obtenidas por los sensores *Microsoft Kinect* para diferentes disposiciones. La *Figura 57* representa la escena con un único sensor *Microsoft Kinect* encendido, observándose que prácticamente no existen interferencias sobre la superficie de interacción representada por la mesa. Sí se observan pequeños huecos a cada lado de la superficie, los cuales se corresponden a los sensores *Leap Motion* que, como se detalló en el *Capítulo 2*, también proyectan rayos infrarrojos y por lo tanto también generan una pequeña interferencia sobre el lugar que están colocados.

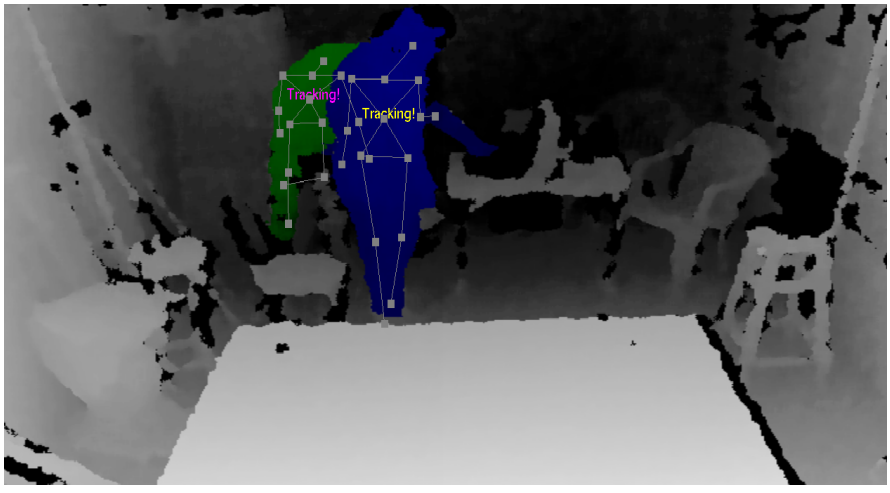


Figura 57: Superficie de interacción con un único sensor *Microsoft Kinect* activo

Al encender un segundo sensor *Microsoft Kinect* la generación de ruido es inminente, tal como se puede observar en la *Figura 58*. En este caso los huecos negros casi cubren la totalidad de la superficie de interacción.

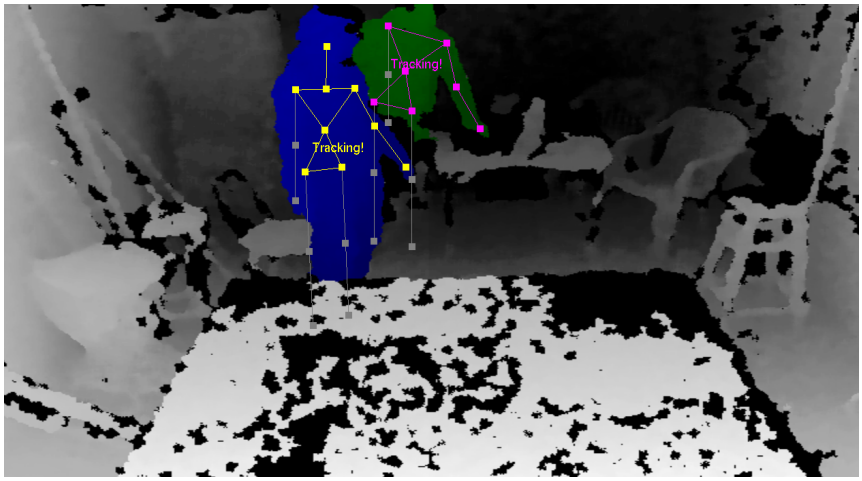


Figura 58: Superficie de interacción con dos sensores *Microsoft Kinect* activos

Este ruido generado por la interferencia de los rayos de ambos sensores causa una gran pérdida de información, haciendo impracticable cualquier tipo de interacción sobre la superficie de forma eficiente. En el *Anexo J* se describen más en detalle algunas de las soluciones investigadas en lo que refiere al problema de la interferencia entre sensores. Como resultado de esta investigación, se puede concluir que la mejor solución es evitar la interferencia en la medida de lo posible, ya sea evitando utilizar varios sensores o de lo contrario procurando que los patrones de rayos infrarrojos no se superpongan. Dado que esto no es posible para la disposición de sensores utilizada como parte de este trabajo, la solución que genera mejores resultados es la introducción de vibraciones. Para esto se deben instalar pequeños motores sobre cada sensor *Microsoft Kinect* con el fin de generar una pequeña vibración que lo haga oscilar levemente, generando como consecuencia el desfase de los patrones para cada sensor. Debido a la imposibilidad de contar con motores especialmente diseñados para tal fin, se considera como una alternativa la introducción de pequeños dispositivos masajeadores como el de la *Figura 59*, los que presentan características similares a los motores en cuanto a las vibraciones generadas.

Figura 59: Dispositivo utilizado para generar vibraciones

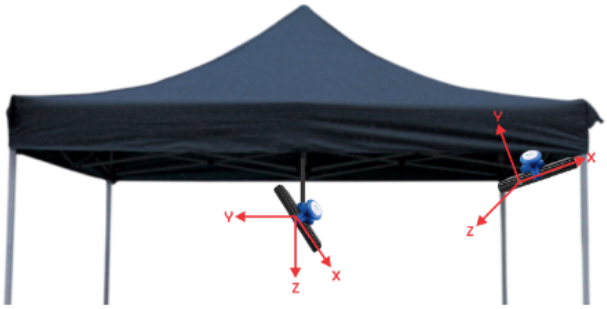


Figura 60: Disposición de motores vibratorios

Colocando estos dispositivos sujetos a cada uno de los dos sensores *Microsoft Kinect* que conforman la solución, tal como muestra el diagrama de la *Figura 60*, se logra generar una vibración suficiente para desfasar los patrones de rayos infrarrojos de los sensores. Como consecuencia se disminuye radicalmente la cantidad de "huecos negros" en los mapas de profundidad resultantes.

Inicialmente se agregó un único dispositivo vibratorio, obteniendo resultados que, si bien eran

notoriamente mejores en comparación con los obtenidos sin introducir vibraciones, aún presentaban algunos huecos que ponían en riesgo la implementación eficiente de la interacción sobre la superficie. Esto se puede ver ilustrado en la *Figura 61*.

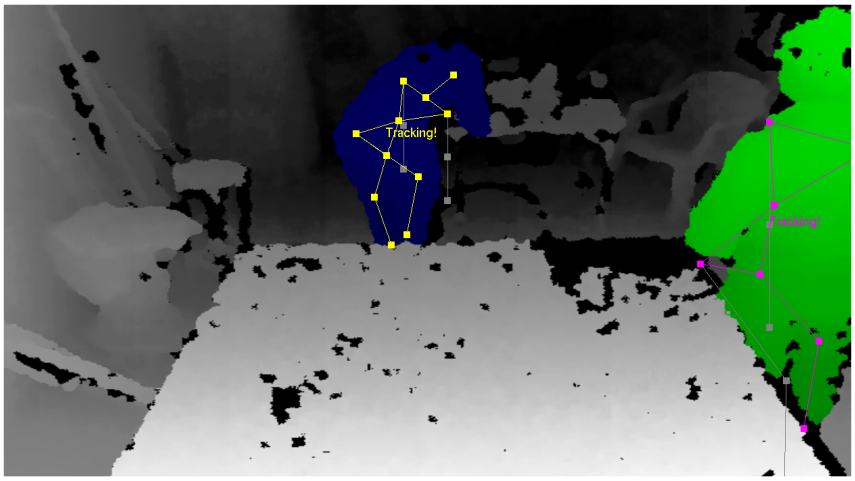


Figura 61: Superficie de interacción con un motor vibratorio

Para la solución final se agregó un segundo dispositivo vibratorio junto al sensor *Microsoft Kinect* restante, de forma de generar vibraciones en ambos sensores. En este caso, los resultados en cuanto a interferencia fueron muy cercanos al óptimo, tal como se pueden ver en la *Figura 62*. Se puede concluir entonces, que la técnica de reducción de ruido mediante vibraciones arroja muy buenos resultados y su implementación es muy sencilla, razón por la cual fue la solución adoptada para abordar el problema de la interferencia de los rayos infrarrojos entre los diferentes sensores utilizados como parte de la solución propuesta.

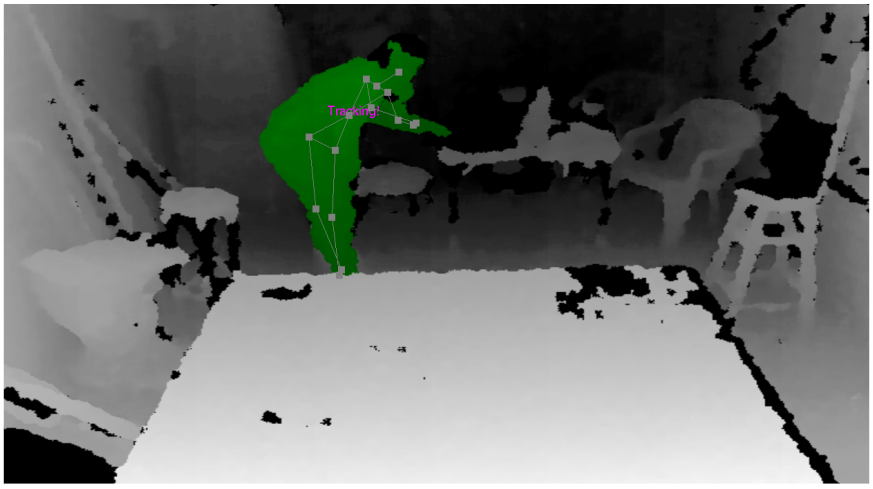


Figura 62: Superficie de interacción con dos motores vibratorios

4.7. Arquitectura distribuida y asincrónica

La arquitectura de la solución, con el fin de contemplar la distribución y disposición necesaria de los diferentes sensores dadas las restricciones ya descritas, requiere que sea naturalmente distribuida. Por esta razón, cada uno de los componentes distribuidos que ejecutan como parte de dicha arquitectura debe comunicarse mediante un protocolo de red, ya sea sobre *Internet* o sobre una red de área local. Se define

entonces un protocolo de aplicación, *ANIMuS Framework Protocol* (*AFP*), basado en el envío de mensajes sobre conexiones *UDP* y *TCP* según el tipo de mensaje, el cual se describe con más detalle en el *Capítulo 5*. Los mensajes entre los diferentes nodos son básicamente mensajes de registro de sensores y envío de eventos de reconocimiento, ya sea de usuarios en la escena o de gestos por ellos realizados.

Es preciso mencionar que el hecho de que la arquitectura sea distribuida agrega una complejidad adicional a la solución, ya que más allá de la dificultad inherente en definir e implementar un protocolo de aplicación que sea lo suficientemente performante como para soportar el flujo de datos que viaja entre los diferentes nodos, se generan dificultades referentes al control que se tiene sobre los datos que se envían y reciben entre los nodos de forma asincrónica, lo que dificulta a su vez, la depuración y pruebas de los diferentes nodos que conforman el sistema. Esta asincronía implica también que se deban definir correctamente las regiones de mutuoexclusión, es decir, que se debe asegurar que los recursos compartidos son manipulados mediante operaciones atómicas en el sentido de que no se pueden ejecutar dos operaciones diferentes sobre el mismo recurso en el mismo momento. Además, esta asincronía natural de la arquitectura distribuida se ve potenciada aún más debido a que existen múltiples hilos de procesamiento ejecutando en un mismo nodo.

Dada la característica distribuida del sistema es necesario también serializar los diferentes tipos de datos intercambiados entre los nodos. La complejidad respecto a este tema es que gran parte de la información a intercambiar no son tipos básicos y en algunos casos serializarlos no es trivial, principalmente las estructuras jerárquicas y otros tipos de datos complejos. Utilizando la biblioteca *Boost Serialization*, ya descrita en el *Capítulo 2*, para serializar los diferentes tipos de datos, es posible serializar estructuras complejas, ganando además gran flexibilidad en lo que refiere al tipo de serialización a utilizar, ya sea archivos *XML*, texto plano con formato, datos binarios, etc. Para depurar y probar la serialización de los tipos de datos intercambiados entre los diferentes nodos se realizaron múltiples pruebas consistentes en serializar cada tipo de datos particular en uno de los extremos y verificar que se reciba sin alteraciones en el otro extremo. Estas pruebas de concepto fueron útiles también para probar unitariamente los aspectos básicos de la comunicación por la red entre los nodos y la correctitud del protocolo de comunicación entre ellos.

Finalmente, un tema no menor al momento de definir los nodos de la arquitectura distribuida y la información que fluye entre ellos, fue decidir qué datos enviar de los obtenidos desde los diferentes sensores. En particular, una decisión crucial es si enviar solo los datos procesados (eventos y gestos) desde un nodo a otro o si enviar además los datos crudos completos obtenidos por los diferente sensores. Centrándose en uno solo de los sensores, asumiendo que el sensor *Microsoft Kinect* genera mapas de profundidad con resoluciones de 640x480 píxeles (donde cada píxel es representado por 16 *bits*) a una tasa de 30 *frames* por segundo, se observa que:

		Total (aprox.)
Bytes* por *frame	640x480x16/8	614.400 (614.4 kB)
***Bytes* por segundo**	614.400*30	18.432.000 (18.4 MBps)
***Bits* por segundo**	18.4*8	**147.4 Mbps**

Esto es, que para soportar uno solo de los sensores de forma fluída la red debería ser capaz de transportar información a una tasa sostenida de 147.4 Mbps. Si bien este es un número relativamente razonable para casi cualquier red de área local actual capaz de transportar datos a cientos o incluso miles de Mbps (no así para Internet), a esta cantidad hay que sumarle el flujo de datos generados por los demás sensores, más los eventos y gestos enviados luego del procesamiento de la información cruda, más información necesaria para el correcto funcionamiento de la propia red. Esto hace que en la práctica no sea

factible enviar los datos crudos completos, o al menos no sin comprimirlos de alguna forma. De hecho, se pudo comprobar empíricamente que incluso con un único sensor *Microsoft Kinect* generando datos crudos y enviandolos al otro nodo, la red es claramente el cuello de botella que hace que la demora sea tal que se perciba una notoria latencia en los gestos reconocidos. Es por esto que, la solución final envía únicamente datos procesados entre nodos; principalmente eventos de reconocimiento de gestos y otros mensajes de control para el correcto funcionamiento del sistema como fue mencionado anteriormente. El protocolo particular definido para la comunicación será descrito en detalle en el *Capítulo 5*.

4.8. Arquitectura extensible

Otro de los puntos a abordar fue el hecho de diseñar una arquitectura extensible en el sentido que permita el agregado de más sensores de interacción, así como también, la posibilidad de agregar nuevos gestos a ser reconocidos por el *framework*. Este último punto, como ya fue descrito en la *Sección 4.6*, fue tenido en cuenta mediante un mecanismo de generación de datos de entrenamiento y la definición de diferentes reconocedores configurables a utilizar por parte de cada sensor. En lo que respecta a la extensión de la solución con nuevos sensores, el hecho de agregar más sensores de tipo *Leap Motion* permite que más usuarios puedan realizar gestos tridimensionales. Así mismo, el agregado de más sensores *Microsoft Kinect* permite, por un lado, extender la superficie de interacción multitáctil tal como se detalló en la *Sección 4.4* (siendo necesario en este caso al agregado de un proyector adicional que permita proyectar información sobre la nueva zona de interacción multitáctil), y por otro, extender el espectro de detección de usuarios en la escena tal como ilustra la *Figura 63*. Esto último habilita la detección de usuarios no solo en la zona hacia la que apunta el sensor *Microsoft Kinect* de usuario ya descrito como parte de la solución, sino alrededor de toda la superficie de interacción.

Figura 63: Espacio de interacción extendido con un sensor *Microsoft Kinect* adicional

Por otra parte, si se desea extender la solución con otros tipos de sensores de interacción como pueden ser el sensor *MYO* [90][91], sensores *ASUS Xtion* [159] u otros, la arquitectura lo soporta configurando e implementando un nuevo componente dentro del nodo encargado de obtener datos desde los diferentes sensores. Este componente debe implementar ciertas operaciones predefinidas (una interfaz, tal como se detallará en el *Capítulo 5*), necesarias para obtener información desde cada sensor particular. Una vez obtenida esta información, es luego pre procesada y enviada en forma de eventos al otro nodo, donde se procesará junto a demás información obtenida para la escena.

Mediante cualquiera de las extensiones mencionadas anteriormente es posible enriquecer la capacidad de interacción de la solución por parte de los usuarios, estando todas ellas soportadas por el *framework* propuesto gracias a la forma en que se diseñó la arquitectura. De todos modos, cabe mencionar que, si bien la arquitectura fue ideada y diseñada para dar soporte a estas problemáticas, no fueron incluídas en la solución final debido a restricciones de tiempo y recursos necesarios para su implementación. Como se ya se mencionó, todos los detalles de implementación sobre las diferentes formas de extender la solución se presentan en el *Capítulo 5*.

4.9. Definición de un *framework*

Como ya se detalló en el *Capítulo 1*, uno de los objetivos centrales del proyecto es la definición de un *framework* que, como tal, exponga todos los servicios que el sistema provee y estos puedan ser consumidos mediante una interfaz o *API* (*Application Programming Interface*). De esta forma, cualquier aplicación puede hacer uso del *framework*, simplemente extendiendo de la interfaz e implementando los métodos que provee según el fin específico de la aplicación, tal como se describirá en el siguiente capítulo.

El lenguaje de programación utilizado para generar el *framework* es *C++* mediante la utilización del entorno de desarrollo *Microsoft Visual Studio 2012*, por lo que cualquier aplicación que desee hacer uso del *framework* debe estar desarrollada en este mismo lenguaje. Se genera además una solución portable mediante la utilización de bibliotecas compatibles con los distintos sistemas operativos como los sistemas operativos *Windows*, *Linux* y *Mac*. Por detalles sobre la aplicación particular desarrollada utilizando los servicios provistos por el *framework* como parte de este trabajo referirse al *Capítulo 6*.

El *API* provista por el *framework* expone cada uno de los servicios que pueden ser consumidos por las aplicaciones, definiendo el alcance de la solución desarrollada. Cabe mencionar en este punto que durante el transcurso del proyecto el alcance fue modificándose en base a ciertas restricciones, haciendo que los servicios provistos por el *framework* sea vean alterados. A modo de ejemplo, uno de los servicios que inicialmente se pensaba proveer como parte de la interfaz estaba relacionado con el reconocimiento de sensores y otros objetos que estuvieran sobre la superficie de interacción, con el fin de que se pudiera interactuar con ellos de alguna forma. Si bien todos los detalles del *API* provista por el *framework* se describen en el *Anexo M* correspondiente a la guía de referencia del *framework*, se presenta a continuación una agrupación de los diferentes servicios expuestos, junto a una descripción de cada uno de los grupos de servicios definidos.

4.9.1 Reconocimiento de gestos tridimensionales

Este grupo de servicios permite notificar a las aplicaciones registradas sobre los diferentes eventos y gestos espaciales reconocidos por el sistema. Los servicios de este grupo son los siguientes:

- `waveHand3DGestureDetected`
- `raiseHand3DGestureDetected`
- `clickHand3DGestureDetected`
- `crossedHands3DGestureDetected`
- `psiHand3DGestureDetected`
- `screenTapFinger3DGestureDetected`
- `keyTapFinger3DGestureDetected`
- `swipeFinger3DGestureDetected`
- `circleFinger3DGestureDetected`
- `pinchHand3DGestureDetected`
- `grabHand3DGestureDetected`
- `pushHand3DGestureDetected`

- turnHand3DGestureDetected
- zoomHand3DGestureDetected
- zoomFinger3DGestureDetected
- letterFinger3DGestureDetected
- symbolFinger3DGestureDetected

4.9.2 Reconocimiento de gestos multitáctiles

Este grupo de servicios permite notificar a las aplicaciones registradas sobre los diferentes eventos y gestos multitáctiles reconocidos por el sistema. Los servicios de este grupo son los siguientes:

- tapTouchFingerGestureDetected
- longTapTouchFingerGestureDetected
- doubleTapTouchFingerGestureDetected
- dragTouchFingerGestureDetected

4.9.3 Reconocimiento de usuario

Este grupo de servicios permite notificar a las aplicaciones registradas los usuarios que ingresan o se van de la escena, y las actualizaciones correspondientes (posiciones, *joints*, etc). Los servicios de este grupo son los siguientes:

- userDetected
- userUpdated
- userLost

4.9.4 Inicialización y registro

Este grupo de servicios permite a las aplicaciones realizar todo lo relacionado con la configuración e inicialización del contexto del *framework.* Los servicios de este grupo son los siguientes:

- startAnimus
- stopAnimus
- registerToGestures

4.9.5 Información de sistema

Este grupo de servicios permite a las aplicaciones obtener información general del estado del sistema. Los servicios de este grupo son los siguientes:

- getAnimusUsersInformation
- getAnimusDevicesInformation

4.9.6 Eventos usuales

Este grupo de servicios permite notificar a las aplicaciones registradas sobre los eventos usuales que se dan con *mouse* y/o teclado en el sistema. Si bien no son servicios utilizados como parte de la interacción natural provista por el sistema, son brindados con fines de *debug* a los usuarios finales del *framework.* Los servicios de este grupo son los siguientes:

- keyPressed

- keyReleased
- mouseMoved
- mouseDragged
- mousePressed
- mouseReleased

4.10. Construcción del espacio de trabajo

Uno de los aspectos más complejos de la solución fue la conformación de un espacio de trabajo, principalmente por la disponibilidad de una red de conexión (tanto de área local como a *Internet*) de buena calidad y de un espacio físico lo suficientemente amplio como para la disposición de los diferentes dispositivos y la superficie de interacción. Luego de múltiples dificultades se logró montar la estructura que da soporte a la solución haciendo uso de un gazebo, pudiendo posicionar los diferentes sensores a los lados y colocar la superficie de interacción en el medio. Así, en uno de los lados del gazebo se monta un sensor *Microsoft Kinect* apuntando en dirección paralela a la superficie de interacción junto a uno de los motores detallados en la *Sección 4.6,* tal como se puede apreciar en la *Figura 64*. Este es el sensor encargado de detectar los usuarios presentes en la escena.

Figura 64: Sensor *Microsoft Kinect* para detección de usuarios y motor vibratorio en estructura

El restante sensor *Microsoft Kinect* se monta de la parte superior del gazebo apuntando perpendicularmente a la superficie de interacción, y también se le sujeta un motor, como se muestra en la *Figura 65.* Este sensor es el encargado de detectar la interacción multitáctil de los usuarios presentes en la escena. En esta misma estructura se coloca también el proyector, así como también una cámara con lente de ángulo amplio utilizada para grabar la interacción de los usuarios de forma que abarque la mayor cantidad de escena posible.

El proyector es colocado de forma tal que cubra un área de la mesa similar a la cubierta por el campo de visión del sensor de interacción multitáctil, de forma que todos los elementos proyectados sean alcanzables durante la interacción. El sensor *Microsoft Kinect* correspondiente a la detección y seguimiento de usuario abarca el mayor espectro posible bajo la condición de que su campo de visión también debe incluir a la superficie de interacción. De esta forma, es posible determinar en todo momento qué usuario está realizando qué gesto multitáctil mediante el mapeo de la posición de los *joints* de las manos y las posiciones de los *touchs* realizados. Adicionalmente, como se muestra en la *Figura 66*, bajo el gazebo se coloca la mesa que va a oficiar de superficie para la interacción multitáctil y sobre ésta se colocan los dos sensores *Leap Motion.*

Figura 65: Sensor *Microsoft Kinect* para detección de interacción multitáctil y motor vibratorio en estructura

Como se puede apreciar también, el gazebo es cubierto de tela negra a modo de oscurecer lo más posible es espacio de interacción. Este detalle es debido a la falencia del nivel de luminosidad del proyector con el que se cuenta para este trabajo, que apenas posee un nivel de 15 lúmenes provocando que la proyección sobre la zona de interacción no sea visible al menos que se encuentre en un espacio totalmente a oscuras. Como se describe en el *Capítulo 7* si se contara con un proyector de mayor luminosidad podría subsanarse esta restricción referente a que el espacio de interacción se encuentre totalmente a oscuras.

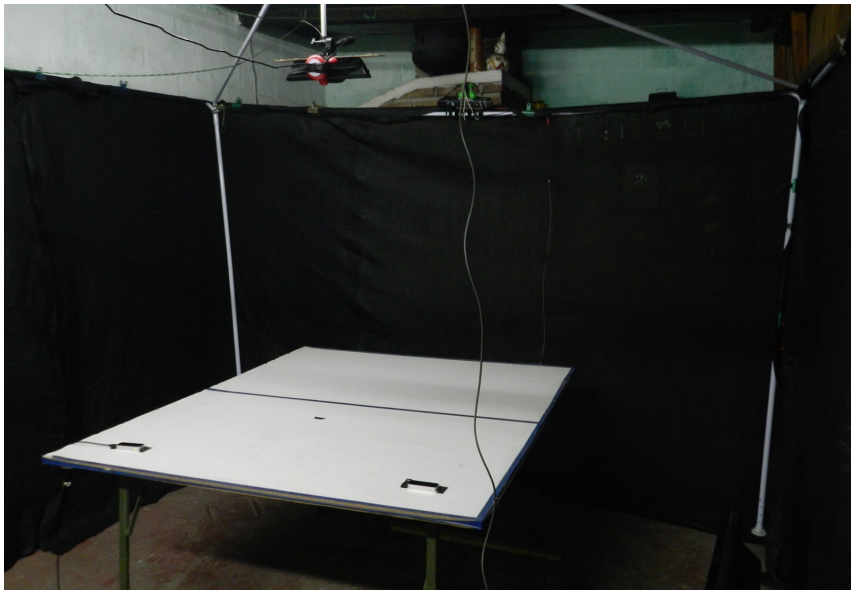


Figura 66: Mesa que oficia de superficie de interacción y sensores *Leap Motion*

Finalmente, en la *Figura 67* se pueden apreciar los computadores y el *router* que da soporte a la solución (ver especificaciones en *Sección 5.1* del *Capítulo 5*). A cada uno de los computadores se conecta un sensor *Microsoft Kinect* y un sensor *Leap Motion* tal como se detalló en la *Sección 4.1*. Además, uno de estos equipos es utilizado a su vez como nodo de procesamiento (*Core*) del sistema. Si bien, como se detallará en el *Capítulo 5*, lo ideal es contar con tres equipos físicos para la ejecución del sistema de forma que el nodo de procesamiento quede en un equipo exclusivo, dados los recursos disponibles se distribuyeron todos los componentes en dos equipos para la solución final. De todas formas, por cómo está diseñada la arquitectura del sistema, es perfectamente escalable para utilizar un tercer equipo que aloje exclusivamente el nodo de procesamiento en caso de contar con los recursos suficientes. Por otra parte, mediante la introducción del *router* se construyó una red de área local para conectar los diferentes computadores mejorando así la performance de la comunicación. Si bien inicialmente se comenzó utilizando una red *WiFi*, se percibió cierta latencia en la llegada de los paquetes de datos que se vió reflejada en la

aplicación final en forma de *delays* en la llegada de las notificaciones.

Figura 67: Equipos físicos que dan soporte a la solución final

4.11. Desarrollo de aplicaciones

Como parte de la solución propuesta se desarrolló una aplicación interactiva a modo de prototipo, que permite visualizar algunos de los servicios provistos por el *framework* de interacción generado. Los detalles de la aplicación se presentan en el *Capítulo 6*.

4.12. Resumen del capítulo

El presente proyecto implicó la resolución de diferentes problemáticas y la toma de diferentes decisiones para poder llegar a una resolución final que cumpliera con los objetivos planteados. En este capítulo se describe cada uno de los puntos principales a resolver, conjuntamente con las decisiones tomadas, así como también, las dificultades encontradas a lo largo del camino para su resolución. Los puntos principales a resolver incluyen la problemática de integrar múltiples sensores y de diferente tipo en una misma solución global, así como también, manejar los diferentes tipos de sistemas de coordenadas que se presentan. Por otro lado, también se plantea la problemática de lograr realizar el seguimiento esqueletal de los diferentes usuarios presentes en la escena y determinar cuáles realizan gestos del tipo multitáctiles y/o espaciales. Construir un espacio de trabajo que dé soporte a una arquitectura distribuida, extensible y asincrónica en la que se apoye la construcción de un *framework* que brinde los servicios para la generación de aplicaciones que saquen provecho de la interacción multitáctil y espacial también son de los puntos principales detallados que se debieron resolver.

Capítulo 5: Diseño de Arquitectura

En este capítulo se detallan todas las características y las decisiones tomadas para definir el diseño de la arquitectura que brinda soporte a la solución propuesta. Se describe el modelo de capas, naturalmente presente por ser una arquitectura distribuida orientada a servicios, la interrelación entre ellas y el protocolo de comunicación utilizado, sus componentes y configuraciones principales.

5.1 Modelo de capas

La arquitectura desarrollada para dar soporte a la solución se encuentra compuesta básicamente de tres capas, *Data Access*, *Core* y *Application*, dispuestas tal como muestra la *Figura 68*. Como se puede apreciar, las capas están organizadas en forma de pila o *stack* (siendo la capa inferior la de más arriba en la figura), de forma que una capa consume servicios de la capa inmediatamente superior y provee servicios a la capa inmediatamente inferior. Esta forma de disponer las capas facilita la encapsulación de funcionalidades y permite generar una solución más modular, genérica y flexible.

Cada una de las capas se ejecuta en uno o varios de los nodos físicos que dan soporte a la solución, a algunos de los cuales se conectan sensores mientras que otros son puramente nodos de procesamiento de información. De esta forma, existen nodos dedicados a la lectura de datos provenientes de los diferentes sensores y otros abocados a obtener un valor agregado de dichos datos en base a su procesamiento y correlación dentro del sistema. En las subsecciones que siguen se describe cada una de las capas mencionadas y sus principales responsabilidades.

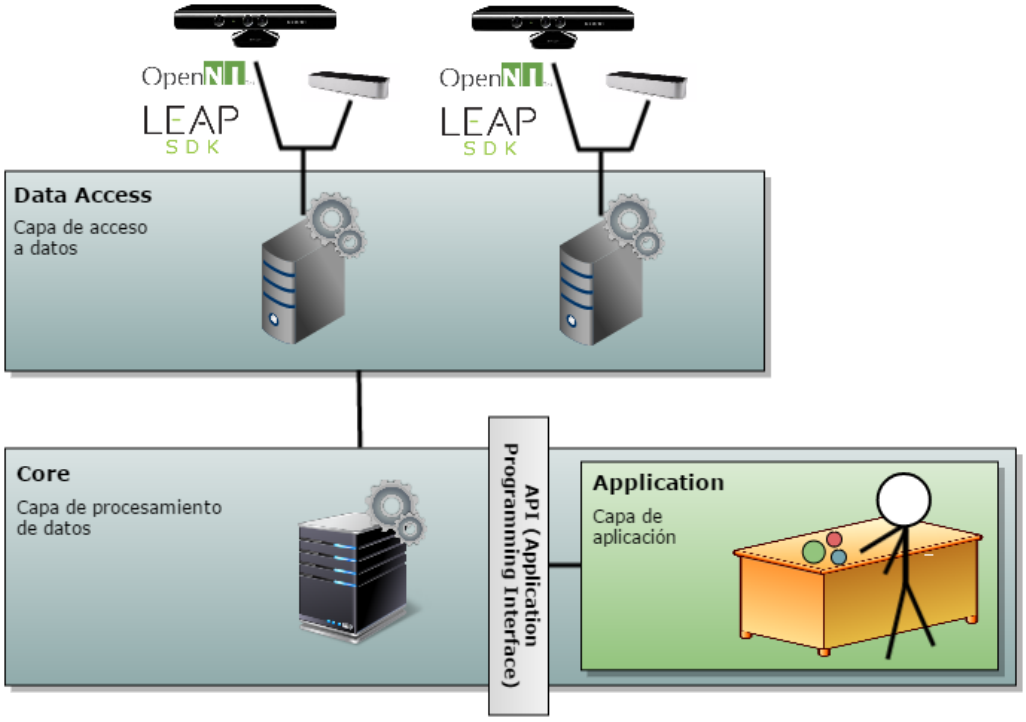


Figura 68: Diagrama de capas

5.1.1. *Data Access*

La capa *Data Access* provee todo lo referente al acceso a los datos provistos por los diferentes sensores que se encuentren configurados en el sistema. El sistema está conformado entonces por una o

varias instancias de esta capa, cada una de las cuales ejecuta en un nodo físico independiente. De esta forma, la capa *Data Access* es provista en forma de una aplicación *standalone* que ejecuta en cada nodo que requiera acceder a datos crudos provistos por un sensor.

Esta capa es la encargada de registrar en el sistema los diferentes sensores *Leap Motion* y *Microsoft Kinect* configurados según el nodo, así como también recabar todos los datos provenientes de ellos. También es responsabilidad de ésta capa que los datos obtenidos sean dispuestos en mensajes acordes al protocolo de comunicación definido para el sistema (*ANIMuS Framework Protocol*, detallado en la *Sección 5.5*) y enviados al nodo que ejecuta la capa *Core* para su procesamiento. Estos mensajes contendrán eventos de actualización de los usuarios presentes en la escena o eventos informando gestos reconocidos.

5.1.2. *Core*

La capa *Core* contiene la lógica principal del *framework* construido, siendo responsable del procesamiento de los mensajes recibidos desde la capa *Data Access*, es decir, eventos de registro de sensores, eventos de reconocimiento y actualización de usuarios, y eventos de reconocimiento de gestos. Esta capa es la responsable de procesar estos eventos y actuar acordemente, notificando a la capa *Application* de los gestos y actualizaciones reconocidos para los diferentes usuarios presentes en la escena.

Dado que el *Core* es el componente centralizado de procesamiento para los diferentes nodos de acceso a datos, el sistema incluye una única instancia de esta capa ejecutando en un nodo independiente para alivianar la carga de los demás nodos y evitar de esta forma posibles cuellos de botella en el procesamiento de los mensajes. Como parte de la solución propuesta, esta capa es provista en forma de biblioteca estática, brindando una *API* (*Application Programming Interface*) a las aplicaciones que deseen utilizar los servicios del *framework*. En el *Anexo M* se presenta con más detalle el *API* de servicios provista por el *framework*.

5.1.3. *Application*

La capa *Application* consiste básicamente de la aplicación final desarrollada para un fin específico. La aplicación consume los servicios provistos por el *framework*, registrándose a diferentes tipos de eventos para los cuales desea actuar en consecuencia modificando los elementos gráficos correspondientes. Esto hace que la aplicación desarrollada como parte de esta capa deba seguir el paradigma de programación conocido como programación orientada a eventos, del inglés *Event Driven Programming* [26], consistente en definir el comportamiento de los elementos de la aplicación y el flujo en general en base a ciertos eventos a los cuales se registró previamente.

Es responsabilidad de esta capa el procesamiento de las notificaciones provenientes desde la capa *Core*, y la actualización y el renderizado de los diferentes elementos de la escena según las notificaciones recibidas y el estado en que se encuentre. Cabe mencionar que dado que el *API* provista por la capa *Core*, como ya se detalló, es provista en forma de biblioteca estática, la instancia de la capa de aplicación ejecuta en el mismo nodo que la capa *Core* con el fin de consumir correctamente los servicios. Una alternativa natural es que la capa *Core* sea una aplicación *standalone* al igual que las diferentes instancias de la capa *Data Access*, proveyendo los servicios a través de la red a la capa *Application*, pero dada la complejidad, tanto en recursos necesarios como en lógica adicional requerida de agregar un nuevo nivel de comunicación, la alternativa fue descartada.

Por más detalles referentes a esta capa y a la aplicación particular desarrollada como prototipo como parte de este proyecto, referirse al *Capítulo 6*.

5.2 Diagrama de despliegue

En esta sección se presenta un diagrama de despliegue, del inglés *deployment diagram*, en el que se ilustra qué componentes *hardware* (nodos) existen, y qué componentes *software* de la solución (capas descritas anteriormente) ejecutan en cada uno de ellos. Si bien el presentado no es un diagrama de despliegue a bajo nivel, se complementa con un diagrama de comunicación donde se pueden apreciar los diferentes tipos de comunicación existentes entre los nodos con el fin de comunicar eficientemente los datos provistos por las distintas capas que conforman la solución.

Como se mencionó anteriormente, la solución presentada incluye dos instancias de la aplicación *standalone* que implementa la capa *Data Access*, cada una ejecutando en un nodo físico diferente. De esta forma, la solución combina dos nodos de obtención de datos (aunque pueden existir más, eventualmente con otros tipos de sensores como se detallará en la *Sección 7.2*), a cada uno de los cuales se conecta un sensor *Leap Motion* y un sensor *Microsoft Kinect*, gestionados por la capa *Data Access* que ejecuta en cada uno de ellos. Por otra parte, a los efectos de tener un mejor rendimiento la solución también requiere de un nodo específico para alojar la aplicación particular de la capa *Application*, la cual, como ya se detalló, debe compartir el nodo con la capa *Core* con el fin de poder consumir los diferentes servicios brindados por este componente central de procesamiento. Así, la distribución final de nodos físicos y capas lógicas queda tal como en la *Figura 69.*

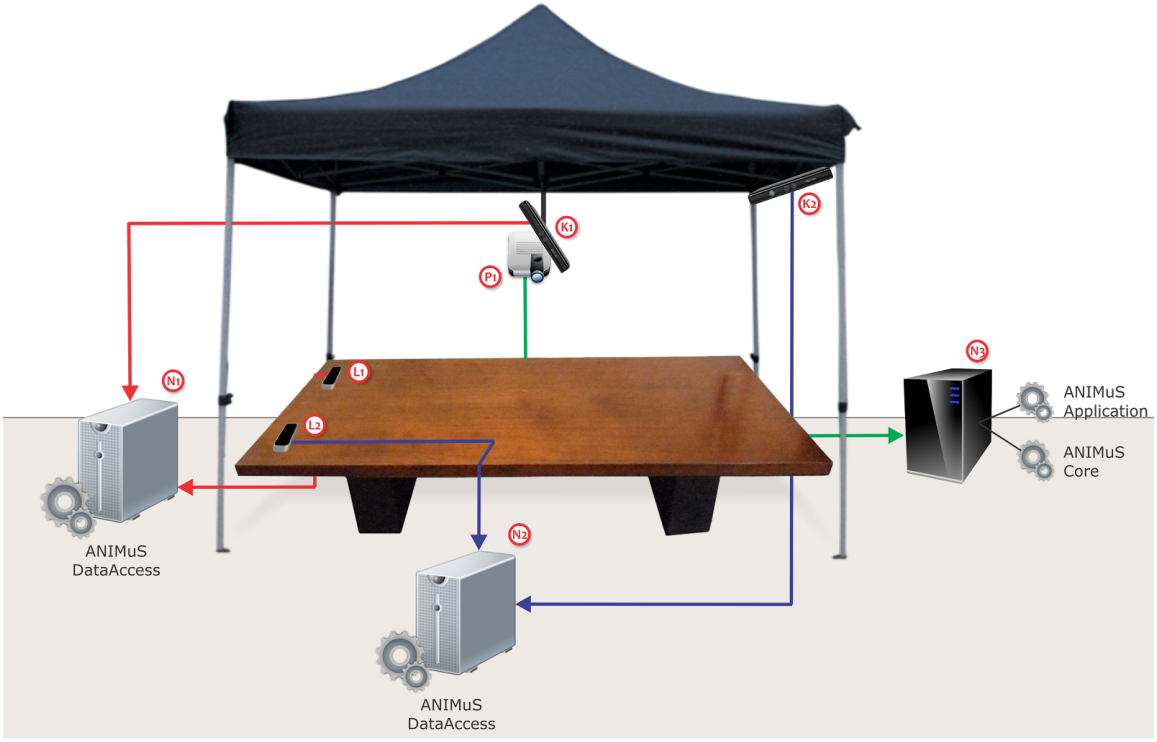


Figura 69: Diagrama de despliegue parcial, disposición de nodos y capas de la solución

Más específicamente, cada nodo del diagrama anterior tiene las siguientes características:

1. *Nodo 1 (N1 en el diagrama)*: este nodo es un computador en el que ejecuta la aplicación *standalone* de la capa *Data Access*, permitiendo la obtención de información desde el sensor *Microsoft Kinect* encargado del reconocimiento de interacción multitáctil (K1 en el diagrama) y uno de los sensores *Leap Motion* encargado del reconocimiento gestual tridimensional de los usuarios (L1 en el diagrama). Las especificaciones *hardware* y *software* de este nodo para la solución final son las siguientes:

Software	
Sistema operativo	Windows 7 Professional
Dependencias externa	Leap Motion SDK OpenNI 2.0 Visual C++ Redistributable for Visual Studio 2012

Hardware	
CPU	Intel Core i7-4790 @2.4Ghz
Memoria RAM	4 GB
VGA	AMD Radeon HD 6000 series (1GB DDR3)
Red	Broadcom NetLink Gigabit Ethernet (10/100/1000MBit)
Conexiones	3 USB 2.0, 1 VGA, 1 HDMI

2. *Nodo 2 (N2 en el diagrama):* este nodo es análogo al anterior en el sentido que ejecuta la aplicación *standalone* de la capa *Data Access*, pero en este caso obteniendo información desde un sensor *Microsoft Kinect* encargado del reconocimiento de los usuarios presentes en la escena (K2 en el diagrama) y del restante sensor *Leap Motion* con el mismo objetivo que en el nodo anterior (L2 en el diagrama). Las especificaciones *hardware* y *software* de este nodo para la solución final son las siguientes:

Software	
Sistema operativo	Windows 8.1 Pro
Dependencias externa	Leap Motion SDK OpenNI 2.0 Visual C++ Redistributable for Visual Studio 2012

Hardware	
CPU	Intel Core i7-4710HQ @2.5Ghz
Memoria RAM	8 GB
VGA	Nvidia GeForce GTX 860M (4GB DDR5)
Red	Broadcom NetLink Gigabit Ethernet (10/100/1000MBit)
Conexiones	3 USB 3.0, 1 HDMI, 1 DVI

3. *Nodo 3 (N3 en el diagrama):* este nodo es un computador preferentemente con mayor capacidad de cómputo que los anteriores, ya que en él se ejecuta la aplicación interactiva de la capa *Application* que se proyectará sobre la superficie, y que además hace uso de la biblioteca provista por la capa *Core* encargada del procesamiento de la información enviada por los nodos N1 y N2. Todo esto requiere que este nodo tenga por un lado una buena capacidad de procesamiento gráfico (ya que debe renderizar los elementos de la aplicación a una tasa aceptable) y por otro, gran capacidad de cómputo de propósito general con el fin de procesar adecuadamente los mensajes recibidos desde los nodos de la capa *Data Access*. Las especificaciones *hardware* y *software* de este nodo para la solución final son las siguientes:

Software	
Sistema operativo	Windows 7 Professional
Dependencias externa	Leap Motion SDK OpenNI 2.0 Visual C++ Redistributable for Visual Studio 2012

Hardware	
CPU	Intel Core i7-4790 @2.4Ghz
Memoria RAM	4 GB
VGA	AMD Radeon HD 6000 series (1GB DDR3)
Red	Broadcom NetLink Gigabit Ethernet (10/100/1000MBit)
Conexiones	3 USB 2.0, 1 VGA, 1 HDMI

En lo que respecta a la comunicación entre los diferentes nodos y capas, la más trivial se da entre las capas *Application* y *Core*, ya que como se detalló anteriormente, la comunicación es puramente lógica directo desde la memoria en un mismo *host* (dado que la aplicación particular de la capa *Application* se enlaza al *Core* del sistema en tiempo de compilación).

Por otra parte, la comunicación existente entre las capas *Core* y *Data Access* es más compleja, ya que dado que estas capas ejecutan en nodos físicos separados, tal como se puede apreciar en la *Figura 68*, el intercambio de información debe establecerse mediante una red de comunicación. En este caso, el flujo de información comienza cuando la capa *Data Access* obtiene datos desde los sensores configurados en los diferentes nodos físicos que ejecutan esta capa. En base a los datos obtenidos, se generan mensajes de aplicación definidos por el protocolo *ANIMuS Framework Protocol* (*AFP*), a detallar en la *Sección 5.5*. Estos mensajes contienen información relacionada a eventos de reconocimiento de gestos o información de actualización de los usuarios en la escena, y son enviados a través de la red sobre *UDP/IP* hacia la capa *Core* para su procesamiento. Una vez procesados estos mensajes y obtenido el usuario al cual corresponde el evento, se notifica a la capa *Application* ya sea el evento de actualización de usuario, información general del sistema o evento de reconocimiento gestual según la suscripción a eventos configurada para esta capa.

Cabe mencionar además que previo a enviar cualquier tipo de información recabada por los sensores por parte de alguno de los nodos de la capa *Data Access*, se deben registrar los correspondientes sensores en el sistema con el fin de que se les asigne un identificador global y luego se puedan correlacionar e identificar correctamente los futuros mensajes recibidos. Dicho registro consiste en que cada nodo que forma parte de la capa *Data Access* (y por lo tanto que cuenta con uno o varios sensores asociados), envíe un mensaje inicial de registro de sensores en el sistema. La información correspondiente a cada sensor contenida en este mensaje de registro se encuentra establecida en un archivo de configuración de la capa *Data Access*, detallado en la *Sección 5.6*. Este mensaje de registro también sigue la especificación del protocolo de aplicación *AFP*, y viaja desde la capa *Data Access* a la capa *Core* sobre *TCP/IP* a través de la red que comunica los nodos respectivos.

En este punto es preciso detallar la razón por la cual los mensajes que informan el registro de los sensores viajan sobre *TCP/IP*, en contraste a los mensajes correspondientes a datos obtenidos desde los sensores que viajan sobre *UDP/IP*. Dado que los mensajes de datos no requieren gran fiabilidad en la entrega al extremo opuesto, ya que la pérdida de un mensaje (debido a limitaciones o congestión de la red) no es crítica para el funcionamiento del sistema, pero sí requieren un gran ancho de banda para las tasas de

envío manejadas, *UDP* es el protocolo que más se ajusta a estas necesidades. Por el contrario, el mensaje de registro de un sensor desde un nodo de acceso a datos hacia el *Core* del sistema sí es crítico para su correcto funcionamiento, ya que de otra forma el *Core* no sería capaz de correlacionar y registrar futuros mensajes que transporten datos para determinado sensor. Es por esto que el protocolo elegido para esta comunicación es *TCP*. En este sentido, se puede decir que la comunicación entre las capas *Data Access* y *Core* es una comunicación bidireccional de doble canal, del inglés *dual channel*, ya que se transfieren mensajes mediante dos tipos de conexión diferentes.

Por último, también es preciso mencionar la comunicación existente entre las diferentes instancias de la capa *Data Access* y los sensores específicos conectados a cada nodo en los que ésta se ejecuta. En este caso la comunicación es gestionada por el *driver* de cada uno de los sensores presentes en el nodo, y los datos entre los nodos y los sensores fluyen a través de los puertos USB de los *hosts*.

En la *Figura 70* se puede apreciar un diagrama de comunicación que ilustra los múltiples tipos de comunicación existentes entre los distintos niveles del sistema, incluyendo el protocolo de comunicación y el medio por el cual fluyen los datos entre componentes.

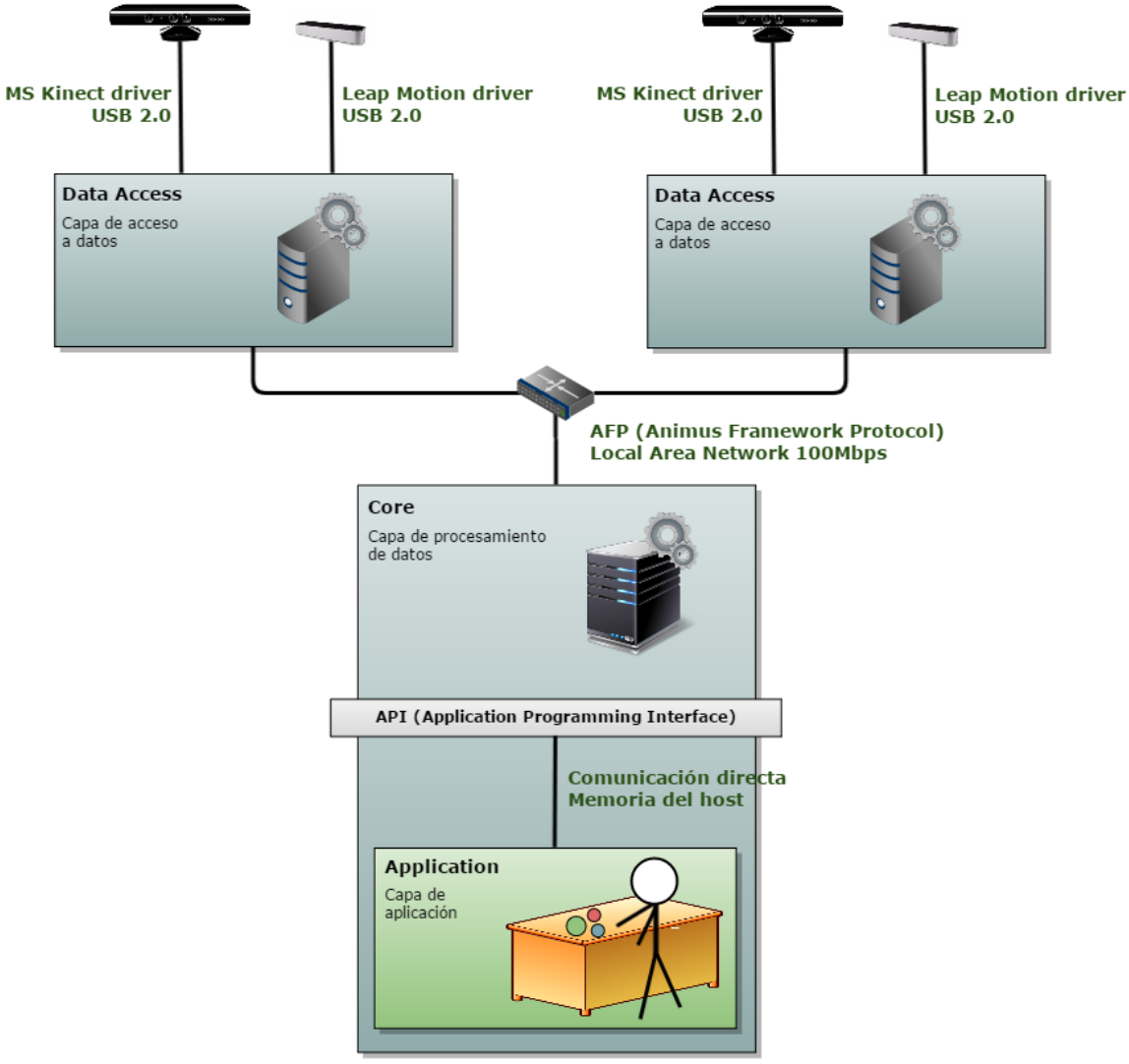


Figura 70: Diagrama de comunicación

5.4. Modelo de clases

En esta sección se detallan las clases principales de cada una de las capas mencionadas, clasificadas según el módulo en el que se encuentran. Para cada módulo se describen las funcionalidades que tiene a cargo, y para cada una de las clases que contiene se presentan brevemente las funcionalidades principales junto con el diagrama de clases correspondiente. Se puede apreciar el diagrama de clases para la arquitectura completa, incluyendo las tres capas que la conforman, en ficha *"Diagrama de clases"* anexada al final de este mismo capítulo. En lo que resta de la sección se describe con más detalle cada uno de los módulos existentes.

5.4.1 *Data Access*

La capa *Data Access* está compuesta básicamente por tres grandes módulos. Un módulo *Devices* encargado del mantenimiento y gestión de los sensores registrados en el sistema, un módulo *Managers* encargado de gestionar todo el flujo de datos general dentro de esta capa, y finalmente un módulo *Gesture Recognizer* encargado del reconocimiento de gestos; en las siguientes subsecciones se presenta el detalle para cada uno de ellos.

5.4.1.1. *Devices*

El módulo *Devices* es el encargado de mantener la información de todos los sensores configurados en el sistema (id, posición, descripción, etc.), así como también de orquestar la extracción de información desde cada uno de ellos para su posterior envío a la capa *Core*. El módulo está diseñado de forma tal que el manejo de los diferentes sensores sea lo más transparente posible para el desarrollador. Esto es implementado gracias a la inclusión de una clase base *Device*, que permite representar y gestionar un sensor genérico, de la que descienden clases específicas para cada uno de los sensores particulares soportados por el sistema. Para la solución propuesta se incluye la clase específica *LeapDevice* para representar sensores *Leap Motion*, y las clases *SingleKinectDevice* y *TouchKinectDevice* para representar sensores *Microsoft Kinect* con diferentes fines. Más específicamente, *LeapDevice* permite representar un sensor *Leap Motion* desde el cual se puede obtener información dentro de la capa de acceso a datos, mientras que *SingleKinectDevice* permite representar a un sensor *Microsoft Kinect* destinado al seguimiento de usuarios, y *TouchKinectDevice* un sensor del mismo tipo pero destinado a la interacción multitáctil. El diagrama de clases de la *Figura 71* ilustra la jerarquía descrita anteriormente.

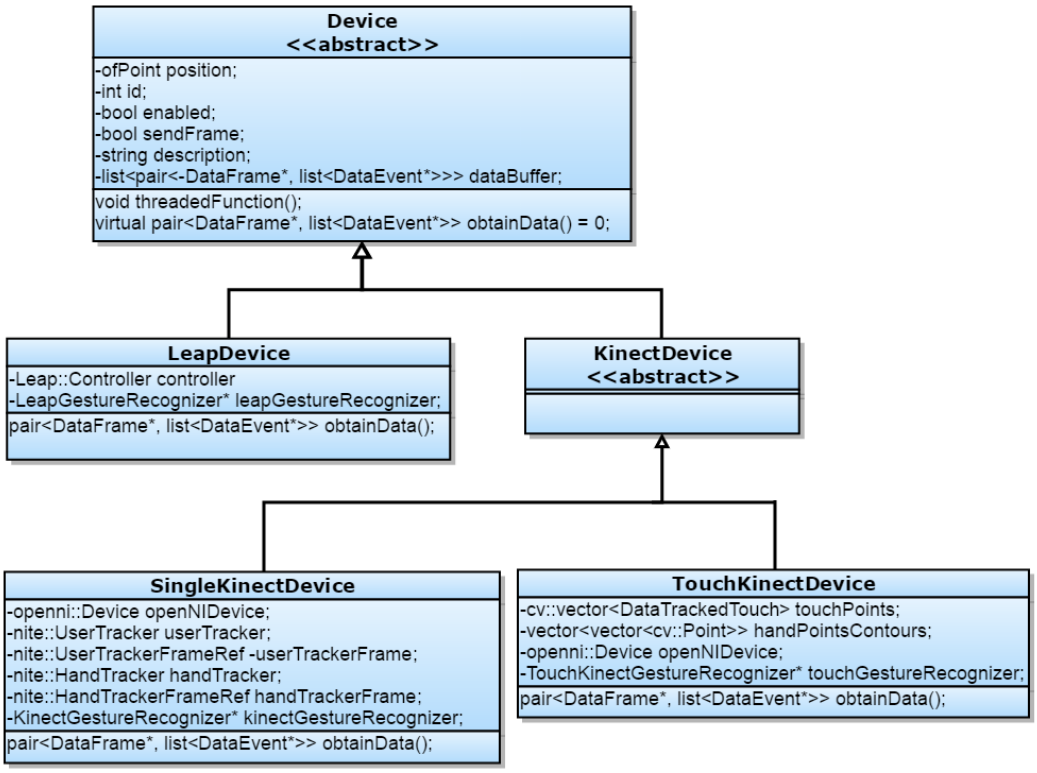


Figura 71: Diagrama de clases del módulo *Devices* de la capa *Data Access*

Cada una de estas clases específicas tiene como objetivo obtener la información desde el sensor particular (utilizando el *driver* correcto según el tipo de sensor gestionado). Esta lógica de extracción de información desde cada sensor forma parte del método sobrecargado *obtainData* visualizado en la *Figura 71*, lo que lo hace uno de los métodos principales de la capa de acceso a datos. En cada iteración entonces, cada uno de estos componentes obtiene datos crudos desde el sensor que tiene asociado y los almacena en una estructura de datos apropiada en la que se incluyen los datos del *frame* obtenido y la lista de eventos reconocidos en ese *frame* o iteración particular. Esta información es colocada en un *buffer* específico a cada sensor, y es luego recopilada por un componente del módulo *Managers* para su envío a la capa *Core*. En la *Tabla 1* se incluye un pseudocódigo simplificado del mecanismo de obtención de datos.

```
while (true) {
    foreach (device in Devices) {

        // Obtiene datos para el sensor particular desde la clase más específica
        Data data = device.obtainData()

        device.addTobuffer(data)
    }
}
```

Tabla 1: Pseudocódigo del método de extracción de datos desde los sensores

Cabe mencionar que gracias a que el método de obtención de información está definido en la clase padre *Device*, pero es sobrecargado e implementado en las clases más específicas, la obtención de información dentro de la capa de acceso a datos es completamente transparente del tipo de sensor desde el cual se obtienen, así como también lo suficientemente flexible como para agregar nuevos tipos de sensores simplemente implementando el método *obtainData* de extracción para el nuevo sensor particular.

5.4.1.2. Managers

El módulo *Managers* de la capa *Data Access* es el módulo que contiene los diferentes manejadores, es decir, aquellos componentes encargados de gestionar el flujo de ejecución general de la capa (configuraciones, envío y recepción de información hacia otras capas, etc.). El módulo contiene básicamente tres manejadores, representados por las clases *ConfigurationManager*, *DeviceManager* y *PacketManager*, tal como se ilustra en el diagrama de clases de la *Figura 72.* A grandes rasgos, el *ConfigurationManager* se encarga de gestionar todos los parámetros de configuración de esta capa, el *DeviceManager* se encarga de gestionar todos los sensores configurados, y el *PacketManager* se encarga de enviar y recibir paquetes hacia y desde la capa *Core*.

Durante la inicialización de la capa de acceso a datos, el *ConfigurationManager* carga todos los parámetros de configuración de esta capa desde un fichero de configuración del sistema y los mantiene almacenados para que puedan ser accedidos en cualquier momento y desde cualquier otro módulo que los requiera. Por más detalles sobre los parámetros configurables para esta capa referirse a la *Sección 5.6*.

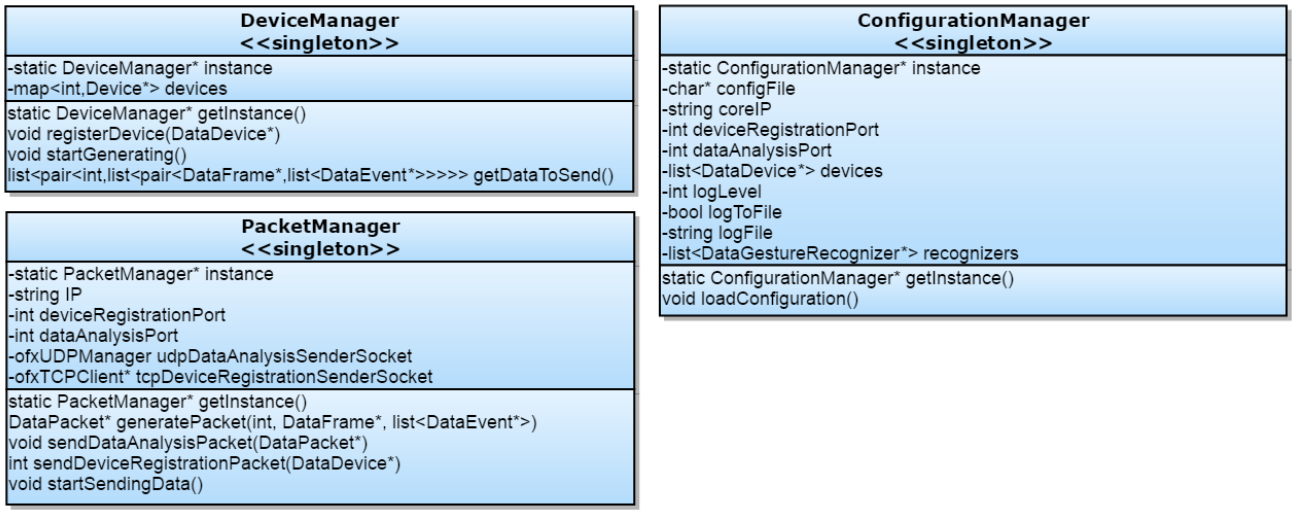


Figura 72: Diagrama de clases módulo *Managers*

Por otra parte, el *DeviceManager* es el encargado de llevar el registro y realizar la gestión de todo lo referente a los sensores configurados según la configuración del sistema. En particular, una vez obtenida la información sobre los sensores desde el *ConfigurationManager*, el *DeviceManager* crea instancias específicas del tipo *Device* según el tipo de sensor a instanciar, tal como se describió para el módulo *Devices*. Luego de creados todos los sensores existentes en el sistema, comienzan a extraer datos, los cuales, como ya mencionó, son dispuestos en un *buffer* particular para cada sensor. El *DeviceManager* es el responsable de recopilar los datos almacenados en los *buffers* de los distintos sensores y transmitirlos a medida que quedan disponibles al *PacketManager*. Este último es el encargado de obtener todos los datos recopilados, generar mensajes acordes al protocolo utilizado por el sistema (*ANIMuS Framework Protocol*, detallado en la Sección 5.5) y enviarlos hacia el *Core* para su procesamiento. Cabe mencionar que previo al envío de datos provenientes de los sensores existe un proceso de registro del sensor en el *Core* por parte del *DeviceManager*, quien envía un mensaje de registro y aguarda la recepción de un identificador para el sensor particular. Una vez finalizado este intercambio, el sensor queda registrado en el sistema, tanto en la capa de acceso a datos como en la capa *Core*. A continuación en la *Tabla 2* se presenta un pseudocódigo del proceso de recopilación y envío de datos hacia el *Core*.

```
While (true) {
```

```
    // Se obtienen los datos almacenados en buffer por los diferentes sensores
    Data[] data = deviceManager.getBufferedData()

    foreach (d in data) {

        // Para cada dato a enviar se genera y envía el mensaje de aplicación apropiado
        AFPacket p = generatePacket(d)
        dataSocket.sendDataPacket(p)
    }
}
```

Tabla 2: Pseudocódigo del método de recopilación y envío de datos hacia el *Core*

5.4.1.2. Gesture Recognizers

En el *Capítulo 2* se describieron los gestos que los diferentes sensores utilizados en la solución propuesta reconocen nativamente. El módulo *Gesture Recognizers* es el responsable del reconocimiento de gestos no nativos en base a la información cruda provista por estos sensores, y está fuertemente basado en algoritmos de aprendizaje automático mediante la librería *GRT* (*Gesture Recognition Toolkit*), ya detallada en el *Capítulo 2*. Este módulo incluye componentes diseñados de forma tal que el reconocimiento de gestos no nativos sea lo suficientemente flexible y genérico como para poder agregar nuevos gestos de forma transparente y sencilla, simplemente mediante la configuración de ciertos parámetros en el fichero de configuración del sistema. Estos componentes son *LeapGestureRecognizer*, *KinectGestureRecognizer* y *TouchKinectGestureRecognizer*, tal como se ilustra en el diagrama de clases de la *Figura 73.* Cada una de estas clases determina el reconocedor de gestos no nativos para los sensores *Leap Motion*, *Microsoft Kinect* de usuario y *Microsoft Kinect* de interacción multitáctil, respectivamente. Adicionalmente se incluye un componente de extracción de características, *FeatureExtractor* a detallar posteriormente, con el fin de facilitar el reconocimiento para ciertos gestos particulares.

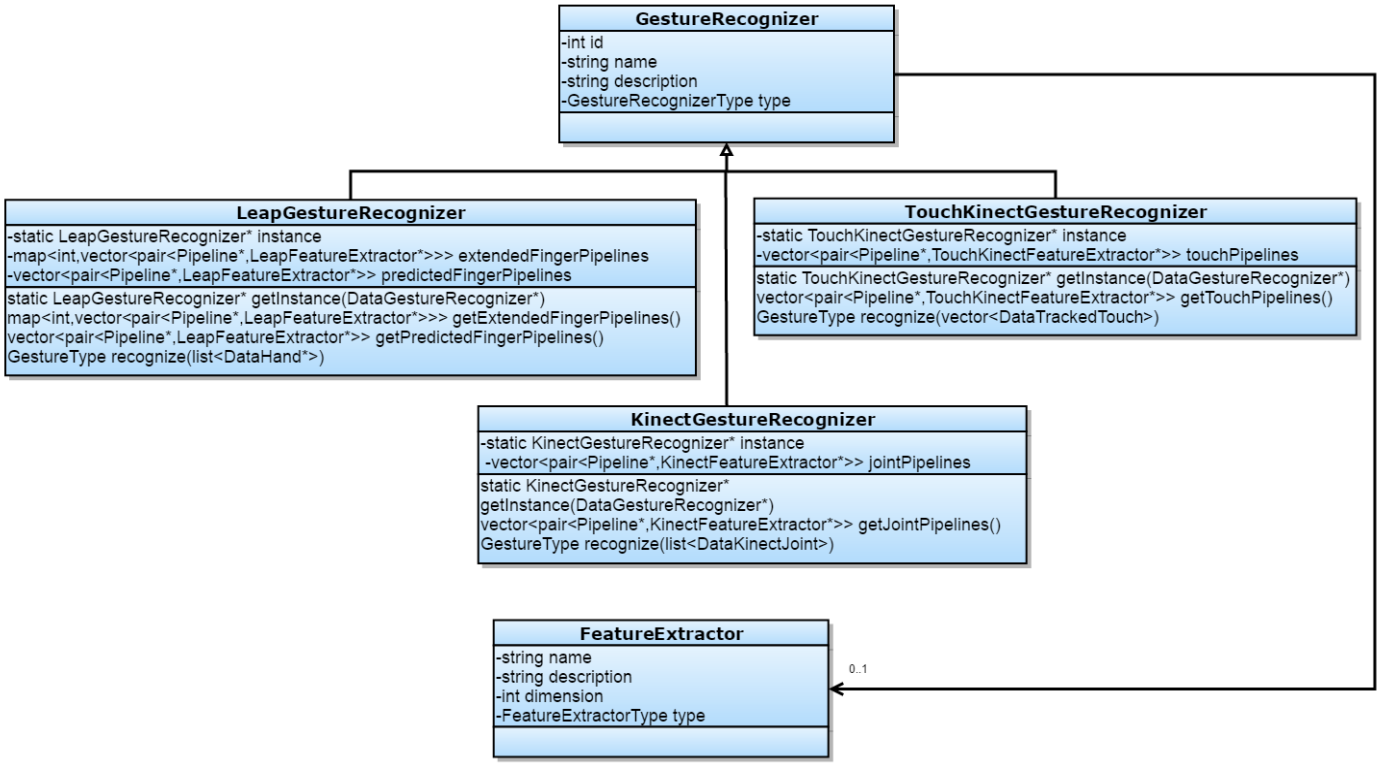


Figura 73: Diagrama de clases módulo *Gesture Recognizer*

Como se detalló en la *Sección 2.2.5.1*, la librería *GRT* en la que está basada este módulo hace uso intensivo del concepto de *pipeline* para el reconocimiento de gestos en base a la aplicación de algoritmos de aprendizaje automático a un conjunto inicial de datos (conjunto de entrenamiento), con el fin de aprender a reconocer y clasificar futuras muestras en tiempo real. Existen básicamente dos fases para el reconocimiento de gestos. Una primera fase, denominada fase de entrenamiento, en la que cierta cantidad de datos son pasados por el *pipeline* de reconocimiento con el fin de obtener un modelo de predicción entrenado, y una segunda etapa en la que se ingresan datos que se desea predecir al *pipeline* y se obtiene como resultado la predicción particular para dichos datos. Estos diferentes conceptos son básicamente los que mantienen los componentes del módulo de predicción *Gesture Recognizer*. Cada reconocedor para un sensor particular de los presentes en la *Figura 73*, incluye uno o más *pipelines* de reconocimiento configurados acordemente según los gestos que se desea reconocer. De esta forma, todo dato obtenido por el módulo de obtención de datos, previo a su envío hacia la capa *Core* es pasado por el módulo de reconocimiento (más específicamente, por el método *recognize* del reconocedor apropiado en la *Figura 73*) con el fin de reconocer gestos no nativos que también deban informarse al *Core*. Es preciso mencionar que si bien es posible agregar más de un *pipeline* para cada reconocedor a modo de ampliar el espectro de reconocimiento, al momento de clasificar una nueva muestra se recorren secuencialmente los diferentes *pipelines* hasta dar con el primero que clasificar la muestra correctamente (prioridad en orden de definición). Adicionalmente, cada uno de estos *pipelines* puede activarse o desactivarse según se desee desde la propia configuración del sistema.

```
Data[] data = packetManager.getDataToSend()

foreach (d in data) {

    // Se recorren todos los pipeline del reconocer y se retorna el gesto reconocido (si existe)
    Gesture g = gestureRecognizer.recognize(d)

    If (isValid(g)) {
        // Se genera mensaje de aplicación para gesto no nativo y se agrega a la cola de envío
        AFPacket p = packetManager.generatePacket(g)
        packetManager.addToQueue(p)
    }
}
```

Tabla 3: Pseudocódigo del proceso de reconocimiento de gestos no nativos

Además de definir las propiedades generales de cada *pipeline* a utilizar (algoritmo de reconocimiento, método de validación de datos, etc.), también es posible definir desde la configuración del sistema para este módulo los denominados extractores de características, representados por componentes del tipo *FeatureExtractor* tal como se muestra en la *Figura 74*. Los extractores de características son otro concepto utilizado por la librería de base *GRT*, basados en la premisa de agregar uno o más componentes adicionales al *pipeline* encargados de extraer características relevantes de los datos que fluyen por el *pipeline*, simplificando el posterior reconocimiento. Dentro del módulo de reconocimiento, cada uno de los extractores específicos según el sensor está definido por las clases *LeapFeatureExtractor*, *KinectFeatureExtractor* y *TouchKinectFeatureExtractor*, como se puede apreciar en el diagrama de clases de la *Figura 74*. De esta forma, como parte del proceso de reconocimiento para cada sensor particular, se agrega una nueva fase en la que eventualmente se extraen ciertas características relevantes a partir de los datos ingresados al *pipeline*, y se utilizan los datos resultantes simplificados para el reconocimiento.

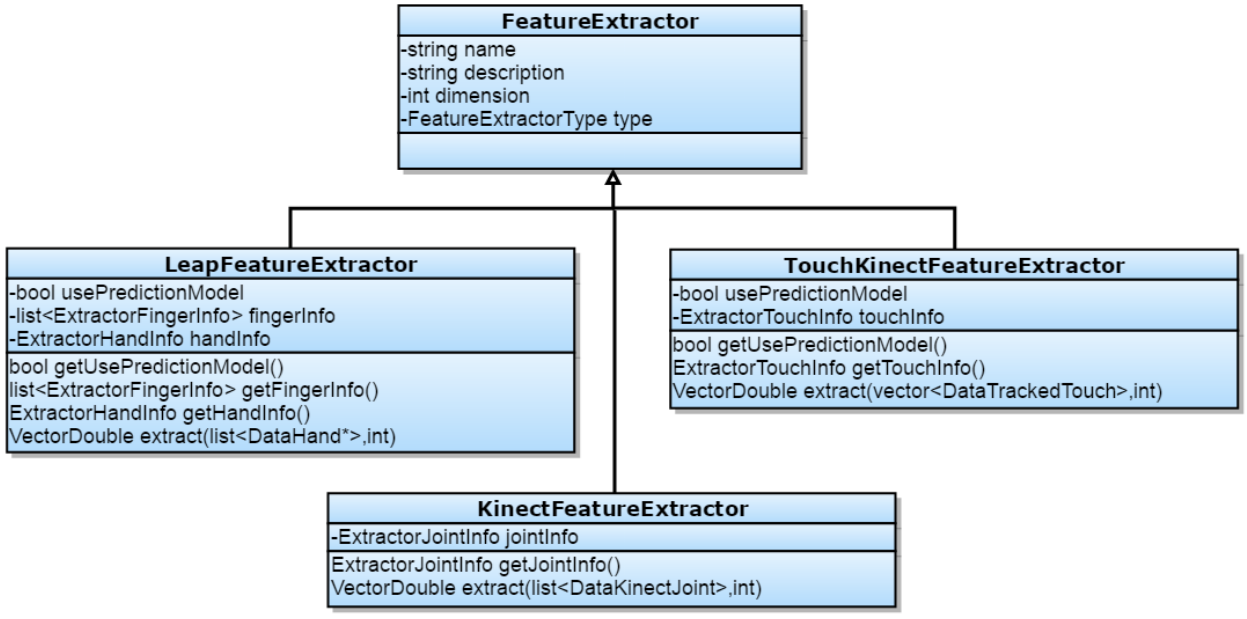


Figura 74: Diagrama de clases de los extractores módulo *Gesture Recognizer*

Así, por la forma en que está diseñado el módulo de reconocimiento y la posibilidad de configurarlo según las necesidades particulares, el reconocimiento se hace de forma transparente independientemente de si los datos a reconocer provienen de un sensor *Leap Motion* o uno de los sensores *Microsoft Kinect*, lo cual resulta en una arquitectura más genérica, modular y flexible. De esta forma, si se desea agregar un nuevo tipo de sensor para el cual se quiera reconocer gestos no nativos, se debe únicamente agregar un nuevo reconocedor específico a la jerarquía de reconocedores y configurar el *pipeline* de reconocimiento acordemente para datos provenientes de dicho sensor, así como también definir los extractores de características apropiados en caso de ser necesarios. También es importante notar que como parte de la inicialización del sistema, se ejecuta la fase inicial de entrenamiento para todos los reconocedores existentes, según ficheros de entrenamiento especificados en la configuración del sistema. Estos ficheros deben ser generados *offline* previo a la inicialización del sistema haciendo uso de las aplicaciones utilitarias *standalone* del módulo *GestureTrainers* incluidas como parte de la solución para dicho fin.

Por más detalles sobre el proceso general de reconocimiento de gestos no nativos referirse a la *Sección 2.2.5.1* del documento. De todas formas, cabe mencionar que si bien este módulo se incluye como parte de la solución propuesta, el reconocimiento de gestos no nativos no fue utilizado en la práctica para la aplicación desarrollada como prototipo.

5.4.2 *Core*

La capa *Core* está compuesta básicamente por cuatro grandes módulos, dos de los cuales son análogos a los ya descritos para la capa de acceso a datos. El módulo *Devices*, análogo al presentado para la capa *Data Access*, es responsable del mantenimiento y la identificación de los sensores registrados en el sistema, mientras que el módulo *Managers*, análogo al detallado para la capa *Data Access*, se encarga de gestionar el flujo de datos general dentro de esta capa. Adicionalmente, existen dos nuevos módulos específicos de la capa *Core*. Por un lado, el módulo *Users*, responsable del mantenimiento de la información de todos los usuarios reconocidos por el sistema. Y por el otro lado, el módulo *Interfaces*, responsable de brindar mecanismos de acceso a los servicios provistos por la capa *Core* a las capas que los requieran. En esta sección se presenta más en detalle cada uno de los cuatro módulos que conforman esta capa.

5.4.2.1. *Devices*

Análogamente a los componentes definidos para este módulo de la capa de acceso a datos, el módulo *Devices* de la capa *Core* contiene los componentes *LeapDevice*, *SingleKinectDevice* y *TouchKinectDevice*, tal como se ilustra en el diagrama de clases de la *Figura 75.* También en este caso el módulo está diseñado de forma tal que el manejo de los diferentes sensores sea lo más transparente posible para el usuario de esta capa, por lo que se incluye una clase base *Device* que permite almacenar toda la información de un sensor genérico, de la que descienden clases específicas para cada uno de los sensores particulares soportados por el sistema. Así, *LeapDevice* permite representar un sensor *Leap Motion* registrado en el sistema, *SingleKinectDevice* permite representar a un sensor *Microsoft Kinect* de usuario registrado en sistema, y por último, *TouchKinectDevice* permite representar un sensor *Microsoft Kinect* para interacción multitáctil registrado en el sistema. El diagrama de clases de la *Figura 75* ilustra la jerarquía descrita anteriormente.

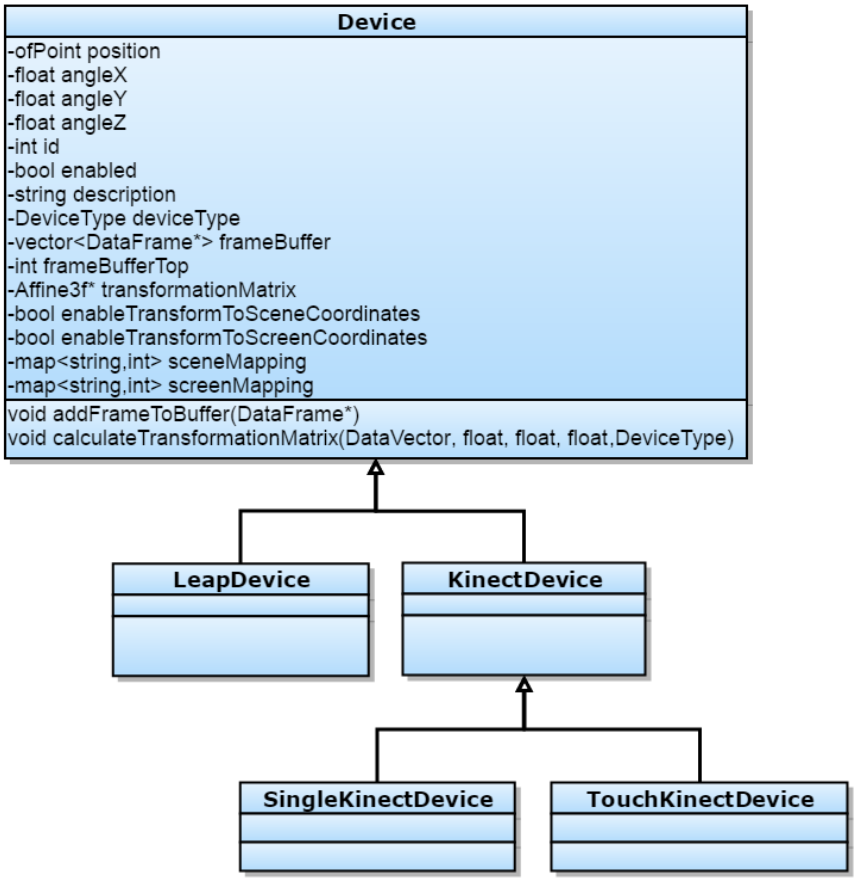


Figura 75: Diagrama de clases del módulo *Devices* de la capa *Core*

Como se puede apreciar, a diferencia de los presentes en la capa *Data Access*, en este caso los componentes específicos para cada sensor no cuentan con un método de obtención de datos, ya que llegan a través de la red luego de ser enviados por la capa de acceso a datos. En cambio, en esta capa estos componentes se encargan principalmente de almacenar toda la información particular para cada sensor, con el fin de poder procesar correctamente los datos provenientes de cada uno de ellos. Uno de los datos más importantes almacenados en estos componentes es la matriz de transformación para el sensor particular, la cual permite llevar a cabo la transformación de los datos representados en coordenadas de sensor a coordenadas del sistema, según se detalló en la *Sección 4.2* del *Capítulo 4*. Asociado a esto, también poseen las correspondencias para la transformación de los datos recibidos a coordenadas de pantalla y coordenadas de escena. La habilitación de estas transformaciones y los parámetros asociados se configuran en el fichero de configuración del sistema para esta capa (ver *Sección 5.6*).

5.4.2.2. *Users*

El módulo *Users* contiene las clases *User, Hand, Joint y Finger*, tal como se ilustra en el diagrama de clases de la *Figura 76*, las cuales permiten almacenar todos los datos referentes a los usuarios reconocidos por el sistema. El componente *User* permite representar a un usuario registrado en el sistema, es decir, un usuario para el cual llegó una notificación de reconocimiento de nuevo usuario desde la capa de acceso a datos. Así, mientras el usuario está activo (es decir, mientras no llegue una notificación desde la capa de acceso indicando que el usuario ha salido de la escena), el componente *User* es el responsable de mantener actualizados todos los datos del usuario a medida que se reciben notificaciones de actualización. Estas actualizaciones son enviadas desde la capa de acceso a datos, y permiten actualizar un conjunto de puntos de interés del esqueleto del usuario, cada uno de los cuales se denomina *joint*, tal como se describió en el *Capítulo 2*, y está representado por el componente *Joint* del diagrama de la *Figura 76*. Cada usuario tiene asociado quince puntos de interés o *joints*, así como también información específica a sus manos y dedos, almacenada en los componentes *Hand* y *Finger* respectivamente.

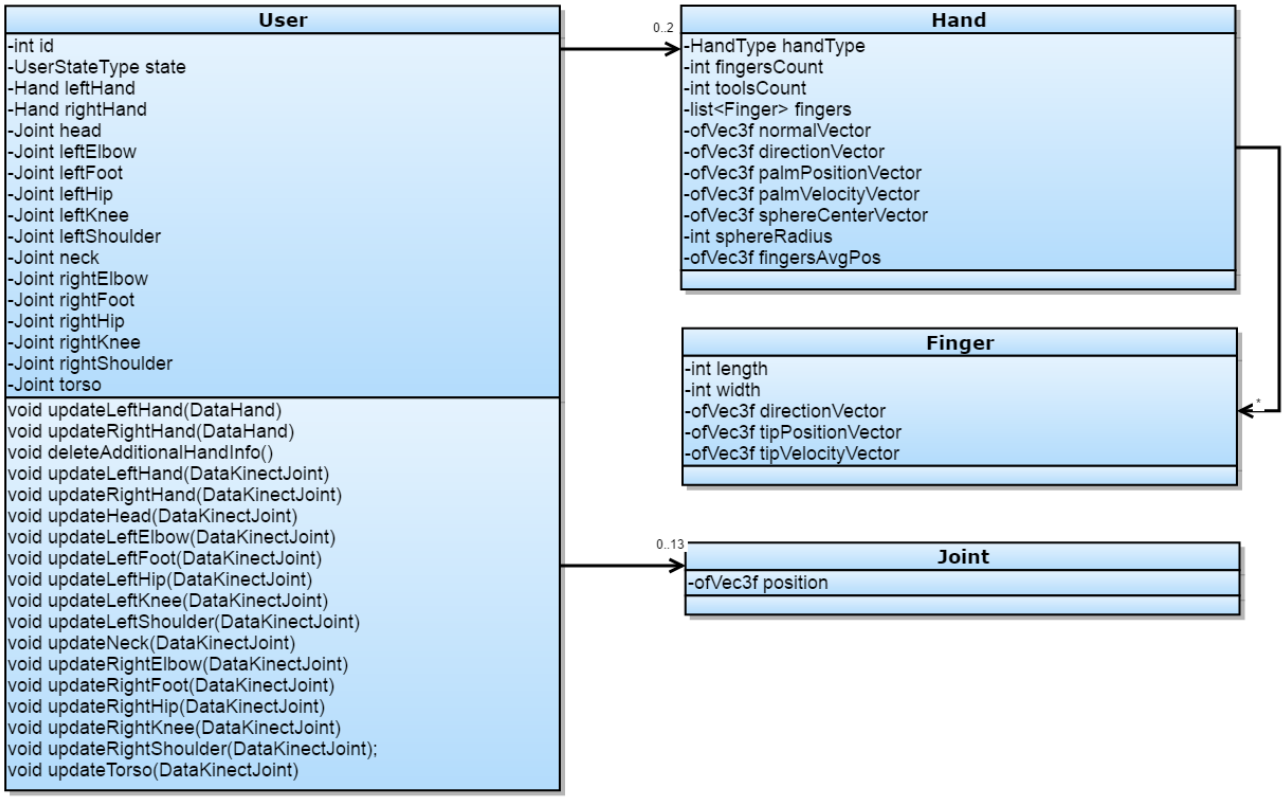


Figura 76: Diagrama de clases del módulo *Users*

Tanto el sensor *Leap Motion* como el sensor *Microsoft Kinect* de usuario configurados en la capa de acceso a datos son responsables de proporcionar la información necesaria para generar las notificaciones de actualización de usuario que llegan al *Core*. El sensor *Leap Motion* genera datos que permiten actualizar las manos y dedos del usuario, mientras que el sensor *Microsoft Kinect* genera datos que permiten actualizar los restantes *joints*. De esta forma, se logra obtener información precisa de todo el esqueleto del usuario, incluyendo cuerpo, manos y dedos, lo cual no sería posible únicamente con el sensor *Microsoft Kinect*, tal como fue descrito en el *Capítulo 2*.

En este sentido, existen dos eventos de actualización de usuario diferentes que pueden llegar al *Core*. Por un lado, una notificación de actualización de *joints*, y por otro, una notificación de actualización específicamente de manos y/o dedos. En el primer caso, es trivial actualizar acordemente los *joints* del componente *User* que representa al usuario en cuestión ya que el identificador que viaja en el mensaje de actualización coincide con el identificador de un usuario ya registrado en el sistema. En el segundo caso, la

primera vez que llega un evento de actualización de manos se desconoce a priori a qué usuario pertenece, por lo que no es posible actualizar su información. En este caso se debe realizar un proceso de emparejamiento para asociar el mensaje al usuario correcto y actualizarlo adecuadamente. Esto se lleva a cabo mediante una correlación entre las coordenadas del centro de la mano reportada en el mensaje y las coordenadas de las manos de los usuarios ya registrados. Aquel usuario para el cual se detecte que alguna de sus manos coincide (o está lo suficientemente cerca según un umbral preestablecido) con la contenida en el nuevo mensaje de actualización, será el usuario seleccionado a actualizar. Más específicamente, se convierten los diferentes pares de coordenadas a coordenadas globales de sistema y se realiza una diferencia de distancia espacial comparando dicha diferencia contra un margen aceptable bajo el cual se pueda afirmar que se trata del mismo punto en el espacio. De este modo es posible determinar a qué usuario pertenece una actualización de mano así como también, un gesto realizado con una o ambas manos. En la *Tabla 4* se incluye un pseudocódigo que ilustra a grandes rasgos el proceso de correlación de posiciones. Cabe mencionar que el comportamiento del componente *SceneManager* incluído en el pseudocódigo se detallará en la *Sección 5.4.2.4*; por ahora solo es suficiente tener en cuenta que este componente es uno de los componentes principales de esta capa, siendo responsable de gestionar todos los datos de la escena actual (usuarios, sensores, etc.).

```
while (true) {

    Message m = receiveFromDataAccess()
    UserInfo uinfo = m.getUserInfo()

    // Si se recibe notificación de nuevo usuario (usuario entró en escena)
    // Se registra al usuario en el sistema
    if (m.type == NEW_USER) {
         User u = new User(uinfo)
         sceneManager.registerUser(u)
    }
    // Si se recibe notificación de actualización de joints para un usuario ya existente
    // Se busca usuario por ID y se actualiza
    else if (m.type == USER_UPDATE) {
            int id = m.getUserID()
            User u = sceneManager.findUser(id)
            u.updateJoints(uinfo) // Se actualiza solo los joints (sin incluir manos/dedos)
    }
    // Si se recibe actualización de manos o dedos, se ejecuta el proceso de matcheo
    // Se obtiene el usuario que mejor coincide con la información contenida en la
    // notificación
    else if (m.type == HAND_UPDATE) {
          User bestMatch
          foreach (u in Users) {
               if (u.match(uinfo)) {
                   if (betterMatch(u,bestMatch))
                      bestMatch = u
               }
          }
          // Se actualizan las manos/dedos para el usuario que mejor coincidió con la información
          bestMatch.updateHands(uinfo)
          exit
    // Si se recibe notificación de usuario perdido (usuario se fue de escena)
    // Se elimina al usuario del sistema
    } else if (m.type == USER_LOST) {
           int id = m.getUserID()
           sceneManager.deleteUser(id)
    }
```

```
}
```

Tabla 4: Pseudocódigo del proceso de actualización ante un paquete recibido

5.4.2.3. *Managers*

Análogamente a la capa de acceso a datos, el módulo *Managers* de la capa *Core* es el módulo que contiene los diferentes manejadores, es decir, aquellos componentes encargados de gestionar el flujo de ejecución general de la capa (configuraciones, envío y recepción de información hacia otras capas, etc). El módulo contiene básicamente cinco manejadores, representados por las clases *ConfigurationManager*, *DeviceManager, PacketManager*, *GestureManager* y *SceneManager* tal como se ilustra en el diagrama de clases de la *Figura 77.* Los primeros tres componentes tienen las mismas responsabilidades que en la capa *Data Access*, mientras que el *GestureManager* se encarga de gestionar todos los gestos reconocidos por el sistema, y el *SceneManager* se encargada de mantener actualizada toda la información referente a la escena (usuarios activos, sensores activos. etc.) a partir del procesamiento de los diferentes mensajes recibidos.

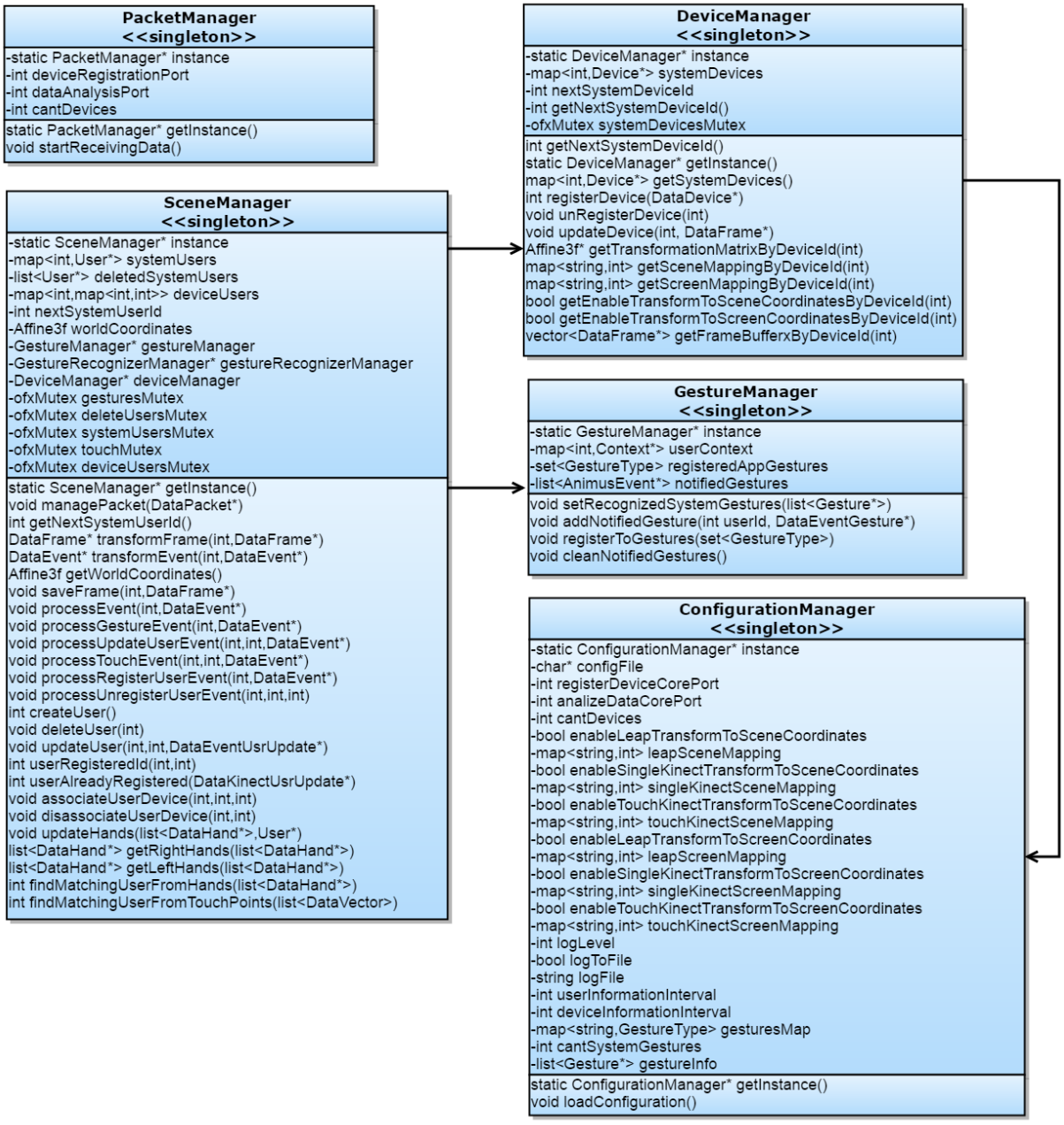


Figura 77: Diagrama de clases del módulo *Managers*

De forma análoga al comportamiento ya descrito para la capa de acceso a datos, durante la inicialización de la capa *Core*, el *ConfigurationManager* carga todos los parámetros de configuración de esta capa desde un fichero de configuración del sistema y los mantiene almacenados para que puedan ser accedidos en cualquier momento y desde cualquier otro módulo que los requiera. Por más detalles sobre los parámetros configurables para esta capa referirse a la *Sección 5.6*.

La clase *DeviceManager* se encarga de registrar los sensores en el sistema cuando se recibe un mensaje de registro desde la capa *Data Access*, asignándole un identificador global. Como se mencionó anteriormente, una vez registrado el sensor en el sistema y calculado su identificador, es enviado hacia la capa de acceso a datos para informarlo a modo de poder correlacionar los mensajes de datos posteriores entre ambas capas. A su vez, este componente tiene la responsabilidad de almacenar y aplicar las transformaciones configuradas para cada sensor cada vez que se requiera realizar una conversión entre sistemas de coordenadas, ya sea a coordenadas de mundo, coordenadas de pantalla o coordenadas de escena.

El componente *PacketManager* es el responsable de la comunicación entre la capa *Core* y la capa de acceso a datos. Con el fin de mejorar el rendimiento del sistema, esta comunicación se lleva a cabo mediante dos hilos independientes representados por las clases *DeviceRegistrationThread* y *DataAnalysisThread*, tal como se puede apreciar en la *Figura 78*. A grandes rasgos, el hilo *DeviceRegistrationThread* está a la escucha de mensajes de registro de sensores desde la capa *Data Access*, mientras que el hilo *DataAnalysisThread* está a la escucha de paquetes de datos proveniente desde la capa de acceso a datos para los sensores registrados. Ambos hilos son instanciados durante la inicialización del *Core*, y en conjunto permiten implementar la comunicación de doble banda o *dual channel* ya descrita en la *Sección 5.2*, ya que cada uno utiliza una conexión diferente para la recepción de mensajes de registro y de datos en paralelo. Esta ejecución en paralelo permite, por ejemplo, que en la medida que se finalice el registro de un sensor sea posible obtener datos de forma inmediata, sin la necesidad de esperar a que finalice el registro de los otros sensores eventualmente configurados. A su vez, se obtiene un mejor diseño pudiendo gestionar ambos hilos de forma independiente.

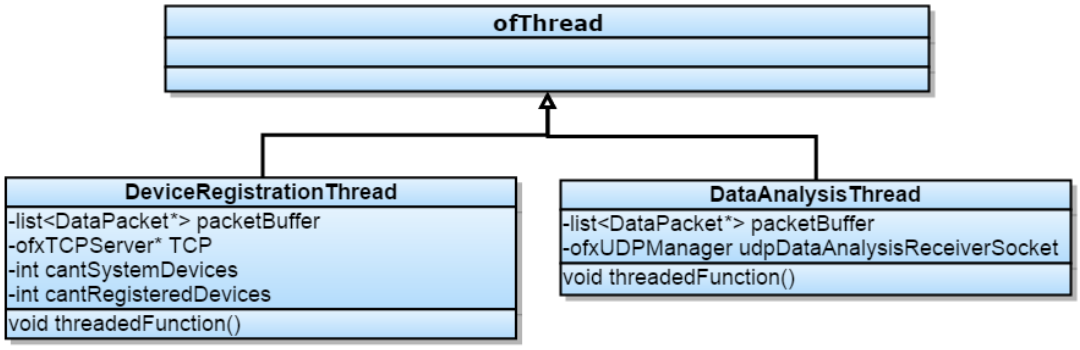


Figura 78: Diagrama de clases del módulo *Threads*

Más específicamente, el hilo *DeviceRegistrationThread* estará escuchando mensajes de registro de sensores provenientes de las diferentes instancias de la capa *Data Access* mediante un *socket TCP*. Una vez establecida la conexión en el puerto configurado, se obtienen los datos del mensaje de registro recibido y se envían al *DeviceManager* para su registro en el sistema. El proceso de registro, como ya se mencionó, implica la generación de un identificador global para el sensor en el sistema, y el envío de esta información hacia la capa *DataAccess* con el fin de poder correlacionar los futuros mensajes de datos que esta capa envíe. Este hilo es finalizado una vez se registra la cantidad total de sensores configurados, evitando *polling* innecesario y mejorando el rendimiento global del sistema.

```
int cantSensores = ConfigurationManager.getCantSensores()
int sensoresReg = 0

// Se reciben mensajes de registro mientras no se registren todos los sensores configurados
while (sensoresReg < cantSensores) {

    // Se recibe el mensaje y se registra el sensor en el sistema
    Message m = socketTCP.receive()
    int id = DeviceManager.registerDevice(m)

    // Si se registró el sensor correctamente se informa el ID a la capa de acceso a datos
    if (isValid(id))
        if (socketTCP.send(id))
            sensoresReg++
}
```

Tabla 5: Pseudocódigo del proceso de registro de un sensor

Por otro lado, el hilo *DataAnalysisThread* estará escuchando mensajes de datos provenientes de las diferentes instancias de la capa *Data Access* mediante un *socket UDP*. Una vez se reciben datos en el puerto configurado, estos son delegados inmediatamente al *SceneManager* para su análisis y procesamiento. Como se detallará, el *SceneManager* es el componente encargado de extraer la información de actualización de usuarios o reconocimiento de gestos contenida en el mensaje y procesarla acordemente para luego actualizar la información de los usuarios registrados o bien almacenar notificaciones a ser luego informadas a la capa *Application*. A diferencia del hilo de registro de sensores, el hilo *DataAnalysisThread* estará siempre activo con el fin de atender en todo momento la recepción de información proveniente de la capa de acceso a datos.

La clase *SceneManager* es el manejador principal de la capa *Core*, ya que es a quien llegan los paquetes recibidos desde el hilo de recepción de paquetes para su procesamiento. El procesamiento de cada mensaje consiste primeramente en aplicar todas las transformaciones habilitadas para el sensor al que corresponda el evento, para luego discernir qué acción tomar según el tipo de evento que se haya recibido. En este sentido, existen básicamente cuatro caminos que el flujo puede seguir, ya incluídos como parte del pseudocódigo de la *Tabla 4* pero descritos con mayor detalle a continuación:

1. Si la notificación se corresponde a la detección de un nuevo usuario (mensaje de registro de usuario), éste es agregado al mapa de usuarios globales del sistema como una instancia del componente *User* ya descrito, asignándole un identificador global y rellenando toda la información de usuario con que se cuenta al momento. Adicionalmente, se asocia el nuevo usuario con el sensor que lo reconoció con el fin de simplificar futuras actualizaciones correspondientes al mismo sensor.
2. Si el evento es una actualización para un usuario, se busca al usuario en el mapa de usuarios, eventualmente ayudados por la asociación usuario-sensor existente, y luego se actualiza su información. Como el sensor *Microsoft Kinect,* ya se posee un identificador de usuario asociado para este sensor (gracias al mapeo usuario-sensor), es trivial encontrar a qué usuario del sistema se corresponde la actualización recibida.
3. Si el evento es de actualización de las manos y dedos de los usuarios dado por un sensor *Leap Motion*, no se envía un identificador de usuario como parte de la notificación de actualización. Dada esta complejidad, se debe realizar un procesamiento adicional consistente en comparar la información de cada mano de los usuarios registrados en el sistema con la información proporcionada por el respectivo evento de actualización (para un mayor entendimiento de este punto referirse a *Tabla 4*). Como resultado de este proceso de emparejamiento se determina a qué usuario del sistema se corresponde la notificación y se agrega la relación usuario-sensor

para evitar comprar nuevamente en una nueva actualización de manos contra todos los usuarios del sistema.

4. Si el evento indica que se dejó de reconocer un usuario se elimina toda la información asociada a dicho usuario.

Además de los eventos puramente relacionados con información de usuario (registro y actualización), también pueden recibirse mensajes que contengan eventos de reconocimiento de gestos por parte de los usuarios. En este caso, el componente *SceneManager* obtiene y actualiza la información del usuario que lo realizó, eventualmente ayudado por el mapeo usuario-sensor, y envía esta información junto a la información del gesto particular recibido al componente *GestureManager* para su procesamiento. Con toda esta información, este componente es responsable de generar una representación del gesto acorde a las estructuras de datos especificadas por la *API* del *framework* (a detallarse en *Sección 5.4.2.3*), y agregarlo a la lista de gestos a notificar a la aplicación. De este modo, cada vez que la capa *Application* realice el ciclo de actualización como parte del procesamiento de cada *frame*, recibirá las notificaciones de gestos reconocidos por parte del *Core*.

5.4.2.4. *Interfaces*

Este componente tiene a cargo dos tareas esenciales del *Core* del sistema. Por un lado, define el *API* de servicios que proverá el *framework* a la capa *Application*, y por otro lado, se encarga de hacer llegar las diferentes notificaciones ya procesadas y reconocidas por el *Core* a la aplicación que se está ejecutando en la capa superior. En este sentido, se puede decir que el módulo *Interface* es el puente que habilita la comunicación entre las capas superiores *Core* y *Application*. Como se mencionaba anteriormente, la forma en que se comunican estas dos capas es mediante comunicación directa, es decir, mediante recursos compartidos en tiempo de ejecución. Más específicamente, al inicializar el sistema el nodo *Core* almacena una referencia a la aplicación particular que está ejecutando como parte de la capa *Application*, y es mediante esta referencia que se notifican los diferentes eventos a la capa *Application*, tal como se muestra en el pseudocódigo de la *Tabla 6*. Si bien esta restricción hace que las capas *Core* y *Application* estén de cierta forma ligadas en la arquitectura del sistema en el sentido de que no pueden ejecutar en nodos independientes, esto permite el envío de notificaciones de forma muy rápida (memoria en un mismo *host* físico) desde el *Core* hacia la aplicación según se vayan procesando.

```
while(true) {

    // Se obtienen notif. de gestos pendientes de envío y se notifican a la aplicación asociada
    // Notar que la referencia a la aplicación fue almacenada en el Core durante su
    // inicialización
    AnimusEvent[] eventGestures = GestureManager.getNotifiedGestures()
    foreach (e in eventGestures) {
         switch (e.type)
         {
            case ANIMUS_3D_PINCH_HAND: app->pinchHand3DGestureDetected(userId,e)
            case ANIMUS_3D_ZOOM_HAND: app->zoomHand3DGestureDetected(userId,e)
            …
            case ANIMUS_TOUCH_TAP_FINGER: app->tapTouchFingerGestureDetected(userId,e)
         }
    }

    // Se obtienen act. de usuario pendientes de envío y se notifican a la aplicación asociada
    AnimusEvent[] userUpdates = SceneManager.getNotifiedUserUpdates()
    foreach (u in userUpdates) {
         switch (u.type)
         {
```

```
            case ANIMUS_NEW_USR_STATE: app->userDetected(u)
            case ANIMUS_UPDATE_USR_STATE: app->userUpdate(u)
        }
    }
}
```

Tabla 6: Pseudocódigo del proceso de notificación de eventos desde el *Core* a la aplicación

El módulo *Interfaces* está compuesto por las clases *IAnimusCore* e *IAnimusApplication*, tal como se ilustra en el diagrama de clases de la *Figura 79*, las cuales en conjunto definen la lista completa de servicios brindados por el *framework*. *IAnimusCore* provee servicios relacionados al flujo general del sistema para su correcto funcionamiento, brindando las vías para dar comienzo o fin al *Core* del sistema y registrar los gestos a los cuales se suscribe la aplicación. Por otra parte, *IAnimusApplication* brinda todos los servicios correspondientes a notificaciones de usuarios reconocidos, notificaciones de actualización de usuarios y notificaciones de reconocimiento de gestos. Así, cualquier aplicación que desee recibir notificaciones desde el sistema debe implementar la interface *IAnimusApplication*, de modo que el *Core* pueda hacerle llegar los diferentes eventos generados, tal como se ilustró en el pseudocódigo de la *Tabla 6*.

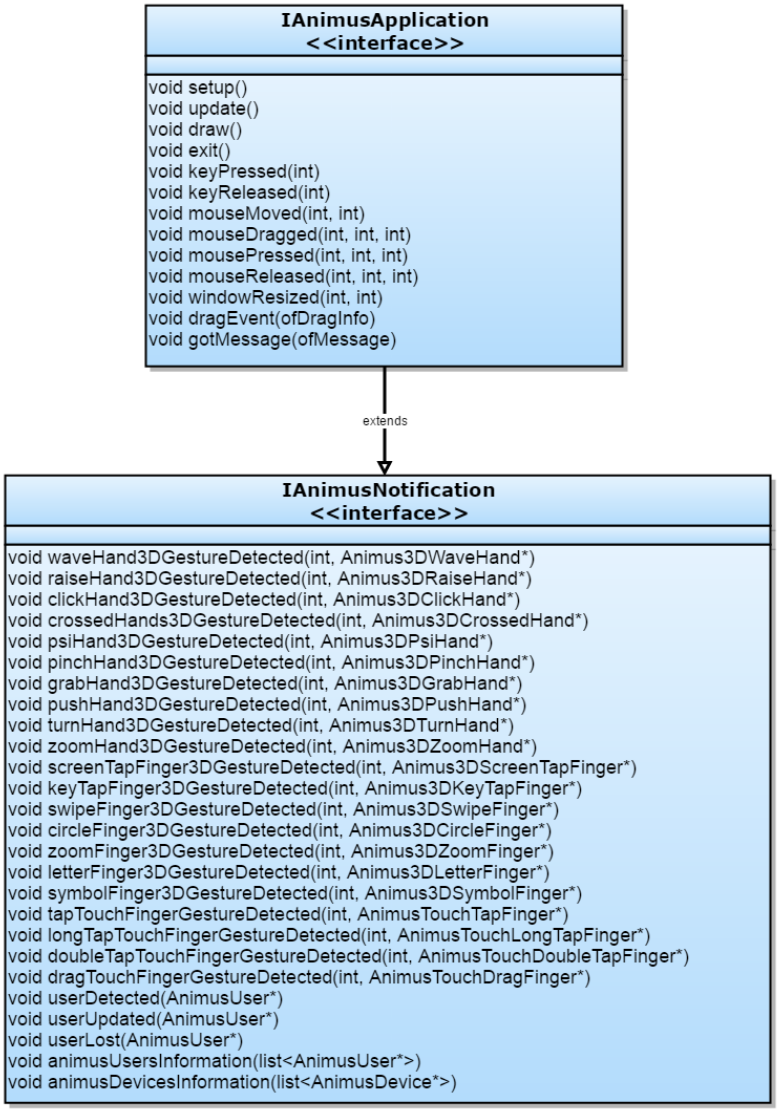


Figura 79: Diagrama de clases del módulo *Interfaces*

Dentro del módulo *Interfaces* también se encuentran los submódulos *Devices*, *Gestures* y *Users*, incluídos en el diagrama de clases de la *Figura 80.* Estos módulos contienen todas las estructuras de datos

necesarias para representar los diferentes eventos de actualización de usuarios y reconocimiento de gestos que habilitan la comunicación entre el *Core* y la *Application*. Si bien, como ya se mencionó, la comunicación en este caso es directa dentro de un mismo *host*, se deben definir estructuras de datos acordes a modo de unificar la comunicación entre ambas capas y obligar a que los datos se representen de la misma forma sin inconsistencias. Estas estructuras de datos se definen como parte del *API* de servicios provistas por el *framework* a las aplicaciones presentada en detalle en el *Anexo M*.

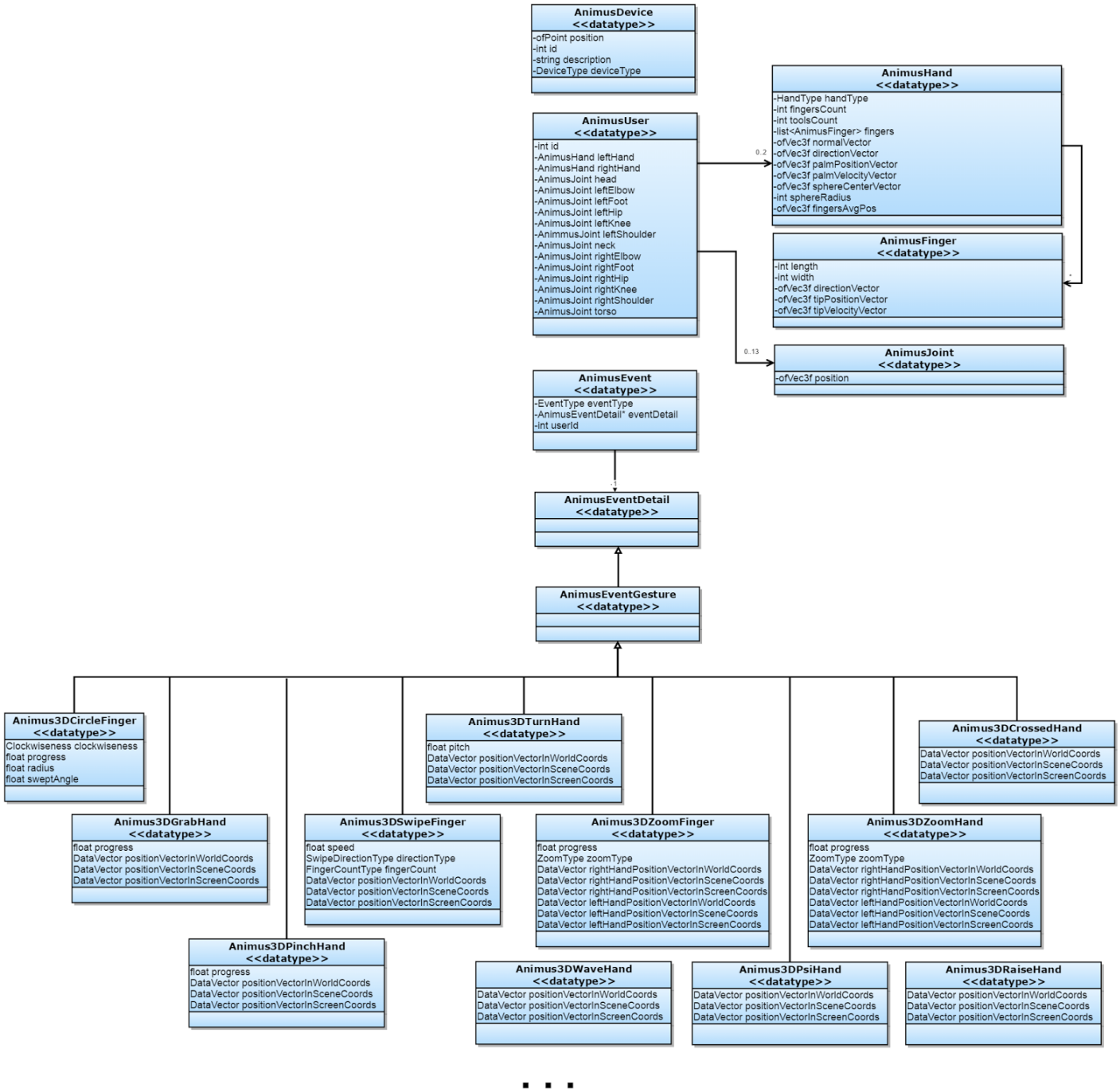


Figura 80: Diagrama de clases (parcial) de los submódulos *Device*, *Events*, *Gesture* y *Users*

5.4.3 *Application*

La capa *Application* está compuesta básicamente por dos componentes. Por un lado, el componente de inicio *main* que permite inicializar el sistema, y por otro lado la aplicación cliente particular (junto con

todos los componentes y clases adicionales que requiera para su funcionamiento). En la *Figura 81* se presenta el diagrama de clases de la capa *Application* incluyendo las clases y atributos principales, considerando que la aplicación está construida haciendo uso del *toolkit* de programación *openFrameworks*, ya detallado en el *Capítulo 2*. Cabe mencionar que la aplicación perfectamente podría utilizar otro *toolkit* de programación diferente, pero lo importante para poder consumir los servicios del *framework* de la solución propuesta es implementar la interface *IAnimusApplication* que define todas las posibles notificaciones generadas por el sistema como ya se describió en la *Sección 5.4.2.3*.

Figura 81: Clases y atributos principales de la *Application*

5.4.3.1. *Main*

La clase *main* debe estar presente en cada aplicación cliente de la capa *Application* ya que es la encargada de instanciar a la clase particular que representa la aplicación, registrar los gestos a soportar y enviar la señal de inicialización del sistema indicando que comience a brindar los diferentes servicios. En la *Tabla 7* se incluye el pseudocódigo correspondiente al programa principal de una aplicación genérica que hace uso de los servicios provistos por el *framework*. Como se puede apreciar, se utiliza la interface *IAnimusCore* descrita en la *Sección 5.4.2.3* para registrar los gestos a considerar e inicializar el sistema.

```
main() {

    // Aplicación de usuario que implementa la API del sistema
    IAnimusApplication* userApp = new UserApplication()

    // Se registran los gestos a ser soportados por la aplicación
    IAnimusCore::registerToGesture(ANIMUS_3D_PINCH_HAND)
    IAnimusCore::registerToGesture(ANIMUS_3D_ZOOM_HAND)
    …
    IAnimusCore::registerToGesture(ANIMUS_TOUCH_TAP_FINGER)

    // Se envía la señal de inicialización del sistema
    IAnimusCore::startAnimus(userApp);
}
```

Tabla 7: Pseudocódigo del flujo principal de la capa *Application*

5.4.3.2. User *application*

Este módulo puede estar compuesto por múltiples componentes que permitan representar los diferentes elementos de la aplicación particular que se desee ejecutar en esa capa. De todas formas, si se desea hacer uso de los servicios provistos por el *framework*, uno de estos componentes debe implementar la interfaz de sistema *IAnimusApplication* expuesta por el *framework*, incluyendo todos los métodos expuestos por la interfaz junto con la lógica particular que a cada a uno se desee dotar según las necesidades de la aplicación. Este componente será luego el utilizado para instanciar la aplicación particular e inicializar el sistema según se describió en el pseudocódigo de la *Tabla 7*. De esta forma, la lógica de cada uno de los métodos de interfaz contiene las instrucciones de actualización de estado que se deben ejecutar en la aplicación para la notificación particular recibida. De esta forma, por cada gesto al que se ha registrado la aplicación recibirá la notificación por parte del *Core* en cada uno de los métodos o *handlers* correspondientes, así como también las notificaciones genéricas de registro, pérdida y actualización de usuarios generados nativamente por el sistema tal como se detalló en la *Sección 5.4.2.3*. En la *Tabla 8* se incluye una definición simplificada de una clase genérica de aplicación que hace uso de los servicios provistos por el *framework*. De este modo es posible construir cualquier tipo de aplicación que haga uso tanto de vías de interacción multitáctiles como gestuales.

```
class UserApplication implements IAnimusApplication {

    // Métodos que permiten definir el flujo general de la aplicación
    void setup() { … }
    void update() { … }
    void draw() { … }

    // La aplicación de usuario debe implementar handlers para cada notif. de IAnimusApplication
    // Cada handler contiene la lógica de aplicación a ejecutar según la notificación recibida
    UserApplication::pinchHand3DGestureDetected(userId,e) { … }
    UserApplication::zoomHand3DGestureDetected(userId,e) { … }
    …
    UserApplication::tapTouchFingerGestureDetected(userId,e) { … }
}
```

Tabla 8: Definición de la clase de aplicación de usuario

5.5. Protocolo de comunicación

Como se ha mencionado anteriormente, los nodos que componen la arquitectura se encuentran distribuidos en diferentes nodos físicos, por lo que es necesario que se comuniquen a través de la red. En particular, la comunicación a través de la red se da entre las capas *Data Access* y *Core* como ya se ha detallado (referirse a *Figura 70*). Para ello, fue necesaria la definición de un protocolo de comunicación al que se le denominó *ANIMuS Framework Protocol* (*AFP*), que establece el formato y el flujo que deben cumplir los mensajes que se envían entre estas capas de la arquitectura diseñada. Un mensaje AFP es un mensaje de largo variable, ya que la cantidad de datos que se envían depende fuertemente del tipo particular de datos que viajan. Como parte del mensaje viaja o bien la estructura de datos *DataPacket* o bien la estructura de datos *DataEvent*, ambas incluidas en el diagrama de clases de la *Figura 82*.

Como se detalla más adelante en esta misma sección, la estructura *DataPacket* permite representar información de registro de sensores o información de datos crudos obtenidos por los sensores, mientras que la estructura *DataEvent* permite representar ya sea un evento de actualización de usuario (*joints* o manos) o un evento de reconocimiento de gestos proveniente de alguno de los diferentes sensores.

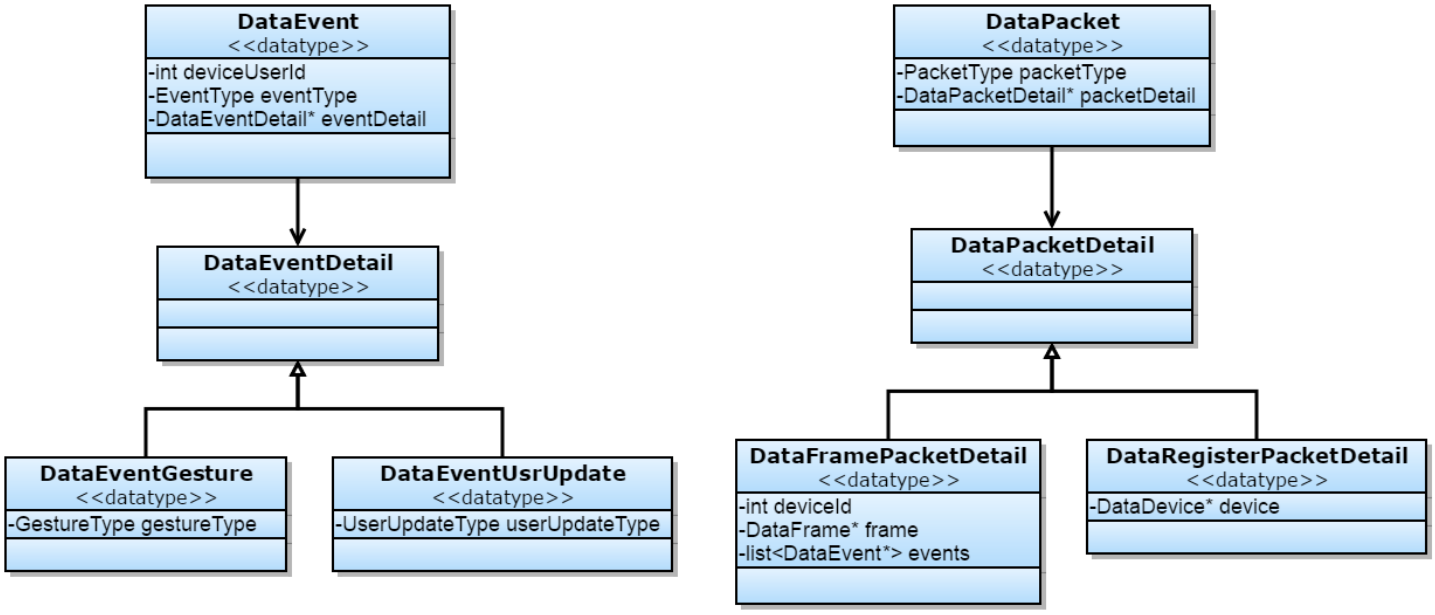


Figura 82: Diagrama parcial de las posibles estructuras que pueden viajar en un mensaje *AFP*

Así, el flujo general definido por el protocolo para el correcto funcionamiento del sistema es el ilustrado en el diagrama de secuencia de la *Figura 83*. El cliente en este caso se corresponde con la capa de acceso a datos *Data Access*, que es quien se conecta al otro extremo para comenzar el intercambio, mientras que el servidor se corresponde con la capa *Core*, ya que es quien está en modo *always-on* escuchando nuevas conexiones. Inicialmente el cliente envía *N* mensajes de registro de sensores al servidor mediante *TCP/IP* (ya que es de interés que no se pierdan los paquetes de registro), uno para cada sensor que se desee registrar. Para cada uno de estos mensajes, el servidor responde enviando un identificador para el sensor registrado. Una vez recibido el identificador por parte del cliente, se está en condiciones de comenzar a enviar datos provenientes de los diferentes sensores registrados, es decir, mensajes de actualización, mensajes de reconocimiento de gestos tridimensionales o interacción multitáctil. Estos mensajes de datos viajan sobre *UDP/IP*, dado que la pérdida de alguno de ellos no es crítica para el correcto funcionamiento del sistema, y además el protocolo establece que el nodo *Core* no debe enviar ninguna respuesta ante estos mensajes.

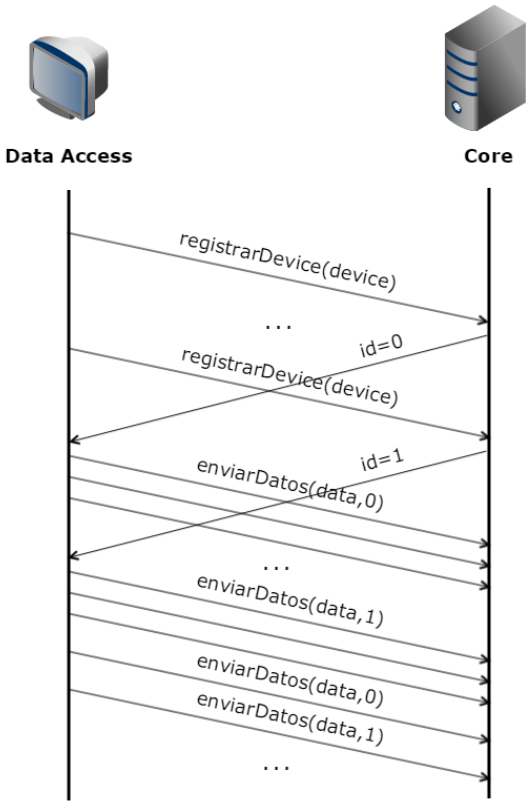


Figura 83: Diagrama de secuencia del protocolo *AFP*

En lo que respecta a las estructuras específicas utilizadas para generar los diferentes mensajes, la estructura *DataPacket* tiene una organización jerárquica, por lo que de ella descienden estructuras de datos adicionales que permiten representar mensajes más específicos. Esta jerarquía establece que un *DataPacket* contiene un *DataPacketDetail*, que puede ser o bien un *DataFramePacketDetail* o bien un *DataRegisterPacketDetail*, como se puede observar en la *Figura 84*.

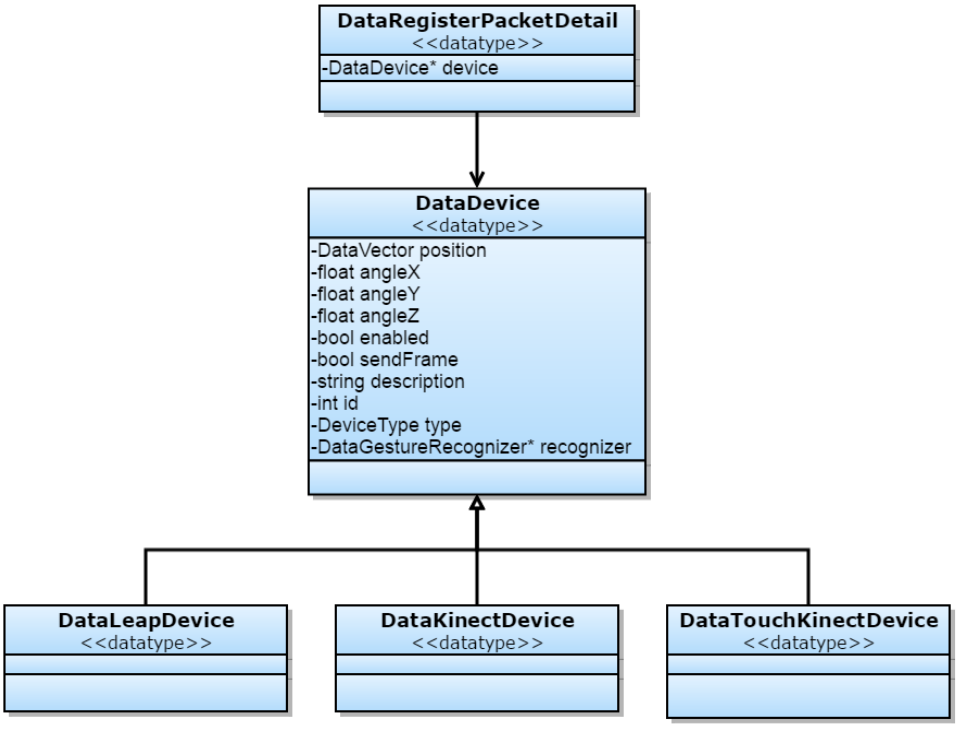


Figura 84: Diagrama de clases de la estructura de datos *DataRegisterPacketDetail*

Un *DataRegisterPacketDetail* a su vez, contiene un *DataDevice* que permite representar toda la información de un sensor particular como se muestra en la *Figura 84*. Así, este último puede tomar la forma de un *DataLeapDevice*, *DataKinectDevice* o un *DataTouchKinectDevice* según el sensor que corresponda. Al momento de registrar un sensor por parte de la capa de acceso a datos en el *Core*, el mensaje *AFP* de registro viaja con el campo de datos *data* de la *Figura 82* relleno con una estructura de este tipo más específico, conteniendo todos los datos relevantes del sensor a registrar.

A modo de contrastar lo anterior con el flujo general definido para el protocolo, como ya se mencionó, en la práctica el protocolo de comunicación establece que primero se debe registrar el sensor particular antes de que comience a generar datos. Para ello, se envía un mensaje *AFP* sobre *TCP/IP* conteniendo un *DataRegisterPacketDetail*, y se especifica en el campo *type* el tipo de sensor que se está registrando. Dentro del *DataRegisterPacketDetail* se envía toda la información sobre el sensor a registrar, incluyendo el *DataDevice* específico que corresponda.

Por otra parte, un *DataFramePacketDetail* contiene un *DataFrame* que permite representar toda la información correspondiente a un *frame* específico para un sensor particular como se muestra en la *Figura 85*. Un *frame* puede contener los píxeles de un mapa de profundidad o los píxeles de una imagen *RGB* en el caso de un sensor *Microsft Kinect*, o los datos crudos de manos y dedos en el caso de un sensor *Leap Motion*. De este modo, un *DataFrame* puede tomar la forma de un *DataLeapFrame*, *DataKinectFrame* o *DataTouchKinectFrame* según el sensor que corresponda. Como se observa en el diagrama, cada una de estas estructuras contiene toda la información relevante provista por cada *frame* generado por un sensor particular. De todos modos, como ya se ha mencionado anteriormente, dadas las limitantes de enviar tales cantidades de datos a través de la red, si bien estas estructuras de datos están definidas como parte de la definición del protocolo *AFP*, no son utilizadas en la práctica.

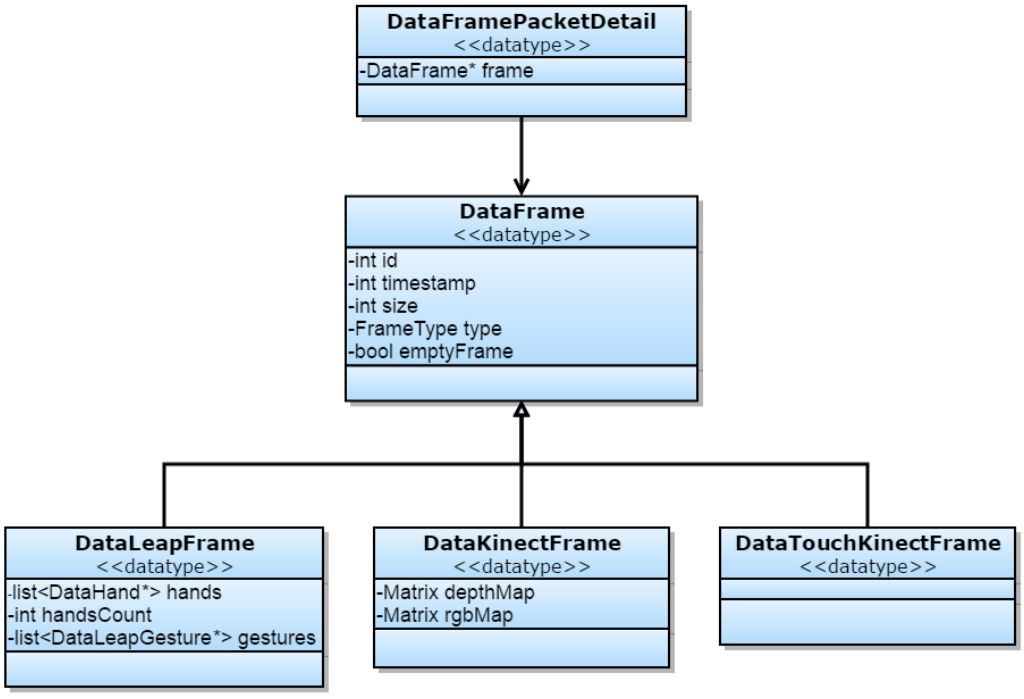


Figura 85: Diagrama de clases de la estructura de datos *DataFramePacketDetail*

En lo que respecta a la estructura *DataEvent*, contiene el identificador global del sensor que generó el evento de forma que pueda ser identificado en el extremo receptor, así como también, el tipo de evento que se está informando. Además, está organizada jerárquicamente de forma similar a *DataPacket*, por lo que de ella descienden estructuras de datos adicionales que permiten representar eventos más específicos. Esta jerarquía establece que un *DataEvent* contiene un *DataEventDetail*, que puede ser o bien un *DataEventUsrUpdate, o* un *DataEventGesture*, como se puede observar en la *Figura 82*. Un

DataEventUsrUpdate a su vez puede tomar la forma de un *DataLeapUsrUpdate* o un *DataKinectUsrUpdate* según el sensor que corresponda, tal como se aprecia en la *Figura 86*. Así, un *DataLeapUserUpdate* permite representar la actualización de posiciones de manos y dedos de los usuarios, mientras que un *DataKinectUsrUpdate* permite representar la actualización de posiciones de los diferentes *joints* del esqueleto del usuario. Cabe mencionar que las actualizaciones de posiciones multitáctiles son reportadas mediante eventos de gesto reconocido, como se detallará más adelante en esta misma sección.

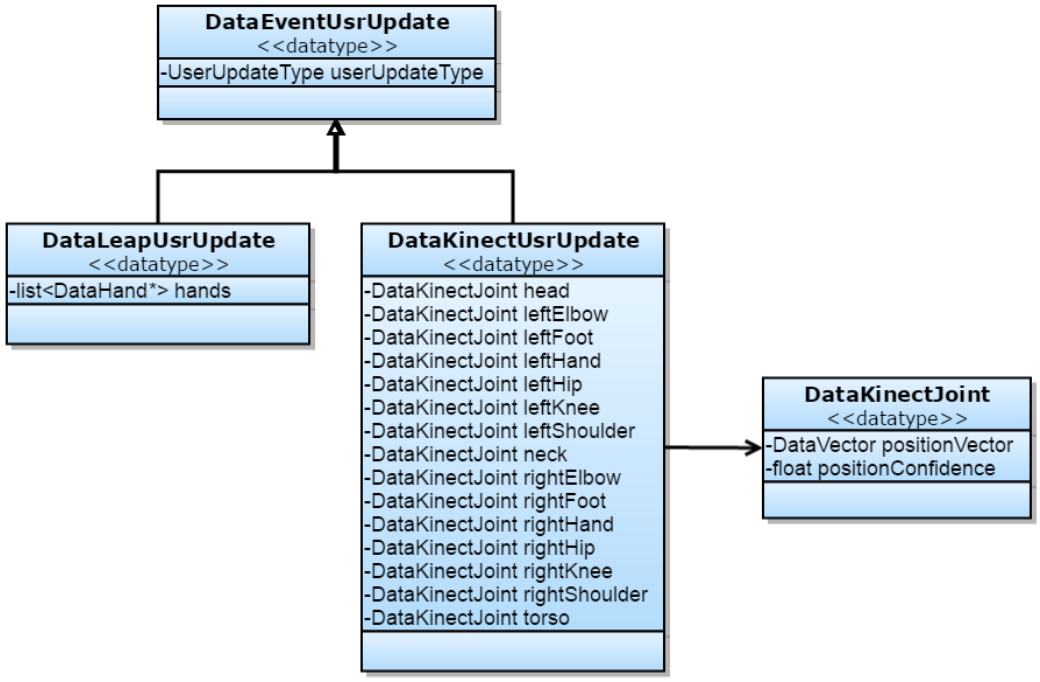


Figura 86: Diagrama de clases de la estructura de datos *DataEventUsrUpdate*

Por otra parte, un evento de tipo *DataEventGesture* permite representar un mensaje de reconocimiento gestual por parte de alguno de los sensores. Por esto, puede tomar la forma de un *DataLeapGesture*, *DataKinectGesture* o *DataTouchKinectGesture* según el sensor del cual proviene, tal como se muestra en la *Figura 87*.

A su vez, para cada sensor existen múltiples gestos diferentes, por lo que cada uno de ellos es representado por una estructura de datos específica que contiene toda la información para el gesto en cuestión, tal como se observa en la *Figura 87*. Así, por ejemplo, un *DataLeapGesture* puede tomar la forma de un *DataLeapPinchGesture*, el cual incluye toda la información relevante para informar un gesto de *pinch* reconocido (referirse al *Capítulo 2*), o de cualquiera de los demás posibles gestos (nativos o no) reconocidos con datos provenientes del sensor *Leap Motion*. De la misma forma, un *DataKinectGesture* puede tomar la forma, por ejemplo, de un *DataKinectRaiseHandGesture* que incluye toda la información correspondiente al gesto de *raise hand* reconocido (referirse a *Capítulo 2*), y un *DataTouchKinectGesture* puede tomar la forma, por ejemplo, de un *DataTouchTapGesture* que incluye toda la información correspondiente a un evento de *tap* o presión táctil sobre la superficie. Es preciso observar en este punto que la interacción multitáctil se informa mediante eventos de reconocimiento utilizando esta estructura de datos, existiendo estructuras similares para representar eventos de *double tap* (*DataTouchDoubleTapGesture*), *long tap* (*DataTouchLongTapGesture*), etc.

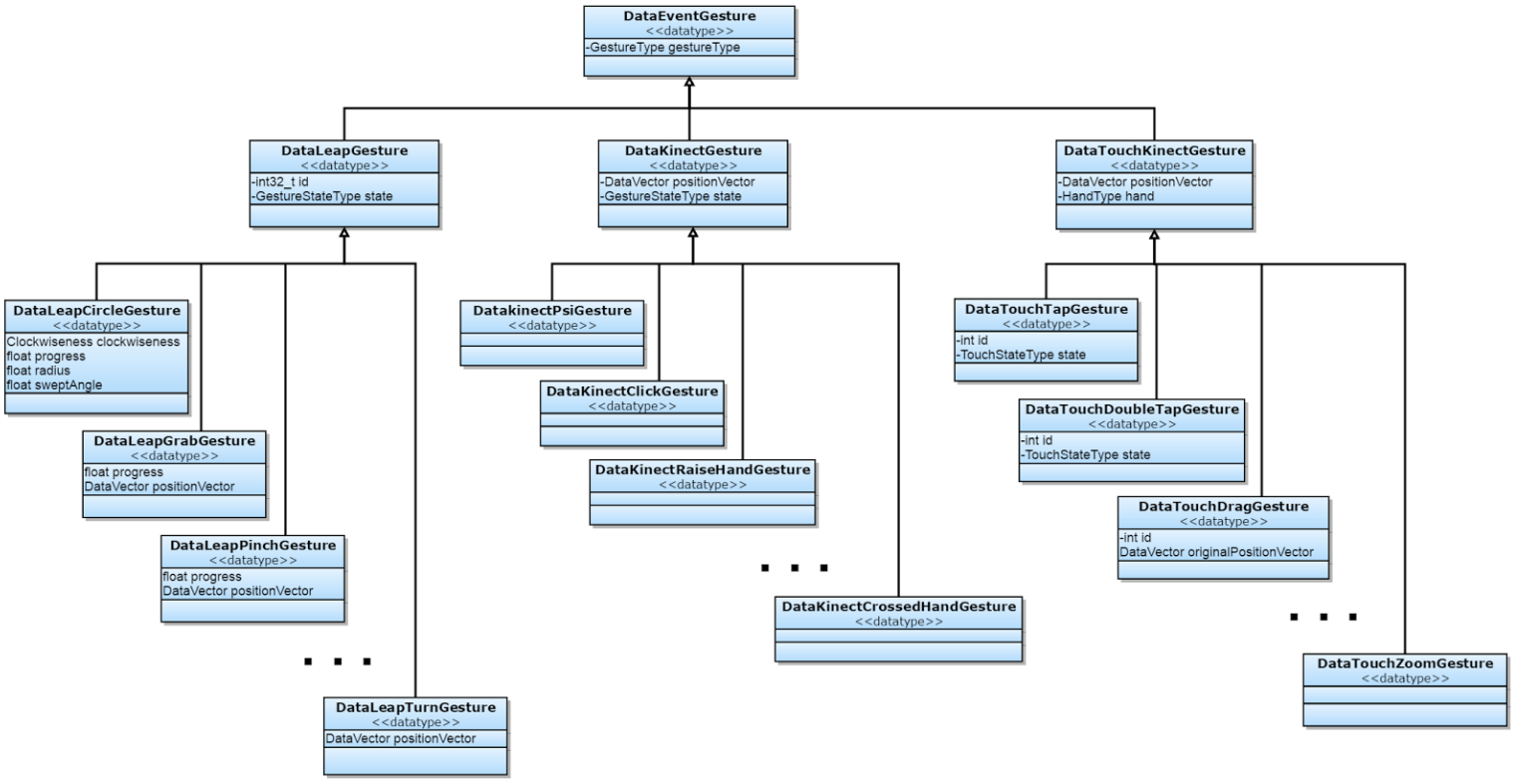


Figura 87: Diagrama de clases de la estructura de datos *DataEventGesture*

En las tablas *Tablas 9*, *Tabla 10* y *Tabla 11*, se listan todas las estructuras de datos específicas utilizadas para representar los diferentes gestos soportados por cada uno de los sensores.

DataLeapGesture	
Nombre	**Descripción**
DataLeapCircleGesture	Estructura de datos utilizada para representar un gesto *circle* reconocido por el sensor *Leap Motion*. Desciende de la estructura base *DataLeapGesture*.
DataLeapGrabGesture	Estructura de datos utilizada para representar un gesto *grab* reconocido por el sensor *Leap Motion*. Desciende de la estructura base *DataLeapGesture*.
DataLeapKeyTapGesture	Estructura de datos utilizada para representar un gesto *key tap* reconocido por el sensor *Leap Motion*. Desciende de la estructura base *DataLeapGesture*.
DataLeapLetterGesture	Estructura de datos utilizada para representar un gesto *letter* reconocido por el sensor *Leap Motion*. Desciende de la estructura base *DataLeapGesture*.
DataLeapPinchGesture	Estructura de datos utilizada para representar un gesto *pinch* reconocido por el sensor *Leap Motion*. Desciende de la estructura base *DataLeapGesture*.
DataLeapPushGesture	Estructura de datos utilizada para representar un gesto *push* reconocido por el sensor *Leap Motion*. Desciende de la estructura base *DataLeapGesture*.
DataLeapScreenTapGesture	Estructura de datos utilizada para representar un gesto *screen tap* reconocido por el sensor *Leap Motion*. Desciende de la estructura base *DataLeapGesture*.

DataLeapSwipeGesture	Estructura de datos utilizada para representar un gesto *swipe* reconocido por el sensor *Leap Motion*. Desciende de la estructura base *DataLeapGesture*.
DataLeapSymbolGesture	Estructura de datos utilizada para representar un gesto *symbol* reconocido por el sensor *Leap Motion*. Desciende de la estructura base *DataLeapGesture*.
DataLeapTurnGesture	Estructura de datos utilizada para representar un gesto *turn* reconocido por el sensor *Leap Motion*. Desciende de la estructura base *DataLeapGesture*.
DataLeapZoomGesture	Estructura de datos utilizada para representar un gesto *zoom* reconocido por el sensor *Leap Motion*. Desciende de la estructura base *DataLeapGesture*.

Tabla 9: Especificación del tipo de datos de *DataLeapGesture*

DataKinectGesture	
Nombre	**Descripción**
DataKinectClickGesture	Estructura de datos utilizada para representar un gesto *click* reconocido por el sensor *Microsoft Kinect* (de usuario). Desciende de la estructura base *DataKinectGesture*.
DataKinectCrossedHandsGesture	Estructura de datos utilizada para representar un gesto *crossed hands* reconocido por el sensor *Microsoft Kinect* (de usuario). Desciende de la estructura base *DataKinectGesture*.
DataKinectPsiGesture	Estructura de datos utilizada para representar un gesto *psi pose* reconocido por el sensor *Microsoft Kinect* (de usuario). Desciende de la estructura base *DataKinectGesture*.
DataKinectRaiseHandGesture	Estructura de datos utilizada para representar un gesto *raise hands* reconocido por el sensor *Microsoft Kinect* (de usuario). Desciende de la estructura base *DataKinectGesture*.
DataKinectWaveGesture	Estructura de datos utilizada para representar un gesto *wave hand* reconocido por el sensor *Microsoft Kinect* (de usuario). Desciende de la estructura base *DataKinectGesture*.

Tabla 10: Especificación del tipo de datos de *DataKinectGesture*

DataTouchKinectGesture	
Nombre	**Descripción**
DataTouchDoubleTapGesture	Estructura de datos utilizada para representar un gesto *double tap* reconocido por el sensor *Microsoft Kinect* (de interacción multitáctil). Desciende de la estructura base *DataTouchKinectGesture*.
DataTouchDragGesture	Estructura de datos utilizada para representar un gesto *drag* reconocido por el sensor *Microsoft Kinect* (de interacción multitáctil). Desciende de la estructura base *DataTouchKinectGesture*.
DataTouchLongTapGesture	Estructura de datos utilizada para representar un gesto *long tap* reconocido por el sensor *Microsoft Kinect* (de interacción multitáctil). Desciende de la estructura base *DataTouchKinectGesture*.
DataTouchTapGesture	Estructura de datos utilizada para representar un gesto *tap* reconocido por el sensor *Microsoft Kinect* (de interacción multitáctil). Desciende de la estructura base *DataTouchKinectGesture*.
DataTouchCircleGesture	Estructura de datos utilizada para representar un gesto *circle*

	reconocido por el sensor *Microsoft Kinect* (de interacción multitáctil). Desciende de la estructura base *DataTouchKinectGesture*.
DataTouchRotateGesture	Estructura de datos utilizada para representar un gesto *rotate* reconocido por el sensor *Microsoft Kinect* (de interacción multitáctil). Desciende de la estructura base *DataTouchKinectGesture*.
DataTouchLetterGesture	Estructura de datos utilizada para representar un gesto *letter* reconocido por el sensor *Microsoft Kinect* (de interacción multitáctil). Desciende de la estructura base *DataTouchKinectGesture*.

Tabla 11: Especificación del tipo de datos de *DataTouchKinectGesture*

La jerarquía de estructuras de datos detallada en esta sección es utilizada para incluir diferentes tipo de datos datos en los mensajes *AFP* intercambiados según corresponda en cada caso, por lo que cada una de ellas forma parte de la definición y estructura del protocolo. De todos modos, es preciso tener en cuenta que el manejo de estos diferentes tipos de datos dentro de un mensaje *AFP* es gestionado de forma transparente gracias a la librería *Boost::Serialization* ya detallada en el *Capítulo 2*, ya que permite serializar los datos en un extremo y crear exactamente la misma estructura de datos una vez llegan al extremo opuesto. Esto evita tener que definir y procesar explícitamente de forma manual la estructura de un mensaje *AFP* según el tipo de datos que se esté transportando, simplemente defiendo la estructura de un mensaje genérico de largo variable y delegando a *Boost::Serialization* su serialización para el envío por la red. Esto evita también la necesidad natural de incluir campos específicos para el manejo del largo del mensaje como parte del encabezado de cada mensaje en caso de tener que parsear manualmente los mensajes una vez llegan a destino. Finalmente, cabe mencionar que si bien existe una gran cantidad de estructuras de datos que permiten representar mensajes específicos del protocolo *AFP*, por simplicidad de los diagramas presentados, se incluyen solo las más representativas en cada caso.

5.6. Archivos de configuración del sistema

Como se mencionó en secciones anteriores, existen dos archivos de configuración que permiten definir los diferentes parámetros del sistema de forma que se comporte correctamente. Uno de estos archivos se corresponde con la configuración de la capa *Data Access* (un archivo por cada instancia que se esté ejecutando), mientras que el otro se corresponde con la configuración de la capa *Core* del sistema. Los archivos de configuración tienen formato *XML*, por lo que se debe conocer la jerarquía adecuada de etiquetas a modo de configurar los diferentes parámetros. Si bien la estructura completa de los archivos de configuración se incluye en el *Anexo M*, en las subsecciones siguientes se presentan a grandes rasgos los parámetros configurables más relevantes junto con las etiquetas *XML* en las que se deben configurar.

5.6.1. Data Access

El archivo de configuración de la capa de acceso a datos está organizado básicamente en cuatro grandes grupos de parámetros:

1. Configuración de *logging*: permite definir todo lo referente al mecanismo de *logging* de la capa, incluyendo el nombre y la ubicación del fichero de *log*, pudiendo especificar que se *loguee* absolutamente todo, que sólo se logueen algunos mensajes específicos, que sólo se logueen errores o que no se loguee nada en absoluto. Cabe mencionar que las configuraciones de *logging* de cualquiera de las capas del sistema siguen las pautas definidas por el módulo de *logging* de la librería *openFrameworks* ya detallada en el *Capítulo 2*. La jerarquía de etiquetas específicas utilizadas para este grupo de parámetros son las siguientes:

```
<logging writeToFile="1"> <!-- 0-OF_LOG_VERBOSE 1-OF_LOG_NOTICE      2-OF_LOG_WARNING >
                               3-OF_LOG_ERROR   4-OF_LOG_FATAL_ERROR 5-OF_LOG_SILENT -->
    <logLevel>3</logLevel>
    <logPath>../../../log</logPath>
    <logFile>animusDataAccess.log</logFile>
</logging>
```

2. Configuración de acceso al *Core*: permite especificar las configuraciones de red para establecer una comunicación correcta entre la capa de acceso a datos y la capa *Core*. En particular, se define la dirección *IP* del nodo que aloja la capa *Core*, así como también los puertos en los que se escucha la petición de registro de nuevos sensores y los mensajes de datos recabados por los sensores configurados. La jerarquía de etiquetas específicas utilizadas para este grupo de parámetros son las siguientes:

```
<coreAccess>
    <ip>127.0.0.1</ip>
    <deviceRegistrationPort>1111</deviceRegistrationPort>
    <dataAnalysisPort>2222</dataAnalysisPort>
</coreAccess>
```

3. Configuración de sensores: permite especificar el tipo y las características de los sensores a utilizar, incluyendo su posición con respecto al sistema de coordenadas global definido (ya detallado en la *Sección 4.2*), si se encuentra activo o no, si debe enviar los datos crudos completos al nodo servidor, etc. Estas configuraciones son definidas mediante la etiqueta <*devices*>, dentro de la cual se puede configurar un sensor *Leap Motion* utilizando la subetiqueta <*leap*> o un sensor *Microsoft Kinect* utilizando la subetiqueta <*kinect*>:

```
<devices>
    <leaps>
        <leap>
            <id>0</id>
            <description>Leap 1 para usuario</description>
            <position x="-510.0" y="320.0" z="1060.0"/>
            <active>0</active>
            <angleX>0.5</angleX>
            <angleY>0.0</angleY>
            <angleZ>0.0</angleZ>
            <sendframe>0</sendframe>
            <recognizer>
```

```
<kinects>
    <kinect>
        <position x="0.0" y="-1400.0" z="-200.0"/>
        <active>1</active>
        <angleX>0.35</angleX>
        <angleY>0.0</angleY>
        <angleZ>0.0</angleZ>
        <sendframe>0</sendframe>
        <description>Kinect 1 para usuario</description>
        <id>1</id>
        <instanceType>0</instanceType>
        <recognizer>
```

Además para cada uno de los sensores es posible definir un reconocedor de gestos no nativos mediante la etiqueta <*recognizers*>, según lo detallado en la *Sección 5.4.1.2*, especificando uno o más *pipelines* a utilizar mediante la etiqueta <*pipelines*>. Para cada *pipeline* a incluir se debe especificar su nombre, descripción, qué algoritmo de reconocimiento se utilizará, y en caso de haberlo, un eventual extractor de características mediante la etiqueta <*extractor*>. Se define aquí también el mapeo entre un posible gesto reconocido por el *pipeline* y el identificador del gesto para el sistema, de forma que se puedan correlacionar adecuadamente. Si bien existen más etiquetas específicas del mecanismo de reconocimiento de gestos no nativos, escapan al alcance de este capítulo; por más detalles referirse al *Anexo M*. La jerarquía de etiquetas específicas utilizadas para definir un reconocedor son las siguientes:

```
<recognizer>
    <id>0</id>
    <name>LeapGestureRecognizer</name>
    <description>Reconocedor para gestos generados con Leap</description>
    <pipelines>
        <pipeline>
            <name>LeapPipeline2D1F</name>
            <description>Pipeline que reconoce gestos estaticos (letras, simbolos, etc) en dos dimensiones</description>"">
            <active>0</active>
            <dimension>2</dimension>
            <trainingFile>LeapPipeline2D1F.txt</trainingFile>
            <algorithm>0</algorithm> <!-- 0-DTW, 1-MinDist, 2-ANBC, 3-KNN, 4-SVM -->
            <nullRejection enabled="1">
                <coeff>1</coeff>
            </nullRejection>
            <ZNormalization enabled="1"/>
            <trimTrainingData enabled="1">
                <param1>0.1</param1>
                <param2>50</param2>
            </trimTrainingData>
            <offsetTimeseriesUsingFirstSample enabled="1"/>
            <tests enabled="2">
                <mode>0</mode> <!-- 0-MANUAL_MODE, 1-CROSS_VALIDATION, 2-PIPELINE_TEST -->
                <partition>20</partition>
            </tests>
            <extractor>
                <name>OneFingerPositionOnlyExtractor</name>
                <description>Extractor que obtiene unicamente la posicion de un unico dedo extendido</description>
                <usePredictionModel enabled="0"/>
                <dimension>2</dimension>
                <fingers>
                    <finger>
                        <!-- No type means ANY finger -->
                        <position>1</position>
                        <direction>0</direction>
                        <velocity>0</velocity>
                    </finger>
                </fingers>
                <hands>
                    <position>0</position>
                    <direction>0</direction>
                    <velocity>0</velocity>
                    <normal>0</normal>
                    <radius>0</radius>
                </hands>
            </extractor>
            <gestureMapping>
                <classLabels>1</classLabels>
                <gestureCodes>42</gestureCodes> <!-- 42 - ANIMUS_PINCH -->
            </gestureMapping>
        </pipeline>
    </pipelines>
</recognizer>
```

4. Otros parámetros: permiten especificar al sistema que muestre información adicional específica para el sensor *Microsoft Kinect*. En particular, es posible indicar que se visualice el mapa de profundidad o las imágenes *RGB* obtenidas, mapas con los interacciones táctiles detectadas, etc.

```
<showmaps>
    <color>0</color>
    <depth>1</depth>
    <background>0</background>
    <contournous>1</contournous>
    <contournousHands>1</contournousHands>
    <touch>1</touch>
    <touchHands>1</touchHands>
    <foreground>0</foreground>
</showmaps>
```

5.6.2. Core/Application

Al igual que para la capa de acceso a datos, el archivo de configuración de la capa *Core* también se estructura básicamente en cuatro secciones:

1. Configuraciones de *logging*: al igual que para la capa de acceso a datos, permite definir todo lo referente al mecanismo de *logging* de la capa, incluyendo el nombre y la ubicación del fichero de *log*, pudiendo especificar que se *loguee* absolutamente todo, que sólo se *logueen* algunos mensajes específicos, que sólo se *logueen* errores o que no se *logueé* nada en absoluto. Cabe mencionar que las configuraciones de *logging* de cualquiera de las capas del sistema siguen las pautas definidas por el módulo de *logging* de la librería *openFrameworks* ya detallada en el *Capítulo 2*.

```
<logging writeToFile="1"> <!-- 0-OF_LOG_VERBOSE 1-OF_LOG_NOTICE      2-OF_LOG_WARNING >
                               3-OF_LOG_ERROR   4-OF_LOG_FATAL_ERROR 5-OF_LOG_SILENT -->
    <logLevel>3</logLevel>
    <logPath>../../../log</logPath>
    <logFile>animusDataAccess.log</logFile>
</logging>
```

2. Configuración de acceso a *Data Access*: permite especificar las configuraciones de red para establecer una comunicación correcta entre la capa *Core* y la capa de acceso a datos. En particular, se definen los puertos en los que se escuchará la petición de registro de nuevos sensores y los mensajes de datos recabados por los sensores configurados. Los puertos deben ser los mismos que los configurados en la capa *Data Access* para que la comunicación se pueda llevar a cabo.

```
<coreAccess>
  <registerdeviceport>1111</registerdeviceport>
  <analizedataport>2222</analizedataport>
</coreAccess>
```

3. Configuración de sensores: permite especificar cuántos y qué tipo de sensores se van a registrar en el sistema, así como también configurar los parámetros que definen las diferentes transformaciones que posibilitan la conversión al sistema de coordenadas de proyección y al sistema de coordenadas de escena para cada uno de ellos. Estas configuraciones son definidas dentro de la etiqueta <*devices*>, utilizando las subetiquetas <*sceneMapping*> y <*screenMapping*> según el mapeo a definir. Además, según el tipo de sensor que se esté configurando se utilizan las subetiquetas <*leap*>, <*singlekinect*> y <touchkinect>.

```
<devices>
    <quantity>2</quantity>
    <sceneMapping>
        <leap>
            <active>1</active>
            <deviceMinRange x="-100" y="10" z="-100"/>
            <deviceMaxRange x="100" y="200" z="100"/>
            <applicationMinRange x="-500" y="-500" z="-500"/>
            <applicationMaxRange x="500" y="500" z="500"/>
        </leap>
        <singlekinect>
            <active>1</active>
            <deviceMinRange x="-500" y="-300" z="1400"/>
            <deviceMaxRange x="500" y="300" z="1800"/>
            <applicationMinRange x="-500" y="-500" z="-500"/>
            <applicationMaxRange x="500" y="500" z="500"/>
        </singlekinect>
        <touchkinect>
            <active>0</active>
            <deviceMinRange x="-550" y="-180" z="1100"/>
            <deviceMaxRange x="550" y="480" z="1100"/>
            <applicationMinRange x="-500" y="-500" z="-500"/>
            <applicationMaxRange x="500" y="500" z="500"/>
        </touchkinect>
    </sceneMapping>
    <screenMapping>
        <leap>
            <active>0</active>
            <leftUpperCorner x="0" y="0"/>
            <rightUpperCorner x="0" y="0"/>
            <rightBottomCorner x="0" y="0"/>
            <leftBottomCorner x="0" y="0"/>
        </leap>
        <singlekinect>
            <active>0</active>
            <leftUpperCorner x="0" y="0"/>
            <rightUpperCorner x="0" y="0"/>
            <rightBottomCorner x="0" y="0"/>
            <leftBottomCorner x="0" y="0"/>
        </singlekinect>
        <touchkinect>
            <active>0</active>
            <leftUpperCorner x="-145" y="-445"/>
            <rightUpperCorner x="0" y="0"/>
            <rightBottomCorner x="0" y="0"/>
            <leftBottomCorner x="0" y="0"/>
        </touchkinect>
    </screenMapping>
</devices>
```

4. Otros parámetros: permiten especificar los intervalos de tiempo de espera para recibir notificaciones sobre información del sistema; específicamente información sobre usuarios y sensores.

```
<animusInformation>
  <userInformationInterval>30</userInformationInterval>
  <deviceInformationInterval>30</deviceInformationInterval>
</animusInformation>
```

5.7. Resumen del capítulo

En este capítulo se presentó la arquitectura que da soporte a la solución propuesta, describiendo cómo están diseñados los diferentes componentes lógicos del sistema basados fuertemente en un modelo de capas, así como también la forma en que éstos son dispuestos en nodos físicos y la comunicación existente entre ellos. Se describen también las especificaciones *software* y *hardware* de los nodos utilizados para dar soporte a la solución, a modo de brindar una pauta sobre los requerimientos recomendados para el correcto funcionamiento del sistema. Por otra parte, se detallan los módulos más importantes que componen cada una de las capas del sistema, y para cada uno de ellos se presentan las clases más importantes y su rol dentro del módulo. Se incluyen pseudocódigos que permiten visualizar de mejor forma la lógica más relevante dentro de cada módulo, facilitando así su comprensión. Finalmente, se presenta el protocolo de comunicación utilizado por el *framework* para intercambiar mensajes a través de la red entre las capas de procesamiento (*Core*) y acceso a datos (*Data Access*), así como también una descripción a grandes rasgos de los principales parámetros configurables del sistema.

Capítulo 6: Aplicación

En este capítulo se detalla la aplicación propuesta como prototipo de uso del *framework* generado, denominada *Solar System Application.* Se describe desde la perspectiva del usuario las diferentes funcionalidades provistas por la aplicación así como también, las distintas formas de interacción que brinda a los usuarios. En el *Anexo N*, se detallan los aspectos de implementación para esta aplicación particular.

6.1. Descripción general

Solar System Application es una aplicación desarrollada sobre el *framework* generado como parte del presente proyecto, por lo que la interacción está fuertemente basada en los diferentes servicios que brinda (ya presentados con más detalle en el *Capítulo 5*). La aplicación pretende recrear el sistema solar, permitiendo consultar y visualizar diversa información de los elementos que lo componen. Esto es realizado mediante un subconjunto de los gestos disponibles como parte del *framework* construido, incluyendo tanto gestos multitáctiles como espaciales. Según el gesto realizado el usuario podrá comandar al sistema ya sea para modificar la forma en que se muestran los elementos, seleccionar elementos u obtener información de ellos.

En la *Sección 6.3* de este mismo capítulo se detallan los gestos particulares utilizados como parte de la aplicación, junto a las acciones específicas que cada uno de ellos permite realizar. Adicionalmente, en el vídeo de desarrollo *"Interacción mediante gestos espaciales" [v.2]* se muestran los gestos del tipo espaciales en acción, mientras que en el vídeo de desarrollo *"Interacción mediante gestos multitáctiles" [v.3]* se muestran los gestos multitáctiles en acción. De todas formas, a grandes rasgos la aplicación permite interactuar con la escena de forma global, pudiendo modificar la forma en que se visualiza la escena principal que muestra todos los planetas del sistema solar girando en torno al Sol, así como también, interactuar con cada planeta particular mediante una selección previa. En lo que respecta a la interacción global, la aplicación permite mover el plano completo de la escena, pudiendo desplazarse, alejarse o acercarse, y habilitar o deshabilitar el movimiento natural de los planetas en conjunto. Por otro lado, en cuanto a la interacción individual con cada elemento, es posible seleccionar un elemento particular (planeta o Sol), visualizar un conjunto de imágenes asociadas al elemento seleccionado junto a la información que cada una de estas imágenes representa, rotar un elemento previamente seleccionado y aumentar o reducir su tamaño.

6.2. Aplicación del estado del arte

Teniendo en cuenta los aspectos de interacción presentados en el *Capítulo 3* como parte del estudio del estado del arte para este trabajo, la aplicación desarrollada intenta utilizar técnicas de interacción que no presumen ningún conocimiento previo del usuario ni le atribuyan ninguna otra habilidad interactiva que no sea la que le permite relacionarse con otras personas o el entorno que lo rodea. Para ello, se utiliza un subconjunto de gestos considerados sencillos de entre los proporcionados por el *framework* generado, buscando que los usuario finales comprendan exactamente qué están haciendo en cada momento, y a su vez, que sean conscientes en todo momento del lugar en que se encuentran dentro de las diferentes posibilidades de navegación. Esto es logrado mediante transiciones sencillas establecidas por una máquina de estados específica para cada situación dentro de la escena de forma que la interacción sea lo más intuitiva posible, complementado con una retroalimentación acorde a cada interacción establecida. Esto evita la necesidad de que el usuario deba pensar cómo funciona o cómo encontrar la forma de realizar cierta tarea, pudiendo centrarse en cumplir las tareas requeridas.

La retroalimentación se manifiesta de diferentes formas. Por un lado se utilizan barras de progreso que permiten mostrar el grado de progreso al realizar la selección de un elemento (el elemento se selecciona cuando la barra de progreso de selección está llena), proveyendo retroalimentación constante al usuario, tal como se muestra en la *Figura 88*. A su vez, se escogen tiempos de selección que no frustren al usuario evitando acciones demasiado largas o cansadoras que hagan que la precisión decremente afectando la performance en la interacción general.

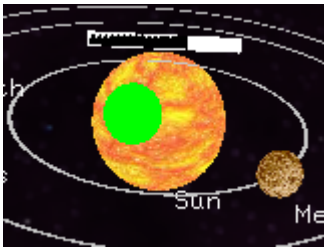


Figura 88: Barra de progreso para selección de elemento

Por otro lado, se utilizan animaciones para reubicar los elementos seleccionados lo más cerca de cada usuario posible, de forma de facilitar la interacción multitáctil sobre cada elemento. La animación brinda un efecto de acercamiento al usuario, lo cual complementado con el hecho de quitar el elemento seleccionado de la vista global, permite que el usuario verdaderamente perciba que tiene el elemento seleccionado a su disposición, tal como se puede observar en la *Figura 89*. A su vez, independientemente de cualquier selección realizada por parte de los usuarios, la vista global del sistema solar se encuentra en continuo movimiento, por lo que es capaz de atraer más la atención en comparación con una escena estática, siendo este movimiento esencial como parte de la interacción.

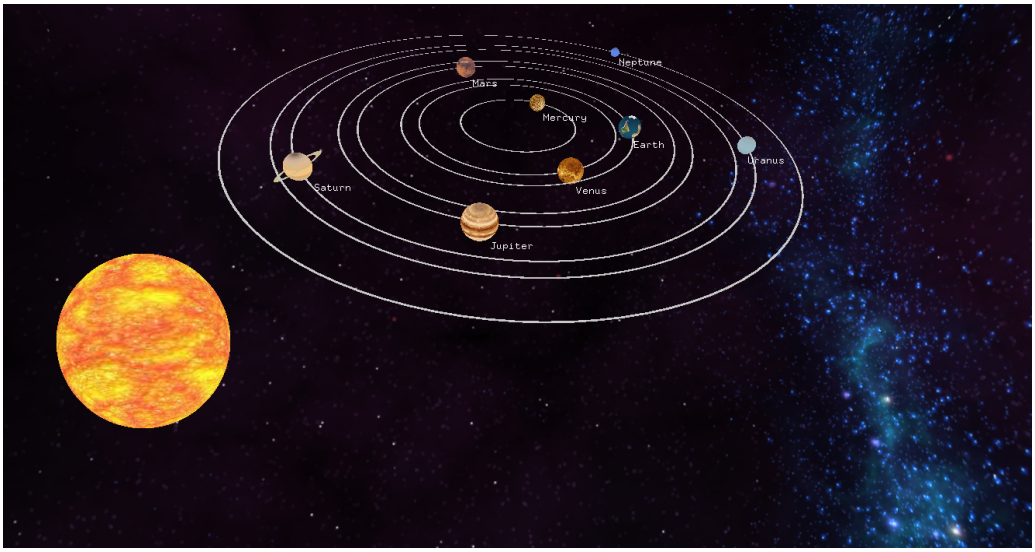


Figura 89: Elemento seleccionado por el usuario izquierdo

Finalmente, otro mecanismo de retroalimentación esencial en este tipo de sistemas refiere a que el usuario sepa con certeza si los diferentes dispositivos y sensores lo están captando o no en cada momento, más específicamente a cuántas personas están captando, y si el sensor está listo para llevar a cabo la interacción. Para ello, la aplicación brinda información de retroalimentación del seguimiento de los usuarios cada cierto tiempo, indicando cuántos sensores están activos y cuántos usuarios están siendo reconocidos en la escena, esto se puede observar en la *Figura 90*.

Figura 90: Retroalimentación cantidad usuarios reconocidos y sensores activos

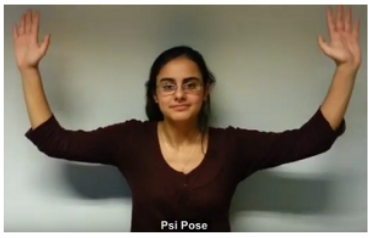
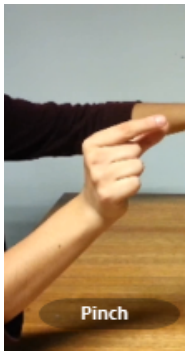
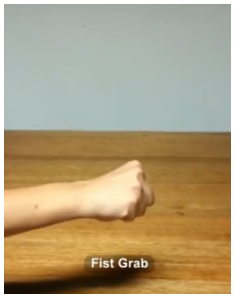
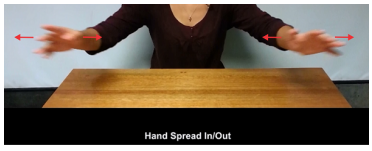
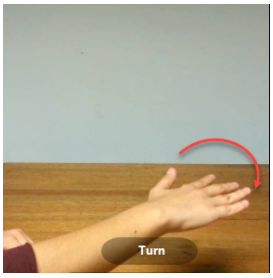
En cuanto al subconjunto de gestos escogidos para ser utilizados en la aplicación, fueron cuidadosamente seleccionados de modo que sean intuitivos, fáciles de aprender y recordar por los usuarios, y lo suficientemente distintos entre sí como para evitar colisiones entre ellos. A su vez, se intenta mantener coherencia con otras aplicaciones en el sentido de que ciertos gestos frecuentemente utilizados se utilicen exactamente de la misma forma (como por ejemplo, la forma de seleccionar un elemento o hacer *zoom*), ya que esto ayuda a que los usuarios se sientan familiarizados con gestos típicos o universales, reduciendo así la cantidad de gestos nuevos que se deben aprender y consecuentemente la carga cognitiva necesaria para interactuar con la aplicación. Se evita también la utilización de gestos que deban realizarse únicamente con una mano particular, dado que poseen una pobre accesibilidad y son más difíciles de detectar. Así, los gestos pueden ser realizados con cualquiera de las dos manos de forma indistinta, por lo que el usuario puede optar por intercambiar de mano sin inconvenientes, disminuyendo la fatiga y evitando el esfuerzo adicional inherente en memorizar con qué mano debe realizar determinado gesto. Además, esto resulta en una interacción apropiada tanto para usuarios diestros como zurdos, haciendo que la aplicación tenga mayor accesibilidad. A su vez, teniendo en cuenta que los gestos realizados con una sola mano son altamente preferibles por el simple hecho de mejorar la eficiencia de la realización/detección, se incluye únicamente una minoría de gestos que deben ser realizados con ambas manos. Estos gestos son simétricos en el sentido de que ambas manos realizan el mismo movimiento, facilitando su realización y la capacidad de recordarlos. Si bien en esta sección se describen las pautas generales del diseño de interacción al momento de definir los gestos a utilizar, en las siguientes secciones se presentará cada uno de los gestos particulares y la acción que disparan en la aplicación.

Por otro lado, la aplicación construida permite la competencia y colaboración entre usuarios; en particular para la aplicación desarrollada, varios usuarios podrían interactuar colaborativamente con el sistema solar en la vista global, e individualmente mediante la selección de un elemento. Otra ventaja de la aplicación es que permite a los usuario sentirse libres pudiendo moverse sin restricciones dentro o alrededor del espacio interactivo de la superficie, brindando la libertad suficiente como para asegurar que el usuario pueda decidir si interactuar o no.

Por último, como se mencionó anteriormente, *Solar System Application* hace uso del *framework* desarrollado, el cual requiere de la utilización de un conjunto de servidores y sensores conectados entre sí. Estos dispositivos controlan de forma simultánea el entorno donde se encuentra inmerso el usuario y son completamente transparentes dado que el usuario no es consciente de su existencia para llevar a cabo la interacción, haciendo que el sistema como un todo se considere un sistema ubicuo por naturaleza. A su vez, se utiliza la superficie de una mesa del mundo real para proyectar un mundo virtual con el cual el usuario pueda interactuar mediante diferentes vías de interacción, por lo que el sistema se puede considerar un sistema basado en realidad aumentada, donde los diferentes sensores proveen la información del mundo real que los computadores computan y presentan de forma visual sobre la superficie física de la mesa de interacción, la cual se ve aumentada con elementos virtuales interactivos. Se logra entonces, integrar diferentes componentes como proyectores, cámaras, sensores y computadoras, de forma eficiente en el sentido de que desaparecen como objetos individuales e independientes para el usuario, generando así la ilusión de que se está interactuando con una única entidad.

6.3. Gestos utilizados

Como ya se mencionó, la aplicación utiliza un subconjunto de todos los gestos provistos por el *framework* desarrollado. En la *Tabla 12* se describe cada uno de los gestos involucrados en la interacción de la aplicación, incluyendo una imagen descriptiva del gesto en cuestión y una descripción de cómo realizarlo. En la siguiente sección se detallarán las posibles acciones y el resultado dentro de la escena al realizar cada uno de estos gestos en la aplicación.

Nombre	Gesto	Descripción
Psi Hand		Consiste en mantener ambas manos levantadas por encima de los hombros por un cierto lapso de tiempo.
Pinch		Consiste en juntar los dedos pulgar e índice entre sí (como si se sostuviera algo entre ellos) por un cierto lapso de tiempo.
Grab		Consiste en mantener cerrada una mano por un cierto lapso de tiempo.
Zoom hand		Consiste en mantener ambas manos abiertas mientras se acercan y se alejan entre sí.
Turn hand		Consiste en mantener una mano abierta con la palma hacia abajo y luego girarla hasta que la palma quede hacia arriba.

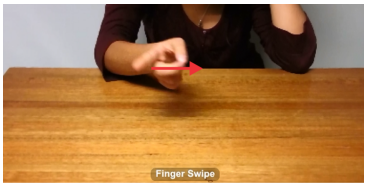
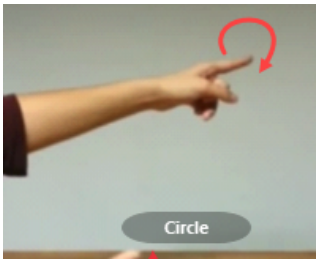
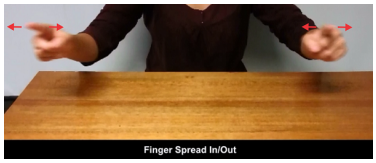
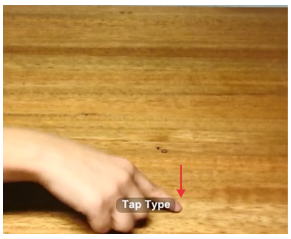
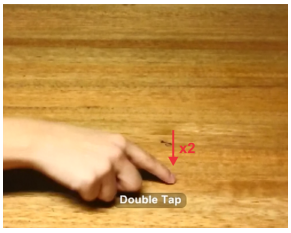
Finger swipe		Consiste en mantener estirado el dedo índice de una mano y luego deslizarlo hacia uno de los lados.
Circle		Consiste en mantener en posición horizontal el dedo índice de una mano mientras se mueve formando un círculo.
Finger Zoom		Consiste en mantener estirado el dedo índice de cada mano mientras se acercan y se alejan entre sí.
Touch tap		Consiste en mover el dedo índice de una mano de arriba hacia abajo hasta establecer contacto con la superficie.
Touch double tap		Consiste en mover el dedo índice de una mano de arriba hacia abajo hasta establecer contacto con la superficie dos veces consecutivas.

Tabla 12: Banco de gestos utilizados en *Solar System Application*

6.4. Interacción

En esta sección se presentan los diferentes tipos de interacción basadas en el banco de gestos seleccionado ya descrito en la sección anterior. Como ya se mencionó, la interacción dentro de la aplicación se puede clasificar básicamente en dos tipos. Por un lado interacción global, mediante la que los usuarios pueden interactuar colaborativamente en la vista global del sistema solar, y por otro lado, interacción individual o específica, mediante la cual cada usuario interactúa individualmente con el elemento que tiene seleccionado en un momento dado. Cada uno de estos tipos de interacción involucra diferentes gestos. En la *Sección 6.4.1* y *6.4.2* se detallan dichos gestos junto con las acciones asociadas para la aplicación particular y el resultado que se obtiene al realizarlos.

6.4.1. Interacción global

Mediante este tipo de interacción es posible modificar la forma en que se visualiza la vista global, pudiendo habilitar o deshabilitar el modo estático, moverse a través de la escena, y aumentar o reducir el tamaño de los elementos en conjunto. Más específicamente, al realizar un gesto de *Psi Hand*, es posible activar el modo estático, haciendo que los elementos dejen de moverse hasta que se vuelva a realizar el

mismo gesto. Esto es conveniente dado que algunos elementos se mueven rápidamente y su selección puede ser dificultosa mientras están en movimiento. Por otra parte, mediante un gesto de *Grab* es posible moverse a través de la escena. Al realizar un gesto de *Grab* se muestra un indicador coloreado según el usuario al que corresponda (verde o rojo) que permite visualizar el centro de desplazamiento, es decir, el punto a partir del cual se está moviendo la escena. En la *Figura 91* se puede visualizar el resultado de realizar este gesto sobre la vista global de la aplicación.

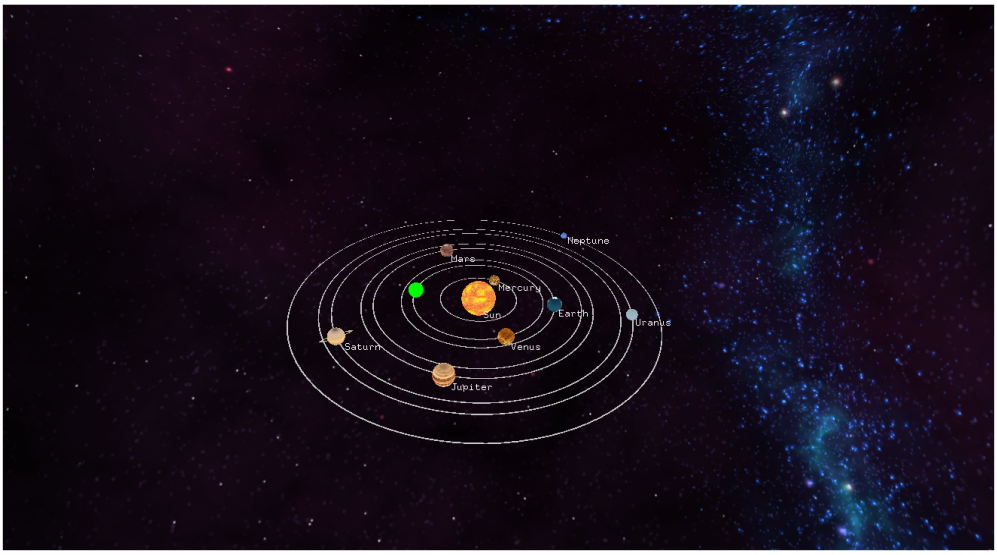


Figura 91: Resultado de realizar un gesto de *Grab* en la vista global

Finalmente, un gesto de *Zoom Hand* permite aumentar o reducir el tamaño de todos los elementos de la escena, o visto de otro modo, acercarse (*zoom in*) o alejarse (*zoom out*) de ellos respectivamente, dependiendo de la dirección en que se muevan las manos. La *Figura 92* permite ilustrar el resultado de realizar un acercamiento y un alejamiento sobre la vista global de la aplicación.

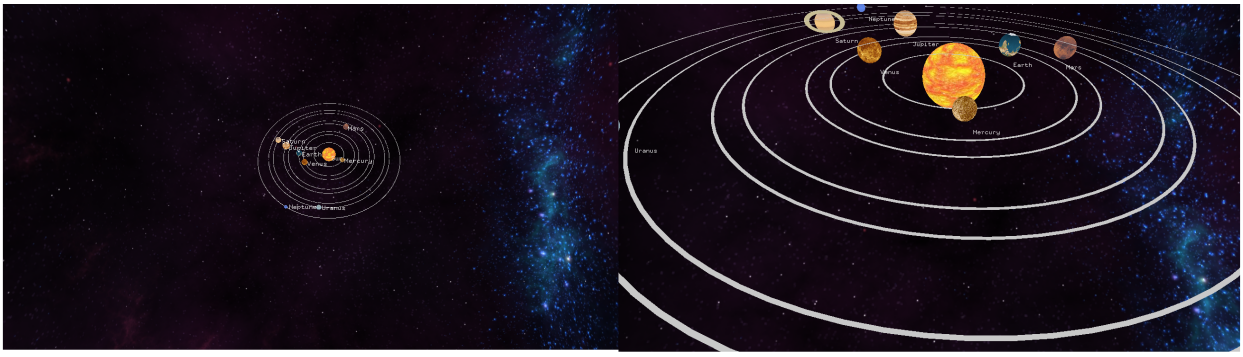


Figura 92: Resultado de realizar un gesto de *Zoom Hand* (*zoom out*) en la vista global (izq.). Resultado de realizar un gesto de *Zoom Hand* (*zoom in*) en la vista global (der.)

6.4.2. Interacción individual

Este tipo de interacción refiere a la interacción llevada a cabo sobre un elemento previamente seleccionado por uno de los usuarios, por lo que para llevar a cabo todas las acciones consecuentes se requiere necesariamente haber realizado una selección. Para seleccionar un elemento se debe realizar un gesto de *Pinch*, visualizando un indicador coloreado según el usuario al que corresponda (verde o rojo) que permite conocer la posición en la escena sobre la que se está interactuando, tal como se muestra en la *Figura 93*. Haciendo que dicho indicador coincida con uno de los elementos presentes, se muestra una barra de progreso que se irá llenando en la medida que se mantenga la selección sobre el elemento. Este comportamiento se puede visualizar en la *Figura 94*. Una vez se completa la barra de progreso, el elemento

en cuestión es seleccionado y desplazado a la zona de interacción particular para el usuario que lo seleccionó (ya sea en la esquina inferior izquierda o derecha), tal como se puede apreciar en la *Figura 95*.

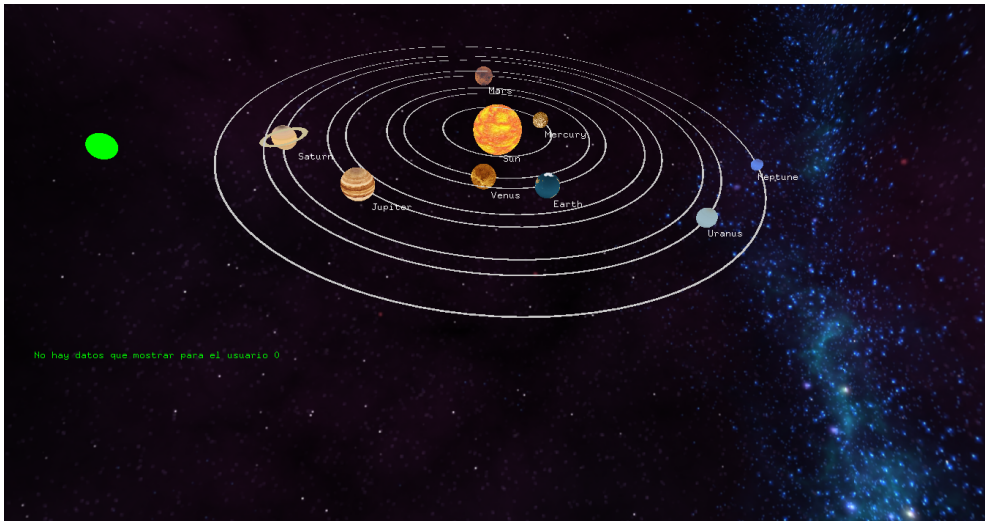


Figura 93: Resultado de realizar un gesto de *Pinch* para seleccionar un elemento

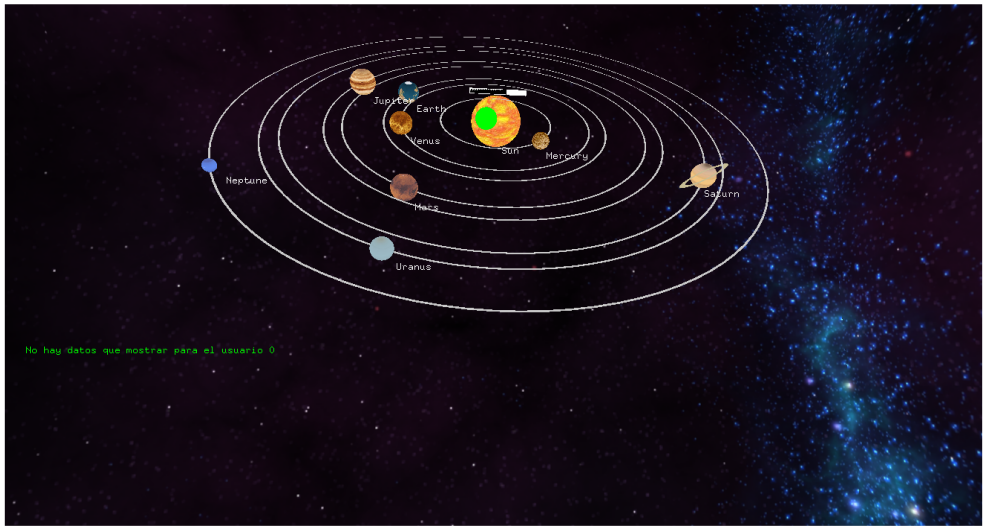


Figura 94: Resultado de realizar un gesto de *Pinch* sobre un elemento

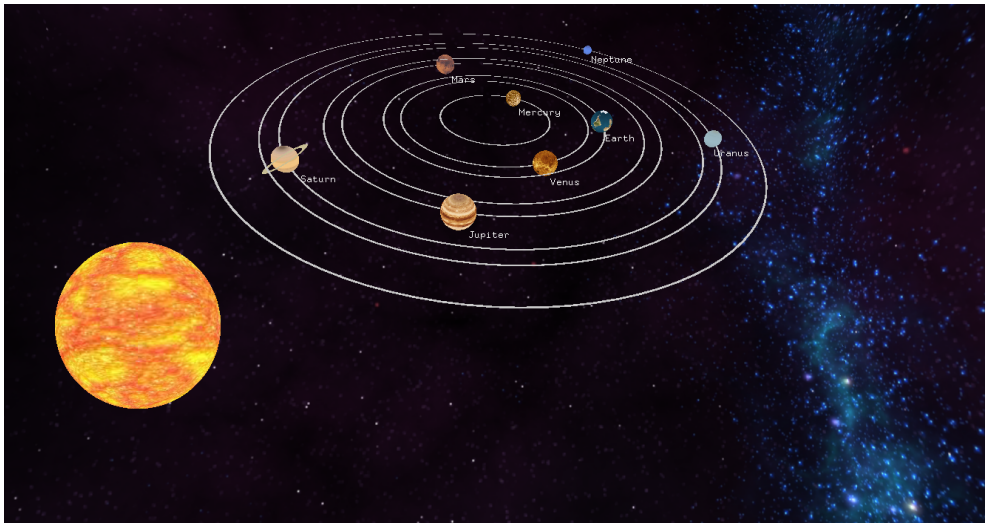


Figura 95: Selección de un elemento

Una vez seleccionado un elemento, se abre un abanico de posibilidades de interacción para ese elemento particular seleccionado. Así, por ejemplo, es posible rotar el elemento seleccionado mediante un gesto de *Circle*, modificar su tamaño realizando un gesto de *Finger Zoom* como se puede apreciar en la *Figura 96*, o realizar un *Touch Double Tap* sobre el elemento con el fin de visualizar un texto descriptivo como se puede apreciar en la *Figura 97.* A su vez, mediante un gesto de *Turn Hand* es posible visualizar las imágenes asociadas al elemento seleccionado, tal como muestra la *Figura 98.* En este estado, también es posible realizar un *Touch Tap* sobre alguna de las imágenes con el fin de visualizar un texto descriptivo como se puede apreciar en la *Figura 99.* Por último, en caso de querer retornar el elemento a la vista global se debe realizar un gesto de *Finger Swipe* hacia alguno de los lados.

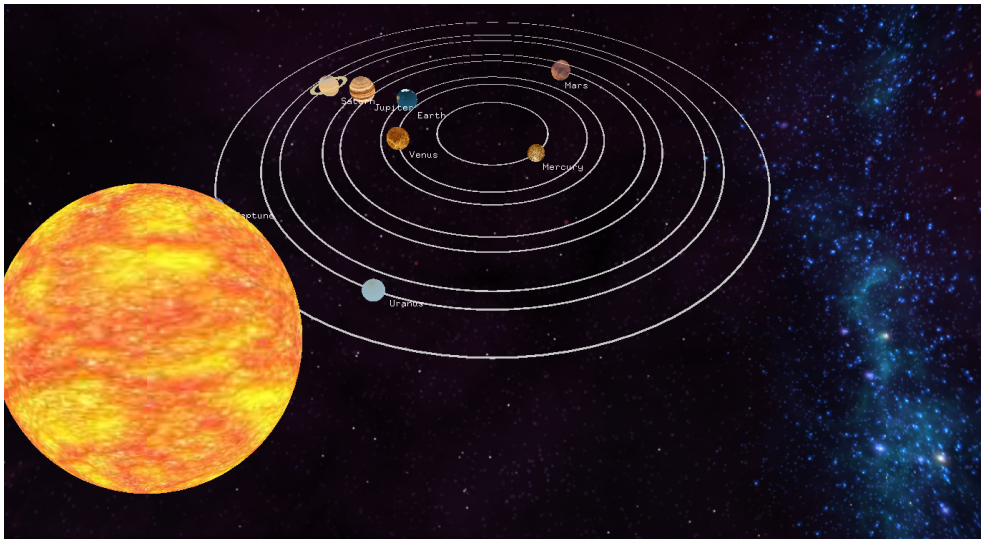


Figura 96: Resultado de realizar un gesto de Finger Zoom

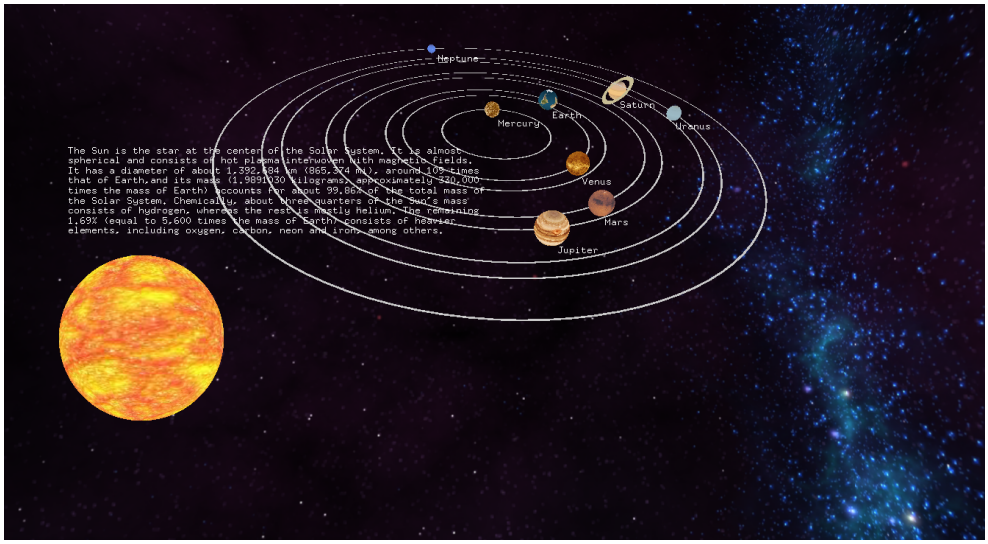


Figura 97: Resultado de realizar un gesto de *Touch Double Tap*

Figura 98: Resultado de realizar un gesto de *Turn Hand*

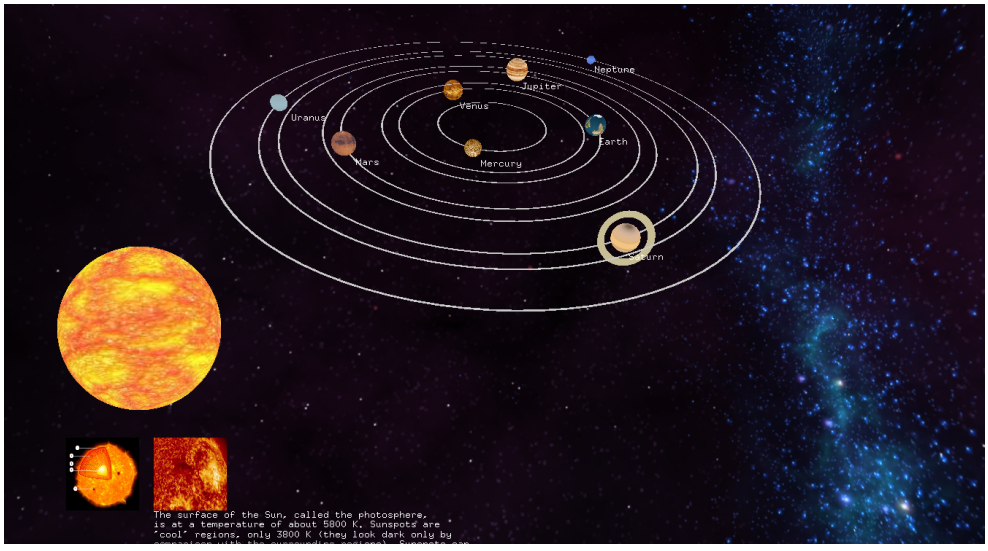


Figura 99: Resultado de realizar un gesto de *Touch Tap* sobre una de las imágenes

6.5. Resumen del capítulo

En este capítulo se describió, desde el punto de vista del usuario final, la aplicación prototipo desarrollada sobre el *framework* generado. Se presentó una descripción general de las funcionalidades de la aplicación así como también, los diferentes aspectos que se tuvieron en cuenta para el diseño de la interacción. La aplicación presenta una versión interactiva del sistema solar, pudiendo interactuar con cada uno de sus elementos mediante un conjunto preestablecido de gestos multitáctiles y espaciales. Se describió cada uno de los posibles gestos a realizar y el resultado que cada uno de ellos genera dentro de la aplicación.

Capítulo 7: Conclusiones

En este capítulo se detallan los resultados obtenidos luego de culminado el presente trabajo, como así también, las posibles mejoras y trabajos futuros que se podrían agregar a la solución propuesta. Adicionalmente, se describe los usos que se le podría dar, es decir, dónde y cómo aplicar los resultados obtenidos del presente trabajo.

7.1. Resultados obtenidos

Como resultado de este trabajo se logró construir un *framework* de interacción natural que brinda los servicios necesarios para poder desarrollar aplicaciones basadas en dos tipos de interacción. Por un lado, interacción tridimensional, es decir, basada en gestos realizados en el aire, y por otro lado, interacción bidimensional o multitáctil, esto es, realizada directamente sobre una superficie física aprovechando así las ventajas de cada vía de interacción. Dicho *framework* permite hacer una correspondencia de los diferentes gestos realizados por los usuarios reconocidos mediante sensores *Microsoft Kinect* y *Leap Motion*, a acciones sobre los objetos virtuales con los que se está interactuando. La solución fue desarrollada enteramente en el lenguaje *C++*, utilizando una combinación de bibliotecas externas para diferentes tareas como son *OpenNI*, *NiTE*, *Leap SDK*, *openFrameworks*, *OpenCV*, *GRT*, *Boost* y *Eigen* (referirse al *Capítulo 2* por un detalle completo de cada un de ellas). Adicionalmente, se puso especial hincapié en lograr una solución con una arquitectura extensible y escalable, que facilite el agregado de nuevos sensores de interacción a la escena. Se brindan también, mecanismos para poder extender los gestos soportados nativamente por el *framework*, y un mecanismo de suscripción a gestos flexible que permite a las aplicaciones suscribirse únicamente a aquellos gestos que se desee.

Adicionalmente, a modo de marco teórico, se realizó un estudio sobre diversas disciplinas relacionadas a este trabajo, como son la Interacción Persona-Computadora, la Interacción Natural y la Interacción Multitáctil, todas ellas áreas claves que estudian las mejores formas de obtener una interacción lo más sencilla posible, y una retroalimentación acorde a cada tipo de interacción. A su vez, se realizó un estudio en profundidad sobre los diferentes dispositivos y sensores a utilizar para construir la solución, y la mejor forma para que éstos puedan trabajar de forma combinada con un objetivo común.

Finalmente, se encontraron soluciones a diversos problemas que indirectamente se debieron resolver para cumplir los objetivos del trabajo. Este es el caso del desarrollo de un módulo propio de interacción multitáctil basada en sensores de profundidad, implementación propia de la calibración proyector sensor *Microsoft Kinect*, implementación de varios sistemas de coordenadas, resolución de la interferencia provocado por el uso de múltiples sensores en simultáneo, el desarrollo de una aplicación que hacen uso del *framework* generado con el fin de demostrar su viabilidad así como el beneficio de la combinación de ambas técnicas de interacción, entre otros, muchos de los cuales ya fueron detallados en el *Capítulo 4*.

A modo de resumen, el presente proyecto de grado produjo los siguientes resultados:

1. Un *framework* de interacción natural desarrollado con *software* libre y multiplataforma, extendiendo el uso de los sensores *Microsoft Kinect* y *Leap Motion*, sobre el cual se pueden desarrollar aplicaciones basadas en interacción natural mediante gestos espaciales y multitáctiles.
2. Una aplicación construida a modo de prototipo, que permite corroborar el correcto funcionamiento del *framework* generado.
3. El presente documento de proyecto de grado, que incluye una investigación exhaustiva del estado del arte al momento de la realización del trabajo, una investigación de los diferentes

dispositivos y sensores utilizados, y los diversos problemas que se debieron resolver para cumplir con los objetivos planteados.

7.2. Posibles mejoras y trabajo futuro

Si bien se logró cumplir con los objetivos planteados para el proyecto, se pueden distinguir algunas posibles mejoras que serían de gran interés para complementar o extender lo ya desarrollado. Se listan a continuación los diferentes puntos identificados como trabajos futuros a realizar sobre la solución propuesta.

1. Ubicación de sensores: se puede lograr una solución más cómoda para el usuario a nivel estructural posicionando los sensores *Leap Motion*, no directamente sobre la superficie, sino directamente embebidos en ella. Esto se podría llevar a cabo colocando dichos sensores en pequeños huecos cubiertos por vidrio u otro material que permita el pasaje de los rayos *IR* sin interferencias, evitando así, que se deban evadir los sensores para interaccionar con la superficie, mejorando entonces, la experiencia de usuario. Esta oportunidad de mejora está inspirada en un ejemplo ya existente y es el caso de las computadoras portátiles que incluyen el sensor *Leap Motion* embebido como parte de su estructura tal como en la *Figura 100* permitiendo al usuario complementar las formas usuales de interacción directa (teclado, mouse, *touch pad*) con nuevos tipos de interacción más innovadores como lo es la interacción con gestos espaciales.

Figura 100: Sensor *Leap Motion* embebido [42]

Otra forma de disponer los sensores *Leap Motion* de forma de enriquecer la experiencia del usuario al interactuar con el sistema es colocarlos, ya no sobre sobre la superficie o embebidos en ella, sino directamente sobre el usuario. Así, cada sensor *Leap Motion* podría colocarse unido de cierta forma a la cabeza del usuario (mediante un sistema de montura para cabeza apropiado), permitiendo una interacción más intuitiva, ya que en este caso el usuario podría realizar los gestos directamente en el lugar donde está posicionado sin la necesidad de colocar sus manos sobre la superficie en la que se encuentra el sensor. Esto elimina también la restricción del rango de visión fijo del sensor al ser colocado sobre la superficie, ya que el usuario tenderá a realizar los gestos en dirección hacia donde está mirando, y al llevar el sensor colocado sobre su cabeza el rango de visión varía dinámicamente con su movimiento, coincidiendo generalmente con la dirección a la que el usuario mira. Esto hace que el uso del sensor sea todavía más invisible, logrando un sistema todavía más ubicuo que plantea una forma de interacción aún menos intrusiva para el usuario. Cabe mencionar que para lograr esto de forma satisfactoria se debería realizar un seguimiento constante de la posición del sensor, de forma que se pueda realizar correctamente la transformación de coordenadas de sensor a coordenadas de sistema independientemente de su posición. Un ejemplo de uso de este tipo de montura para el sensor *Leap Motion* que viene en creciente aumento surge gracias al advenimiento de los cascos de realidad virtual, como el *Oculus Rift* [93, 69] o el *HTC Vive* [68,

58]. En este caso, el sensor es colocado directamente sobre el casco de realidad virtual mediante una montura o adaptador especialmente diseñado, por lo que el usuario puede interactuar con los diferentes objetos que se le presentan en el mundo virtual de forma completamente natural e intuitiva utilizando solo sus manos. Además, gracias a esto, a diferencia de años anteriores hoy en día el sensor *Leap Motion* está optimizado para su uso *top-down* sobre la cabeza, y provee diferentes tipos de configuraciones a nivel de programación para su correcta utilización de este modo. Lo anterior se puede apreciar en la *Figura 101*.

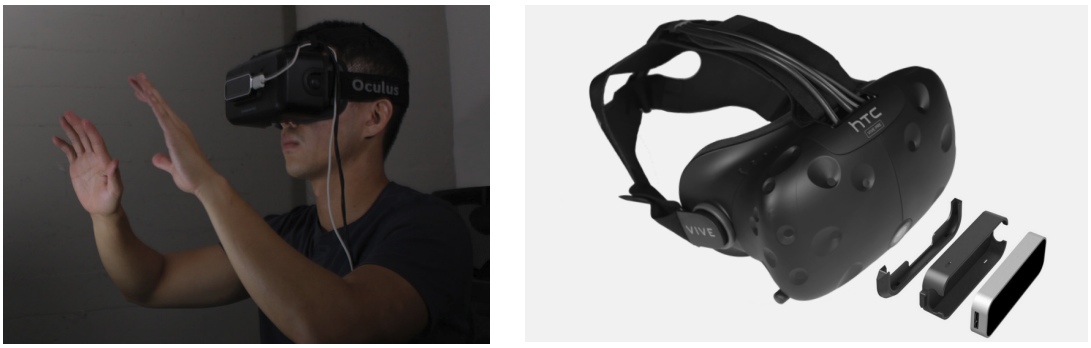


Figura 101: Montura del sensor *Leap Motion para* el casco de realidad virtual *Oculus Rift* (izq.) [97] y *HTC Vive (der.)* [59]

2. Dispositivos de realidad aumentada: si bien un casco de realidad virtual no es una posibilidad para la solución propuesta, dado que el usuario se vería inmerso en un mundo completamente virtual, perdiendo noción de dónde se encuentra la superficie de interacción así como también, los demás usuarios, sí es factible introducir nuevos elementos virtuales que se proyecten directamente sobre los objetos del mundo real que se encuentran en la escena. Esto puede ser realizado con diferentes dispositivos que han surgido recientemente como el *Google Glass* [37] o el *Microsoft HoloLens* [80] mencionados en el *Capítulo 3*, ver *Figura 22* y *Figura 24* respectivamente. Cualquier de ellos, mediante el uso de lentes especiales, permiten aumentar la realidad que ve el usuario mediante la proyección de contenido virtual adicional frente a sus ojos. De esta forma, se logra una realidad aumentada aún más inmersiva, en comparación con la simple proyección de contenido bidimensional sobre una superficie física como es el caso de la solución propuesta.

3. Extender el área de reconocimiento de usuarios: otro punto que sería conveniente para mejorar la experiencia del usuario es la integración de más sensores *Microsoft Kinect* de reconocimiento de usuario, de forma de poder capturar más espacio de la escena. De este modo, se logra un sistema de reconocimiento más flexible y genérico, que es capaz de reconocer usuarios incluso estando de espaldas o a cualquiera de los lados de la superficie de interacción (no únicamente al frente como en la solución propuesta). Como se mencionó en el *Capítulo 5*, la arquitectura que da soporte a la solución fue diseñada con el fin de permitir la integración de nuevos sensores de interacción. De todos modos, en este caso se debe de tener en cuenta que cuanto mayor sea la cantidad de sensores interactuando sobre la misma área, mayor será el ruido generado por la interferencia generada, por lo que quizás en este caso sea necesario incorporar nuevos mecanismos de reducción de interferencia o mejorar el mecanismo ya incorporado en la solución, descrito en el *Capítulo 4*. Un ejemplo de esto se puede encontrar en *RoomAlive* [132], ver *Figura 102*, en la que se puede apreciar cómo se podrían distribuir los sensores *Microsoft Kinect* con el fin de captar un mayor área de interacción. Cabe mencionar también que, si bien es posible configurar el sistema para obtener datos de sensores adicionales, se deben incorporar mecanismos para la correcta integración de los datos provistos por cada uno ellos de forma que los resultados sean consistentes.

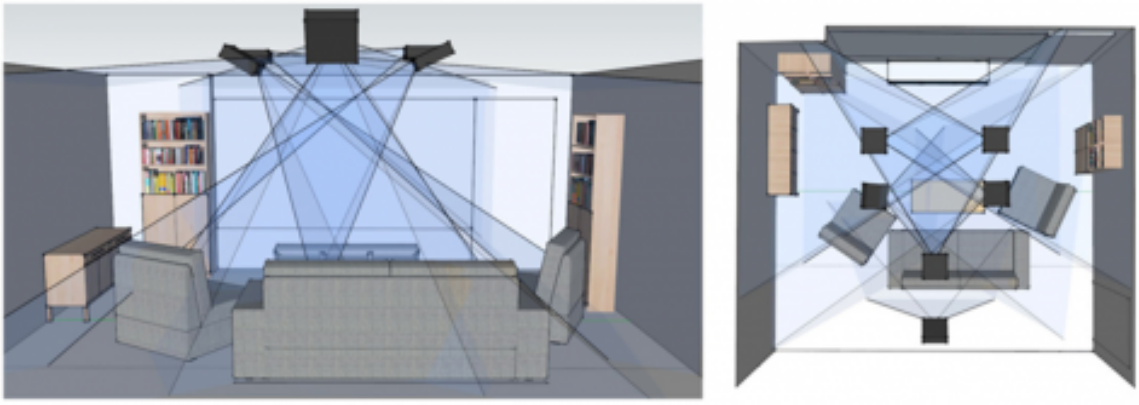


Figura 102: Distribución de sensores *Microsoft Kinect* [133]

4. Extender el área de interacción multitáctil: relacionado con el punto anterior, también se podría extender la superficie de interacción multitáctil mediante la utilización de más sensores *Microsoft Kinect* y proyectores de forma de abarcar una mayor área de la mesa tal como se puede observar en la *Figura 103*. Esto permite que la interacción multitáctil y la visualización de contenido no esté acotada a una pequeña porción de la superficie de interacción, sino eventualmente a toda la superficie.

Figura 103: Superficie de interacción multitáctil extendida

5. Proyector de mejor calidad: otra posible mejora que acrecentaría notoriamente la calidad de la visualización de los diferentes contenidos virtuales presentados sobre la superficie, es utilizar un proyector de mejor calidad que posea un nivel de luminosidad considerablemente mayor a los 15 lúmenes que posee el picoproyector con el que cuenta la solución actual (3M MPro 150). De este de modo, se mejora la calidad general de las proyecciones y adicionalmente se eliminan ciertas restricciones y limitaciones inherentes a la baja luminosidad del proyector actual, como por ejemplo, la necesidad de que la habitación esté completamente oscura para que el usuario sea capaz de visualizar de forma clara el contenido presentado. Hay que tener en cuenta que el picoproyector utilizado, si bien no presenta una buena calidad de visualización, tiene la característica de ser pequeño y liviano, lo que permite que sea colocado sin mayores inconvenientes en la estructura que da soporte a la solución propuesta. En caso de utilizar un proyector con más potencia, probablemente sea de mayor tamaño y peso, por lo que se debe

evaluar la forma apropiada de colocarlo en la estructura física junto a los demás dispositivos y sensores.

6. Superficies de interacción adicionales: la solución propuesta hace uso de una única superficie de interacción horizontal (mesa), por lo que un posible trabajo a futuro es que el sistema pueda reconocer y gestionar varias superficies de interacción con las que los usuarios puedan interactuar libremente, eventualmente compartiendo e intercambiando información entre las diferentes superficies. Así, por ejemplo, se podrían incorporar nuevas superficies como paredes, mesas adicionales, etc. En este caso, la superficie de interacción vertical representada por la pared podría utilizarse para brindar una visibilidad global del contenido hacia toda la audiencia como parte de una presentación, mientras que la superficie compuesta por la mesa podría dividirse en secciones virtuales próximas a cada usuario y presentar en cada una de ellas información referente a cada usuario particular (una agenda de contactos, eventos, calendario, etc.). Esta potencial mejora para el sistema está inspirado en el proyecto *LightSpace* ya detallado en el *Capítulo 3*. En la *Figura 104* se puede apreciar el uso de varias superficies de interacción con diferentes objetivos como parte de la solución propuesta por *LightSpace*.

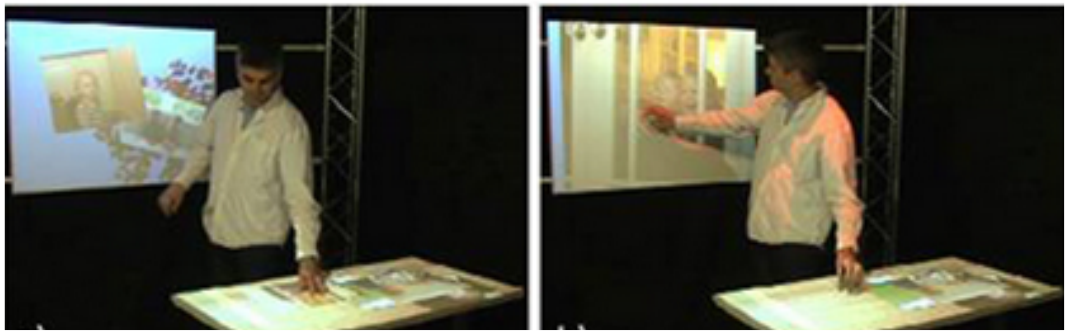


Figura 104: *LightSpace* y diferentes superficies de interacción soportadas [74]

7. Multiplataforma: si bien se probó el sistema únicamente en la plataforma *Windows*, y no se realizó ni compilación ni ejecución en otras plataformas como *Linux* y *OSX*, es posible portar el *framework* para su uso en otras plataformas de forma sencilla gracias a la utilización de bibliotecas multiplataforma como *OpenNI*, *openFrameworks*, *Boost*, etc. tal como se detalló en el *Capítulo 2*.

8. Envío y procesamiento de datos crudos: otro punto interesante mediante el cual se puede mejorar al sistema es logrando enviar al nodo de procesamiento de forma total o parcial la información cruda que se obtiene desde cada sensor utilizado. De esta forma, el sistema puede almacenarlo y, teniendo más control sobre esta información, realizar un procesamiento de más bajo nivel, lo cual en la solución actual no es posible dado que únicamente se envía al nodo *Core* información preprocesada por los diferentes *drivers* y bibliotecas de alto nivel que controlan los sensores. De esta forma, complementado con la incorporación de nuevos algoritmos de visión por computador al sistema, es posible obtener más información de la escena y brindar a las aplicaciones información más rica, como por ejemplo objetos reconocidos sobre la mesa, dispositivos particulares que estén siendo utilizados por los usuarios, gestos específicos más complejos que los usuales basados en el análisis de la forma de la mano u otras partes del cuerpo, entre otras diversas posibilidades. Como se mencionó en el *Capítulo 5*, actualmente se puede indicar al sistema que se envíe la información cruda por parte de los sensores mediante un fichero de configuración, pero esta opción fue desactivada para la solución final propuesta ya que requiere profundizar en el estudio de la compresión de datos sobre la red, debido a que la información proporcionada por los sensores es de gran tamaño y la tasa de generación es muy alta. En el caso particular del envío de datos crudos para el

reconocimiento de gestos, es posible utilizar la información ya conocida sobre los *joints* o puntos de interés de un usuario para optimizar el proceso, obteniendo la posición de los *joints* del usuario y enviando únicamente un área acotada a dichos puntos de interés de los datos crudos completos. Se podría a su vez configurar qué puntos de interés considerar dependiendo de si se desea reconocer gestos con las manos, tronco o cabeza para un posible reconocimiento facial, etc.

9. Reconocimiento de gestos híbridos: otro trabajo a futuro propuesto es brindar la posibilidad de reconocer gestos combinados en simultáneo, es decir, brindar mecanismos para detectar gestos híbridos conformados por un gesto multitáctil y un gesto espacial realizados en conjunto, es decir, al mismo tiempo. La utilidad de este tipo de gestos se visualiza mejor en el caso en que se tengan múltiples superficies de interacción y se desea, por ejemplo, mover elementos entre ellas. El reconocimiento de gestos híbridos posibilitaría entonces seleccionar un elemento mediante interacción multitáctil en la superficie origen mientras se apunta con el otro brazo en dirección hacia la superficie destino, resultando en la transferencia del elemento en cuestión desde la superficie origen hacia la superficie destino.

10. Utilización de comandos por voz: si bien no fue incorporado como parte de la solución propuesta, los sensores *Microsoft Kinect* cuentan con micrófonos que permitirían reconocer comandos por voz. Estos podrían ser utilizados para extender las posibles formas de interacción del sistema, agregando también la interacción mediante comandos de voz haciendo que los usuarios puedan comandar al sistema de acciones específicas haciendo uso únicamente de su voz. Dado que esta vía de interacción es muy poco intrusiva, no agregaría complejidad adicional a la interacción general con el sistema. Sí habría que estudiar en este caso las tasas de reconocimiento y cantidad de falsos negativos y positivos para este tipo de reconocimiento por parte de los sensores. Cabe mencionar que los *drivers* oficiales de *Microsoft Kinect* cuentan nativamente con la lógica necesaria para reconocer comandos por voz, aunque no es el caso de la biblioteca *OpenNI* utilizada.

11. Interfaz de usuario más atractiva: una mejora posible que elevaría de gran forma lo llamativo de la aplicación construida sería utilizar una herramienta como *Unity*. Mediante ella se podría construir una interfaz más rica, y con más realismo siendo así más atractiva para un usuario final.

7.3. Posibles usos

Si bien como se planteó el objetivo era desarrollar un *framework* para la interacción en una oficina futurista, lo desarrollado puede ser llevado a casi cualquier tipo de ambiente y/o utilidad. Por ello, diferentes tipos de aplicaciones pueden sacar provecho de la combinación de las técnicas de interacción multitáctil e interacción espacial. Aplicaciones desde lo que tiene que ver con una oficina futurista donde pueda interactuar con por ejemplo, la agenda de reuniones, un escritorio virtual, presentaciones etc. como hasta aplicaciones mas relacionadas a la salud para ayudar a la percepción motora donde el usuario tenga que realizar acciones con sus manos para seleccionar objetos etc. También se puede llevar al área culinaria por ejemplo para activar la llamada a un mozo disponible para la mesa o presentación de la carta virtual sobre la mesa, incluso el área de los videojuegos. Hay un sinfín entonces, de posibilidades donde se puede sacar un buen provecho del trabajo desarrollado.

Anexo A: Principios para el buen diseño en *HCI*

Unos de los principios más importantes para el buen diseño son los detallados por *Donald A. Norman* [143], concebidos para el diseño general y aplicables a cualquier disciplina, en particular, a los sistemas interactivos basados en *software*. Se enumeran a continuación algunos de dichos principios:

1. Visibilidad: hace referencia a hacer que todos los elementos involucrados estén visibles, incluyendo el modelo conceptual del sistema, las acciones alternativas y los resultados de las acciones, haciendo que sea fácil determinar qué acciones son posibles de realizar en cada momento. En definitiva, que con solo mirar las salidas del sistema, el usuario pueda determinar cuál su estado y las opciones de acción. Este principio puede ser trivial pero en muchos casos, no es tenido en cuenta. Algunos ejemplos usuales que pasan por alto este principio son puertas en las que no se sabe para qué lado abren o cierran, o botones para los que su ícono no es lo suficientemente descriptivo por lo que se debe revisar la descripción asociada.
2. Buena topografía: por topografía se refiere a la relación entre dos cosas, como pueden ser por ejemplo, el movimiento de un volante y el correspondiente giro de un coche, por lo que el principio refiere a que sea posible determinar las relaciones entre los actos y los resultados, entre los mandos y sus efectos y entre el estado del sistema y lo que es visible. Algunos ejemplos usuales que pasan por alto este principio son botones que no parecen ser botones o una canilla que bombea agua fría si es abierta con el mando para la derecha y caliente si se abre para la izquierda, pero en su leyenda de uso indica lo contrario (rojo del lado derecho y azul del lado izquierdo).
3. Retroalimentación (*feedback* y *feedforward*): en lo que respecta al *feedback*, implica que el usuario reciba una retroalimentación completa y constante acerca de los resultados de sus actos. Cada acción con el sistema debe tener una clara reacción, sea esta brindada de forma táctil, auditiva, verbal, visual o una combinación de todas ellas. Por otro lado, *feedforward* refiere a informar al usuario de los efectos de una acción antes de que la realice. Estas características incentivan además un diseño que permita que sea fácil recuperarse ante algún error. Un claro ejemplo que no sigue este principio y es muy común es un botón que aparenta no hacer nada al presionarlo dado que no arroja ningun mensajes de resultado. Esto es crítico en un sitio de banca electrónica por ejemplo, dado que el usuario puede estar haciendo una transferencia varias veces pensando que el botón no funciona, cuando en realidad, el único problema es que falta la retroalimentación a de la acción correspondiente.
4. Un buen modelo conceptual: hace referencia a que el diseñador debe proporcionar al usuario un buen modelo conceptual, coherente en la exposición de las operaciones y los resultados y con una imagen del sistema coherente y pertinente. Cuando las personas usan un objeto o una interfaz generan un modelo mental de cómo ésta funciona en base a la experiencia. La correctitud de este modelo dependerá de los tres puntos anteriores, es decir, que se visualicen de forma sencilla las posibles acciones, que sea topográficamente correcta y que provea buena retroalimentación. Otro punto importante para crear buenos modelos conceptuales es utilizar prestaciones y limitaciones siempre que sea posible, mostrando qué es lo que se puede hacer y lo que no. Un ejemplo de esto son las tijeras, que cuentan con uno de los agujeros más grande que el otro como forma de indicar que hay una única forma de utilizarla (el dedo pulgar colocado en el agujero más grande y el índice en el más pequeño).

Por otra parte, una de las características básicas que las computadoras, y en particular, los sistemas interactivos, deben poseer para incrementar su capacidad de interacción es la presencia, debiendo ser capaces de reconocer cuándo un usuario entra o sale del área de interacción y realizar así, acciones que motivan o suplen una necesidad del usuario mejorando la comunicación [53]. Es importante también la característica de ubicación, que permita llevar el seguimiento de la posición y movimiento de los usuarios,

con el fin de brindar información de forma ágil y de fácil acceso al contenido. Adicionalmente, el entender los movimientos corporales del usuario permite reconocer sus intenciones y generar respuestas que satisfagan las exigencias de dichas expresiones, estableciendo así, una relación más transparente entre el humano y la máquina. La interacción puede durar unos segundos, minutos o incluso horas, por lo que es importante que los usuarios se sientan a gusto utilizando la interfaz, que no se sientan cohibidos si otras personas los observan utilizándola y que no pierdan el interés mientras aprenden a interactuar. Así, cuanto más transparente y flexible sea la interfaz, permitiendo averiguar qué está pasando y cómo se utiliza de manera sencilla, más intuitivo resultará para los usuarios.

Otro punto que repercute fuertemente en la calidad de la interacción de los usuarios con el sistema es el contexto o espacio físico en el que se está inmerso, ya que influye directamente en su comportamiento, su estado de ánimo y su concentración, por lo que se debe diseñar la interacción teniendo claro cuál será el contexto en el que el usuario estará utilizando el sistema interactivo.

Las interfaces naturales buscan permitir a los usuarios que puedan comunicarse con los computadores de la misma forma que lo harían con otros seres humanos [151]. Esto se logra gracias a la utilización de la capacidad de cómputo creciente que es utilizada para comprender qué es lo que el usuario intenta realizar. Para esto, se requiere que los computadores entiendan aspectos comunes de las comunicaciones entre seres humanos, como pueden ser gestos, el habla, la visión y el tacto. Comprender esto permite que los usuarios se alejen de la interacción usual con *mouse* y teclado para poder interactuar en cualquier forma y en cualquier momento, surgiendo así el concepto de computación ubicua.

Las personas utilizan gestos, expresiones y movimientos de forma natural para poder comunicarse entre ellas. De hecho, se ha demostrado que los niños se logran comunicar mediante gestos antes de que aprendan a hablar. A menudo, se utilizan gestos acompañados o no mediante comunicación del tipo verbal. Si se lleva esto al terreno de la comunicación con las diferentes aplicaciones se podría llegar a establecer una comunicación con ellas sin la necesidad de ningún tipo de dispositivo. De esta forma, el uso de gestos y movimientos naturales e intuitivos del cuerpo, podrían ser utilizados como comandos o interfaces de comunicación para operar distintos dispositivos inteligentes utilizados en las actividades diarias a la cuales se enfrentan los usuarios. Este tipo de interfaces gestuales son de aplicación cuando se requiere alguna libertad de movimiento o una sensación de inmersión, como es el caso de ambientes de realidad virtual, tecnología que actualmente está siendo ampliamente utilizada en una gran cantidad de aplicaciones.

El éxito de un sistema basado en interacción natural está en cómo éste influencia a las personas que lo experimentan, haciéndole creer que están lidiando con algo que realmente está vivo y de esta forma, el usuario experimenta cierta magia que obedece a su control, buscando sorprender, divertir o resolver preguntas del usuario [142]. Se debe hacer creer al usuario que no están manipulando medios digitales abstractos, sino objetos de la vida real, logrando reducir la carga mental requerida para interaccionar con el sistema y consecuentemente aumentar la atención al contenido por parte de los usuarios.

La interfaz natural activa las dinámicas cognitivas de la persona que hace uso de ella, lo que aumenta el nivel de atención en el contenido y reduce la carga necesaria para realizar las tareas. Cabe mencionar que, la tecnología no define la naturaleza del sistema, sino que es simplemente una herramienta para crear un espacio comunicativo en comparación con las interfaces comunes. Es por ello que, las interfaces deben ir en dirección de una mayor coherencia con las características perceptivas de los humanos.

Es preciso incluir también, los principios de diseño estudiados por *Alessandro Valli*, que si bien algunos puntos ya fueron mencionados a lo largo del presente documento, se enumeran a continuación algunos conceptos básicos sobre los cuales se funda la interacción natural [156]:

1. Natural: a pesar de los miles de años de evolución humana son los primeros días de la vida de una persona los que dictaminan qué es natural y qué no. Se determinan como naturales a

aquellas acciones para las cuales el ser humano está hecho por naturaleza, aquellas implícitamente grabadas en su cuerpo y mente, así como también, las actividades realizadas cotidianamente.

2. Interacción: se define como la acción mutua o recíproca entre objetos inanimados o personas. Si un sistema simula objetos físicos y además es capaz de percibir acciones e intenciones humanas, las personas podrán interactuar simplemente utilizando los mecanismos que utilizan en la vida real, la cual es la clave para un buen nivel de satisfacción del usuario y es a lo que se debe apuntar, es decir, a definir formas de interacción lo más parecidas a los mecanismos de interacción que utilizan las personas en su vida diaria.
3. Cognición: es recomendable reducir al máximo la carga cognitiva necesaria para interactuar con el sistema. Esta carga cognitiva se define como la cantidad total de actividad que se encuentra en la memoria de trabajo (o memoria a corto plazo) del usuario en un momento. Este tipo de memoria es la utilizada para razonar, pensar y aprender y por lo tanto, es la que más carga conlleva cuando se está frente a un sistema interactivo. De esta forma, lo ideal es representar la información mediante estímulos elementales sensoriales, de forma que no requiera gran cantidad de actividad mental.
4. Contexto: no solo el sistema en sí mismo debe ser construído pensando en cómo influenciará las acciones de los usuarios sino también, el contexto y estructura en el que este está inmerso. Los humanos son seres altamente perceptivos y cada detalle en el área cubierta por sus sentidos puede ayudar en pos de una mejor experiencia. A su vez, un sistema informático puede ser muy complejo de entender y percibir, por lo que se debe simplificar la información provista mediante agrupaciones y jerarquías, de forma tal que esté clara la ubicación y contexto de cada elemento y el usuario pueda moverse a elementos similares o relacionados haciendo el menor uso posible de su capacidad cognitiva.
5. Atención: el sistema debe ser capaz de captar y mantener la atención de los usuarios, donde la interfaz debería ser tan invisible como sea posible, de forma de reducir el cansancio y la distracción de los usuarios.
6. Social: la interacción con el computador usualmente hace que se pierda la relación entre la persona y el mundo real, siendo lo que está en frente al usuario su único foco de atención. Por ello, se debe apuntar a una interfaz natural, y que ésta incite a una interacción semejante la relación normal entre personas, sea en contacto directo o indirecto con el sistema, permitiendo así que se puedan mover libremente a lo largo de todo el espacio interactivo y que no estén obligadas a sentarse frente a un monitor ni a sentirse atados a algún dispositivo particular.
7. Competencia y colaboración: la restricción a que sea una sola persona la que pueda interactuar con un sistema interactivo no está bien aceptada por los usuarios, siendo ideales las soluciones más flexibles en cuanto a cantidad de usuarios concurrentes utilizando la interfaz. Debido a la necesidad de múltiples personas con diferentes objetivos que desean hacer uso del sistema, pueden aparecer conflictos en su uso. Estos problemas se pueden solucionar por ejemplo, mediante espacios que permitan tanto comportamientos colaborativos donde los usuarios pueden trabajar juntos en pos de un objetivo común, como también competitivos donde los recursos deben dividirse entre las diferentes tareas.
8. Estética y emoción: todas las personas son naturalmente atraídas por el concepto de la belleza y es por ello que, cualquier interfaz natural debería de parecerse a una instalación artística, enfocándose entonces al concepto de lo "bonito que se ve" ya sea una mesa inteligente, un mapa interactivo, etc. Generalmente, las instalaciones del tipo minimalistas y sencillas funcionan mejor que las complejas, permitiendo hacer hincapié en el contenido brindado por el sistema. Sin embargo, un espacio interactivo más elaborado y complejo estéticamente, provoca más emoción en las personas que cualquier otra sistema basado en computadores, y aspectos emocionales como el asombro, el miedo o la admiración repercuten en la percepción y el aprendizaje de los usuarios. Es por esto que, el correcto uso de los recursos que aportan a la esteticidad del sistema, como son aspectos lumínicos, de video y de audio, pueden hacer que los usuarios se sientan emocionalmente más atraídos e interesados con el sistema.
9. Movimiento: una escena en movimiento es capaz de atraer mucho más la atención de un

usuario en comparación con una escena estática, razón por la cual el movimiento es esencial como parte de la interacción. Adicionalmente, esta noción de movimiento puede ser bidireccional, en el sentido de que el sistema puede atraer a los usuarios moviendo objetos visuales y por otro lado, el sistema puede ser capaz de detectar y reconocer movimientos voluntarios de los usuarios y actuar consecuentemente. El objetivo es quitar cualquier nivel de mediación entre las personas y los computadores, convirtiendo al medio en un elemento activo.

10. Libertad: las personas deben sentirse libres, pudiendo moverse libremente dentro o alrededor del espacio interactivo que corresponde al sistema, sin restricciones aparentes o múltiples reglas a seguir. A su vez, la interacción no debe ser mecánica en el sentido de que los usuarios no deben sentirse como parte de un mecanismo, sino que el sistema debe brindar la cantidad suficiente de grados de libertad como para asegurar que el usuario pueda descubrir, explorar e incluso fallar y aprender de los errores mientras interactúa con el sistema.
11. Espacio físico: debido a que la interacción ya no está más restringida a un escenario reducido, como por ejemplo únicamente al uso de un teclado y *mouse*, la interfaz apunta a abarcar largos espacios físicos como son proyecciones en paredes, pisos y otras superficies, lo que hace que el espacio se convierta en un elemento muy influyente en cómo es percibido por el usuario del sistema, y de esta forma se apunta a que el usuario sea capaz de poder interactuar con todo el entorno en donde se encuentra inmerso.
12. Integración: un sistema que quiera hacer uso de interacción natural debe de lograr integrar diferentes componentes como objetos físicos, proyectores, pantallas, cámaras, sensores, computadoras, etc. y lograr que todos ellos se integren de forma eficiente con el objetivo de que desaparezcan como objetos individuales e independientes para el usuario, generando así la ilusión de que se está interactuando con una única entidad. Adicionalmente, con este mismo fin, se requieren tiempos de respuesta muy bajos, ya que la latencia en las respuestas puede hacer que la interacción se vuelva frustrante. Por ejemplo, para comandos discretos como un gesto de apuntado un tiempo de respuesta de 600 milisegundos es tolerable, mientras que para comandos continuos como puede ser un *drag and drop*, se debe garantizar un tiempo de respuesta no mayor a 150 milisegundos.

Por otra parte, existen ciertos conceptos estrictamente relacionados a cómo los usuarios perciben la interacción con el sistema. Se enumeran a continuación algunos de ellos:

1. Percepción: la capacidad de los computadores de entender diferentes posibles entradas definen los posibles niveles de interacción. Las tecnologías basadas en sensores no deberían ser consideradas como periféricos adicionales al sistema sino como parte la inteligencia global, la cual debe ser completamente robusta y confiable. El hecho de detectar cuando una persona está frente al sistema y cuando no, es la capacidad más básica que permite interactividad, ya que ayuda a la comunicación, brindando información selectiva.
2. Gestos: son movimientos corporales expresivos realizados con los dedos, las manos, los brazos o cualquier otra parte del cuerpo. La mayoría de los gestos que realizan los humanos son gesticulaciones que acompañan el habla y pequeños movimientos que brindan *confort* e incluso equilibrio. Estos gestos, permiten interactuar con el entorno y por lo tanto también, con cualquier tipo de computador. No sólo los gestos que impliquen un movimiento continuo expresan una intención del usuario, sino que incluso los gestos clasificados como estáticos (poses o posturas concretas) permiten expresar un estado de ánimo, una intención o un interés por parte del usuario. En cuanto al reconocimiento de los gestos, es útil el conocimiento previo del contexto en el que se dará el gesto con el objetivo de construir buenas heurísticas y algoritmos de reconocimiento. Dichas heurísticas, permiten reconocer gestos basados en movimientos simples e intuitivos que no incrementan la carga cognitiva necesaria para la interacción ni fatigan al usuario. Una clasificación de los gestos en lo que respecta a la interacción es la que se enumera a continuación:

a. *Diectic gestures*: son aquellos que se asocian a acciones de señalamiento, como por ejemplo, señalar con el dedo índice. En otras palabras, son acciones que pretenden indicar la posición de un objeto.
b. *Pathic gestures:* son aquellos que representan un camino o una dirección, como por ejemplo, la acción de caminar a la izquierda.
c. *Mimic gestures:* son aquellos que representan acciones que imitan otros movimientos, como por ejemplo, agitar los brazos imitando el movimiento de las alas de un pájaro.
d. *Ergotic gestures:* la ergótica está asociada con el trabajo que realizan las personas y su capacidad de manipular el mundo físico con el objetivo de satisfacer determinadas necesidades propias. Por ello, gestos naturales como empujar, agarra o soltar objetos son categorizados como gestos ergóticos.

3. *Computer vision:* hoy en día, las cámaras digitales son excelentes dispositivos de entrada debido a su bajo costo, su tamaño y a que pueden ser puestas lejos de los usuarios, permitiendo así, un sensado no intrusivo. Mediante el uso de dichas cámaras se pueden utilizar técnicas de visión por computadora en tiempo real con el fin de procesar los videos provistos y extraer información útil para la interacción. Éstas técnicas de visión por computadora deben manejar el gran flujo de datos proveniente de estos dispositivos y para ello deben emplear algoritmos eficientes que permitan analizar la información, haciendo que la inteligencia de sensado se mueva desde el dispositivo *hardware* hacia el código del *software*.
4. Visible e infrarrojo: el ojo del ser humano tiene la capacidad de captar el espectro de luz que va desde la luz roja a la luz violeta, mientras que toda luz que escape de dicho espectro es invisible. El espectro de luz más allá de la luz violeta es la denominada luz ultravioleta y el espectro de luz por debajo de la luz roja se denomina luz infrarroja; ambos invisibles para el ojo humano. A la hora de escoger un sensor se pueden utilizar sensores de luz visible como las cámaras, las cuales permiten el seguimiento e interpretación de cierta escena basada en información de color, como por ejemplo detección de piel o de objetos de cierto color, aunque tienen algunas restricciones, dado que están fuertemente ligadas a las condiciones lumínicas. Por otro lado, se pueden utilizar sensores basados en espectros de luz no visible, los cuales no proveen información cromática pero no son sensibles a la luz externa, pudiendo incluso utilizarse en entornos completamente oscuros, lo cual permite más libertad en lo que respecta a los requerimientos arquitectónicos de la solución.
5. Retroalimentación: las acciones humanas son guiadas por la retroalimentación, conocido comúnmente como *feedback*, que brinda el sistema, por lo que un sistema puede mejorar notablemente la forma en que los usuarios interactúan si provee una correcta retroalimentación visual, auditiva y/o táctil. Dicha retroalimentación debe proveer un *feedback* inmediato, tan pronto como el sistema reconoce una nueva interacción del usuario a modo de brindar una mayor fluidez.
6. Estímulos: toda interacción, sea del tipo que sea, tiene un comienzo, por lo que el usuario debe ser capaz de reconocer cómo comenzar a interactuar con el sistema. Por ello, la interfaz del sistema debe contener todas las pistas necesarias para una interacción satisfactoria. El trabajo más difícil es definir el estímulo inicial, es decir, aquella pista que hace que el usuario realice la primer acción voluntaria con el sistema, pero una vez que la interacción está en marcha será más fácil aprender las características adicionales de interacción.
7. Intuitividad: la intuitividad de una interfaz que pretende ser natural es una característica obligatoria que debe estar presente en cualquier sistema. Cuanto más transparente sea la interfaz, es decir, cuanto más fácil le sea al usuario descubrir lo que está pasando en el sistema, más rápido aprenderá cómo interactuar intuitivamente, logrando de esta forma que el usuario no se frustre y no tienda a rechazar el sistema.
8. Imitación: las personas aprenden muy bien mediante la imitación y el ejemplo, por lo que la mejor forma que tienen las personas de aprender a utilizar algo es ver a otros utilizándolo. Las personas que están observando a otro usuarios utilizar el sistema aprenden las posibilidades de

interacción mientras disfrutan de la presentación de contenido, por lo que también es importante que quienes lo están utilizando se sientan confortables siendo observados gracias a una interfaz intuitiva, fácil de aprender y que no requiera movimientos exagerados que puedan avergonzar a quien la utiliza.

Por otra parte, a continuación se enumeran ciertos conceptos estrictamente relacionados a cómo la información debe ser presentada a los usuarios por parte del sistema y a cómo esto repercute positiva o negativamente en la experiencia percibida:

1. Estilo: la naturaleza de los sistemas interactivos sugiere quebrar las reglas que gobiernan los interfaces gráficas estándar basadas en *scrollbars*, ventanas, botones, etc. debido a que deben ser estéticamente efectivos y lograr una experiencia de inmersión como forma de captar la atención del usuario en el contenido que se presenta. En este sentido, las interfaces naturales son más cercanas a un videojuego moderno que a un sistema operativo basado en interfaces gráficas estándar, debiendo presentar los contenidos digitales en diferentes tamaños y formas que hagan que se comporten como objetos de la vida real.
2. Física: el objetivo de la interacción natural es permitir a las personas interactuar con objetos digitales de la misma forma que lo hacen con los objetos reales, con el fin de minimizar el esfuerzo cognitivo necesario para la interacción. Por ello, los objetos digitales deben parecerse y comportarse lo más posible como si fueran reales. Es preciso tener en cuenta que los objetos físicos obedecen a las leyes físicas, mientras que los objetos virtuales no, y esto se debe simular para cuando los objetos virtuales son manipulados por personas. Usualmente, los cambios visuales en las interfaces de usuario son repentinos e inesperados, sorprendiendo a los usuarios y forzandolos a alejarse mentalmente de su tarea para intentar entender qué es lo que está pasando en la interfaz. Las interfaces naturales introducen restricciones físicas simuladas que permiten controlar las transformaciones de los objetos digitales, de forma que el usuario pueda seguir los cambios y entender más fácilmente qué es lo que está pasando. Esto puede lograrse mediante movimientos fluidos y acercamientos, de forma que los objetos no aparezcan y desaparezcan repentinamente. A su vez, es preciso simular de forma real el movimiento de los objetos tal como si fueran objetos reales con masa, haciendo que no se muevan de forma lineal, sino que su velocidad se incremente desde cero hasta cierta cantidad de forma paulatina. Incluso, algunos autores proponen implementar interfaces que sigan los criterios de los dibujos animados, en el sentido de que los movimientos y las dinámicas sean exagerados con el objetivo de hacerlos más comprensibles. Así, por ejemplo, los objetos antes de moverse en una cierta dirección se mueven un poco en la dirección opuesta y se hacen vibrar levemente luego de una parada repentina.
3. Organización espacial: la información espacial, es decir, la información representada en base al lugar en el que se encuentra un objeto, es una manera muy poderosa de organizar contenidos. Si todas las relaciones entre los objetos deben ser constantemente visualizadas en la memoria de los usuarios, se incrementa el esfuerzo cognitivo requerido por la interacción, por lo que se debe establecer una relación semántica entre el significado de la información y su representación. Por lo tanto, se recomienda que contenidos similares estén cercanos y se utilicen colores y formas para representar agrupación y clasificación conceptual.
4. Menos es más: la presentación de contenido debe ser lo más sencilla y clara posible para facilitar la explotación de la interfaz y reducir la carga cognitiva del usuario. Se debe preservar la manipulación directa de los objetos, sustituyendo íconos y símbolos por el objeto en sí mismo o su representación más sencilla.
5. Multimedia: se debe hacer un uso eficiente de cada tipo de recurso multimedia para lograr una comunicación exitosa, utilizando por ejemplo 2D para mostrar contenido bidimensional como fotografías o videos y 3D para presentar objetos que no existen realmente. El renderizado puede ser realizado en tiempo real para brindar una interactividad completa, ya que en este caso, el aspecto de cada *frame* es afectado directamente por la entrada del usuario. Adicionalmente, se puede mejorar la experiencia visual agregando información auditiva,

brindando retroalimentación mediante sonidos para ciertas acciones reconocidas como la selección o el *drag and drop* de elementos de la interfaz.

6. Tangible: con el objetivo de permitir a las personas interactuar naturalmente con el sistema, es necesario no introducir dispositivos que los usuarios deban llevar puestos o utilizar para ser reconocidos por el sistema. De todas formas, en algunos casos los objetos físicos pueden favorecer la interacción, como en las interfaces tangibles, que utilizan objetos palpables como dispositivos de entrada, proveyendo un mapeo directo entre estos objetos físicos y el medio digital. Los datos se vuelven tangibles, pudiendo ser modificados manipulando el objeto que los representa. Así, un anillo puede transformarse en un lente para leer un mapa digital o un elemento o producto colocado sobre el escritorio puede ser rodeado de proyecciones que brinden información sobre sus propiedades y atributos.
7. Juego: en una interfaz natural está siempre presente una dimensión jugable, ya que una forma completamente inusual de interactuar con un computador puede ser visto como un juego en sí mismo, donde el factor entretenimiento es el descubrimiento del comportamiento del sistema mediante intentos hasta obtener la reacción deseada. La interfaz debe ser lo suficientemente sencilla como para ser intuitiva, pero lo suficientemente compleja a modo de ocultar ciertas características avanzadas que puedan ser descubiertas por casualidad. El juego tiene el poder de ganar la atención de los usuarios de forma que no se aburran, ayudando a su vez, que el posible cansancio natural de la interacción pase desapercibido.

Si bien la tecnología de hoy en día promueve nuevas formas de interacción, éstas se encuentran limitadas por las restricciones humanas existentes. *Bill Verplank* [93] propone el modelo de interacción persona-computador ilustrado en la *Figura 105* en la que describe a las interacciones de los usuarios como una función que depende de los siguientes tres factores humanos:

1. *Input efficacy:* cómo y qué tan bien los usuarios perciben lo que el sistema está comunicando.
2. *Processing model:* cómo y qué tan bien los usuarios entienden y piensan sobre la comunicación establecida.
3. *Output efficacy:* cómo y qué tan bien los usuarios se comunican en respuesta con el sistema.

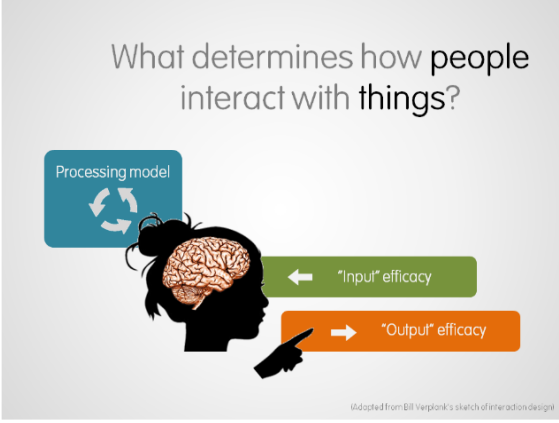


Figura 105: Factores humanos relevantes en la interacción [93]

Por otro lado, *Jacob Nielsen,* establece cuatro consideraciones adicionales a tener en cuenta si las interacciones son basadas en gestos del tipo 3D:

1. *Output efficacy*: los gestos realizados sobre las superficies no son extremadamente precisos, y mucho menos lo son los gestos realizados en el aire. Es por esto que, no se debe esperar mucha precisión, ya que esto acarrearía muchos errores. Por lo antedicho se debe manejar un margen de error aceptable al momento de reconocer estos gestos, es decir que, los gestos deben poseer una granularidad alta.
2. *Processing efficacy:* cuando se establece, por ejemplo, una comunicación mediante comando

por voz, los usuarios tienden a personalizar aquello con lo que están interactuando, por lo cual, si el usuario recibe respuesta del sistema, la personificación se asienta más en su mente. Es por esto que, la personalidad de un sistema debe ser adecuada a su función. Incluso si ciertos programas de uso cotidiano agregaran el uso de comandos por voz de forma amigable y entendible harían que los usuario no sean tan reacios a su uso. A modo de ejemplo, el sistema *Siri* de *Apple* permite realizar tareas comunes como agendar un evento mediante comandos por voz, tornando la interacción hasta divertida dado el sentido del humor con el que se dotó al sistema, logrando de este modo que el usuario tenga una afinidad mayor con el sistema.

3. *Output efficacy:* gestos repetitivos o prolongados hacen que el usuario se agote, determinando que sus músculos se contraigan y provocando que precisión de sus gestos decremente afectando de esta forma la performance del sistema. Por ello, el sistema no debería requerir a los usuarios que realicen gestos de forma repetida o que se prolonguen demasiado en el tiempo.
4. *Input and output efficacy:* el usuario no va a adoptar un sistema que lo haga sentir ridículo en público. Por ejemplo, un sistema que sea para un contexto de oficina, no debería requerir que realice largos gestos o tener que pronunciar en voz alta comandos extraños. Sin embargo, incluir gestos y comandos torpes en una aplicación orientando al entretenimiento infantil puede llegar a ser más acertado. Es por esto que, los comandos deben ser adecuados según el contexto en el que van a ser aplicados, esto es, adecuados al ambiente en el cual se encuentra inmerso el usuario.

Anexo B: *HCI* como disciplina

Para que los aspectos estudiados por la *HCI* concluyan en buenos resultados en cuanto al diseño de la interacción entre el humano y el computador, debe ser complementados con conocimiento de otras disciplinas, a modo de complementar el rol del ingeniero de *software* con el objetivo de producir sistemas que recojan todas y cada una de las necesidades de los usuarios, el contexto en el que se encuentra y la forma en que utilizarán el sistema. Es recomendado entonces, la formación de equipos multidisciplinarios de desarrollo, conformados por técnicos de diferentes disciplinas, las cuales se dividen en las áreas comunes de conocimiento ilustradas en la *Figura 106*.

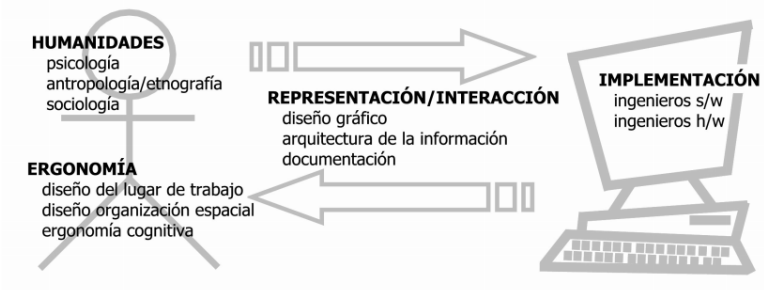


Figura 106: Áreas de conocimiento y su papel dentro de un sistema interactivo [79]

La parte humana de la interacción, es decir, lo referente al usuario, es analizado por las ciencias de humanidades y la ergonomía. Las ciencias de humanidades, en términos generales, son interpretaciones intelectuales de distintos aspectos de la experiencia humana que engloba disciplinas como la historia, la filosofía, las ciencias sociales, la psicología y la antropología, siendo estas últimas las que mayoritariamente suelen estar presentes en los equipos de *HCI*. Por otro lado, la ergonomía es quien estudia las características físicas de la interacción (entorno físico, controles, pantallas, etc.), por lo que permite diseñar la solución acorde al contexto en el que se encuentra inmerso el sistema, haciendo de esta forma que se incremente la eficiencia de la interacción y que se simplifiquen las tareas, logrando una mayor sensación de *confort* por parte de los usuarios. Por otro lado, el diseño es la disciplina que ayuda a que el sistema sea visualmente agradable, por lo que es también esencial para lograr sistemas usables. Finalmente, la parte técnica es estudiada por la ingeniería de *software* y la programación, las que permiten aplicar técnicas de diseño y desarrollo de *software* para construir la solución siguiendo buenas prácticas. Como parte de estas disciplinas entra también la inteligencia artificial, la cual permite dotar al sistema de cualidades que permitan simular la inteligencia humana con el objetivo de generar una solución más amigable para el usuario, que pueda ser percibida de forma lo más similar a un humano posible.

De esta forma, la *HCI* se convierte en una disciplina naturalmente multidisciplinaria, que requiere de otras áreas de conocimiento como la psicología cognitiva, la lingüística y la informática, para realizar un correcto análisis y diseño de interfaces entre el hombre y la máquina, esto es, la interfaz de usuario [39]. La psicología cognitiva, aunque no lo parezca, es una de las principales áreas de conocimiento de *HCI*, ya que estudia la percepción, la memoria, los modelos mentales y el modo en que las personas obtienen, perciben y procesan la información, puntos esenciales para conocer cómo utilizan una interfaz. Los avances en la psicología cognitiva son justamente quienes permiten el desarrollo de sistemas interactivos cada vez más adaptados a los usuarios. Si bien como disciplina surge en los años 50, no fue hasta la década de los 80' que comenzaron a surgir publicaciones científicas y diversos estudios en el área de HCI, así como también evidenciar avances en sus aplicaciones prácticas debido a la aparición de los computadores personales dirigidos a usuarios finales no expertos en informática y la necesidad de facilitar su utilización.

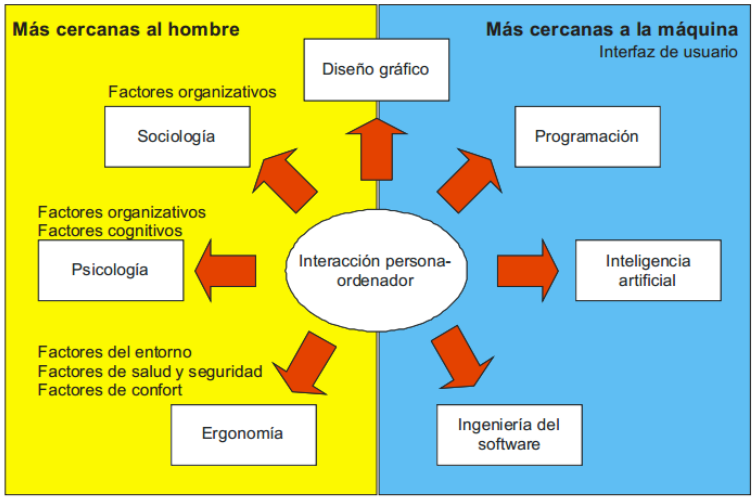


Figura 107: Principales disciplinas involucradas con la Interacción Persona-Computadora [30]

Sin embargo, es muy difícil formar un verdadero equipo multidisciplinario de las características del necesario para llevar a cabo el desarrollo de un sistema usable centrado en el usuario con la meta prioritaria de mejorar sus parámetros de usabilidad y accesibilidad. La razón de esto es que existen restricciones y limitaciones de comunicación entre los diferentes involucrados debido a que provienen de ramas disciplinarias complementamente diferentes y es difícil adaptarse a los diferentes modelos mentales presentes en un equipo multidisciplinario de este tipo. Cada persona de cada una de las múltiples disciplinas dispone de una base cognitiva diferente; si se pretende que las personas de las diferentes disciplinas utilicen por ejemplo, solo los formalismos de la ingeniería de *software*, estos serán lo suficientemente desconocidos por el resto como para provocar un distanciamiento que no beneficie al avance del proyecto. Por el contrario, si desarrollamos el sistema sin utilizar estos formalismos también repercutirá negativamente, dado que éstos constituyen el mejor soporte metodológico implementado y más utilizado históricamente. La mejor opción es la integración de ambas aproximaciones, aprovechando los beneficios de cada una de ellos, utilizando técnicas basadas en materiales de uso cotidiano y/o descripciones en lenguaje natural ampliamente conocidas por todos los participantes, independientemente de su disciplina, y en general cualquier otro formalismo que sea lo suficientemente intuitivo como para ser comprendido por el resto del equipo. De todas formas, cabe mencionar que cada integrante debe utilizar las técnicas propias de cada disciplina, ya que esto garantiza la resolución de cada parte individual del problema, teniendo siempre en cuenta no utilizar técnicas lo suficientemente específicas en cierto aspecto de forma tal que aísle al resto de los integrantes del equipo. En definitiva, es cuestión de encontrar un balance entre las tareas que cada disciplina debe realizar para llevar a buen puerto el desarrollo del sistema de principio a fin.

Se debe entonces utilizar un modelo de proceso que cubra el hueco, conocido como "*HCI-SE gap*", existente entre la ingeniería de *software* por sí sola y la disciplina Interacción Persona-Computadora, de forma tal de disponer de un marco común que asegure tanto calidad de código como usabilidad y accesibilidad del sistema construido. Uno de los modelos que surgen para paliar este problema es el presentado en la *Figura 108*, donde se aprecian las etapas usuales del proceso de desarrollo (análisis de requisitos, diseño, implementación y lanzamiento), pero además, se agrega un fuerte componente de prototipado y evaluación que termina generando *feedback* a las demás etapas en base a pruebas y diversas tareas centradas en el usuario. Este modelo está basado en que muchas veces se suele creer que porque un desarrollador piense que cierta funcionalidad es *user-friendly*[9], efectivamente lo sea, ya que quien tiene la potestad de calificar algo como *user-friendly* es el usuario que interacciona con el sistema. El modelo deja entonces bien clara la posición del usuario, quien influye directamente en las decisiones tomadas en las distintas etapas que conforman el proceso, estando en el centro del propio desarrollo, haciendo que este

[9] *User-friendly* es un término utilizado comúnmente para referirse a un objeto o componente que tiene la característica de ser amigable para el usuario, es decir, que es de fácil uso y/o aprendizaje por parte de quienes lo utilizan

modelo se conozca como diseño centrado en el usuario, del inglés *user-centered design*.

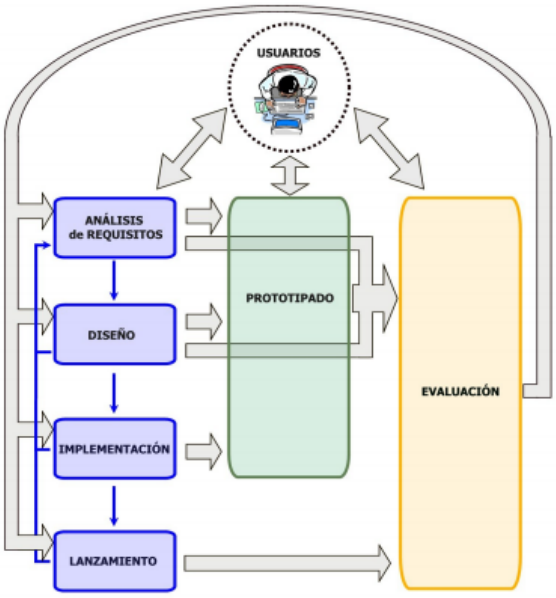


Figura 108: Diagrama de un proceso de desarrollo [79]

El diseño centrado en el usuario se define como una actividad multidisciplinaria que tiene como objetivo mejorar la productividad, la efectividad, la salud y la seguridad de los usuarios mediante la incorporación de conocimiento sobre los factores humanos, psicológicos y ergonómicos, pudiendo así generar un producto acorde a sus necesidades, expectativas, motivaciones y capacidades. En definitiva, es un tipo de diseño que está centrado en la creación de productos que resuelvan necesidades, deseos y limitaciones concretas de los usuarios finales, y que está enfocado en brindar la mejor experiencia de uso posible. Hoy en día, existen incluso estándares, como el *ISO 9241*, que definen ciertos puntos claves que caracterizan este tipo de diseño, algunos de estos puntos son los enumerados a continuación:

1. El diseño debe estar basado fuertemente en la comprensión de los usuarios, el contexto y las tareas que estos realizan.
2. El diseño debe ser guiado y refinado por evaluaciones centradas en el usuario, ejemplos de esto son las técnicas de prototipado rápido que se detallarán más adelante.
3. El proceso es iterativo en el sentido de que se deben pulir partes del sistema en base a los resultados de las evaluaciones y re-evaluar para verificar el impacto de los cambios.
4. El equipo de diseño debe ser principalmente multidisciplinario, incluyendo especialistas en psicología, ergonomía, desarrollo de *software*, entre otros.

La *ISO 9241-210* define también la experiencia de usuario, del inglés *UX* o *user experience*, como las percepciones y respuestas de una persona como resultado del uso de un producto, sistema o servicio. En otras palabras, la experiencia de usuario es la sensación del usuario al utilizar un producto; la satisfacción que siente al usarlo, tocarlo o agarrarlo, pudiendo ser este un sistema informático o cualquier producto tangible como por ejemplo un periódico, una almohada, u otro.

Disponer de un equipo de desarrollo con integrantes de disciplinas tan diversas no implica que todos estén presentes en todas las tareas y fases del proyecto, ni tampoco incluso que tengan una visión general de la evolución del proyecto, sino que es suficiente con que cada integrante tenga la visión necesaria del sistema en lo que concierne a su participación y la disciplina subyacente. En la *Figura 109* se puede visualizar las distintas etapas en las que debe participar cada uno de los roles del equipo multidisciplinario que llevará a cabo el desarrollo del sistema de principio a fin, considerando el ciclo de desarrollo usual conformado por las etapas de análisis de requerimientos, diseño, implementación y lanzamiento, más las etapas adicionales de prototipado y evaluación, agregadas en pos de mejorar la usabilidad de la solución.

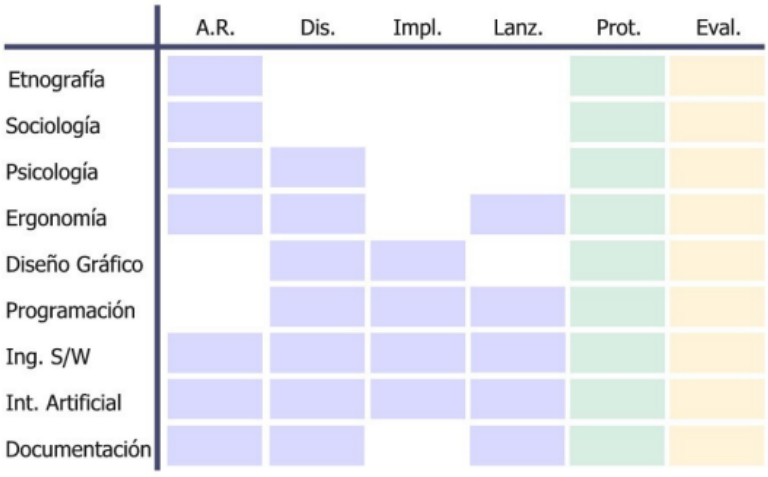


Figura 109: Participación de cada disciplina por cada etapa del proceso [79]

Anexo C: Prototipado

Una de las etapas más importantes del proceso centrado en el usuario es la etapa de prototipado, denominado "Prototipado Rápido", del inglés "*Rapid Prototyping"* debido a las características que posee. El Prototipado Rápido refiere a un conjunto de técnicas utilizadas en el proceso de diseño de interacción para la exploración de la aplicación de forma temprana mediante la producción de prototipos de diversa naturaleza que permiten establecer un primer acercamiento a cómo será el flujo de interacción. Mediante esta técnica se intenta encontrar de forma temprana eventuales problemas de usabilidad en la aplicación explorada.

"Prototipado", es una palabra que no se encuentra en el diccionario de la *RAE*, pero es de uso común en el contexto de la Interacción Persona-Computadora, utilizándose como traducción del anglicismo "*prototyping"* (que de hecho a pesar de ser muy utilizada, tampoco se encuentra en los diccionarios de la lengua inglesa) el cual refiere a la actividad de utilizar herramientas técnicas que permitan generar prototipos. Por prototipo se entiende a un documento, diseño o parte del sistema que simula de cierta forma el sistema final y permiten verificar funcionalidades, validar la navegación general del sistema y muchos otros aspectos relacionados con la interfaz definida. Más específicamente, en lo que respecta a una interfaz de usuario los prototipos se utilizan con la finalidad de explorar los aspectos interactivos del sistema desde etapas tempranas del proceso de desarrollo, siendo además un excelente medio de comunicación entre los integrantes del equipo de desarrollo y los usuarios finales del sistema.

Generalmente, durante el proceso de diseño de interacción, los prototipos utilizados varían en su fidelidad, siendo inicialmente prototipos de baja fidelidad y poco demandantes en cuanto a tiempo (prototipado de baja fidelidad), convergiendo más adelante en el proceso a prototipos de mayor nivel de fidelidad (prototipado de alta fidelidad). Por un lado, los prototipos de baja fidelidad se caracterizan por ser económicos, rápidos de construir y modificar. Éstos consisten en prototipos sencillos que implementan ciertas características generales del sistema sin entrar en detalles. Debido a estas razones, permiten que los usuarios no se cohiban y se sientan más cómodos proponiendo cambios, lo cual es la base del diseño centrado en el usuario. Por otra parte, los prototipos de alta fidelidad simulan un aspecto semejante al sistema final de forma completamente interactiva, como si del sistema final se tratase, abarcando aspectos más precisos y detallados, pudiendo incluso servir como una especificación o hasta como una herramienta de *marketing* que permita realizar demostraciones del producto. Por este motivo, se caracterizan por ser más caros en cuanto al tiempo de implementación y al personal especializado que los debe realizar.

Muchos autores aseguran que en las primeras etapas del diseño, los prototipos de baja fidelidad son tan efectivos como los de alta fidelidad, aunque en cada etapa se debe utilizar el prototipo adecuado. Algunas de las técnicas de prototipado rápido comúnmente utilizadas en las diferentes etapas son las que se se pueden observar en la *Figura 110*.

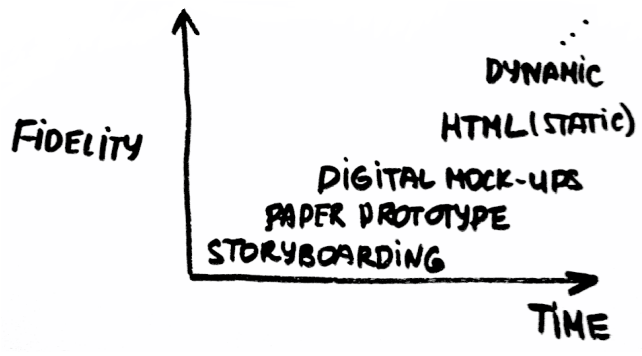


Figura 110: Gráfica conceptual de técnicas de Prototipado Rápido [117]

Una de las técnicas más básicas y de más alto nivel de abstracción dentro del prototipado rápido es

storyboarding, consistente en realizar diagramas o historietas del flujo de interacción centrados en las tareas, necesidades y objetivos que el usuario espera suplir con el sistema [138]. Así, un *storyboard* debe mostrar qué puede realizar el usuario a través de la interfaz sin atarse a un diseño específico, permitiendo estudiar la interacción desde un punto altamente abstracto. Mediante *Storyboarding* se puede plasmar las tareas y escenarios de interacción que se tendrá en cuenta en la aplicación desarrollada, pudiendo estudiar la interacción desde un punto altamente abstracto basado únicamente en las necesidades y objetivos del usuario. Otra de las técnicas de prototipado ampliamente utilizada es la conocida como prototipado en papel, del inglés *paper prototyping*. Ésta técnica se encuentra en un nivel de abstracción más bajo con respecto al *storyboarding*, ya que consiste en la construcción en papel de las pantallas, botones y demás elementos que componen el sistema a probar, a modo de poder mostrar el flujo de interacción de forma rápida sin escribir código [104, 106]. Mediante *Paper Prototyping* se logra obtener rápidamente una interfaz tangible que se podrá utilizar para probar la interacción con el objetivo de encontrar posibles problemas de usabilidad y elementos/funcionalidades faltantes o mejoras a las existentes. Este tipo de técnicas se complementa con otras técnicas como la denominada Mago de Oz, del inglés *wizard-of-Oz,* que permite simular el correcto comportamiento interactivo de la interfaz, y el pensamiento en voz alta, del inglés *think aloud,* que permite obtener retroalimentación adicional de los usuarios mientras utilizan la interfaz, brindando una forma sencilla, rápida y barata de estudiar a los usuarios interactuando con algo similar a la interfaz con la que contará el sistema. Su uso permite detectar múltiples oportunidades de mejora en lo que respecta a la interacción sin necesidad de escribir ni siquiera una línea de código [158]

Cabe mencionar que estas dos técnicas, que se describen con más detalle en las siguientes secciones de este anexo, fueron utilizadas en los inicios de este proyecto para conceptualizar muchas de las características del sistema a desarrollar. En etapas siguientes y ya teniendo un diseño de interacción más pulido gracias al *feedback* obtenido en base a los prototipos anteriores, se suelen construir *Digital Mock-Ups* haciendo uso de cierto *software* específico para este fin. Los *Digital Mock-Ups* surgen ante la necesidad de los usuarios de verse más involucrados con la interfaz a nivel de píxeles y no en papel, razón por la que insumen mucho más tiempo que los anteriores. Finalmente, se construyen los primeros prototipos por código, siendo inicialmente precarios en su funcionalidad pero agregando dinamismo de forma incremental a medida que se avanza en el proceso de diseño y desarrollo.

C.1. *Storyboarding*

Esta técnica tiene como objetivo principal mitigar el riesgo de concentrarse únicamente en la interfaz de usuario sin haberse previamente concentrado en las tareas, objetivos y necesidades de los usuarios que utilizarán la aplicación en cuestión. Consiste en realizar diagramas del flujo de interacción centrándose esencialmente en los aspectos referente a las necesidades y objetivos que la aplicación suplirá y de esta forma, en pocos paneles plasmar en qué aspectos la aplicación ayudará a los usuarios, sin centrarse en los elementos de interfaz que luego darán soporte a la interacción.

En términos generales, un *storyboard* es una especie de *comic* o historieta que ilustra cómo una o varias personas utilizan la aplicación para satisfacer sus necesidades en escenarios reales de uso. Un buen *storyboard* debe mostrar de forma clara quién es el usuario, cuál es la situación de uso y la motivación del usuario para utilizar la aplicación. Además, debe mostrar qué puede realizar el usuario a través de la interfaz, sin atarse a un diseño específico. Esto lo hace una herramienta muy potente para entender la interacción a un nivel alto de abstracción, enfocándose solo en las necesidades y objetivos de los usuarios. El objetivo de un prototipo de papel no es probar o verificar lo bonito que es el diseño, sino que se trata de verificar si los usuarios son capaces de realizar sus tareas con la interfaz propuesta. Algunos ejemplos de *storyboards* son los que se pueden observar en la *Figura 111.*

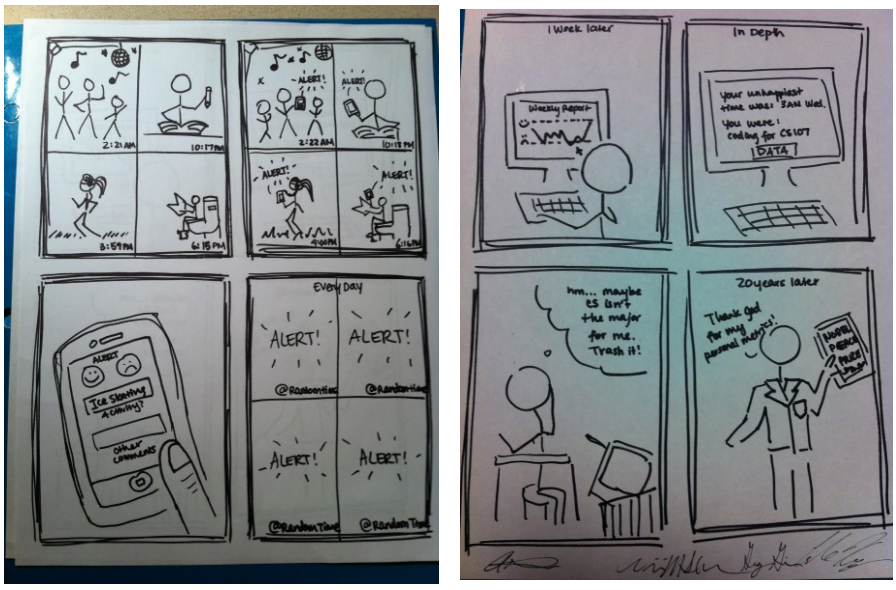


Figura 111: Diferentes ejemplos de *storyboards* [43]

Cabe mencionar que el *storyboarding* no requiere dibujar bien, sino simplemente saber cómo comunicar las ideas mediante un simple dibujo. Incluso, el dibujar "mal y rápido" puede ser beneficioso justamente para concentrarse en el contenido y no en los detalles de las ilustraciones. En definitiva, cuanto menos sofisticado sea el dibujo mejor aplicada estará la técnica y más rápido se obtendrá *feedback*, siempre y cuando permitan comunicar las ideas que se desean transmitir. Se enumera a continuación, tres características básicas que debe incluir un *storyboard*:

1. Presentar quiénes son los involucrados, esto es, quiénes son los usuarios, así como también el entorno en el que se encuentran inmersos y la tarea que pretenden realizar.
2. Desarrollar cómo se debe realizar la tarea, cuáles son los pasos necesarios y qué lleva a un usuario a utilizar la aplicación, mostrando cuál es el rol que juega la interfaz en ayudarlo a lograr sus objetivos.
3. Mostrar la satisfacción del usuario por haber cumplido el objetivo pretendido.

En la siguiente *Figura 112* se puede ver un *storyboard* inicial para la solución desarrollada como parte de este documento, donde se ilustra la funcionalidad correspondiente al seguimiento de las interacciones multitáctiles del usuario y la interacción con diferentes elementos que se muestran en la superficie de interacción. En este caso se detalla la compartición de imágenes y videos y la selección de componentes para edición y/o modificación.

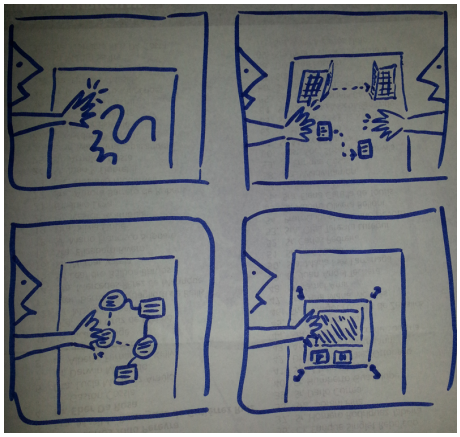


Figura 112: *Storyboard* de ejemplo para el presente trabajo

C.2. *Paper Prototyping*

Paper Prototyping es un método de prototipado rápido de bajo costo para el testeo de usabilidad de aplicaciones, que se basa en la construcción en papel de las pantallas, botones y demás elementos que componen la aplicación a *testear*. Ésta técnica permite esclarecer los requerimientos y el correcto flujo de interacción de forma rápida y sin escribir una sola línea de código. De esta forma, cualquier posible inconveniente puede ser detectado de forma temprana sin haber desarrollado nada concerniente al código de la aplicación y consecuentemente se ahorra dinero y tiempo en el desarrollo de posibles prototipos. En definitiva, consiste en hacer un *mockup* de la interfaz de usuario, pero en vez de implementarlo en una computadora se realiza utilizando papel y otros materiales. En este caso, al igual que con los *storyboards*, es importante obtener un prototipo de forma rápida.

Mediante el uso de esta técnica se puede representar cualquier tipo de interfaz, la cual puede ser dibujada o representada con fotografías, *screenshots* o dibujos sin la necesidad de ningún conocimiento extra. Para este tipo de prototipado no se necesitan más materiales que papel, tijera, lápices, lapiceras o marcadores e imaginación. De todas formas, se puede complementar con otros materiales de uso típico en *collages* como cinta adhesiva, cascola, *post-it notes*, u otros que se crean útiles para representar lo que se desee. Algunos ejemplos de *paper prototypes* son los que se pueden observar en la *Figura 113.*

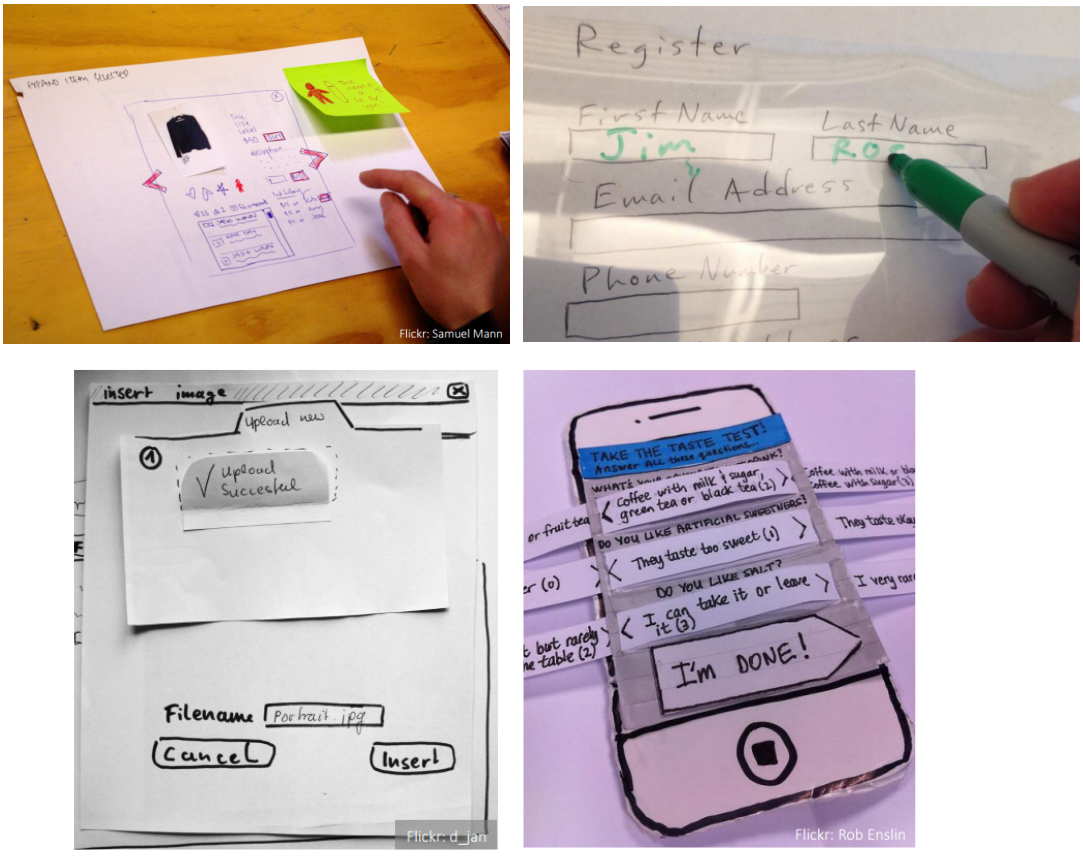


Figura 113: Diferentes ejemplos de *paper prototype* [111]

Se enumeran a continuación algunas de las ventajas que presenta la técnica de *Paper Prototyping*:

1. Brinda los mecanismos necesarios para generar prototipos de forma rápida sin la necesidad de generar código.
2. Permite lograr la localización de una gran cantidad de problemas de forma temprana entre los cuales se encuentran:

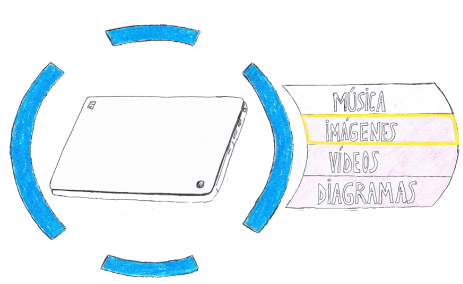
a. Conceptos confusos.
b. Interfaces incorrectas.
c. Navegabilidad incorrecta.
d. Carencia de información al usuario.
e. Requerimientos no contemplados.

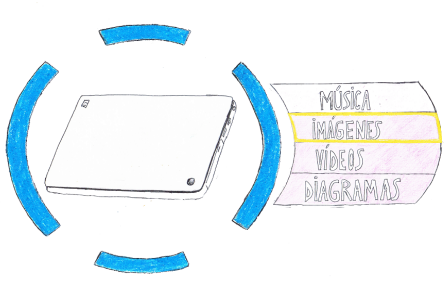
3. Permite redefinir la interfaz gracias a los aportes (*feedback*) del usuario sin necesidad de cambiar código.
4. Sumamente flexible y ágil para aplicar cambios.
5. Fomenta la creatividad en cuanto al diseño de la interfaz.
6. Independiente de la tecnología y por ende, de *bugs* para la realización de demos.

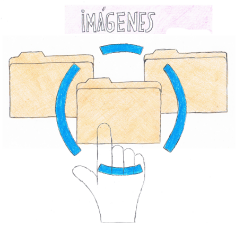
Sin embargo, ésta técnica posee algunas desventajas, las que se enumeran a continuación:

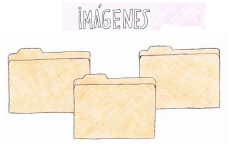
1. El desarrollo no produce código.
2. Como cualquier tipo de *testing*, no asegura la detección de todos los problemas que pueda contener una aplicación.
3. En algunos casos puede ser considerado poco profesional.
4. Puede ser complejo representar en papel algunos conceptos e interacciones como por ejemplo el *scrolling*.

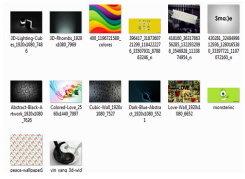
En la siguiente secuencia de imágenes se puede apreciar el *paper prototype* realizado en los inicios del presente proyecto como forma de bajar a tierra conceptualmente algunas de las funcionalidades a ser provistas por la solución. En este caso la aplicación en cuestión es referente a la consulta de recursos multimedia provenientes de uno de los dispositivos apoyados sobre la superficie de interacción, pudiendo seleccionar imágenes, videos y archivos de música a ser visualizados por los diferentes usuarios en la zona compartida. Si bien esta es una funcionalidad finalmente quedó fuera del alcance del proyecto, el prototipo fue útil en su momento para visualizar de mejor forma alguna de las características que la solución debería brindar inicialmente.











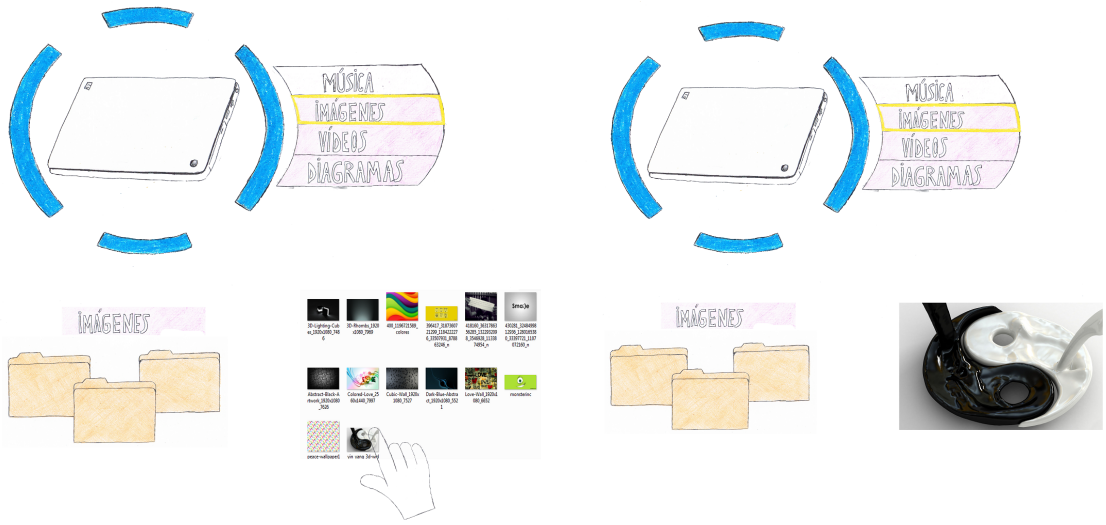


Figura 114: *Paper prototype* conceptual inicial del presente trabajo

Una vez que se obtiene el prototipo se desea poder utilizarlo para obtener *feedback* de los usuarios involucrados. Para ello, una de las técnicas más utilizadas es la denominada *Wizard-of-Oz Prototyping*, que hace honor a la película *Wizard of Oz (El Mago de Oz)*. Más específicamente, se centra en la escena donde *Dorothy* y sus amigos llegan a *Ciudad Esmeralda* y se encuentran con el *Mago de Oz*, quien aparenta ser un personaje siniestro que genera mucho miedo. Sin embargo, cuando se corren las cortinas resulta no ser más que un inofensivo anciano. De la misma forma, lo que intenta explotar la técnica *Wizard-of-Oz Prototyping* es el hecho de cómo un simple humano tiene la habilidad de montar una situación o escena tan realista e inmersiva que hace creer a los usuarios ser alguien mucho más poderoso de lo que en realidad es. La idea detrás de la técnica es entonces que un operador humano, denominado *wizard*, simule el comportamiento interactivo y las funcionalidades de la aplicación mediante la manipulación de los diferentes elementos que conforman la interfaz, haciendo que el usuario perciba estar ante un sistema inteligente con el que puede interactuar como lo haría con el sistema real. El *wizard* procesa la entrada del usuario y simula la salida del sistema permitiendo así simular cierta inteligencia artificial de forma rápida y poco costosa. Al momento de realizar un testeo mediante esta técnica se definen los siguientes roles:

1. *User*: persona que hace uso de la aplicación y por ello se convierte en el usuario del sistema.
2. *Facilitator*: persona que cumple el rol de facilitador, siendo la persona capacitada en los aspectos de usabilidad y quien está a cargo de la sesión.
3. *Human computer:* persona que hace de "Mago de Oz" detrás de escena, simulando ser la inteligencia que le da vida a la interfaz. En otras palabras, es la encargada de realizar las mímicas correspondientes al sistema en respuesta a las acciones del usuario.
4. *Observers*: personas que observan la interacción de los usuarios frente a la aplicación y toman nota de aquello que funciona de manera correcta y de aquello que resulta ser confuso. Generalmente son integrantes del equipo encargado de llevar a cabo el desarrollo de la aplicación.

Así, los usuarios son indicados para hacer *clicks* en botones, *links* u otros elementos que conforman la interfaz. Luego, la persona que encarne el rol de *human computer* responderá acordemente tal como lo haría el sistema y sin explicar al usuario el funcionamiento de la interfaz, lo cual permite verificar si el usuario es capaz de interpretarla por sí mismo. Durante todo la sesión de prueba el facilitador tratará de que la interacción no se torne estresante, evitando por ejemplo, que los usuario empiecen a culparse a sí mismos cuando se enfrentan a dificultades con el uso de la interfaz. En la *Figura 115* se muestran algunos ejemplos de sesiones basadas en las técnicas *Wizard-of-Oz* y *Paper prototypes*.

Figura 115: Sesiones de prueba con *Paper Prototyping* y *Wizard-of-Oz* [111]

Los prototipos basados en *Wizard-of-Oz* también se categorizan como de alta o baja fidelidad dependiendo de la fidelidad del prototipo que se tome como base. Si el prototipo es un prototipo muy sofisticado que junto con la inteligencia simulada hace que el usuario perciba al sistema tal cual fuera el sistema real y ni siquiera se de cuenta que está ante un *wizard,* será de alta fidelidad, mientras que si se utiliza un prototipo más sencillo, como por ejemplo uno basado en *paper prototyping*, entonces será de baja fidelidad. Algunos estudios indican que la segunda opción es una mejor alternativa, ya que, aparte de ser una opción más rápida, brinda más *feedback* y críticas debido a que los usuarios suelen tener más confianza para criticar una interfaz de baja fidelidad con respecto a una de alta fidelidad.

Otro aspecto importante que se debe tener en cuenta en este tipo de pruebas es el hecho de hacer entender a los usuarios que no se los está poniendo bajo un microscopio, recordándoles tantas veces como sea necesario que lo que se está poniendo a prueba es la aplicación, la interfaz de usuario o el sistema, pero no a ellos. Si el usuario es incapaz de realizar cierta tarea, la falla es del sistema y no del usuario, ya que los usuarios simplemente están ayudando a *testear* el sistema y en caso de bloquearse en algún punto, deben tener claro que no es debido a su falta de habilidad, sino que, por el contrario, habrán hecho el trabajo correctamente ayudando a encontrar fallas en la interfaz. Para capturar el *feedback* de los usuarios es recomendable tomar anotaciones sobre los incidentes críticos que se van presentando y se deberán solucionar, así como también, aspectos que funcionaron correctamente o cualquier otro tipo de anotación relevante que pueda ser útil para mejorar la aplicación.

Otro método que se utiliza comúnmente en conjunto es el método de *think aloud*, el cual suele ser una forma muy potente de obtener *feedback*. Como su nombre indica, el método consiste básicamente en saber lo que los usuarios están pensando en todo momento y no solamente lo que están haciendo, indicandoles que piensen en voz alta. Esto no es algo que los usuarios estén acostumbrados a hacer naturalmente, por lo que es necesario que se los solicite antes de comenzar la sesión o eventualmente que se les recuerde mediante preguntas durante la sesión de prueba. Preguntas como "¿qué estás pensando?", "¿qué estás intentando hacer?" o "¿qué estás leyendo?" pueden ser muy útiles. De todas formas, esta técnica puede alterar la veracidad de los resultados, ya que como los usuarios deben hablar mientras interactúan con el sistema, en algunos casos la forma en que realizan las tareas puede verse afectada.

Finalmente, es preciso mencionar que está comprobado que tiene un mayor valor y se obtienen mejores resultados producir y compartir varias alternativas de prototipo de forma rápida en lugar de una única alternativa que requiera más tiempo de construcción y sea de mejor calidad. Esto implica que lo ideal sería tener varios posibles prototipos a evaluar, con cambios a nivel de interfaz y/o interacción y luego en base al *feedback* obtenido para cada uno lograr un prototipo final. A esta técnica se le denomina *Parallel Prototyping*, contrariamente a la que utiliza un único prototipo, denominada *Serial Prototypin*g. En la *Figura 116* se presenta un diagrama conceptual de ambas técnicas. A su vez, algo que también suele suceder es el problema comúnmente denominado *functional fixation* que refiere a que inicialmente se elige un prototipo inicial y luego es difícil apartarse de la línea propuesta en dicho prototipo para crear algo diferente. Por ello, hay que buscar formas de evitar estos casos mediante la generación de varios prototipos alternativos.

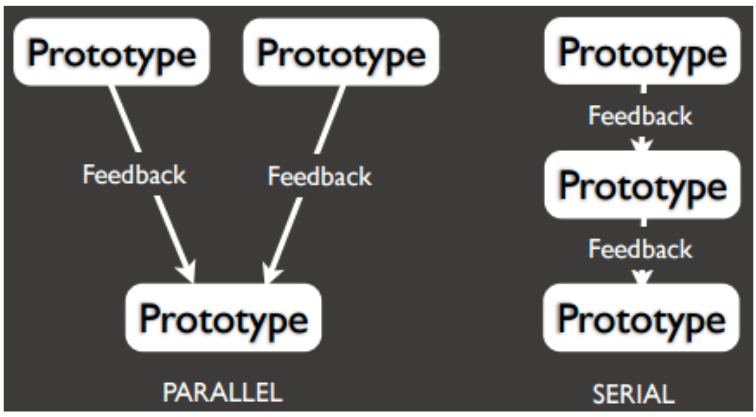


Figura 116: ***Parallel vs. Serial Prototyping*** **[113]**

Si bien no se terminaron aplicando las técnicas de *Wizard-of-Oz*, *Think aloud* y *Parallel prototyping* durante el desarrollo del proyecto, las mismas fueron investigadas, razón por la cual se consideró oportuno incluir parte de dicha investigación en el presente anexo dada la relevancia de este tipo de técnicas en lo que respecta al diseño de interacción y el desarrollo centrado en el usuario.

Anexo D: La ciudad ubicua

La computación ubicua ha avanzado considerablemente en los últimos años con el advenimiento de nuevas tecnologías de comunicación, la proliferación de lo que se denomina *Internet of Things (IoT)*, permitiendo contar con múltiples dispositivos conectados en red, y los avances en el diseño y construcción de *software*. Tal es el grado de avance, que actualmente existe una ciudad ubicua en construcción, de la cual se puede observar una imagen en la *Figura 138*, también denominada *U-city* ubicada en *New Songdo City* [153], una isla ubicada a 60 kilómetros del oeste de Seúl, Corea del Sur. Con un presupuesto de casi 40 billones de dólares, una vez finalizada, la ciudad contará con cerca de 7.000 kilómetros cuadrados y tendrá una capacidad para 65.000 personas, disponiendo de ordenadores integrados que permiten interconectar los grandes sistemas de información residenciales, de uso médico, gubernamentales, entre otros, para formar una red *WAN* metropolitana de información que permita integrar completamente la vida en el hogar con la vida en las calles. En términos generales, una ciudad ubicua implica integrar de la mejor forma posible la computación ubicua en el entorno urbano, haciendo que todos los dispositivos y servicios estén conectados mediante redes inalámbricas y diferentes tipos de sensores. Algunos de los servicios que se pretenden incorporar en esta ciudad son los enumerados a continuación:

1. Estaciones de reciclaje inteligentes en las calles capaces de reconocer a quien recicla la basura y realizar un pago automático a modo compensatorio.
2. Casas con pisos inteligentes sensibles a la presión que permitan informar automáticamente si, por ejemplo, una persona anciana tuviera una caída.
3. Celulares con información médica que pueden ser utilizados directamente para pagar por medicamentos.
4. Una red de tubos que evitan el uso de camiones de basura, succionando directamente la basura y transportandola eficientemente a las unidades de tratamiento.
5. Tuberías capaces de distinguir entre agua potable y no potable, enviando solo esta última a aparatos como duchas e inodoros evitando desperdiciar agua potable.

Figura 117: *U-city* [46]

Según *John Kim*, vicepresidente estratégico del proyecto de desarrollo de la ciudad, cada habitante contará con una tarjeta inteligente que le permitirá no solo acceder a su hogar, sino también al transporte público, a lugares específicos para entretenimiento como cines y teatros, realizar videoconferencias entre vecinos y acceder a vídeo bajo demanda y otro contenido digital desde cualquier parte de la ciudad [58]. A su vez, esta tarjeta será anónima y no estará asociada a la identidad de la persona.

Otra de las características notorias de la ciudad es que utiliza mayormente tecnologías verdes o *green technology*, causando el menor impacto posible en el medio ambiente. De esta forma, la emisión de dióxido de carbono generado por parte del uso de energía de la ciudad así como también, por los medios de transporte es varios órdenes menor que en una ciudad normal, gracias al uso de transporte impulsado por células de hidrógeno, entre muchas otras *green technologies* certificadas por el conocido sistema de certificación *LEED,* del inglés *Leadership in Energy & Environmental Design*, lo que implica que la ciudad esté adherida a los más altos estándares ambientales de consumo de energía y la convierte en la única ciudad no estadounidense en contar con ello.

Anexo E: Historia de los dispositivos de interacción

Tanto la disciplina HCI que pretende estudiar la interacción entre el hombre y la computadora como los dispositivos de interacción en sí mismos, han ido evolucionando notablemente a lo largo de los años, comenzando en la década de los 60' y 70' cuando aparecen los denominados *mainframes,* macro computadoras pensadas especialmente para el uso de los profesionales del procesamiento de datos. Pero, en aquella época, los usuarios promedio no estaban satisfechos con los costes, los retardos y la poca flexibilidad que estos grandes equipos brindaban. Avanzados los años 70' aparecen las minicomputadoras para profesionales informáticos y otros profesionales no informáticos, pero aún así se debe realizar mucha programación para utilizar los computadores. Es hacia los años 80' cuando los computadores se popularizan y se dirigen hacia el público general, siendo la usabilidad el principal requisito. La computación personal se desarrolló con el modelo de ventanas, iconos, menús y punteros (*WIMP* del inglés, *Windows, Icons, Menus and Pointers*) como paradigma dominante para la interacción con las aplicaciones de escritorio. Comienzan entonces a surgir los primeros sistemas comerciales que permiten la manipulación de objetos gráficos, algunos de los cuales fueron el *Xerox Star* (1981), *Apple Lisa* (1982), la *Apple Macintosh* (1984) y finalmente el *Microsoft Windows* 1.0 (1985). Durante la década siguiente aparecen nuevos dispositivos de pequeño tamaño y es entonces cuando ocurre la explosión de la informática móvil, dirigida también al público en general, por lo que la usabilidad sigue siendo la principal exigencia.

Las posibilidades tecnológicas de las últimas tres décadas han permitido llevar la computación a casi todos los artefactos de la vida cotidiana. De este modo, a principios del año 1991 surge el concepto de computación ubicua, la cual refiere al hecho de que los ordenadores actuales ya no están vinculados únicamente al escritorio, sino que están integrados en todos los aparatos y aspectos de las vidas de las personas, quienes se relacionan con ellos de forma transparente y natural. Estos nuevos ordenadores van desde grandes paneles en una pared para trabajo colaborativo hasta pequeños dispositivos para tomar notas, pasando por sensores de presencia que activan sistemas de iluminación y seguridad, entre muchos otros. Adicionalmente a la computación ubicua, surge la interacción gestual, que se plantea como la creación, manipulación y distribución de significado mediante una interacción activa con los artefactos. Esta resulta un marco conceptual ineludible a la hora de afrontar el desarrollo de sistemas con interacción basada en gestos sin reducir el enfoque al trasladar el paradigma de ventanas, punteros y ratones fuera de la pantalla. En esta línea, el proceso evolutivo se destaca enfocándose en establecer parámetros que no le impliquen un esfuerzo cognitivo sustancial al usuario.

La disciplina HCI comienza a ver sus frutos en el momento en que los ordenadores dejan de ser un misterio para los usuarios no profesionales y salen a un mercado más amplio. La interacción con el ordenador va pasando desde la línea de comandos donde se le daban las órdenes en un lenguaje puramente informático a otras formas más amigables como los menús de opciones o la manipulación directa, en la que el usuario tiene la posibilidad de manejar los diferentes elementos de forma similar a la que acostumbra en la vida real. En la evolución de la disciplina han influido diversos factores como los que se enumeran a continuación:

1. La creatividad humana: especialmente en los inicios de la ciencia informática diversos visionarios realizaron proyecciones imaginarias sobre lo que podrían llegar a ser las computadoras.
2. El estado del arte de la tecnología: actuando como limitante al diseño de las nuevas computadoras.
3. El mercado de los computadoras: directamente relacionado con el costo de las computadoras que incide directamente tanto en el tipo de usuario como en el uso que hacen sobre ellas.

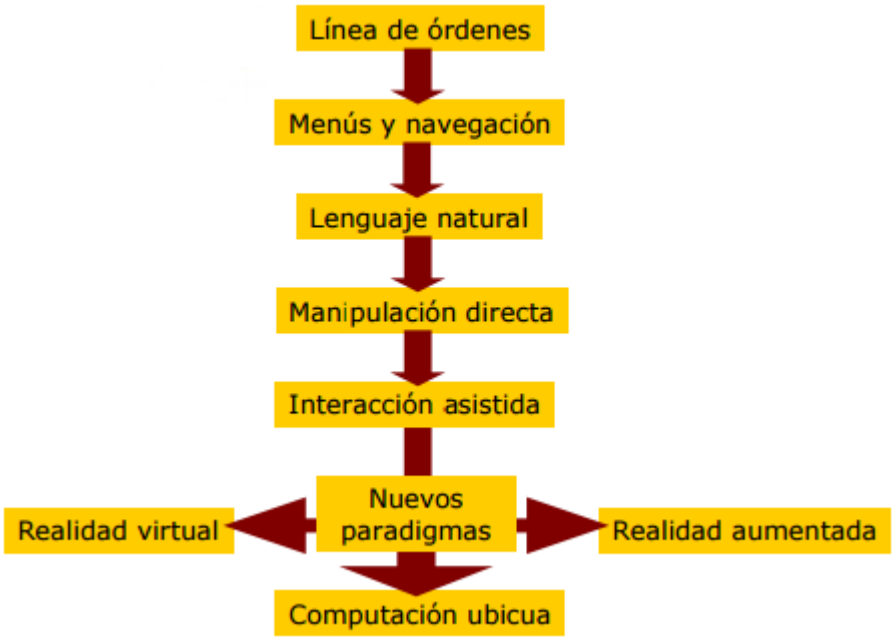
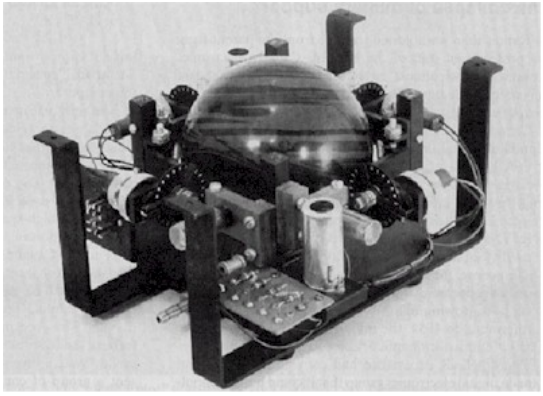


Figura 118: Evolución de los paradigmas de la HCI [47]

La historia del diseño de la interfaz y la interacción es un camino que va desde la complejidad a la simplicidad. Si bien se partió de máquinas diseñadas para fines científicos que sólo podían ser utilizadas por un grupo reducido de usuarios técnicos, hoy en día existen dispositivos ubicuos que pretenden brindar mecanismos de interacción similares a los presentes en la vida cotidiana de las personas, logrando de esta forma que la interacción sea más sencilla y abarque un mayor espectro de usuarios.

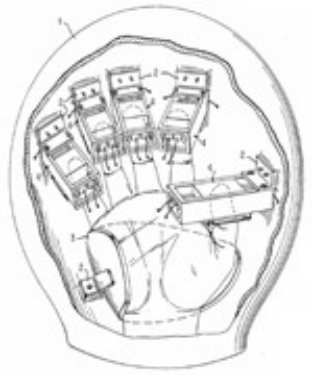
Como se mencionó anteriormente, hasta hace algunos años prácticamente el único mecanismo de interacción con la interfaz era mediante el *mouse* y el teclado, principalmente debido a que la gran mayoría de los sistemas eran implementados siguiendo el paradigma *WIMP*. Sin embargo, en las últimas dos décadas esto ha venido cambiando y se han introducido nuevos dispositivos y formas de interactuar con la interfaz [54], adaptando el *mouse* a los diferentes dispositivos que han surgido, como laptops y dispositivos móviles, surgiendo así nuevas formas de interacción mediante *trackballs*, *trackpads*, *touchpads* y la interacción directa con pantallas táctiles inteligentes, las que permiten además, detectar gestos realizados con los dedos. Las imágenes de la *Figura 140* resumen la evolución de los dispositivos de interacción a lo largo de la historia [148]. Cabe destacar que lo presentado no es un detalle continuo, ya que se presentan solamente aquellos dispositivos más interesantes ya sea por haber marcado un hito importante en la evolución o por ser curioso debido a su modo de uso.



Primer *trackball* utilizado (*Tom Cranston*, 1952). Utilizaba una bola de bolos soportada por pequeños propulsores de

Primer *lightpen* (*Ben Gurley*, 1957), utilizado en grandes computadores *mainframe*, permitiendo

aire para minimizar la fricción y mantenerla "flotando".

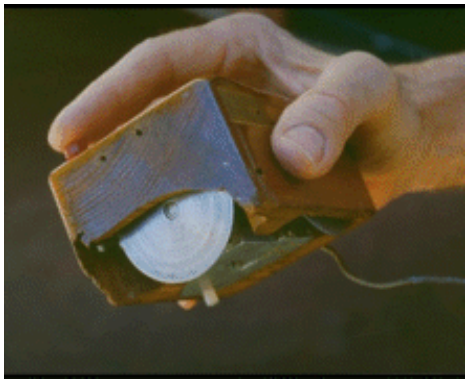


Teclado para entrada de texto en forma de guante de boxeo (*Robert Seibel*, 1960), a diferencia de los guantes inteligentes livianos que conocemos hoy en día.

Primer pantalla táctil (*E.A. Johnson*, 1965), basaba en las características capacitivas de la pantalla al igual que la gran mayoría de las pantallas táctiles de hoy en día.

Primer pantalla táctil utilizada masivamente, denominada *PLATO IV* (*University of Illinois*, 1972). Obtenía la posición del dedo al interrumpir los rayos de luz que se emitían paralelos a la pantalla.

marcar *pixeles* en su pantalla.

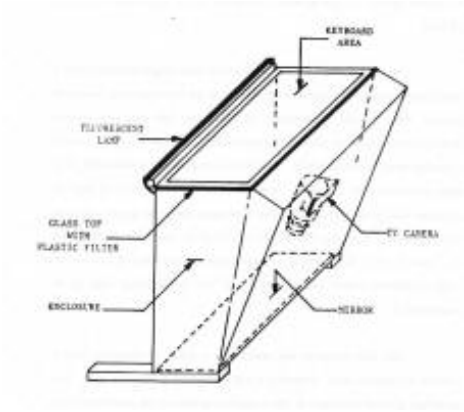


Primer *mouse* (*Doug Engelbart* y *William English*, 1964 aunque la patente no fue aplicada hasta 1967). Construído con una caja de madera, contaba con un único botón y una rueda que permitía moverse en dos ejes**.**

Primer *ball mouse* (*Rainer Mallebrein* de *Telefunken*, 1968) incluido en la primer computadora comercializada, la *Telefunken TR86*.

Primer *mouse* óptico (*Steven Kirsch*, 1981), que evitaba el problema de acumulación de polvo de los *mouse* mecánicos. Sin embargo, requería el uso de un *pad* especial con ciertos patrones utilizados ópticamente para realizar el seguimiento del dispositivo.

Guante óptico (*Zimmerman*, 1982),que surgió gracias a la introducción de un sensor óptico flexible, que fue luego incorporado a los dedos de un guante. Permitía medir el grado de curvatura del *joint* que cubría.



Primer superficie multitáctil (*University of Toronto*, 1982). La superficie era un filtro translúcido de plástico que cubría una capa de vidrio, iluminado por una lámpara fluorescente. Una video cámara colocada debajo de la superficie capturaba las sombras que aparecían en el filtro.

Primer *PC* comercializada con una pantalla táctil (*Hewlitt Packard*, 1983). Utilizaba la misma tecnología que la *PLATO IV* descrita anteriormente (posición del dedo al interrumpir rayos de luz).

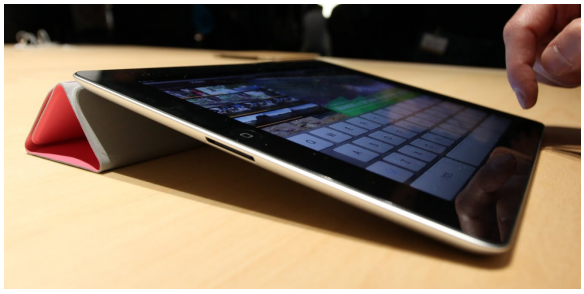
Primer computadora en contar con *mouse* y teclado inalámbricos (*Metaphor Computer*, 1984). Utilizaba señales infrarrojas para transmitir datos desde los periféricos a la computadora.

Trackpoint (*Ted Selker de IBM*, 1987). Como consecuencia de algunos estudios que afirmaban que un tipeador demoraba casi un segundo en llevar su mano desde el teclado hasta el *mouse*, *Selker* construyó

Primer *mouse* con *scroll-wheel (Apple Computer,* 1989). Contaba con una rueda en el lateral que se giraba con el dedo pulgar para desplazarse. De todas formas no se comercializó; el primer

un prototipo del primer *trackpoint*. Luego trabajando para *IBM* fue llevado a las computadoras personales.

Intellimouse (*Microsoft Corporation*, 1999), primer *mouse* óptico que no requería un *mouse pad* con patrones especiales para su funcionamiento.



Pantalla táctil capacitiva, *iPad* (*Apple Inc*., 2010). Es un teléfono inteligente y además un computador portátil que cuenta con una pantalla multitáctil de casi 10''. Además tiene la característica de ser muy fino y liviano, lo que lo hace un dispositivo de uso cotidiano que el usuario puede llevar consigo prácticamente todo el tiempo.

Microsoft Kinect (Microsoft Corporation, 2010), incluye sensores que permiten reconocer los movimientos del cuerpo del usuario, haciendo que la interacción sea completamente natural.

mouse scroll-wheel comercializado masivamente fue el *Genius EasyScroll* recién en 1995.

Magic trackpad (*Apple Inc*., 2010). Es un *trackpad* multitáctil similar al que se encuentra hoy en día en una *Macbook*, aunque un 80% mayor en tamaño. Cumple la función de un *trackpad* convencional y además cuenta con una amplia gama de gestos que permiten controlar más intuitivamente las aplicaciones.

*Microsoft PixelSens*e, primer superficie multitáctil de *pixeles* bidireccional (*Microsoft Corp*. y *Samsung*, 2011). Lanzada inicialmente como *Microsoft Surface V2* y también conocida como *Samsung SUR40*, es una superficie de 40'' en la que los *pixeles* no solo son utilizados para emitir luz sino también para capturarla. De esta forma, funciona como una pantalla *LCD* y además como un capturador de imágenes, dado que puede "visualizar" los dedos u cualquier otro elemento que haga contacto con la superficie.

Leap Motion Controller (*Leap Motion Inc*., 2012) permite realizar un seguimiento preciso de las manos y dedos de los usuarios así como también,

reconocer los gestos realizados.

MYO (*Thalmic Labs*, 2013), es un dispositivo en forma de banda que se coloca en el brazo del usuario y permite reconocer gestos mediante el análisis de las señales eléctricas que generan sus músculos.

Figura 119: Evolución histórica de los diferentes dispositivos de interacción [142, 49, 87]

Anexo F: Principios para el buen diseño con sensores de interacción

Adicionalmente a la teoría general y a las buenas prácticas en lo que refiere a la interacción natural, y en particular, a los diferentes tipos de interacción natural detallados en el documento principal del presente trabajo y su anexo, es también oportuno incluir la investigación sobre los principios de diseño particulares para los sensores utilizados, los que como se ha mencionado anteriormente, son los sensores *Microsoft Kinect* y *Leap Motion*. Los puntos más importantes a tener en cuenta, particularmente en el diseño de interacción basado en movimiento, son los que refieren a la ergonomía, la postura y al entorno en el que los usuarios utilizarán el sistema. Generalmente, no es cómodo para las personas permanecer con las manos extendidas por largos períodos de tiempo, así como tampoco dejarlas completamente quietas o realizar figuras geométricas con gran precisión. Se debe entonces, diseñar siempre teniendo en cuenta el *confort* del usuario evaluando la aplicación desde el punto de vista del cansancio que ésta produce.

A su vez, con el fin de mejorar la interacción y disminuir la cantidad de movimientos a realizar por parte de los usuarios es importante hacer una correcta correlación entre las coordenadas tridimensionales captadas por el sensor y las correspondientes al dispositivo bidimensional en donde se mostrará la información visual, la que es, por lo general, una pantalla o superficie de proyección. Si se hacen corresponder coordenadas de sensor grandes con coordenadas de pantalla pequeñas se obtendrá una buena precisión y estabilidad, pero se requerirá que el usuario realice movimientos más pronunciados para lograr observar algún efecto visual. Por otro lado, si se corresponden coordenadas de sensor pequeñas a coordenadas de pantalla más grandes se obtendrá más sensibilidad pero se decrementará la precisión en de la interacción. Lo ideal entonces, sería encontrar un punto medio que equilibre todos estos factores.

A modo de ejemplo, considerando los comentarios mencionados, la famosa interfaz utilizada en la película *Minority Report* no estaría bien diseñada para el mundo real, ya que la interacción involucra demasiado movimiento de las manos por sobre el nivel de los hombros, lo que generalmente ocasiona gran cansancio y tensión, incluso el propio *Tom Cruise* (actor principal de la película) debía descansar su brazos mientras realizaba la filmación. Dicho fenómeno se conoce bajo el nombre de "*gorilla-arm*" y surge debido a que los seres humanos no están diseñados para mantener los brazos levantados frente a su cabeza realizando pequeños movimientos durante largos períodos de tiempo, ya que generalmente luego de unas pocas interacciones el brazo comienza a sentirse molesto y hasta dolorido [139]. Este fenómeno fue la razón por la que las pantallas multitáctiles verticales como tecnología principal de entrada no tuvieran el éxito que se esperaba a pesar de que hayan sido muy prometedoras en la década de los 80'.

F.1. *Leap Motion*

En esta subsección se presentan algunas guías y buenas prácticas utilizadas para diseñar interacciones intuitivas y buenas experiencias de usuario utilizando el sensor *Leap Motion*. De todas formas, muchas de ellas son lo suficientemente genéricas como para ser aplicadas también para el diseño de interacción en general, independientemente del sensor a utilizar. A continuación se enumeran algunas recomendaciones en cuanto a cómo utilizar de mejor manera el sensor *Leap Motion* dependiendo de la posición en la que el usuario se encuentre al momento de interactuar con el sistema [17]:

1. Si el usuario se encuentra de pie, es recomendable que mantenga los codos a los costados, con los antebrazos paralelos al piso como se puede ver en la imagen de la izquierda de la *Figura 120*.
2. Por el contrario, si el usuario se encuentra sentado a nivel de un escritorio, es recomendable que apoye los codos o antebrazos en él, evitando apoyarlos sobre el borde y manteniendo las

manos en la misma línea que las muñecas en la medida de lo posible, tal como muestra las imágenes del centro y de la derecha de la *Figura 120*. En cualquiera de los casos, las manos deben estar justo por encima del dispositivo y los hombros deben estar relajados.

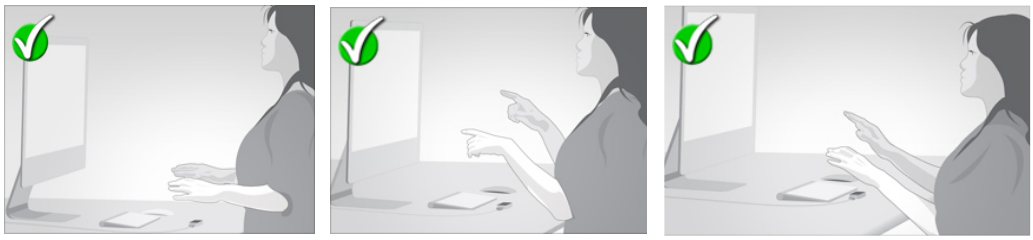


Figura 120: Patrones para el uso ergonómicamente correcto del sensor *Leap Motion* [17]

3. No es recomendable abalanzarse sobre el dispositivo como se muestra en la primer imagen de la *Figura 121.*
4. Tampoco es recomendable doblar las muñecas con los codos juntos apoyados sobre la superficie, como se muestra en la segunda imagen de la *Figura 121.*
5. A su vez, no es recomendable mantener los brazos estirados en el aire como muestra la tercer imagen de la *Figura 121*.
6. Finalmente, no es recomendable apoyar los brazos en la superficie de forma tal que estos ingresen inclinados al espacio de interacción como muestra la cuarta image de la *Figura 121*.

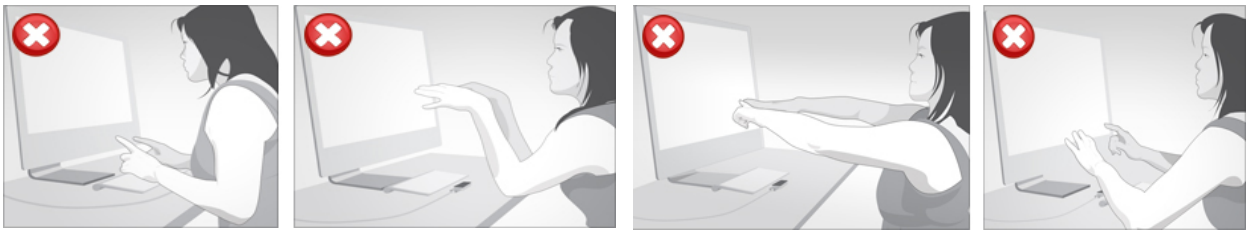


Figura 121: Antipatrones para el uso ergonómicamente correcto del sensor *Leap Motion* [17]

Existen un conjunto de heurísticas, que han resultado útiles para evaluar el diseño de interacción, algunas de éstas son las enumeradas a continuación [20]:

1. Consistencia del *tracking*: dado que algunos movimientos son reconocidos más consistentemente que otros, es bueno siempre asegurar que la confidencia del mecanismo de seguimiento es aceptable. En el caso del sensor *Leap Motion* esto es expuesto por el *SDK* como un atributo que indica cuán confiable es el seguimiento en cada momento dado.
2. Facilidad de detección: refiere a una medida de qué tan difícil es reconocer los movimientos y/o gestos, en términos de falsos positivos y falsos negativos. Gestos fáciles de reconocer tendrán tasas menores de falsos positivos y falsos negativos, haciendo que la aplicación sea más usable. Adicionalmente, cuanto antes se detecte la intención de un cierto movimiento o gesto, antes podrá la interfaz proveer la retroalimentación correspondiente, haciendo que el sistema se perciba más responsivo y que los usuarios sientan tener más control.
3. Transiciones: refiere a qué tan complejas son las transiciones entre los diferentes gestos que provee la interacción. En este aspecto, pueden surgir varios problemas, debido a por ejemplo, que dos acciones sean muy similares, haciéndolas difícil de recordar para los usuarios y consecuentemente disminuyendo la facilidad de aprendizaje del sistema.
4. *Feedback:* refiere a qué tan bien percibida está la retroalimentación generada en consecuencia de una interacción, sea ésta mediante audio, retroalimentación visual o cualquier otro tipo. Es importante contar con un elemento en la pantalla que brinde el usuario retroalimentación uno a uno con sus movimientos, permitiéndole conocer en todo momento con qué parte de la interfaz está interactuando. En el caso del sensor *Leap Motion,* esto puede implementarse mediante una representación completa de la mano en el espacio virtual o al menos un punto

de referencia, como por ejemplo el centro de la mano, de modo que permita a los usuarios mapear sus acciones con la aplicación.

5. Interacciones cortas: las personas se cansan de realizar una y otra vez la misma interacción, por lo que se deben utilizar interacciones cortas.
6. Descansos: se debe crear un modo de descanso para cuando el usuario quiera relajarse luego de una interacción larga, como puede ser esculpir un modelo tridimensional o recorrer un globo terráqueo con *Google Earth*. En dichos casos, el usuario puede, por ejemplo, cerrar la mano para indicar que desea tomar un descanso y luego retirarla sin alterar el estado y el control de la aplicación. Incluso una buena práctica es pausar las aplicaciones cuando no hay ninguna mano dentro del espacio de interacción.
7. Mecanismos ante error: tener en cuenta con mecanismos que permitan revertir situaciones de error es muy importante, tal como lo propone *Jakob Nielsen* en su lista de heurísticas. Cuando el usuario comete un error, no quiere que dicho error sea permanente, por lo que se debe brindar la opción de deshacer la acción permitiendo que se tenga más control sobre el sistema y que el usuario se sienta más seguro. En general, aplican también perfectamente todas las heurísticas genéricas de *Nielsen* aplicables para cualquier tipo de sistema interactivo [1].

F.2. *Microsoft Kinect*

Las interacciones e interfaces basadas en un sensor *Microsoft Kinect* pueden generar experiencias únicas para sus usuarios. La magia y el deleite de este sensor aparecen cuando el usuario sabe cómo utilizar la interfaz y se siente cómodo al hacerlo, por lo que el desarrollo de una interfaz de usuario natural es el comienzo de una revolución que da forma a la manera de experimentar e interactuar con las aplicaciones de *software*. Con este objetivo, *Microsoft* en su liberación 1.5 del *SDK* de *Kinect for Windows*, proporciona un documento con pautas y consejos a la hora de diseñar interacciones e interfaces para aplicaciones utilizando este sensor, respondiendo a las lecciones por ellos aprendidas de forma que los nuevos desarrolladores puedan evitar una serie de problemas y malas decisiones bien conocidas. En esta sección se presentan los aspectos más relevantes del documento mencionado denominado *Human Interface Guidelines* [56], prestando especial atención a cuestiones aplicables al presente trabajo. Se enumeran a continuación algunos principios básicos de aplicación general:

1. Mejores experiencia de usuario: ésta ocurre cuando el usuario es consciente del contexto, por tal motivo, los diseños también deben tener en cuenta el contexto. En este sentido, la interfaz debe adaptarse a la distancia entre los usuarios y el sensor *Microsoft Kinect* y debe responder también al número de usuarios y a la participación de ellos con el sistema.
2. Usuario felices: son aquellos seguros de sí mismos, por ello es importante mantener las interacciones simples y fáciles de aprender y dominar. Para aumentar la confianza del usuario es aconsejable utilizar combinaciones de entrada, tales como entrada por voz o gestos, procurando no malinterpretar la intención del usuario. Por otro lado, se debe dar una retroalimentación constante para que los usuarios siempre sepan qué es lo que está pasando y qué pueden esperar que suceda.
3. Ventajas y sus desventajas: cada método de entrada tiene sus ventajas y desventajas y los usuarios normalmente podrán elegir qué entrada utilizar, lo que se traduce por lo general en esfuerzo. Los usuarios tienden a adherirse a una sola entrada cuando no se ha dado una razón lo suficientemente importante como para utilizar otra. Es por esto que los diseños deben ser confiables, consistentes y convenientes, ya que de lo contrario, los usuarios buscarán opciones alternativas en caso de complicaciones. Si existen varios tipos de entrada, los cambios de una a la otra deben ocurrir de forma natural, o en los puntos naturales de transición en el escenario.
4. Pruebas de usuario: *Microsoft Kinect* permite una gran cantidad de nuevas interacciones, pero también nuevos desafíos. Es especialmente difícil adivinar lo que funcionará y lo que no antes de ponerlos en práctica, y muchas veces, pequeños ajustes a lo largo del proceso de diseño pueden hacer una gran diferencia. El hecho de realizar pruebas de usuario con frecuencia y de

forma temprana ayuda a conseguir un producto bien diseñado en cuanto a la interacción.

A continuación, se presenta una categorización de los tipos de gestos existentes y se enumera una serie de consejos a tener en cuenta según la guía propuesta por *Microsoft Kinect*:

1. Gestos estáticos: son aquellos en los que el usuario debe mantener una posición hasta que ésta sea reconocida. En este tipo de gestos, se debe prestar especial atención en cuáles son las posturas elegidas y a qué público apunta, debido a que, por ejemplo, un mismo gesto puede tener diferentes significados en distintas culturas, más allá de lo ya mencionado en cuanto a tiempo y comodidad en la ejecución del gesto.
2. Gestos dinámicos: son aquellos que brindan retroalimentación al usuario antes, durante o después de realizar el movimiento definido por el gesto en sí mismo.
3. Gestos continuos: son aquellos que realizan un seguimiento del usuario a medida que se mueven frente al dispositivo.

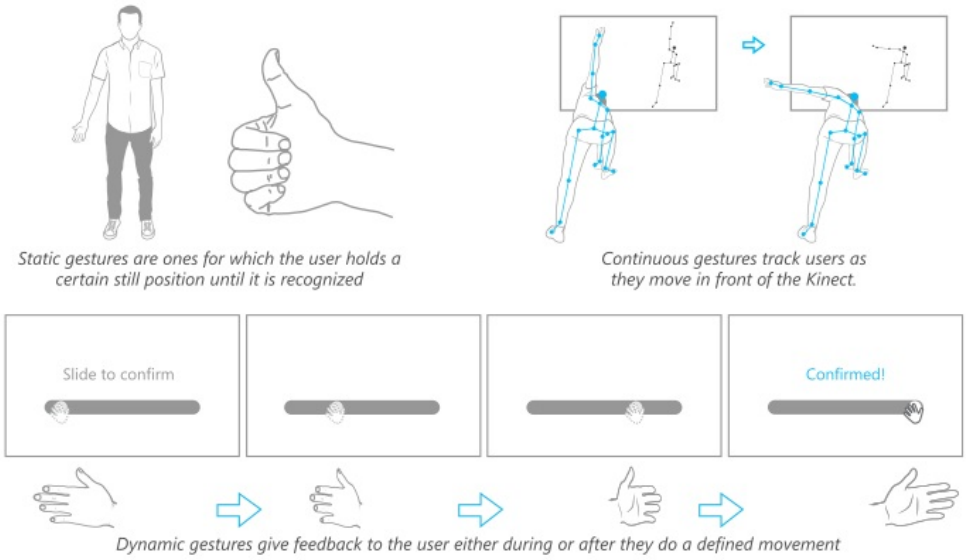


Figura 122: Categorización de gestos en estáticos, dinámicos y continuos [44]

Adicionalmente, la guía propuesta por *Microsoft Kinect* propone otra categorización a los gestos tal como se ilustra en la *Figura 123* y se enumera a continuación:

1. Gestos innatos: son aquellos que el usuario sabe realizar intuitivamente y tienen su base en cómo es la comprensión del mundo que percibe.
2. Gestos aprendidos: son aquellos que, con el fin de saber cómo usarlos para interactuar con el sistema, el usuario debe ser instruido y conllevan un aprendizaje para poder ejecutarlos.

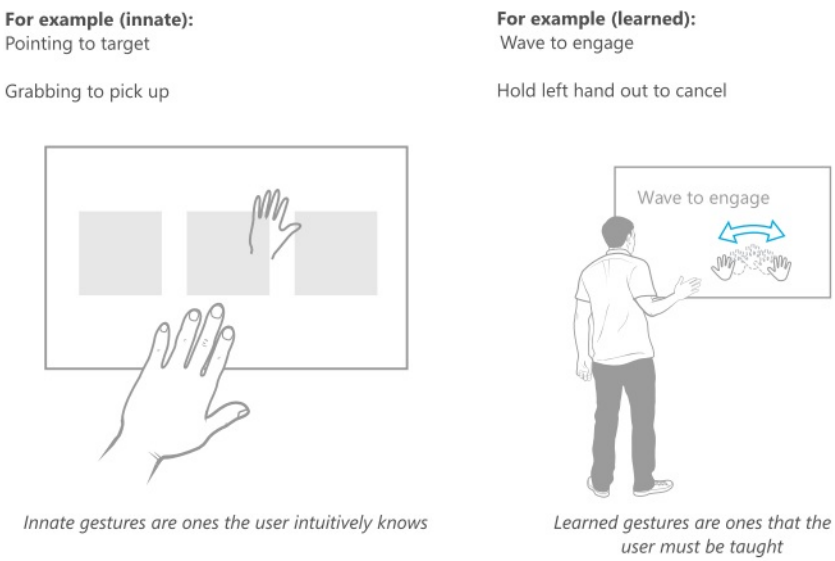


Figura 123: Categorización de gestos en innatos y aprendidos [44]

El diseño de las interacciones gestuales es un problema relativamente nuevo que *Microsoft Kinect* ha permitido experimentar y comenzar a entender. Se enumera a continuación algunas de las conclusiones y consideraciones claves que *Microsoft* ha detectado para hacer de sus diseños de interacción basados en gestos una componente de interacción lo más intuitiva y flexible posible:

1. Seguir los principios de los gestos: para que el sistema brinde una buena experiencia en cuanto a interacción gestual, los usuarios deben ser capaces de aprender rápidamente todos los controles básicos, poder realizar los gestos aprendidos rápidamente y con precisión, encontrarse ergonómicamente cómodo con los gestos y percibir que el sistema es sensible y proporciona retroalimentación continua al realizar un gesto.

2. Diseño adecuado para la mentalidad del usuario: las personas suelen asociar a menudo a *Microsoft Kinect* con videojuegos, pero cuando se está diseñando para aplicaciones de propósito general hay que recordar que la mentalidad del juego no es la misma que la de la interfaz de usuario. Si un usuario con mentalidad de interfaz de usuario no puede realizar un gesto se sentirá frustrado y tendrá una baja tolerancia para cualquier tipo de aprendizaje. A su vez, bajo la mentalidad de interfaz de usuario un gesto calificado como torpe puede llegar a ser considerado como no profesional, mientras que bajo la mentalidad de juego es una acción considerada generalmente divertida.

3. Diseñar interacciones naturales: no es una buena práctica forzar la forma de entrada gestual en base a otra forma de entrada de una interfaz de usuario ya existente, como por ejemplo, suponiendo una interfaz táctil, simplemente asignar gestos a la aplicación para cada uno de los tipos de entrada táctil. Los gestos pueden proporcionar un método novedoso de interactuar con la aplicación, pero se debe tener en cuenta que su uso debe tener algún propósito definido. Como muestra la *Figura 124*, para diseñar interacciones con gestos y evaluar si es una buena opción a la interacción o no, se deben tener en cuenta algunos puntos básicos, como por ejemplo, si la situación requiere que los usuarios interactúen a cierta distancia, si los gestos permiten una interacción que otros sensores de entrada no pueden lograr o si es bueno centrar parte o incluso la totalidad de la interacción en los gestos innatos, ya que son más naturales para el usuario y se suelen sentir más intuitivos.

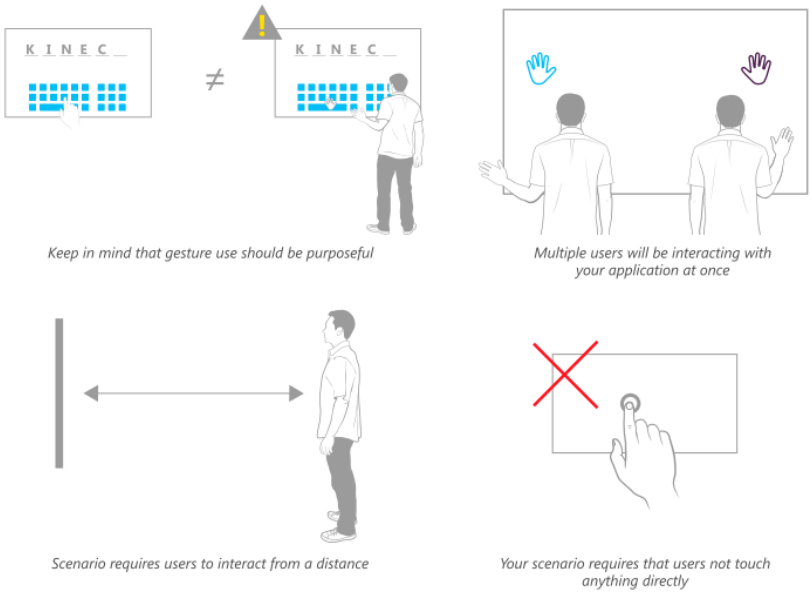


Figura 124: Buenas y malas prácticas de interacción [44]

4. Determinar la intención y participación del usuario: la determinación de la intención del usuario es el desafío más importante, y es un problema muy difícil de resolver, ya que, a diferencia de otros dispositivos de entrada el usuario está siempre activo. Parte de la complejidad de determinar la intención del usuario es tratar de reconocer cuándo los usuarios desean interactuar con el sistema. Para algunos escenarios puede ser deseable que el sistema esté siempre escuchando y observando, permitiendo a los usuarios indicar de forma explícita cuando quieren comenzar a interactuar con gestos y/o voz, lo cual ayudará a evitar reconocimientos de gestos innecesarios, denominados (falsos positivos). Por ejemplo, un método que se ha utilizado y probado ampliamente es solicitar que el usuario realice un gesto del tipo saludo, del inglés *Wave*, para iniciar la interacción como se observa en la *Figura 125*.

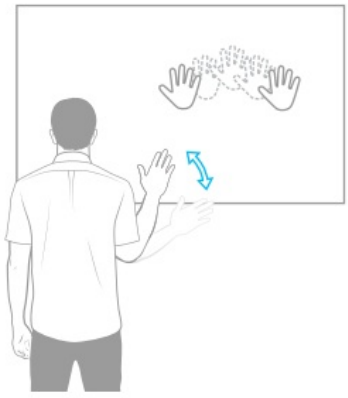


Figura 125: Gesto del tipo *Wave* [44]

Otras maneras de reconocer la intención de interactuar son en base a la distancia del usuario al sensor, la dirección con la cual el usuario se enfrenta, el número de personas visibles o mediante la detección de cierta pose determinada. También, se puede optar por reconocer cuándo un usuario se "desconecta" mediante una forma explícita en lugar de tener un método pasivo de tiempo de espera o de otro tipo. Por ejemplo, cuando un usuario da la espalda a la pantalla se puede considerar como una señal de que el usuario ya no está queriendo interactuar con el sistema. En definitiva, para poder determinar de la mejor manera la intención y participación de un usuario en el sistema se deberían seguir los lineamientos enumerados a continuación:

a. Brindar a los usuarios de forma clara y simple una vía de interacción para comenzar y terminar de interactuar con el sistema.
b. Establecer que los gestos o interacciones reconocidos sean un movimiento unívoco y con un propósito definido.
c. Establecer que los movimientos naturales del cuerpo no sean reconocidos como gestos. De esta forma, el usuario no debe alterar su comportamiento natural para realizar una correcta interacción.
d. Asegurar que no se pierdan interacciones intencionales (falsos negativos) ni se reconozcan interacciones no intencionadas (falsos positivos). Es muy importante distinguir las interacciones intencionales e ignorar otros movimientos, como por ejemplo, el realizado al tocarse la cara, el ajustarse de lentes, beber, entre otros. En este sentido, para aumentar la confianza, es esencial proporcionar al usuario la mayor cantidad de retroalimentación posible.

5. Diseñar sujeto a entradas variables: las experiencias y las expectativas de los usuarios pueden llevar a diferentes efectos del sistema en sus interacciones. Se debe tener en cuenta que la interpretación y/o representación de un gesto por parte de un usuario puede ser completamente diferente de la interpretación y/o representación que otro usuario le dé al mismo gesto. Un ejemplo de este tipo de ambigüedad es, por ejemplo, solicitar a un usuario que realice una onda, ya que si bien la mayoría de las personas saben lo que es una onda, el gesto en representación que realiza cada usuario posiblemente sea muy variable y por lo tanto no se garantiza que se genere el mismo movimiento. Algunas personas pueden realizar la onda con su muñeca, otras con el brazo entero, otras pueden utilizan la mano abierta que se mueve de izquierda a derecha, mientras que otras mueven sus dedos juntos hacia arriba y hacia abajo. Para estos casos, la predicción de la intención de lo que quiere realizar un usuario es un reto considerable, ya que no existe un medio físico para la detección de la intención como podría ser una pantalla táctil o el *mouse*. Una vez que el usuario ha iniciado una interacción con el dispositivo, el sistema está siempre activo en busca de patrones que coincidan con un gesto a fin de detectar lo que él mismo pretende realizar.

6. Diseñar para que la aplicación sea fiable: la fiabilidad, es decir, obtener una baja tasa de falsos positivos y negativos, debe ser la prioridad número uno para que la aplicación no se torne difícil de usar y provoque que los usuarios se sientan frustrados, ya que si la fiabilidad es pobre probablemente sucederá que los usuarios busquen utilizar otros tipos de mecanismos de interacción. Por ello, tratar de encontrar el equilibrio de una buena fiabilidad es clave y esto se verá afectado por el diseño del gesto. Para encontrar el punto de equilibrio en este sentido se debe considerar la frecuencia y el costo de las activaciones falsas. Si el gesto es demasiado rígido, no habrá activaciones falsas, pero puede ser difícil de realizar y los usuarios tienden a rechazarlo. Por otro lado, si el gesto es demasiado flexible, será fácil de realizar, pero puede tener demasiadas falsas alarmas y/o colisiones con otros gestos.

7. Diseñar conjuntos fuertes de gestos: se recomienda que se mantenga un número pequeño de gestos utilizados en la aplicación, tanto para que sea fácil de aprender y recordar por los usuarios como para que sean lo suficientemente distintos entre sí para evitar colisiones entre ellos. A su vez, es deseable que haya coherencia con otras aplicaciones, ya que esto ayudará a que los usuarios se sientan familiarizados con gestos típicos y altamente utilizados en otras aplicaciones, y se reduzca así la cantidad de gestos nuevos que debe aprender. Algunas características de importancia a tener en cuenta al definir un conjunto de gestos son las enumeradas a continuación:

    a. Es deseable asegurarse que los gestos similares utilicen la misma semántica. Por ejemplo, si apuntar a la izquierda lleva a una dirección, es deseable que un apuntar a la

derecha lleve a la dirección opuesta.

b. Es recomendable que cuando se deseen realizar gestos muy diferentes, las diferencias sean evidentes. De esta forma se trata de reducir los falsos reconocimientos y evitar así la duplicación de reconocimientos.
c. No se debe sobrecargar el usuario con demasiados gestos a recordar en una aplicación, ya que los usuarios sólo pueden recordar hasta seis gestos promedio de forma satisfactoria.
d. Siempre observar y utilizar gestos ya existentes que se han establecido en otras aplicaciones y que son de uso típico para ciertos tipos de interacción. Esto familiariza al usuario en la nueva aplicación definida.
e. Es conveniente agregar tiempo muerto o más comúnmente denominado tiempo de *cool-off*, que consiste en ignorar todas las entradas inmediatamente posteriores al reconocimiento de un gesto a modo de que el usuario pueda restablecerse a una posición neutral antes de iniciar la siguiente acción, reduciendo la cantidad de posibles falsos positivos.

8. Manejar consistentemente los gestos que se repiten: en ocasiones, los usuarios requieren llevar a cabo un cierto gesto repetidas veces, por ejemplo, múltiples desplazamientos para ir rápidamente a través de varias páginas o diapositivas de contenido. En estos casos, se debe contar también con el diseño del gesto opuesto que permita revertir la acción realizada por el gesto original.

9. Evitar los gestos para una mano específica: los gestos que se pueden realizar únicamente con una mano específica, derecha o izquierda, poseen una pobre accesibilidad y son más difíciles de detectar. Por ello, se recomienda diseñar los gestos para que puedan ser realizados con cualquiera de las dos manos de forma indistinta. De esta forma, se logra mitigar la fatiga del usuario, quien tiende a intercambiar de mano, evitando el esfuerzo adicional inherente en memorizar con qué mano debe realizar determinado gesto y a su vez, resulta más apropiado tanto para usuarios diestros como zurdos, haciendo que la aplicación tenga mayor accesibilidad.

10. Elegir entre gestos realizados con una mano o con ambas manos: los gestos con una sola mano son altamente preferibles por el simple hecho de mejorar la eficiencia de la realización/detección. Por ello, es importante que los gestos realizados con las dos manos no sean gestos frecuentes ni utilizados para tareas críticas, y en caso de ser necesarios, los mismos deben ser simétricos, es decir, ambas manos deben realizar el mismo movimiento para que sean más fáciles de realizar y recordar.

11. Evitar la fatiga del usuario: si el usuario se cansa, probablemente bajará su rendimiento y se sentirá frustrado, inclinándose por dejar de utilizar la aplicación debido a la mala experiencia. Existen algunas acciones y gestos que se conoce producen especial fatiga al usuario. Estos son, por ejemplo, la repetición excesiva de un solo gesto, el *scroll* en una página a través de una larga lista, mantener el brazo arriba o abajo para desplazarse a través de una lista vertical, entre otros. Es interesante entonces, ofrecer alternativas como pueden ser listas de navegación y posibilidad de realizar filtros sobre ellas, zoom in/out, etc.

12. Tener en cuenta la postura del usuario y los rangos de movimiento: con el fin de considerar dónde y cómo los usuarios van a utilizar la aplicación, es necesario considerar su postura para el diseño de un gesto, determinando las limitantes que se imponen en los movimientos, ya que, no es lo mismo, por ejemplo, estar sentado que estar de pie. Se debe tener en cuenta entonces, los rangos normales y por sobre todo, la comodidad de interacción para el cuerpo humano en las distintas posiciones. Otra consideración importante es tener en cuenta que no es lo mismo que el usuario se encuentre de frente o esté de lado al sensor, ya que los distintos puntos de vista alteran el seguimiento de su esqueleto.

13. Enseñanza y descubrimiento de los gestos: para cada gesto disponible como una forma de interactuar con el sistema, es preciso encontrar la manera de comunicarlo al usuario. Algunas formas de realizar esto puede ser mediante la realización de un tutorial rápido para nuevos usuarios, una imagen estática, una animación o una señal visual cuando el usuario entra en contacto por primera vez o durante toda la interacción. Es importante usar un método adecuado según el gesto que se está intentando que el usuario aprenda como se esquematiza en la *Figura 126*. En caso que el gesto sea estático, es preferible utilizar una imagen estática, mientras que si el gesto es dinámico, es más apropiado utilizar una animación para ilustrar su funcionamiento.

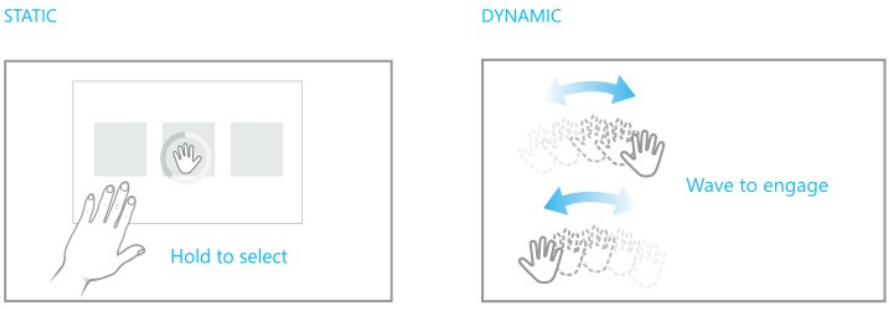


Figura 126: Cómo enseñar gestos estáticos o dinámicos [44]

14. Ser conscientes de los obstáculos técnicos: si para definir los gestos se hace uso de los datos del esqueleto del usuario, hay que tener en cuenta sus limitaciones y cuándo han de ser menos fiables. En la *Figura 127* se pueden visualizar algunos tipos de obstáculos a tener en cuenta siendo enumerados a continuación:

    a. Cuadrante I: los movimientos realizados con los brazos a los lados del cuerpo son más fáciles de seguir, y por por lo tanto, tienen una mayor fiabilidad; contrariamente a aquellos movimientos realizados en la parte frontal que poseen una menor fiabilidad.
    b. Cuadrante II: los gestos realizados por encima de la posición de la cabeza tienen mayor probabilidad de no ser reconocidos, a menos que el campo de visión del sensor *Microsoft Kinect* esté configurado de forma tal que abarque un espectro superior a la posición de la cabeza de los usuarios.
    c. Cuadrante III: la cantidad de *joints* a los cuales se le puede hacer seguimiento varía según la posición en la que se encuentre el usuario. Existen aproximadamente veinte *joints* que pueden ser seguidos cuando el usuario se encuentra de pie y sólo diez cuando se encuentra sentado.
    d. Cuadrante IV: se determina a su vez que el seguimiento esqueletal es más estable y fiable cuando el usuario se encuentra de cara al sensor.
    e. Cuadrante V: para gestos muy rápidos se debe tener en consideración otro conjunto de variables a modo de una buena detección, como por ejemplo, la velocidad de rastreo y los fotogramas por segundo o tasa a la que se está obteniendo información gestual.

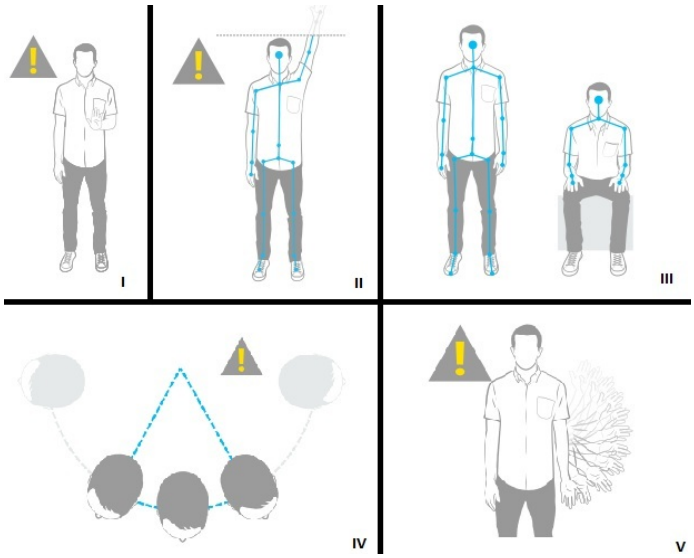


Figura 127: Posibles dificultades técnicas de reconocimiento con sensor *Microsoft Kinect* [44]

15. Mantener al público objetivo en mente: independientemente de cómo se definen los gestos, se debe mantener al público objetivo de la aplicación siempre en mente, de modo que los gestos se adecuen a los usuarios que harán uso final de dicha aplicación. Se debe prestar atención a toda la gama de distancias en que los usuarios se pueden plantar, los ángulos a los que pueden estar, sus rangos de altura, así como también, sus habilidades cognitivas. A modo de ejemplo, si la aplicación va a ser utilizada por adultos y niños, se deberá soportar varias alturas y longitudes de las extremidades, como se ilustra en la *Figura 128.* Los niños suelen hacer movimientos muy diferentes a los adultos para una misma acción debido a las diferencias en su destreza y control, y tienden a hacer los movimientos más rápidos y exagerados en comparación con los adultos.

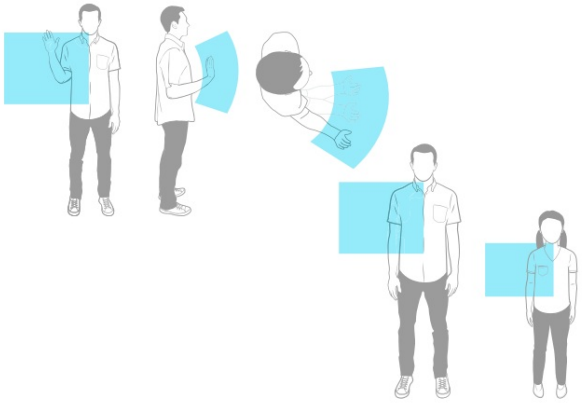


Figura 128: Consideraciones de rangos de movimiento de los usuarios [44]

16. Parametrizar e iterar: es importante diseñar los gestos de forma iterativa y paramétrica, a los efectos de poder realizar ajustes y testeo con usuarios de forma más sencilla y así obtener resultados más apropiados.

Como ya se ha mencionado anteriormente, es fundamental además proporcionar una buena retroalimentación en este tipo de interacción, con el objetivo de que los usuarios puedan sentir que tienen el control y permitirles así entender lo que realmente está sucediendo, sobre todo, si están de pie y a cierta distancia de la interfaz. Por lo tanto, es importante tener especial cuidado en mostrar las acciones o comandos que se van emitiendo a la aplicación y las respuestas que ésta proporciona.

Partiendo de lo más sencillo, un gesto sólo es eficaz y fiable cuando el usuario está en el rango correcto de visibilidad del sensor. Es importante a su vez, que el usuario sepa con certeza si el sensor está efectivamente viéndolo o no, razón por la cual es importante brindar la retroalimentación adecuada para ayudar al usuario a inferir si el sensor está listo, cuánto es capaz de reconocer, qué partes del usuario puede ver y cuáles no, a cuántas personas está viendo, cuándo y dónde realizar un gesto, y en general, ser capaz de brindar información de retroalimentación del seguimiento esqueletal. Algunas técnicas para brindar retroalimentación de este tipo son las ilustradas en la *Figura 129* y enumeradas a continuación:

1. Cuadrante I: la aplicación debe brindar un mecanismo para indicar dónde se debe parar el usuario a modo que el seguimiento sea óptimo.
2. Cuadrante II: se recomienda realizar movimientos suaves para evitar movimientos que no se asemejan a los realizados naturalmente por las personas.
3. Cuadrante III: brindar retroalimentación del tipo espejo, lo cual implica permitir al usuario visualizar sus acciones como si se tratara de un espejo, brindando información de seguimiento del elemento con el que está interactuando, sea una mano, su cabeza o un objeto de la aplicación. Se debe asegurar que las acciones se realicen en tiempo real, ya que de lo contrario el usuario supondrá que no tiene el control.
4. Cuadrante IV: en el caso hipotético que se quiera que el usuario copie un gesto o acción se puede mostrar un avatar o una animación, ya sea antes o durante el lapso en el que se espera que el usuario realice dicho gesto. En el último caso, es importante encontrar algún mecanismo visual que permita indicar en tiempo real si se está haciendo bien o no el gesto en cuestión.
5. Cuadrante V: es recomendable mostrar una pequeña ventana donde se pueda visualizar qué es lo que realmente el sensor puede ver en un momento dado, lo que puede ayudar a entender cómo mantenerse en el rango correcto de visión y entender por qué las cosas no funcionan cuando se está fuera de él.

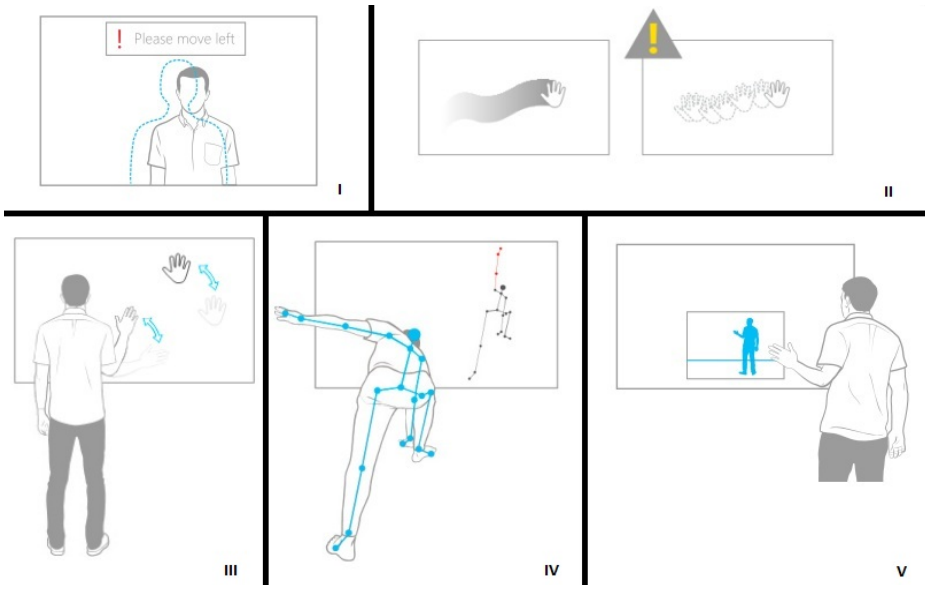


Figura 129: Técnicas de retroalimentación para el seguimiento esqueletal [44]

Existen también otros tipos de retroalimentación visual que permiten al usuario obtener información adicional del estado en el que se encuentra el sistema y brindar así información contextual de las acciones a poder realizar sobre el mismo. Algunos de dichos tipos son los que se enumeran a continuación:

1. Dejar en claro qué es procesable y como accionarlo: utilizar iconografía, colores o tutoriales para mostrar a los usuarios cómo diferenciar entre los textos estáticos, los controles que pueden accionar y los contenidos editables. Adicionalmente, se pueden utilizar elementos de

este tipo para mostrar los métodos de entrada disponibles para el usuario y qué gestos están disponibles en un momento dado. Otra práctica útil es mostrar siempre un cursor que permita visualizar el seguimiento de la mano en caso que se le esté realizando un seguimiento.

2. Mostrar qué se está sobrevolando y qué está seleccionado: el término "sobrevolar" se refiere a cuando el usuario tiene control de la interacción dado que, por ejemplo, a su mano se le está realizando un seguimiento, pero no está interactuando con ningún elemento particular de la aplicación. Se debe utilizar el color, la inclinación, los cambios de orientación o el tamaño en sobrevuelo para proporcionar información al usuario acerca de qué elementos son seleccionables y en qué estado se encuentran.

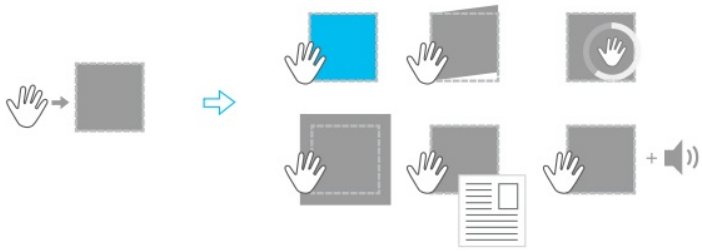


Figura 130: Retroalimentación para sobrevuelo y selección [44]

3. Manipulación directa de objetos: si se permite a un usuario controlar ciertos elementos con manipulación directa, es recomendable mostrar el progreso de manera que se traduzca en el movimiento que él mismo está realizando. A modo de ejemplo, si el usuario pasa la página de un libro con un gesto, una buena práctica es mostrar la página dándose vuelta cuando el usuario mueve su mano en sentido horizontal, tal como se muestra en la *Figura 131*.

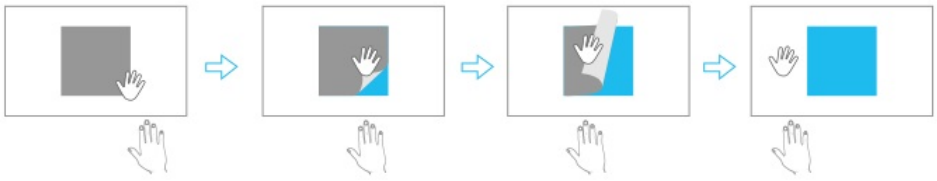


Figura 131: Retroalimentación para la manipulación directa [44]

4. Progreso: es de utilidad mostrar el grado de progreso para realizar la selección de un elemento, sobre todo para evitar que la aplicación parezca estar detenida. Es importante elegir tiempos que no frustren al usuario debido a una acción demasiado larga o cansadora. Si para mostrar el grado de progreso se está utilizando un contador de tiempo o una cuenta regresiva, es necesario utilizar elementos visuales claros para demostrar la progresión real.

5. Animación: la animación es una gran manera de ayudar a los usuarios a mantenerse en su contexto. Una animación puede ayudar, por ejemplo, a mostrar donde se reubica un contenido que se estaba viendo en cierto lugar cuando hay un cambio en la organización de los contenidos (debido a un gesto o por el mismo flujo de la aplicación). También es útil utilizar la animación para ayudar al usuario a entender el diseño en caso de que el usuario se encuentra navegando, tal como se muestra en la *Figura 132*.

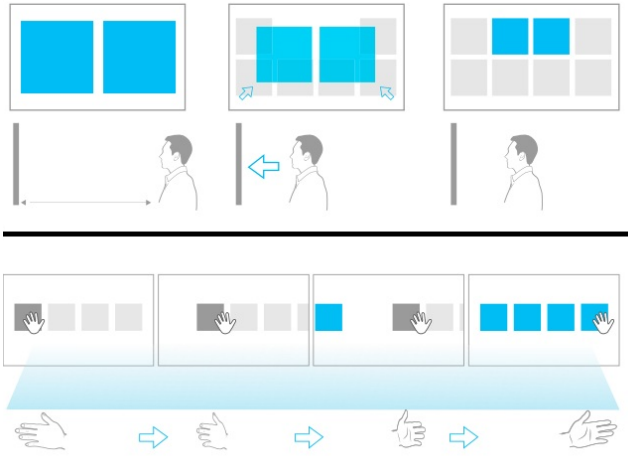


Figura 132: Retroalimentación tipo animación [44]

A su vez, adicionalmente de la retroalimentación visual existe también retroalimentación mediante otros medios, como por ejemplo, la retroalimentación auditiva, consistente en brindar información sobre las acciones del usuario mediante sonidos que permitan inferir el estado del sistema y el resultado de las diferentes acciones realizadas. Se enumera a continuación algunos casos en los que este tipo de *feedback* es preferible ante otros:

1. Conseguir la atención del usuario: el audio puede ser una buena forma de llamar la atención de los usuarios si tienen que ser notificados de algo. Los patrones de sonido o señales, como pueden ser alertas o advertencias, se pueden utilizar como forma de comunicar un mensaje simple o enseñar a los usuarios cuándo es necesario que presten mayor atención.
2. Estados de sobrevuelo o selección: los sonidos pueden ser una buena manera de indicar a un usuario de que algún elemento ha cambiado o pasó a otro estado. El uso de los sonidos que responden a la acción que el usuario está tomando pueden ser útiles para hacer valer su sensación de control y familiaridad. Un ejemplo típico de esto es reproducir el sonido de las teclas al introducir texto.
3. Comentarios de audio basados en la orientación del usuario: si el usuario no está frente a la pantalla o se encuentra muy lejos de ésta, se puede considerar el uso de notificaciones de audio como una forma de comunicarse con ellos. Por ejemplo, emitiendo las acciones y comandos disponibles, instrucciones, alertas, etc.

Otra característica que el sensor *Microsoft Kinect* ofrece, es la posibilidad de realizar el seguimiento de varios usuarios al mismo tiempo, abriendo una puerta para algunas interacciones interesantes que involucran colaboración entre los diferentes usuarios, así como algunos nuevos desafíos en torno al control y los recursos. *Microsoft Kinect* puede reconocer hasta seis personas y se puede realizar un seguimiento de dos de ellos de forma simultánea (aunque estas cantidades dependen fuertemente de la librería que se esté utilizando para controlar y acceder a la información del sensor). Para los esqueletos a los que se les realiza el seguimiento, *Microsoft Kinect* devuelve información completa de sus articulaciones, mientras que para los restantes, devuelve sólo cierta información útil especificando la silueta del usuario y una posición de centro, tal como se puede observar en la *Figura 133*. La aplicación puede elegir a cuál de los usuarios se le debe realizar el seguimiento del esqueleto completo en un momento dado.

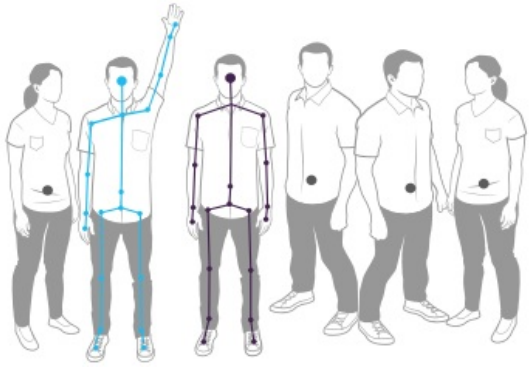


Figura 133: Detección de múltiples usuarios por el sensor *Microsoft Kinect* [44]

Al realizar el seguimiento a varios usuarios, se debe tener en cuenta cómo la distancia afecta la cantidad de personas que pueden estar físicamente en el rango de visión del sensor. Los valores recomendados para interactuar cómodamente con la aplicación, o al menos, ser detectado por ella, son los que se enumeran a continuación:

1. Modo táctil (de 0 a 0.4 metros): detección de 0 personas.
2. Modo cercano (de 0.4 a 2 metros): detección de 1 o 2 personas.
3. Modo lejano (de 2 a 4 metros): detección de 1 a 6 personas.
4. Fuera de rango (más de 4 metros): detección de 0 personas.

Tener una aplicación que soporte varios usuarios significa que se pueden tener muchas combinaciones de usuarios a diferentes distancias, por lo cual, dependiendo del modelo de colaboración, hay algunas consideraciones importantes que deben ser tenidas en cuenta para la buena gestión de todos los usuarios en la escena. Para las interacciones colaborativas, es decir, aquellas en que los usuarios están interactuando con el mismo espacio de pantalla y sensor, donde cualquier acción tomada por un usuario afectará el estado de la solicitud de todos los usuarios, existen dos modelos para el control del flujo de la interacción en lo que respecta a la colaboración y son los enumerados a continuación:

1. Controlador único: este modelo asigna a uno de los usuarios el rol de conductor en un momento dado y sólo registra las acciones adoptadas por éste. El papel de conductor puede seleccionarse o transferirse de varias formas, por ejemplo, mediante la elección manual del usuario a tomar este rol o mediante la selección automática del conductor en base al usuario que esté más cerca del sensor. Esta es una manera de evitar conflictos entre entradas, indicando por imágenes a qué persona se le está realizando un seguimiento como conductor, utilizando un único cursor en pantalla en todo momento.

2. Igualdad de participación: este modelo registra los aportes de todos los usuarios, a menudo simplemente en el orden que se dan. Este puede ser un modelo muy complejo para navegar o para realizar interacciones básicas, pero funciona muy bien para situaciones en las que cada usuario tiene una experiencia única dentro de la aplicación, como puede ser por ejemplo, conducir un vehículo en un juego de carreras.

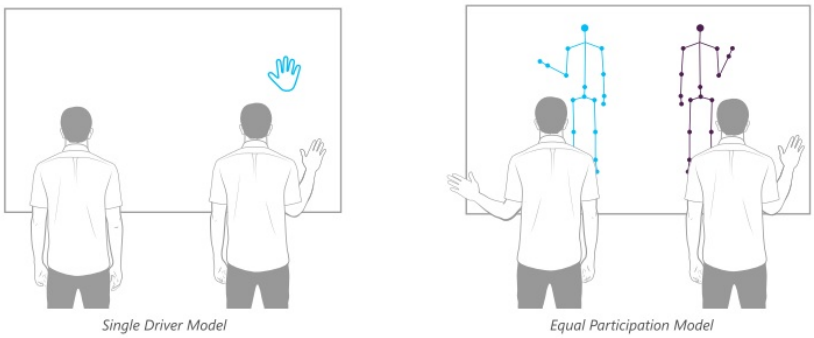


Figura 134: Modelos de participación para interacciones colaborativas [44]

Por otro lado, existen las denominadas "interacciones no colaborativas", siendo aquellas que brindan a cada usuario su propia experiencia dentro de la interfaz mediante, como puede ser por ejemplo mediante una pantalla subdividida. Algunos retos adicionales que se presentan en este caso son la correcta detección del habla del usuario y las acciones de la interfaz correspondiente a los comandos por voz emitidos por cada uno, complementado al hecho de que los gestos deben tener una retroalimentación visual para cada usuario por separado.

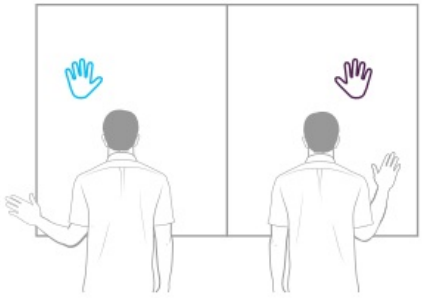


Figura 135: Modelo de participación para interacciones no colaborativas [44]

Anexo G: Otros proyectos estudiados

G.1. *Put-that-there*

Este trabajo fue llevado a cabo por Richard A. Bolt, quien para entonces formaba parte del *Architecture Machine Group* del *MIT* [120]. Su desarrollo data del año 1980, lo cual lo hace particularmente interesante debido a que permite demostrar que desde los comienzos de la computación personal se ha estudiado y analizado la interacción gestual y la interacción mediante comandos por voz. La idea por detrás del proyecto fue construir una solución en la que el computador del usuario no estuviera limitado a un monitor, sino que fuera la habitación completa. En definitiva, algo similar a lo que se busca con el presente proyecto, o al menos el objetivo más general al que éste apunta, que es poder contar con superficies con las que se puedan interactuar de forma ubicua y transparente en todo momento.

La solución consiste de una interfaz gráfica con la que se puede interactuar mediante una combinación de gestos y comandos de voz con el objetivo de manipular objetos que son presentados en una proyección sobre una pared. El sistema es capaz de realizar síntesis de voz, brindando información sobre el estado del sistema y solicitando más detalles al usuario en caso de existir ambigüedades con el comando emitido. El usuario se encuentra ubicado justo enfrente de la pantalla, ya sea sentado o de pie como se puede observar en la *Figura 136*, y es capaz de utilizar comandos por voz en combinación con gestos para agregar nuevos elementos, mover y/o eliminar elementos ya existentes.

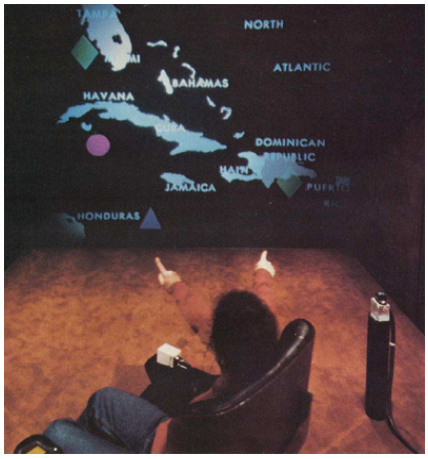


Figura 136: Usuario emitiendo órdenes al sistema *Put-that-there del MIT* [120]

En cuanto a la estructura utilizada para dar soporte al sistema, consiste de una habitación de 5 metros de altura, 3x3 metros de ancho y largo respectivamente, en la que se incorporan altavoces y computadoras que controlan tanto la pantalla como los diferentes dispositivos existentes. Las imágenes son proyectadas en una de las paredes gracias a un proyector a color ubicado en una habitación contigua. Adicionalmente, en cada posabrazo de la silla se cuenta con *joysticks* sensibles a la presión y dirección, y junto a estos, un *touchpad* sensible al tacto. Por último, se incluyen monitores sensibles al tacto a cada lado de la silla en la que está sentado el usuario. En cada uno de estos monitores se muestra una versión simplificada de lo que se está mostrando en la pantalla principal, incluyendo un la selección actual del usuario dentro de la escena y los diferentes elementos que se están visualizando en la pantalla principal. El usuario puede cambiar su selección utilizando el *touchpad* derecho ubicado en los reposabrazos de su silla o directamente seleccionando una nueva posición en alguno de los monitores. El área seleccionada es mostrada en la pantalla principal con más detalle, tal como si se hubiera realizado un gesto de *zoom in* en dicha área. Es posible también realizar los gestos de *zoom in* y/o *zoom out* para acercarse y alejarse respectivamente en la escena, mediante el *touchpad* ubicado en el posabrazo izquierdo de la silla del usuario.

En cuanto a la tecnología utilizada para el reconocimiento del habla, *Put-that-there* utiliza el sistema *DP-100 Connected Speech Recognition System* (*CSRS*) desarrollado por *Nippon Electric Company (NEC)* [22]. En esa época, la mayoría de los sistemas de reconocimiento del habla estaban limitados a reconocer expresiones discretas, debiendo emitir los comandos palabra a palabra ya que el reconocimiento de palabras conectadas era aún un desafío. Sin embargo, el sistema de *NEC* era capaz de reconocer una cantidad limitada de palabras conectadas sin pausas entre ellas. Luego de cada oración, el sistema genera la respuesta en solo 300 ms y su vocabulario provee hasta 120 palabras en modo normal y hasta 1000 en el modo de expresiones discretas.

Por otra parte, en lo que refiere a la tecnología utilizada para el posicionamiento del usuario dentro de la escena, se utiliza el sistema *Remote Object Position Attitude Measurement System* (*ROPAMS*) desarrollado por *Polhemus Navigation Science*. Este sistema está basado en el electromagnetismo y consiste de dos pequeños cubos de plástico que permiten hacer un seguimiento de la posición espacial de un objeto [128]. Cada uno de estos cubos incluye tres bobinas ortogonales que representan los tres ejes de coordenadas X, Y y Z. El primer cubo es el transmisor, que emite un campo magnético constante, y el segundo cubo es el sensor, siendo muy liviano ya que va atado a la muñeca del usuario (o en general, al objeto para el cual se quiera conocer su posición espacial). La organización de las bobinas en cada cubo básicamente crea una antena sensible en las tres orientaciones, y la distancia hacia el cubo que hace de sensor es computada en base a características de la señal emitida desde el transmisor. Finalmente, la posición es determinada mediante la transformación de las diferentes señales de las tres bobinas ortogonales en el sensor.

De esta forma, el espacio gráfico virtual y el espacio real en el que está ubicado el usuario dentro de la habitación convergen para crear un único espacio interactivo continuo, haciendo que el usuario pueda apuntar, realizar gestos o referenciar lugares o zonas mediante comandos por voz como "arriba", "abajo" o "a la izquierda de" de forma natural. Algunos de los comandos disponibles son los siguientes [16]:

1. Crear: la pantalla principal está inicialmente vacía o eventualmente muestra algún fondo particular como puede ser un mapa o similar. Sobre este fondo se pueden agregar diferentes elementos y luego interactuar con ellos para modificarlos, moverlos, eliminarlos, etc. Los posibles elementos a utilizar son figuras básicas, como cuadrados, círculos, rombos, etc., cada una con ciertos atributos variables como el color y el tamaño. De esta forma, por ejemplo, el comando de voz "*create a circle here*" permite crear un círculo en el lugar de la pantalla que se esté apuntando con el dedo (indicado con un cursor a modo de retroalimentación). En la *Figura 136* se puede apreciar el estado del sistema luego de haber creado varios elementos en pantalla.
2. Mover: el sistema permite mover cualquiera de los objetos mostrados en pantalla de una gran variedad de formas diferentes, algunas de las cuales son:

    a. Emitiendo comandos por voz del estilo de "*move the blue circle to the right of the green square*" para mover un cierto elemento referenciado en el mismo comando hacia una posición relativa a otro elemento existente.
    b. Emitiendo comandos por voz del tipo "*move this to the right of the green square*", donde se debe apuntar con la mano a cuál de los elementos se refiere simultáneamente a la emisión del comando.
    c. Haciendo honor al nombre del sistema y emitiendo el comando "*put that there*", debiendo señalar de forma simultánea al comando el elemento que se quiere mover y luego el lugar hacia donde quiere ser movido.
    d. Modificar atributos: se puede modificar los atributos de color y tamaño de un elemento mediante comandos similares a los detallados anteriormente. Por ejemplo, "*make that smaller*" hará que el objeto al que se está apuntando se haga más pequeño. Incluso se pueden copiar los atributos de un objeto a otro objeto, señalando primero al objeto destino, luego el objeto origen y emitiendo un

comando de tipo "*make that like that*".

3. Eliminar: mediante el comando "*delete that*" se puede eliminar el objeto al que se esté apuntando. Adicionalmente, se puede emitir el comando "*delete everything*" para limpiar la pantalla completa.

Así, mediante estos comandos las indicaciones de lo que se quiere hacer con los elementos visibles en pantalla puede ser expresado de forma natural y espontánea, siguiendo el uso más intuitivo en este tipo de grandes pantallas que se encuentran distantes, es decir, señalar objetos para seleccionarlos y luego emitir una acción mediante comandos por voz. Esto es un punto a tener en cuenta en el presente proyecto, en el sentido de considerar aquella forma de interacción más intuitiva según la superficie con la que se está interactuando.

G.2. *DreamSpace*

DreamSpace es un proyecto de cierta forma similar al detallado anteriormente, al menos en lo que respecta a funcionalidad y modo de uso [24]. En este caso, el sistema fue desarrollado por *IBM Research* en el año 1997, y permite que múltiples usuarios colaboren en un espacio compartido de forma simultánea. A su vez, como cuenta con una red de alta velocidad los usuarios puedan trabajar colaborativamente incluso de forma remota y manipular gran cantidad de datos debido a su gran capacidad de procesamiento.

DreamSpace permite reconocer comandos por voz emitidos por los usuarios, realizar el seguimiento de la posición de su cuerpo y reconocer ciertos gestos predefinidos. De este modo, permite a los usuarios interactuar con la computadora de forma natural de manera similar a cómo lo harían con otra persona, sin necesidad de instrucciones ni conocimientos previos, ni de utilizar ningún tipo de dispositivo especial. Así, los usuarios se pueden enfocar en las ideas, mensajes y significado de la información presentada, minimizando eventuales distracciones. La computadora está presente sólo como una pantalla en la pared que muestra imágenes y emite sonidos, pero no requiere el uso ni de teclado, ni de ratón, ni de cables, sino únicamente de dos entradas, una de video y otra de audio respectivamente. A su vez, al igual que en el sistema "*Put-that-there*", es posible combinar los comandos por voz con señalización para crear elementos en cierta posición de la pantalla o modificar alguno de sus atributos [2, 22].

Figura 137: Usuario interactuando con el sistema *DreamSpace* [24]

La voz es capturada mediante un micrófono que no necesariamente debe ser cargado por el usuario sino que puede formar parte de la habitación. El reconocimiento de los comandos emitidos por el usuario es realizado por el sistema *IBM ViaVoice*, un sistema de *IBM* específico para reconocimiento y síntesis del habla. Por otra parte, la entrada de vídeo es capturada a partir de una cámara que se sitúa encima de la pantalla y es procesada mediante algoritmos de visión por computadora. El audio y el vídeo son tratados de forma paralela y el resultado de este análisis se combina para proporcionar una salida adecuada que se

muestra al usuario a través de la pantalla. Tanto el componente *IBM ViaVoice* para reconocimiento del habla como el módulo de visión por computadora operan en una *IBM IntelliStation Z-Pro*, una de las computadoras comercializadas por *IBM* a fines de la década del 90', que incluye una tarjeta gráfica dedicada para proveer la potencia requerida para la visualización de los gráficos. En esta misma computadora se ejecuta la interfaz de integración y comunicación y el *software* de aplicación *DreamSpace*. Un esquema de alto nivel de la solución es el ilustrado en la *Figura 138*.

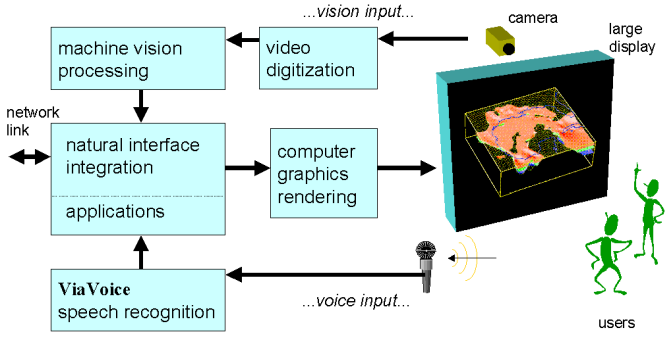


Figura 138: Diagrama de componentes *hardware* y *software* de *DreamSpace* [24]

El presente proyecto se correlaciona con *DreamSpace* en el sentido de que también busca la colaboración de múltiples usuarios en un espacio compartido de forma simultánea, realizando el seguimiento de la posición de cada uno de ellos. La entrada de vídeo se pretende capturar de la misma forma a partir de una cámara, como es la proporcionada por el sensor *Microsoft Kinect*, que es luego procesada y analizada mediante algoritmos de visión por computadora para reconocer las acciones de los usuarios. A su vez, también es crucial para el presente proyecto el reconocimiento de ciertos gestos de forma tal que el usuario pueda interactuar lo más natural posible, sin necesidad de conocimientos previos ni del uso de ningún tipo de dispositivo especial intrusivo que deba colocarse. Sin embargo, a diferencia de *DreamSpace*, el presente trabajo no se pretende incluir el reconocimiento de voz ni ningún tipo de interacción mediante esta vía.

G.3. *Omnitouch*

Otro proyecto llevado a cabo por *Microsoft Research* en el 2011 es un prototipo denominado *Omnitouch: Wearable Multitouch Interaction Everywhere* [102]. *Omnitouch* consiste en un sistema portable basado en una cámara de profundidad y un pico-proyector que hace posible que cualquier superficie que esté dentro de su alcance sea convertida en una superficie multitáctil. Este prototipo presenta más restricciones que el detallado en la subsección anterior pero ofrece mayor precisión a la hora de detectar los dedos y las diferentes interacciones realizadas por el usuario sobre la superficie. El sistema se compone de un pico-proyector y un sensor *Microsoft Kinect*, montados de manera rígida en el hombro del usuario y conectados a una computadora de escritorio para el procesamiento de los datos obtenidos. Así, *Omnitouch* provee las mismas funcionalidades que cualquier *touch-screen* en cualquier superficie dentro su alcance, permitiendo visualizar objetos virtuales sobre ésta e interactuar con ellos de forma multitáctil. En la *Figura 139* se visualiza cómo está conformado el sistema y cuál es su ubicación relativa al usuario

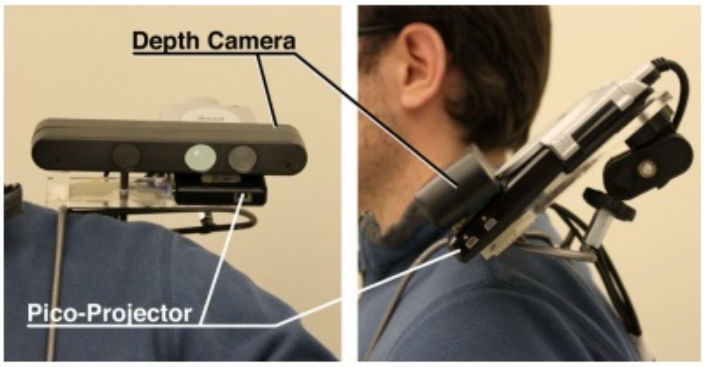


Figura 139: Prototipo para hombro *Omnitouch* [102]

Una notoria diferencia de *Omnitouch* con las soluciones descritas anteriormente es que no se limita a un espacio físico predefinido, sino que se caracteriza por tener completa movilidad, eliminando la restricción de que la superficie sea la misma durante todo el ciclo de funcionamiento. Incluso la superficie de interacción puede modificarse a demanda del usuario, lo que hace que el sistema presente gran portabilidad y flexibilidad. El prototipo para hombro permite además que se puedan manipular superficies de varias naturalezas, tal como se puede apreciar en la *Figura 140*. Así, se puede utilizar superficies del entorno cotidiano como paredes, pizarras, mesas, partes del cuerpo como brazos o manos y hasta otros tipos de objetos como cuadernos o tabletas.

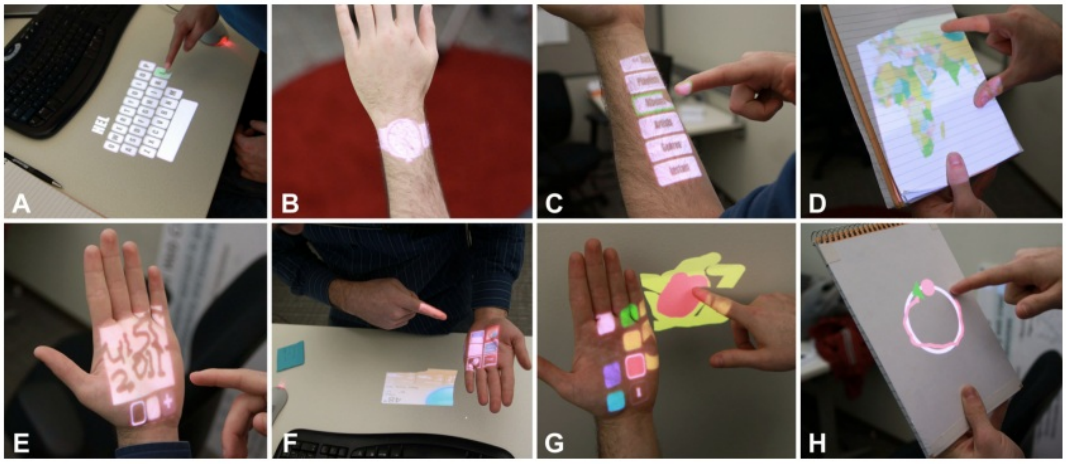


Figura 140: Diferentes aplicaciones creadas para *Omnitouch* [102]

Otra ventaja particularmente novedosa del sistema *Omnitouch* es que no requiere de entrenamiento ni calibración previa, ya que se calibra en tiempo real a demanda, haciendo que siempre esté disponible. Por otro lado, gracias a la cercanía de la superficie táctil al sensor de profundidad, *Omnitouch* presenta una gran precisión que le permite detectar los dedos del usuario y sus movimientos de forma eficiente, en contraste con soluciones que utilicen los sensores de profundidad a distancias mayores. El sistema tiene la funcionalidad de detectar las tres coordenadas de los dedos y la capacidad de diferenciar si están tocando o sobrevolando una superficie. Con este mecanismo, en la práctica se pueden detectar *clicks* realizados con los dedos cuando se encuentran a 1 cm o menos de distancia con la superficie. A partir de los 2 cm de distancia se considera que está sobrevolando, y entre 1 y 2 cm la respuesta es ambigua (puede ser una u otra aleatoriamente). A grandes rasgos, la detección de los dedos consta de los siguientes pasos:

1. Obtener el mapa de profundidad de la escena.
2. Derivar el mapa según el eje X y el eje Y utilizando deslizamiento. En la *Figura 141.b* se utiliza el color azul para el eje X y el color rojo para el eje Y.
3. Buscar posibles candidatos a dedos mediante la detección de desplazamientos similares a

cilindros. Para ser considerados candidatos, deberán tener una pendiente positiva de la derivada seguida inmediatamente de una región de suavidad, terminando con una pendiente negativa. Lo anterior se puede observar más claramente en la *Figura 142*. Es de vital importancia que el orden sea respetado ya que de otra manera se podrían detectar objetos cóncavos en lugar de cilindros.

4. Los candidatos deberán medir entre 5 y 25 mm de diámetro, medida típica de un dedo.
5. Cuando todos los candidatos están identificados, se procede a agrupar los próximos entre sí en caminos contiguos. De esta manera, los más cortos o más largos son descartados.

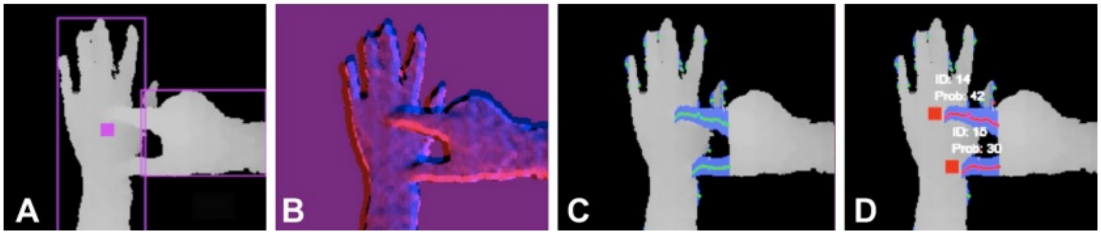


Figura 141: Proceso de detección de dedos [102]

Esta técnica consistente en derivar el mapa de profundidad trae consigo una serie de beneficios importantes. En primer lugar, permite que la escena sea tratada como una típica imagen en dos dimensiones, permitiendo que sea más fácil de procesar utilizando técnicas de visión por computadora. Adicionalmente, independientemente de la superficie donde los dedos se encuentren, su derivada es invariante, lo cual ayuda a simplificar el proceso de reconocimiento. Sin embargo, en este prototipo existen algunas restricciones en lo que respecta a la detección de dedos que agregan algunos limitantes a la solución. En primer lugar, se asume que la mano a ser detectada es la derecha. Por otro lado, se asume que los dedos de dicha mano se encuentran completamente estirados.

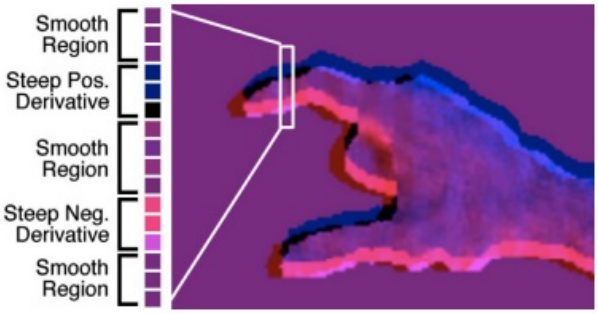


Figura 142: Ejemplo de detección de un candidato a dedo [102]

Dadas las restricciones de Omnitouch, no es posible aplicar los algoritmos presentados para el presente proyecto dado que muchos de los gestos a detectar naturalmente requieren que los dedos no estén completamente estirados. Por otro lado, el método propuesto por *Omnitouch* para la detección de dedos implica una carga de procesamiento que para el presente proyecto agregaría demasiado retardo en la detección de gestos, imposibilitando la fluidez en la interacción.

G.4. *Interactions in the Air*

Interactions in the Air: Adding further depth to interactive tabletops [48] es un proyecto desarrollado por *Microsoft Research* en el año 2009 cuyo objetivo es alternar la interacción sobre una superficie con la interacción por encima de ella, logrando una interacción más intuitiva en la manipulación de contenido digital. El trabajo pretende establecer una forma de interacción que se asemeje a la manipulación de los objetos físicos, tal como se está acostumbrado a realizar en el día a día en el mundo real. En particular, como ejemplo de interacción por encima de la mesa, el sistema permite tomar una pelota e introducirla en un taza, gesto que utilizando únicamente interacción directa sobre la superficie no

se podría llevar a cabo de forma natural debido a las limitaciones bidimensionales de la superficie, tal como se muestra en la *Figura 143*. De esta forma, los gestos por encima de la mesa no reemplazan los gestos directos sobre la mesa, sino que los complementan de buena manera.

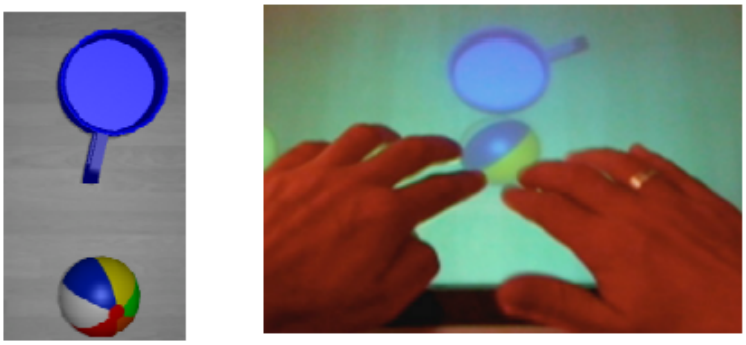


Figura 143: Limitaciones *Interactions in the air* [48]

Como ya se ha mencionado anteriormente en el presente informe, para poder interactuar por encima de la superficie hay que, o bien instrumentar al usuario (p.ej.: mediante guantes especiales) o bien instrumentar el ambiente donde se encuentra (p.ej.: mediante cámaras de profundidad) a modo de poder sensar lo que sucede en la escena en todas las dimensiones, en particular, en el eje Z. En *Interactions in the Air* se implementan dos enfoques diferentes para captar la entrada del usuario en el eje Z: uno basado en la utilización de cámaras *RGB* regulares complementadas con un sistema de iluminación difusa infrarroja (*IR*) que permite calcular la altura de las manos y detectar gestos sencillos, y otro basado en una cámara de captura de profundidad permitiendo interacciones más complejas.

En lo que respecta a la pantalla donde se muestra la información, lo más común para superficies planas como mesas o escritorios, es que sea un dispositivo plano como las pantallas *LCD* o bien proyecciones directas sobre las superficies. Para la representación gráfica en tres dimensiones, hasta hace no muchos años se solía tener que instrumentar al usuario con pantallas estereoscópicas y gafas de obturación o pantallas de realidad virtual o realidad aumentada. Hoy en día, se ha avanzado a tal punto que existen pantallas autoestereoscópicas capaces de direccionar los rayos a cada uno de los ojos de los usuarios de forma que no se necesite ningún tipo de dispositivo adicional para poder observar el efecto en tres dimensiones en los objetos desplegados. Sin embargo, están apuntadas al uso de un sólo usuario y son muy dependientes de su punto de visión, por lo que ésta técnica hace que no sea demasiado considerada para la interacción con mesas. Debido a estas limitaciones presentes en el contexto de la interacción sobre una mesa, en *Interactions in the Air* se opta por tener una salida en dos dimensiones basados en dos enfoques diferentes. Por un lado, una pantalla basada en la tecnología *SecondLight* de *Microsoft Research* [135] que permite presentar imágenes en dos dimensiones mediante el alternado de proyecciones difusas a una tasa lo suficientemente alta como para que el ojo humano no pueda captar la diferencia. Por otro lado, una pantalla basada en la tecnología *TouchLight*, también desarrollada por *Microsoft Research* [150], que permite implementar una pantalla multitáctil combinando la salida de dos cámaras de video posicionadas justo detrás de un plano semi transparente ubicado frente al usuario.

Teniendo en cuenta las formas descritas sobre la captura de las interacciones del usuario y las formas de presentar la información, *Interaction in the Air* plantea la construcción de un sistema del tipo *rear projection-vision*, es decir, un sistema en el que los componentes que habilitan tanto la interacción como la visualización se encuentran por detrás de la superficie, en contraste con un enfoque *top-down*, en el que los componentes se ubican por encima de la superficie. Mediante esta forma se evitan algunos problemas conocidos del enfoque *top-down* como la oclusión y la complejidad de desarrollo.

Otro punto desafiante a considerar es que se tiene una entrada en tres dimensiones para una salida en dos dimensiones, por lo que existirán acciones que no van a estar correspondidas de forma directa, como por ejemplo, aquellas llevadas a cabo por encima de la superficie de la mesa. Los usuarios pueden

interactuar con los objetos presentes en la mesa de forma multitáctil para mover los objetos virtuales en dos dimensiones, así como también tomar el objeto realizando un gesto de *pinch* o *pick up* encima de la superficie de la mesa, tal como se muestra en el primer cuadro de la *Figura 144*. Para detectar el momento en que un usuario realiza el *pick up* de un objeto, se utiliza un algoritmo que detecta en tiempo real cuándo el usuario acerca su dedo pulgar a su dedo índice luego de haber reconocido el hueco existente entre ambos dedos. El algoritmo retorna la posición en dos dimensiones del centro de masa del espacio entre los dedos, así como también su orientación. Con estos datos, se traza un rayo para determinar con qué objeto interseca y así saber qué objeto se desea mover. Una vez que se deja de detectar el gesto, el objeto vuelve a quedar en estado estático.

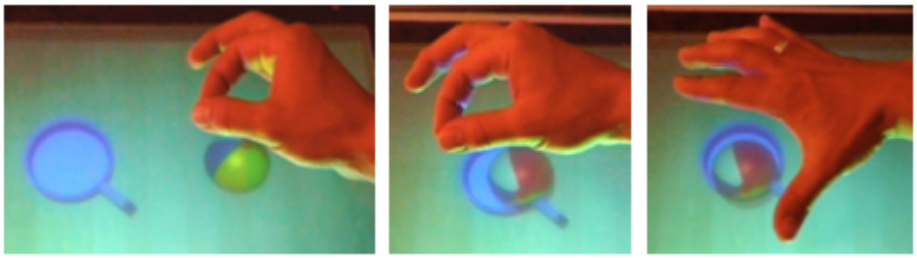


Figura 144: Gesto de *pick up* sobre un objeto [48]

Adicionalmente, a modo de retroalimentación se utiliza una técnica consistente en la generación de sombras utilizando la información obtenida por las cámaras de profundidad, lo cual permite al usuario conectar su mano en el mundo real con el objeto virtual que está moviendo. De esta forma, se trata de aumentar la percepción por parte del usuario para brindar la sensación de que está interactuando de forma directa con los objetos de la escena. Una primera solución consiste en dibujar directamente los datos crudos obtenidos por la cámara, pero esto presenta el problema de que no toma en cuenta la altura de las manos para dibujar las sombras acordemente. Una alternativa consiste en generar un mapa de sombras para todos los objetos presentes en la escena y luego combinarlo con el generado a través de las imágenes provenientes de las cámaras de profundidad. Así, el valor almacenado en el mapa final será el mayor valor de profundidad obtenido. En este caso, a medida que la mano se aleja de la mesa, la sombra mostrada es de menor tamaño, lo cual es algo contradictorio a lo que sucede en el mundo real pero hace que la información necesaria sea menor.

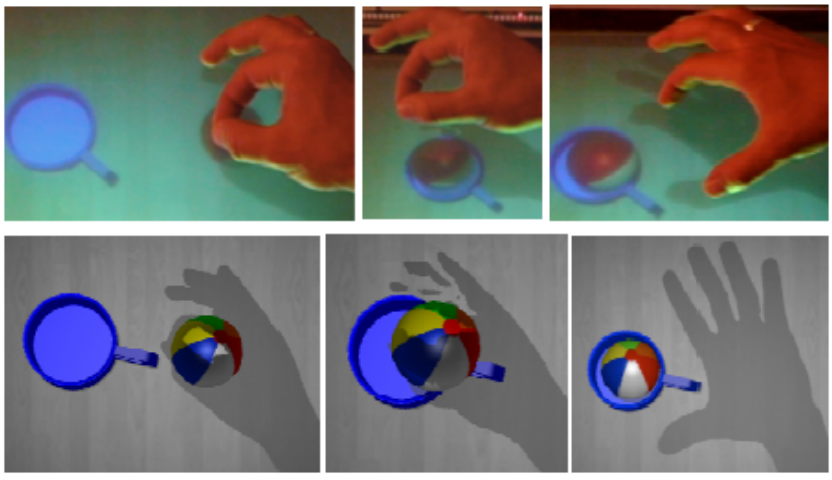


Figura 145: Simulación de sombras durante la interacción [48]

De forma semejante a *Interactions in the Air*, el presente proyecto establece que los gestos por encima de la mesa no reemplazan los gestos sobre la mesa sino que los complementan con el fin de mejorar la interacción. En ambos casos se posee una entrada en tres dimensiones y una la salida en dos dimensiones, por lo que se debe resolver el desafío referente a que existan acciones que no estén correspondidas de forma directa con lo que se está visualizando. *Interactions in the Air* presenta varias

técnicas para dar realismo a esta situación, una de las cuales está basada en la técnica de generación sombras. Sin embargo, para este trabajo esta técnica no presenta grandes beneficios dado que la interacción en tres dimensiones se realiza sobre los sensores *Leap Motion* y no directamente sobre los objetos representados en pantalla.

G.6. *Code Space*

Code Space: Touch + Air Gesture Hybrid Interactions for Supporting Developer Meetings [15] es un proyecto llevado a cabo por *Microsoft Research* a finales del año 2011. El proyecto consiste en integrar gestos en dos y tres dimensiones para posibilitar reuniones más eficientes entre personas físicamente presentes en una misma sala. Su objetivo principal es democratizar el acceso, el control y el intercambio de información a través de múltiples dispositivos tanto personales como compartidos, como por ejemplo pantallas multitáctiles, *smartphones*, tabletas, *laptops*, etc. La motivación por detrás del trabajo es posibilitar reuniones específicamente para desarrolladores de *software*, las cuales generalmente se llevan a cabo en salas en las que sólo se puede conectar un proyector a un único dispositivo. Esto implica que no se pueda tener acceso por parte de todos los usuarios de forma democrática a lo que se está visualizando, y hace que se tienda a elegir a un miembro de la reunión para que oficie de presentador o moderador de la reunión. A su vez, los usuarios por lo general llevan información relevante en sus dispositivos personales, que no es posible compartir con los demás asistentes ya que realizar este tipo de intercambio entre dispositivos de forma manual requiere cierto tiempo adicional, interrumpiendo el flujo de la reunión y distrayendo a los participantes.

La sala de reuniones presentada por *Code Space* se puede visualizar en la *Figura 146*. Ésta incluye una pantalla multitáctil compartida de acceso público para todos los intervinientes y dos sensores *Microsoft Kinect* para detectar los gestos y las posiciones de los usuarios. Uno de estos sensores es colocado por encima de la pantalla multitáctil, en dirección a aquellos usuarios que estén ocupando el rol de presentadores. El otro sensor se ubica direccionado a los usuarios que se encuentran en la audiencia, es decir, sentados en la mesa de frente a la pantalla multitáctil.

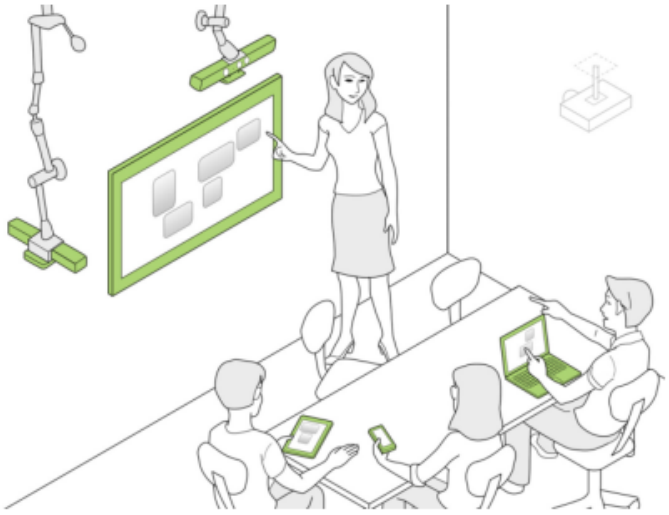


Figura 146: Sala de reunión de *Code Space* [15]

Un posible flujo de interacción en *Code Space* podría ser, por ejemplo, cuando un usuario interactúa con el sistema mediante su *smartphone* a modo de control remoto apuntando con él hacia la pantalla multitáctil compartida. Cuando esto ocurre, se visualiza un cursor en la pantalla compartida gracias al seguimiento del brazo del usuario realizado por el sensor *Microsoft Kinect* ubicado de frente a los usuarios de la audiencia. De este modo, el usuario puede interactuar con los elementos existentes en la pantalla compartida a través de la pantalla multitáctil de su celular pudiendo transferir nuevos elementos, seleccionar elementos existentes para moverlos, compartirlos y/o borrarlos. Esto hace que se proporcione

una interacción híbrida que combina por una lado, la señalización en el aire mediante el dispositivo celular, y por otro, gestos multitáctiles realizados sobre la pantalla del celular que permitan confirmar y completar las acciones. Este escenario puede incluso extenderse de forma colaborativa, en el sentido de que si otro usuario está manipulando la pantalla compartida al mismo tiempo, puede por ejemplo, seleccionar un objeto transferido hacia la pantalla y copiarlo a su dispositivo personal. De forma más genérica, se utiliza seguimiento esqueletal en base a movimientos familiares y simples como apuntar, mover, etc., y gestos multitáctiles socialmente aceptados y precisos para completar y complementar estas acciones.

Las interfaces basadas en el seguimiento esqueletal tienen un gran conjunto de ventajas, aunque también poseen algunas desventajas que deben ser tenidas en cuenta e intentar resolver en la medida que se quiera hacer uso de este tipo de interacción, algunas de estas desventajas nombradas por el proyecto *Code Space* son las que se enumeran a continuación:

1. Se debe evitar un extenso uso de movimientos de manos y brazos ya que puede no ser aceptado socialmente.
2. Siempre existe una cierta tasa de reconocimientos erróneos, por lo que se debe tener en cuenta la posibilidad de manejar posibles falsos positivos y negativos.
3. Se debe suplir la falta de información sobre las acciones disponibles de manera conveniente.

A su vez, es sabido que los gestos que implican que los participantes realicen largas acciones son los más rechazados, así como también gestos inusuales, gestos incómodos y aquellos que puedan interferir con la comunicación. Por otro lado, en lo que concierne al uso de comandos por voz, en el contexto de una reunión generalmente potencia la distracción en lugar de ayudar a generar una interacción más eficiente y además, puede llegar a generarse ambigüedades debido a la cantidad de personas involucradas. Algunas estadísticas obtenidas de un estudio sobre la aceptabilidad de los diferentes mecanismos de interacción que se realizó como parte del proyecto *Code Space* son las enumeradas a continuación:

1. El 98% de los encuestados indicó que se sentiría cómodo en hacer uso de una pantalla multitáctil en reuniones.
2. El 93% indicó que se sentiría cómodo en interactuar con dicha pantalla apuntando desde el lado de la audiencia.
3. El 80% indicó que se sentiría cómodo realizando pequeños gestos con las manos a modo de comandos.
4. Sólo el 29% indicó que estaría cómodo en hacer largos gestos con los brazos, el resto consideró el hecho como embarazoso y como un distractor del cometido de la reunión.
5. El 95% indicó que se sentiría cómodo en usar dispositivos táctiles como *tablets* o *laptops touch* para interactuar con la pantalla compartida.
6. El 80% indicó que se sentiría cómodo utilizando *smartphones* para compartir elementos con los demás asistentes.

Por otro lado, en la sala de reuniones donde varios usuarios están interactuando a distancia, es importante poder identificar cuál de los participantes es el que se encuentra ejecutando determinada acción, ya que, por ejemplo, en el caso que alguien esté moviendo un objeto de forma remota no se sabría quien está llevando a cabo dicha acción y aparentaría que el objeto virtual se está moviendo solo de un lado hacia otro. Si bien al utilizar dispositivos táctiles para interactuar con la pantalla compartida se logra una interacción directa y precisa, por otro lado oculta quién es que está realizando dicha acción, y a su vez, interfiere con las habilidades humanas usuales para manipular objetos en una pantalla táctil. Para resolver esto, se plantea combinar gestos multitáctiles, señalización en el aire y gestos a modo de proporcionar una interacción híbrida. Más específicamente, se utiliza seguimiento esqueletal para identificar modos y operandos en base a movimientos familiares y simples como apuntar y confirmar, y completar las acciones mediante gestos del tipo multitáctiles los cuales son más socialmente aceptados y precisos.

Dado que el objetivo que se plantea en este proyecto es lograr fluidez y compartir de forma

democrática información sobre una pantalla común, *Code Space* plantea cumplir con las siguientes seis nociones básicas de diseño:

1. Todos los usuarios pueden interactuar con la pantalla compartida, desde cualquier sector de la sala y cualquiera sea el dispositivo que posean.
2. Las interacciones deben ser socialmente aceptadas, sin causar movimientos embarazosos o que perturben la fluidez de la reunión.
3. Cada modalidad debe tener un uso bien definido.
4. Las interacciones deben posibilitar gestualidad entre dispositivos.
5. Las interacciones deben manifestarse a los participantes para crear conciencia de las posibles acciones a realizar.
6. Las formas de interactuar entre los dispositivos deben ser sencillas a modo de evitar un aprendizaje engorroso y la introducción de errores.

Para cumplir con las nociones de diseño enumeradas se establece los siguientes aspectos tenidos en cuenta para el desarrollo del proyecto *Code Space*, comenzando por aspectos relacionados al diseño en general, luego aspectos específicos sobre las posibles formas de compartir información y finalmente, algunos aspectos de diseño extra que fueron contemplados:

1. *Code Bubbles*: la información a mostrar en la pantalla multitáctil es representada a través de objetos denominados burbujas como se muestra en la *Figura 147*, donde cada una de ellas despliega información diferente. A estas burbujas, se las puede manipular mediante comandos multitáctiles que posibilitan moverlas mediante *touch-dragging* (tocar y arrastrar), cambiar el tamaño mediante un gesto de *pinch* y realizar un gesto de *pan* o deslizamiento para recorrer toda la escena con la burbuja seleccionada.

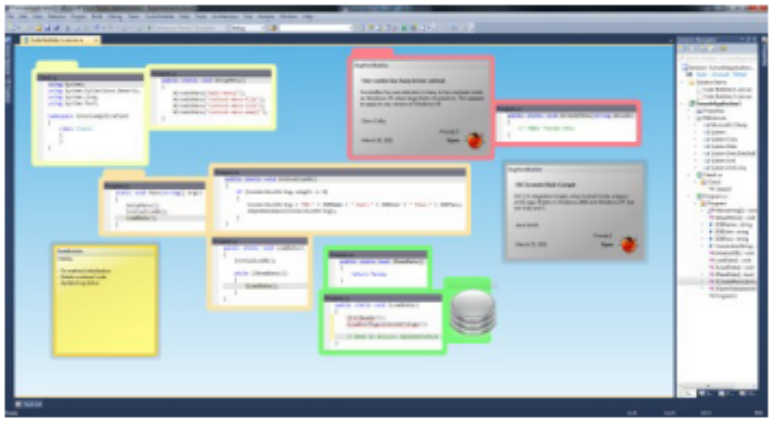


Figura 147: Representación de objetos digitales mediante burbujas [15]

2. Interacción a distancia: es posible tener acceso democratizado a la pantalla multitáctil desde diferentes dispositivos que posea el usuario, como son *smartphones* y *laptops* multitáctiles, o simplemente mediante la utilización de sus brazos. Para ello, la implementación de *Code Space,* que es una extensión del entorno de desarrollo *Microsoft Visual Studio*, ejecuta tanto en la pantalla compartida así como también, en los dispositivos multitáctiles mencionados anteriormente. Para el caso de los *smartphones* se cuenta con una aplicación que permite conectarse a *Code Space*, permitiendo así visualizar cierta cantidad de las burbujas presentes en la pantalla compartida. A su vez, tanto en esta aplicación como en las que ejecutan en la pantalla compartida y en las *laptops* se permite realizar acciones de *debugging* y edición sobre las diferentes burbujas.

3. Señalización mediante el uso de los brazos: cuando un usuario apunta a la pantalla utilizando

su brazo, se detecta dicha acción mediante la utilización del seguimiento esqueletal y se dibuja, como consecuencia de dicho reconocimiento, un cursor a modo de retroalimentación, tal como muestra la *Figura 148*.

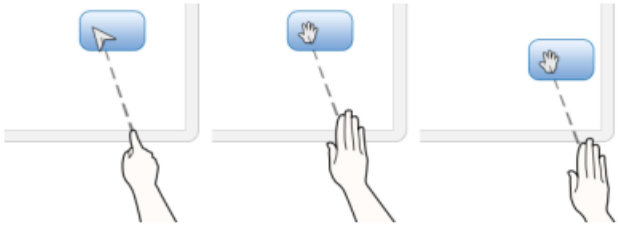


Figura 148: Señalización y manipulación con brazos y mano [15]

4. Manipulación mediante el uso de los brazos: el usuario puede manipular las burbujas de la pantalla compartida realizando un gesto en donde la palma de la mano apunte a la pantalla y los dedos se encuentren extendidos. Cuando el sistema detecta este gesto cambia al modo *drag* (arrastrar), informado al usuario mediante un cambio de cursor mientras el gesto sea mantenido. En este modo, el usuario puede mover las diferentes burbujas a modo de reordenarlas de la forma que desee.

5. Señalización y manipulación mediante el uso de brazos y *smartphone*: una alternativa a la descripción anterior es que en lugar de que el usuario apunte a la pantalla utilizando únicamente su brazo, éste lo haga apuntando a la pantalla compartida con su *smartphone*, logrando de esta forma interacción entre los dos dispositivos, *smartphone* y pantalla compartida. En base al seguimiento esqueletal se reconoce el gesto del brazo estirado y se muestra un cursor en la pantalla. Para mover un objeto el usuario debe apuntar a él, tocar sobre la superficie del *smartphone* y mover el dispositivo. Cuando deja de tocar la pantalla la acción de mover se da por finalizada, quedando el objeto en la nueva posición, tal como se muestra en la *Figura 149*.

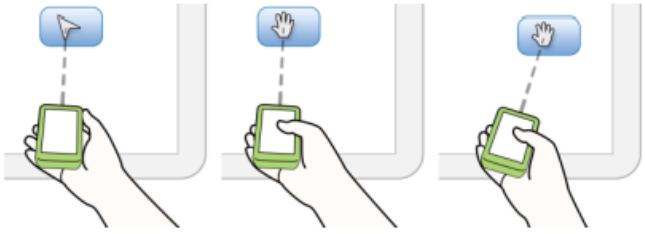


Figura 149: Señalización y manipulación con brazos y dispositivo [15]

6. Anotaciones temporales: para potenciar el uso de la señalización *Code Space* incluye la posibilidad de realizar marcas temporales a modo de dibujo sobre la pantalla compartida. Estas anotaciones se realizan a través de la selección de un icono en la aplicación del *smartphone*, pudiendo dibujar realizando movimientos en el aire mientras se mantenga seleccionado. Cuando el usuario deja de presionar el icono en el dispositivo las marcas desaparecen.

7. Gestos de la audiencia: los gestos realizados en el aire dentro de la escena de interacción presentan ciertos problemas, siendo alguno de ellos los que se enumeran a continuación:

    a. Activación accidental: como el sistema captura todos los movimientos de los usuarios se podría llegar a interpretar gestos que no fueron llevados a cabo de forma intencional como parte de la interacción, sino que son gestos involuntarios.
    b. Segmentación ambigua: los gestos son por naturaleza continuos, por lo que son de por sí difíciles de segmentar.

c. Respuesta táctil: los gestos puramente hechos en el aire no poseen respuesta de tipo táctil, como sí la tienen los gestos realizados sobre una superficie multitáctil. Sin embargo, cuando por ejemplo, un dedo y el pulgar se presionan juntos al realizar un gesto puramente en el aire, hay información táctil que pasa desapercibida y no se toma en cuenta.

Para mitigar los problemas antes mencionados, se diseñó una forma de interacción en donde, en lugar de hacer gestos en el aire a modo de comandos de entrada, se utiliza la información de seguimiento esqueletal para activar diferentes modos en los dispositivos de los usuarios. De esta forma, se remueven eventuales comandos que puedan activarse de forma accidental si fuesen permitidos de realizar de forma directa mediante gestos en el aire. Así, los diferentes modos que posibilita el sistema se activan mediante gestos, para los cuales las acciones se ejecutan desde el dispositivo de usuario mediante gestos táctiles. Por ejemplo, en la *Figura 150*, el usuario apunta hacia la pantalla con su celular y como consecuencia se despliega la información sobre qué gestos se pueden realizar, luego realiza un gesto de *swipe* hacia la derecha y la pantalla completa se mueve en esa dirección.

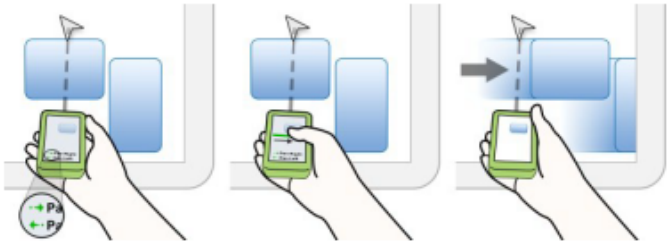


Figura 150: Interacción entre dispositivos [15]

8. Espacio extra: *Code Space* también hace uso de espacios proyectados por fuera de la pantalla multitáctil, permitiendo que, por ejemplo, las paredes oficien de un espacio útil en donde se puede copiar burbujas fuera de la pantalla compartida. Si bien la altura puede llegar a ser incómoda para aquellos usuarios que no alcancen a aplicar técnicas multitáctiles, el sistema brinda la posibilidad de realizar señalización en el aire y *swipes* para poder interactuar con este tipo de zonas de otra forma inalcanzables. Este comportamiento se puede observar en la *Figura 151*.

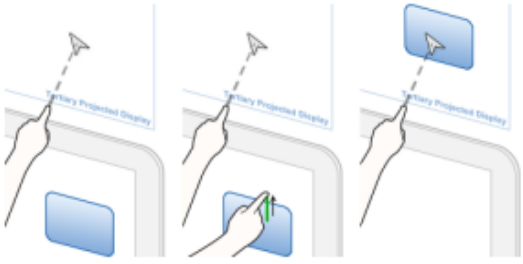


Figura 151: Mover burbujas fuera de la pantalla multitáctil [15]

9. Paletas: las paletas de herramientas fijadas en cierto lugar pueden llegar a ser ineficientes en el contexto de pantallas muy grandes, ya que esto requeriría que el usuario se desplace hasta donde se encuentra ubicada la paleta. Una alternativa natural sería abrir un menú contextual tocando en el fondo de la pantalla con un dedo, pero esto se podría mezclar con el fondo en casos donde la pantalla tenga demasiados elementos. Así, el enfoque que sigue *Code Space* en estos casos es abrir un menú contextual reconociendo el gesto de la palma en paralelo a la pantalla con los dedos extendidos. Cuando se reconoce este gesto se cambia el color en el área debajo de la palma a modo de retroalimentación y se abre la paleta mientras el gesto siga siendo reconocido, tal como se muestra en la *Figura 152*. De esta forma, el usuario tiene la posibilidad de abrir la paleta en el momento que quiera, en el lugar que quiera y sin la

necesidad de tocar la pantalla. Adicionalmente, si el usuario quiere fijar la paleta en cierta posición puede hacerlo tocando el fondo de la pantalla con cualquiera de los dedos utilizados para abrir la paleta. Mediante este criterio, se sigue la regla de establecer modos mediante posturas y acciones mediante gestos táctiles.

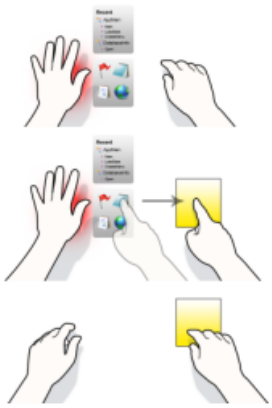


Figura 152: Representación de paletas de herramientas [15]

10. Dibujado en pantalla: un problema conocido en lo que refiere al dibujado en una pantalla es identificar cuándo el usuario pretende utilizar este modo. Para determinar esto, se podría tocar un botón pero se estaría ante el problema de que el usuario puede requerir desplazarse para poder hacer uso de él. Es por esto que el enfoque seguido por *Code Space* es el ilustrado en la *Figura 153*, el cual consiste en reconocer el gesto con la palma paralela a la pantalla y los dedos extendidos para luego tocar la pantalla con un dedo de la otra mano, dibujando entonces como si la superficie fuese una pizarra. De esta forma, el usuario no debe aprender otro gesto sino que utiliza el mismo que para el caso de las paletas contextuales, diferenciándose en que para realizar dibujos debe tocar la pantalla con un dedo de la otra mano, mientras que para abrir un menú contextual no. Más aún, apenas se establece contacto con el dedo el modo es reconocido y ya no es necesario mantener el gesto con la palma de la mano. En este caso, el modo dibujo permanece hasta que el dedo que dibuja deje de estar en contacto con la superficie.

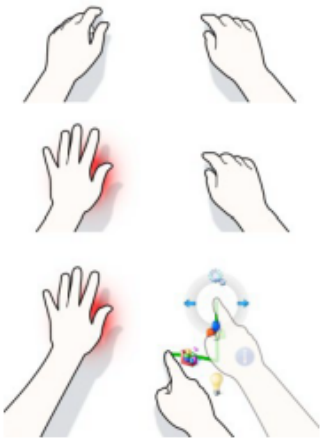


Figura 153: Gesto para dibujar [15]

11. Modos de interacción: *Code Space* es capaz de determinar cuántas personas están presentes en la pantalla como presentadores y cuántas están formando parte de la audiencia. Esta información es utilizada para hacer que la interacción sea lo más sencilla posible, mejorando así la usabilidad global. A continuación, se enumeran los cinco modos de interacción posibles en *Code Space* con el fin de que la solución sea lo más sencilla, robusta y predecible posible:

a. *Ambient display mode:* el sistema se encuentra en este modo de interacción cuando no se encuentran ni presentadores ni audiencia en la escena. Este modo cambia luego de pasados dos minutos desde que un usuario entra a la sala o interactúa con la pantalla, momento en el cual el sistema muestra el calendario de reuniones así como también un listado de *bugs.*
b. *Single speaking presenter:* el sistema se encuentra en este modo cuando el presentador se encuentra de cara a la audiencia pero distante de la pantalla. En este momento, el sistema oculta los elementos de ayuda al presentador (paletas y mensajes) ya que no las está mirando y oscurecen la visibilidad del contenido presente en la pantalla a la audiencia.
c. *Single working presenter:* el sistema se encuentra en este modo cuando hay un presentador de cara a la pantalla. En este momento, el sistema despliega los elementos de ayuda al presentador, visualizando las paletas, los mensajes, las posibles acciones, etc.
d. *Two or more working presenters:* el sistema se encuentra en este modo cuando dos o más presentadores están de cara a la pantalla. Dado que todos los usuarios presentes puedan desplazarse y hacer *zoom* en la pantalla compartida, *se* pueden presentar potenciales problemas. Para evitarlo, la pantalla es dividida automáticamente cuando más de un presentador se encuentra trabajando sobre ella, incluyendo un sector individual para cada uno con el mismo contenido y las mismas posibilidades de acciones a ejecutar.
e. *Audience only (working meeting):* el sistema se encuentra en este modo cuando solamente la audiencia está presente en la escena, es decir, no se cuenta con un presentador.

12. Señalización y gestos 2D: el usuario puede copiar burbujas desde la pantalla compartida hacia su dispositivo o viceversa. Para esto, el usuario debe apuntar con el dispositivo a la burbuja a copiar y hacer un gesto de *swipe* hacia abajo sobre el dispositivo. De esta forma, el tamaño de la burbuja compartida se reducirá acorde al tamaño del *display* destino y podrá manipularse directamente en el dispositivo. De forma análoga, realizando un gesto de *swipe* hacia arriba sobre el dispositivo y apuntando a una sección de la pantalla se pueden copiar objetos hacia la pantalla compartida. Este mecanismo se puede apreciar en la *Figura 154*.

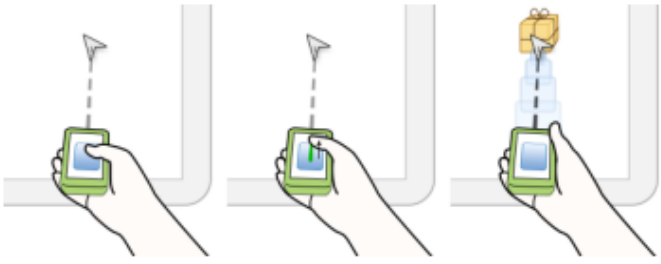


Figura 154: Compartir objetos mediante un dispositivo de tipo *smartphone* [15]

Para los casos en los que se quiera compartir contenido desde una *laptop* o *tablet* se debe apuntar con un brazo extendido hacia el lugar donde se quiere copiar en la pantalla. De esta forma, se entra en un modo en el que se muestra un mensaje en el dispositivo, se reconoce el gesto y se brinda información de las posibles acciones a realizar. Luego, haciendo un gesto de *swipe* hacia arriba con la otra mano la burbuja es copiada hacia el lugar señalado. Por otro lado, si se quiere mover una burbuja desde la pantalla compartida hacia el dispositivo de usuario se debe realizar un gesto análogo, señalando la burbuja a copiar con una mano y haciendo un *swipe* hacia abajo con la otra para copiar la burbuja al dispositivo destino. Este mecanismo se puede visualizar en la *Figura 155*.

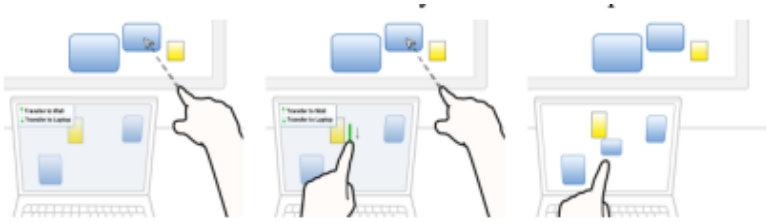


Figura 155: Compartir objetos mediante dispositivo de tipo *laptop* o *tablet* [15]

13. Compartir de forma temporal: *Code Space* brinda la posibilidad de compartir contenido de forma temporal mediante un dispositivo de tipo *smartphone*, para ello el usuario lo debe colocar de cara a la pantalla compartida y con el brazo extendido, el dispositivo de forma que quede perpendicular al piso. Así, como se ilustra en la *Figura 156*, el contenido será compartido de forma temporal hasta que el usuario baje su mano. En cambio, para *tablets* y *laptops* esto no se realiza manteniendo dicho dispositivos suspendidos ya que son más pesados y resultaría incómodo, sino que para estos casos se realiza un gesto con el brazo extendido y la mano perpendicular al piso, mientras la otra mantiene contacto con la pantalla del dispositivo. Adicionalmente, si se desea que la transferencia temporal se haga de forma permanente el usuario que oficia de presentador puede realizar un gesto tocando el objeto compartido y luego arrastrándolo para que se agregue a una burbuja.

Figura 156: Compartir objetos de forma temporal [15]

14. Compartir entre usuarios: de forma análoga a lo planteado anteriormente, el sistema puede transferir información entre los participantes de la reunión, siendo posible enviar información a otros usuarios simplemente apuntando su dispositivo hacia la dirección del usuario destino.

15. Organizar el compartir de contenido: en reuniones a las cuales asisten muchas personas puede ocurrir que todas deseen compartir cierto contenido en la pantalla, lo que puede ocasionar que la misma se llene rápidamente de información. Para evitar esto, *Code Space* incluye el concepto de *package metaphor*, haciendo que se visualice un icono con forma de paquete cuando un usuario transfiere un objeto a la pantalla compartida. Así, si el usuario envía varios objetos de forma secuencial, todos serán agrupados bajo el mismo paquete como se puede observar en la *Figura 157*. Para visualizar su contenido, el presentador debe posicionar su dedo encima del paquete mientras que para abrir y acceder a sus elementos debe realizar un tap sobre el mismo.

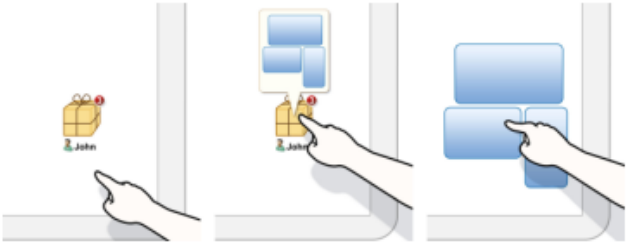


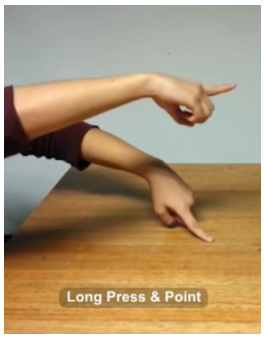
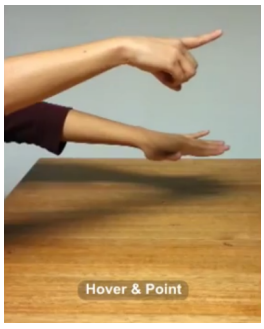
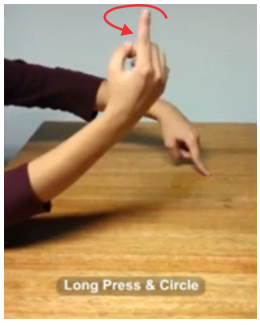
Figura 157: *Package metaphor* [15]

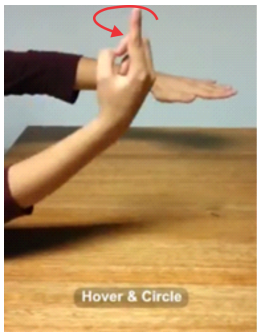
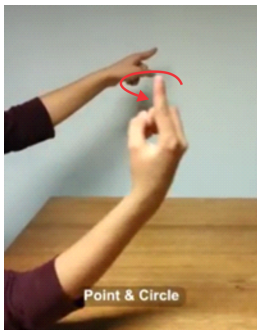
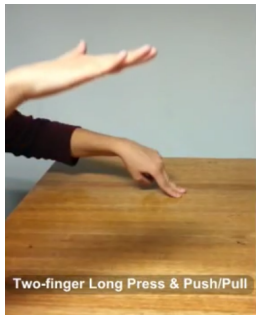
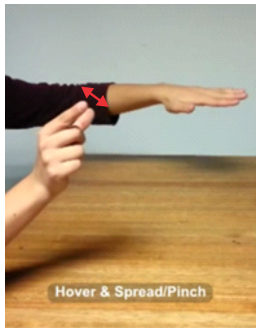
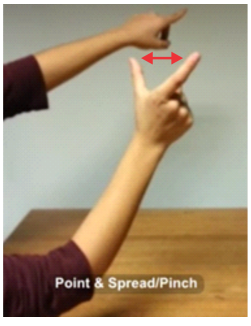
16. Compartir desde el presentador: el sistema también permite que el presentador comparta burbujas con los usuario de la audiencia, ya sea con uno solo o con todos los presentes en la sala. Así, para compartir una burbuja con una única persona, debe apuntar hacia ella mientras toca la burbuja a compartir, visualizando en pantalla las posibles acciones a realizar. Por otra parte, para compartir cierto contenido con todos los asistentes, el presentador debe dirigir su brazo en dirección a la audiencia pero en este caso, apuntando hacia el piso.

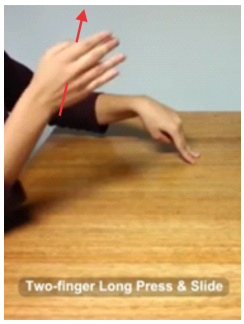
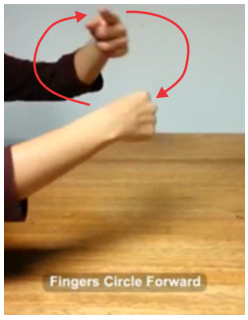
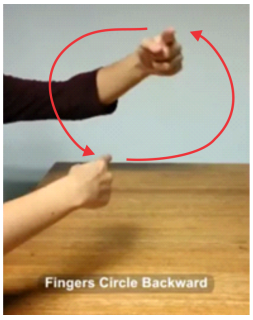
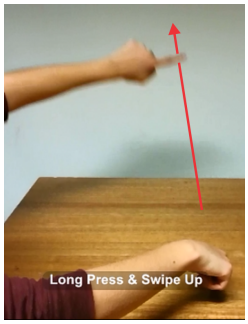
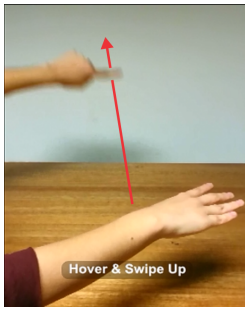
Anexo H: Definición de un banco de gestos

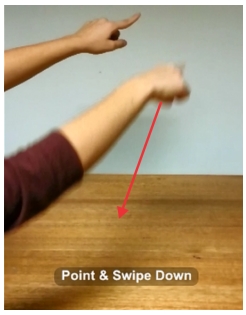
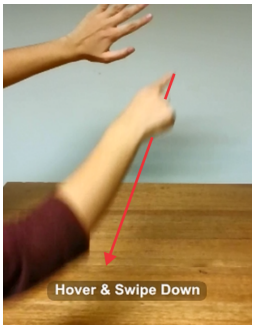
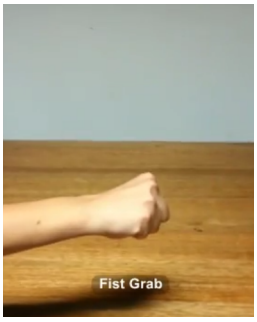
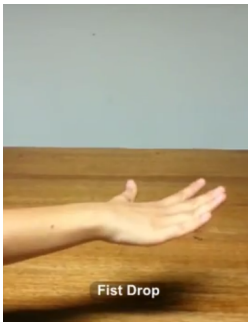
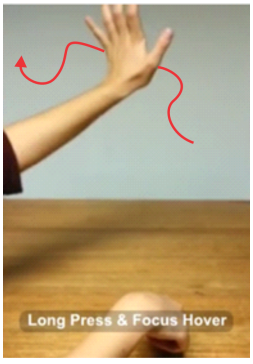
Adicionalmente a los gestos tanto multitáctiles como espaciales ya mencionados, para el presente trabajo se definió un conjunto de gestos propios considerados de interés. Estos gestos fueron pensados desde el punto de vista referente al proyecto donde se encuentra inmerso este trabajo, es decir, en el contexto de una oficina del futuro. Este grupo de gestos son gestos multitáctiles, espaciales y una combinación de ellos, a los cuales se les denominó naturalmente gestos híbridos.

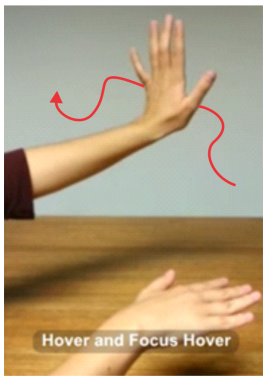
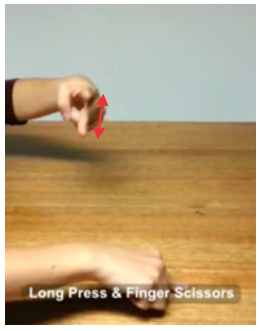
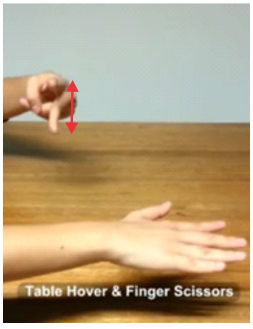
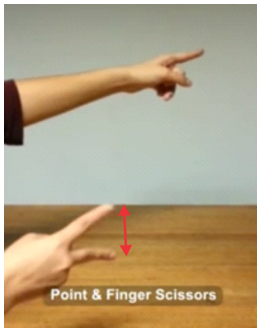
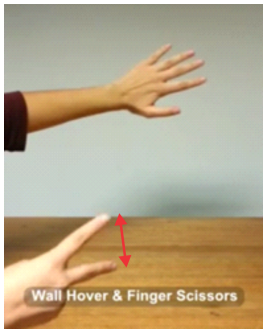
En la *Tabla 13* se detalla cada uno de los gestos definidos, mostrándose en rojo la trayectoria de movimiento necesaria para realizar el gesto en cuestión. Adicionalmente, se pueden visualizar los gestos en acción en el vídeo de demo *"Definición de un banco de gestos"* [v.1].

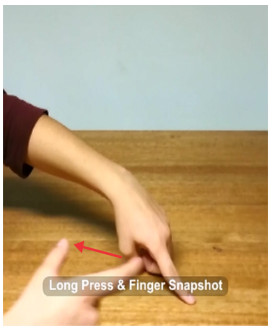
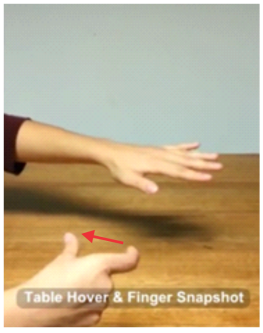
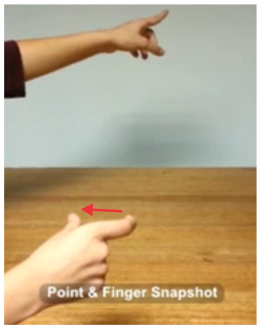
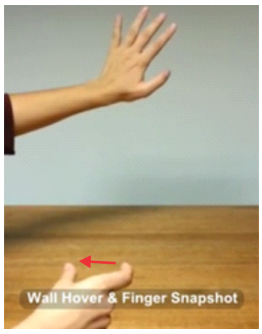
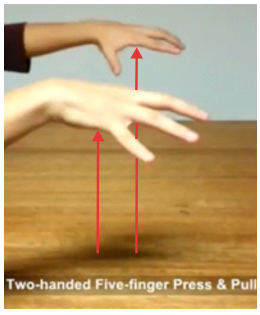
Nombre	Gesto	Descripción	Ejemplo de uso
Long Press & Point		Consiste en presionar con un dedo de una mano un objeto por un cierto lapso de tiempo y luego apuntar con un dedo de la otra mano un objetivo posiblemente distante.	Seleccionar un objeto, específico, como por ejemplo una presentación, mediante el gesto *Long Press* y que el mismo sea copiado a una superficie que se encuentra en la dirección del gesto *Point*.
Hover & Point		Consiste en mantener una mano paralela a la superficie por un cierto lapso de tiempo y apuntar con un dedo de la otra mano un objetivo posiblemente distante.	Seleccionar un conjunto de objetos, como por ejemplo diagramas colaborativos, que se encuentran bajo el área del gesto *Hover* y que los mismos sean copiados a una superficie que se encuentra en la dirección del gesto *Point*.
Long Press & Circle		Consiste en presionar con un dedo de una mano un objeto por un cierto lapso de tiempo y realizar un círculo con un dedo de la otra mano.	Seleccionar un objeto, específico, como por ejemplo un archivo de audio, mediante el gesto *Long Press* y que el mismo sea compartido con todos los presentes mediante el gesto *Circle*.

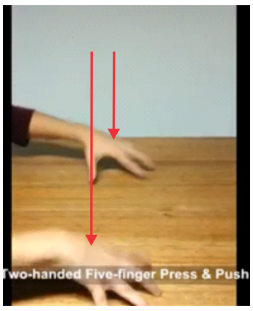
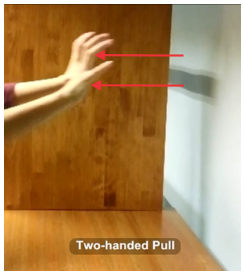
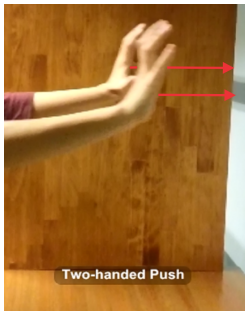
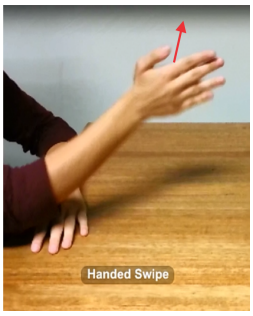
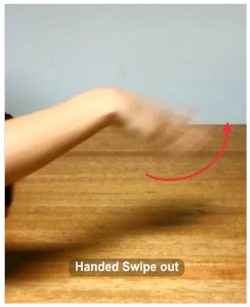
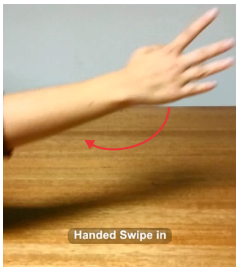
Hover & Circle		Consiste en mantener una mano paralela a la superficie por un cierto lapso de tiempo y luego realizar un círculo con un dedo de la otra mano.	Seleccionar un conjunto de objetos, como por ejemplo archivos de audio, que se encuentran bajo el área del gesto *Hover* y que los mismos sean compartidos con todos los presentes mediante el gesto *Circle*.
Point & Circle		Consiste en apuntar con un dedo de una mano un objetivo posiblemente distante y luego realizar un círculo con un dedo de la otra mano.	Seleccionar un objeto específico, como por ejemplo una representación de un objeto 3D,posiblemente distante, mediante el gesto *Point* y que el objeto sea puesto a girar mediante el gesto *Circle*.
Two-finger Long Press & Push/Pull		Consiste en presionar con dos dedos de una mano un objeto por un cierto lapso de tiempo, manteniendo la otra mano horizontal y desplazándola de arriba hacia abajo y viceversa.	En un modo escena que afecte a todos los elementos mediante el gesto *Two-finger Long Press* y que, por ejemplo, se maximice o minimice todos los objetos de la superficie mediante el gesto *Pull/Push*.
Hover & Spread/Pich		Consiste en mantener una mano horizontal paralela a la superficie por un cierto lapso de tiempo y luego acercar o alejar entre sí los dedos índice y pulgar de la otra mano.	Seleccionar un conjunto de objetos, como por ejemplo una serie de fotos, que se encuentran bajo el área del gesto *Hover* y que las mismas sean maximizadas/minimizadas mediante el gesto *Spread/Pinch*.
Point & Spread/Pich		Consiste en apuntar con un dedo de un mano un objetivo posiblemente distante y luego acercar o alejar entre sí los dedos índice y pulgar de la otra mano.	Seleccionar un objeto específico, como por ejemplo una foto, posiblemente distante, mediante el gesto *Point* y que la misma sea maximizada/minimizada mediante el gesto *Spread/Pinch*.

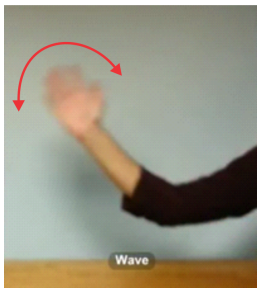
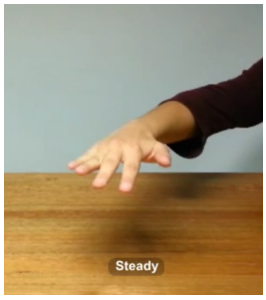
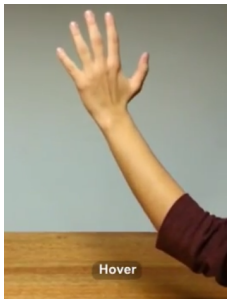
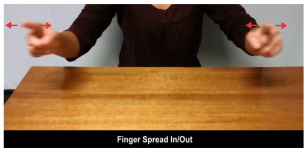
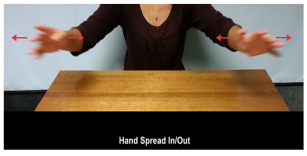
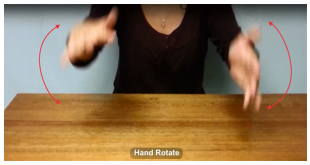
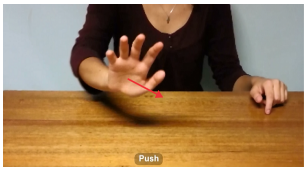
Two-finger Long Press & Slide		Consiste en presionar con dos dedos de una mano un objeto por un cierto lapso de tiempo, y luego desplazar la otra mano de izquierda a derecha o viceversa.	En un modo escena que afecte a todos los elementos mediante el gesto *Two-finger Long Press* y que se pongan a rotar todos los objetos, por ejemplo una serie de fotos, mediante el gesto *Slide*.
Fingers Circle Forward		Consiste en girar los dedos índice de cada mano en sentido horario.	Re hacer la última acción realizada mediante el gesto *Fingers Circle Forward*.
Fingers Circle Backward		Consiste en girar los dedos índice de cada mano en sentido anti-horario.	Deshacer la última acción realizada *Fingers Circle Backward*.
Long Press & Swipe Up		Consiste en presionar con un dedo de una mano un objeto por un cierto lapso de tiempo y luego deslizar hacia arriba un dedo de la otra mano.	Seleccionar un objeto específico, como por ejemplo una nota rápida (*post it*), mediante el gesto *Long Press* y que el mismo sea movida a una superficie que se encuentra en la dirección del gesto *Swipe Up*.
Hover & Swipe Up		Consiste en mantener una mano paralela a la superficie por un cierto lapso de tiempo y luego deslizar hacia arriba un dedo de la otra mano.	Seleccionar un conjunto de objetos, como por ejemplo un conjunto de notas rápidas (*post its*) ,que se encuentran bajo el área del gesto *Hover* y que los mismos sean movidos a una superficie que se encuentra en la dirección del gesto *Swipe Up.*

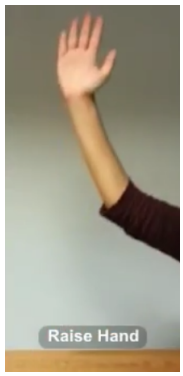
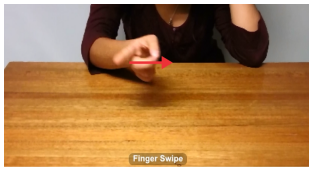
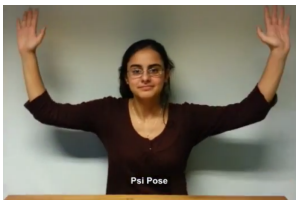
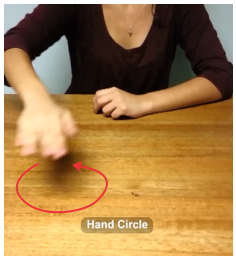
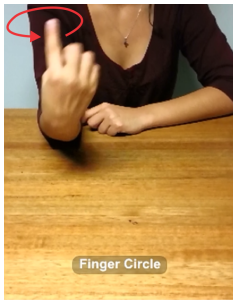
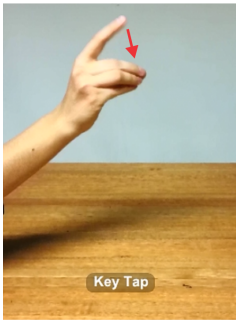
Point & Swipe Down		Consiste en apuntar con un dedo de un mano un objetivo posiblemente distante y luego deslizar hacia abajo un dedo de la otra mano.	Seleccionar un objeto específico, como por ejemplo un diagrama, posiblemente distante mediante el gesto *Point* y que el mismo sea movido a una superficie que se encuentra en la dirección del gesto *Swipe Down*.
Hover & Swipe Down		Consiste en mantener una mano paralela a una superficie posiblemente distante por un cierto lapso de tiempo y luego deslizar hacia abajo un dedo de la otra mano.	Seleccionar un conjunto de objetos, como por ejemplo un conjunto de diagramas, posiblemente distantes que se encuentren bajo el área del gesto *Hover* y que los mismos sean movidos a una superficie que se encuentra en la dirección del gesto *Swipe Down*.
Fist Grab		Consiste en mantener cerrada una mano por un cierto lapso de tiempo.	Mantener seleccionados un conjunto de objetos, como por ejemplo una serie de fotos que se encuentran bajo el área del gesto *Fist Grab.*
Fist Drop		Consiste en abrir una mano luego de que la misma estuviera cerrada.	Soltar un conjunto de objetos, como por ejemplo una serie de fotos que se encontraban seleccionadas mediante el gesto *Fist Drop.*
Long Press & Focus Hover		Consiste en mantener una mano paralela a una superficie posiblemente distante y luego presionar con un dedo de la otra mano un objeto por un lapso de tiempo prolongado.	Seleccionar un objeto específico, como por ejemplo una agenda de reuniones, mediante el gesto *Long Press* y que el mismo sea mostrado temporalmente, mientras se mantiene el *Long Press*, en una superficie que se encuentra en la dirección paralela del gesto *Focus Hover*.

Hover & Focus Hover		Consiste en mantener una mano paralela a la superficie principal por un cierto lapso de tiempo y luego mantener la otra mano paralela a una superficie posiblemente distante.	Seleccionar un conjunto de objetos, como por ejemplo una serie de notas rápidas (*post its*) , que se encuentran bajo el área del gesto *Hover* y que las mismas sean mostradas temporalmente, mientras se mantiene el *Hover*, en una superficie que se encuentra en la dirección paralela del gesto *Focus Hover*.
Long Press & Finger Scissors		Consiste en presionar con un dedo de una mano un objeto por un cierto lapso de tiempo y luego juntar y alejar los dedos índice y mayor de la otra mano en posición vertical (similar al movimiento de una tijera).	Seleccionar un objeto específico, como por ejemplo una nota rápida (*post it*), mediante el gesto *Long Press* y que la misma sea recortada de la superficie mediante el gesto *Finger Scissors*.
Table Hover & Finger Scissors		Consiste en mantener una mano paralela a la superficie por un cierto lapso de tiempo y luego juntar y alejar los dedos índice y mayor de la otra mano en posición vertical (similar al movimiento de una tijera).	Seleccionar un conjunto de objetos, como por ejemplo una serie de notas rápidas (*post its*), que se encuentran bajo el área del gesto *Hover* y que las mismas sean recortadas de la superficie mediante el gesto *Finger Scissors*.
Point & Finger Scissors		Consiste en apuntar con un dedo de una mano un objetivo posiblemente distante y luego juntar y alejar los dedos índice y mayor de la otra mano en posición vertical (similar al movimiento de una tijera).	Seleccionar un objeto específico, como por ejemplo una nota rápida (*post it*), posiblemente distante mediante el gesto *Point* y que la misma sea recortada de la superficie mediante el gesto *Finger Scissors*.
Wall Hover & Finger Scissors		Consiste en mantener una mano paralela a una superficie posiblemente distante y luego juntar y alejar los dedos índice y mayor de la otra mano en posición vertical (similar al movimiento de una tijera).	Seleccionar un conjunto de objetos, como por ejemplo una serie de notas rápidas (*post its*), posiblemente distantes, que se encuentran bajo el área del gesto *Hover* y que los mismos sean recortados de la superficie mediante el gesto *Finger Scissors*.

Long Press & Finger Snapshot		Consiste en presionar con un dedo de una mano un objeto por un cierto lapso de tiempo y luego flexionar el dedo índice de la otra mano en dirección del dedo pulgar (como si se estuviera jalando un gatillo).	Seleccionar un objeto específico, como por ejemplo una agenda de reunión del día, mediante el gesto *Long Press* y que la misma sea copiada tantas veces en la superficie como ejecuciones del gesto *Finger Snapshot*.
Table Hover & Finger Snapshot		Consiste en mantener una mano paralela a la superficie por un cierto lapso de tiempo y luego flexionar el dedo índice de la otra mano en dirección del dedo pulgar (como si se estuviera jalando un gatillo).	Seleccionar un conjunto de objetos, como por ejemplo una serie de tarjetas de contactos, que se encuentran bajo el área del gesto *Hover* y que las mismas sean copiados tantas veces en la superficie como ejecuciones del gesto *Finger Snapshot*.
Point & Finger Snapshot		Consiste en apuntar con un dedo de una mano un objetivo posiblemente distante y luego flexionar el dedo índice de la otra mano en dirección del dedo pulgar (como si se estuviera jalando un gatillo).	Seleccionar un objeto específico, como por ejemplo una agenda de reunión del día, posiblemente distante, mediante el gesto *Point* y que la misma sea copiada tantas veces en la superficie como ejecuciones del gesto *Finger Snapshot*.
Wall Hover & Finger Snapshot		Consiste en mantener una mano paralela a una superficie posiblemente distante y luego flexionar el dedo índice de la otra mano en dirección del dedo pulgar (como si se estuviera jalando un gatillo).	Seleccionar un conjunto de objetos, como por ejemplo una serie de tarjetas de contactos, posiblemente distantes, que se encuentren bajo el área del gesto *Hover* y que las mismas sean copiadas tantas veces en la superficie como ejecuciones del gesto *Finger Snapshot*.
Two-handed Five-finger Press & Pull		Consiste en posicionar ambas manos sobre la superficie y luego desplazarlas de abajo hacia arriba.	En un modo escena que afecte a todos los elementos, por ejemplo para pasar niveles de capas renderizando distintas imágenes sobre la superficie, mediante el gesto *Two-handed Fiver-finger* subiendo de nivel mediante el gesto *Pull*.

Two-handed Five-finger Press & Push		Consiste en posicionar ambas manos paralelas a la superficie y desplazarlas de arriba hacia abajo hasta establecer contacto con la misma.	En un modo escena que afecte a todos los elementos, por ejemplo para pasar niveles de capas renderizando distintas imágenes sobre la superficie, mediante el gesto *Two-handed Fiver-finger* bajando de nivel mediante el gesto *Push*.
Two-handed Pull		Consiste en posicionar ambas manos paralelas a una superficie posiblemente distante y luego desplazarlas de adentro hacia afuera (en dirección al usuario).	En un modo escena que afecte a todos los elementos, por ejemplo para pasar niveles de capas, posiblemente distantes, renderizando distintas imágenes sobre la superficie, mediante el gesto *Two-handed* subiendo de nivel mediante el gesto *Pull*.
Two-handed Push		Consiste en posicionar ambas manos paralelas a una superficie posiblemente distante y luego desplazarlas de afuera hacia adentro (en dirección a la superficie).	En un modo escena que afecte a todos los elementos, por ejemplo para pasar niveles de capas, posiblemente distantes, renderizando distintas imágenes sobre la superficie, mediante el gesto *Two-handed* bajando de nivel mediante el gesto *Push*.
Handed Swipe		Consiste en posicionar una mano en posición vertical y desplazarla de izquierda a derecha o viceversa.	En un modo escena que afecte a todos los elementos donde se haga pasar los objetos de uno, como por ejemplo pasaje de slides de una presentación, mediante el gesto Slide.
Handed Swipe Out		Consiste en posicionar una mano en posición horizontal y desplazarla de adentro hacia afuera (en dirección opuesta al usuario).	Enviar a la papelera de reciclaje un objeto seleccionado previamente, por ejemplo un archivo de texto, mediante el gesto de Handed Swipe Out.
Handed Swipe In		Consiste en posicionar una mano en posición horizontal y desplazarla de afuera hacia adentro (en dirección al usuario).	Restaurar la papelera de reciclaje, mediante el gesto de Handed Swipe In.

Wave		Consiste en mantener una mano en posición vertical con el brazo levantado y moverla de un lado al otro (como si se saludara).	En un modo escena donde se determine que se active la interacción mediante el gesto *Wave*.
Steady		Consiste en mantener una mano paralela a la superficie por un cierto lapso de tiempo.	Accionar un conjunto de objetos, por ejemplo abrir álbums de fotos que se encontraban cerrados, mostrando así una galería de fotos de cada uno en la superficie mediante el gesto *Steady.*
Hover		Consiste en mantener una mano paralela a una superficie posiblemente distante por un cierto lapso de tiempo.	Accionar un conjunto de objetos posiblemente distantes, por ejemplo abrir álbums de fotos que se encontraban cerrados, mostrando así una galería de fotos de cada uno en la superficie mediante el gesto *Hover*.
Finger Spread In/Out		Consiste en mantener el dedo índice de cada mano estirados mientras se acercan y se alejan entre sí.	Incrementar/decrementar un atributo, por ejemplo en un reproductor de música subir/bajar el volumen de la reproducción, mediante el gesto *Finger Spread Out/In*.
Hand Spread In/Out		Consiste en mantener ambas manos abiertas mientras se acercan y se alejan entre sí.	En un modo escena que afecte a todos los elementos, por ejemplo para maximizar/minimizar el tamaño de imágenes, posiblemente distantes, sobre la superficie, mediante el gesto *Hand Spread In/Out*.
Hand Rotate		Consiste en mantener ambas manos abiertas en posición vertical mientras se giran en mueven formando un círculo.	En un modo escena que afecte a todos los elementos, posiblemente distantes, se pongan a rotar todos los objetos, por ejemplo imágenes de objetos en 3D, mediante el gesto *Hand Rotate* .
Push		Consiste en mantener una mano abierta en posición vertical mientras se desplaza de afuera hacia adentro (en dirección opuesta al usuario).	Pasar a la siguiente opción, por ejemplo en una paleta de colores mover el selector al siguiente color, mediante el gesto *Push*.

Raise Hand		Consiste en mantener la mano abierta con el brazo levantado por un cierto lapso de tiempo.	En un modo escena donde se determine que se desactive la interacción mediante el gesto *Raise Hand*.
Finger Swipe		Consiste en mantener el dedo índice de una mano estirado y luego deslizarlo hacia la izquierda o la derecho.	Pasar a una siguiente acción, por ejemplo en un reproductor de música pasar a la siguiente pista, mediante el gesto *Finger Swipe*.
Psi Pose		Consiste en mantener ambas manos abiertas con los brazo levantados por sobre la altura de los hombros por un cierto lapso de tiempo.	En un modo escena que afecte a todos los elementos, parar de girar todos los objetos de la escena mediante el gesto *Psi Pose*.
Hand Circle		Consiste en mantener una mano paralela a la superficie mientras se mueve formando un círculo.	En un modo escena que afecte a todos los elementos, girar todos los objetos de la escena en sentido horario mediante el gesto *Hand Circle*.
Vertical Finger Circle		Consiste en mantener el dedo índice de una mano en posición vertical mientras se mueve formando un círculo.	En un modo escena que afecte a todos los elementos, girar todos los objetos de la escena en sentido horario mediante el gesto *Fingers Circle*.
Key Tap		Consiste en mover el dedo índice de una mano de arriba hacia abajo (como si se presionara una tecla).	Pausar/Reactivar una acción, por ejemplo en un reproductor de música pausar/reproducir una pista de audio, mediante el gesto *Key Tap* pausando en la primera ejecución y reproduciendo en la siguiente ejecución del gesto.

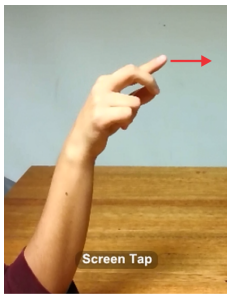
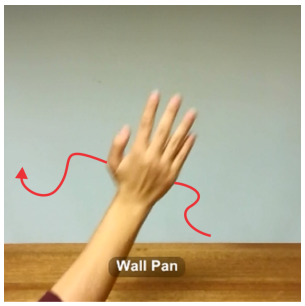
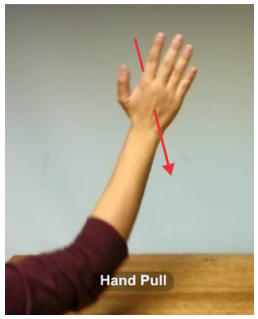
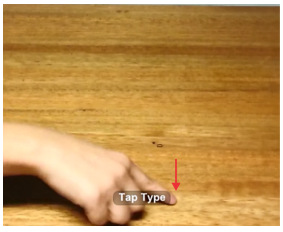
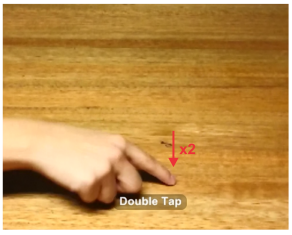
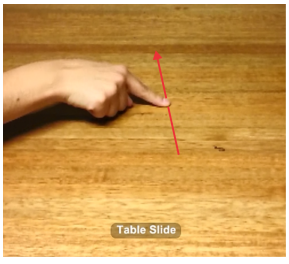
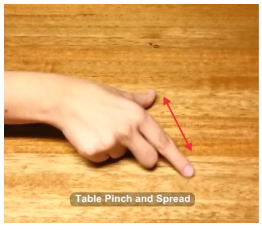
Screen Tap		Consiste en mover el dedo índice de una mano de adentro hacia afuera (como si se tocara una pantalla vertical).	Presionar un objeto, posiblemente distante como por ejemplo un botón de la superficie mediante el gesto *Screen Tap*.
Wall Pan		Consiste en mantener una mano paralela a una superficie posiblemente distante durante cierto lapso de tiempo y luego moverla libremente en cualquier dirección.	Desplazar un conjunto de objetos posiblemente distantes, como por ejemplo esquemas realizados, a lo largo de la superficie para dejarlas en otra posición mediante el gesto *Wall Pan*.
Hand Pull		Consiste en mantener una mano abierta en posición vertical mientras se desplaza de adentro hacia afuera (en dirección al usuario).	Pasar a la anterior opción, por ejemplo en una paleta de colores mover el selector al color anterior, mediante el gesto *Hand Pull*.
Tap Type		Consiste en mover el dedo índice de una mano de arriba hacia abajo hasta establecer contacto con la superficie.	Presionar un objeto, como por ejemplo un botón de la superficie mediante el gesto *Tap Type*.
Double Tap		Consiste en mover el dedo índice de una mano de arriba hacia abajo hasta establecer contacto con la superficie dos veces consecutivas.	Accionar un objeto, por ejemplo, abrir un álbum de fotos que se encontraba cerrado, mostrando así una galería de fotos en la superficie mediante el gesto *Double Tap*.
Table Slide		Consiste en mantener el dedo índice de una mano extendido sobre la superficie y desplazarlo en cierta dirección.	Deslizar un objeto de la superficie, como por ejemplo pasar fotos en un álbum de fotos, de la superficie mediante el gesto *Table Slide*.
Table Pinch/Spread		Consiste en acercar o alejar entre sí los dedos índice y pulgar de una mano, mientras se establece contacto con la superficie.	Minimizar/Maximizar un objeto, como por ejemplo una foto, (modificando así su tamaño), de la superficie mediante el gesto *Table Pich/Spread*.

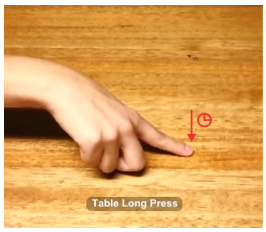
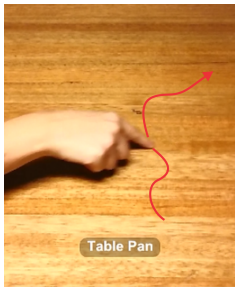
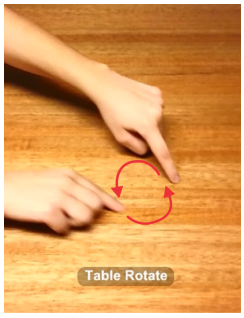
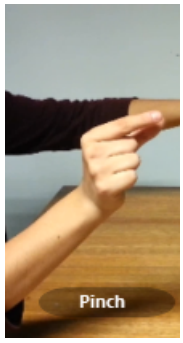
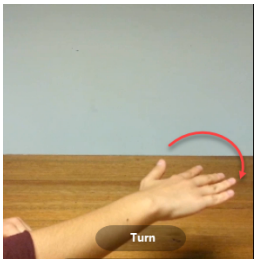
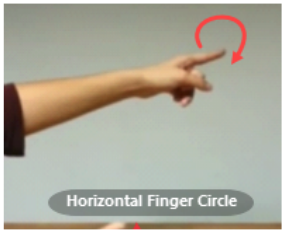
Table Long Press		Consiste en mover el dedo índice de una mano de arriba hacia abajo hasta establecer contacto con la superficie y mantenerlo en contacto por un lapso de tiempo predeterminado.	Desplegar opciones de un objeto, como por ejemplo un menú de opciones para editar una foto de la superficie mediante el gesto *Table Long Press*.
Table Pan		Consiste en mantener el dedo índice de una mano en contacto con la superficie mientras se mueve libremente en cualquier dirección.	Desplazar un objeto, como por ejemplo una foto a lo largo de la superficie para dejarla en otra posición mediante el gesto *Table Pan*.
Table Rotate		Consiste en mantener ambos dedos índice en contacto con la superficie mientras se mueven formando un círculo.	Rotar una objeto, como por ejemplo una foto seleccionado sus puntas mediante el gesto *Table Rotate*.
Pinch		Consiste en mantener juntos los dedos índice y pulgar como sosteniendo algo entre ambos dedos.	Levantar un objeto desde el plano, como por ejemplo una pelota mediante el gesto *Pinch*.
Turn		Consiste en mantener un mano abierta palma abajo y luego darla vuelta hasta que queda la palma para arriba.	Abrir y cerrar una opción, por ejemplo abrir una ventana de ícono con opciones mediante el primer gesto de *Turn* y cerrarla al volver hacer el gesto *Turn*.
Horizontal Finger Circle		Consiste en mantener el dedo índice de una mano en posición horizontal mientras se mueve formando un círculo.	En un modo escena que afecte a todos los elementos, girar todos los objetos de la escena en sentido horario mediante el gesto *Horizontal Fingers Circle*.

Tabla 13: Banco teórico de gestos definidos

Anexo I: Bibliotecas de interacción multitáctil

Para llevar a cabo la necesidad planteada sobre la realización de interacción multitáctil se hicieron estudios entre diferentes *frameworks* ya existentes a modo de evaluar la viabilidad de su uso como parte de la solución a este problema. Se enumera a continuación, los distintos *frameworks* analizados:

I.1. dSensingNI

dSensingNI es un *framework* que brinda soporte de interacción multitáctil sobre cualquier tipo de superficie, a gestos espaciales y a interacción tangible con cualquier tipo de objeto haciendo uso de una cámara de profundidad.

Este *framework* es capaz de detectar y hacer seguimiento de *touchs points* sobre cualquier tipo de superficie permitiendo incluso tener interacción sobre superficies curvas y sobre cualquier tipo de objeto. Por otro lado, es capaz de detectar y hacer seguimiento de brazos, señaladores, dedos y palmas que se encuentren dentro de la zona de interacción. A su vez, permite detectar y hacer seguimiento de objetos tangibles en tiempo real como así también, detectar gestos naturales que son habituales de realizarse diariamente. Inicialmente, *dSensingNI* fue diseñado para su utilización en superficies del estilo semejante a una mesa pero es también aplicable a superficies verticales como son paredes interactivas, pizarras, etc. utilizando el protocolo *TUIO* para transferir la información detectada desde el *framework* a la aplicación cliente en cuestión.

dSensingNI presenta características valorables para ser incluido como parte de este trabajo ya que permite resolver el problema de gestualidad multitáctil entre los otros aportes que brinda y que podrían ser de utilidad. A su vez, se observa que la arquitectura provista por este *framework* es del tipo cliente-servidor la cual es la adecuada para la solución de este trabajo, como así también, la disposición en que se debe de colocar la cámara de profundidad, que da soporte a la solución ya que es colocada por sobre la superficie de interacción. Sin embargo, se observa que, para quien oficie de aplicación cliente se distribuye sólo la librería *TUIO3DLib* que se encarga de traducir todos los datos *TUIO* que envía el *framework* a eventos más abstractos (*fingerEvent*(), *handEvent*(), *etc*.), pero ésta, está solo disponible para el lenguaje *.NET*. Si bien se podría implementar una traducción propia para *C++* el principal problema es que el *framework* no está disponible públicamente como tampoco así su código, si no que se debe solicitar a demanda. Teóricamente, existe la posibilidad de adquirirlo rellenando un formulario de solicitud en la página *web* del *framework*, sin embargo, se la solicitó en dos oportunidades y no se obtuvo respuestas a ninguna de dichas solicitudes, por lo cual, no se pudo obtener el *framework* y se tuvo que descartar la posibilidad de utilizar la solución provista por *dSensingNI*.

I.2. Kinect Touch/KinectFingerTip

Kinect Touch es un *"Kinect Hack"* creado por *Robert Walter* que permite la detección de *touchs points* sobre una superficie arbitraria. Se basa en una implementación sencilla que utiliza los mapas de profundidad para determinar la detección de *touch points* sobre la superficie y la utilización de *TUIO* para la notificación de los eventos. Básicamente, utiliza un mapa fondo de base al cual se lo contrasta con un mapa actual para determinar según la cantidad de píxeles que se encuentren próximos a la superficie determina si han ocurrido un *touch point* o no.

El código es de acceso público y viene con distribuciones de *TUIO* (y otras dependencias como

oscpack) para *Linux*. Tanto *TUIO* como *oscpack* tienen sus versiones para *Windows* y *Mac* por lo cual, se pasó a bajar las versiones de *Windows* y realizar la pruebas de funcionamiento. Si bien se logró compilar la solución ésta o funcionó de forma correcta, luego de varias pruebas no fue posible lograr la detección de los *touchs points* con el código proporcionado y al no contar con una comunidad donde se pudiera consultar sobre posibles problemas detectados es que su utilización fue descartada. Luego de este *"Kinect Hack" Robert Walter* presentó otro *framewor*k denominado *"Kinect Finger Tip"* el cual es una versión aparentemente mejorada del hack *"Kinect Touch"* presentando anteriormente. Si bien pretende ser una mejora no se logró observar ningún cambio que hiciera que el código actuara de forma correcta, por lo cual, también esta versión fue descartada.

I.3. *libTISCH*

libTISCH es otro de los *framework* analizados, el cual puede ser utilizado tanto en *Linux*, *Mac* como así también en *Windows*. *libTISCH* permite la interacción multitáctil así como también, la interacción mediante la utilización de todo el cuerpo del usuario. Por otro lado, es un *framework opensource*, disponible para descargar y que utiliza como protocolo de comunicación para el envío de notificaciones a la aplicación cliente el protocolo *TUIO*.

Si bien es un *framework* posible de utilizarse como parte de la solución al presente trabajo, presenta como desventaja que para su utilización es necesario contar con la librería *libfreenect* lo que lo hace incompatible con *OpenNI* y por lo tanto, con el lineamiento escogido para el desarrollo del trabajo, ya que éste se basa en la utilización del *framework* de *OpenNI*.

I.4. *TouchLib*

TouchLib es otro *framework* analizado que permite el desarrollo de superficies multitáctiles y capaz de enviar notificaciones a la aplicación de usuario cuando ocurren eventos del tipo multitáctil haciendo uso del protocolo *TUIO*. Este *framework* está escrito con lenguaje *C++* y su código es libre y gratuito para descargar. Posee una versión para *Visual Studio* 2005 lista para descargar y compilar desde su sitio. Por otro lado, posee un buen foro de discusión y una página de *wiki* bastante completa. Si bien este *framework* podría ser de utilidad su versión asociada a *Visual Studio* 2005 la hace una versión descontinuada. A su vez, este *framework* está únicamente disponible para sistemas operativos del tipo *Windows*, lo cual, de utilizarlo, haría que la solución del presente trabajo no quedara portable violando uno de los requerimientos necesarios y entonces, por este motivo es que se ha descartado su uso.

I.5. *reacTIVision*

reacTIVision es un *framework opensource*, multiplataforma, cuyo último *release* data del 2009 por lo cual muestra que su versión está discontinuada. Este *framework* provee mecanismos para hacer un seguimiento rápido y robusto a *fiducials* montados en cualquier tipo de objeto, como así también, permite el seguimiento y detección de eventos multitáctiles. Este *framework* fue diseñado como una herramienta para el desarrollo rápido de interfaces tangibles del estilo tipo mesa y superficies multitáctiles. *reacTIVision* también utiliza el protocolo de comunicación *TUIO* para la notificación de eventos multitáctiles ocurridos, justamente en el marco del proyecto de desarrollo de este *framework* es que se dio origen al protocolo *TUIO*.

Para hacer uso de *reacTIVision* se debe de cumplir ciertos requerimientos en cuanto a cómo debe ser la superficie con la que se quiere interactuar. Específicamente, se plantea que, la superficie debe de ser de un material especial, esto es, la superficie de interacción debe ser de un material transparente y no opaco, este material debe tener a su vez, la capacidad de no reflejar la luz. Adicionalmente, se establecen restricciones en cuanto a cómo poner la cámara a utilizar, específicamente, la cámara a utilizar para analizar la escena debe estar colocada por debajo de la superficie de la mesa apuntando hacia ella. Y por último, se

plantea que debe de haber una muy buena iluminación para lograr el contraste necesario para obtener un buen seguimientos de los objetos o manos en la escena. Estos dos puntos que hacen referencia tanto a la superficie de interacción y a cómo debe estar posicionada la cámara en la escena hizo que se descarta también a este *framework*, ya que, por lo nombrado anteriormente, la superficie de interacción es muy restrictiva limitando el total de superficies sobre las cuales se puede llevar la interacción hecho que va en en contra de uno de los requerimientos planteados deseables para el presente trabajo. Debido a las restricciones recientemente mencionadas en cuanto a la superficie de interacción es que su utilización fue descartada.

I.6. *Single Kinect Touch / Ludique's Kinect Bundle* (LKB)

LKB es un aplicación que utiliza cámaras de profundidad para transformar cualquier superficie en una superficie multitáctil [77] . Este proyecto comenzó bajo el nombre de *Simple Kinect Touch* pero luego este quedó descontinuado cambiando el nombre a *Ludique's Kinect Bundle* (*LKB*). *LKB* consiste en una aplicación a la que se le debe indicar cuál es la superficie con la que se quiere interactuar y ésta retorna los eventos *touch points* que observe mediante notificaciones *TUIO* posibles de enviar a cualquier aplicación que oficie de aplicación cliente. Por otro lado, *LKB* envía notificaciones sobre los *joints* de los usuarios como también permite calibrar hasta dos cámaras para extender la zona de interacción. A su vez, *LKB* se encuentra disponible para *Windows*, *Linux* como así también, para *Mac*.

Si bien esta aplicación podría incluirse en la solución al presente trabajo, aunque sería una aplicación externa que debe correr en conjunto con las aplicaciones que brinden la solución al *framework* a desarrollar, no mostró una muy buena *performance*. Por otro lado, para determinar la superficie de interacción la aplicación está acotada a un cantidad máxima de dos sensores no permitiendo la posibilidad de agregar cuantos sensores se deseen para generar una superficie de interacción aún más amplia; por estas razones es que esta solución también fue descartada.

Anexo J: Soluciones a la interferencia

Existen distintas técnicas para la resolución del problema de interferencia causado por el solapamiento de los rayos de múltiples sensores *Microsoft Kinect*; éstas técnicas son enumeradas a continuación[127]:

1. *Avoid the Interference*: esta técnica hace referencia al hecho de evitar la interferencia. En muchas ocasiones, es posible posicionar los sensores de forma tal que la superposición sea nula o que tienda a ser prácticamente baja. Por ejemplo, en la imagen de la *Figura 158* para representar de forma tridimensional a un usuario, es posible posicionar las cámaras de tal forma que la interferencia sea prácticamente nula.

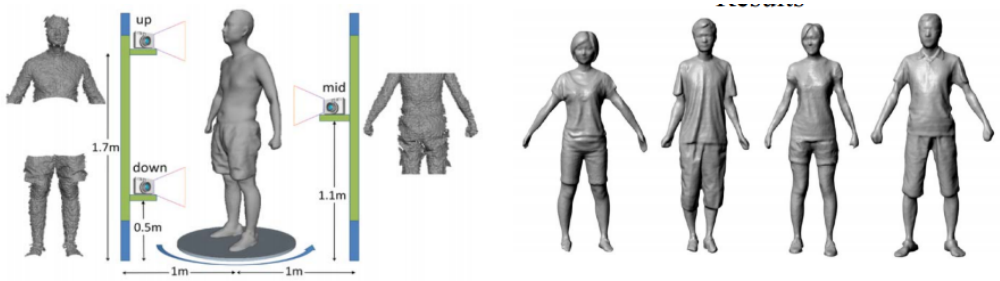


Figura 158: Técnica *Avoid the Interference* [136]

Si bien esta es una solución sencilla, no se adapta a todos los tipos de aplicaciones. Justamente para el desarrollo del presente trabajo esta no es una solución viable, ya que, el sensor que se encarga de seguir a los usuarios debe ver la zona de interacción que ve el sensor encargado de la detección de gestos multitáctiles para poder realizar así, la correspondencia entre ellos.

2. *Time Multiplexing:* esta técnica consiste en instalar sobre los sensores una especie de ventanas que permiten pasar los rayos que son emitidos de forma sincronizada y así, evitar que ocurra interferencia. Estas ventanas pueden ser definidas como piezas de plástico como las que se muestra en la *Figura 160* siendo controladas programáticamente siguiendo una patrón serial tal como se muestra en la *Figura 159*.

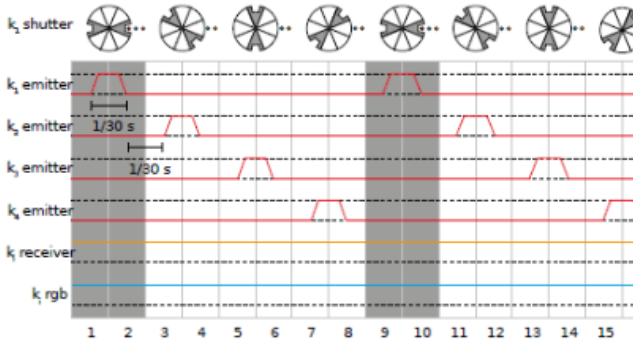


Figura 159: Sincronización de pasaje de rayos [136]

Figura 160: Implementación de ventanas [136]

Si bien es una solución sencilla, no posee buenos resultados. Lo que ocurre es una gran pérdida de *frames* asociados a los tiempos que se está tapando la emisión de los rayos y por otro lado, la sincronización está asociada al sistema que maneja las ventanas y no a nivel de sensor.

En la *Figura 161*, se puede observar la diferencia que hay en cuanto al porcentaje de pérdida de información para las dos técnicas mencionadas (*Avoid the interference y Time multiplexing*) según la posición de los sensores y el tipo de escena donde se encuentran inmersos. Como se ejemplifica en la tabla de la *Figura 161* la pérdida de información para la técnica *Avoid the Interference* es prácticamente nula mientras que para la técnica *Time Multiplexing* llega a alrededor del 20% sumada a una alta pérdida de *frames*.

	Concurrently running					Time-multiplexed		
Material vs. Setup	1	2 adjacent	2 opposite	3	4	1	2 adjacent	2 opposite
Blank room	0.1159%	0.5908%	0.4420%	0.0253%	2.8267%	1.8413%	23.2563%	4.3864%
Diffuse Carpet	0.3696%	0.9446%	0.9643%	1.6493%	2.5914%	2.1332%	22.0778%	5.1351%
Mirroring foil	3.3111%	6.0228%	5.6591%	7.8242%	10.2082%	5.9220%	25.1686%	13.2228%
Plastic Pipe	0.5351%	1.1571%	1.1257%	1.8063%	3.1328%	2.6534%	23.1102%	4.3260%
Specular Cans	0.4908%	1.2881%	1.0737%	2.1112%	3.4963%	2.4954%	21.7885%	5.4662%
fps / ms	30 / 33					30 / 33	15 / 66	15 / 66

Figura 161: Comparación entre técnicas *Avoid the Interference* y *Time Multiplexing* [136]

3. *Vibrations:* esta técnica hace referencia al hecho de agregar "vibraciones" mediante un pequeño motor sujetado al sensor, tal como se muestra en la *Figura 162,* permitiéndole así que el sensor sufra pequeñas oscilaciones.

Figura 162: Motor para técnica de *Vibrations* [136]

Mediante las vibraciones producidas es que ocurre que los patrones de los rayos proyectados sufran una especie de *"blur"* que hace que se eliminen parcialmente las intersecciones y con ello, la pérdida de información. Un ejemplo del efecto resultante de aplicar esta técnica se puede ver en la *Figura 163,* donde la *Figura 163.a* es la escena con un único sensor, la escena de la *Figura 163.b* es la escena con dos sensores sin aplicar ninguna técnica de resolución de interferencia, y por último, en la *Figura 163.c* se introduce la resolución mediante vibraciones donde se puede observar la gran mejoría en cuanto a la interferencia.

Figura 163: Aplicación técnica *Vibrations* [127]

Ésta técnica de resolución mediante la introducción de vibraciones es escalable en cuanto a la cantidad de sensores presentes en la escena que se superpongan y sin tener pérdida de *frames* como la técnica *Time Multiplexing* explicada anteriormente. La única pequeña desventaja

presente en esta técnica es que el mapa de color (mapa *RGB*) también se va a ver modificado por las vibraciones, si bien esto es un desventaja en caso de que se quiera usar dicho mapa, se puede corregir por *software* mediante un *post* proceso.

Se establece experimentalmente que el uso de la técnica de vibraciones en una escena donde se ubica un arreglo de seis sensores *Microsoft Kinect* que se solapan se logra reducir la pérdida de información de un 16,6% a un 1,4%. Por otro lado, otro dato experimental arroja que mediante el uso de un *array* de tres sensores se observa una reducción del 82,2% de pérdida de información.

4. *Hole Filling Filter:* esta técnica aplica la metodología de rellenar los "huecos" causados por la interferencia. Para esto, se aplica un algoritmo para calcular el valor a asignar en el hueco generado según los valores próximos a dicho hueco, de esta forma, lo que se logra es "suavizar" la zonas de los huecos asignándoles valores promedios calculados por los valores de los píxeles más próximos.

   La técnica consiste en que para cada píxel con valor 0 que se encuentre en un cuadrado (N) de un lado definido (R) para los cuales se establecen *"thresholds"* (Th1, Th2, Th3) se deben de seguir los siguientes pasos:

   a. Contar la cantidad de píxeles válidos en `N (Vp)`
   b. Calcular el valor promedio de los píxeles válidos `(M)`
   c. Calcular la cantidad de píxeles válidos en el perímetro de `N (Ve)`
   d. Calcular la diferencia entre el máximo y el mínimo valor de un píxel `(N) max(N) - mín (N) (D)`
   e. Corroborar los valores hallados con los *thresholds* definidos
   f. `If (D < Th1) & (Ve > Th2) & (Vp > Th3) replace 0 with M`

   Lo que se pretende es que se repita dicho algoritmo a lo largo de todo el mapa de profundidad definiendo los cuadrados R y los *thresholds* lo más pequeños posibles para suavizar la superficie. En la imagen de la izquierda de la *Figura 164* se puede observar la escena con un único sensor, en la imagen central de la *Figura 164* se puede observar la escena con dos sensores y sin aplicar ninguna técnica de resolución mientras que en la imagen de la derecha de la *Figura 164* se puede observar la escena con dos sensores y con la técnica de resolución *Hole Filling Filter*.

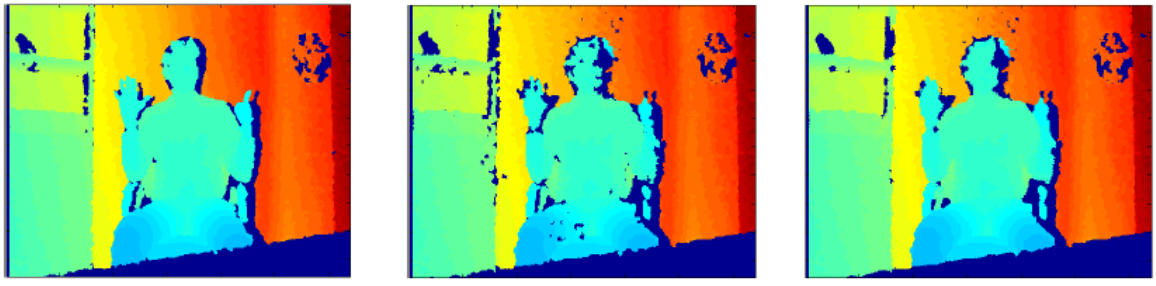


Figura 164: Aplicación técnica *Hole Filling Filter* [136]

Luego del análisis de las distintas soluciones para la problemática de la interferencia se resume lo observado en el cuadro de la *Figura 165.*

	Avoidance	Time multiplexing	Vibrations	Hole filling
Result accuracy	Excellent	Bad	Very good	Good
Handling large number of Kinects	Excellent (assuming no overlaps)	Bad	Excellent	Medium
Frame rate	30fps	30/n fps	30 fps	30fps
Side effects	None	none	Blurred color image	none
Robustness to different environments	Change Kinects placement	System setup adjustment	None	Parameters adjustments

Figura 165: Cuadro comparativo entre diferentes técnicas para resolver interferencia [136]

Para el análisis se midieron los aspectos de "precisión" donde el mejor resultado se da para la técnica de *Avoid the Interference* seguido luego por la técnica de *Vibrations*. El siguiente punto analizado es "gran cantidad de sensores", esto quiere decir, que si se montan en la escena una gran cantidad de sensores qué tan buena es la técnica para evitar la pérdida de información causada por la interferencia. Para este análisis la técnica que mejor resultados arroja es también *Avoid the Interference* pero *Vibrations* arroja el mismo nivel de resultado. Luego, el siguiente análisis se basa en *"frame rate"* donde A*void the Interference*, *Vibrations* y *Hole Filling* empatan en cuanto a que arrojan excelentes resultados. Luego, los últimos dos aspectos que se analizan son si se agrega algún efecto secundario tras la utilización de la técnica y cuánto se adapta a las distintas situaciones. Se puede observar que, únicamente la técnica de *Vibrations* posee un efecto secundario causando de efecto *blur* en el mapa de color y por ende la imagen *RGB* se va a ver distorsionada. En cuanto a lo segundo la técnica *Avoid the interference*, está ligada a la posición en que los dispositivos han de colocarse, la técnica de *Time Multiplexing* queda determinada por los parámetros del sistema que mueve las ventanas mientras que *Holding Filling* queda ligado a los parámetros de ajuste.

Anexo K: Configuración del entorno de desarrollo

Como ya se ha mencionado, para la realización de este trabajo se han utilizado diferentes tecnologías como son *OpenNI, NITE, openFrameworks, OpenCV, GRT, Eigen, Leap SDK* y *Boost* en el entorno de desarrollo *Visual Studio 2012*. Se describirá a continuación entonces, el conjunto de pasos que se deben de seguir a modo de configurar cada uno de estos *frameworks* dentro del entorno de desarrollo mencionado.

Lo más simple para dejar configurado el compilador y *linker* en *Visual Studio* es utilizar lo que se denominan hojas de propiedades, las cuales permiten crear paquetes de reglas para los diferentes modos de compilación, sea *Debug* o *Release*, que definen las dependencias y otros elementos referentes a la configuración del proyecto. La ventaja de este mecanismo es que luego estas hojas de propiedades se pueden referenciar y reutilizar desde otros proyectos para dejar así configuradas las dependencias y todos los demás elementos que se hayan incluído, habiendo configurado todo una única vez, siempre y cuando se tengan en cuenta algunas consideraciones, como por ejemplo, la de no utilizar rutas relativas en las definiciones sino, utilizar siempre variables de entorno previamente definidas, entre otras que se detallaran a continuación.

Las definiciones de hojas de propiedades se realizan en la pestaña *Property Manager* del *Visual Studio*. En esta pestaña, se listan los diferentes conjuntos de reglas que existen actualmente para cada uno de los modos de compilación, *Debug* y *Release*. De esta forma, cada vez que se desee configurar en un nuevo proyecto un *framework* o librería compleja (muchas dependencias) previamente configuradas, simplemente se debe agregar la hoja de propiedades correspondiente al conjunto de hojas de propiedades para el(los) modo(s) de compilación a utilizar. Esto se realiza con la opción *Add Existing Property Sheet* del *Property Manager* y permite agregar al nuevo proyecto un conjunto de paquetes de reglas que se haya definido previamente. Debido a ello, es recomendable tener un respaldo de estos paquetes de reglas a medida que se vayan definiendo, ya que, una vez que se definen quedan creadas dentro de la carpeta donde está alojado el proyecto en *Visual Studio* en el que fueron definidas (ficheros con extensión *.props*) y por lo tanto, ligadas a ese proyecto específico, lo cual implica que si se elimina el proyecto también se eliminarán las diferentes hojas de propiedades creadas.

A continuación se detalla el proceso de instalación y configuración para cada uno de los *frameworks* y librerías utilizadas, teniendo presente que las configuraciones se realizarán en base al uso de hojas de propiedades, resultando entonces, en una hoja de propiedades lo suficientemente genérica como para que pueda ser incluída en otros proyectos sin ningún tipo de inconvenientes.

K.1. *NiTE* 2.0

K.1.1 Instalar *NiTE* 2.0

1. Descargar e instalar los binarios de *NiTE 2.0*, preferentemente en su versión estable en su edición para desarrollo. Eventualmente se puede modificar la ruta en la que se instalará *NiTE 2.0* moviéndolo a la carpeta deseada.
2. Descargar e instalar el *driver* de *Microsoft Kinect*. *NiTE* se comunica con este módulo y este último es quien se encarga de hablar con el sensor *Microsoft Kinect* en su "idioma *hardware*".
3. Comprobar que se haya instalado correctamente el *driver* verificando en el administrador de dispositivos que se tengan los sensores *Kinect Camera*, *Kinect Audio* y *Kinect Motor*.

4. Una vez que se escogió y se movieron los datos a la ubicación elegida, para simplificar y eliminar las dependencias de la ubicación en la que esté la librería, se debe agregar una variable de entorno llamada *NITE2_INCLUDE* y asociar el valor según la ruta en la cual quedaron alojados los archivo de encabezado de *NiTE 2.0*. El valor de esta variable sería por ejemplo: *D:\FIng\Proyecto de grado\Programas\NiTE2\Include.*
5. Análogamente al paso anterior se debe crear otra variable de entorno llamada *NITE2_LIB* y asociar el valor según la ruta en la cual quedaron alojadas las librerías de *NiTE 2.0*. El valor de esta variable sería por ejemplo: *D:\FIng\Proyecto de grado\Programas\NiTE2\Lib.*

K.1.2 Configurar *NiTE* 2.0 en VS2012

1. Crear una solución nueva o abrir una existente a la que se le quiera incorporar NiTE 2.0.
2. Ir a la pestaña *Property Manager* donde se listan las diferentes reglas que hay para cada uno de los modos de compilación *Debug* y *Release*. Agregar un nuevo conjunto de reglas al modo *Debug* con nombre *NiTE_Debug* y configurar lo siguiente:
    a. Bajo *C/C++ ‣ General*, agregar en *Additional Include Directories* el valor *$(NITE2_INCLUDE)*.
    b. Bajo *Linker ‣ General*, agregar en *Additional Library Directories* el valor $(NITE2_LIB).
    c. Bajo *Linker ‣ Input*, agregar en *Additional Dependencies* la dependencia *NiTE2.lib*.
    d. Aplicar las modificaciones y guardar el paquete de reglas.
3. Realizar los mismos pasos pero ahora agregando una nueva regla al paquete de reglas correspondiente a *Release*. Se debe tener en cuenta que en este caso las librerías que se indican en *Linker ‣ Input* serán las mismas que para el caso anterior pero ahora deberán ser en su versión *release*.
4. Conectar el sensor *Microsoft Kinect* y verificar que funcionen correctamente los ejemplos que vienen con *NiTE 2.0*; algunos ejemplos recomendados son *Sample-PointViewer*, *Sample-Players* y *Sample-SingleControl*.
5. Compilar la aplicación que hace uso de código *NiTE 2.0* y ejecutar.

K.2. *OpenNI* 2.0

K.2.1. Instalar *OpenNI* 2.0

1. Descargar e instalar los binarios de *OpenNI 2.0*, preferentemente en su versión estable en su edición para desarrollo. Eventualmente se puede modificar la ruta en la que se instalará *OpenNI 2.0* moviéndolo a la carpeta deseada.
2. Una vez que se escogió y movieron los datos a la ubicación elegida, para simplificar y eliminar las dependencias de la ubicación en la que esté la librería, se debe agregar una variable de entorno llamada *OPENNI2_INCLUDE* y asociar el valor según la ruta en la cual quedaron alojados los archivo de encabezado de *OpenNI 2.0*. El valor de esta variable sería por ejemplo: *D:\FIng\Proyecto de grado\Programas\OpenNI2\Include.*
3. Análogamente al paso anterior se debe crear otra variable de entorno llamada *OPENNI2_LIB* y asociar el valor según la ruta en la cual quedaron alojadas las librerías de *OpenNI 2.0*. El valor de esta variable sería por ejemplo: *D:\FIng\Proyecto de grado\Programas\OpenNI2\Lib*.

Cabe mencionar que la secuencia de pasos anterior aplica solamente para el sensor *Kinect for XBox360*. Luego *Microsoft* lanzó *Kinect for Windows*, el cual agrega algunas características interesantes como el *near mode*, dotado de reconocimiento a partir de los 30 cm. aproximadamente. En principio, el sensor no era compatible con *OpenNI*, pero luego la comunidad de *OpenNI* desarrolló un puente

kinect-mssdk-openni [55] que permite utilizar *Kinect for Windows* sobre *OpenNI* incluyendo *hand-tracking* y *gesture recogition* con *NiTE*. En este caso entonces, se deberá agregar un paso adicional a la secuencia para instalar dicho complemento.

K.2.2. Configurar *OpenNI* 2.0 en VS2012

1. Crear una solución nueva o abrir una existente a la que se le quiera incorporar *OpenNI 2.0*.
2. Ir a la pestaña *Property Manager* donde se listan las diferentes reglas que hay para cada uno de los modos de compilación *Debug* y *Release.* Agregar un nuevo conjunto de reglas al modo *Debug* con nombre *OpenNI2_Debug* y configurar lo siguiente:

a. Bajo *C/C++ ‣ General*, agregar en *Additional Include Directories* el valor *$(OPENNI2_INCLUDE)*.
b. Bajo *Linker ‣ General*, agregar en *Additional Library Directories* el valor *$(OPENNI2_LIB)*.
c. Bajo *Linker ‣ Input*, agregar en *Additional Dependencies* la dependencia *openNI2.lib*.
d. Aplicar las modificaciones y guardar el paquete de reglas.

3. Realizar los mismos pasos pero ahora agregando una nueva regla al paquete de reglas correspondiente a *Release*. Se debe tener en cuenta que en este caso las librerías que se indican en *Linker ‣ Input* serán las mismas que para el caso anterior pero ahora deberán ser en su versión *release*.
4. Conectar el sensor *Microsoft Kinect* y verificar que funcionen correctamente los ejemplos que vienen con *OpenNI 2.0*; algunos ejemplos recomendados son *NiSimpleRead* y *NiViewer*.
5. Compilar la aplicación que hace uso de código *OpenNI 2.0* y ejecutar.

K.3. *openFrameworks* 0.8.0

K.3.1. Instalar *openFrameworks* 0.8.0

1. Descargar el último *release* 0.8.0 de *openFrameworks*.
2. Extraer los contenidos del fichero descargado en el lugar donde se quiera tener instalado *openFrameworks* 0.8.0 en el sistema.
3. Una vez que se escogió y movieron los datos a la ubicación elegida, para simplificar y eliminar las dependencias de la ubicación en la que esté la librería, se debe agregar una variable de entorno llamada *OPENFRAMEWORKS_INCLUDE* y asociar el valor según la ruta en la cual quedaron alojados los archivo de encabezado de *openFrameworks* 0.8.0. El valor de esta variable sería por ejemplo: *D:\FIng\Proyecto de grado\Programas\of_v0.8.4_vs_release\libs*.
4. Análogamente al paso anterior se debe crear otra variable de entorno llamada *OPENFRAMEWORKS_LIB* y asociar el valor según la ruta en la cual quedaron alojadas las librerías de *openFrameworks* 0.8.0. El valor de esta variable sería por ejemplo: *D:\FIng\Proyecto de grado\Programas\of_v0.8.4_vs_release\libs*.
5. Compilar *openFrameworks* 0.8.0 para generar los binarios (*openFrameworksLib.lib* y *openFrameworksLib_debug.lib*) indicando *Static Library (.lib)* en *Properties ‣ General ‣ Configuration Type* para que se genere un fichero del tipo *lib* y no una *dll*. *openFrameworks* no distribuye binarios sino sólo el código junto con puntos de partida para compilarlo en diferentes *IDEs*. En este caso, el que se debe bajar es el correspondiente a *VS2012*, ya que viene con una solución inicial que se puede encontrar en *$(OPENFRAMEWORKS_LIB)\openFrameworksCompiled\project\vs*, que viene con todo configurado como para compilar *openFrameworks* 0.8.0 sin mayores inconvenientes. Lo que se debe hacer entonces, es abrir la solución y compilar, generando un *.lib* por cada modo de compilación, que se deberá referenciar en cualquier proyecto donde se quiera utilizar *openFrameworks* 0.8.0.

K.3.2. Configurar *openFrameworks* 0.8.0 en VS2012

1. Crear una solución nueva o abrir una existente en la que se quiera incorporar *openFrameworks* 0.8.0.
2. Ir a la pestaña *Property Manager* donde se listan los diferentes conjuntos de reglas que hay para cada uno de los modos de compilación *Debug* y *Release*. Se debe agregar un nuevo conjunto de reglas al modo *Debug* con nombre *OF_Debug* y configurar lo siguiente:

    a. Bajo C/C++ › General, agregar en *Additional Include Directories* los valores:

```
$(OPENFRAMEWORKS_INCLUDE)\assimp\include
$(OPENFRAMEWORKS_INCLUDE)\tess2\include
$(OPENFRAMEWORKS_INCLUDE)\openFrameworks
$(OPENFRAMEWORKS_INCLUDE)\openFrameworks\3d
$(OPENFRAMEWORKS_INCLUDE)\openFrameworks\gl
$(OPENFRAMEWORKS_INCLUDE)\addons
$(OPENFRAMEWORKS_INCLUDE)\cairo\include\cairo
$(OPENFRAMEWORKS_INCLUDE)\poco\include
$(OPENFRAMEWORKS_INCLUDE)\rtAudio\include
$(OPENFRAMEWORKS_INCLUDE)\potaudio\include
$(OPENFRAMEWORKS_INCLUDE)\kiss\include
$(OPENFRAMEWORKS_INCLUDE)\glut\include
$(OPENFRAMEWORKS_INCLUDE)\glu\include
$(OPENFRAMEWORKS_INCLUDE)\glew\include
$(OPENFRAMEWORKS_INCLUDE)\videoInput\include
$(OPENFRAMEWORKS_INCLUDE)\fmodex\include
$(OPENFRAMEWORKS_INCLUDE)\freeImage\include
$(OPENFRAMEWORKS_INCLUDE)\freetype\include\freetype2
$(OPENFRAMEWORKS_INCLUDE)\freetype\include
$(OPENFRAMEWORKS_INCLUDE)\quicktime\include
$(OPENFRAMEWORKS_INCLUDE)\openFrameworks\events
$(OPENFRAMEWORKS_INCLUDE)\openFrameworks\types
$(OPENFRAMEWORKS_INCLUDE)\openFrameworks\
$(OPENFRAMEWORKS_INCLUDE)\openFrameworks\math
$(OPENFRAMEWORKS_INCLUDE)\openFrameworks\video
$(OPENFRAMEWORKS_INCLUDE)\openFrameworks\communication
$(OPENFRAMEWORKS_INCLUDE)\openFrameworks\utils
$(OPENFRAMEWORKS_INCLUDE)\openFrameworks\sound
$(OPENFRAMEWORKS_INCLUDE)\openFrameworks\app
$(OPENFRAMEWORKS_INCLUDE)\openFrameworks\graphics
$(OPENFRAMEWORKS_INCLUDE)\openFrameworks\ofxXmlSettings
$(OPENFRAMEWORKS_INCLUDE)\openFrameworks\ofxNetwork
$(OPENFRAMEWORKS_INCLUDE)\glfw\include
```

    Esto permite mantener la referencia a todos los ficheros de cabecera utilizados por el *framework*, los cuales pertenecen a varias librerías diferentes como se puede observar en las rutas de la lista anterior. Afortunadamente todas las librerías dependientes se distribuyen junto con *openFrameworks* 0.8.0.

    b. Bajo Linker › General agregar en *Additional Library Directories* los valores:

```
$(OPENFRAMEWORKS_LIB)\FreeImage\lib\vs
$(OPENFRAMEWORKS_LIB)\freetype\lib\vs
$(OPENFRAMEWORKS_LIB)\quicktime\lib\vs
$(OPENFRAMEWORKS_LIB)\cairo\lib\vs
$(OPENFRAMEWORKS_LIB)\glew\lib\vs
$(OPENFRAMEWORKS_LIB)\tess2\lib\vs
$(OPENFRAMEWORKS_LIB)\Poco\lib\vs
$(OPENFRAMEWORKS_LIB)\glu\lib\vs
$(OPENFRAMEWORKS_LIB)\videoInput\lib\vs
$(OPENFRAMEWORKS_LIB)\fmodex\lib\vs
$(OPENFRAMEWORKS_LIB)\rtAudio\lib\vs
$(OPENFRAMEWORKS_LIB)\glut\lib\vs
$(OPENFRAMEWORKS_LIB)\openFrameworksCompiled\lib\vs
$(OPENFRAMEWORKS_LIB)\glfw\lib\vs
```

c. Bajo *Linker › Input*, agregar en *Additional Dependencies* la siguiente lista de dependencias:

```
openframeworksLib.lib
cairo-static.lib
pixman-1.lib
msimg32.lib
OpenGL32.lib
GLu32.lib
kernel32.lib
setupapi.lib
Vfw32.lib
comctl32.lib
glut32.lib
rtAudioD.lib
videoInput.lib
libfreetype.lib
FreeImage.lib
qtmlClient.lib
dsound.lib
user32.lib
gdi32.lib
winspool.lib
comdlg32.lib
advapi32.lib
shell32.lib
ole32.lib
oleaut32.lib
uuid.lib
glew32s.lib
fmodex_vc.lib
glu32.lib
PocoFoundationmd.lib
PocoNetmd.lib
PocoUtilmd.lib
PocoXMLmd.lib
Ws2_32.lib
tess2.lib
glfw3.lib
```

d. Aplicar las modificaciones y guardar el paquete de reglas.

3. Realizar los mismos pasos pero ahora agregando una nueva regla al paquete de reglas correspondiente a *Release*. Tener en cuenta que en este caso las librerías que se indican en *Linker › Input* en el paso (c) serán las mismas que para el caso anterior pero ahora las correspondientes deberán ser en su versión *release*.
4. Compilar la aplicación que hace uso de código *openFrameworks* 0.8.0 y ejecutar.

K.3.3. Configurar *addons* de *openFrameworks* para uso en VS2012

Si se quiere utilizar algún *addon* de *openFrameworks* se lo debe de descargar y copiarlo a la carpeta *$(OPENFRAMEWORKS_INCLUDE)\addons*; una vez copiado a esta carpeta, existen varias opciones para poder utilizarlo las cuales son las enumeradas a continuación:

1. Generar una librería estática *(.lib)* en base al código fuente descargado que pueda luego ser agregada como dependencia en el proyecto *(Linker > Input > nombreAddon.lib)* más los ficheros *include* correspondientes.
2. Copiar todos los fuentes del *addon* a las carpetas *include* y *src* del proyecto para que formen parte directa de los fuentes del proyecto.
3. Referenciar directamente los fuentes del *addon* desde el proyecto. Para esto se suele agregar al

proyecto de *Visual Studio* una carpeta (*filter*) de nombre *addons*, que tiene como hijos todos los *addons* necesarios de *openFrameworks*. Cada *addon* tendrá subcarpetas *include* y *src*, en las cuales se deben agregar las referencias a los ficheros de cabecera y fuente respectivamente. Esto hará que se compile el código del *addon* como parte del proyecto, quedando los ficheros objeto en la carpeta que hayamos especificado para estos fines *(General > Intermediate Directory)* y siendo así accesibles al proyecto.

4. Otra alternativa es hacer que el nuevo *addon* sea compilado directamente como parte de *openFrameworks*. Para esto se debe copiar el *addon* a la carpeta *addons* de *openFrameworks* y referenciarlo correctamente desde el proyecto *Visual Studio* que permite compilar *OF* (viene incluido en el *release* de *OF* que se haya descargado). A su vez, es conveniente agregar al *ofMain.h* el *include* correspondiente al nuevo *addon*, de forma tal que al usar luego en alguno de los proyectos, solo incluyendo *ofMain* como es usual ya se puede utilizar al nuevo *addon* sin problemas.

K.4. *OpenCV*

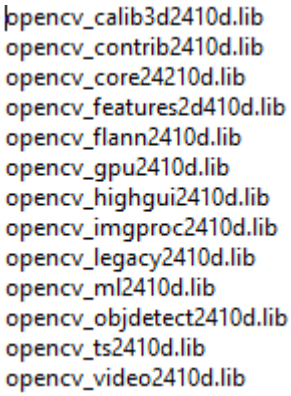
K.4.1. Instalar *OpenCV*

1. Descargar la última *release* de *OpenCV*.
2. Extraer los contenidos del fichero descargado en el lugar donde se quiera tener instalado *OpenCV* en el sistema.
3. Una vez se escogió y movieron los datos a la ubicación elegida, para simplificar y eliminar las dependencias de la ubicación en la que esté la librería, se debe agregar una variable de entorno llamada *OPENCV_INCLUDE* y asociar el valor según la ruta en la cual quedó alojado *OpenCV* y luego concatenar *build* para referenciar a los ficheros precompilados. El valor de esta variable de entorno será por ejemplo *D:\FIng\Proyecto de grado\Programas\opencv\build\include.*
4. Análogamente al paso anterior se debe crear otra variable de entorno llamada *OPENCV_LIB* y asociar el valor según la ruta en la cual quedaron alojadas las librerías de *OpenCV*. El valor de esta variable sería por ejemplo: *D:\FIng\Proyecto de grado\Programas\opencv\build\x86\vc12\lib*.
5. Agregar las librerías dinámicas de *OpenCV* al *PATH* del sistema. Para esto se debe concatenar a la variable de entorno *PATH* la ruta en la cual quedaron alojadas las librerías de *OpenCV*. El valor de esta variable sería por ejemplo: *D:\FIng\Proyecto de grado\Programas\opencv\build\x86\vc12\bin* (o *x64* si se quiere desarrollar para *64 bits*).

K.4.2. Configurar *OpenCV* en VS2012

1. Crear una solución nueva o abrir una existente para la que se quiera incorporar *OpenCV*.
2. Ir a la pestaña *Property Manager* donde se listan los diferentes conjuntos de reglas que hay para cada uno de los modos de compilación *Debug* y *Release*. Se debe agregar un nuevo conjunto de reglas al modo *Debug* con nombre *OpenCV_Debug* y configurar lo indicado en los siguientes pasos:

    a. Bajo *C/C++ › General*, agregar en *Additional Include Directories* el valor *$(OPENCV_INCLUDE)*.
    b. Bajo *Linker › General* agregar en *Additional Library Directories* el valor *$(OPENCV_LIB)*.
    c. Bajo *Linker › Input* agregar en Additional Dependencies la siguiente lista de dependencias:



Observar que el formato de las librerías es *opencv_NOMBRExxxd.lib*, donde NOMBRE es el nombre de la librería en cuestión, xxx es la versión de *OpenCV* que se haya descargado, y la d final indica que información de *Debug*.

d. Aplicar las modificaciones y guardar el paquete de reglas.

3. Hacer lo mismo pero ahora agregando una nueva regla al paquete de reglas correspondiente a *Release*. Tener en cuenta que en este caso las librerías que se indican en *Linker ‣ Input* serán las mismas que para el caso anterior pero quitando la letra d del final (que indica *debug*).
4. Compilar la aplicación que hace uso de código OpenCV y ejecutar.

K.5. *Leap SDK*

K.5.1. Instalar *Leap SDK*

1. Descargar la última *release* del *Leap SDK*.
2. Extraer los contenidos del fichero descargado en el lugar donde se quiera tener instalado *Leap SDK* en el sistema.
3. Una vez que se escogió y movieron los datos a la ubicación elegida, para simplificar y eliminar las dependencias de la ubicación en la que esté la librería, se debe agregar una variable de entorno llamada *LEAP_INCLUDE* y asociar el valor según la ruta en la cual quedaron alojados los archivo de encabezado de *Leap SDK*. El valor de esta variable sería por ejemplo: *D:\FIng\Proyecto de grado\Programas\LeapDeveloperKit_2.2.1+24116_win\LeapSDK\include.*
4. Análogamente al paso anterior se debe crear otra variable de entorno llamada *LEAP_LIB* y asociar el valor según la ruta en la cual quedaron alojadas las librerías de *Leap SDK*. El valor de esta variable sería por ejemplo: *D:\FIng\Proyecto de grado\Programas\LeapDeveloperKit_2.2.1+24116_win\LeapSDK\lib.*

K.5.2. Configurar *Leap SDK* en VS2012

1. Crear una solución nueva o abrir una existente para la que se quiera incorporar *Leap SDK*.
2. Ir a la pestaña *Property Manager* donde se listan los diferentes conjuntos de reglas que hay para cada uno de los modos de compilación *Debug* y *Release*. Se debe agregar un nuevo conjunto de reglas al modo *Debug* con nombre *LeapSDK_Debug* y configurar lo siguiente:

    a. Bajo *C/C++ ‣ General*, agregar en *Additional Include Directories* el valor *$(LEAP_INCLUDE)*, lo cual permite mantener la referencia a los ficheros de cabecera utilizados por el *framework*.
    b. Bajo *Linker ‣ General* agregar en *Additional Library Directories* el valor

$(LEAP_LIB)\x86, lo cual permite mantener la referencia a las librerías utilizadas por el *framework*.

c. Bajo *Linker › Input*, agregar en *Additional Dependencies* la dependencia *Leapd.lib*.
d. Aplicar las modificaciones y guardar el paquete de reglas.

3. Hacer lo mismo pero ahora agregando una nueva regla al paquete de reglas correspondiente a *Release*. Tener en cuenta que en este caso la librería que se indica en *Linker › Input* en el paso (c) será la misma que para el caso anterior pero ahora la correspondientes en su versión *release (Leap.lib)*.
4. Compilar la aplicación que hace uso de código *LeapSDK*.
5. Previo a ejecutar la aplicación recién compilada se debe incluir la librería dinámica *Leap.dll* o *Leapd.dll* (según se esté en modo *Release* o *Debug* respectivamente). Para esto hay varias alternativas:

    a. Copiar la librería en la misma carpeta donde se creó el ejecutable de la aplicación.
    b. Copiar la librería a la ubicación donde están todas las librerías del sistema, es decir, *Windows\system32* o *Windows\SysWOW64* según el sistema sea de 32 o 64 bits respectivamente.
    c. Agregar al *PATH* del sistema la ruta donde se encuentra la librería.

K.6. *GRT*

K.6.1. Instalar *GRT*

1. Descargar el código de repositorio de *GRT*.
2. Mover el contenido descargado en el lugar donde se quiera tener instalado *GRT* en el sistema.
3. Una vez que se escogió y movieron los datos a la ubicación elegida, para simplificar y eliminar las dependencias de la ubicación en la que esté la librería, se debe agregar una variable de entorno llamada *GRT_INCLUDE* y asociar el valor según la ruta en la cual quedaron alojados los archivo de encabezado de
4. El valor de esta variable sería por ejemplo: *D:\FIng\Proyecto de grado\Programas\GRT\trunk\GRT*.
5. Análogamente al paso anterior se debe crear otra variable de entorno llamada *GRT_LIB* y asociar el valor según la ruta en la cual quedaron alojadas las librerías de GRT. El valor de esta variable sería por ejemplo: *D:\FIng\Proyecto de grado\Programas\GRT\GRTLib\lib.*
6. Compilar *GRT* para generar los binarios (*GRTLib.lib* y *GRTLibDebug.lib*) indicando *Static Library (.lib)* en Properties *› General › Configuration Type* para que se genere un fichero del tipo *lib* y no una *dll*. Lamentablemente *GRT* no distribuye binarios ni tampoco la configuración de puntos de partida para compilarlo en diferentes *IDEs* sino sólo el código. En este caso, lo que se debe realizar entonces, es crear un proyecto en *Visual Studio* denominado *GRTLib* y colocarlo en la ubicación tal que respete de donde va a sacar la variable de entorno *GRT_LIB* la librería generada. Este proyecto contendrá dos subcarpetas "*include*" y "*src"*, en las cuales se deben agregar las referencias a los ficheros de cabecera y fuente descargados del repositorio respectivamente. Luego, lo que se debe hacer entonces es compilar, lo cual generará un *.lib* por cada modo de compilación, que se deberá referenciar en cualquier proyecto donde se quiera utilizar *GRT*.

K.6.2. Configurar *GRT* en VS2012

1. Crear una solución nueva o abrir una existente en la cual se quiera incorporar *GRT*.

2. Ir a la pestaña *Property Manager* donde se listan los diferentes conjuntos de reglas que hay para cada uno de los modos de compilación *Debug* y *Release*. Se debe agregar un nuevo conjunto de reglas al modo *Debug* con nombre *GRT_Debug* y configurar lo siguiente:

    a. Bajo *C/C++ › General*, agregar en *Additional Include Directories* los valores:

```
$(GRT_INCLUDE)\ClassificationModules\AdaBoost\WeakClassifiers
$(GRT_INCLUDE)\ClassificationModules\AdaBoost
$(GRT_INCLUDE)\ClassificationModules\ANBC
$(GRT_INCLUDE)\ClassificationModules\BAG
$(GRT_INCLUDE)\ClassificationModules\DecisionTree
$(GRT_INCLUDE)\ClassificationModules\DTW
$(GRT_INCLUDE)\ClassificationModules\GMM
$(GRT_INCLUDE)\ClassificationModules\HMM
$(GRT_INCLUDE)\ClassificationModules\KNN
$(GRT_INCLUDE)\ClassificationModules\LDA
$(GRT_INCLUDE)\ClassificationModules\MinDist
$(GRT_INCLUDE)\ClassificationModules\RandomForests
$(GRT_INCLUDE)\ClassificationModules\Softmax
$(GRT_INCLUDE)\ClassificationModules\SVM\LIBSVM
$(GRT_INCLUDE)\ClassificationModules\SVM
$(GRT_INCLUDE)\ClusteringModules\GaussianMixtureModels
$(GRT_INCLUDE)\ClusteringModules\HierarchicalClustering
$(GRT_INCLUDE)\ClusteringModules\KMeans
$(GRT_INCLUDE)\ContextModules
$(GRT_INCLUDE)\CoreAlgorithms\EvolutionaryAlgorithm
$(GRT_INCLUDE)\CoreAlgorithms\ParticleFilter
$(GRT_INCLUDE)\CoreAlgorithms\ParticleSwarmOptimization
$(GRT_INCLUDE)\DataStructures
$(GRT_INCLUDE)\FeatureExtractionModules
$(GRT_INCLUDE)\FeatureExtractionModules\FFT
$(GRT_INCLUDE)\FeatureExtractionModules\KMeansQuantizer
$(GRT_INCLUDE)\FeatureExtractionModules\MovementIndex
$(GRT_INCLUDE)\FeatureExtractionModules\MovementTrajectoryFeatures
$(GRT_INCLUDE)\FeatureExtractionModules\PCA
$(GRT_INCLUDE)\FeatureExtractionModules\TimeDomainFeatures
$(GRT_INCLUDE)\FeatureExtractionModules\TimeseriesBuffer
$(GRT_INCLUDE)\FeatureExtractionModules\ZeroCrossingCounter
$(GRT_INCLUDE)\GestureRecognitionPipeline
$(ProjectDir)\include
$(GRT_INCLUDE)\PostProcessingModules
$(GRT_INCLUDE)\PreProcessingModules
$(GRT_INCLUDE)\RegressionModules\ArtificialNeuralNetworks\MLP
$(GRT_INCLUDE)\RegressionModules\LinearRegression
$(GRT_INCLUDE)\RegressionModules\LogisticRegression
$(GRT_INCLUDE)\RegressionModules\MultidimensionalRegression
$(GRT_INCLUDE)\Util
$(GRT_INCLUDE)
```

    Esto permite mantener la referencia a todos los ficheros de cabecera utilizados por el *framework*, los cuales pertenecen a varias librerías diferentes como se puede observar en las rutas de la lista anterior. Afortunadamente todas las librerías dependientes se distribuyen junto con *GRT*.

    b. *Bajo Linker › General* agregar en *Additional Library Directories* los valores: *$(GRT_LIB);*
    c. *Bajo Linker › Input*, agregar en *Additional Dependencies* la siguiente lista de dependencias: *GRTLibDebug.lib;*
    d. Aplicar las modificaciones y guardar el paquete de reglas.

3. Realizar los mismos pasos pero ahora agregando una nueva regla al paquete de reglas correspondiente a *Release*. Tener en cuenta que en este caso las librerías que se indican en *Linker ‣ Input* en el paso (c) serán las mismas que para el caso anterior pero ahora las correspondientes a la versión en *release*.
4. Compilar la aplicación que hace uso de código GRT y ejecutar.

K.7. *Boost*

K.7.1. Instalar *Boost*

1. Descargar la última *release* de *Boost*.
2. Mover el contenido descargado en el lugar donde se quiera tener instalado *Boost* en el sistema.
3. Una vez que se escogió y movieron los datos a la ubicación elegida, para simplificar y eliminar las dependencias de la ubicación en la que esté la librería, se debe agregar una variable de entorno llamada *BOOST_INCLUDE* y asociar el valor según la ruta en la cual quedaron alojados los archivo de encabezado de *Boost*. El valor de esta variable sería por ejemplo: *D:\FIng\Proyecto de grado\Programas\boost_1_54_0.*
4. Análogamente al paso anterior se debe crear otra variable de entorno llamada *BOOST_LIB* y asociar el valor según la ruta en la cual quedaron alojadas las librerías de *Boost*. El valor de esta variable sería por ejemplo: *D:\FIng\Proyecto de grado\Programas\boost_1_54_0\stage\lib.*
5. La mayoría de las librerías de *Boost* son del tipo *header-only*, esto quiere decir que las mismas solo consisten de archivos del tipo encabezado por lo cual, no necesitan un archivo compilado aparte. Sin embargo, si se quieren hacer uso de algunas librerías especiales de forma separada, las mismas si necesitan generar los binarios asociados; alguna de estas librerías son por ejemplo *Boost::Serialization*, *Boost::RegEx*, entre otras. Para generar los binarios necesarios se debe primero desde una consola de *Visual Studio*, Herramientas -> Símbolo del Sistema de *Visual Studio*, dirigirse hacia donde está instalado *Boost* para luego ejecutar el comando *boostrap* y luego el comando *./b*". El primero de estos comandos prepara el entorno *Boost.Build* para ejecutar el *build* y el segundo lo lleva a cabo generando los binarios correspondientes a todas las librerías (se puede especificar la solamente la que se requiera). Los binarios resultados del *build* quedan dentro del directorio de instalación bajo la ruta *stage/lib*.

K.7.2. Configurar *Boost* en VS2012

1. Crear una solución nueva o abrir una existente en la cual se quiera incorporar Boost.
2. Ir a la pestaña *Property Manager* donde se listan los diferentes conjuntos de reglas que hay para cada uno de los modos de compilación *Debug* y *Release*. Se debe agregar un nuevo conjunto de reglas al modo *Debug* con nombre *BoostSerealization_Debug* y configurar lo siguiente:

    a. Bajo *C/C++ ‣ General*, agregar en *Additional Include Directories* los valores: *$(BOOST_INCLUDE);*
    b. Bajo *Linker ‣ General* agregar en *Additional Library Directories* los valores: *$(BOOST_LIB);*
    c. Bajo *Linker ‣ Input*, agregar en *Additional Dependencies* la siguiente lista de dependencias: *libboost_serialization-vc110-mt-gd-1_54.lib;*
    d. Aplicar las modificaciones y guardar el paquete de reglas.

3. Realizar los mismos pasos pero ahora agregando una nueva regla al paquete de reglas correspondiente a *Release*. Tener en cuenta que en este caso las librerías que se indican en

Linker › Input en el paso (c) serán las mismas que para el caso anterior pero ahora las correspondientes a su versión en *release*.

4. Compilar la aplicación que hace uso de código *Boost* y ejecutar.

K.8. Eigen

K.8.1. Instalar Eigen

1. Descargar la última *release* de *Eigen*.
2. Mover el contenido descargado en el lugar donde se quiera tener instalado *Boost* en el sistema.
3. Una vez que se escogió y movieron los datos a la ubicación elegida, para simplificar y eliminar las dependencias de la ubicación en la que esté la librería, se debe agregar una variable de entorno llamada *EIGEN_INCLUDE* y asociar el valor según la ruta en la cual quedaron alojados los archivo de encabezado de Boost. El valor de esta variable sería por ejemplo: *"D:\FIng\Proyecto de grado\Programas\eigen-eigen-36fd1ba04c12".*
4. Las librerías de *Eigen* son del tipo *header-only*, esto quiere decir que las mismas solo consisten de archivos del tipo encabezado por lo cual, no necesitan un archivo compilado aparte y por ello, no es necesario ejecutar pasos de compilación.

K.8.2. Configurar *Eigen* en VS2012

1. Crear una solución nueva o abrir una existente en la cual se quiera incorporar *Boost*.
2. Ir a la pestaña *Property Manager* donde se listan los diferentes conjuntos de reglas que hay para cada uno de los modos de compilación *Debug* y *Release*. Se debe agregar un nuevo conjunto de reglas al modo *Debug* con nombre *Eigen_Debug* y configurar lo siguiente:

    a. Bajo *C/C++ › General*, agregar en *Additional Include Directories* los valores: *$(EIGEN_INCLUDE);*
    b. Aplicar las modificaciones y guardar el paquete de reglas.

3. Realizar los mismos pasos pero ahora agregando una nueva regla al paquete de reglas correspondiente a *Release*. Tener en cuenta que en este caso las librerías que se indican en *Linker › Input* en el paso (c) serán las mismas que para el caso anterior pero ahora las correspondientes su versión en *release*.
4. Compilar la aplicación que hace uso de código *Eigen* y ejecutar.

K.9. Posibles problemas durante la configuración

1. Si al momento de *linkeditar* no se encuentran alguna de las las librerías estáticas es porque no están bien referenciadas en el proyecto. En este caso se debe revisar que la ruta especificada en las hojas de propiedades correspondiente a la librería faltante sea correcta.
2. Si al momento de ejecutar no se encuentran las *DLLs* de alguno de los *frameworks* es debido a que no están bien referenciadas en el *PATH* del sistema o bien porque no se ha copiado a la carpeta donde se encuentra el ejecutable de la aplicación.
3. Si al momento de ejecutar, la aplicación arroja un error referente a la falta de la *DLL tbb_debug.dll*, es un *bug* conocido del *framework* de *OpenCV*. Según se comenta en los foros de la comunidad sólo ocurría este error de la versión 2.3 hacia atrás, pero al parecer también ocurre para la versión 2.4.2. Para las versiones viejas de *OpenCV* había que descargarse la

librería manualmente, pero la versión 2.4.2 de *OpenCV* ya trae las *DLLs* de *TBB*; lo único que hay que hacer entonces es o bien agregar esa librería dinámica al *PATH* del sistema o bien copiarla a la carpeta donde se encuentra el ejecutable de la aplicación.

4. Si al compilar la aplicación retorna un error al compilar en *LeapMath.h*, en principio lo que se debe hacer es comentar las líneas de definición de las tres constantes en este fichero. Hecho esto todo debe compilar de forma correcta, pero aún no se sabe el motivo de este error de compilación.
5. En los últimos *releases* del *Leap SDK* (por ejemplo para las versiones *v2* del *SDK*) están funcionando solo las librerías dinámicas de 32 bits, para las de 64 bits se lanza un error al intentar levantar el *LeapService*. Lo que se debe hacer entonces es copiar las librerías que se encuentran en *LeapSDK\x86* tanto a *Windows\system32* como a *Windows\SysWOW64*. Para copiar las librerías puede ser necesario bajar el *LeapService* si es que estaba levantado (sino puede lanzar un error como que no se puede modificar/borrar el fichero porque está en uso). Para detener el *LeapService* se debe abrir una línea de comandos y ejecutar *"net stop LeapService"* y para levantarlo una vez copiadas todas las librerías *"net start LeapService"*.
6. Al querer iniciar el *Leap Motion Visualizer* para algunas versiones no levanta el ejecutable encargado de registrar el servicio y por lo tanto el *Leap Motion* ni siquiera prenderá los infrarrojos. Una primera solución, según se menciona en los foros de la comunidad, es desactivar el antivirus *Avast* durante la instalación, ya que aparentemente filtra la ejecución de ese ejecutable; pero probando esta solución no se tuvo resultados positivos. Lo que sí brindó una solución positiva fue otra recomendación de la comunidad la cual indica ejecutar a mano el *LeapSvc.exe* que levanta los servicios. Para esto se debe ir al directorio donde quedó instalado el *Leap SDK* y dentro del directorio *Core Services*, ejecutar *LeapSvc.exe --run*.

Anexo L: Pruebas de concepto

L.1. Primer prueba de concepto

L.1.1 Descripción

Se partió primeramente con las construcción de una aplicación sencilla que consiste básicamente en interactuar mediante distintas vías con un conjunto de esferas que se muestran en pantalla. En esta primera aplicación se ahondaron en los conocimientos referentes a *OpenNI/NITE 1.5*, *openFrameworks* y el uso del sensor *Microsoft Kinect*, como así también, a los conceptos de *hand-tracking* y reconocimiento de gestos.

Las interacciones en esta aplicación pueden ser de dos tipos, o más específicamente, mediante dos tipos de entradas diferentes; por un lado, interacción básica proveyendo a la aplicación con entrada mediante teclado y mouse (*GUI interaction*), y por otro lado, haciendo uso del sensor *Microsoft Kinect* para el reconocimiento de gestos y *hand-tracking* (*NUI interaction*). Las esferas actuarán entonces en consecuencia según la entrada provista por el usuario.

Mientras no se interactúe con la aplicación, la misma estará en un estado inicial en el cual simplemente se visualizarán cada uno de los diferentes tipos de esferas con las que se cuenta. Dichas esferas se clasifican básicamente en cuatro tipos dependiendo de su comportamiento como se puede ver en la *Figura 166*:

1. *Little Ball:* pequeña esfera de color azul que parece estar cargada eléctricamente, presentando un movimiento de "temblor" constante.
2. *Static Ball:* esfera de color rosa que presenta la característica de permanecer quieta a menos que se mueva explícitamente.
3. *Annoying Ball:* pequeña esfera de color rojo que se caracteriza por seguir el puntero constantemente.
4. *Walking Ball:* esfera de color amarillo que se traslada constantemente de un extremo al extremo opuesto de la pantalla en forma horizontal.

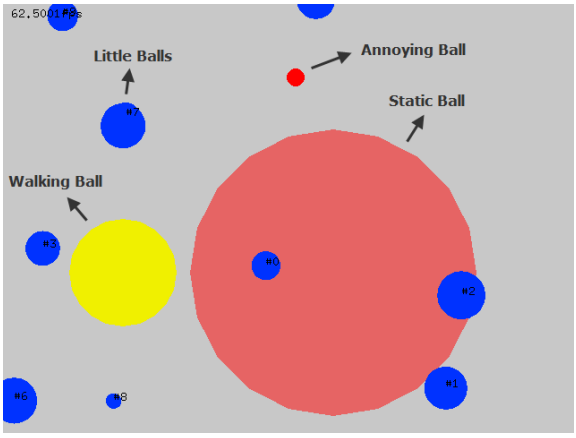


Figura 166: Tipos de esferas en la aplicación de prueba de concepto

En lo que respecta a la interacción básica mediante teclado y/o *mouse*, la aplicación provee las siguientes funcionalidades:

1. *Click* con el *mouse* sobre cualquiera de las esferas: hace que la esfera a la cual se le hizo *click* expanda su tamaño en cierta proporción.
2. Selección de una esfera mediante teclado para tomar su control: las esferas están etiquetadas con números y podrán ser seleccionadas individualmente digitando el número correspondiente en el teclado.
3. Movimiento mediante teclado: una vez se tiene seleccionada una esfera según lo indicado en el paso anterior, se la podrá trasladar a una nueva posición utilizando las teclas de dirección W-S-A-D del teclado, haciendo que se mueva hacia arriba, abajo, izquierda y derecha respectivamente.
4. Deseleccionar una esfera: se puede deseleccionar una esfera previamente seleccionada digitando nuevamente el número correspondiente en el teclado para liberar el control sobre ella.

Por otra lado, en cuanto a la interacción gestual mediante el sensor *Microsoft Kinect*, las funcionalidades provistas por la aplicación son las siguientes:

1. Reconocimiento de usuario: mediante el reconocimiento del gesto *wave* debiendo realizar un saludo con cualquiera de las manos en alto de cara al sensor es posible determinar el reconocimiento de un usuario.
2. Control de las esferas: una vez reconocido un usuario en la escena, éste tendrá control sobre la esfera seleccionada, es decir que, en caso de existir una esfera seleccionada se moverá en base a los movimientos de la mano del usuario hasta que sea deseleccionada.

A grandes rasgos, la arquitectura general de la aplicación se puede observar en la *Figura 167,* la cual permite visualizar los diferentes componentes que la conforman.

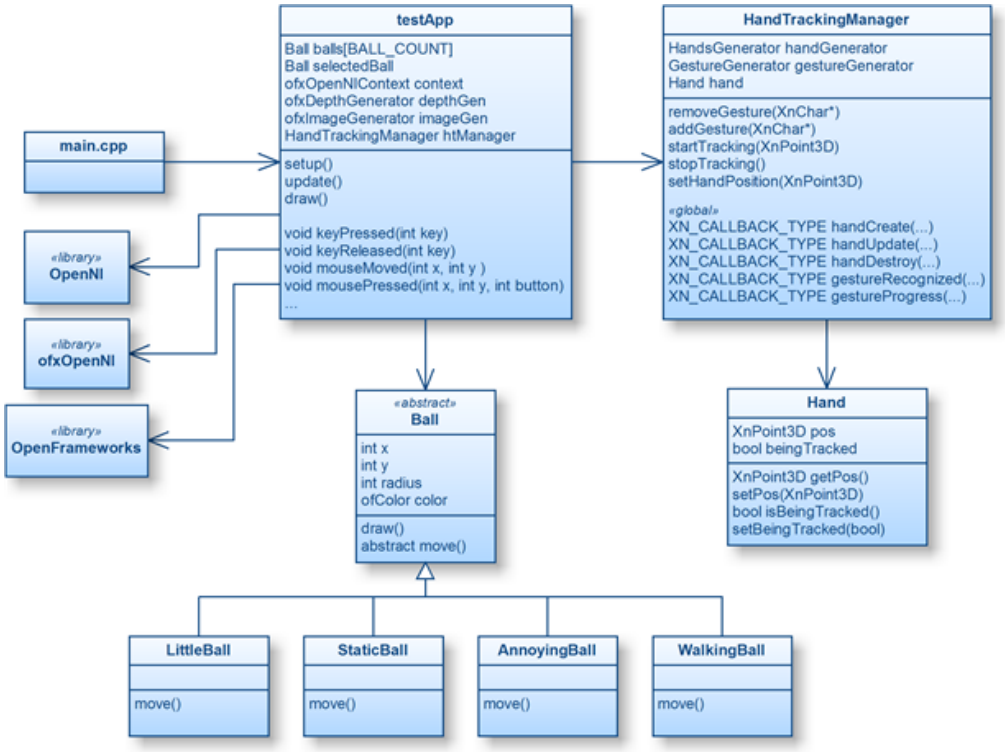


Figura 167: Diagrama de clases de la aplicación

Como se mencionó en el *Capítulo 2* cualquier aplicación que utilice *openFrameworks* tiene una estructura bien definida dada por los métodos *setup(), update()* y *draw()* que se ejecutan en ese orden (*setup()* una única vez, mientras que los restantes se ejecutan secuencialmente en un *loop* infinito). La lógica que se ejecuta en este caso en cada uno de los métodos anteriores es como se detalla a

continuación.

En el método *setup()* se realizan varias acciones importantes:

1. Se instancian los diferentes tipos de esferas que conformarán la escena, inicializando ciertos parámetros obligatorios como tamaño, posición inicial y color.
2. Se inicializa el contexto *OpenNI 1.5*, componente principal que se encarga de mantener en todo momento el estado completo de la aplicación, entendiendo por esto al estado de todos los módulos (sensores y componentes *middleware*) registrados en el *framework* y los *production nodes* utilizados por la aplicación (generadores de mapas de profundidad, mapas *RGB*, etc). El contexto debe ser inicializado una única vez antes de poder utilizar cualquier componente del *framework*. Cuando se inicializa, todos los módulos registrados son cargados y analizados para su posterior uso.
3. Se setea la propiedad *mirror* del contexto en verdadero para hacer que los *frames* se espejen una vez obtenidos. Esto es útil cuando el sensor se coloca enfrente al usuario, ya que hace que la mano derecha del usuario se visualice a la derecha en los mapas obtenidos mediante *OpenNI*, la mano izquierda a la izquierda, etc.
4. Se inicializan los nodos *OpenNI 1.5* a utilizar. Para esta aplicación los nodos utilizados son básicamente los siguientes cuatro:

    a. *DepthGenerator:* para la obtención del mapa de profundidad de la escena.
    b. *ImageGenerator:* para la obtención del mapa *RGB* de la escena.
    c. *GestureGenerator:* para el reconocimiento de gestos.
    d. *HandsGenerator:* para el reconocimiento y seguimiento de una mano.

En este punto, todos los generadores ya quedan configurados y generando datos para ser luego consumidos por la aplicación en cada nuevo *frame*. Los primeros dos generadores se utilizaron encapsulados mediante el *addon ofxOpenNI* por ciertas razones que se detallan más adelante, mientras que, los restantes son los nativos de *OpenNI 1.5* y son utilizados como tales, aunque su inicialización y uso está encapsulado en la clase *HandTrackingManager* incluida en el diagrama de la *Figura 167*, la cual se encarga de gestionar todo lo referente al *hand-tracking*.

Luego, en el método *update()* que se ejecuta previo a dibujar cada *frame*, se realizan las siguientes tareas:

1. Se verifica si se está realizando el seguimiento de una mano mediante el método *isBeingTracked()* del objeto que representa la mano del usuario y en caso afirmativo se actualiza la posición de la esfera actualmente seleccionada (si es que existe) según la posición actual de la mano.
2. Para las demás esferas simplemente se actualiza la posición según el método *move()* para implementar el movimiento de ese tipo de esfera en particular.
3. Se actualiza el contexto y los generadores para obtener nuevos datos a procesar en el siguiente *frame*.

Finalmente, en el método *draw()*, ejecutado para dibujar cada *frame*, se llevan a cabo las siguientes acciones:

1. Se dibuja cada una de las esferas según los datos de estado que posea en ese momento (posición, color, etc.) mediante el método *draw()* implementado por la esfera.
2. Se dibuja el mapa de profundidad y el mapa *RGB* simplemente invocando la operación *draw()* del componente *ofxOpenNI* que los representa, especificando previamente una posición en la cual dibujarlos y un tamaño para cada uno de ellas.

L.1.2. Resultados obtenidos

Como se mencionó se utilizó el sensor *Microsoft Kinect* y el *addon* de *openFrameworks ofxOpenNI* que permite abstraer el manejo del *framework OpenNI/NITE 1.5*. La razón por la cual se utilizó *ofxOpenNI* en estos casos fue principalmente debido a que permite visualizar de forma sencilla estos mapas en pantalla mediante la invocación a un único método *draw()* que implementan los distintos generadores provistos por *ofxOpenNI*. Esta función, dado el conjunto de píxeles de los mapas, los dibuja en cierta posición de la ventana con cierto tamaño previamente especificado, evitando que el programador deba manipular los píxeles a bajo nivel para mostrarlos correctamente en pantalla.

Otro problema que ayudó a resolver de forma sencilla el uso del *addon ofxOpenNI* fue el hecho de que una vez que se obtiene la posición de la mano que está siendo seguida, se debe convertir esos datos obtenidos en coordenadas de mundo (o más específicamente, coordenadas de sensor) a coordenadas de pantalla para su correcta visualización; en este caso, para trasladar la esfera seleccionada a la nueva posición en base al *hand-tracking* de la mano del usuario. La operación que se encarga de esto, permitiendo ahorrar todos los cálculos necesarios para realizar la proyección, es *ConvertRealWorldToProjective()* que se incluye como parte de los generadores de *ofxOpenNI*. Si no se realiza esta conversión los movimientos realizados con la mano no se mapean correctamente con los movimientos de la esfera en pantalla.

Algunos problemas que se debieron resolver fueron los referente a los *callbacks* de *OpenNI* (funciones que implementan la lógica a ejecutar al dispararse cierto evento, por ejemplo, el reconocimiento de un nuevo usuario), más específicamente sobre cómo y dónde definirlos para que todo funcione como es esperado. Idealmente deberían ser definidos e implementados de forma global a cualquier clase, en el mismo fichero en el que se realiza el registro para los diferentes nodos. Sin embargo, un problema concreto con el que se topó relacionado con los *callbacks* fue que algunos *callbacks* requerían interactuar con un generador que no estaba en su alcance, y esto no es posible por cómo se deben definir los *callbacks* (de forma global, no pertenecientes a ninguna clase particular) y debido a que los generadores no fueron definidos globalmente sino como parte de una clase como se puede ver en el diagrama presentado en la *Figura 167.* Por ejemplo, el *callback* correspondiente al reconocimiento de un gesto, *Gesture_Recognized*, debe lanzar el *tracking* de la mano invocando al método *StartTracking()* del *HandGenerator*, indicando que ya se reconoció el gesto y que está en condiciones de comenzar a *trackear*. El problema puntual, como se mencionaba, se da debido a que al no ser globales los objetos que representan ambos *generators*, no son visibles en los *callbacks* según cómo fueron declarados. Se buscaron otras formas de declarar los *callbacks* para solucionar este problema pero no se tuvo éxito. Así, la solución por la que se optó fue incluir a la solución una nueva clase que se encargue de manejar todo lo referente al *hand-tracking*, e incluya como atributos tanto el *HandGenerator* como el *GestureGenerator*. De esta forma, al registrar los *callbacks* para los diferentes eventos se pasa en el parámetro *pCookie* la instancia de la clase *HandTrackingManager*, haciendo que desde los *callbacks* se tenga acceso a todos los atributos sus atributos, incluyendo, los necesarios *HandGenerator* y *GestureGenerator*. A pesar de que el parámetro *pCookie* no parece estar para eso, este fue el *workaround* adoptado para que los *callbacks* tengan visibilidad tal que, abarque nodos a los cuales no están "asociados", en este caso para que desde *callbacks* referentes al *GestureGenerator* se pueda utilizar el *HandGenerator* y viceversa como es necesario.

Luego de la descripción presentada se puede observar que la aplicación es muy sencilla y su funcionalidad es bastante limitada. De todos modos, fue útil como forma de comprobar que las tecnologías utilizadas pueden interactuar sin problemas, además de tener un primer acercamiento al desarrollo de aplicaciones basadas en NUI (*Natural User Interaction).* Específicamente, esta aplicación brindó un primer acercamiento a la interacción con el sensor *Microsoft Kinect* que mediante el uso de *OpenNI/NiTE 1.5* permitieron el reconocimiento de gestos así como también al *hand-tracking* y el manejo de los datos que provee *Microsoft Kinect*. A su vez, se implementó una aplicación que haga uso de *openFrameworks* y con ello, el manejo de su estructura básica para el desarrollo de aplicaciones. Todos estos *frameworks* y vías de interacción probadas son centrales para el desarrollo de la solución global al proyecto y esta primer prueba de concepto también permitió buscar las primeras soluciones a los problemas que pueden surgir con el uso

de estas tecnologías.

El objetivo principal de esta prueba de concepto fue utilizar esta aplicación como base para implementar prototipos que sirvieran para mitigar ciertos riesgos, expandiendo sus funcionalidades con las nuevas funcionalidades que se deseen probar conceptualmente. En el vídeo de desarrollo "*Interactive circles*" [v.4] se puede observar el funcionamiento de la prueba de concepto descrita.

L.2. Segunda prueba de concepto

L.2.1 Descripción

Como se planteó en el *Capítulo 2* este proyecto hace uso de dos sensores diferentes los cuales son los ya nombrados *Leap Motion* y *Microsoft Kinect*, a su vez, se hará uso de múltiples unidades de estos sensores. Por ello, el punto siguiente a abordar fué cómo integrar los diferentes sensores para que trabajen en la construcción de un único *framework* y así sacar el mayor provecho de las mejores cualidades de cada uno de ellos.

Para abordar lo mencionado, resulta vital que se pueda llevar a un mismo sistema de referencia lo que cada uno de estos sensores capta de forma de enriquecer lo que uno de estos dispositivos "ve" mediante lo que el otro también puede "ver". A modo de ejemplo, se podría detectar la presencia de un usuario en la escena mediante el sensor *Microsoft Kinect*, identificar también con éste sus distintas articulaciones (*joints*) y por el otro lado, mediante *Leap Motion* detectar los dedos de dicho usuario. Llevando esta información a un único sistema de referencia se podría entonces asociar los dedos detectados por el sensor *Leap Motion* a un usuario determinado y detectado con *Microsoft Kinect* logrando de esta forma obtener más información que la proporcionada de forma aislada por cada uno de los sensores.

A modo de resumen, a continuación se vuelven a detallar cada uno de los sistemas de referencia de los sensores *Leap Motion* y *Microsoft Kinect* brindados a través de la *SDK* propia de *Leap Motion* y de *OpenNI* respectivamente.

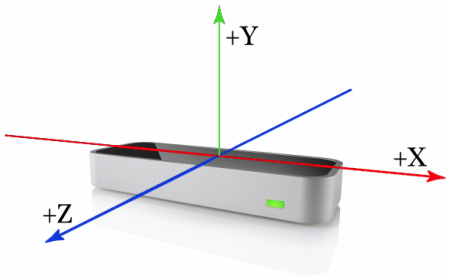


Figura 168: Sistema de referencia del sensor *Leap Motion* [62]

Leap Motion a través de su *SDK* utiliza un sistema de referencia cartesiano diestro. Los valores reportados por el sensor son medidos en milímetros. El origen del centro de referencia es el centro del sensor como se muestra en la *Figura 168*. El eje X y el eje Z conforma un plano horizontal donde el eje X se encuentra paralelo al lado más largo del dispositivo. Por otro lado, el eje Y es vertical, considerando los valores positivos a aquellos que se encuentren por encima del dispositivo mientras que el eje Z toma valores positivos a aquellos por delante del mismo.

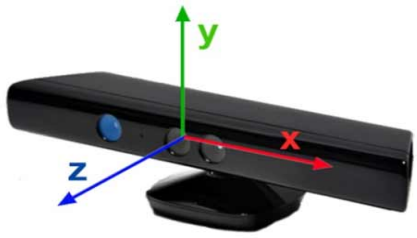


Figura 169: Sistema de referencia del sensor *Microsoft Kinect* [4]

Por otro lado, *Microsoft Kinect* al utilizar *OpenNI* también utiliza un sistema de referencia cartesiano derecho. Los valores reportados por el sensor son medidos en milímetros igual que para *Leap Motion*. El origen del centro de referencia es el centro de la cámara infrarroja del sensor como se muestra en la *Figura 169.* El eje X y el eje Z conforma un plano horizontal donde el eje X se encuentra paralelo al lado más largo del dispositivo. Por otro lado, el eje Y es vertical, considerando los valores positivos a aquellos que se encuentren por arriba del sensor mientras que el eje Z toma valores positivos a aquellos por delante del mismo exactamente como los valores reportados por *Leap Motion*.

La segunda prueba de concepto para llevar a cabo lo explicitado anteriormente consistió en tomar como sistema de referencia global al sistema del sensor *Microsoft Kinect*. De esta forma, lo captado por *Microsoft Kinect* ya va a estar con coordenadas correctas pero, lo captado por el *Leap Motion* no. Entonces, para que lo captado por *Leap Motion* sea llevado al sistema de referencia de *Microsoft Kinect* se debe de realizar una transformación y para aplicar dicha transformación se debe hacer uso de operaciones matemáticas en el espacio. En concreto, aplicar producto de matrices que representen las rotaciones y traslaciones necesarias para pasar de un sistema de referencia a otro.

Se plantea un caso sencillo, extendiendo la primer prueba de concepto, en donde se enfrentan ambos sensores como se muestra en la *Figura 170.* En este caso, es fácil observar que para llevar lo captado por *Leap Motion* al sistema de referencia de *Microsoft Kinect* se debe aplicar una transformación en el eje Z y en el eje Y, ya que hay una diferencia de altura en Y y una separación en Z; específicamente entonces, es necesario hacer una traslación en ambos ejes.

Para efectuar dicha transformación es necesario representarla mediante el producto de matrices de rotación y traslación. Para trabajar con matrices de transformación se utilizó la librería llamada *Eigen* (descrita en el *Capítulo 2*) la cual permitió un conjunto de facilidades de operación matemática.

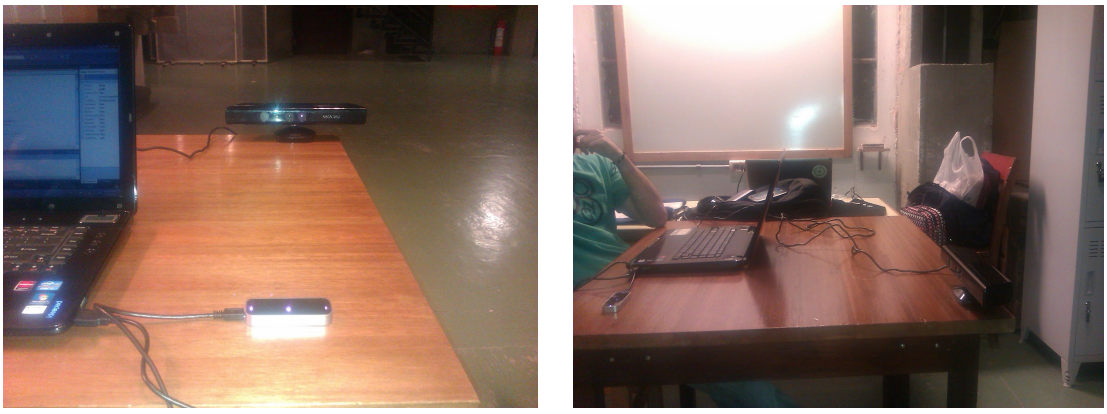


Figura 170: Disposición de los sensores *Leap Motion* y *Microsoft Kinect*

Específicamente, en esta prueba los sensores se encuentran separados una distancia de 60 cm en el eje Z y una distancia de 8 cm en el eje Y. Al momento de establecer las distancias se debe tener en cuenta que para que *Microsoft Kinect* detecte movimiento se debe de tener con respecto al usuario una distancia mínima de 40 cm, un ejemplo de posición entonces es el presentado anteriormente. De esta

forma, construimos un vector de traslación formado por los valores (0,8,60) y construimos mediante *Eigen* una matriz de transformación.

Para comprobar la correcta transformación entre los sistemas de referencia se tomó como punto referente el centro de la mano del usuario. De esta forma, se puede obtener dicho centro mediante el sensor *Leap Motion*, aplicarle la matriz de transformación y luego comparar el valor obtenido con el valor obtenido del centro de la mano detectado por *Microsoft Kinect*. Los resultados obtenidos fueron buenos, exceptuando por pequeños márgenes de error predecibles, está dentro de lo esperado.

L.2.2 Resultados obtenidos

Mediante una prueba de concepto simple, se logró demostrar que es relativamente sencillo lograr llevar lo captado por distintos sensores a un único sistema de referencia para poder operar en conjunto con los datos obtenidos. Se pudo observar que, creando de forma correcta una matriz de transformación cualquier posición, salvando las distancias mencionadas más arriba, es "mapeable" a un único sistema de referencia. A su vez, este sistema de referencia no tiene porque ser el de un sensor en concreto, puede estar por ejemplo, en el medio de la distancia entre los dos o en cualquier lugar de interés, sólo basta definir una matriz correcta de transformación que se ajuste a la posición deseada.

Anexo M: *API Reference Guide*

En este anexo se detallan todos los aspectos referentes a la implementación de aplicaciones sobre el *framework* construido, a modo de guía para los programadores de aplicaciones, que deseen utilizar los servicios brindados por la solución. Se describe entonces, el proceso de inicialización del sistema con el fin de que brinde los diferentes servicios correctamente, la forma de definir el comportamiento de la aplicación ante la recepción de los diferentes eventos por parte de los usuarios regida por los distintos manejadores de eventos que forman parte del *API* del *framework*, y finalmente, se detallan las configuraciones necesarias para el correcto funcionamiento del sistema.

M.1. Inicialización

El *API* del *framework* provee básicamente tres funciones que permiten gestionar el proceso de inicialización y finalización general del sistema. A continuación, se presenta la firma de cada una de estas funciones a modo de referencia, una descripción de lo que realiza cada una ellas y un ejemplo concreto de uso. Todas estas funciones están definidas como parte de la interfaz del sistema *IAnimusCore* que forma parte del *Core* del sistema, la cual es una de las dos componentes que conforman el *API* completa del *framework*. Cabe mencionar a su vez, que las funciones están definidas dentro del *namespace animus*, por lo que se debe utilizar dicho *namespace* dentro de la aplicación que requiera su uso.

int registerToGestures(set<GestureType> gestures)

Descripción	Permite indicar los diferentes gestos que va a soportar la aplicación. Esto permite al sistema conocer para cuáles de los gestos reconocidos se deben generar notificaciones y para cuáles no. Aquellos gestos reconocidos por el sistema que no estén dentro de los gestos registrados por la aplicación serán filtrados sin generar la notificación correspondiente a la aplicación.
Valor de retorno	Se retorna un entero cuyo valor será 1 si el registro de los gestos en el sistema se realizó correctamente y 0 en caso contrario.
Parámetros	Como parámetro de entrada se debe especificar el conjunto de gestos que se desea registrar. Cada uno de los gestos de este conjunto debe ser de tipo *GestureType*, cuyos valores posibles están definidos en la *Figura 171.*

int startAnimus(IAnimusApplication* app)

Descripción	Permite inicializar el sistema, instanciando tanto la aplicación desarrollada como los diferentes servicios provistos por el *framework*.
Valor de retorno	Se retorna un entero cuyo valor será 1 si se inicia el sistema correctamente y 0 en caso contrario.
Parámetros	Como parámetro de entrada se debe ingresar una referencia al punto de entrada de la aplicación desarrollada, es decir, aquel componente de la aplicación que implementa la interfaz del sistema *IAnimusApplication*, y por ende contiene la lógica correspondiente a los diferentes eventos que puede enviar el sistema hacia la aplicación.

int stopAnimus()

Descripción	Permite indicar al sistema que deje de brindar los servicios. Debe ser utilizada una vez que la aplicación finalice su ejecución, con el fin de que el sistema libere las diferentes

	estructuras que está utilizando y no se generen inconsistencias en futuras ejecuciones.
Valor de retorno	Se retorna un entero cuyo valor será 1 si se detiene el sistema correctamente y 0 en caso contrario.
Parámetros	No requiere ningún parámetro.

De esta forma, el flujo general de inicialización y gestión del sistema implica inicialmente el registro de los gestos a soportar por la aplicación mediante la función de sistema *registerToGestures*. Luego, en caso de no haber errores, se da la orden de inicializar el sistema mediante la función *startAnimus*, y finalmente, una vez se desee finalizar la ejecución de la aplicación se debe invocar la función *stopAnimus* para que el sistema pueda liberar correctamente los diferentes recursos que tiene asignados. En el código de ejemplo incluido en la *Tabla 14* se puede apreciar este proceso (no se tiene en cuenta el manejo de errores por simplicidad del pseudocódigo), el cual debe incluirse como parte del programa principal (*main*) en la aplicación. Cabe mencionar su vez, que el componente genérico *UserApplication* es quien contiene la lógica principal de la aplicación desarrollada sobre el *framework*, es decir, aquella que implementa la interfaz del sistema *IAnimusApplication*.

```
int main() {

        IAnimusApplication* app = UserApplication::getInstance();
        set<GestureType> gestures;

        // HAND GESTURES
        gestures.insert(GestureType::ANIMUS_3D_PINCH_HAND);
        gestures.insert(GestureType::ANIMUS_3D_GRAB_HAND);

        // FINGER GESTURES
        gestures.insert(GestureType::ANIMUS_3D_SWIPE_FINGER);
        gestures.insert(GestureType::ANIMUS_3D_CIRCLE_FINGER);

        // TOUCH GESTURES
        gestures.insert(GestureType::ANIMUS_TOUCH_TAP_FINGER);
        gestures.insert(GestureType::ANIMUS_TOUCH_DOUBLE_TAP_FINGER);

        int animusRc = animus::registerToGestures(gestures);

        if (animusRc) {

                animusRc = animus::startAnimus(solarSystemApp);
        }

        return 0;
}
```

Tabla 14: Pseudocódigo de ejemplo para la inicialización del sistema

M.2. Gestión de eventos

Adicionalmente a las funciones de inicialización que forman parte del *API* provista por el *framework*, otra parte esencial del *API* son las funciones manejadoras de eventos. Estas funciones están definidas como parte de la interfaz del sistema *IAnimusApplication*, y todas ellas son funciones virtuales puras que deben ser implementadas por la aplicación a desarrollar. En cada una de estas funciones se debe incluir la lógica necesaria para manejar el evento al cual corresponda la función. Así, por ejemplo, la componente de la aplicación que implemente esta interfaz debe definir el comportamiento correspondiente a la recepción de un evento de gesto reconocido de entre cualquier de los registrados, un evento de actualización de usuario, etc.

Los eventos están representados en la *API* por la clase *AnimusEvent* cuyos atributos son el enumerado *EventType* y *AnimusEventDetail* que representa el evento en particular ocurrido. Éste último se encuentra organizado jerárquicamente por lo que de él descienden estructuras de datos adicionales que

permiten representar eventos más específicos. Esta jerarquía establece que un *AnimusEventDetail*, puede ser o bien un *AnimusEventGesture,* o un *AnimusEventUsrUpdate*. A su vez, *AnimusEventUsrUpdate* posee un atributo enumerado del tipo *UserUpdateType* cuyos valores están relacionados a la forma que puede tomar el *AnimusEventUsrUpdate,* siendo éste un *AnimusBodyUpdate* o un *AnimusHandsUpdate* según la actualización que corresponda. Así, un *AnimusHandsUpdate* permite representar la actualización de posiciones de manos y dedos de los usuario, mientras que un *AnimusBodyUpdate* permite representar la actualización de posiciones de los diferentes *joints* del esqueleto del usuario.

Por otra parte, un evento de tipo *AnimusEventGesture* posee dos atributos enumerados del tipo *GestureType* (definido anteriormente) y *GestureStateType.* De esta forma el enumerado del tipo *GestureType* permite saber qué forma toma el *AnimusEventGesture* ya que representa un evento de reconocimiento gestual por la *API* y por lo tanto, existen múltiples gestos diferentes por lo que cada uno de ellos es representado por una estructura de datos específica que contiene toda la información para el gesto en cuestión (el diagrama de todas las estructuras mencionadas se encuentra en el *Capítulo 5* del informe principal).

En lo que resta de esta sección se listan las firmas de las funciones manejadoras de eventos a implementar por el programador de la aplicación, detallando sus parámetros, así como también, un código de ejemplo que ilustra una implementación correcta de estas funciones de sistema.

Cabe mencionar que la aplicación no debe implementar todas y cada una de estas funciones. Si bien, por ejemplo, las funciones *setup*, *update* y *draw*, son importantes dado que rigen las inicializaciones dentro de la aplicación y las diferentes actualizaciones de los componentes en cada *frame*, existen manejadores que son opcionales como por ejemplo, los correspondientes a eventos realizados con teclado y *mouse*, dado que la idea de una aplicación que utilice los servicios del *framework* es que la interacción sea puramente táctil y/o gestual. De todas formas, este tipo de eventos pueden utilizarse a modo de *debug* mientras se está desarrollando la aplicación. Del mismo modo, no es necesario que la aplicación implemente todos los manejadores para cada posible evento que pueda enviar el sistema, ya que muchos de ellos directamente se corresponden con gestos a los que la aplicación no está suscrita. En este sentido, sólo se deben implementar aquellos manejadores correspondientes a gestos soportados por la aplicación (para los cuales se registró previamente) o eventos generales como pueden ser las notificaciones de usuario nuevo o actualización de usuario.

void setup()

Descripción	Utilizada para inicializar las diferentes variables que forman parte de la aplicación de usuario.
Valor de retorno	No retorna ningún valor.
Parámetros	No requiere ningún parámetro.

void update()

Descripción	Esta función se ejecuta en cada iteración del ciclo de ejecución de la aplicación, previo a dibujar cada *frame*, por lo que es utilizada para actualizar las diferentes variables que controlan el flujo de ejecución de la aplicación de usuario.
Valor de retorno	No retorna ningún valor.
Parámetros	No requiere ningún parámetro.

void draw()

Descripción	Esta función se ejecuta en cada iteración del ciclo de ejecución de la aplicación con el objetivo de dibujar el *frame* actual, inmediatamente luego de ejecutar la función *update*. De esta forma, es utilizada para dibujar los diferentes elementos gráficos a

	visualizar como parte de la aplicación.
Valor de retorno	No retorna ningún valor.
Parámetros	No requiere ningún parámetro.

void exit()

Descripción	Análoga a la función *setup*, pero en este caso se utiliza para liberar los diferentes recursos que la aplicación esté utilizando, ya que se ejecuta justo antes de cerrar la aplicación.
Valor de retorno	No retorna ningún valor.
Parámetros	No requiere ningún parámetro.

void keyPressed(int key)

Descripción	Permite capturar el evento que se envía cada vez que se presiona una tecla en el teclado.
Valor de retorno	No retorna ningún valor.
Parámetros	En el parámetro de entrada *key* queda almacenada la clave ordinal de la tecla presionada.

void keyReleased(int key)

Descripción	Permite capturar el evento que se envía cada vez que se deja de presionar una tecla en el teclado.
Valor de retorno	No retorna ningún valor.
Parámetros	En el parámetro de entrada *key* queda almacenada la clave ordinal de la tecla que estaba presionada.

void mouseMoved(int x, int y)

Descripción	Permite capturar el evento que se envía cada vez que el puntero del *mouse* cambia de posición.
Valor de retorno	No retorna ningún valor.
Parámetros	En los parámetros de entrada x e *y* quedan almacenadas las coordenadas del puntero en la nueva posición, en los ejes X e Y respectivamente.

void mouseDragged(int x , int y, int button)

Descripción	Permite capturar el evento que se envía cada vez que el puntero del *mouse* cambia de posición mientras se mantiene presionado alguno de los botones del *mouse*.
Valor de retorno	No retorna ningún valor.
Parámetros	En los parámetros de entrada x e *y* quedan almacenadas las coordenadas del puntero en la nueva posición, en los ejes X e Y respectivamente, y en el parámetro *button* se almacena el código del botón presionado (izquierdo 0, centro 1, derecho 2).

void mousePressed(int x, int y, int button)

Descripción	Permite capturar el evento que se envía cada vez que se presiona alguno de los botones del *mouse*.
Valor de retorno	No retorna ningún valor.
Parámetros	En los parámetros de entrada x e *y* quedan almacenadas las coordenadas del puntero en la nueva posición, en los ejes X e Y respectivamente, y en el parámetro *button* se almacena el código del botón presionado (izquierdo 0, centro 1, derecho 2).

void mouseReleased(int x, int y, int button)

Descripción	Permite capturar el evento que se envía cada vez que se deja de presionar alguno de los botones del *mouse*.
Valor de retorno	No retorna ningún valor.
Parámetros	En los parámetros de entrada x e *y* quedan almacenadas las coordenadas del puntero en la nueva posición, en los ejes X e Y respectivamente, y en el parámetro *button* se almacena el código del botón que había sido presionado (izquierdo 0, centro 1, derecho 2).

void windowResized(int w, int h)

Descripción	Permite capturar el evento que se envía cada vez que se modifica el tamaño de la ventana de la aplicación.
Valor de retorno	No retorna ningún valor.
Parámetros	En los parámetros de entrada *w* y *h* quedan almacenadas las nuevas dimensiones de la ventana, es decir, ancho y altura respectivamente.

void dragEvent(ofDragInfo dragInfo)

Descripción	Permite capturar el evento que se envía cada vez que se arrastra un elemento nuevo a la ventana de la aplicación.
Valor de retorno	No retorna ningún valor.
Parámetros	En el parámetro de entrada *dragInfo* queda almacenada toda la información del evento, incluyendo una lista de las rutas de los archivos arrastrados y la posición en que se soltaron dentro de la ventana de la aplicación.

void gotMessage(ofMessage msg)

Descripción	Permite capturar el evento que se envía cada vez que se genera un mensaje genérico con el método *ofSendMessage*. La versión actual del sistema actualmente no genera eventos genéricos, por lo que esta función manejadora nunca se invocará.
Valor de retorno	No retorna ningún valor.
Parámetros	En el parámetro de entrada *msg* queda almacenada la información del evento genérico que provocó la notificación.

Las funciones manejadoras siguientes son unas de las más importantes del *API* del sistema ya que permiten recibir notificaciones correspondientes a gestos tridimensionales realizados con las manos por parte de los usuarios, los cuales presentan una de las principales vías de interacción con el sistema. Para el detalle sobre cada uno de estos gestos referirse al *Anexo H*.

void waveHand3DGestureDetected(int userID, Animus3DWaveHand* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto tridimensional de *wave hand* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *wave hand* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. GestureType toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. GestureStateType toma los valores definidos en *Figura 171.*

void raiseHand3DGestureDetected(int userID, Animus3DRaiseHand* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto tridimensional de *raise hand* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *raise hand* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*

void clickHand3DGestureDetected(int, Animus3DClickHand* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto tridimensional de *click* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *click hand* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*

void crossedHands3DGestureDetected (int userID, Animus3DCrossedHand* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto tridimensional de *crossed hands* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada g*Info* queda almacenada la información asociada al gesto tridimensional *click hand* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*

void psiHand3DGestureDetected(int userID, Animus3DPsiHand* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto tridimensional de *psi* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *psi* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*

void pinchHand3DGestureDetected(int userID, Animus3DPinchHand* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto tridimensional de *pinch hand* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *pinch hand* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*
progress	float	Representa el progreso del gesto.
positionVectorInWorldCoords	DataVector	Representa la posición del dedo índice en coordenadas de mundo.
positionVectorInSceneCoords	DataVector	Representa la posición del dedo índice en coordenadas de escena.
positionVectorInScreenCoords	DataVector	Representa la posición del dedo índice en coordenadas de pantalla.

void grabHand3DGestureDetected(int userID, Animus3DGrabHand* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto tridimensional de *grab hand* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *grab hand* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*

progress	float	Representa el progreso del gesto.
positionVectorInWorldCoords	DataVector	Representa la posición del centro de la mano en coordenadas de mundo.
positionVectorInSceneCoords	DataVector	Representa la posición del centro de la mano en coordenadas de escena.
positionVectorInScreenCoords	DataVector	Representa la posición del centro de la mano en coordenadas de pantalla.

void pushHand3DGestureDetected(int userID, Animus3DPushHand* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto tridimensional de *push hand* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *push hand* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*
progress	float	Representa el progreso del gesto.
pushType	PushType	Representa el tipo de *push hand* ocurrido. *PushType* toma los valores definidos en *Figura 171.*
rightHandPositionVectorInWorldCoords	DataVector	Representa la posición del centro de la mano derecha en coordenadas de mundo.
rightHandPositionVectorInSceneCoords	DataVector	Representa la posición del centro de la mano derecha en coordenadas de escena.
rightHandPositionVectorInScreenCoords	DataVector	Representa la posición del centro de la mano derecha en coordenadas de pantalla.
leftHandPositionVectorInWorldCoords	DataVector	Representa la posición del centro de la mano izquierda en coordenadas de mundo.
leftHandPositionVectorInSceneCoords	DataVector	Representa la posición del centro de la mano izquierda en coordenadas de escena.
leftHandPositionVectorInScreenCoords	DataVector	Representa la posición del centro de la mano izquierda en coordenadas de

		pantalla.

void turnHand3DGestureDetected(int userID, Animus3DTurnHand* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto tridimensional de *turn hand* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *turn hand* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*
pitch	float	Representa la inclinación del gesto.
positionVectorInWorldCoords	DataVector	Representa la posición del centro de la mano en coordenadas de mundo.
positionVectorInSceneCoords	DataVector	Representa la posición del centro de la mano en coordenadas de escena.
positionVectorInScreenCoords	DataVector	Representa la posición del centro de la mano en coordenadas de pantalla.

void zoomHand3DGestureDetected(int userID, Animus3DZoomHand* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto tridimensional de *zoom hand* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *zoom hand* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*

progress	float	Representa el progreso del gesto.
rightHandPositionVectorInWorldCoords	DataVector	Representa la posición del centro de la mano derecha en coordenadas de mundo.
rightHandPositionVectorInSceneCoords	DataVector	Representa la posición del centro de la mano derecha en coordenadas de escena.
rightHandPositionVectorInScreenCoords	DataVector	Representa la posición del centro de la mano derecha en coordenadas de pantalla.
leftHandPositionVectorInWorldCoords	DataVector	Representa la posición del centro de la mano izquierda en coordenadas de mundo.
leftHandPositionVectorInSceneCoords	DataVector	Representa la posición del centro de la mano izquierda en coordenadas de escena.
leftHandPositionVectorInScreenCoords	DataVector	Representa la posición del centro de la mano izquierda en coordenadas de pantalla.

Al igual que las anteriores, las funciones manejadoras siguientes son unas de las más importantes del *API* del sistema ya que permiten recibir notificaciones correspondientes a gestos tridimensionales realizados con los dedos por parte de los usuarios, los cuales presentan una de las principales vías de interacción con el sistema. Para el detalle sobre cada uno de estos gestos referirse al *Anexo H*.

void screenTapFinger3DGestureDetected(int userID, Animus3DScreenTapFinger* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto tridimensional de *screen tap* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *screen tap finger* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*
positionVectorInWorldCoords	DataVector	Representa la posición del dedo que efectúa el gesto en coordenadas de mundo.
positionVectorInSceneCoords	DataVector	Representa la posición del dedo que efectúa el gesto en coordenadas de

		escena.
positionVectorInScreenCoords	DataVector	Representa la posición del dedo que efectúa el gesto en coordenadas de pantalla.
directionVectorInWorldCoords	DataVector	Representa la dirección del movimiento del dedo que efectúa el gesto en coordenadas de mundo.
directionVectorInSceneCoords	DataVector	Representa la dirección del movimiento del dedo que efectúa el gesto en coordenadas de escena.
directionVectorInScreenCoords	DataVector	Representa la dirección del movimiento del dedo que efectúa el gesto en coordenadas de pantalla.

void keyTapFinger3DGestureDetected(int userID, Animus3DKeyTapFinger* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto tridimensional de *key tap* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *key tap finger* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*
positionVectorInWorldCoords	DataVector	Representa la posición del dedo que efectúa el gesto en coordenadas de mundo.
positionVectorInSceneCoords	DataVector	Representa la posición del dedo que efectúa el gesto en coordenadas de escena.
positionVectorInScreenCoords	DataVector	Representa la posición del dedo que efectúa el gesto en coordenadas de pantalla.
directionVectorInWorldCoords	DataVector	Representa la dirección del movimiento del dedo que efectúa el gesto en coordenadas de mundo.
directionVectorInSceneCoords	DataVector	Representa la dirección del movimiento del dedo que efectúa el gesto en coordenadas de escena.

directionVectorInScreenCoords	DataVector	Representa la dirección del movimiento del dedo que efectúa el gesto en coordenadas de pantalla.

void swipeFinger3DGestureDetected(int userID, Animus3DSwipeFinger* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto tridimensional de *swipe* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *swipe finger* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*
speed	float	Representa la velocidad del gesto.
directionType	SwipeDirectionType	Representa la dirección del gesto ocurrido. *SwipeDirectionType* toma los valores definidos en *Figura 171.*
FingerCountType	fingerCountType	Representa la cantidad de dedos involucrados en el gesto ocurrido. *FingerCountType* toma los valores definidos en *Figura 171.*
directionVectorInWorldCoords	DataVector	Representa la dirección del movimiento del dedo que efectúa el gesto en coordenadas de mundo.
directionVectorInSceneCoords	DataVector	Representa la dirección del movimiento del dedo que efectúa el gesto en coordenadas de escena.
directionVectorInScreenCoords	DataVector	Representa la dirección del movimiento del dedo que efectúa el gesto en coordenadas de pantalla.

void circleFinger3DGestureDetected(int userID, Animus3DCircleFinger* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto tridimensional de *circle* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *circle finger* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*
clockwiseness	Clockwiseness	Representa el sentido de firo del gesto ocurrido. Clockwisenesstoma los valores definidos en *Figura 171.*
progress	float	Representa el progreso del gesto.
radius	float	Representa el radio de giro del gesto.
sweptAngle	float	Representa el ángulo del gesto.

void zoomFinger3DGestureDetected(int userID, Animus3DZoomFinger* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto tridimensional de *zoom finger* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *zoom finger* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*
progress	float	Representa el progreso del gesto.
zoomType	ZoomType	Representa el tipo del gesto ocurrido. ZoomType toma los valores definidos en *Figura 171.*
rightHandPositionVectorInWorld Coords	DataVector	Representa la posición del centro de la mano derecha en coordenadas de mundo.
rightHandPositionVectorInScene Coords	DataVector	Representa la posición del centro de la mano derecha en coordenadas de escena.

rightHandPositionVectorInScreenCoords	DataVector	Representa la posición del centro de la mano derecha en coordenadas de pantalla.
leftHandPositionVectorInWorldCoords	DataVector	Representa la posición del centro de la mano izquierda en coordenadas de mundo.
leftHandPositionVectorInSceneCoords	DataVector	Representa la posición del centro de la mano izquierda en coordenadas de escena.
leftHandPositionVectorInScreenCoords	DataVector	Representa la posición del centro de la mano izquierda en coordenadas de pantalla.

void letterFinger3DGestureDetected(int userID, Animus3DLetterFinger* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto tridimensional de *letter finger* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *letter finger* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*

void symbolFinger3DGestureDetected(int userID, Animus3DSymbolFinger* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto tridimensional de *symbol finger* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *symbol finger* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.

state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*

Por otra parte, al igual que las anteriores, las funciones manejadoras siguientes son unas de las más importantes del *API* del sistema ya que permiten recibir notificaciones correspondientes a gestos multitáctiles realizados por parte de los usuarios, los cuales presentan una de las principales vías de interacción con el sistema. Para el detalle sobre cada uno de estos gestos referirse al *Anexo H*.

void tapTouchFingerGestureDetected(int userID, AnimusTouchTapFinger* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto multitáctil de *tap* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *touch tap* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*
id	int	Representa el identificador del *touch*.
state	TouchStateType	Representa el estado del touch ocurrido. *TouchStateType* toma los valores definidos en *Figura 171.*
hand	HandType	Representa el tipo de mano utilizada en el gesto. *HandType* toma los valores definidos en *Figura 171.*
positionVectorInWorldCoords	DataVector	Representa la posición del centro del *touch* en coordenadas de mundo.
positionVectorInSceneCoords	DataVector	Representa la posición del centro del *touch* en coordenadas de escena.
positionVectorInScreenCoords	DataVector	Representa la posición del centro del *touch* en coordenadas de pantalla.

void longTapTouchFingerGestureDetected(int userID, AnimusTouchLongTapFinger* gInf

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto multitáctil de *long tap* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *touch long tap* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*
id	int	Representa el identificador del *touch*.
hand	HandType	Representa el tipo de mano utilizada en el gesto. *HandType* toma los valores definidos en *Figura 171.*
positionVectorInWorldCoords	DataVector	Representa la posición del centro del *touch* en coordenadas de mundo.
positionVectorInSceneCoords	DataVector	Representa la posición del centro del *touch* en coordenadas de escena.
positionVectorInScreenCoords	DataVector	Representa la posición del centro del *touch* en coordenadas de pantalla.

void doubleTapTouchFingerGestureDetected(int userID, AnimusTouchDoubleTapFinger* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto multitáctil de *double tap* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *touch double tap* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*
id	int	Representa el identificador del *touch*.
hand	HandType	Representa el tipo de mano utilizada en el gesto. *HandType* toma los valores definidos en *Figura 171.*
positionVectorInWorldCoords	DataVector	Representa la posición del centro del *touch* en coordenadas de mundo.

positionVectorInSceneCoords	DataVector	Representa la posición del centro del *touch* en coordenadas de escena.
positionVectorInScreenCoords	DataVector	Representa la posición del centro del *touch* en coordenadas de pantalla.

void dragTouchFingerGestureDetected(int userID, AnimusTouchDragFinger* gInfo)

Descripción Permite capturar el evento que se envía cada vez que se detecta un gesto multitáctil de *touch drag* por parte de alguno de los usuarios reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *userID* queda almacenado el identificador del usuario para el que se notificó el evento. En el parámetro de entrada *gInfo* queda almacenada la información asociada al gesto tridimensional *touch drag tap* cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
gestureType	GestureType	Representa el tipo de gesto ocurrido. *GestureType* toma los valores definidos en la *Figura 171*.
state	GestureStateType	Representa el estado del gesto ocurrido. *GestureStateType* toma los valores definidos en *Figura 171.*
id	int	Representa el identificador del *touch*.
hand	HandType	Representa el tipo de mano utilizada en el gesto. *HandType* toma los valores definidos en *Figura 171.*
originPositionVectorInWorldCoords	DataVector	Representa la primer posición del centro del *touch* en coordenadas de mundo.
originPositionVectorInSceneCoords	DataVector	Representa la primer posición del centro del *touch* en coordenadas de escena.
originPositionVectorInScreenCoords	DataVector	Representa la primer posición del centro del *touch* en coordenadas de pantalla.
positionVectorInWorldCoords	DataVector	Representa la última posición del centro del *touch* en coordenadas de mundo.
positionVectorInSceneCoords	DataVector	Representa la última posición del centro del *touch* en coordenadas de escena.
positionVectorInScreenCoords	DataVector	Representa la última posición del centro del *touch* en coordenadas de pantalla.

Finalmente, las funciones manejadoras siguientes también forman parte crucial dentro del *API* del sistema ya que permiten recibir notificaciones correspondientes a información de actualización de usuarios

(reconocimientos, pérdidas, etc.) e información general del estado del sistema.

void userDetected(AnimusUser* uInfo)

Descripción Permite capturar el evento que se envía cada vez que un nuevo usuario es reconocidos por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *uInfo* queda almacenada la información asociada al usuario detectado, cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
id	int	Representa el identificador del usuario en el sistema.
leftHand	Hand	Representa la mano izquierda del usuario. Todos los tipos de datos referentes a *Hand* se pueden observar en la *Tabla 16*.
rightHand	Hand	Representa la mano derecha del usuario. Todos los tipos de datos referentes a *Hand* se pueden observar en la *Tabla 16*.
head	Joint	Representa la cabeza del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
leftElbow	Joint	Representa el codo izquierdo. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
leftFoot	Joint	Representa el pie izquierdo del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
leftHip	Joint	Representa la cadera izquierda del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
leftKnee	Joint	Representa la rodilla izquierda del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
leftShoulder	Joint	Representa el hombro izquierdo del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
neck	Joint	Representa el cuello del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
rightElbow	Joint	Representa el codo derecho. Todos los tipos de datos referentes a *Joint*

		se pueden observar en la *Tabla 17*.
rightFoot	Joint	Representa el pie derecho del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
rightHip	Joint	Representa la cadera derecha del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
rightKnee	Joint	Representa la rodilla derecha del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
rightShoulder	Joint	Representa el hombro derecho del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
torso	Joint	Representa el torso del cuerpo del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.

void userUpdated(AnimusUser* uInfo)

Descripción Permite capturar el evento que se envía cada vez que un usuario es actualizado en el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *uInfo* queda almacenada la información asociada al usuario actualizado, cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
id	int	Representa el identificador del usuario en el sistema.
leftHand	Hand	Representa la mano izquierda del usuario. Todos los tipos de datos referentes a *Hand* se pueden observar en la *Tabla 16*.
rightHand	Hand	Representa la mano derecha del usuario. Todos los tipos de datos referentes a *Hand* se pueden observar en la *Tabla 16*.
head	Joint	Representa la cabeza del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
leftElbow	Joint	Representa el codo izquierdo. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.

leftFoot	Joint	Representa el pie izquierdo del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
leftHip	Joint	Representa la cadera izquierda del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
leftKnee	Joint	Representa la rodilla izquierda del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
leftShoulder	Joint	Representa el hombro izquierdo del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
neck	Joint	Representa el cuello del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
rightElbow	Joint	Representa el codo derecho. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
rightFoot	Joint	Representa el pie derecho del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
rightHip	Joint	Representa la cadera derecha del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
rightKnee	Joint	Representa la rodilla derecha del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
rightShoulder	Joint	Representa el hombro derecho del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
torso	Joint	Representa el torso del cuerpo del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.

void userLost(AnimusUser* uInfo)

Descripción Permite capturar el evento que se envía cada vez que un usuario es dejado de reconocer por el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *uInfo* queda almacenada la información asociada al

usuario que se ha ido del sistema, cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
id	int	Representa el identificador del usuario en el sistema.
leftHand	Hand	Representa la mano izquierda del usuario. Todos los tipos de datos referentes a *Hand* se pueden observar en la *Tabla 16*.
rightHand	Hand	Representa la mano derecha del usuario. Todos los tipos de datos referentes a *Hand* se pueden observar en la *Tabla 16*.
head	Joint	Representa la cabeza del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
leftElbow	Joint	Representa el codo izquierdo. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
leftFoot	Joint	Representa el pie izquierdo del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
leftHip	Joint	Representa la cadera izquierda del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
leftKnee	Joint	Representa la rodilla izquierda del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
leftShoulder	Joint	Representa el hombro izquierdo del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
neck	Joint	Representa el cuello del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
rightElbow	Joint	Representa el codo derecho. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
rightFoot	Joint	Representa el pie derecho del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
rightHip	Joint	Representa la cadera derecha del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.

rightKnee	Joint	Representa la rodilla derecha del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
rightShoulder	Joint	Representa el hombro derecho del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
torso	Joint	Representa el torso del cuerpo del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.

void animusUsersInformation(list<AnimusUser*> uInfo)

Descripción Permite obtener la información de todos los usuarios registrados y activos en el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro de entrada *uInfo* queda almacenada la información asociada al usuario que se ha detectado en el sistema, cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
id	int	Representa el identificador del usuario en el sistema.
leftHand	Hand	Representa la mano izquierda del usuario. Todos los tipos de datos referentes a *Hand* se pueden observar en la *Tabla 16*.
rightHand	Hand	Representa la mano derecha del usuario. Todos los tipos de datos referentes a *Hand* se pueden observar en la *Tabla 16*.
head	Joint	Representa la cabeza del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
leftElbow	Joint	Representa el codo izquierdo. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
leftFoot	Joint	Representa el pie izquierdo del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
leftHip	Joint	Representa la cadera izquierda del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
leftKnee	Joint	Representa la rodilla izquierda del usuario. Todos los tipos de datos

		referentes a *Joint* se pueden observar en la *Tabla 17*.
leftShoulder	Joint	Representa el hombro izquierdo del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
neck	Joint	Representa el cuello del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
rightElbow	Joint	Representa el codo derecho. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
rightFoot	Joint	Representa el pie derecho del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
rightHip	Joint	Representa la cadera derecha del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
rightKnee	Joint	Representa la rodilla derecha del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
rightShoulder	Joint	Representa el hombro derecho del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.
torso	Joint	Representa el torso del cuerpo del usuario. Todos los tipos de datos referentes a *Joint* se pueden observar en la *Tabla 17*.

void animusDevicesInformation(list<AnimusDevice*> dInfo)

Descripción Permite obtener la información de todos los sensores registrados y activos en el sistema.

Valor de retorno No retorna ningún valor.

Parámetros En el parámetro *dInfo* queda almacenada la información asociada a los sensores presentes en el sistema, cuyos atributos son los siguientes:

Atributo	Tipo	Descripción
id	int	Representa el identificador del usuario en el sistema.
position	ofPoint	Representa la posición del sensor.
description	string	Representa la descripción del sensor.
deviceType	DeviceType	Representa el tipo de sensor. *DeviceType* toma los valores definidos en la *Figura 171*.

En el pseudocódigo de ejemplo incluido en la *Tabla A.3* se puede apreciar como debe ser la declaración y uso de las funciones manejadoras de eventos provista por el *API*.

```
class UserApplication {

   void SolarSystemApplication::setup() {
      ...
   }

   void SolarSystemApplication::update() {
      ...
   }

   void SolarSystemApplication::draw() {
      ...
   }

   void SolarSystemApplication::waveHand3DGestureDetected(int userid, Animus3DWaveHand* gesture) {

        cout << "Se detecto el gesto Animus3DWaveHand del usuario " << userid;

        // Hacer algo con la información contenida en el gesto
   }
}
```

Tabla 15: Pseudocódigo de ejemplo para la definición de los manejadores de eventos

En la siguiente figuras se detallan los tipos de enumerados utilizados por las firmas anteriormente descritas.

```
// Enumerados auxiliares
enum Clockwiseness {CLOCK_WISE, COUNTER_CLOCK_WISE};

// Enumerados especificos de los devices
enum DeviceType             {KINECT_DEVICE, LEAP_DEVICE, TOUCH_DEVICE};
enum PacketType             {KINECT_PACKET, LEAP_PACKET, TOUCH_PACKET};
enum FrameType              {KINECT_FRAME, LEAP_FRAME, TOUCH_FRAME};

// Enumerados de la API ANIMuS
enum EventType              {ANIMUS_USR_NEW, ANIMUS_USR_LOST, ANIMUS_USR_UPDATE,
                             ANIMUS_3D_GR_DETECTED, ANIMUS_TOUCH_GR_DETECTED,
                             ANIMUS_OBJECT_NEW, ANIMUS_OBJECT_LOST, ANIMUS_OBJECT_UPDATED,
                             ANIMUS_OBJECT_MOVED, ANIMUS_OBJECT_TOUCHED};
```

```
enum GestureType            {EMPTY,
                            // Hands
                             ANIMUS_3D_WAVE_HAND, ANIMUS_3D_RAISE_HAND, ANIMUS_3D_CLICK_HAND,
                             ANIMUS_3D_CROSSED_HAND, ANIMUS_3D_PSI_HAND, ANIMUS_3D_PINCH_HAND,
                             ANIMUS_3D_GRAB_HAND, ANIMUS_3D_ZOOM_HAND, ANIMUS_3D_PUSH_HAND,
                             ANIMUS_3D_TURN_HAND,
                             // Fingers
                             ANIMUS_3D_SCREEN_TAP_FINGER, ANIMUS_3D_KEY_TAP_FINGER,
                             ANIMUS_3D_SWIPE_FINGER, ANIMUS_3D_CIRCLE_FINGER,
                             ANIMUS_3D_ZOOM_FINGER, ANIMUS_3D_LETTER_V_FINGER,
                             ANIMUS_3D_LETTER_S_FINGER, ANIMUS_3D_LETTER_M_FINGER,
                             ANIMUS_3D_LETTER_X_FINGER, ANIMUS_3D_LETTER_Z_FINGER,
                             ANIMUS_3D_LETTER_B_FINGER, ANIMUS_3D_LETTER_A_FINGER,
                             ANIMUS_3D_LETTER_K_FINGER, ANIMUS_3D_LETTER_E_FINGER,
                             ANIMUS_3D_LETTER_F_FINGER, ANIMUS_3D_LETTER_H_FINGER,
                             ANIMUS_3D_LETTER_Y_FINGER, ANIMUS_3D_LETTER_U_FINGER,
                             ANIMUS_3D_SYMBOL_4_FINGER,
                             // Touchs
                             ANIMUS_TOUCH_TAP_FINGER, ANIMUS_TOUCH_LONG_TAP_FINGER,
                             ANIMUS_TOUCH_DOUBLE_TAP_FINGER, ANIMUS_TOUCH_DRAG_FINGER
                            };
enum GestureStateType       {ANIMUS_START_GR_STATE, ANIMUS_INVALID_GR_STATE,
                             ANIMUS_STOP_GR_STATE, ANIMUS_UPDATE_GR_STATE};
enum UserUpdateType         {ANIMUS_HANDS_UPDATE, ANIMUS_BODY_UPDATE};
enum UserStateType          {ANIMUS_NEW_USR_STATE, ANIMUS_WAITING_FOR_UPDATE,
                             ANIMUS_UPDATE_USR_STATE};

enum JointType              {ANIMUS_JOINT_HEAD, ANIMUS_JOINT_NECK,
                             ANIMUS_JOINT_LEFT_SHOULDER, ANIMUS_JOINT_RIGHT_SHOULDER,
                             ANIMUS_JOINT_LEFT_ELBOW, ANIMUS_JOINT_RIGHT_ELBOW,
                             ANIMUS_JOINT_LEFT_HAND, ANIMUS_JOINT_RIGHT_HAND,
                             ANIMUS_JOINT_TORSO, ANIMUS_JOINT_LEFT_HIP,
                             ANIMUS_JOINT_RIGHT_HIP, ANIMUS_JOINT_LEFT_KNEE,
                             ANIMUS_JOINT_RIGHT_KNEE, ANIMUS_JOINT_LEFT_FOOT,
                             ANIMUS_JOINT_RIGHT_FOOT};

 enum HandType              {ANIMUS_UNKNOWN_HAND, ANIMUS_LEFT_HAND, ANIMUS_RIGHT_HAND};

 enum FingerType            {ANIMUS_UNKNOWN_FINGER, ANIMUS_THUMB_FINGER,
                             ANIMUS_INDEX_FINGER, ANIMUS_MIDDLE_FINGER, ANIMUS_RING_FINGER,
                             ANIMUS PINKY FINGER};
 enum FingerJointsType      {ANIMUS_FINGER_MCP, ANIMUS_FINGER_PIP, ANIMUS_FINGER_DIP,
                             ANIMUS_FINGER_TIP};

enum FingerCountType        {ANIMUS_UNKNOWN_FINGER_GESTURE, ANIMUS_ONE_FINGER_GESTURE,
                             ANIMUS_TWO_FINGER_GESTURE, ANIMUS_MULTIPLE_FINGER_GESTURE};

enum TapCountType           {ANIMUS_UNKNOWN_TAP, ANIMUS_SINGLE_TAP, ANIMUS_DOUBLE_TAP,
                             ANIMUS_MULTIPLE_TAP};

enum TouchStateType         {ANIMUS_START_TCH_STATE, ANIMUS_RELEASED_TCH_STATE};

enum SwipeDirectionType     {ANIMUS_UNKNOWN_DIRECTION_SWIPE, ANIMUS_RIGHT_SWIPE,
                             ANIMUS_LEFT_SWIPE, ANIMUS_UP_SWIPE,
                             ANIMUS_DOWN_SWIPE, ANIMUS_BACKWARD_SWIPE,
                             ANIMUS_FORWARD_SWIPE};

 enum ZoomType              {ANIMUS_UNKNOWN_ZOOM, ANIMUS_ZOOM_IN, ANIMUS_ZOOM_OUT};
 enum PushType              {ANIMUS_UNKNOWN_PUSH, ANIMUS_PUSH_DOWN, ANIMUS_PUSH_UP};
```

```
// Enumerados especificos de los recognizers
enum GestureRecognizerType  {LEAP_GESTURE_RECOGNIZER, KINECT_GESTURE_RECOGNIZER,
                             TOUCH_GESTURE_RECOGNIZER};
enum AlgorithmType          {DTW_ALGORITHM, MIN_DIST_ALGORITHM, ANBC_ALGORITHM,
                             KNN_ALGORITHM, SVM_ALGORITHM};
enum TestModeType           {MANUAL_MODE, CROSS_VALIDATION, PIPELINE_TEST};
enum FeatureExtractorType   {LEAP_FEATURE_EXTRACTOR, KINECT_FEATURE_EXTRACTOR,
                             TOUCH_FEATURE_EXTRACTOR};
```

Figura 171: Valores del tipo de enumerado

En las siguientes tablas se detallan tipos de datos utilizados por las firmas anteriormente descritas.

Atributo	Tipo	Descripción
handType	HandType	Representa el tipo de mano utilizada. *HandType* toma los valores definidos en *Figura 171.*
fingersCount	int	Representa la cantidad de dedos intervinientes.
toolsCount	int	Representa la cantidad de herramientas intervinientes.
fingers	list<Finger>	Representa los dedos del usuario. Todos los tipos de datos referentes a *Finger* se pueden observar en la *Tabla 18*
normalVector	ofVec3f	Representa la normal a la mano del usuario.
directionVector	ofVec3f	Representa la dirección de la mano del usuario.
palmPositionVector	ofVec3f	Representa la posición de la mano del usuario.
palmVelocityVector	ofVec3f	Representa la velocidad de la mano del usuario.
sphereCenterVector	ofVec3f	Representa el centro de la curvatura de la mano del usuario.
sphereRadius	int	Representa el radio de la curvatura de la mano del usuario.
fingersAvgPos	ofVec3f	Representa el promedio de la posición de los dedos de la mano del usuario.

Tabla 16: Valores del tipo de datos *Hand*

Atributo	Tipo	Descripción
position	ofVec3f	Representa la posición del punto de interés.

Tabla 17: Especificación del tipo de datos *Joint*

Atributo	Tipo	Descripción
length	int	Representa el largo del dedo.
width	int	Representa el ancho del dedo.
directionVector	ofVec3f	Representa la dirección del dedo.
tipPositionVector	ofVec3f	Representa la posición de la punta del dedo.
tipVelocityVector	ofVec3f	Representa la velocidad de la punta del dedo.

Tabla 18: Valores del tipo de datos *Finger*

M.3. Configuración

Antes de dar comienzo a la ejecución de toda aplicación construida, se debe de configurar un conjunto de propiedades referente a la cantidad de sensores que serán utilizados, rangos de coordenadas y alcance, puertos y notificaciones. Dichas propiedades deben estar definidas en un archivo de configuración del tipo *XML* con nombre *animusCoreConfig.xml* y debe ser alojado bajo el directorio *resources* del directorio principal de la aplicación construida. El formato completo del archivo mencionado es el que se puede observar a continuación:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<animus>
  <devices>
    <quantity>...</quantity>
    <sceneMapping>
      <leap>
        <active>...</active>
        <deviceMinRange x="..." y="..." z="..."/>
        <deviceMaxRange x="..." y="..." z="..."/>
        <applicationMinRange x=".." y="..." z="..."/>
        <applicationMaxRange x="..." y="..." z="..."/>
      </leap>
      <singlekinect>
        <active>...</active>
        <deviceMinRange x="..." y="..." z="..."/>
        <deviceMaxRange x="..." y="..." z="..."/>
        <applicationMinRange x="..." y="..." z="..."/>
        <applicationMaxRange x="..." y="..." z="..."/>
      </singlekinect>
      <touchkinect>
        <active>...</active>
        <deviceMinRange x="..." y="..." z="..."/>
        <deviceMaxRange x="..." y="..." z="..."/>
        <applicationMinRange x="..." y="..." z="..."/>
        <applicationMaxRange x="..." y="..." z="..."/>
      </touchkinect>
    </sceneMapping>
    <screenMapping>
```

```
<leap>
  <active>...</active>
  <calibration>
    <stage>...</stage> <!-- 0-COMPLETE_CALIBRATION,
                            1-ONLY_CAMERA_CALIBRATION,
                            2-TEST_POINT, 3-REALTIME -->
    <mirror>...</mirror>
    <gridRows>...</gridRows>
    <gridColumns>...</gridColumns>
    <nProjPoints>...</nProjPoints>
    <projPoints>
      <pointList></pointList>
    </projPoints>
    <nCamPoints>...</nCamPoints>
    <camPoints></camPoints>
  </calibration>
</leap>
<singlekinect>
  <active>...</active>
  <calibration>
    <stage>...</stage> <!-- 0-COMPLETE_CALIBRATION,
                            1-ONLY_CAMERA_CALIBRATION,
                            2-TEST_POINT, 3-REALTIME -->
    <mirror>...</mirror>
    <gridRows>...</gridRows>
    <gridColumns>...</gridColumns>
    <nProjPoints>...</nProjPoints>
    <projPoints>
      <pointList></pointList>
    </projPoints>
    <nCamPoints>...</nCamPoints>
    <camPoints></camPoints>
  </calibration>
</singlekinect>
```

```
      <touchkinect>
        <active>1</active>
        <calibration>
          <stage>...</stage>
          <mirror>...</mirror>
          <gridRows>...</gridRows>
          <gridColumns>...</gridColumns>
          <nProjPoints>...</nProjPoints>
          <projPoints>
            <pointList>...</pointList>
            <point>
              <x>...</x>
              <y>...</y>
            </point>
          </projPoints>
          <nCamPoints>...</nCamPoints>
          <camPoints>
            <point>
              <x>...</x>
              <y>...</y>
              <z>...</z>
            </point>
          </camPoints>
        </calibration>
      </touchkinect>
    </screenMapping>
  </devices>

  <logging writeToFile="1">
    <!-- 0-OF_LOG_VERBOSE 1-OF_LOG_NOTICE      2-OF_LOG_WARNING >
         3-OF_LOG_ERROR   4-OF_LOG_FATAL_ERROR 5-OF_LOG_SILENT -->
    <logLevel>3</logLevel>
    <logPath>../../../log</logPath>
    <logFile>animusCore.log</logFile>
  </logging>

  <coreAccess>
    <registerdeviceport>1111</registerdeviceport>
    <analizedataport>2222</analizedataport>
  </coreAccess>

  <animusInformation>
    <userInformationInterval>5</userInformationInterval>
    <deviceInformationInterval>5</deviceInformationInterval>
  </animusInformation>

</animus>
```

M.3.1 Cantidad de sensores

La propiedad de la etiqueta *quantity* indica la cantidad de sensores totales que estará haciendo uso la aplicación desarrollada. Un ejemplo de configuración de esta propiedad, suponiendo que la aplicación hará uso de cuatro sensores, se puede observar a continuación:

```
<quantity>4</quantity>
```

M.3.2 Transformación a coordenadas de escena

La propiedades de las etiquetas *deviceMinRange*, *deviceMaxRange*, *applicationMinRange* y *applicationMaxRange* indican la correspondencia entre los rangos de alcance de los sensores y su efecto dentro de la escena donde se desarrolla la aplicación, es decir, definen la transformación de coordenadas de mundo a coordenadas de escena. A su vez, la etiqueta *active* indica si se encuentra activa dicha correspondencia para cada sensor. Un ejemplo de configuración de estas propiedades, suponiendo que la aplicación hará uso sensores del tipo *Leap Motion*, *Microsoft Kinect* y *Microsoft Kinect* para interacción multitáctil, se puede observar a continuación:

```
<sceneMapping>
  <leap>
    <active>1</active>
    <deviceMinRange x="-100" y="10" z="-100"/>
    <deviceMaxRange x="100" y="200" z="100"/>
    <applicationMinRange x="-500" y="-500" z="-500"/>
    <applicationMaxRange x="500" y="500" z="500"/>
  </leap>
  <singlekinect>
    <active>1</active>
    <deviceMinRange x="-500" y="-300" z="1400"/>
    <deviceMaxRange x="500" y="300" z="1800"/>
    <applicationMinRange x="-500" y="-500" z="-500"/>
    <applicationMaxRange x="500" y="500" z="500"/>
  </singlekinect>
  <touchkinect>
    <active>0</active>
    <deviceMinRange x="-550" y="-180" z="1100"/>
    <deviceMaxRange x="550" y="480" z="1100"/>
    <applicationMinRange x="-500" y="-500" z="-500"/>
    <applicationMaxRange x="500" y="500" z="500"/>
  </touchkinect>
</sceneMapping>
```

M.3.3 Transformación a coordenadas de pantalla

La propiedades de las etiquetas *stage*, *mirror*, *gridRows*, *gridColumns*, *nProjPoints, projPoints*, *nCamPoints*, indican la correspondencia entre las coordenadas de puntos específicos proporcionada por los sensores y su correspondencia a la coordenada de pantalla de la aplicación. A su vez, la etiqueta *"active"* indica si se encuentra activa dicha correspondencia para cada sensor. Un ejemplo de configuración de estas propiedades, suponiendo que la aplicación hará uso de sensores del tipo *Leap Motion*, *Microsoft Kinect* y *Microsoft Kinect* para interacción multitáctil, se puede observar a continuación:

```
<screenMapping>
  <leap>
    <active>0</active>
    <calibration>
      <stage>3</stage> <!-- 0-COMPLETE_CALIBRATION,
                            1-ONLY_CAMERA_CALIBRATION,
                            2-TEST_POINT, 3-REALTIME -->
      <mirror>0</mirror>
      <gridRows>0</gridRows>
      <gridColumns>0</gridColumns>
      <nProjPoints>0</nProjPoints>
      <projPoints>
        <pointList></pointList>
      </projPoints>
      <nCamPoints>0</nCamPoints>
      <camPoints></camPoints>
    </calibration>
  </leap>
  <singlekinect>
    <active>0</active>
    <calibration>
      <stage>3</stage> <!-- 0-COMPLETE_CALIBRATION,
                            1-ONLY_CAMERA_CALIBRATION,
                            2-TEST_POINT, 3-REALTIME -->
      <mirror>0</mirror>
      <gridRows>0</gridRows>
      <gridColumns>0</gridColumns>
      <nProjPoints>0</nProjPoints>
      <projPoints>
        <pointList></pointList>
      </projPoints>
      <nCamPoints>0</nCamPoints>
      <camPoints></camPoints>
    </calibration>
  </singlekinect>
```

```
    <touchkinect>
      <active>1</active>
      <calibration>
        <stage>3</stage>
        <mirror>0</mirror>
        <gridRows>1</gridRows>
        <gridColumns>1</gridColumns>
        <nProjPoints>4</nProjPoints>
        <projPoints>
          <pointList>0,5,18,23</pointList>
          <point>
            <x>0.145144850</x>
            <y>0.200892866</y>
          </point>
          <point>
            <x>0.834357381</x>
            <y>0.200892866</y>
          </point>
          <point>
            <x>0.145144850</x>
            <y>0.744419694</y>
          </point>
          <point>
            <x>0.834357381</x>
            <y>0.744419694</y>
          </point>
        </projPoints>
        <nCamPoints>4</nCamPoints>
        <camPoints>
          <point>
            <x>-278</x>
            <y>-270</y>
            <z>0</z>
          </point>
          <point>
            <x>235</x>
            <y>-258</y>
            <z>0</z>
          </point>
          <point>
            <x>-285</x>
            <y>26</y>
            <z>0</z>
          </point>
          <point>
            <x>232</x>
            <y>32</y>
            <z>0</z>
          </point>
        </camPoints>
      </calibration>
    </touchkinect>
  </screenMapping>
</devices>
```

M.3.4 Configuración de logging

La propiedad de la etiqueta *logLevel* indica el nivel de logging que estará utilizando la aplicación desarrollada. Un ejemplo de configuración de esta propiedad se puede observar a continuación:

```
<logging writeToFile="1">
  <!-- 0-OF_LOG_VERBOSE 1-OF_LOG_NOTICE      2-OF_LOG_WARNING >
       3-OF_LOG_ERROR   4-OF_LOG_FATAL_ERROR 5-OF_LOG_SILENT -->
  <logLevel>3</logLevel>
  <logPath>../../../log</logPath>
  <logFile>animusCore.log</logFile>
</logging>
```

M.3.4 Configuración de puertos

Las propiedad de las etiquetas *registerdeviceport* y *analizedataport* indican el puerto de registro de sensores al sistema y el puerto donde llegará la información sensada de la escena respectivamente. Un ejemplo de configuración de estas propiedades se puede observar a continuación:

```
<coreAccess>
  <registerdeviceport>1111</registerdeviceport>
  <analizedataport>2222</analizedataport>
</coreAccess>
```

M.3.6 Configuración de notificaciones

Las propiedades de las etiquetas *userInformationInterval* y *deviceInformationInterval* indican cada cuánto tiempo se quiere recibir información del sistema, es decir, cada cuántos segundos se quiere recibir notificaciones con información de los usuarios y de los sensores respectivamente. Un ejemplo de configuración de estas propiedades se puede observar a continuación:

```
<animusInformation>
  <userInformationInterval>5</userInformationInterval>
  <deviceInformationInterval>5</deviceInformationInterval>
</animusInformation>
```

M.4. Generales

Un punto adicional que se debe de tener en cuenta al momento de la ejecución de la aplicación desarrollada que haga uso del *framework*, es que se deben de agregar bajo el directorio *bin/release* de la aplicación todas las *dll* y carpetas que se encuentren bajo el directorio *Redist* del directorio de instalación de *NiTE2* y *OpenNI2*. Adicionalmente, bajo el directorio base de la aplicación construida se deben agregar solamente las carpetas que se encuentren bajo el directorio *Redist* del directorio de instalación de *NiTE2* y *OpenNI2*.

Referirse al *Anexo K* de configuración del entorno de desarrollo para un detalle sobre los pasos requeridos para la instalación de las diferentes librerías y módulos necesarios para el correcto funcionamiento de la solución.

Anexo N: Aspectos de implementación de la aplicación de prototipo

Como parte del presente trabajo se desarrolló una aplicación interactiva a modo de prototipo, con el fin de ilustrar las diferentes características y funcionalidades brindadas por el *framework* generado. La aplicación ya fue descrita en detalle desde el punto de vista del usuario en el *Capítulo 6* del cuerpo principal del presente documento. En este anexo, se describe la aplicación desarrollada desde el punto de vista de la programación, brindando los detalles inherentes al desarrollo propiamente dicho, incluyendo los componentes principales utilizados, la implementación del flujo principal de ejecución mediante máquinas de estados, etc. Cabe mencionar que los puntos aquí descritos son exclusivamente referentes a los aspectos de implementación de la aplicación desarrollada (componentes propias de la aplicación, flujo de ejecución mediante máquina de estados, elementos gráficos, configuración, etc). No se pretende detallar de forma general la forma correcta de desarrollar una aplicación sobre el *framework* generado; en este caso referirse al *Anexo N* donde se presenta la guía de programación del *framework*.

N.1. Componentes principales

En esta sección se detalla los componentes principales que dan soporte a la aplicación desarrollada, indicando qué representan y qué rol cumplen dentro de la aplicación, así como también, los atributos principales que almacenan con el fin de asegurar su correcto funcionamiento.

N.1.1. *SolarSystemApplication*

Este componente es la clase principal de la aplicación construida. Naturalmente, dado que la aplicación hace uso del *framework* desarrollado, esta clase implementa la interfaz de servicios *IAnimusApplication* expuesta por el *framework*, con el objetivo de que la aplicación pueda recibir los diferentes eventos que el *framework* envía. Más específicamente, mediante esta interfaz es posible conocer los diferentes usuarios que se encuentran en la escena y sus posiciones, los diferentes gestos realizados por los usuarios, tanto multitáctiles como tridimensionales, e información general del estado del sistema.

En cuanto a los atributos y propiedades principales de este componente, en la *Tabla 19* se detallan las más relevantes; donde, se incluyen diferentes atributos que permiten controlar el estado general de la aplicación, así como también, atributos que contienen información específica de los usuarios que están siendo reconocidos por el sistema. En particular, cada usuario tiene su propia máquina de estados (atributo *appStateMachine*), su propio planeta seleccionado (atributo *selectedPlanet*), etc., razón por la cual cualquier de estos atributos, al igual que otros, son arreglos indexados por el identificador en el sistema para cada usuario, donde *APP_MAX_USERS* es la cantidad máxima de usuarios posibles en la interacción.

Atributo	Tipo	Descripción
appStateMachine	AppStateMachine*[APP_MAX_USERS]	Mantiene las máquinas de estados de los diferentes usuarios reconocidos por el sistema.
selectedPlanet	Planet*[APP_MAX_USERS]	Mantiene el planeta que tiene seleccionado cada uno de los usuarios reconocidos por el sistema (en caso de haberlo).

selectionSphere	ofSpherePrimitive*[APP_MAX_USERS]	Mantiene el elemento de selección tridimensional para cada uno de los usuarios reconocidos por el sistema.
selectionProgressBar	ofxProgressBar*[APP_MAX_USERS]	Mantiene la barra de progreso de selección tridimensional para cada uno de los usuarios reconocidos por el sistema.
gestureMode	GestureMode[APP_MAX_USERS]	Mantiene los tipos de interacción para los diferentes usuarios reconocidos por el sistema. Donde GestureMode puede tomar alguno de los valores siguientes {NORMAL_MODE, GRAB_MODE, PINCH_MODE, TOUCH_TAP_MODE, TOUCH_DOUBLE_TAP_MODE, RIGHT_SWIPE, LEFT_SWIPE, ZOOM_IN, ZOOM_OUT}
userColor	ofColor[APP_MAX_USERS]	Mantiene los colores asignados a de selección para cada uno de los usuarios reconocidos por el sistema.
barProgress	int[APP_MAX_USERS]	Mantiene el progreso de selección tridimensional para cada uno de los usuarios reconocidos por el sistema.
tapTouchPoint	AppCircle*[APP_MAX_USERS]	Mantiene el elemento de selección multitáctil para cada uno de los usuarios reconocidos por el sistema.
solarSystem	SolarSystem*	Permite representar el Sistema Solar (referirse a componente *SolarSystem*).
originalCamPosition	ofVec3f	Mantiene la posición original de la cámara de la escena.
originalLookAtCamPosition	ofVec3f	Mantiene la posición original hacia donde apunta la cámara de la escena.
cam	ofEasyCam*	Cámara de la escena.
orthoCam	ofCamera	Cámara de la selección de los usuarios.

Tabla 19: Propiedades principales de la clase *SolarSystemApplication*

A su vez, dado que la aplicación fue desarrollada utilizando el conjunto de librerías *openFrameworks* (referirse a *Sección 2.2.3* del cuerpo principal del documento), este componente incluye el método *setup* de inicialización de la aplicación, y los métodos de actualización, *update*, y de dibujado, *draw*, que se ejecutan en cada iteración durante la ejecución de la aplicación interactiva con el fin de actualizar los diferentes elementos presentes en la escena y dibujarlos en pantalla respectivamente. Esto permite, por ejemplo, realizar una determinada acción (modificando uno o más de los atributos presentados en la *Tabla 19*) acorde al estado del sistema una vez se recibe una notificación de gesto reconocido por parte de un usuario. Las diferentes acciones que este componente debe realizar con los elementos de la escena ante las

diferentes notificaciones están regidas por una máquina de estados que controla el flujo general de ejecución de la aplicación. Esta máquina de estados será descrita más adelante como parte de este mismo anexo.

N.1.2. *Main*

Este componente representa el punto de entrada principal de la aplicación, por lo que contiene el código de configuración del *framework* (referirse al *Anexo N*), incluyendo el registro de los gestos que la aplicación aceptará desde el sistema y su posterior inicialización, así como también, la inicialización de la aplicación particular.

N.1.3. *Planet*

Este componente permite representar tanto un planeta del Sistema Solar como al Sol con respecto al cual los demás planetas rotan y se trasladan. Por esto, como parte de la inicialización de la aplicación, se instancia cierta cantidad de componentes de este tipo dependiendo de la cantidad de planetas que se quieran representar (en este caso particular, la cantidad de planetas de la aplicación está dada por la cantidad de planetas del Sistema Solar y el Sol). Este componente incluye los diferentes atributos de un planeta, como por ejemplo su tamaño, su color, su velocidad de traslación, su velocidad de rotación, etc. A su vez, cada planeta cuenta con una lista de imágenes descriptivas del planeta en cuestión. En la *Tabla 20* se pueden apreciar estas y otras de las propiedades principales de este componente.

Atributo	Tipo	Descripción
sphere	ofSpherePrimitive	Elemento primitivo de esfera que da forma al planeta.
texture	ofImage	Archivo de textura del planeta.
drawOrbit	bool	Indica si se está dibujando la órbita del planeta.
originalSize	float	Indica el tamaño original del planeta.
size	float	Indica el tamaño actual del planeta.
originalPosition	ofVec3f	Indica la posición original del planeta.
position	ofVec3f	Indica la posición actual del planeta.
orbitRadius	float	Indica el radio de órbita del planeta.
trAngle	float	Indica el ángulo de traslación del planeta.
trVelocity	float	Indica la velocidad de translación del planeta.
originalRotAngle	float	Indica el ángulo de rotación original del planeta.
rotAngle	float	Indica el ángulo de rotación actual del planeta.
rotVelocity	float	Indica la velocidad de rotación original del planeta.
selectedRotVelocity	float	Indica la velocidad de rotación actual del planeta.
hasRing	bool	Indica si el planeta posee anillos.
selectedBy	int	Indica qué usuario lo ha seleccionado.
name	string	Nombre del planeta.
description	string	Descripción general del planeta.
planetImages	list<PlanetImage*>	Lista de imágenes descriptivas asociadas al planeta.

Tabla 20: Propiedades principales de la clase *Planet*

N.1.4. *SolarSystem*

Este componente es el que permite representar al Sistema Solar de forma global, conteniendo todos los planetas que incluye y el Sol, así como también, diferentes parámetros que permiten llevar el rastro de los planetas seleccionados, controlar el modo de movimiento (planetas trasladándose y rotando o estáticos), etc. Al inicializar la aplicación se instancia este componente, lo cual crea todas las instancias de los planetas y el Sol, cada uno con las características según lo descrito en el componente *Planet*. En la *Tabla 21* se listan las propiedades principales de esta clase.

Atributo	Tipo	Descripción
planets	vector<Planet*>	Mantiene todos los planetas que componen el Sistema Solar.
drawOrbits	bool	Indica si se muestran o no las órbitas de los planetas.
selectedPlanets	int[APP_MAX_USERS]	Mantiene el identificador de los planetas seleccionados, donde APP_MAX_USERS indica la cantidad posibles de usuarios interactuando.
movementMode	Enum {MOVING_SS_MODE, STATIC_SS_MODE}	Indica el tipo de movimiento de los planetas del Sistema Solar (en movimiento o rotando).

Tabla 21: Propiedades principales de la clase *SolarSystem*

N.1.5. *PlanetImage*

La clase *PlanetImage* permite representar imágenes con información para cada planeta particular. Una vez que un planeta es seleccionado mediante un gesto determinado es posible desplegar una lista de imágenes asociadas, y mediante otro gesto se puede observar la descripción de la información que cada imagen contiene. En la *Tabla 22* se detallan las propiedades principales de esta clase.

Atributo	Tipo	Descripción
image	ofImage	Archivo de imagen.
position	ofVec3f	Posición donde se debe desplegar la imagen.
drawingPosition	ofVec3f	Posición relativa a la escena donde se debe desplegar la imagen.
height	int	Largo de la imagen.
width	int	Ancho de la imagen.
information	string	Información correspondiente a la imagen.
informationDisplayed	bool	Indica si la información se está mostrando.

Tabla 22: Propiedades principales de la clase *PlanetImage*

N.1.6. *AppStateMachine* y *AppState*

Como ya se mencionó, el flujo general de ejecución de la aplicación *Solar System Application* para cada usuario está controlado por una máquina de estados, en la que dependiendo del estado en que se esté y del evento recibido (gesto multitáctil o tridimensional) se pasa a un estado diferente. En la *Figura 172* se

ilustra el diagrama conceptual de la máquina de estados construída para esta aplicación particular.

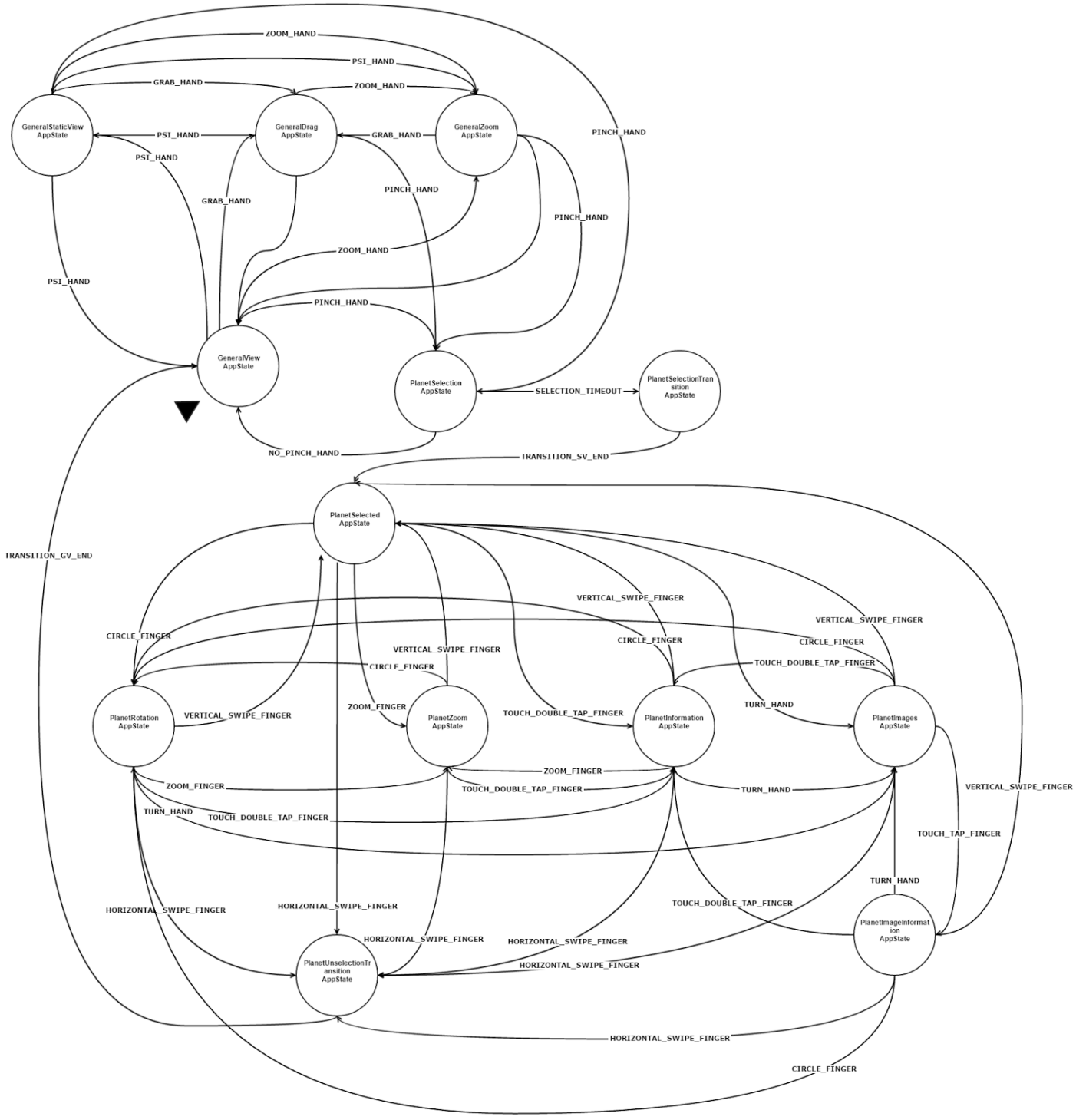


Figura 172: Máquina de estados de la aplicación

Así, por ejemplo, si estamos en el estado de *General View App* (vista inicial por defecto de la aplicación) y se recibe un gesto de *pinch*, se pasa al estado *Planet Selection*, en el que se esperará que el usuario seleccione un elemento para pasar al siguiente estado. La anterior es la representación conceptual de la máquina de estados a implementar; se detallarán las características de cada uno de los estados más adelante en esta misma subsección.

Las clases *AppState* y *AppStateMachine* son los componente principales para la representación de la máquina de estados en la que se basa el flujo de ejecución de la aplicación. Por un lado, *AppState* permite

representar básicamente los estados y las diferentes transiciones para cada uno de ellos, es decir, define a qué estado se pasa si ocurre un determinado evento. La *Tabla 23* lista las propiedades principales de esta clase.

Atributo	Tipo	Descripción
id	int	Identificador del estado.
name	string	Nombre del estado.
transitions	map<AppStateTransition,AppState*>	Mapa de transiciones, dada una transacción para un estado determina el siguiente estado al que se debe dirigir.

Tabla 23: Propiedades principales de la clase *AppState*

El atributo principal es *transitions*, el cual permite definir las transiciones a los demás estados ante la recepción de un evento particular. Los posibles eventos que pueden ocurrir en el sistema están representados por la estructura *AppStateTransition*, y pueden ocurrir o bien por la detección de un gesto o bien por otros sucesos específicos (temporizadores de tiempo, estados particulares de elementos, etc). En la *Tabla 24* se listan los valores posibles para las transiciones específicas que se pueden dar, el evento que la genera y su descripción. Referirse a la *Sección 6.3* del *Capítulo 6* del cuerpo principal del informe para el detalle de los diferentes gestos listados.

Valor	Evento	Descripción
PINCH_HAND	Gesto *pinch*	Transición que ocurre cuando se detecta el gesto de *pinch*.
GRAB_HAND	Gesto *grab*	Transición que ocurre cuando se detecta el gesto de *grab*.
PSI_HAND	Gesto *psi hand*	Transición que ocurre cuando se detecta el gesto *psi hand*.
ZOOM_HAND	Gesto *zoom hand*	Transición que ocurre cuando se detecta el gesto *zoom hand*.
TURN_HAND	Gesto *turn hand*	Transición que ocurre cuando se detecta el gesto *turn hand*.
HORIZONTAL_SWIPE_FINGER	Gesto *horizontal finger swipe*	Transición que ocurre cuando se detecta el gesto *horizontal finger swipe*.
VERTICAL_SWIPE_FINGER	Gesto *vertical finger swipe*	Transición que ocurre cuando se detecta el gesto *vertical finger swipe*.
CIRCLE_FINGER	Gesto *finger circle*	Transición que ocurre cuando se detecta el gesto *finger circle*.
ZOOM_FINGER	Gesto *finger zoom*	Transición que ocurre cuando se detecta el gesto *finger zoom*.
TOUCH_TAP_FINGER	Gesto *tap*	Transición que ocurre cuando se detecta el gesto *tap*.
TOUCH_DOUBLE_TAP_FINGER	Gesto *double tap*	Transición que ocurre cuando se detecta el gesto *double tap.*
SELECTION_TIMEOUT	-	Transición que ocurre cuando se está detectando el gesto *pinch* luego de un

		intervalo de tiempo. Determina la selección del objeto bajo la posición del gesto y moverse a un nuevo estado.
TRANSITION_SV_END	-	Transición que ocurre luego que un objeto de la escena es seleccionado. Determina el movimiento de la cámara de la escena hasta centrar el objeto al usuario que lo seleccionó y moverse a un nuevo estado.
TRANSITION_GV_END	-	Transición que ocurre luego que un objeto de la escena es deseleccionado. Determina el movimiento de la cámara de la escena hasta volver el objeto a su posición original y moverse a un nuevo estado.
NO_PINCH_HAND	-	Transición que ocurre cuando no se detecta el gesto *pinch* por un intervalo de tiempo. Determina moverse a un nuevo estado.

Tabla 24: Transiciones de la máquina de estados

Existen diferentes clases específicas que descienden de la clase base *AppState* y permiten representar cada posible estado particular de los existentes en la máquina de estados. Cada uno de estos estados específicos define a qué estado se pasa ante un cierto evento de los listados en la *Tabla 24* mediante el atributo *transitions*. Así, por ejemplo, dado el *AppState* específico para representar el estado de vista general (denominado *GeneralStaticViewAppState*, tal como el incluido en la máquina de estados de la *Figura 172*), para el cual se define que ante un evento de *pinch* se pasa a un estado de selección de planeta (denominado *PlanetSelectionAppState)*, también representado mediante un *AppState* específico*.* En la *Tabla 25* se pueden apreciar las diferentes clases específicas descendiente de la clase base *AppState*, cada una de las cuales permite representar los diferentes estados de la máquina de estados, junto con una descripción del comportamiento de la aplicación para cada uno de ellos.

Estado (subclase)	Descripción
GeneralStaticViewAppState	Este estado representa a la escena de forma estática, en la que no se observa rotación ni traslación de los planeta, sino que están fijos en determinada posición.
GeneralViewAppState	Este estado representa a la escena en su forma original, donde los diferentes planetas se trasladan y rotan con respecto al sol.
GeneralZoomAppState	Este estado representa el momento en el que se detecta el gesto de *zoom in* o *zoom out* en la escena global. Se puede visualizar cómo el plano de la escena es acercado o alejado respectivamente.
GeneralDragAppState	Este estado representa el momento en el que se detecta el gesto de *grab* en la escena global. Se puede visualizar cómo el plano de la escena es desplazado acorde al movimiento del gesto.
PlanetSelectionAppState	Este estado representa el momento en el que se realiza el gesto *pinch,* visualizando el selector que permite seleccionar un planeta para mostrarlo en el área específica del usuario que corresponda.

PlanetSelectionTransitionAppState	Este estado representa la animación de transición desde que se detectó la selección de un planeta hasta que se muestra en el área específica del usuario que corresponda. En definitiva, este estado sirve para quitar el planeta del Sistema Solar y colocarlo cerca del usuario mediante una animación acorde a la interacción.
PlanetUnselectionTransitionAppState	Este estado representa la animación de transición desde que se indicó la deselección de un planeta previamente seleccionado hasta que se muestra nuevamente como parte del Sistema Solar. En definitiva, este estado sirve para incluir el planeta nuevamente en el Sistema Solar global mediante una animación, luego de que el usuario interactuara con él en el área específica que corresponde.
PlanetSelectedAppState	Este estado representa el momento en que finaliza la animación de transición de un planeta seleccionado, y se muestra en el área específica para el usuario que corresponda.
PlanetRotationAppState	Este estado representa el momento en el que al tener un planeta seleccionado se realiza el gesto de *circle* a modo de poner a rotar al planeta entorno a su eje.
PlanetZoomAppState	Este estado representa el momento en el que al tener un planeta seleccionado se realiza el gesto *zoom in* y *zoom out* del planeta en cuestión. Se puede visualizar como el planeta es acercado o alejado respectivamente.
PlanetImagesAppState	Este estado representa el momento en el que al tener un planeta seleccionado se le realiza el gesto de *turn* a modo de poder listar las imágenes asociadas al planeta.
PlanetInformationAppState	Este estado representa el momento en el que al tener un planeta seleccionado se realiza el gesto de *double tap* a modo de desplegar su información general.
PlanetImageInformationAppState	Este estado representa el momento en el que al tener desplegada la lista de imágenes asociadas a un planeta seleccionado, se realiza el gesto *tap* sobre algunas de dichas imágenes para observar el detalle de información correspondiente.

Tabla 25: Diferentes estados de la aplicación

Por otro lado, la clase *AppStateMachine* permite definir la máquina de estados para cada usuario en el sistema, especificando un estado inicial, un estado actual y un estado previo al actual (de los estados incluidos en la *Tabla 25*). De esta forma, dado que los diferentes estados específicos ya contienen sus propias transiciones, es posible representar la máquina de estados completa que controla el flujo general de la aplicación. El componente *AppStateMachine* es el encargado de instanciar los diferentes componentes específicos que permiten representar los estados de la máquina de estados, especificando acordemente cuál de ellos será el estado inicial. Los valores de estado actual y previo se irán cargando adecuadamente en la medida que se reciben eventos y se ejecutan transiciones sobre la máquina de estados definida. La *Tabla 26* resumen las propiedades principales del componente *AppStateMachine*.

Atributo	Valor	Descripción
currentState	AppState*	Estado actual de la máquina de estados.

previousState	AppState*	Estado previo de la máquina de estados.
initialState	AppState*	Estado inicial de la máquina de estados.
user	int	Identificador del usuario al que pertenece la máquina de estados.

Tabla 26: Propiedades principales de la clase *AppStateMachine*

N.2. Generales

A modo de mejorar la estética de la aplicación construida se aplicó el uso de *addons* dentro del que se destacan los utilizados para generar la técnica de *skybox*. Mediante dicha técnica, se trata de representar a la escena en un entorno 3D y no plano como si se cargara como fondo una simple imagen, obteniendo así un realismo mejor de profundidad al interactuar con la aplicación. Por otro lado, si bien se realizaron investigaciones para cargar *shaders* y tener efectos visuales más llamativos, no se obtuvieron resultados positivos con el *hardware* con el que se cuenta, y en sí profundizar en el tema escapa a los objetivos del proyecto.

También se deben configurar acordemente los ficheros de configuración del sistema para que la aplicación pueda consumir los servicios del *framework* sin inconvenientes. En el caso de *Solar System Application*, se utilizan cuatro sensores; por un lado dos sensores *Leap Motion* y por otro lado dos sensores *Microsoft Kinect*. Para un detalle de los campos específicos que se deben configurar en los archivos de configuración del sistema referirse al *Anexo N*.

Glosario

A

Addon	Programa del tipo extensión que funciona anexado a otro programa que sirve para incrementar o complementar las funcionalidades del correspondiente programa.
API	Del inglés *"Application Programming Interface",* es un conjunto de métodos que ofrece cierta biblioteca para ser consumidos por otro programa.
Aprendizaje automático **Aprendizaje por computador** **Aprendizaje de máquina**	Área de la inteligencia artificial que se ocupa del desarrollo de algoritmos que mejoran su *performance* a través de la experiencia.

B

Body-tracking	Técnica de seguimiento del esqueleto de un usuario.
Bugs	Error o fallo de un programa.

C

Capacidad de cómputo	Cantidad de procesamiento que un computador puede realizar por unidad de tiempo.

D

Distribución *Gaussiana*	En estadística y probabilidad se llama distribución normal, distribución de *Gauss* o distribución *Gaussiana*, a una de las distribuciones de probabilidad de variable continua que con más frecuencia aparece aproximada en fenómenos reales.

E

Estado *Beta*	Representa generalmente la primera versión completa de un programa que es posible que sea inestable, es decir, que contenga fallas.

F

Fiducial	Objeto utilizado como marcador de referencia para la observación que puede ser usado como punto de referencia o de medida.
FOV	Del inglés *"Field Of View"*, es el campo de visión observable.
FPS	Del inglés *"Frames per second"*, medida de imágenes por segundo en

	películas, animaciones o videojuegos.
Frame	Contenido de una pantalla de datos.
Framework	Conjunto de bibliotecas, componentes y clases que componen un diseño reutilizable que facilita y agiliza el desarrollo ágil, seguro y escalable de una aplicación.

G

GPU	Del inglés "*Graphic Processing Unit*", es un procesador dedicado al procesamiento de gráficos u operaciones de coma flotante, para aligerar la carga de trabajo del procesador central.

H

Host	Computador conectado a una red, que provee y utiliza servicios de sde/hacia dicha red.

I

IDE	Del inglés *"Integrated Development Environment",* es una aplicación informática que proporciona servicios integrales para facilitarle al desarrollador el desarrollo de *software*.
Imagen estereoscópica	Imagen conformada a partir de dos imágenes tomadas desde puntos de vista ligeramente diferentes, las cuales son enviadas de forma separada a cada ojo, haciendo que el cerebro crea que se trata de una imagen tridimensional, percibiendo así, la sensación de profundidad, cercanía o lejanía de los objetos de la escena.
***Interfaz* de usuario**	Es el medio con que el usuario puede comunicarse con un computador o dispositivo y comprende todos los puntos de contacto entre ambos.
Intranet	Red de conexiones en un ámbito doméstico entre dos o más computadores basándose en los protocolo de Internet.
IP	Del inglés *"Internet Protocol"*, es un número que identifica de manera lógica y jerárquica, a una interfaz en red de un dispositivo.

L

LED	*Light-Emitting Diode*, es una lámpara de estado sólido que usa diodos emisores de luz como fuente lumínica.
Ley de *Moore*	Ley formulada por *Gordon Moore* en los años sesenta que expresa que la cantidad de transistores que se pueden incluir en un circuito integrado se duplica cada 18 meses, lo cual aumenta la capacidad de cómputo y decrementa su costo. En definitiva, cada vez existen dispositivos más baratos, más pequeños y más potentes.
Biblioteca de *software*	Conjunto de datos y de código de programación que se utiliza para desarrollar programas de *software* y aplicaciones. Está diseñado para

ayudar tanto al programador y al compilador de lenguaje de programación en la construcción y ejecución de *software*.

M

Middleware *Software* que asiste a una aplicación para interactuar o comunicarse con otras aplicaciones pasando los datos entre ellas.

N

Nvidia Nvidia Corporation es una empresa multinacional que se ha convertido en uno de los principales proveedores de unidades de procesamiento gráfico GPU y conjuntos de chips usados en tarjetas de gráficos en videoconsolas y placas base de computadoras personales.
Nvidia produce las GPUs de la serie GeForce para videojuegos, la serie NVIDIA Quadro de diseño asistido por computador y la creación de contenido digital en las estaciones de trabajo, y la serie de circuitos integrados nForce para placas base.

O

Opensource El *software* de código abierto es aquel cuyo código fuente y otros derechos que normalmente son exclusivos para quienes poseen los derechos de autor, son publicados bajo una licencia de software compatible con la Open Source Definition o forman parte del dominio público. Esto permite a los usuarios utilizar, cambiar, mejorar el software y re distribuirlo, ya sea en su versión modificada, o en su versión original.

P

Píxel Menor unidad homogénea en color que forma parte de una imagen digital, ya sea esta una fotografía, un fotograma de vídeo o un gráfico.

Plugin Análogo a *Addon*.

R

Release Corresponde a una nueva versión distribuible de un programa que incluye nuevas funcionalidades.

RGB *Red, Green, Blue,* es la composición del color en términos de la intensidad de los colores primarios de la luz.

S

SDK Del inglés *"Software Development Kit",* es un conjunto de herramientas de desarrollo de software que le permite al programador crear aplicaciones para un sistema concreto.

Sensor Dispositivo que capta magnitudes físicas como puede ser variaciones de luz, temperatura, sonido, etc. u otras alteraciones de su entorno.

Serialización Proceso de codificación de un objeto en un medio de almacenamiento como puede ser un archivo, o un *buffer* de memoria con el fin de transmitirlo desde un origen a un destino a través de una conexión en red como una serie de *bytes* o en un formato humanamente más legible como *XML* o *JSON*, entre otros. La serie de *bytes* o el formato luego pueden ser usados para crear un nuevo objeto que es idéntico en todo al origina en el destino de la conexión.

Sistema de coordenadas cartesiano Las coordenadas cartesianas se usan para definir un sistema cartesiano o sistema de referencia respecto ya sea a un solo eje (línea recta), respecto a dos ejes (un plano) o respecto a tres ejes (en el espacio), perpendiculares entre sí, que se cortan en un punto llamado origen de coordenadas. Las coordenadas cartesianas se definen como la distancia al origen de las proyecciones ortogonales de un punto dado sobre cada uno de los ejes.

T

TCP Del inglés *"Transfer Control Protocol",* es un protocolo de red de capa de transporte que permite el envío de datagramas a través dela red estableciendo una conexión entre las extremos. Este protocolo posee control de flujo, control de congestión asegurando que los paquetes lleguen correctamente a destino.

U

UDP Del inglés *"User Datagram Protocol"*, es un protocolo de red de capa de transporte que permite el envío de datagramas a través de la red sin que se haya establecido previamente una conexión. Este protocolo no posee control de flujo, por lo que los paquetes pueden adelantarse unos a otros; ni tampoco se sabe si llegan correctamente, ya que no hay confirmación de entrega y/o recepción.

USB Del inglés *"Universal Serial Bus"*, es un *bus* estándar industrial que define los cables, conectores y protocolos usados para conectar, comunicar y proveer de alimentación eléctrica entre computadoras, periféricos y dispositivos electrónicos.

V

VGA Del inglés *"Video Graphics Array",* se utiliza para denominar a una pantalla de computadora analógica estándar.

Video mapping Técnica que consiste en proyectar una animación o imágenes sobre superficies reales, generalmente inanimadas, para conseguir un efecto

	artístico basado en los movimientos que crea la animación sobre dichas superficies.
Visión por computadora	Disciplina que incluye métodos para adquirir, procesar, analizar y comprender las imágenes del mundo real con el fin de producir información numérica o simbólica para que puedan ser tratados por un computador.

W

Wrapper	Entidad que encapsula y oculta la complejidad subyacente de otra entidad por medio de interfaces bien definidas.

X

Xml	Del inglés *"eXtensible Markup Language",* es un lenguaje de etiquetas ampliable o extensible desarrollado por el *World Wide Web Consortium (W3C)*.

Repositorio de código

El control de código se realizó mediante SVN utilizando el servicio XP-Dev.com para alojar los diferentes componentes de la solución. La URL correspondiente al código del proyecto es la siguiente: https://xp-dev.com/svn/animusproject/branches/animus_09082015/.

En la *Figura 171* se puede observar la estructura de directorios del repositorio completo. Cada carpeta se corresponde a una capa de las que conforman la solución. Los directorios *animusCommons*, *animusCore*, *animusDataAcess* se corresponden al código de los tipos de datos comunes, y las capas *Core* y *DataAccess* respectivamente. El directorio *solarSystemApplication* contiene todo el código correspondiente a la aplicación de prototipo desarrollada, el directorio *CalibrationTool* contiene el código de la herramienta de calibración y en *GestureTrainers* se puede encontrar el código referente a la herramienta de entrenamiento de gestos. Por otro lado, se encuentran los directorios de los proyectos *GRT* y *OF* de los cuales luego se genera las bibliotecas que son utilizadas por el presente proyecto, así como también los zips correspondientes a las versiones de *OpenNI* y *NiTE* utilizadas. Por último, *animus.sln* es el archivo solución *Visual Studio 2012* que incluye todos los componentes necesarios para ejecutar el sistema.

Figura 173: Estructura de repositorio de código

Referencias bibliográficas

[1] Jakob Nielsen, «10 Heuristics for User Interface Design», *Nielsen Norman Group*. [En línea]. Disponible en: https://www.nngroup.com/articles/ten-usability-heuristics/. [Accedido: 03-nov-2016].

[2] T. K. Bruce D. Lucas, «An iterative image registration technique with an application to stereo vision», *An Iterative Image Registration Technique with an Application to Stereo Vision*. [En línea]. Disponible en: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.77.1570&rep=rep1&type=pdf. [Accedido: 01-nov-2016].

[3] The Apache foundation, «Apache license», *Licenses*. [En línea]. Disponible en: http://www.apache.org/licenses/LICENSE-2.0. [Accedido: 01-nov-2016].

[4] J. A.-M. Jose Antonio Diego-Mas, «Aplicación de Kinect para la evaluación de la ergonomía de puestos de trabajo». [En línea]. Disponible en: http://www.ergonautas.upv.es/lab/articulos/kinect.htm. [Accedido: 04-dic-2016].

[5] BBC News, «Apple buys motion sensor maker PrimeSense», *BBC News*, 25-nov-2013.

[6] Eva Mosquera Rodríguez, «Así puede manipularte una casa inteligente», *ELMUNDO*, 26-mar-2016. [En línea]. Disponible en: http://www.elmundo.es/tecnologia/2016/03/26/563a51cc46163f17028b4644.html. [Accedido: 11-dic-2016].

[7] Ronald T. Azuma, «A survey of augmented reality». [En línea]. Disponible en: http://www.cs.unc.edu/~azuma/ARpresence.pdf. [Accedido: 01-nov-2016].

[8] Oliver Kreylos, «Augmented Reality Sandbox». [En línea]. Disponible en: http://idav.ucdavis.edu/~okreylos/ResDev/SARndbox/index.html. [Accedido: 20-nov-2016].

[9] G. D. Stephan Schlogl y Nikiforos Karamanis, «A wizard of Oz prototyping framework», *WebWOZ: A Wizard of Oz Prototyping Framework*. [En línea]. Disponible en: https://www.cs.tcd.ie/Gavin.Doherty/papers/EICS-WebWOZ.pdf. [Accedido: 25-nov-2016].

[10] J. H. Johannes Schoening *et al.*, «Background modeling using mixture of Gaussians», *Building Interactive Multi-touch Surfaces*. [En línea]. Disponible en: http://www.jonathanhook.co.uk/publications/multitouchguide.pdf. [Accedido: 10-nov-2016].

[11] Boost, «Boost serialization». [En línea]. Disponible en: http://www.boost.org/doc/libs/1_62_0/libs/serialization/doc/index.html. [Accedido: 13-nov-2016].

[12] Boost, «Boost software license». [En línea]. Disponible en: http://www.boost.org/users/license.html. [Accedido: 01-nov-2016].

[13] María Amparo Gaitán J., «Casas inteligentes». [En línea]. Disponible en: http://www.infochannel.info/casas-inteligentes-el-nuevo-negocio-de-los-canales-0. [Accedido: 11-dic-2016].

[14] Cinder, «Cinder». [En línea]. Disponible en: https://libcinder.org/. [Accedido: 01-nov-2016].

[15] R. D. Andrew Bragdon y M. R. M. Ken Hinckley, «Code Space». [En línea]. Disponible en: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/codespace.pdf. [Accedido: 03-dic-2016].

[16] Stefan Marti, *Comandos put-that-there*.

[17] Leap Motion, «Comfortable use of your Leap Motion Controller», *Leap Motion Support*. [En línea]. Disponible en: http://support.leapmotion.com/hc/en-us/articles/223782288-Comfortable-use-of-your-Leap-Motion-Controller. [Accedido: 03-nov-2016].

[18] M. O. C. Friedemann Mattern y Jesús Lorés Vidal, «Computación Ubicua, la tendencia hacia la informatización y conexión en red de todas las cosas». [En línea]. Disponible en: http://www.lsi.us.es/~ortega/domotica/novaticaUbicua2.pdf. [Accedido: 01-nov-2016].

[19] «Cramer's Rule». [En línea]. Disponible en: http://www.purplemath.com/modules/cramers.htm. [Accedido: 24-abr-2017].

[20] Leap Motion, «Designing intuitive applications». [En línea]. Disponible en: https://developer.leapmotion.com/articles/designing-intuitive-applications. [Accedido: 01-nov-2016].

[21] I. J. O. Ing. Juan Eduardo Salvatore y Ing. Martin Morales, «Detección de objetos utilizando el sensor Kinect», *Detección de objetos utilizando el sensor Kinect*. [En línea]. Disponible en: http://www.laccei.org/LACCEI2014-Guayaquil/RefereedPapers/RP178.pdf. [Accedido: 13-nov-2016].

[22] Andreas Farner, «DP-100 Connected Speech Recognition System», *Report on Speech & Gesture Recognition Systems*. [En línea]. Disponible en: http://embots.dfki.de/doc/seminar_ss09/Andreas%20Farner.pdf. [Accedido: 01-nov-2016].

[23] Mark Lucente, *Dream Space commands*.

[24] IBM, «DreamSpace natural interaction». [En línea]. Disponible en: http://lucente.us/past/career/natural/dreamspace/index.html. [Accedido: 01-nov-2016].

[25] Eigen, «Eigen». [En línea]. Disponible en: http://eigen.tuxfamily.org/index.php?title=Main_Page. [Accedido: 01-nov-2016].

[26] Bertrand Meyer, «Event driven programming», *Event-driven design*. [En línea]. Disponible en: http://se.inf.ethz.ch/old/teaching/ss2007/0050/downloads/event.pdf. [Accedido: 20-nov-2016].

[27] Juan Carlos González, «Facebook compra Oculus por 2.000 millones de dólares», *Xataka*, 25-mar-2014. [En línea]. Disponible en: http://www.xataka.com/videojuegos/facebook-compra-oculus-por-2-000-millones-de-dolares. [Accedido: 20-nov-2016].

[28] Flatfrog, «Flatfrog multitouch 3200». [En línea]. Disponible en: http://www.flatfrog.com/sites/default/files/flatfrog_multitouch_3200_en_130311.pdf. [Accedido: 30-nov-2016].

[29] FlatFrogAB, *FlatFrog Multitouch 3200*.

[30] Luis Rodríguez Baena, «Fundamentos de la interacción persona-computadora», *Fundamentos de la interacción persona-computadora*. [En línea]. Disponible en: http://edu.enredo.org/IMG/pdf/Interaccion_persona-computadora.pdf. [Accedido: 06-dic-2016].

[31] Oculus, «Gear VR Powered by Oculus». [En línea]. Disponible en: https://www3.oculus.com/en-us/gear-vr/. [Accedido: 01-nov-2016].

[32] R. P. Moniruzzaman Bhuiyan, «Gesture-controlled user interfaces, what have we done and what's next?», *Gesture-controlled user interfaces, what have we done and what's next?* [En línea]. Disponible en: https://www.glyndwr.ac.uk/computing/research/pubs/sein_bp.pdf. [Accedido: 10-nov-2016].

[33] GRT, «Gesture Recognition Toolkit». [En línea]. Disponible en: http://www.nickgillian.com/wiki/pmwiki.php/GRT/GestureRecognitionToolkit. [Accedido: 01-nov-2016].

[34] GRT, «Gesture Recognition Toolkit automatic gesture spotting». [En línea]. Disponible en: http://www.nickgillian.com/wiki/pmwiki.php/GRT/AutomaticGestureSpotting. [Accedido: 01-nov-2016].

[35] GRT, «Gesture Recognition Toolkit forum». [En línea]. Disponible en: http://nickgillian.com/forum/. [Accedido: 01-nov-2016].

[36] GRT, «Gesture Recognition Toolkit reference». [En línea]. Disponible en: http://www.nickgillian.com/wiki/pmwiki.php/GRT/Reference. [Accedido: 01-nov-2016].

[37] Matt Swider, «Google Glass review», *TechRadar*. [En línea]. Disponible en: http://www.techradar.com/reviews/gadgets/google-glass-1152283/review. [Accedido: 15-nov-2016].

[38] D. P. Bertsekas, *Hand gesture recognition using Kinect*, vol. 1. Athena Scientific Belmont, MA, 1995.

[39] Mari-Carmen Marcos, «HCI concepto y desarrollo», *HCI (human computer interaction): concepto y desarrollo*. [En línea]. Disponible en: http://www.elprofesionaldelainformacion.com/contenidos/2001/junio/1.pdf. [Accedido: 03-nov-2016].

[40] Microsoft Research, *High performance touch*.

[41] Leap Motion, «How Does the Leap Motion Controller Work?», *Leap Motion Blog*, 09-ago-2014. [En línea]. Disponible en: http://blog.leapmotion.com/hardware-to-software-how-does-the-leap-motion-controller-work/. [Accedido: 13-nov-2016].

[42] DannBerg, «HP's new gesture-control laptop is the first with Leap Motion», *The Verge*, 19-sep-2013. [En línea]. Disponible en: http://www.theverge.com/2013/9/19/4745964/hp-envy-17-leap-motion-se-notebook. [Accedido: 30-nov-2016].

[43] Mark's Corner, «Human-Centered Design: an Introduction», *Mark's Corner*.

[44] Microsoft, «Human interface guidelines».

[45] K. K. Lamtharn Hantrakul, «Implementations of the Leap Motion in sound synthesis, effects modulation and assistive performance tools», *Implementations of the Leap Motion in sound synthesis, effects modulation and assistive performance tools*. [En línea]. Disponible en: http://quod.lib.umich.edu/cgi/p/pod/dod-idx/implementations-of-the-leap-motion-in-sound-synthesis.pdf?c=icmc;idno=bbp2372.2014.100. [Accedido: 13-nov-2016].

[46] VisitKorea.org, «Incheon Songdo International City, a City Embracing the Future», *Incheon Songdo International City, a City Embracing the Future*. [En línea]. Disponible en: http://english.visitkorea.or.kr/enu/ATR/SI_EN_3_6.jsp?cid=1718482. [Accedido: 05-dic-2016].

[47] Universidad de Sevilla, «Interacción persona ordenador», *Interacción Persona-Ordenador*. [En línea]. Disponible en: http://www.eici.ucm.cl/Academicos/R_Villarroel/descargas/com_h_m/IPO%20Escalona.pdf. [Accedido: 06-dic-2016].

[48] S. I. Otmar Hilliges, S. H. Andrew D. Wilson, y A. B. Armando Garcia-Mendoza, «Interactions in the air». [En línea]. Disponible en: https://www.microsoft.com/en-us/research/wp-content/uploads/2009/10/hilliges2009uist.pdf. [Accedido: 16-nov-2016].

[49] R. J. Nicolai Marquardt y J. A. J. Saul Greenberg, «Interaction techniques unifying touch and gesture on and above a digital surface», *The Continuous Interaction Space: Interaction Techniques Unifying Touch and Gesture On and Above a Digital Surface*. [En línea]. Disponible en: http://grouplab.cpsc.ucalgary.ca/grouplab/uploads/Publications/Publications/2011-ContinuousInteraction.Interact.pdf. [Accedido: 10-nov-2016].

[50] Sergio Herman Peralta Benhumea, «Interfaz de lenguaje natural usando Kinect», *Interfaz de lenguaje natural usando Kinect*. [En línea]. Disponible en: https://www.cs.cinvestav.mx/TesisGraduados/2012/tesisSergioPeralta.pdf. [Accedido: 21-nov-2016].

[51] Dave Evans, «Internet of things», *Internet de las cosas Cómo la próxima evolución de Internet lo cambia todo*. [En línea]. Disponible en: http://www.cisco.com/c/dam/global/es_mx/solutions/executive/assets/pdf/internet-of-things-iot-ibsg.pdf. [Accedido: 20-nov-2016].

[52] J. C. O. R. Silva y G. A. Giraldi, «Introduction to augmented reality», *Introduction to Augmented Reality*. [En línea]. Disponible en: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.63.4105&rep=rep1&type=pdf. [Accedido: 01-nov-2016].

[53] P. M. DANIEL PLEMMONS, «Introduction to motion control», *Introduction to Motion Control*. [En línea]. Disponible en: https://developer.leapmotion.com/articles/intro-to-motion-control. [Accedido: 01-nov-2016].

[54] Bill Buxton, «Invited Paper», *: Invited Paper: A Touching Story: A Personal Perspective on the History of Touch Interfaces Past and Future*. [En línea]. Disponible en: http://billbuxton.com/SID10%2031-1.pdf. [Accedido: 03-nov-2016].

[55] Tomoto S. Washio, «Kinect for Windows brigde», *Kinect for Windows brigde*. [En línea]. Disponible en: https://code.google.com/archive/p/kinect-mssdk-openni-bridge/. [Accedido: 03-nov-2016].

[56] Microsoft, «Kinect for Windows human interface guidelines», *Kinect for Windows Human Interface Guidelines v1.8.0*. [En línea]. Disponible en: https://msdn.microsoft.com/en-us/library/jj663791.aspx. [Accedido: 03-nov-2016].

[57] Javier Penalva, «Kinect para Xbox 360, jugar sin mando», *Xataka*, 14-jun-2010. [En línea]. Disponible en: http://www.xataka.com/videojuegos/kinect-para-xbox-360-jugar-sin-mando. [Accedido: 27-nov-2016].

[58] PAMELA LICALZI O'CONNELL, «Korea's high-tech utopia, where everything is observed», *The New York Times*, 05-oct-2005.

[59] Leap Motion, «Leap Motion and HTC Vive», *HTC Vive FAQ: What You Need to Know About Leap Motion + SteamVR*, 05-abr-2016. [En línea]. Disponible en: http://blog.leapmotion.com/leap-motion-htc-vive-faq/. [Accedido: 04-dic-2016].

[60] Leap Motion, «Leap Motion and virtual reality». [En línea]. Disponible en: https://www.leapmotion.com/product/vr. [Accedido: 01-nov-2016].

[61] Amazon, *Leap Motion controller for Mac or PC*.

[62] Leap Motion, «Leap Motion coordinate systems», *Coordinate Systems*. [En línea]. Disponible en: https://developer.leapmotion.com/documentation/cpp/devguide/Leap_Coordinate_Mapping.html. [Accedido: 27-nov-2016].

[63] Leap Motion, «Leap Motion developers», *VR Setup Guides*. [En línea]. Disponible en: https://developer.leapmotion.com/get-started. [Accedido: 13-nov-2016].

[64] Leap Motion, «Leap Motion fingers», *Fingers*. [En línea]. Disponible en: https://developer.leapmotion.com/documentation/cpp/devguide/Leap_Pointables.html. [Accedido: 28-nov-2016].

[65] Leap Motion, «Leap Motion frames», *Frames*. [En línea]. Disponible en: https://developer.leapmotion.com/documentation/csharp/devguide/Leap_Frames.html. [Accedido: 13-nov-2016].

[66] Leap Motion, «Leap Motion gestures», *Gestures*. [En línea]. Disponible en: https://developer.leapmotion.com/documentation/v2/java/devguide/Leap_Gestures.html. [Accedido: 04-dic-2016].

[67] Leap Motion, «Leap Motion hand». [En línea]. Disponible en: https://developer.leapmotion.com/documentation/cpp/api/Leap.Hand.html. [Accedido: 28-nov-2016].

[68] Leap Motion, «Leap Motion positional tracking», *Getting Started with the Leap Motion SDK*, 16-ago-2014. [En línea]. Disponible en: http://blog.leapmotion.com/getting-started-leap-motion-sdk/. [Accedido: 04-dic-2016].

[69] Leap Motion, *Leap Motion VR Mount + HTC Vive*.

[70] Leap Motion, *Leap Motion VR Mount + Oculus Rift CV1*.

[71] Leap Motion, «Leap skeletal mode», *Introducing the Skeletal Tracking Model*. [En línea]. Disponible en: https://developer.leapmotion.com/documentation/csharp/devguide/Intro_Skeleton_API.html. [Accedido: 01-nov-2016].

[72] Dr. Juan Carlos Cheang Wong, «Ley de Moore, nanotecnología y nanociencias», *Ley de Moore, nanotecnología y nanociencias.pdf*. [En línea]. Disponible en: http://www.revista.unam.mx/vol.6/num7/art65/jul_art65.pdf. [Accedido: 20-nov-2016].

[73] «libfreenect», GitHub. [En línea]. Disponible en: https://github.com/OpenKinect/libfreenect. [Accedido: 24-abr-2017].

[74] H. B. Andrew D. Wilson, «Lightspace», *Combining Multiple Depth Cameras and Projectors for Interactions On, Above, and Between Surfaces*. [En línea]. Disponible en: http://research.microsoft.com/en-us/um/people/awilson/publications/wilsonuist2010/wilson%20uist%202010%20lightspace.pdf. [Accedido: 01-nov-2016].

[75] L. S. Massimo Camplani y Romolo Camplani, «Low-cost efficient interactive whiteboard», *Low-Cost Efficient Interactive Whiteboard*. [En línea]. Disponible en: http://oa.upm.es/30488/1/INVE_MEM_2012_173574.pdf. [Accedido: 16-nov-2016].

[76] The Robotics Institute, Carnegie Mellon University, «Lucas-Kanade 20 years on», *Lucas-Kanade 20 Years On: A Unifying Framework*. [En línea]. Disponible en: http://www.ncorr.com/download/publications/bakerunify.pdf. [Accedido: 16-nov-2016].

[77] Ludique's Kinect Bundle, «Ludique's Kinect Bundle», *Ludique's Kinect Bundle*. [En línea]. Disponible en: https://code.google.com/archive/p/lkb-kinect-bundle/. [Accedido: 03-nov-2016].

[78] D. W. Craig Villamor y Luke Wroblewski, «Lukew touch gesture guide», *Touch Gesture*. [En línea]. Disponible en: http://static.lukew.com/TouchGestureGuide.pdf. [Accedido: 01-nov-2016].

[79] Universitat de Lleida, «Metodología que integra la ingeniería del software, la interacción persona ordenador y la accesibilidad en el contexto de equipos de desarrollo multidisciplinares», *Metodología que integra la ingeniería del software, la interacción persona ordenador y la accesibilidad en el contexto de equipos de desarrollo multidisciplinares*. [En línea]. Disponible en: http://www.tdx.cat/bitstream/handle/10803/8120/Tgsa2de5.pdf;jsessionid=7BE96A072864D2DEE483C21ED1CF4334.tdx2?sequence=2. [Accedido: 06-dic-2016].

[80] Digital Living, «Microsoft HoloLens», *Microsoft HoloLens Not Just Another VR Headset*. [En línea]. Disponible en: http://media.sandhills.com/doc.axd?id=2001318971&p=&ext=pdf&dl=False&wt=False&checksum=FXVXzcFC1dxxWvRevJRpCUD8OPhIzC2anzSpBw58ZBrW7rJM4awa9NTQhaPlWIsu6juNJI4IfI8%3D. [Accedido: 15-nov-2016].

[81] Mike Snider, «Microsoft Kinect gets into motion», *Microsoft Kinect gets into motion as E3 confab kicks off*. [En línea]. Disponible en: http://usatoday30.usatoday.com/tech/gaming/2010-06-14-vidgame14_ST_N.htm. [Accedido: 08-nov-2016].

[82] «Microsoft Kinect sensor evaluation». [En línea]. Disponible en: https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20110022972.pdf. [Accedido: 12-nov-2016].

[83] IFIXIT, «Microsoft Kinect teardown», *Xbox 360 Kinect Teardown*. [En línea]. Disponible en: https://www.ifixit.com/Teardown/Microsoft-Kinect-Teardown/4066/1. [Accedido: 01-nov-2016].

[84] Microsoft, «Microsoft XBox360». [En línea]. Disponible en: http://www.xbox.com/es-ES/Kinect/GetStarted. [Accedido: 01-nov-2016].

[85] University of Bristol, «MISTable», *MisTable: Reach-through Personal Screens for Tabletops*. [En línea]. Disponible en: http://research-information.bristol.ac.uk/files/27676880/MISTableFinal.pdf. [Accedido: 05-dic-2016].

[86] «MISTable». [En línea]. Disponible en: http://big.cs.bris.ac.uk/projects/mistable. [Accedido: 01-nov-2016].

[87] Mozilla Foundation, «Mozilla public license», *Mozilla Public License Version 2.0*. [En línea]. Disponible en: https://www.mozilla.org/en-US/MPL/2.0/. [Accedido: 01-nov-2016].

[88] Leap Motion, «Multiple leap motion support», *Leap Motion Community*. [En línea]. Disponible en: https://community.leapmotion.com/t/multiple-leap-motion-support/770/41. [Accedido: 01-nov-2016].

[89] K. Istenič, «Multi-touch surface based on RGBD camera», engd, Univerza v Ljubljani, 2013.

[90] Juan Castromil, «Myo, el brazalete para controlar gadgets con gestos y músculos», *Myo, el brazalete para controlar gadgets con gestos y músculos*.

[91] Myo, «Myo gesture control armband». [En línea]. Disponible en: https://www.myo.com/. [Accedido: 01-nov-2016].

[92] Alessandro Valli, «Natural interaction», *Notes on Natural Interaction*. [En línea]. Disponible en: http://www.idemployee.id.tue.nl/g.w.m.rauterberg/Movies/NotesOnNaturalInteraction.pdf. [Accedido: 01-nov-2016].

[93] UX Magazine, «New design practices for touch-free Interactions», *New Design Practices for Touch-free Interactions*. [En línea]. Disponible en: http://uxmag.com/articles/new-design-practices-for-touch-free-interactions. [Accedido: 03-nov-2016].

[94] openni.ru, «NiTE 2.2.0.11 | OpenNI».

[95] Prime Sense, «NITE algorithms», *PrimeSense™ NITE Algorithms 1.5*. [En línea]. Disponible en: http://openni.ru/wp-content/uploads/2013/02/NITE-Algorithms.pdf. [Accedido: 13-nov-2016].

[96] Leap Motion, «Oculus Rift», *Setting Up Virtual and Augmented Reality Scenes*. [En línea]. Disponible en: https://developer.leapmotion.com/documentation/v2/cpp/devguide/VR_Setup.html?highlight=oculus%20rift. [Accedido: 15-nov-2016].

[97] «Oculus skeletal tracking», *Leap Motion Community*. [En línea]. Disponible en: https://community.leapmotion.com/t/v2-skeletal-tracking-beta-new-sdk-and-build-v2-1-1-21671/1641. [Accedido: 04-dic-2016].

[98] «ofxCamaraLucida», GitHub. [En línea]. Disponible en: https://github.com/chparsons/ofxCamaraLucida. [Accedido: 24-abr-2017].

[99] «ofxKinectProjectorToolkit», GitHub. [En línea]. Disponible en: https://github.com/genekogan/ofxKinectProjectorToolkit. [Accedido: 24-abr-2017].

[100] «ofxProjectorKinectCalibration», GitHub. [En línea]. Disponible en: https://github.com/Kj1/ofxProjectorKinectCalibration. [Accedido: 24-abr-2017].

[101] T. A. N. Bjørnar Steinnes Luteberget, «ofxReprojection», *GitHub*. [En línea]. Disponible en: https://github.com/Illd/ofxReprojection. [Accedido: 01-nov-2016].

[102] H. B. Chris Harrison y Andrew D. Wilson, «OmniTouch», *OmniTouch: Wearable Multitouch Interaction Everywhere*. [En línea]. Disponible en: http://research.microsoft.com/en-us/um/people/awilson/publications/harrisonuist2011/harrisonuist2011.pdf. [Accedido: 01-nov-2016].

[103] OpenCV, «OpenCV», *OpenCV*. [En línea]. Disponible en: http://opencv.org/. [Accedido: 01-nov-2016].

[104] openFrameworks, «openFrameworks», *openFrameworks addons*. [En línea]. Disponible en: http://openframeworks.cc/about/. [Accedido: 13-nov-2016].

[105] openFrameworks, «openFrameworks addons», *openFrameworks addons*. [En línea]. Disponible en: http://ofxaddons.com/categories. [Accedido: 13-nov-2016].

[106] Structure, «OpenNI 2 downloads and documentation», *OpenNI 2 downloads and documentation*. [En línea]. Disponible en: http://structure.io/openni. [Accedido: 01-nov-2016].

[107] OpenNI, «OpenNI migration guide 1 to 2», *OpenNI migration guide 1 to 2*. [En línea]. Disponible en: http://com.occipital.openni.s3.amazonaws.com/OpenNI_Migration_Guide_1_to_2.pdf. [Accedido: 13-nov-2016].

[108] OpenNI, «OpenNI programmers guide», *OpenNI programmers guide*. [En línea]. Disponible en: http://com.occipital.openni.s3.amazonaws.com/OpenNI_Programmers_Guide.pdf. [Accedido: 13-nov-2016].

[109] openni.ru, «OpenNI source SDK for 3D sensors».

[110] Infragistics, «Paper Prototyping», *Paper Prototyping*. [En línea]. Disponible en: http://uxify.net/info/2015/workshops/Paper-Prototyping.pdf. [Accedido: 08-dic-2016].

[111] John Wiley & Son, «Paper Prototyping». [En línea]. Disponible en: http://www.id-book.com/downloads/casestudy_11point1.pdf. [Accedido: 25-nov-2016].

[112] M. T. Reinhard Sefelin y Verena Giller, «Paper Prototyping - What is it good for?», *Paper Prototyping - What is it good for? A Comparison of Paper- and Computer-based Low-fidelity Prototyping*. [En línea]. Disponible en: https://pdfs.semanticscholar.org/ab91/f6fb77f2de3b793bb934aa5a05aa5d5a3f92.pdf. [Accedido: 25-nov-2016].

[113] A. G. Steven P Dow, M. S. Jonathan Kass, y S. R. K. Daniel L Schwartz, «Parallel Prototyping», *Parallel Prototyping*. [En línea]. Disponible en: http://hci.stanford.edu/publications/2010/parallel-prototyping/ParallelPrototyping2010Slides.pdf. [Accedido: 11-dic-2016].

[114] M. Satyanarayanan, «Pervasive Computing», *Pervasive Computing: Vision and Challenges*. [En línea]. Disponible en: http://www.cs.cmu.edu/~aura/docdir/pcs01.pdf. [Accedido: 20-nov-2016].

[115] C. R. Ben Fry, «Processing». [En línea]. Disponible en: https://processing.org/. [Accedido: 01-nov-2016].

[116] Protocol Buffers, «Protocol Buffers», *Google Developers*. [En línea]. Disponible en: https://developers.google.com/protocol-buffers/. [Accedido: 01-nov-2016].

[117] tisquiiirrel, «Prototyping: storyboarding, paper prototyping», *@tisquirrel*, 17-may-2013. .

[118] W. M. Michel Beaudouin-Lafon, «Prototyping tools and techniques», *Prototyping tools and techniques*. [En línea]. Disponible en: https://www.kth.se/social/upload/52ef5ee4f2765445a466a28a/mackay-lafon-prototypes-52-HCI.pdf. [Accedido: 22-nov-2016].

[119] Pure Data, «Pure Data», *Pure Data*. [En línea]. Disponible en: http://puredata.info/. [Accedido: 01-nov-2016].

[120] Richard A. Bolt, «Put that there», *«Put-That-There»: Voice and Gesture at the Graphics Interface*. [En línea]. Disponible en: http://www.paulmckevitt.com/cre333/papers/putthatthere.pdf. [Accedido: 01-nov-2016].

[121] J. N. Gerhard Schall y Dieter Schmalstieg, «Rapid and accurate deployment of fiducial markers for augmented reality».

[122] Daniel Abril Redondo, «Realidad aumentada», *Realidad Aumentada*. [En línea]. Disponible en: http://www.it.uc3m.es/jvillena/irc/practicas/10-11/13mem.pdf. [Accedido: 20-nov-2016].

[123] 1LLUS, *Reality touchscreen University of Groningen*.

[124] «Reality touch theatre University of Groningen», *Reality Touch Theatre*. [En línea]. Disponible en: http://www.rug.nl/news/2011/01/touchscreen1?lang=en. [Accedido: 30-nov-2016].

[125] «Real-time Hand Tracking and Finger Tracking for Interaction».

[126] Y. B. Ronald Azuma, S. F. Reinhold Behringer, y B. M. Simon Julier, «Recent advances in augmented reality», *Recent Advances in Augmented Reality*. [En línea]. Disponible en: http://www.cc.gatech.edu/~blair/papers/ARsurveyCGA.pdf. [Accedido: 21-nov-2016].

[127] H. F. Andrew Maimone, «Reducing interference between multiple structured light depth sensors using motion», *Reducing Interference Between Multiple Structured Light Depth Sensors Using Motion*. [En

línea]. Disponible en: http://www.cs.unc.edu/~maimone/media/kinect_VR_2012.pdf. [Accedido: 03-nov-2016].

[128] Polhemus, «Remote object position attitude measurement system», *ABOUT ELECTROMAGNETICS*. [En línea]. Disponible en: http://polhemus.com/applications/electromagnetics. [Accedido: 01-nov-2016].

[129] V. Hinze-Hoare, «Review and analysis of human computer interaction (HCI) principles», *Review and Analysis of Human Computer Interaction (HCI)*. [En línea]. Disponible en: https://arxiv.org/ftp/arxiv/papers/0707/0707.3638.pdf. [Accedido: 20-nov-2016].

[130] SATO CORPORATION, «RFID Smart Labeling», *Label Gallery RFID Technology White Paper*. [En línea]. Disponible en: http://www.satoworldwide.com/sites/satoworldwide_com/Uploads/Files/LabelGallery/wp-LG%20RFID%20Smart%20Labeling.pdf. [Accedido: 20-nov-2016].

[131] J. M. Zhou Ren y Z. Z. Junsong Yuan, «Robust Hand Gesture Recognition with Kinect Sensor», *Robust Hand Gesture Recognition with Kinect Sensor*. [En línea]. Disponible en: http://eeeweba.ntu.edu.sg/computervision/Research%20Papers/2011/Robust%20Hand%20Gesture%20Recognition%20with%20Kinect%20Sensor.pdf. [Accedido: 10-nov-2016].

[132] R. S. Brett Jones, R. M. Michael Murdock, A. D. W. Hrvoje Benko, B. M. Eyal Ofek, y L. S. Nikunj Raghuvanshi, «Room Alive», *RoomAlive: Magical Experiences Enabled by Scalable, Adaptive Projector-Camera Units*. [En línea]. Disponible en: http://projection-mapping.org/wp-content/uploads/2014/01/RoomAlive_UIST2014.pdf. [Accedido: 01-nov-2016].

[133] Dave Parrack, «RoomAlive transforms your living room into an interactive video game», *RoomAlive transforms your living room into an interactive video game*. [En línea]. Disponible en: http://newatlas.com/roomalive-living-room-video-game/34176/. [Accedido: 01-dic-2016].

[134] Samsung, «Samsung 40 SUR40», *Samsung*. [En línea]. Disponible en: http://fb.uk.samsung.com/business/business-products/smart-signage/professional-display/LH40SFWTGC/EN. [Accedido: 01-nov-2016].

[135] S. H. Shahram Izadi, D. R. Stuart Taylor, A. B. Nicolas Villar, y Jonathan Westhues, «SecondLight», *Going Beyond the Display: A Surface Technology with an Electronically Switchable Diffuser*. [En línea]. Disponible en: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/SecondLight.pdf. [Accedido: 16-nov-2016].

[136] Roy Or-El, «Seminar in "Advanced Topics in Computer Vision"».

[137] UPV / EHU, «Sistemas Ubicuos», *Sistemas Ubicuos*. [En línea]. Disponible en: http://www.sc.ehu.es/acwlaroa/SUB/SU-11.pdf. [Accedido: 11-dic-2016].

[138] L. V. Gayle Curtis, «Storyboards and sketch prototypes for rapid interface visualization», *Storyboards and Sketch Prototypes for Rapid Interface Visualization*. [En línea]. Disponible en: http://ecologylab.net/courses/hcc/hostedMaterials/curtisVertelneyChi90Storyboards.pdf. [Accedido: 24-nov-2016].

[139] Leap Motion, «Taking motion control ergonomics beyond Minority Report», *Leap Motion Blog*, 22-ago-2014. [En línea]. Disponible en: http://blog.leapmotion.com/taking-motion-control-ergonomics-beyond-minority-report/. [Accedido: 03-nov-2016].

[140] Boost, «The Boost C++ libraries», *The Boost C++ Libraries*. [En línea]. Disponible en: https://theboostcpplibraries.com/boost.serialization. [Accedido: 01-nov-2016].

[141] J. S. B. Mark Weiser, «The coming age of calm technology», *THE COMING AGE OF CALM TECHNOLOGY*. [En línea]. Disponible en: http://homes.di.unimi.it/~boccignone/GiuseppeBoccignone_webpage/IUM2_files/weiser-calm.pdf. [Accedido: 01-nov-2016].

[142] Mark Weiser, «The computer for the 21st century», *The Computer for the 21st Century*. [En línea]. Disponible en: http://web.stanford.edu/class/cs344a/papers/computer-for-21-century.pdf. [Accedido: 01-nov-2016].

[143] Don Norman, «The design of everyday things», *The design of everyday things*. [En línea]. Disponible en: http://cc.droolcup.com/wp-content/uploads/2015/07/The-Design-of-Everyday-Things-Revised-and-Expanded-Edition.pdf. [Accedido: 15-nov-2016].

[144] bSkilledCom, *The Kinect effect*.

[145] Microsoft, «The Kinect Effect».

[146] ThinkMoto, «Think Moto gestures», *thinkmoto*. [En línea]. Disponible en: http://www.thinkmoto.de/gestures/. [Accedido: 01-nov-2016].

[147] Kyle Vanhemert, «This Augmented-Reality Sandbox Turns Dirt Into a UI», *WIRED*. [En línea]. Disponible en: https://www.wired.com/2013/08/this-augmented-reality-sandbox-turns-dirt-into-an-interactive-interface/. [Accedido: 20-nov-2016].

[148] Bill Buxton, «Timeline input history», *Some Milestones in Computer Input Devices: An Informal Timeline*. [En línea]. Disponible en: http://www.billbuxton.com/inputTimeline.html. [Accedido: 03-nov-2016].

[149] Johannes Schöning, «Touching the future: the rise of multitouch interfaces», *Touching the future: the rise of multitouch interfaces*. [En línea]. Disponible en: http://www.perada-magazine.eu/pdf/002864/002864.pdf. [Accedido: 07-nov-2016].

[150] Andrew D. Wilson, «TouchLight». [En línea]. Disponible en: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/10/ICMI-2004-TouchLight.pdf. [Accedido: 16-nov-2016].

[151] X. corporation Mark Weiser, «Ubiquitous computing», *Ubiquitous computing*. [En línea]. Disponible en: http://www.cc.gatech.edu/~keith/classes/ubicomplexity/pdfs/foundations/weiser-hot-topics.pdf. [Accedido: 01-nov-2016].

[152] Byeong-Ho KANG, «Ubiquitous Computing Environment Threats and Defensive Measures», *Ubiquitous Computing Environment Threats and Defensive Measures*. [En línea]. Disponible en: http://www.sersc.org/journals/IJMUE/vol2_no1_2007/4.pdf. [Accedido: 21-nov-2016].

[153] Songdo, «U-City», *Songdo IBD*. .

[154] Unity, «Unity», *Unity*. [En línea]. Disponible en: https://unity3d.com. [Accedido: 01-nov-2016].

[155] W. Ij. Maurice van Beurden y Yvonne de Kort, «User experience of gesture-based interfaces: A comparison with traditional interaction methods on pragmatic and hedonic qualities», *User experience of gesture-based interfaces: A comparison with traditional interaction methods on pragmatic and hedonic qualities*. [En línea]. Disponible en: http://access.uoa.gr/gw2011/proceedingsFiles/GW2011_32.pdf. [Accedido: 07-nov-2016].

[156] Andrew D. Wilson, «Using a depth camera as a touch sensor», *Using a Depth Camera as a Touch Sensor*. [En línea]. Disponible en: http://research.microsoft.com/en-us/um/people/awilson/publications/WilsonITS2010/WilsonITS2010.pdf. [Accedido: 15-nov-2016].

[157] vvv, «vvvv - a multipurpose toolkit», *vvvv*. [En línea]. Disponible en: https://vvvv.org/documentation/vvvv-a-multipurpose-toolkit. [Accedido: 01-nov-2016].

[158] Coursera, «Wizard of Oz - Universidad de California en San Diego», *Coursera*. [En línea]. Disponible en: https://www.coursera.org/learn/human-computer-interaction/lecture/9f0pI/wizard-of-oz. [Accedido: 25-nov-2016].

[159] ASUS, «Xtion Asus», *ASUS Global*. [En línea]. Disponible en: https://www.asus.com/3D-Sensor/Xtion_PRO/. [Accedido: 01-nov-2016].

Referencia de videos

[v.1] Proyecto De Grado, *Definición de un banco de gestos*. [En línea]. Disponible en: https://www.youtube.com/watch?v=cJzSwMnLprE. [Accedido: 11-12-2016].
[v.2] Proyecto De Grado, *Interacción mediante gestos espaciales*. [En línea]. Disponible en: https://www.youtube.com/watch?v=hrn95WgjC-s. [Accedido: 11-12-2016].
[v.3] Proyecto De Grado, *Interacción mediante gestos multitáctiles*. [En línea]. Disponible en: https://www.youtube.com/watch?v=2wjM0S5C2f8&feature=youtu.be [Accedido: 16-04-2017].
[v.4] Proyecto De Grado, *Interactive circles*. [En línea]. Disponible en: https://www.youtube.com/watch?v=WUiQKJozbHE. [Accedido: 11-12-2016].
[v.5] Proyecto De Grado, *Paper prototyping para ANIMuS*. [En línea]. Disponible en: https://www.youtube.com/watch?v=0NXKhCkYdj8&feature=youtu.be. [Accedido: 11-12-2016].

Índice de figuras