UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Fingsics

Simulador de colisiones masivas

Santiago Goycoechea

Thomas Wallace

Tesis de grado presentada en la Facultad de Ingeniería de la Universidad de la República, como parte de los requisitos necesarios para la obtención del título de grado en Ingeniería en Computación.

Director:
Eduardo Fernández

Montevideo – Uruguay
Noviembre de 2021

RESUMEN

Los motores físicos son una temática en constante estudio y evolución que abarca una gran cantidad de casos de uso, entre los cuales se encuentran las simulaciones físicas, la industria del cine y la de los videojuegos. Dentro de los motores físicos, una de las funcionalidades más relevantes es el manejo de colisiones, que comprende detectar colisiones entre cuerpos así como aplicar una respuesta adecuada para cada una de ellas.

En este proyecto se presenta Fingsics, un motor de física para cuerpos rígidos de geometría simple, desarrollado con el fin de evaluar diferentes técnicas de manejo de colisiones. Se propone además una nueva técnica de detección de colisiones con un buen desempeño, llegando a ser en ciertos escenarios hasta un 37 % más eficiente que *Sweep-And-Prune*, un algoritmo de detección de colisiones muy relevante en la actualidad.

En el Anexo A se encuentra el manual de usuario de Fingsics. El código desarrollado junto con los archivos binarios del programa están disponibles en el siguiente repositorio:

https://github.com/fingsics/Fingsics

Palabras claves:
Motor físico, Detección de colisiones, Volúmenes acotantes, Broad Phase Collision Detection, Sweep-And-Prune.

Tabla de contenidos

Capítulo 1

Introducción

La simulación física en el ámbito de juegos y efectos especiales es un área muy amplia de la computación gráfica, y ha sido muy estudiada. Esta área está en constante evolución, y esto se evidencia por la aparición de nuevas técnicas y recursos de hardware para lograr mayor fidelidad de forma eficiente, como por ejemplo la posibilidad reciente de realizar *ray tracing* en tiempo real [1]. Dentro de la simulación física, la temática de principal interés para este proyecto de grado es el manejo de colisiones entre cuerpos, un área de la mecánica clásica.

Los primeros ejemplos de software que tratan la física lo hicieron de manera simple: si se desea modelar un efecto particular en un programa se implementa dicho efecto [2]. Este enfoque es usualmente eficiente y aporta control total al dueño del software sobre su implementación, por lo que sigue siendo utilizado en una gran cantidad de programas al día de hoy. Sin embargo, al hacer esto con muchos efectos distintos el costo de que interactúen convincentemente entre sí crece demasiado, y se vuelve inmanejable.

Los motores de física surgen como respuesta al problema anterior: son código reutilizable que implementa los cálculos necesarios para simular física newtoniana en un contexto general, y que es utilizado por otro programa que les indica qué calcular. Su gran ventaja es abstraer a los desarrolladores de la implementación de cuerpos físicos y sus comportamientos, permitiendo enfocar el esfuerzo en otras áreas del desarrollo. La habilidad de trabajar con simulaciones de contexto genérico les permite ser reutilizables, pero su desventaja radica en que al ser genéricos, no pueden adaptarse al contexto donde serán usados. Esto significa que muchos atajos que podrían ser utili-

zados para aumentar el rendimiento de simulaciones particulares no pueden ser aprovechados.

Existen muchas técnicas eficientes para el manejo de colisiones, y aunque ninguna es estrictamente superior en todos los escenarios, la utilización de algunas de ellas es esencial para lograr rendimientos aceptables en las simulaciones cuando se trata de grandes cantidades de objetos. En razón de esto, el objetivo de este proyecto es el desarrollo de un motor físico, llamado *Fingsics* (una composición entre las palabras *Fing* y *Physics*), que permita evaluar el rendimiento de algunas de estas técnicas en una amplia variedad de escenarios, incluyendo escenas con cantidades masivas de objetos.

1.1. Organización del documento

Las secciones restantes del documento se organizan de la siguiente forma. En el Capítulo 2 se hace una recorrida por los fundamentos teóricos del manejo de colisiones, el estado del arte de algunas de sus técnicas de aceleración más conocidas y algunos de los motores físicos más utilizados en la actualidad. En el Capítulo 3 se expone en alto nivel el alcance y el diseño del motor de física a desarrollar, y se detallan los objetivos que se plantean para el proyecto. En el Capítulo 4 se explican los algoritmos, técnicas y estructuras específicas que fueron implementadas en los componentes principales del motor. En el Capítulo 5 se muestran y discuten los resultados obtenidos mediante experimentación. Finalmente, en el Capítulo 6 se presentan las conclusiones finales sobre el proyecto y posibles líneas de trabajo a futuro.

Capítulo 2

Marco teórico

En este capítulo se introducen los conceptos y algoritmos más relevantes para el manejo de colisiones, con un sesgo hacia aquellos aspectos que se utilizaron en el marco del proyecto. Se explicará en qué consiste un motor físico y se presentarán las librerías relacionadas a este tema más importantes en la actualidad.

2.1. Motor físico

Como se mencionó anteriormente, un motor físico implementa un código reutilizable que realiza los cálculos necesarios para simular un sistema físico. El software de este tipo suele ser utilizado como parte de un programa más grande, el cual le indica al motor qué calcular y utiliza los resultados de la simulación para un determinado fin. Este fin puede variar desde una simulación interactiva en tiempo real (usualmente juegos) hasta el renderizado de videos (usualmente efectos especiales).

El propósito de un motor físico es entonces, dado un sistema y su estado —un conjunto de objetos junto con sus características físicas— calcular cuál será el estado del sistema al transcurrir cierta cantidad de tiempo. Para generar una visualización de la simulación, se registra el estado del sistema en una serie de puntos en el tiempo, llamados cuadros (o *frames* en inglés), y se renderiza una imagen para cada uno de ellos.

Una de las funcionalidades más importantes de los motores físicos es el manejo de colisiones, pero existen otras, como las simulaciones del movimiento de cuerpos flexibles, fluidos y fuerzas (como el peso o la fuerza de

flotación de un cuerpo sumergido en un fluido).

El procesamiento realizado para manejar las colisiones se divide en tres etapas que son ejecutadas de forma secuencial: detección de colisiones, determinación de colisiones y respuesta a las colisiones. En el resto de este capítulo se profundizará sobre ellas. Tras aplicar la respuesta a las colisiones y por lo tanto terminar el manejo de colisiones, se suele renderizar un cuadro dibujando el nuevo estado del sistema, para luego volver a comenzar el ciclo realizando la detección de colisiones (pero esta vez con el nuevo estado del sistema), como se ve en la Figura 2.1.

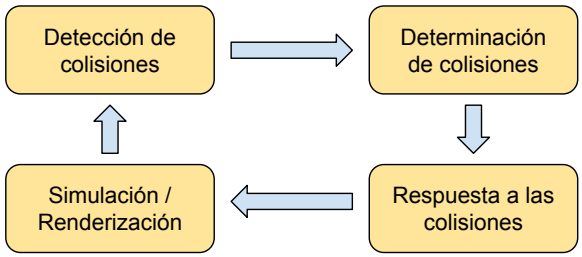


Figura 2.1: Ciclo de un motor físico.

Estas ideas son aplicables tanto a sistemas de dos como de tres dimensiones. Se ejemplificarán algunos conceptos utilizando sistemas bidimensionales, pero todo lo explicado es extensible a tres dimensiones.

2.2. Objetos y sus representaciones

A la hora de implementar un motor físico, se debe definir qué tipos de objetos se desea manejar y cómo se los representará. Los objetos pueden ser divididos en dos tipos: cuerpos rígidos [3] (cumplen que las distancias entre sus partículas no varían) y cuerpos deformables [4, 5] (como puede ser un trozo de tela o una cadena). Estos últimos tienen representaciones mucho más complejas: en el caso de cuerpos blandos (como telas) se suele utilizar una malla de vértices, y en el de cuerpos articulados (como cadenas) se utilizan restricciones, como por ejemplo que un eslabón de una cadena no puede estar a más de cierta distancia de otro. Al tener que manejar representaciones más complejas, los algoritmos de detección y respuesta a las colisiones para

este tipo de cuerpos son más complejos. Además surgen nuevos problemas, como la posibilidad de una colisión de un cuerpo consigo mismo. Dadas estas complejidades, los motores más simples soportan únicamente cuerpos rígidos.

Sobre cada cuerpo soportado por el sistema físico se debe almacenar la información necesaria para determinar si está colisionando o no, y calcular cuál es el resultado de una colisión, entre otras cosas. La representación de los objetos varía según la implementación del motor físico, pero algunas de las más importantes son las siguientes:

- Tipo del objeto (por ejemplo, esfera o cubo) y sus dimensiones, o una forma de caracterizar el espacio que ocupa
- Masa
- Posición lineal
- Posición angular
- Velocidad lineal
- Velocidad angular
- Tensor de inercia (caracterización de la inercia rotacional de un cuerpo, existen fórmulas para calcularlo en objetos simples)

2.3. Detección de colisiones

La detección de colisiones es la primera etapa en la secuencia de manejo de colisiones. Su objetivo es determinar todos los pares de objetos del sistema que están colisionando entre sí. Esta no es una tarea fácil y debe realizarse de forma eficiente, por lo que ha sido objeto de mucho estudio [3, 6].

Hay dos formas diferentes de encarar la detección de colisiones en un motor físico:

1. **Tratar el tiempo en la simulación como una serie de pasos discretos.** Estos pasos representan un instante de tiempo y en cada uno de ellos se suele renderizar un cuadro, por lo que también son llamados de esta forma. El intervalo de tiempo entre ellos puede ser constante o variable, pero el mecanismo de operación es el mismo en ambos casos. En cada paso se mueven todos los objetos en la escena a su nueva posición según sus velocidades y el tiempo transcurrido desde el último

paso, indistintamente de si están colisionando o no con otros. Luego de esto, se realiza el manejo de colisiones, del cual se obtienen las nuevas velocidades para utilizar en el paso siguiente. Esto implica que los objetos pueden no solo entrar en contacto, sino también intersecarse parcialmente o totalmente e incluso atravesarse completamente en el intervalo de tiempo transcurrido entre dos pasos. En el último de estos casos, ilustrado en la Figura 2.2, el motor físico nunca detectará una colisión, ya que los objetos nunca estuvieron en contacto en los instantes de tiempo correspondientes a cada paso donde se ejecuta el manejo de colisiones. Este tipo de simulaciones no son estrictamente realistas, pero por otro lado son rápidas y se adaptan bien a las necesidades de aplicaciones en tiempo real.

2. **Tratar el tiempo en la simulación de manera continua.** Esto usualmente implica proyectar las trayectorias de los objetos en el futuro cercano para detectar todas las colisiones y el instante exacto del tiempo en el que ocurren. Una vez que se detecta una colisión se reacciona a ella instantáneamente y se continúa con la simulación. Esto resuelve el problema de la Figura 2.2 y resulta en una simulación más fiel, pero proyectar las trayectorias de todos los objetos para calcular cuál será la próxima colisión es muy costoso. Por esta razón, este tipo de simulaciones están reservadas para aplicaciones que no necesitan interactividad ni tiempos de respuesta rápidos, como la industria del cine.

(a) Cuadro n **(b)** Cuadro $n + 1$

Figura 2.2: Colisión que no sería detectada por el manejo de tiempo discreto, dado que los objetos involucrados se atraviesan completamente entre dos cuadros consecutivos.

Para calcular todas las colisiones que están ocurriendo en un momento dado, un algoritmo simple (que se podría llamar de fuerza bruta) consis-

te en verificar si cada posible par de objetos está colisionando entre sí. El tiempo de ejecución de este algoritmo sería de orden $O(n^2)$, lo cual suele ser demasiado lento si se quiere manejar grandes cantidades de objetos. El enfoque que se utiliza en la actualidad consiste en dividir la tarea de detección de colisiones en tres etapas o fases que se ejecutan de forma secuencial: *Broad-Phase Collision Detection* (BPCD), *Mid-Phase Collision Detection* (MPCD) y *Narrow-Phase Collision Detection* (NPCD) [6].

La finalidad de las primeras dos etapas es simplificar el trabajo realizado por la que le sigue. La primera descarta pares de colisiones que se determina que no ocurrirán, mientras que la segunda encuentra, para cada par de objetos que podría colisionar, qué partes de ellos son las que estarían penetrándose si lo hicieran. Finalmente, la última etapa es la que efectivamente calcula los pares de objetos que colisionan. A veces, como cuando se trata con objetos simples, la MPCD es omitida, en cuyo caso la salida de la BPCD es transferida directamente a la NPCD. Para simplificar los chequeos en la NPCD, cuando una primitiva es demasiado compleja se puede aproximar su forma con un objeto más simple (por ejemplo, se podría aproximar la forma de una persona con una cápsula). En la Figura 2.3 se puede ver cómo se integran las etapas de la detección de colisiones al manejo de colisiones.

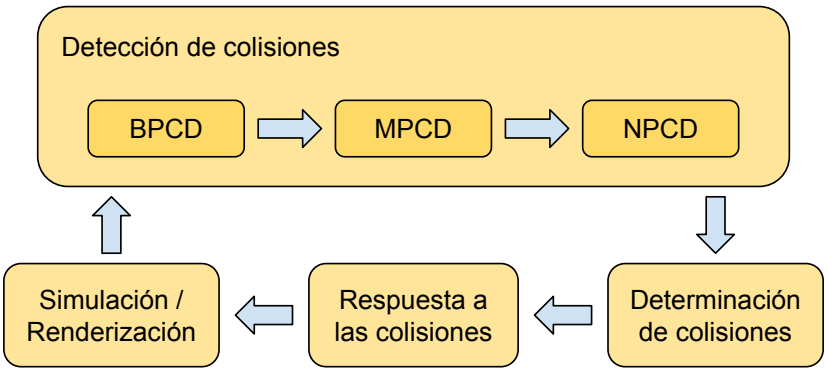


Figura 2.3: Manejo de colisiones, con las etapas de la detección de colisiones incluidas.

2.3.1. Volúmenes acotantes

Un volumen acotante (o *bounding volume* en inglés) es un volumen que contiene a un objeto en su totalidad, y que usualmente es geométricamente más simple que el objeto que encierra. Esta simplicidad los vuelve útiles para la detección de colisiones, ya que hace que detectar colisiones entre ellos sea menos costoso. Como encierran totalmente a su objeto, si se determina que el volumen acotante no colisiona con cierto cuerpo, se sabe que el objeto tampoco lo hará. Aprovechando esta propiedad, es muy usual que la etapa de BPCD consista en encontrar cuáles objetos tienen volúmenes acotantes que colisionan entre sí, para luego chequear las colisiones entre objetos solamente para aquellos cuyos volúmenes acotantes colisionan.

Existen diferentes tipos de volúmenes acotantes que son utilizados en diferentes contextos. Para medir la eficiencia de cierto tipo de volumen hay que tener en cuenta qué tan fácil es realizar un chequeo de colisiones entre dos volúmenes de ese tipo, y qué tan bien aproxima a los objetos (lo cual determina qué tantas colisiones permite descartar). La Figura 2.4 muestra diferentes volúmenes acotantes, algunos de los cuales se explican más adelante. Es interesante resaltar que por lo general a medida que la aproximación del objeto mejora, se vuelve más ineficiente la detección de colisiones entre volúmenes acotantes.

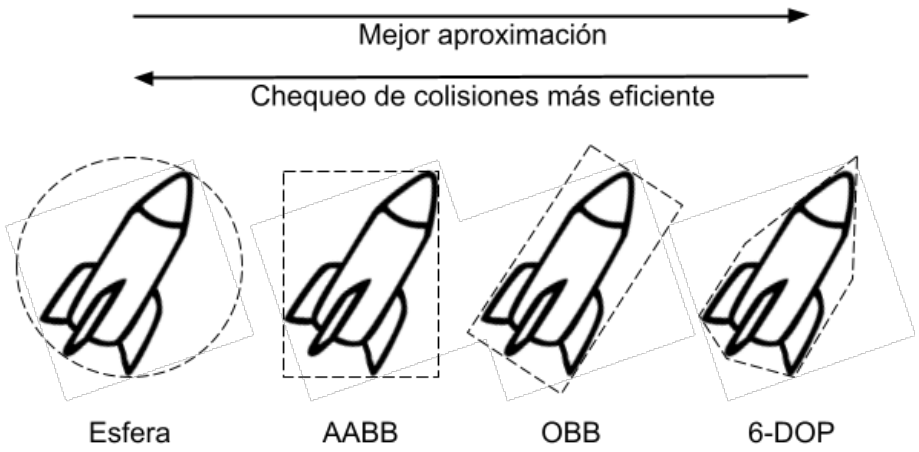


Figura 2.4: Diferentes volúmenes acotantes.

El concepto de la eficiencia de un volumen acotante en el contexto de un algoritmo de BPCD es reflejado por la ecuación 2.1 [7]. Esta indica que el tiempo de la detección de colisiones T es igual a la suma entre el tiempo de la detección de colisiones entre volúmenes acotantes (bB) y el tiempo de la detección de colisiones entre los objetos cuyos volúmenes acotantes colisionan (oO). El tiempo de detección de colisiones para volúmenes acotantes se calcula como la multiplicación entre la cantidad de chequeos realizados (b) y el tiempo de cada chequeo (B). Análogamente se calcula el tiempo de la detección de colisiones entre objetos con la cantidad de chequeos (o) y el tiempo de cada uno (O).

$$T = bB + oO \tag{2.1}$$

Es interesante ver que el valor de o siempre será menor o igual al de b, dado que los pares de objetos que se chequean (o) son solo aquellos para los cuales el chequeo entre volúmenes acotantes (b) resultó en colisión. La razón o/b depende de las características de la escena: la forma de los objetos y sus posiciones. Si todos los objetos están muy cerca entre sí, y sus volúmenes acotantes no los aproximan bien, entonces es probable que el valor de o sea cercano al de b (ya que habrán muchas colisiones entre volúmenes acotantes), y en caso contrario o debería ser bastante menor. Además, el costo del chequeo entre objetos, O, debería ser mayor al de volúmenes acotantes, B, ya que si no lo fuera sería más eficiente chequear colisiones entre los objetos sin realizar el paso previo de los volúmenes acotantes.

Es importante aclarar que no todos los algoritmos de BPCD funcionan de una forma que permita calcular los valores de b y de B. Por ejemplo, en el caso de *Sweep-And-Prune*, un algoritmo que se presentará más adelante, los chequeos entre volúmenes acotantes se realizan de forma implícita, por lo que no se puede calcular la cantidad de chequeos ni su costo. En este caso, se puede utilizar la ecuación 2.2, donde el término bB se sustituye por C, que representa el tiempo total del algoritmo de BPCD.

$$T = C + oO \tag{2.2}$$

Axis-Aligned Bounding Boxes (AABBs)

Una AABB es un tipo de volumen acotante que, como su nombre indica, es un prisma de base rectangular cuyos ejes están alineados a los ejes de coordenadas. Dados tres intervalos, cada uno correspondiente a un eje de coordenadas distinto, se puede definir entonces una AABB como los puntos del espacio que cumplen que cada una de sus tres coordenadas está adentro del intervalo que le corresponde. En base a esta definición, representar en memoria una AABB es muy sencillo: basta con mantener un único valor para sus puntos mínimos y máximos en cada eje de coordenadas (seis en total en un sistema tridimensional). También es usualmente sencillo encontrar la AABB de menor tamaño que encierra a un objeto, dado que esto implica solamente encontrar sus puntos máximos y mínimos en cada eje. En el caso de una malla poligonal, se pueden probar todos sus vértices (dado que los máximos y mínimos se darán en en ellos), y para objetos sencillos suele haber una forma simple de obtenerlos en base a su geometría.

Para detectar una intersección entre dos AABBs alcanza con detectar una intersección en los tres intervalos que las definen, debido a lo siguiente:

1. Si en cualquier eje de coordenadas no hubiera una intersección entre los intervalos correspondientes, entonces no existiría ningún punto del espacio que pertenezca a ambas AABBs, por lo que la intersección sería vacía. Este caso se da, por ejemplo, entre los objetos A y B de la Figura 2.5.
2. Si se da una intersección en todos los ejes de coordenadas, entonces existe un intervalo en cada eje de coordenadas donde hay al menos un punto perteneciente a ambos volúmenes, y estos intervalos definen a su vez una AABB más pequeña cuyos puntos cumplen con las definiciones de las dos AABBs, por lo que todo ese volumen representa la intersección entre ellas dos. Este caso se da entre los objetos B y C de la Figura 2.5.

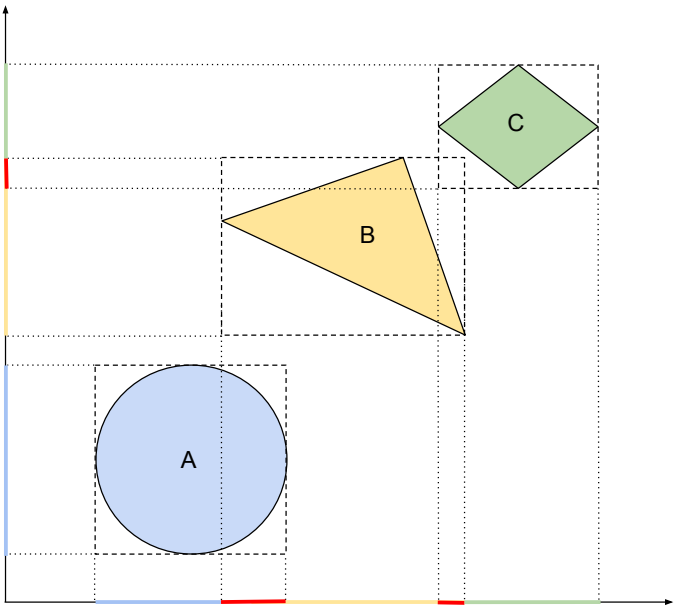


Figura 2.5: Ejemplo de los intervalos en los ejes que representan a las AABBs, mostrando en rojo las intersecciones. Los objetos A y B muestran cómo alcanza con que los intervalos de las AABBs no se intersequen en un solo eje para que sean disjuntas. Los objetos B y C se intersecan en ambos ejes y por lo tanto sus AABBs colisionan.

La mayor desventaja de las AABBs es que no aproximan bien a objetos cuyas dimensiones dominantes no se alinean bien a los ejes. Esto se da, por ejemplo, con objetos alargados (como el cohete mostrado antes en la Figura 2.4). A pesar de esto, gracias a su representación simple y la facilidad para definirlas y operar con ellas, las AABBs son utilizadas en muchas técnicas de detección de colisiones.

Oriented Bounding Boxes (OBBs)

Las OBBs son prismas de base rectangular sin ninguna restricción sobre su orientación, que también son utilizadas como volúmenes acotantes. La ventaja de las OBBs es que permiten aproximar a los objetos de forma más precisa que las AABBs, pero calcular las OBBs que mejor aproximan a un objeto (no necesariamente es una sola) y verificar si dos OBBs colisionan es más costoso que con las AABBs.

Encontrar una OBB que aproxime de forma óptima a un objeto es una tarea compleja, pero si el objeto es rígido esta tarea se debe realizar una sola vez, ya que si este se traslada, rota, o sufre cualquier transformación, se le

puede aplicar la misma transformación a la OBB para no tener que volver a calcularla.

Para modelos poligonales, existe un algoritmo para encontrar las mejores OBBs en tiempo $O(n^3)$, siendo n el número de vértices [8]. Este algoritmo se basa en un teorema que dice que las OBBs que mejor aproximan a un objeto siempre tendrán dos caras adyacentes que están al ras de dos aristas del modelo. Sabiendo esto, en modalidad de fuerza bruta se prueban todos los posibles pares de aristas del modelo y para cada uno se busca la mejor OBB, para elegir la mejor después de haber iterado por todos los pares de aristas. Este algoritmo resulta demasiado ineficiente si el modelo es de muchos vértices.

Por otro lado, se han desarrollado varios métodos que permiten encontrar una buena OBB (aunque no óptima) en tiempo lineal. Uno de ellos es el algoritmo desarrollado por Gottschalk [9], que se basa en hacer análisis de componentes principales (PCA) para hallar buenos ejes para la OBB. Otro método es el desarrollado por Larsson y Kälberg [10], un algoritmo un poco más complejo que halla una OBB casi óptima en tiempo lineal, partiendo de un volumen acotante más complejo: un politopo discreto orientado (k-DOP).

Una vez definidas las OBBs, se necesita un método para verificar si dos de estas colisionan. Existe un algoritmo que lo hace realizando menos de 200 operaciones [11]. Dicho algoritmo se basa en el teorema de separación de hiperplanos, que dice que si dos conjuntos convexos de puntos (como lo son las OBBs) son disjuntos, entonces existe un plano entre ellos que los separa dejando a uno de cada lado. A partir de esto, se puede demostrar que si dos OBBs son disjuntas, entonces existe un plano que las separa y además es paralelo a alguna cara de alguna de las dos OBBs, o a un eje de cada una de ellas. Dado que cada OBB tiene 3 orientaciones únicas de caras y 3 orientaciones únicas de ejes, si las OBBs no colisionan, entonces existe un plano con alguna de las siguientes 15 orientaciones que las separa:

- Alguna de las 3 orientaciones de las caras de la primera OBB
- Alguna de las 3 orientaciones de las caras de la segunda OBB
- Alguna de las 9 (3x3) orientaciones definidas tomando uno de los 3 ejes de la primera OBB y uno de los 3 de la segunda

Sabiendo esto, el algoritmo consiste en proyectar las OBBs sobre el eje normal a cada uno de los planos y verificar si los intervalos resultantes se

intersecan. Si hay intersección en los 15 pares de proyecciones, entonces ninguna de las 15 orientaciones sirve para separar a las OBBs con un plano, por lo que hay una colisión entre las OBBs. Si la intersección es vacía en al menos una de las 15 orientaciones, entonces ya se sabe que no hay intersección, dado que existe un plano con esa orientación que separa a las OBBs.

2.3.2. Broad-Phase Collision Detection (Sweep-And-Prune)

Esta primera etapa de la detección de colisiones consiste en emplear técnicas a gran escala para realizar un primer filtrado sobre todos los posibles pares de objetos. Su principal objetivo, entonces, es descartar la mayor cantidad de pares de objetos posible con el mínimo costo computacional posible. Como es de esperar, los pares de objetos obtenidos luego de realizar la fase de BPCD no indican necesariamente que exista una colisión entre dichos objetos, sino que una colisión es posible.

Para minimizar el tiempo computacional de esta fase se puede utilizar una gran variedad de algoritmos y estructuras acompañantes. Estos algoritmos suelen hacer uso de la coherencia temporal, una propiedad que dice que, en intervalos cortos de tiempo, la posición y orientación de los objetos no varían mucho.

Por otra parte, el mejor algoritmo para utilizar en cada caso depende fuertemente de las características del conjunto de objetos sobre el cual se ejecutará, ya que distintos algoritmos hacen uso de distintas propiedades y condiciones para descartar posibles colisiones, y por lo tanto su rendimiento varía según qué tanto se ajusten los objetos en cuestión a sus fortalezas. A continuación se presentan las estructuras y técnicas más comunes.

Sweep-And-Prune (SAP)

Sweep-And-Prune (SAP) es un algoritmo ampliamente estudiado [12, 13, 14] que hace uso extensivo de AABBs, sus propiedades, y de la coherencia temporal, la cual está presente en la mayoría de los escenarios. El algoritmo cumple su función reportando únicamente como colisiones posibles aquellos pares de objetos cuyas AABBs colisionan. A continuación se explica el funcionamiento del algoritmo, haciendo énfasis en sus estructuras y cómo estas permiten descartar correctamente colisiones.

La primera estructura a considerar, entonces, son las AABBs. Aprovechando la eficiencia de los chequeos de intersecciones entre ellas, ya se puede plantear un posible algoritmo de BPCD: chequear explícitamente las colisiones entre las AABBs de todos los pares posibles de objetos y retornarlas. Asumiendo que el chequeo de colisión entre dos AABBs es menos costoso que el chequeo entre los objetos en sí (lo cual usualmente es el caso) y que se eliminan suficientes colisiones, este método podría llegar a mejorar el rendimiento por sobre no utilizar ningún algoritmo de BPCD. Sin embargo, como ya se verá, es posible mejorar su costo computacional de diversas maneras.

La segunda estructura utilizada por el algoritmo SAP es más compleja. Dado un arreglo ordenado de valores que corresponden a los bordes de intervalos, es posible determinar cuáles de estos intervalos se intersecan en tiempo lineal. Tanto el algoritmo para hacer esto como SAP se basan en la siguiente propiedad: si dos intervalos se intersecan, entonces el mínimo de uno de ellos estará dentro del otro. Aprovechando esto se puede entonces recorrer el arreglo en orden ascendiente manteniendo una lista de los intervalos para los cuales ya se encontró su mínimo pero no su máximo, y cuando se encuentra el mínimo de un nuevo intervalo en el arreglo, se sabe que este se interseca con todos los que están actualmente en la lista.

Basándose en esto, el algoritmo SAP utiliza un arreglo como el recién descrito para cada eje de coordenadas de la simulación, donde los intervalos en cada arreglo son aquellos que definen las AABBs de los objetos en el eje correspondiente. La Figura 2.6 muestra un ejemplo con las estructuras mencionadas hasta el momento. Hay que notar que también es posible utilizar listas enlazadas en lugar de arreglos para este fin, pero en la mayoría de los casos esto es menos eficiente [14].

Hasta el momento se sabe que construyendo estos arreglos y realizando una recorrida por ellos se puede averiguar qué intervalos de las AABBs colisionan en cada eje, y que las AABBs que colisionan son aquellas cuyos intervalos se intersecan en cada eje. Entonces, realizando estos pasos en cada cuadro de la simulación se puede proponer una segunda versión del algoritmo de BPCD que solo utiliza AABBs, la cual también descarta todos los pares de objetos cuyas AABBs no colisionan. Sin embargo, el rendimiento de este algoritmo aún puede ser mejorado.

La primera mejora es persistir los arreglos de valores entre cuadros y

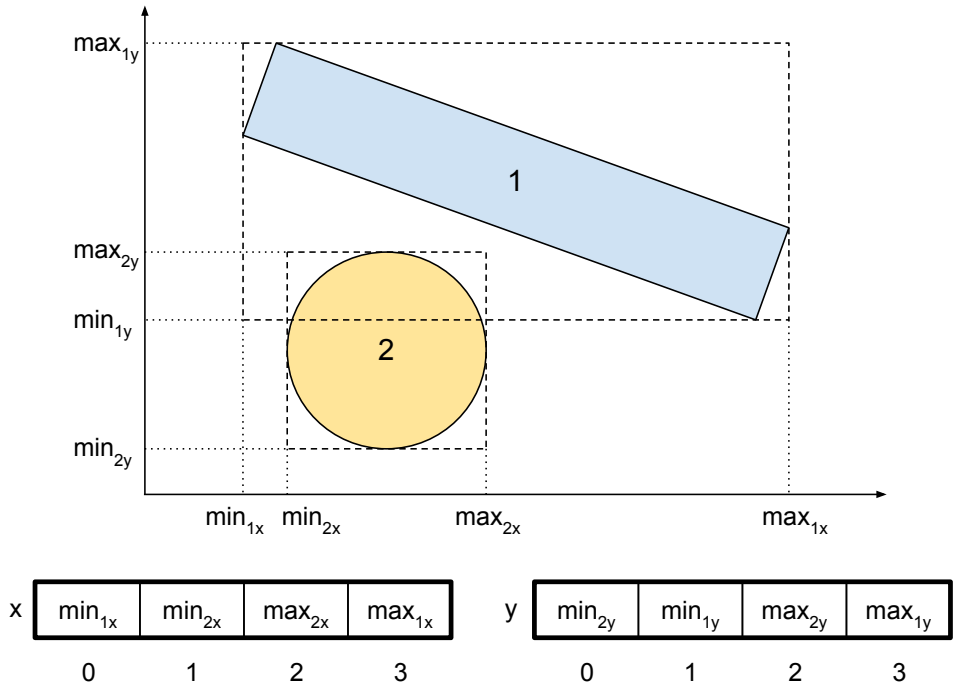


Figura 2.6: Ejemplo de arreglos SAP (con dos objetos en dos dimensiones). Los arreglos contienen en cada posición una referencia a la AABB que representan, su valor y una bandera que indica si es un valor mínimo o un valor máximo.

reordenarlos en cada paso en vez de reconstruirlos. Esto a su vez da lugar a una segunda mejora: utilizar una colección con tiempos de inserción y lectura constantes que contiene los pares de AABBs que se están intersecando actualmente. Estos cambios permiten que en cada cuadro de la simulación se parta de los arreglos con los valores ordenados y la colección de pares de AABBs intersecadas del cuadro anterior, y que al realizar una única recorrida por los arreglos actualizando y reordenando sus valores se actualice completamente la colección de colisiones. A continuación se describe este procedimiento, y qué hace que sea más eficiente que el algoritmo que no persiste las estructuras de los cuadros anteriores.

La actualización de cada arreglo SAP consiste en recorrer todos los objetos actualizando el valor mínimo y el máximo de su AABB que corresponden a ese arreglo. Además, cada vez que se actualiza un valor en un arreglo, se lo mueve en él para que mantenga su orden. Esto se hace utilizando una estrategia similar al algoritmo de ordenamiento *Bubble Sort*: una vez actualizado el valor se lo compara con sus vecinos inmediatos para saber si está correctamente ubicado con respecto a ellos. Si es el caso se procede a actualizar

otro valor, y si no se intercambian las posiciones del valor actualizado con el del vecino adecuado y se repite este proceso hasta que el valor modificado sea mayor que el valor anterior y menor que el valor siguiente en el arreglo. Como solo un valor fue actualizado y el resto del arreglo se encuentra ordenado, se sabe que no es necesario mover ningún otro valor para mantener el orden.

La actualización de la colección de colisiones al reordenar los arreglos explota más características de las AABBs e intervalos, y ocurre en simultáneo con el proceso de actualización de los valores.

Una nueva intersección entre intervalos ocurre cuando un mínimo pasa a estar dentro de otro intervalo (debido a la propiedad explicada antes), y se puede dar de dos formas:

1. Un mínimo pasó a valer menos que un máximo. En este caso se da una nueva intersección, ya que antes, como el mínimo de un intervalo era mayor al máximo del otro, no había intersección.
2. Un mínimo pasó a valer más que otro mínimo. Este caso no es de interés, ya que al cruzarse dos mínimos la intersección existe antes y después del cruce (antes del cruce es el mínimo de una AABB que está dentro del intervalo de la otra, y después es al revés).

Cuando los intervalos correspondientes a un eje de dos AABBs comienzan a intersecarse, se verifica si hay intersección también en los otros ejes, y en ese caso se agrega la nueva colisión.

Análogamente, un mínimo puede escaparse de un intervalo de las siguientes dos formas:

1. Un mínimo pasó a valer más que un máximo. En este caso deja de haber intersección entre estos intervalos, debido a que el mínimo de un intervalo pasa a ser mayor que el máximo del otro.
2. Un mínimo pasó a valer menos que otro mínimo. Como ya se explicó, cuando se cruzan dos mínimos la intersección existe antes y después del cruce, por lo que este caso no es de interés.

Cuando dos intervalos que previamente se intersecaban dejan de hacerlo, se sabe que sus AABBs correspondientes también dejarán de colisionar, ya que para que lo hagan, todos sus intervalos deben intersecarse en los ejes

correspondientes. Entonces, cuando esto ocurre se remueve el par de AABBs de la colección de colisiones sin verificar intersecciones en otros ejes.

Resumiendo, el algoritmo SAP utiliza las siguientes estructuras: una AABB asignada a cada objeto, un arreglo ordenado para cada eje de coordenadas que contiene los valores correspondientes a todas las AABBs en dicho eje, y una colección de pares de objetos que contiene todas las colisiones actuales entre las AABBs de los objetos. En cada cuadro de la simulación, se recorren todos los objetos de a uno y se van actualizando sus valores en los arreglos y la colección de colisiones de forma simultánea. El Anexo B contiene el pseudocódigo de una iteración del algoritmo.

En cuanto a la eficiencia de SAP, es interesante la utilización de un algoritmo de ordenamiento similar a *Bubble Sort*, cuya complejidad computacional en el peor caso, $O(n^2)$, es pobre en comparación a otros. Esta elección se debe a que en conjuntos de objetos que presentan coherencia temporal (es decir, la mayoría de los casos) este método resulta muy eficiente: los objetos no se mueven grandes distancias en cada cuadro, por lo que los valores de sus AABBs no cambian drásticamente y esto se traduce a pocas modificaciones en los arreglos de SAP reordenando valores. En otras palabras, el peor caso para el algoritmo *Bubble Sort* rara vez es encontrado por SAP, y como consecuencia este método de ordenamiento es una buena elección por ser rápido en casos favorables. El resultado de esto es que SAP reduce la complejidad del problema de encontrar pares de AABBs que se intersecan de $O(n^2)$ a $O(n + k)$, donde n es la cantidad de objetos y k es la cantidad de pares que se intersecan.

Grillas

Este algoritmo usa grillas (o *grids* en inglés) n-dimensionales de celdas que dividen el espacio de forma regular y sin superponerse [6]. Las celdas suelen estar alineadas a los ejes de coordenadas y al igual que las AABBs pueden ser representadas por las coordenadas de sus bordes en los ejes.

Su utilidad consiste en hallar las celdas en las que se encuentran los volúmenes acotantes de los objetos. Luego, si los volúmenes acotantes de dos objetos no comparten ninguna celda, se sabe que no colisionarán y por lo tanto ese par de objetos no se agrega a la lista de posibles colisiones. Por otro lado, si dos volúmenes acotantes sí comparten alguna celda, se considera

que es probable que haya una intersección entre ellos, y entonces ese par de objetos se agrega a la lista. La Figura 2.7 muestra un ejemplo de esto.

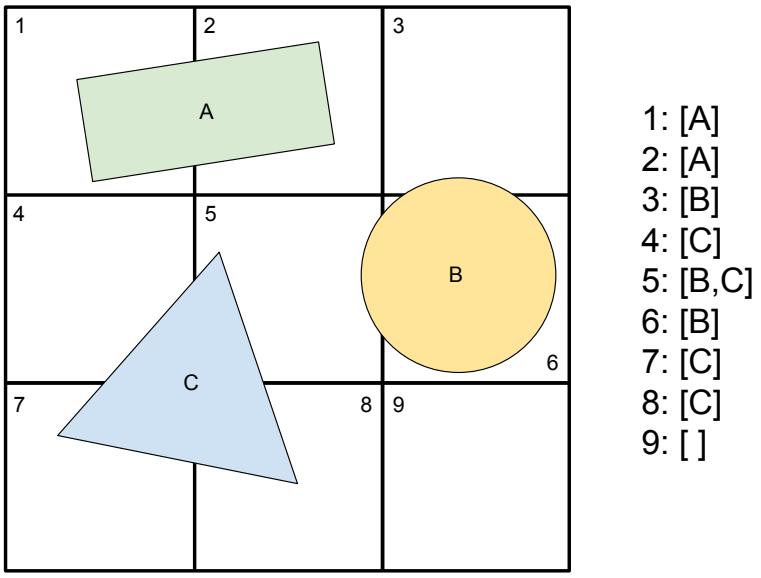


Figura 2.7: Ejemplo de grilla con tres objetos. En este caso, la única colisión retornada sería el par $\{B, C\}$.

El uso de grillas es una aplicación de la estrategia *dividir y conquistar*, pero presenta complejidad en la configuración de las celdas: tamaños de celda muy grandes implican muchos objetos por celda, y tamaños muy pequeños de celda aumenta la cantidad de celdas y hace que cada objeto forme parte de varias celdas, o que muchos objetos cambien de celda entre cuadros consecutivos. Esto disminuye la eficiencia, ya que en un caso se chequean muchas colisiones innecesarias en las celdas y en el otro se debe controlar un gran número de celdas en los chequeos de todos los objetos. Debido a esto, uno de los puntos clave para utilizar grillas efectivamente es encontrar un balance apropiado a las circunstancias para el tamaño de las celdas (también llamado resolución de la grilla).

Otro tema que merece consideración es cómo persistir la información de las celdas a las que pertenece cada objeto. En una escena donde el área de uso sea limitada es posible mantener un arreglo n-dimensional para representar los objetos que se encuentran en cada celda, pero si la cantidad de celdas es demasiado grande los requerimientos de memoria de este enfoque también serán muy altos y una gran cantidad de esta memoria podría estar desperdi-

ciada en celdas vacías. Para remediar esto se pueden utilizar técnicas como *spatial hashing* [3, 15] o *hierarchical grids* [3, 16].

MultiSAP

El algoritmo MultiSAP es una combinación de los conceptos en los que se basan las grillas y SAP [14]. Consiste en subdividir el espacio en sectores al igual que si se estuviera trabajando con grillas, pero ejecutando una instancia de SAP en cada celda. La principal ventaja de este algoritmo es que combate el principal problema del algoritmo SAP: al aumentar mucho la cantidad de objetos el desempeño disminuye significativamente, ya que objetos muy lejanos que coincidan en uno o dos ejes aún así van a interactuar en las estructuras de SAP cuando no es necesario (como muestra la Figura 2.8). El algoritmo MultiSAP también ofrece un enfoque relativamente simple para paralelizar el algoritmo SAP: utilizar hilos independientes para procesar distintas instancias de SAP al mismo tiempo. Al igual que con la característica anterior, esto se vuelve más importante a medida que la cantidad de objetos en la escena crece.

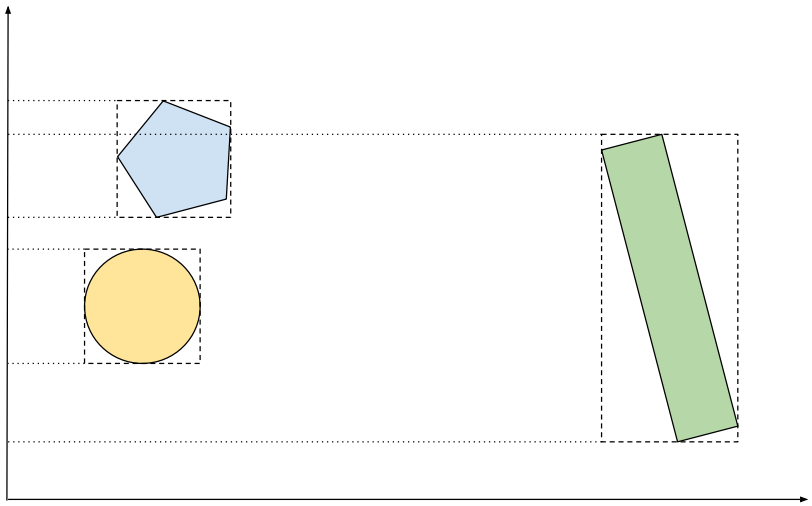


Figura 2.8: Objetos lejanos interactuando innecesariamente en SAP.

En cuanto a desventajas, MultiSAP hereda las complejidades de los dos algoritmos en los que se basa. Al igual que con las grillas, es importante elegir tamaños de celdas adecuados para obtener un buen rendimiento. Una peculiaridad interesante es que con celdas muy pequeñas el comportamiento

del algoritmo se acerca más al de las grillas, mientras que con celdas muy grandes se acerca más al de SAP. Otra desventaja es la complejidad extra del manejo de bordes. Al igual que en las grillas, un objeto que se encuentra en la frontera que une varias celdas debe ser tomado en cuenta por todas ellas. Esto implica trabajo repetido en varias celdas y además requiere lidiar con inserciones y remociones en las instancias SAP. Esto último es bastante más costoso y complejo que insertar y remover objetos de grillas simples, ya que requiere inserciones en más estructuras, las cuales deben mantenerse ordenadas y consistentes luego de cada modificación [14].

2.3.3. Mid-Phase Collision Detection

En la etapa de Mid-Phase Collision Detection (MPCD) se busca encontrar qué partes de dos objetos pueden estar colisionando. Esto es útil ya que al tener objetos de geometría compleja, el procesamiento de la siguiente etapa es mucho más simple si ya se sabe cuál parte de cada uno de los objetos es la que podría estar en intersección. Como se mencionó previamente, esta etapa a veces es omitida (por ejemplo, cuando se trata con objetos muy simples), en cuyo caso se pasa directamente a la etapa de NPCD.

Una técnica muy utilizada es la de construir una jerarquía de volúmenes acotantes para cada objeto, y buscar en esta etapa de la detección de colisiones qué volúmenes acotantes de las jerarquías de ambos objetos son los que efectivamente colisionan entre sí. Para construir las jerarquías una opción es utilizar OBBs, generando OBBTrees [11]. De forma más general, la jerarquía puede construirse utilizando diferentes tipos de politopos discretos orientados (k-DOPs) [17].

Cómo construir jerarquías de volúmenes acotantes de forma que encierren al objeto eficientemente (con poco espacio de sobra) y utilizando la menor cantidad de volúmenes acotantes posible no es un problema fácil de resolver, así como tampoco lo es diseñar algoritmos para chequear la intersección entre dos jerarquías de cierto tipo.

2.3.4. Narrow-Phase Collision Detection

En esta última etapa de la detección de colisiones, se debe terminar de establecer cuáles pares de objetos están intersecándose. Para esto no queda

otra alternativa que chequear todos los pares de primitivas obtenidos en los cálculos realizados en las etapas anteriores. Normalmente, el algoritmo a utilizar aquí es específico para el par de primitivas que haya que chequear. Por ejemplo, si las dos primitivas son esferas, hay una colisión si y solo si la distancia entre sus centros es menor a la suma de sus radios. Sin embargo, también existe un algoritmo que permite calcular la distancia mínima entre dos volúmenes convexos genéricos, llamado GJK por sus inventores (Gilbert, Johnson y Keerthi) que puede ser usado en caso de no encontrar un algoritmo ad hoc más eficiente para cierto par de primitivas [18].

2.4. Determinación de colisiones

Una vez que ya se sabe cuáles pares de objetos colisionan entre sí, el siguiente paso es calcular precisamente toda la información relativa a las colisiones que sea necesaria para luego aplicar las respuestas correspondientes a cada una de ellas. La información que suele interesar es el punto de colisión, la normal de los objetos en el mismo, y la distancia de interpenetración (como el tiempo en las simulaciones avanza de forma discreta, una colisión implica que los objetos se interpenetran, o lo que es lo mismo, que uno esté parcialmente dentro del otro). Al interpenetrarse los objetos, en realidad no hay un único punto de intersección, por lo que se debe elegir un punto de colisión en el volumen de intersección que genere una respuesta lo más correcta posible. También puede ocurrir que dos objetos colisionen en más de un lugar simultáneamente, como ocurre en la Figura 2.9, en cuyo caso interesa calcular todo esto para cada uno de los puntos de colisión (o, en el caso de que haya un área de contacto, para cada uno de sus vértices).

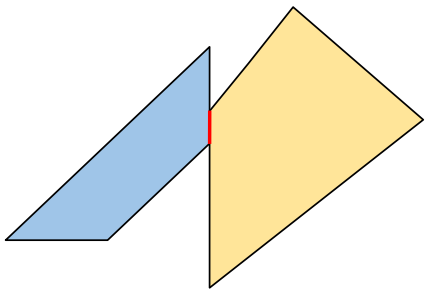


Figura 2.9: Colisión con más de un punto de contacto. La colisión se da a lo largo de toda la linea roja.

Cómo calcular esta información depende de qué primitivas son las que están colisionando. Siguiendo con el ejemplo de las dos esferas, que se muestra en la Figura 2.10, se pueden obtener los datos de la siguiente forma:

- Punto de colisión: está sobre el segmento que une a los centros de las esferas, a una distancia del centro de la primer esfera proporcional a la relación entre su radio y el de la otra esfera.
- Distancia de interpenetración: es la suma de los radios de las esferas menos la distancia entre sus centros.
- Normal en el punto de colisión: es la resta entre el centro de una de las esferas y el punto de colisión.

En esta etapa también puede interesar distinguir las colisiones en dos tipos diferentes: las que son un contacto de reposo y las que no. Es importante distinguir los casos en los cuales un objeto está reposando sobre otro (o ambos están reposando entre sí) ya que la respuesta a este tipo de colisiones suele manejarse de forma distinta.

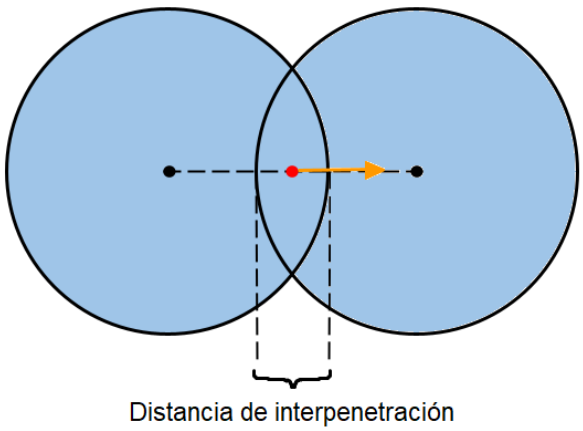


Figura 2.10: Colisión entre dos esferas. En rojo se marca el punto de colisión y en naranja la normal en el punto de colisión.

2.5. Respuesta a las colisiones

Una vez que se tiene la información relativa a las colisiones que se están dando en el momento, se debe aplicar la respuesta (cambios en velocidades y/o fuerzas) correspondiente a cada una de ellas. Al ocurrir una colisión entre dos objetos, cada uno de ellos experimenta una fuerza en el punto de colisión de igual magnitud pero sentido opuesto (como lo indica la segunda ley de Newton). Si bien la ley de la conservación de la energía indica que tras la colisión la cantidad total de energía permanecerá invariable, parte de la energía cinética que existe antes de la colisión se transforma en otros tipos de energía (como calor o la correspondiente a la deformación de un objeto). Qué tanta energía cinética se pierde en la colisión depende del material del cual estén compuestos ambos objetos (no es lo mismo picar una pelota de tenis contra cemento que contra una gelatina), y para cada par de materiales existe un coeficiente de restitución (un número entre cero y uno) que indica la proporción de energía cinética que se conserva tras el choque. Otra componente a tener en cuenta cuando dos objetos entran en contacto es la fuerza de rozamiento, que impide o dificulta el movimiento relativo entre ellos, y es una fuente de pérdida de energía cinética.

Calcular las velocidades finales de los objetos en base a sus características y las fuerzas recién mencionadas no es una tarea simple, y el resultado puede ser uno de varios casos: un deslizamiento de un objeto sobre otro, una colisión elástica (los objetos chocan y rebotan), una colisión inelástica (los objetos chocan y permanecen pegados) o un contacto de reposo, entre otros. Existen varios métodos para enfrentar este problema, como el basado en penalizaciones [19, 20] y el basado en impulsos, que se describe a continuación. Cada uno de estos modelos provee a su vez una forma de modelar la fricción, pero no se entrará en detalle sobre esto.

Respuesta a las colisiones basada en impulsos

El enfoque basado en impulsos es muy utilizado para modelar colisiones, ya que es simple y eficiente [20, 21, 22]. El impulso es una magnitud vectorial que indica la variación de momento lineal que experimenta un cuerpo, y a partir de él se pueden calcular los cambios en las velocidades lineales y angulares de los objetos. El método busca entonces, dada una colisión entre dos objetos, calcular el impulso resultante en cada uno de ellos tras la

colisión.

Sean O_1 y O_2 dos objetos con coeficiente de restitución e, masas m_1 y m_2, tensores de inercia I_1 e I_2, velocidades lineales v_1 y v_2, y velocidades angulares ω_1 y ω_2. Se define el vector $\hat{n}$ como la normal a las superficies de los objetos en el punto de contacto (apuntando hacia el objeto O_2), y r_1 y r_2 como los vectores que van desde el centro de masa de los objetos O_1 y O_2 al punto de colisión p, respectivamente (Figura 2.11).

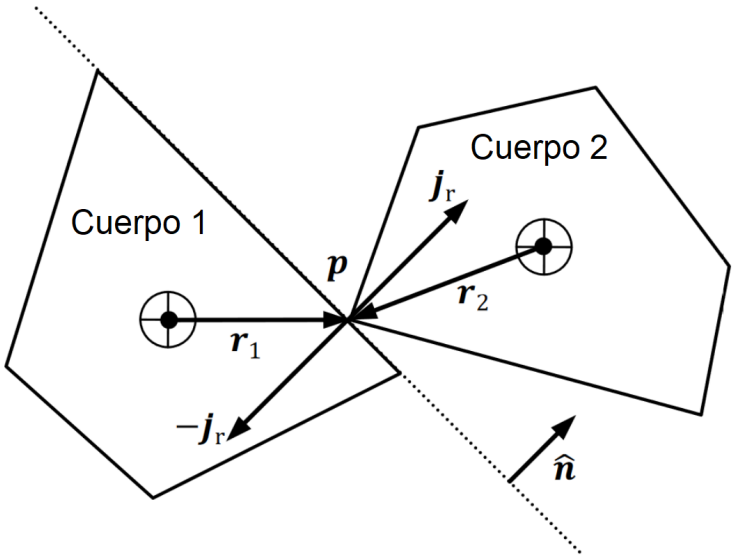


Figura 2.11: Ejemplo para el cálculo del impulso resultante de una colisión.

Para calcular el impulso, el primer paso es calcular la velocidad lineal de ambos objetos en el punto de contacto p, para lo cual se utiliza la siguiente fórmula (siendo i el número de objeto):

$$v_{pi} = v_i + \omega_i \times r_i \tag{2.3}$$

Dadas las velocidades de los objetos en el punto de contacto, se define el vector de velocidad relativa pre-colisión como $v_r = v_{p2} - v_{p1}$. Luego se puede calcular la magnitud del impulso resultante, j_r, que es igual para ambos objetos, de la siguiente forma:

$$j_r = \frac{-(1+e)v_r \cdot \hat{n}}{\frac{1}{m_1} + \frac{1}{m_2} + (I_1^{-1}(r_1 \times \hat{n}) \times r_1 + I_2^{-1}(r_2 \times \hat{n}) \times r_2) \cdot \hat{n}} \tag{2.4}$$

Dada la magnitud del impulso resultante de la colisión, se pueden calcular las velocidades lineales y angulares finales de los objetos:

$$v_1' = v_1 - \frac{j_r}{m_1}\hat{n} \tag{2.5}$$

$$v_2' = v_2 + \frac{j_r}{m_2}\hat{n} \tag{2.6}$$

$$\omega_1' = \omega_1 - j_r I_1^{-1}(r_1 \times \hat{n}) \tag{2.7}$$

$$\omega_2' = \omega_2 + j_r I_2^{-1}(r_2 \times \hat{n}) \tag{2.8}$$

En el caso de que uno de los cuerpos sea un cuerpo inamovible (de masa infinita), a los cuales se llamará *objetos estáticos*, la ecuación (2.4) debe ser sustituida por la siguiente, donde el objeto O_2 es el de masa finita:

$$j_r = \frac{-(1+e)v_r \cdot \hat{n}}{\frac{1}{m_1} + I_1^{-1}(r_1 \times \hat{n}) \times r_1 \cdot \hat{n}} \tag{2.9}$$

Resolución de colisiones simultáneas

En simulaciones de tiempo continuo nunca ocurren colisiones simultáneamente. Estas son una consecuencia directa de la discretización del tiempo, y es deseable poder manejarlas correctamente para mantener la fidelidad de la simulación en estos casos. Resolver las colisiones una por una sin tener en cuenta que un objeto puede estar en contacto con varios otros al mismo tiempo no funciona bien, por lo que se ha trabajado en soluciones que permitan resolver todas las colisiones de forma simultánea. Un método bastante usado se basa en la resolución de una formulación de un Problema Complementario Lineal (LCP) [23, 24, 25], que es un sistema de ecuaciones lineales con ciertas particularidades, cuya solución se puede obtener mediante métodos iterativos (que implican un esfuerzo computacional no menor).

2.6. Bibliotecas relacionadas

A lo largo de los años han existido una gran variedad de motores de física que incluyen manejo de colisiones. A continuación se describen *Bullet* y *PhysX*, dos de las bibliotecas más utilizadas actualmente, que poseen una amplia variedad de herramientas y también ofrecen mejor rendimiento que sus competidores [26, 27].

2.6.1. PhysX

PhysX es un motor físico y SDK escrito en C++, adquirido por NVIDIA en 2008, cuyo código es público desde el 2015 bajo la licencia *BSD 3*. Está disponible para ser utilizado en Microsoft Windows, Linux, macOS, PlayStation 3, PlayStation 4, Xbox 360, Xbox One, Wii, Android y iOS, y es utilizado por los dos motores de videojuegos más establecidos de la industria: Unity y Unreal Engine. Su implementación está pensada para que pueda aprovechar las capacidades de GPUs GeForce con soporte para la plataforma de cómputo CUDA y también soporta ejecución multihilo tradicional en procesadores.

En cuanto a funcionalidades, soporta cuerpos rígidos y deformables (como telas y cuerdas), cuerpos con articulaciones, y fluidos, entre otras cosas. Tiene implementados tres algoritmos diferentes de BPCD: Sweep-And-Prune (SAP), Multi Box Pruning (MBP), y Automatic Box Pruning (ABP) [28]. Mientras que el primero es un algoritmo tradicional, los otros dos fueron desarrollados por NVIDIA en la última década, y son bastante similares entre sí, ya que Automatic Box Pruning es una reimplementación de Multi Box Pruning. El funcionamiento de Multi Box Pruning es similar al de MultiSAP, dado que aplica la misma idea para paralelizar un algoritmo, pero en vez de hacerlo sobre el algoritmo SAP lo hace sobre otro llamado Box Pruning [29]. PhysX no utiliza ningún algoritmo de MPCD.

2.6.2. Bullet

Bullet es una biblioteca de manejo de colisiones implementada en C++ [30]. Fue diseñada para ser utilizada en juegos, efectos especiales y robótica y es abierta y libre para uso comercial bajo la licencia *zlib*. Al igual que PhysX, está soportada en las principales plataformas para juegos y ha visto uso en motores de juegos, como por ejemplo Godot [31]. También posee

integraciones con varios software de modelado 3d como *Maya* y *Blender*, y soporte experimental para aceleración con GPUs mediante OpenGL, pero actualmente la librería solo corre oficialmente sobre procesadores (también soporta ejecución multihilo).

La librería ofrece manejo de colisiones continuo y discreto y soporta objetos rígidos, articulados (mediante el uso de restricciones) y flexibles (incluyendo telas, cuerdas y volúmenes) en simultáneo. Además cuenta con un manejo especial para física de *ragdolls* y vehículos, entre otras cosas. En cuanto a algoritmos, Bullet tiene implementaciones de jerarquías de volúmenes acotantes formadas por AABBs y MultiSAP como opciones para BPCD, y no implementa ningún algoritmo de MPCD.

Capítulo 3

Solución propuesta

En este capítulo se define el alcance del proyecto, explicando qué funcionalidades se implementaron y cuáles no, y por qué. También se presenta el diseño de la solución, mostrando de forma general la arquitectura del programa desarrollado, y detallando qué función cumple cada uno de sus componentes.

3.1. Alcance del proyecto

El foco del proyecto es crear un motor de física que permita reproducir escenas en tiempo real, y evaluar y comparar distintos algoritmos de BPCD en ellas. Para realizar dicha evaluación interesa soportar una amplia variedad de escenarios, incluyendo escenas con miles de objetos. Se decidió implementar el motor utilizando manejo discreto de tiempo, dado que es más simple y eficiente. Además, la ventaja de su contraparte (la detección de colisiones continua) es que la simulación resultante es más realista, y esto no es uno de los objetivos principales del proyecto.

Los algoritmos de BPCD a implementar y evaluar son los siguientes:

- **Fuerzas brutas utilizando AABBs y OBBs**. Son útiles como línea base, ya que es esperable que ofrezcan una mejora de rendimiento considerable por sobre no utilizar ningún algoritmo de BPCD y son fáciles de implementar. Se entiende como *fuerza bruta* a chequear todos los pares de objetos posibles descartando colisiones con la estructura designada.
- **SAP**. Fue elegido porque está presente en las librerías más conocidas (tanto PhysX como Bullet tienen una implementación de SAP, pero

también hay otras).

- **Combinación de SAP y OBBs**. Como las OBBs no son comúnmente utilizadas en algoritmos de BPCD, resulta interesante evaluar su rendimiento utilizándolos para filtrar aún más la salida de SAP, sobre todo cuando se manejan objetos con una dimensión dominante (es decir, objetos alargados). Este algoritmo consiste en ejecutar SAP seguido por una fuerza bruta con OBBs.

Para enfocar el trabajo en el estudio de algoritmos de BPCD, se decidió omitir la fase MPCD y utilizar cuerpos rígidos simples, no implementando objetos compuestos o flexibles en la simulación. En este contexto, la selección de objetos rígidos debe contener suficiente variedad para simular una buena cantidad de escenarios posibles para estos algoritmos. Como todos los algoritmos de BPCD hacen uso de OBBs o AABBs en lugar de la geometría del objeto en sí para hacer comparaciones, la forma específica de los mismos no afecta el funcionamiento de los algoritmos de BPCD. Lo que sí los podría afectar, ya que no es completamente opacado por el uso de OBBs y AABBs, es la relación entre las dimensiones de los objetos: están aquellos que tienen una dimensión dominante (objetos alargados), aquellos que tienen dos dimensiones dominantes (objetos planos) y aquellos que no tienen dimensiones dominantes (objetos compactos). Entonces, para lograr una buena variedad de escenas para estos algoritmos, se implementó un objeto rígido para cada uno de estos tres tipos:

- **Esfera**. Representan los objetos compactos.
- **Cápsula**. Tienen forma de píldora (es decir, un cilindro con una media esfera del mismo radio centrada en cada cara plana) y representan los objetos alargados.
- **Tile**. Secciones con forma rectangular de un plano, representan los objetos planos.

Se implementaron también cohetes, que tienen una geometría más compleja pero internamente se manejan como si fueran cápsulas (a excepción del dibujado).

La respuesta a las colisiones en Fingsics se implementó utilizando impulsos, ya que este enfoque es simple y eficiente, y funciona bien para objetos rígidos poco complejos. Sin embargo, la respuesta a las colisiones es un área

muy profunda, y siempre se pueden agregar más funcionalidades que incrementen la fidelidad de la simulación a costo de rendimiento y tiempo de desarrollo. Algunas de estas funcionalidades que se consideraron pero se decidió no implementar son el rozamiento entre objetos, el manejo diferencial del contacto de reposo, y la resolución de colisiones simultáneas mediante la formulación de un problema complementario lineal (aunque sí se implementó un método de resolución muy básico para esto último). Las tres mejorarían ampliamente la fidelidad de la simulación, pero al igual que la implementación de objetos más complejos, traen consigo muchas dificultades y no aportan nada significativo a la comparación de algoritmos de BPCD.

Similarmente, se decidió no hacer uso de GPUs o programación multihilo para acelerar los cálculos, ya que este tipo de optimización no es central a los algoritmos elegidos.

Algo que sí se decidió implementar son los objetos estáticos (o de masa infinita), es decir, objetos que no se ven afectados por impulsos. Esto quiere decir que estos objetos no son afectados por colisiones pero sí afectan a los objetos con los que colisionan. Son útiles para implementar pisos y paredes, lo cual aumenta mucho la variedad posible de escenas. Las tiles fueron implementadas como objetos que deben ser estáticos.

En cuanto a funcionalidades principales del motor, se proponen las siguientes:

- Interfaz gráfica que permita ver las simulaciones en tiempo real y controlar la vista
- Posibilidad de elegir el algoritmo de BPCD (incluyendo no utilizar ningún algoritmo, salteando esta etapa)
- Generación de reportes con datos que permitan evaluar la eficiencia de los algoritmos, como el tiempo de cálculo por cuadro y la cantidad de chequeos que realizan los algoritmos
- Grabación y reproducción de escenas dentro de Fingsics
- Grabación de la simulación en formato de video

Las últimas dos funcionalidades son útiles cuando la cantidad de objetos es suficientemente alta para que la simulación no funcione en tiempo real (ya que calcular la información necesaria para cada cuadro lleva demasiado tiempo). Al grabar la escena, se realiza el procesamiento y se guarda la

información para después poder verla en tiempo real, sin tener que rehacer los cálculos.

3.2. Diseño

Se decidió implementar Fingsics para la plataforma Windows, y utilizando C++, dado que este es el lenguaje más utilizado para software de este tipo por tener un buen desempeño en términos de tiempo de ejecución. Se desarrolló el proyecto en base al paradigma de programación orientada a objetos buscando producir código fuente modularizable y fácil de entender. Para el renderizado de las escenas se utilizó la API OpenGL, una plataforma muy usada.

A continuación se explican en alto nivel algunas de las clases que componen el sistema, haciendo foco en las más importantes para comprender el funcionamiento del programa. Se comienza mostrando las estructuras de datos que fueron utilizadas, luego se explica qué sucede en la etapa de inicialización del programa y finalmente se entra en detalle sobre su ciclo principal.

3.2.1. Estructuras de datos

Se usaron las siguientes estructuras de datos básicas para facilitar el flujo de datos en el programa:

- **Vector**. Representa un vector en un sistema cartesiano de tres dimensiones. Contiene definiciones de funciones para el manejo básico de vectores, como el producto interno, el producto escalar y la rotación en base a un eje.
- **Matrix**. Una matriz de tamaño 3x3. Es útil para representar los tensores de inercia de los objetos, y contiene funciones para el manejo básico de matrices, como la trasposición, la multiplicación con otras matrices y con vectores, y la inversión.
- **Impulse**. Representa un impulso a ser aplicado a un objeto. Contiene su magnitud, su dirección, una referencia al objeto al cual debe aplicarse, y la masa del objeto que lo generó (el otro involucrado en la colisión).

- **Collision**. Contiene información sobre una colisión, como los dos objetos involucrados, el punto de colisión, la normal en el punto de la colisión y la distancia de interpenetración.
- **AABB**. Contiene los seis puntos necesarios para definir una AABB y el objeto para el cual está funcionando como volumen acotante.
- **OBB**. Contiene su posición, sus dimensiones y sus normales.
- **Scene**. Contiene la información de la escena, como sus objetos y la configuración de la cámara.

Además de estas estructuras, se implementó una clase llamada *Object*, la principal estructura de datos del programa, que representa a los cuerpos rígidos que conforman las escenas. Los objetos comparten la siguiente información:

- Masa
- Posición lineal
- Posición angular
- Velocidad lineal
- Velocidad angular
- Aceleración
- Color

Además, todos los objetos comparten las mismas funciones para encolar impulsos a ser aplicados, para aplicar la lista de impulsos encolados, y para actualizar su posición (en base a su velocidad) y su velocidad (en base a su aceleración).

Como ya se mencionó, en la solución existen tres tipos de cuerpos: cápsulas, esferas y tiles. Cada uno de estos tres tipos está implementado por una subclase de *Object*, y cada subclase contiene rutinas e información específica al tipo de sólido que representa: las esferas están representadas por su radio, las cápsulas por su radio y su largo, y las tiles por su largo y su ancho. Adicionalmente, cada tipo de objeto tiene su tensor de inercia, AABB, OBB y métodos para calcular estas estructuras.

3.2.2. Etapa de inicialización

Lo primero que se hace al iniciar Fingsics es cargar la configuración de la ejecución desde un archivo de texto. Quien se encarga de esto es el

manejador llamado *ConfigLoader*. Tras cargar la configuración, el siguente paso es cargar la escena que se simulará, para lo cual se implementó el manejador *XmlReader*. Teniendo los parámetros de la ejecución y la escena cargados, se inicializan los algoritmos de detección de colisiones y ya se puede comenzar con el ciclo principal de la simulación.

3.2.3. Ciclo principal

En cada iteración del ciclo principal de Fingsics, se ejecuta la detección, determinación y respuesta a las colisiones, para luego renderizar un cuadro de la simulación.

La cantidad de iteraciones del manejo de colisiones que se realizan por segundo de simulación es configurable y constante. Esto define un intervalo de tiempo fijo para cada iteración, y como el tiempo que se precisa para procesar cada iteración es variable se deben contemplar los siguientes escenarios:

1. El tiempo necesario para procesar y renderizar un cuadro es menor que el intervalo definido: en este caso alcanza con esperar el tiempo restante para completar el intervalo. Al hacer esto se evita que la simulación transcurra más rápido de lo especificado, y si este caso se da siempre las FPS del programa se mantendrán constantes e iguales a la cantidad de iteraciones por segundo del manejo de colisiones.
2. El tiempo necesario para procesar y renderizar un cuadro es mayor que el intervalo definido: en este caso se espera a que se termine el procesamiento del cuadro y luego se dibuja. Esto significa que la simulación no podrá ser vista en tiempo real y las FPS del programa serán menores que la cantidad de iteraciones por segundo del manejo de colisiones.

En cualquiera de las dos situaciones anteriores y sin importar el tiempo real transcurrido, el tiempo entre iteraciones del manejo de colisiones utilizado internamente por la simulación será el configurado inicialmente. Esto hace que las simulaciones sean deterministas más allá del tiempo que tome calcularlas.

Por otro lado, es importante observar que al aumentar la cantidad de iteraciones por segundo del manejo de colisiones aumenta la fidelidad y la

cantidad de procesamiento de la simulación. Esto es así porque transcurre menos tiempo entre cada chequeo de colisiones, llevando a que las intersecciones entre objetos se detecten más temprano y la respuesta a ellas sea más fidedigna.

Detección y determinación de colisiones

En el primer paso de la detección de colisiones, una instancia de la clase abstracta *BroadPhaseAlgorithm* se encarga de aplicar el algoritmo de BPCD. De esta clase heredan cinco subclases, una por cada uno de los algoritmos de BPCD implementados:

- NoBroadPhase
- AABBBruteForce
- OBBBruteForce
- SAP
- SAPAndOBBs

Estas cinco clases implementan una única función pública que toma como entrada la lista de todos los objetos, y devuelve una lista de pares que contiene únicamente los pares de objetos que podrían estar colisionando, habiendo descartado otros.

La salida de la etapa de BPCD es la entrada de la de NPCD, que es implementada por la clase *NarrowPhaseAlgorithm*, y cuya función `getCollisions(...)` devuelve la lista de colisiones que ocurren. En esta clase se puede encontrar una función para cada posible par que se puede formar utilizando dos de los tres cuerpos que se manejan (esferas, cápsulas y tiles), correspondiente al algoritmo de NPCD para ese par de objetos en específico. Estas funciones implementan también la determinación de las colisiones, ya que en el proceso de verificar si dos primitivas colisionan se calcula la mayoría de la información que se precisa sobre la colisión.

Respuesta a las colisiones

La funcionalidad de respuesta a las colisiones se implementa en la clase *CollisionResponseAlgorithm*. En dicha clase se puede encontrar la función `collisionResonse(...)`, que tomando como entrada la lista de colisiones,

calcula y aplica los impulsos correspondientes a todos los objetos, modificando sus velocidades. Esta función utiliza otras dos que calculan el impulso resultante de una colisión entre dos objetos específicos, una función para cuando ninguno de los objetos es estático y otra para cuando uno de ellos lo es. La clase también contiene una función `moveObjects(...)` que se encarga de que cada objeto actualice su posición y velocidad.

Renderización y otras funcionalidades

Finalmente, la renderización es realizada por el manejador *SceneRenderer*, quien dibuja en pantalla a los objetos y los ejes de coordenadas. En cada iteración se llevan a cabo además otras funciones del programa. Entre estas están la recaudación de datos para grabar escenas que luego pueden ser cargadas por Fingsics, grabar videos, y generar reportes. Esas tareas son llevadas a cabo por las clases *SceneRecorder*, *VideoRecorder* y *LoggingManager*, respectivamente.

Estas tres últimas clases persisten los datos que les corresponden en el disco una vez terminada la simulación, ya sea por que se alcanzó el número de cuadros indicado en la configuración para la simulación, o porque el usuario decidió terminarla.

Diagrama de clases

En la Figura 3.1 se muestra un diagrama de clases con los componentes principales de la solución. No se incluyeron los parámetros de entrada de las funciones porque el diagrama quedaría muy sobrecargado de esa forma.

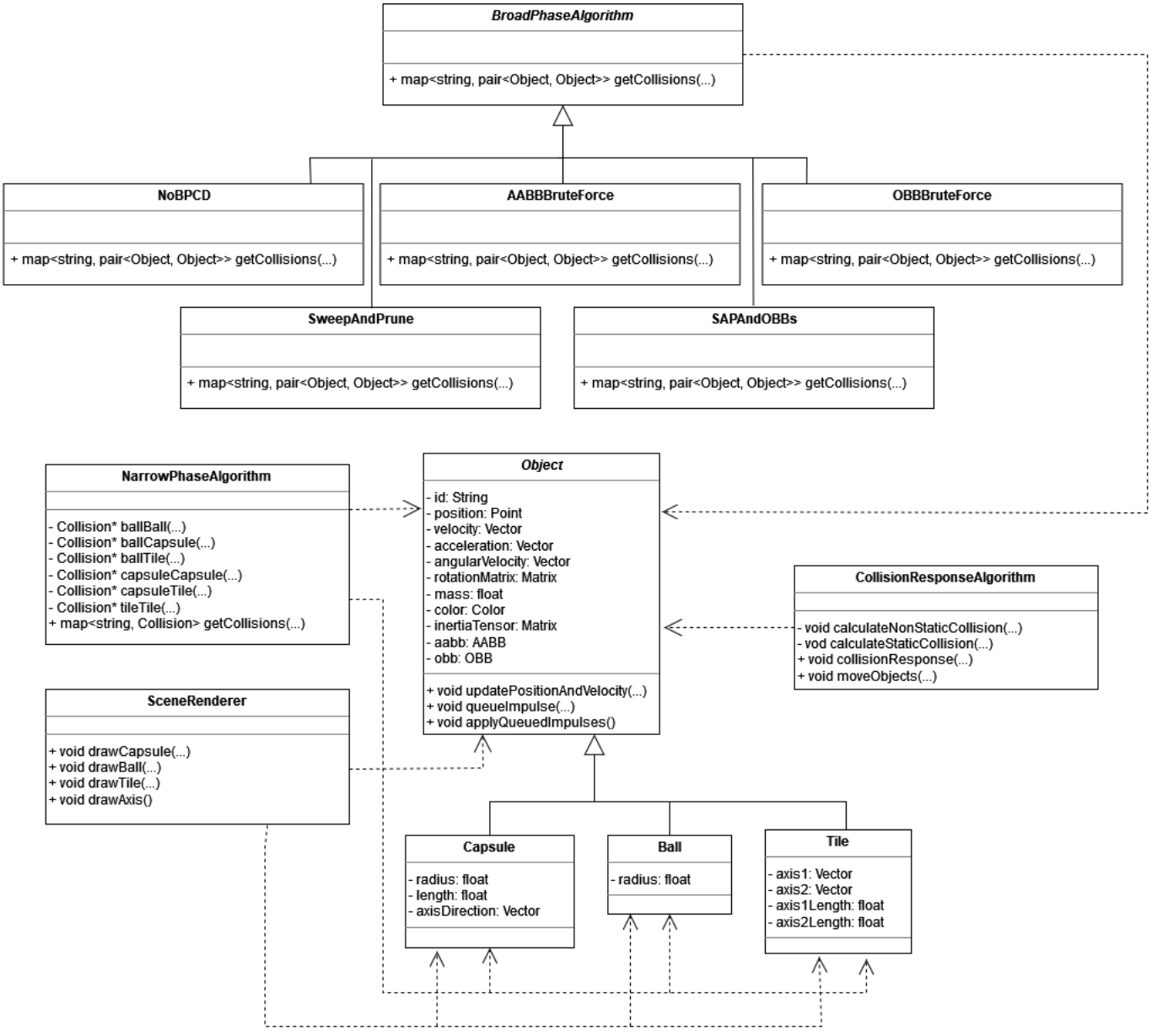


Figura 3.1: Diagrama de clases simplificado de Fingsics

Capítulo 4

Implementación

En este capitulo se presenta la implementación de Fingsics, explicando los diferentes modos de ejecución que tiene, y cómo se llevan a cabo la inicialización, la lectura y persistencia de datos, el renderizado, y el manejo de colisiones. Finalmente, se expone un pseudocódigo muy general de la aplicación.

4.1. Modos de ejecución

Se implementaron los siguientes cuatro modos de ejecución en el programa, que proveen diferentes funcionalidades:

- **Default**. Permite ver la simulación de la escena indicada y persistir sus resultados para ser reproducidos luego.
- **Replay**. Reproduce simulaciones de escenas previamente computadas y almacenadas. Permite adelantar y atrasar la simulación.
- **Benchmark**. Ejecuta una escena varias veces documentando tiempos de ejecución y otros datos para analizar su rendimiento. En este modo la escena no es dibujada, por lo que solo se ve una pantalla negra.
- **Test**. Ejecuta un conjunto de simulaciones en escenas con resultados conocidos para evaluar la correctitud de los cálculos de la simulación. Este modo fue creado con fines de desarrollo y tampoco dibuja la escena.

4.2. Datos de entrada

Fingsics toma datos de entrada de dos archivos: uno para leer la escena y otro para leer la configuración del programa. A continuación se explica cómo funcionan.

4.2.1. Lectura de escenas

Para describir una escena e importarla al programa se utilizan archivos de tipo XML, de los cuales se extraen datos utilizando la librería *tinyxml2*. Estos archivos permiten especificar todos los objetos de la escena con los siguientes parámetros:

- Tipo
- Dimensiones (dependen del tipo de objeto)
- Posición lineal inicial
- Velocidad lineal inicial
- Posición angular inicial
- Velocidad angular inicial
- Aceleración lineal (útil para simular la fuerza de gravedad, pero permite representar aceleraciones constantes en cualquier dirección durante toda la simulación)
- Coeficiente de elasticidad
- Si es estático o no
- Color

Adicionalmente, cada tipo de objeto puede ser configurado con parámetros que solo aplican a sí mismo:

- **Esferas** radio.
- **Cápsulas:** radio, largo (del cilindro y sin incluir las semiesferas), y la posibilidad de ser dibujada como un cohete. Esta última opción es puramente visual y no afecta los cálculos de colisiones.
- **Tiles:** largo y ancho.

También se puede configurar la posición y dirección inicial de la cámara utilizando este archivo, lo cual es útil para grabar varias escenas desde la misma perspectiva.

Todos los parámetros anteriores son opcionales, ya que para todos ellos hay un valor por defecto que se aplica en su ausencia. Sin embargo, para lograr buenos resultados es necesario que los objetos de la escena no se intersequen en su posición inicial, por lo que en la práctica es necesario especificar una posición distinta para cada objeto.

4.2.2. Configuración

Para controlar las opciones del programa que son ajenas a las escenas se utiliza un archivo de configuración llamado *config.txt*. Este contiene una lista de parámetros modificables para controlar diferentes opciones, entre ellas las siguientes:

- Selección de la escena (indicando el nombre del archivo .xml que la describe)
- Resolución de la ventana
- Cantidad de cuadros calculados por cada segundo de simulación
- Selección del modo de ejecución
- Selección del algoritmo de BPCD
- Número de cuadro en el cual terminar la simulación (se puede desactivar)
- Definición de los objetos (modifica la cantidad de polígonos con los que se dibujan los objetos curvos)
- Guardar o no una grabación de la escena en el formato de Fingsics
- Guardar o no una grabación de la escena en formato video
- Generar o no reportes sobre la simulación

4.3. Datos de salida

Se implementaron dos formas distintas de grabar simulaciones: en formato video y en un formato propio. Estas funcionalidades permiten reproducir en tiempo real algunas de las escenas que requieren demasiado procesamiento. También se implementó el registro de datos relacionados al rendimiento y resultados de las simulaciones en archivos *CSV*. Estas tres funcionalidades recaban los datos que les corresponden durante el ciclo principal del programa al final de la actualización de cada cuadro.

4.3.1. Grabación de escenas en formato video

Esta tarea es llevada a cabo por la clase *VideoRecorder*, la cual hace uso de la librería *ffmpeg* para generar videos. En cada cuadro toma la imagen emitida por OpenGL en la ventana de la simulación y la transfiere a la librería, la cual se encarga de realizar la codificación necesaria para formar el video.

4.3.2. Grabación de escenas en formato propio

La clase encargada de la persistencia de datos de escenas es *SceneRecorder*. Esto incluye tanto el guardado de datos en disco como leer los mismos datos para que se puedan reproducir nuevamente. Para lograr la reproducción correcta de las simulaciones es necesario serializar y guardar los siguientes datos de cada objeto:

- Su tipo
- Sus dimensiones (que varían en significado según el tipo de objeto, pero pueden ser representados por tres o menos números de punto flotante)
- Su color
- Las listas de posiciones y posiciones angulares que ocuparon en cada cuadro de la simulación

Los primeros tres datos son obtenidos cuando se cargan los objetos de la escena, mientras que las listas de posiciones y posiciones angulares se construyen a medida que avanzan los cuadros de la simulación y se guardan en memoria. Finalmente, cuando se termina la simulación, la misma clase se encarga de serializar todos estos valores utilizando su representación binaria y de guardarlos en un archivo *.dat*. Luego, cuando se desea reproducir los datos guardados, estos se leen del archivo y se decodifican para obtener la información de los objetos de la escena. Durante la reproducción no se realiza ninguno de los pasos del manejo de colisiones, y simplemente se sustituye la posición y la posición angular de todos los objetos por las del cuadro siguiente. Como los datos guardados por *SceneRecorder* ya contienen todo lo necesario para representar una escena, no se hace uso de *XmlReader* para cargar el archivo *XML* de la escena, y no es necesario conservar estos archivos una vez que se ha persistido la simulación deseada.

4.3.3. Reportes

Además de la persistencia de los datos propios a las simulaciones también se implementó el almacenamiento de registros con los tiempos de ejecución de cada etapa del manejo de colisiones y la cantidad de colisiones detectadas y procesadas por cada etapa. Los resultados son recabados al final de cada cuadro de la simulación y se guardan en un archivo *CSV* cuando esta finaliza, para que puedan ser procesados fácilmente por herramientas externas.

4.4. Interfaz y visualización

Para el manejo de la ventana de la simulación se usa la librería *SDL2*, para renderizar los cuadros en la ventana la librería *OpenGL*, y para facilitar la implementación del manejo de la cámara la librería *FreeGLUT*.

La representación de los objetos es muy similar a la que OpenGL utiliza para su sistema de coordenadas: las posiciones de sus centros se representan con un punto en tres dimensiones, mientras que sus rotaciones se representan con una matriz de rotación.

Como ya se mencionó, el dibujado de todos los elementos visuales de Fingsics se implementó en la clase *SceneRenderer*. Estos incluyen los objetos de la simulación, la representación de sus AABBs y OBBs con *wireframes* y la opción de dibujar las cápsulas como cohetes. Todos estos cuerpos son dibujados con mallas poligonales de cuadriláteros (*QUAD_STRIP* en OpenGL).

En cuanto a la cámara de la simulación, tiene dos modos:

- **Centrada:** en este modo siempre apunta al origen de coordenadas de la escena, siendo sólo modificable su posición.
- **Libre:** en este modo además se puede mover libremente la dirección de la cámara, lo cual aporta mayor libertad para recorrer la escena.

Al ejecutar una simulación, los controles permiten pausar la simulación, ejecutar en cámara lenta y habilitar o deshabilitar el dibujado de las AABBs y OBBs. Cuando el programa se corre en modo *replay*, se tiene acceso a controles adicionales, como avanzar y retroceder en la simulación en distintos intervalos (incluyendo de a un cuadro).

La interfaz gráfica también puede reportar los FPS a los que el programa está ejecutando. Los objetos en sí no poseen opciones de personalización a

excepción de sus dimensiones, propiedades mecánicas y color. Esto es así para mantener el enfoque en el manejo de colisiones.

4.5. Manejo de colisiones

En esta sección se explica el funcionamiento del manejo de colisiones en Fingsics, así como las implementaciones de los algoritmos que lo componen.

Como se mencionó previamente, dado que se manejan objetos de geometría simple, se decidió omitir la etapa de MPCD, que ocurriría entre la de BPCD y la de NPCD. Por lo tanto, el manejo de colisiones implementado involucra las siguientes tres etapas:

- **Broad Phase Collision Detection:** su objetivo es descartar pares de objetos de la lista de posibles colisiones de forma eficiente, utilizando sus volúmenes acotantes para concluir que no habrá colisión entre ellos.
- **Narrow Phase Collision Detection:** tomando como entrada la salida de la etapa anterior, y considerando únicamente esos pares de objetos, se realiza un chequeo para cada uno de ellos, verificando si hay colisión o no. En el caso de haber colisión, se calcula la información relativa a ella (determinación de la colisión).
- **Respuesta a las colisiones:** en base a la información de las colisiones se calcula y efectúa la respuesta correspondiente sobre los objetos.

En cada cuadro de la simulación, Fingsics dibuja los objetos en su posición actual, ejecuta un ciclo completo de las operaciones de cada etapa del manejo de colisiones y así obtiene las nuevas posiciones de los objetos para dibujarlos en el cuadro siguiente.

4.5.1. Broad Phase Collision Detection

Como se explicó antes, existen cinco opciones de algoritmos de BPCD. A continuación se exponen sus principales características.

NoBPCD

Este algoritmo existe con el propósito de simular la ausencia de una BPCD en el programa. Simplemente retorna todos los pares posibles de

objetos cuando se le invoca.

Fuerza bruta con AABBs y OBBs

Ambas variantes del algoritmo de fuerza bruta funcionan de la misma forma: se chequean los volúmenes acotantes de todos los pares de objetos posibles por intersecciones y solo se agrega un par a la lista de posibles colisiones si sus volúmenes acotantes se intersecan. La única diferencia entre ellas es que cada una utiliza el tipo de volúmenes acotantes que su nombre indica. Los algoritmos para la detección de colisiones AABB-AABB y OBB-OBB que se utilizaron son los descritos en la Sección 2.3.1. En el Listado 4.1 se muestra un pseudocódigo de los algoritmos de fuerza bruta.

Listado 4.1 Pseudocódigo de los algoritmos de BPCD de fuerza bruta

```
function fuerzaBruta(objetos):
    colisiones = []
    for objeto1 in objetos
        for objeto2 in objetos
            va1 = objeto1->volumenAcotante()
            va2 = objeto2->volumenAcotante()
            if testVolumenesAcotantes(va1, va2)
                colisiones.agregar(objeto1, objeto2)
            end if
        end for
    end for
    return colisiones
end fuerzaBruta
```

SAP

El algoritmo SAP fue implementado principalmente en base a [14]. Como en el caso de Fingsics no se agregan ni quitan objetos dinámicamente de las escenas, y solo hay una instancia de SAP que contiene a los objetos en todo momento, la cantidad de valores almacenados en las estructuras del SAP es constante. Esto significa que no es necesario implementar la inserción y remoción de objetos, y por lo tanto se decidió utilizar arreglos de largo constante para guardar los valores de las AABBs en los ejes (y no listas). Para guardar los pares de colisiones actuales se decidió utilizar la herramienta *std::map* de C++. Las claves están conformadas por los identificadores de los objetos que conforman el par y los valores son una referencia a dichos objetos.

4.5.2. Narrow Phase Collision Detection

Los algoritmos de NPCD implementados determinan si dos objetos genéricos de dos tipos dados colisionan o no, y en el caso de que colisionen, se devuelve la información relacionada a la colisión. Se implementó un algoritmo para cada par de tipos de objetos posible, exceptuando el par *Tile-Tile*. Esta excepción se debe a que en el programa los objetos de tipo tile deben ser estáticos, y no tiene sentido que dos objetos estáticos choquen entre sí. Debido a esto, si dos tiles se intersecaran, simplemente se atravesarían.

En el resto de esta subsección se explica cada uno de los algoritmos de NPCD implementados, y en el Anexo B se incluye un pseudocódico de cada uno de ellos.

Esfera-Esfera

Este es el algoritmo de NPCD más simple. Para determinar si dos esferas colisionan, basta con verificar que la distancia entre sus centros es menor a la suma de sus radios. Si se intersecan, el punto de colisión se encuentra en el segmento que une los centros de las esferas a una distancia de sus centros proporcional a la relación entre sus radios. La normal tiene la dirección del vector que une a los centros de las esferas, y la distancia de interpenetración se puede calcular como la diferencia entre la suma de los radios de las esferas y la distancia entre sus centros.

Esfera-Cápsula

Para los algoritmos de NPCD con cápsulas, fue útil separarlas en tres componentes: un cilindro y dos esferas centradas en sus bases, como se muestra en la Figura 4.1.

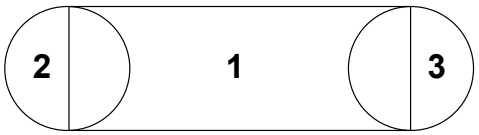


Figura 4.1: Cápsula separada en tres partes

Para verificar si hay una colisión entre una esfera y una cápsula, se chequea por separado si la esfera colisiona con alguna de las tres partes

de la cápsula. El algoritmo para verificar si hay colisión con alguna de sus dos esferas es el explicado anteriormente. Se desarrolló entonces un algoritmo para las primitivas Esfera-Cilindro, en base a que ocurre una colisión entre estas dos primitivas únicamente si se cumplen las siguientes dos condiciones:

1. La proyección del centro de la esfera sobre la recta que pasa por el centro del cilindro debe estar adentro del cilindro.
2. La distancia entre el centro de la esfera y su proyección sobre la recta que pasa por el centro del cilindro debe ser menor a la suma del radio de la esfera y el del cilindro.

En la Figura 4.2 se muestran tres ejemplos relacionados a esto. Es importante aclarar que podría haber una colisión con el cilindro aunque la primera condición no se cumpla, pero en ese caso la esfera también colisionaría con la semiesfera correspondiente a ese lado de la cápsula, por lo que no se precisaría detectar la colisión con el cilindro.

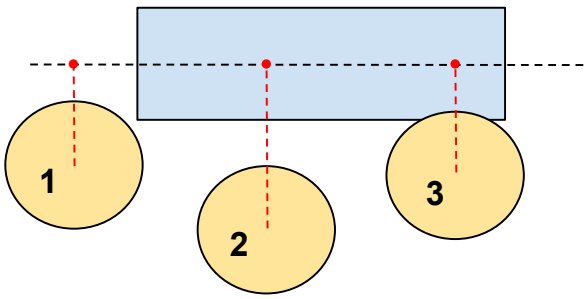


Figura 4.2: Ejemplos de la detección de colisiones entre una esfera y un cilindro. La esfera 1 no colisiona porque no cumple la primera condición, la esfera 2 no colisiona porque no cumple la segunda condición, y la esfera 3 colisiona ya que cumple ambas condiciones.

De haber una colisión con el cilindro, el punto de colisión se encuentra en el segmento que une al centro de la esfera con su proyección, a una distancia de su centro proporcional a la relación entre su radio y el del cilindro. La normal tiene la dirección del vector que une al centro de la esfera con su proyección, y la distancia de interpenetración se puede calcular restándole a la suma de los dos radios la magnitud del vector recién nombrado.

Cápsula-Cápsula

Dadas dos cápsulas, lo primero que se hace para verificar si se intersecan es encontrar los puntos más cercanos entre las rectas que pasan por el centro de cada una de ellas. Dichos puntos son los únicos pertenecientes a las rectas que cumplen que el segmento que los une es perpendicular a ambas rectas, como se muestra en la Figura 4.3. En base a estas restricciones se puede plantear un sistema de ecuaciones de tamaño 3x3 (que puede ser resuelto utilizando el método de eliminación de Gauss-Jordan) cuya solución permite encontrar estos dos puntos. Por una explicación completa sobre cómo plantear el sistema se puede consultar [32].

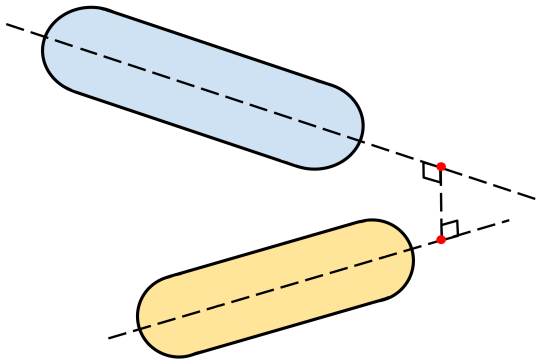


Figura 4.3: Puntos más cercanos entre las rectas que pasan por los centros de dos cápsulas. En este caso ambos puntos quedan por fuera de la cápsula que les corresponde.

Conociendo estos dos puntos, se calcula la distancia entre ellos (que coincide con la distancia mínima entre las rectas), y si esta es mayor a la suma de los radios de las cápsulas, entonces ya se sabe que no hay intersección.

El resto del algoritmo es en base a la descomposición de las cápsulas mostrada antes en la Figura 4.1, y separando en dos casos:

1. Si los dos puntos que se calcularon anteriormente caen dentro del cilindro que compone a la cápsula que les corresponde, ya se puede afirmar que hay una colisión entre los cilindros que componen las cápsulas. En este caso el punto de colisión estará sobre el segmento que une los dos puntos calculados, a una distancia de cada uno proporcional a la relación entre los radios de las cápsulas. La normal tiene la dirección de este segmento, y la distancia de interpenetración es la suma de los radios de las cápsulas menos el tamaño del segmento.

2. Si alguno de los dos puntos cae fuera de su cilindro, entonces se sabe que los cilindros no colisionan entre sí, pero aún podría haber una colisión entre una esfera de una cápsula y el cilindro de la otra, o entre una esfera de cada cápsula. Primero se calcula de qué lado de cada cápsula está su punto más cercano a la otra cápsula, y con esto se sabe cuál de sus dos esferas se debe verificar. Sabiendo esto, y utilizando los algoritmos de Esfera-Esfera y Esfera-Cilindro que se explicaron anteriormente, se chequea si existe alguna colisión entre las dos esferas determinadas, o entre una de ellas y el cilindro de la otra cápsula.

Es importante aclarar que, en el caso particular de que las dos cápsulas sean paralelas entre sí, el sistema para encontrar los dos puntos más cercanos entre sus rectas tiene infinitas soluciones. Es por esto que este caso se maneja por separado, verificando al principio si son paralelas. En caso de que lo sean, se chequea si se da una colisión entre alguna de las esferas de una cápsula y la otra cápsula, o al revés, y en base a esto se determina si hay una colisión o no. Si hay una colisión cuando las cápsulas son paralelas, entonces la colisión se dará a lo largo de un segmento (y no en un único punto). Para manejar esto se utiliza como punto de colisión el centro del segmento, lo cual funciona bastante bien.

Esfera-Tile

El primer paso para chequear una colisión entre una esfera y una tile es chequear si existe una colisión entre la esfera y el plano que contiene a la tile. Para realizar esto se calcula la distancia mínima entre el centro de la esfera y el plano. Si esta distancia es mayor al radio de la esfera se sabe que el plano que contiene la tile y la esfera no colisionan, y por lo tanto la esfera y la tile tampoco.

Asumiendo que la distancia anterior es menor al radio de la esfera es posible que exista una colisión. Para detectarla es necesario encontrar la proyección del centro de la esfera sobre el plano que contiene a la tile. Esta proyección se puede encontrar en uno de nueve cuadrantes con respecto a los bordes de la tile, los cuales están representados en la Figura 4.4. Dependiendo del cuadrante en el que la proyección se encuentre se sabe con qué parte de la tile se podría llegar a encontrar una colisión.

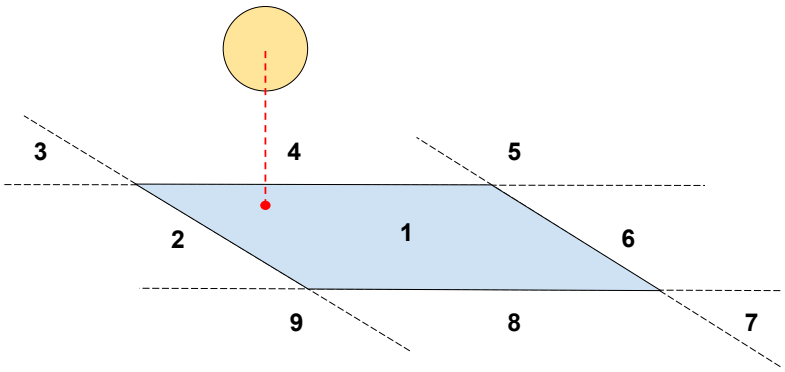


Figura 4.4: Proyección del centro de una esfera sobre el plano de una tile.

El caso más simple se da cuando la proyección del centro de la esfera se encuentra en el cuadrante número 1, ya que este representa a la propia tile. En este caso, como la esfera está a menor distancia del plano que su radio se sabe que existe una colisión entre la esfera y la tile, y que el punto de colisión es la proyección de la esfera sobre el plano. La normal de la colisión es la normal del plano y la distancia de interpenetración es la diferencia entre el radio de la esfera y la distancia entre el centro de la esfera y su proyección sobre el plano.

Si la proyección se encuentra en cualquiera de los cuadrantes con número par se sabe que en caso de existir una colisión, el punto de colisión se encontrará sobre el borde de la tile que delimita ese cuadrante. Para averiguar si existe la colisión se calcula la distancia mínima entre el centro de la esfera y la recta correspondiente al cuadrante. Si esta distancia es mayor al radio de la esfera no hay colisión entre la tile y la esfera, y si es menor entonces se sabe que sí colisionan. En este último caso, el punto de colisión es la proyección del centro de la esfera sobre la recta, la distancia de interpenetración es la resta entre el radio de la esfera y la distancia del centro de la esfera a su proyección en la recta, y la normal de la colisión es un vector cuya dirección se alinea con el centro de la esfera y su proyección en la recta.

Por úlitmo, resta considerar los cuadrantes impares que se encuentran fuera de la tile. Su resolución es similar a la de los cuadrantes pares, pero en vez de considerar el borde de la tile que delimita el cuadrante se considera el punto en la esquina de la tile que se encuentra en el cuadrante. Entonces, el primer paso es hallar la distancia de esta esquina al centro de la esfera.

Si es mayor al radio de la esfera no hay colisión. Si es menor se sabe que hay una colisión, cuyo punto es la esquina de la tile, cuya distancia de interpenetración es la resta del radio de la esfera y la distancia entre la esquina de la tile y el centro de la esfera, y cuya normal es un vector que tiene una dirección que se alinea con el centro de la esfera y la esquina de la tile.

Cápsula-Tile

Este algoritmo es bastante complejo ya que entre una cápsula y una tile se pueden dar muchos casos diferentes: la cápsula puede chocar contra la parte superior de la tile en uno o más puntos, contra cualquiera de sus aristas en uno o más puntos o contra cualquiera de sus esquinas.

Al igual que en los otros algoritmos para las cápsulas, fue útil considerar por separado sus dos esferas y su cilindro. Lo primero que hace el algoritmo es verificar si las esferas colisionan con la tile, con el algoritmo Esfera-Tile explicado anteriormente. En base a esto hay tres casos:

1. Ambas esferas colisionan con la tile. Esto significa que la colisión se da a lo largo de toda la cápsula, y sabiendo esto se puede determinar la colisión (se da en el punto medio entre las dos colisiones encontradas).
2. Ninguna de las esferas colisiona con la tile. En este caso resta verificar si el cilindro colisiona con la tile.
3. Una de las esferas colisiona con la tile. En este caso también se verifica si el cilindro colisiona con la tile. Si no lo hace, entonces se retorna la colisión con la esfera, y si lo hace, se calcula el resultado final en base a estas dos colisiones.

Resta entonces explicar el algoritmo para detectar una colisión entre el cilindro de la cápsula y la tile. Lo que interesa en este algoritmo es detectar las colisiones con los bordes de la tile, ya que si la cápsula colisiona también con la parte del centro de la tile, esto ya se habrá detectado con una de las esferas o se puede representar como el punto medio de las colisiones con los bordes.

Las colisiones del cilindro con los bordes de la tile, que de haber pueden ser una o dos, se detectan de forma similar a la detección de colisiones entre los cilindros de dos cápsulas, explicada en el algoritmo Cápsula-Cápsula (el borde de una tile se podría interpretar como un cilindro de radio cero).

Una vez que se tienen todas las colisiones, se promedian todos sus puntos de contacto y así se obtiene la colisión final.

4.5.3. Respuesta a las colisiones

Como se explicó previamente, la respuesta a las colisiones se hizo utilizando el enfoque basado en impulsos, pero se implementaron además algunas modificaciones y heurísticas para mejorar su funcionamiento.

No se implementaron coeficientes de restitución (que miden el grado de conservación de energía cinética tras un choque entre dos cuerpos de ciertos materiales), porque en Fingsics no se manejan materiales para los objetos. Para poder simular diferentes tipos de colisiones, se implementó en los objetos un coeficiente diferente, que fue llamado coeficiente de elasticidad. Cuando dos objetos colisionan, sus coeficientes se utilizan para realizar un promedio entre ellos y utilizar ese valor como coeficiente de restitución.

Por otro lado, un problema que hubo que resolver es que cuando dos objetos colisionan y se aplica una respuesta, puede suceder que en el siguiente cuadro de la simulación sigan colisionando, ya que no transcurrió suficiente tiempo para que dejen de interpenetrarse. Esto ocurre porque la colisión es detectada una vez que los objetos ya están intersecándose, y trae un problema porque la respuesta debe aplicarse una única vez pero la colisión será detectada en múltiples cuadros consecutivos. Una forma posible de manejar esto es mover a los objetos que se están interpenetrando para que dejen de hacerlo antes de aplicar la respuesta a la colisión. Esto tiene una limitación muy grande: al desplazarlos es posible que entren en contacto con otros objetos cercanos, lo cual derivaría en la misma situación. Otra solución, la cual resulta bastante intuitiva, es aplicar la respuesta solo si el cuadro actual es el primero en el que se detecta la colisión. Sin embargo, esta técnica tampoco logra buenos resultados, ya que cuando hay muchos objetos juntos entre sí, a veces la respuesta no alcanza para que algunos dejen de estar penetrándose entre sí, y ocurre bastante seguido que objetos quedan dentro de otros por intervalos largos de tiempo.

Dados estos problemas, se ideó e implementó la siguiente heurística: la respuesta a la colisión es aplicada únicamente si en el cuadro anterior al actual la distancia de interpenetración era menor a la actual. De esta forma, si dos objetos que se interpenetran ya se están separando, no se vuelve a

aplicar la respuesta, pero si se aplicó una respuesta y los objetos aún siguen intersecándose más que antes, se les vuelve a aplicar impulsos para que se separen. Esta heurística logra resultados aceptables y a simple vista parece más fiel que los enfoques antes descritos, pero hay casos donde los objetos afectados no logran separarse adecuadamente, principalmente cuando se dan varias colisiones simultáneas a la vez. Esto se debe a que por más que se apliquen impulsos sucesivos a objetos que no se están separando, si estos se mueven lentamente los impulsos resultantes también serán muy pequeños, por lo que no experimentarán cambios importantes de velocidad y se separarán muy lentamente.

Otro problema usual es entonces el de las colisiones simultáneas. En estos casos calcular y aplicar los impulsos de forma secuencial para todas las colisiones no funciona bien.

Sea una cápsula que está quieta y es chocada por dos esferas con velocidades opuestas, una de cada lado, como se muestra en la Figura 4.5. Si se aplicara primero a la cápsula el impulso resultante de la colisión contra una de las esferas y luego se calculara el impulso de la segunda colisión, entonces este segundo cálculo se haría con la velocidad modificada de la cápsula (resultante de la primera colisión), y la respuesta sería errónea. Es por esto que se deben calcular primero todos los impulsos del sistema, y una vez calculados todos, aplicarlos. Esto es lo que se hace en Fingsics.

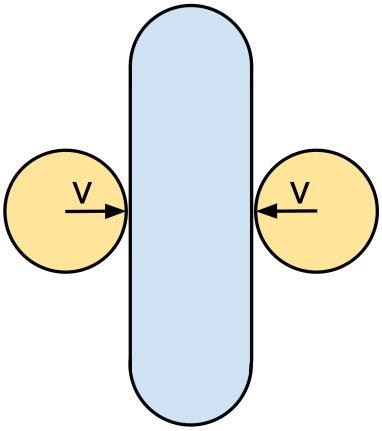


Figura 4.5: Ejemplo de colisión simultánea

Sin embargo, a pesar de que esto mejora los resultados, no es una solución del todo correcta. En el caso recién descrito, también ocurre que el impulso de cada una de las esferas debería ser transferido a la otra a través de la

cápsula (de cierta forma las esferas, que no se tocan, interactúan entre sí). Si las dos esferas tuvieran la misma masa y magnitud en su velocidad, entonces tras chocar con la cápsula el resultado debería ser que las velocidades de las dos se inviertan. Como se mencionó en la Sección 2.5, este problema es muy complejo y costoso, por lo que se decidió no resolverlo en Fingsics, por lo que la transferencia de los impulsos entre objetos no se da.

4.6. Pseudocódigo de Fingsics

Una vez explicados todos los componentes principales de Fingsics, resta ver cómo interactúan entre sí. El Listado 4.2 muestra las acciones más importantes de la inicialización, el ciclo principal y la finalización del programa.

Listado 4.2 Pseudocódigo de Fingsics

```
function main():
    objetos = leerConfigYEscena()
    algoritmoBPCD, algoritmoNPCD = inicializarAlgoritmos()

    while not salir
        finalCuadro = ahora() + duracionCuadro

        if not pausa
            colisionesBPCD = algoritmoBPCD.obtenerColisiones(objetos);
            colisiones = algoritmoNPCD.obtenerColisiones(colisionesBPCD);
            aplicarRespuestas(colisiones);
            moverObjetos(objetos);
        end if

        dibujarEscena(objetos)

        if not pausa
            almacenarCuadroParaGrabaciones()
            almacenarReportesDeCuadro()
        end if

        chequearYAplicarInputsDelUsuario()

        if ahora() < finalCuadro
            esperar(finalCuadro - ahora())
        end if
    end while

    persistirGrabacionesYReportes()
end main
```

Capítulo 5

Resultados

En este capítulo se presentan diferentes resultados de experimentos realizados utilizando Fingsics. En la Figura 5.1 se muestra una imagen del programa desarrollado, y para ver algunos videos de demostración se puede acceder al siguiente enlace:

https://www.youtube.com/channel/UCg4zVIRDI5ksnsaXP7908lA/videos

Figura 5.1: Imagen de ejemplo de Fingsics

Todos los resultados mostrados fueron obtenidos en una PC con las especificaciones mostradas en las Tablas 5.1 y 5.2.

Tabla 5.1: Especificaciones del hardware utilizado para las pruebas

GPU	Nvidia Geforce RTX 3070
CPU	Intel Core i7-10700
RAM	16GB, 3200mhz

Tabla 5.2: Especificaciones del software utilizado para las pruebas

Sistema Operativo	Windows 10 Education 64 bits
Compilador	Visual Studio Community 2019

Se comenzará analizando el desempeño de los diferentes algoritmos de BPCD y NPCD implementados, luego se hará un análisis general del desempeño de Fingsics, y finalmente se discutirá como se compara con Bullet, otro motor físico.

5.1. Análisis de los algoritmos implementados

Recordando lo explicado anteriormente, el objetivo de un algoritmo de BPCD es descartar colisiones de forma eficiente y así alivianar el trabajo realizado por la etapa de NPCD. Su funcionamiento se basa en chequear colisiones entre los volúmenes acotantes de los objetos, tarea que es mucho menos costosa que chequear las colisiones entre los objetos. Para evaluar el desempeño de un algoritmo de BPCD, interesan entonces dos cosas:

1. Qué tan costosa es su ejecución
2. Qué tanto reduce la cantidad de chequeos que deben realizarse en la etapa de NPCD

Midiendo el tiempo total de detección de colisiones al utilizar los diferentes algoritmos de BPCD (y al no utilizar ninguno), se puede ver si la reducción de chequeos de NPCD de un algoritmo de BPCD justifica su costo de ejecución. Es interesante observar que la utilización de un algoritmo de BPCD podría empeorar el rendimiento, si este no lograra descartar suficientes colisiones.

A su vez, además de analizar el tiempo total de ejecución de la detección de colisiones, es interesante analizar los valores de las variables que aparecen en la ecuación 5.1 (explicada en la Sección 2.3.1), que representan lo siguiente:

- **T -** Tiempo total de la detección de colisiones
- **b -** Cantidad de chequeos de colisiones entre volúmenes acotantes
- **B -** Tiempo del chequeo de colisiones entre un par de volúmenes acotantes
- **o -** Cantidad de chequeos de colisiones entre objetos
- **O -** Tiempo del chequeo de colisiones entre un par de objetos

$$T = bB + oO \tag{5.1}$$

Los tiempos promedio de chequeo entre volúmenes acotantes (B) y entre objetos (O) son independientes del algoritmo de BPCD que se use, por lo que se analizarán primero.

Todos los resultados que se muestran son promedios obtenidos ejecutando tres veces cada escena con 60 detecciones y respuestas a las colisiones por segundo de simulación, y durante 600 cuadros, con las funcionalidades de grabado y persistencia de datos apagadas. Para obtener los resultados se utilizó el modo *benchmark* del programa, que deshabilita el renderizado de las escenas y quita el sleep que se hace al final de cada cuadro, dado que al estar evaluando los algoritmos no interesa mantener un tiempo fijo por cuadro.

5.1.1. Algoritmos de NPCD y chequeos entre volúmenes acotantes

Para evaluar estos algoritmos se midió la cantidad total de chequeos y el tiempo total invertido en cada tipo de chequeo a lo largo de toda una escena, calculando luego el promedio. Los resultados son los que se muestran en las Tablas 5.3 y 5.4. Debido a que no se implementó la detección de colisiones para el par *Tile-Tile*, esta entrada no está presente en la segunda tabla.

Tabla 5.3: Tiempo de chequeo de colisiones entre volúmenes acotantes (B)

Primitivas	Tiempo promedio de chequeo
AABB-AABB	12ns
OBB-OBB	25ns

Como era de esperarse, el tiempo que insume detectar una colisión entre OBBs es mayor que entre AABBs, siendo aproximadamente el doble. Otro

Tabla 5.4: Tiempo de chequeo de colisiones entre primitivas (O)

Primitivas	Tiempo promedio de chequeo
Esfera-Tile	25ns
Esfera-Esfera	101ns
Esfera-Cápsula	291ns
Cápsula-Cápsula	546ns
Cápsula-Tile	1733ns

resultado esperado, es que los tiempos de chequeos entre objetos son todos mayores o iguales a los tiempos de chequeos entre volúmenes acotantes.

Por otro lado, los valores de O están ordenados según el nivel de complejidad de los objetos que chequean (siendo la esfera la más simple y el tile el más complejo), a excepción del par *Esfera-Tile*. Esto es razonable, dado que en la escena en la cual se calcularon estos valores (que se describirá al hablar de los algoritmos de BPCD), la gran mayoría de los chequeos de este tipo consisten únicamente en verificar que la esfera no colisiona con la tile porque ni siquiera colisiona con el plano que la contiene, y esto implica únicamente realizar un producto interno entre vectores.

5.1.2. Algoritmos de BPCD

Para cada algoritmo implementado se reportarán los valores que aparecen en la ecuación 5.1: el tiempo total de ejecución (T), la cantidad de chequeos entre volúmenes acotantes (b), y la cantidad de chequeos entre objetos (o). También se analizará el tiempo total de la etapa de BPCD (bB) y el tiempo total de la etapa de NPCD (oO).

Como se mencionó en la Sección 2.3.1, el algoritmo SAP realiza los chequeos entre volúmenes acotantes de forma implícita, por lo que no se puede calcular la cantidad de chequeos ni su costo. En este caso, el análisis se hará en base a la ecuación 5.2, donde el término bB se sustituye por C, que representa el tiempo total del algoritmo de BPCD (SAP).

$$T = C + oO \tag{5.2}$$

Para el algoritmo de SAP + OBBs, el BPCD se ejecuta en dos etapas, primero se descartan colisiones con SAP y luego se aplica una fuerza bruta con OBBs. Únicamente en la primera etapa no se puede calcular la cantidad

de chequeos entre volúmenes acotantes. En este caso se puede escribir la ecuación del tiempo de detección de colisiones utilizando tres términos: uno para el costo del SAP (C), uno para el costo de la fuerza bruta con OBBs (bB), y uno para los chequeos entre objetos (oO), como se puede ver en la ecuación 5.3. En este caso el costo del BPCD corresponde a los dos primeros términos de la ecuación.

$$T = C + bB + oO \tag{5.3}$$

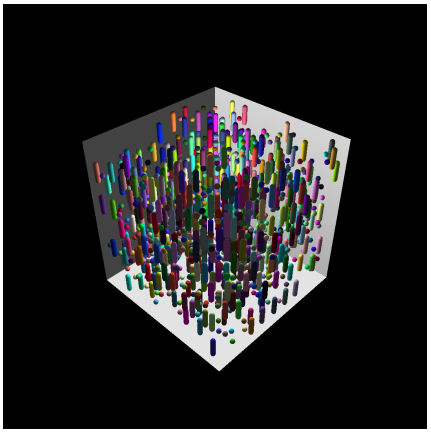
En el resto de esta sección se compara a los algoritmos entre sí utilizando escenas con diferentes cantidades de objetos, y luego se entra en detalle sobre el algoritmo de SAP + OBBs, cómo se compara con SAP, y cómo afecta a su desempeño el largo de los objetos de la escena.

Comparación de los algoritmos

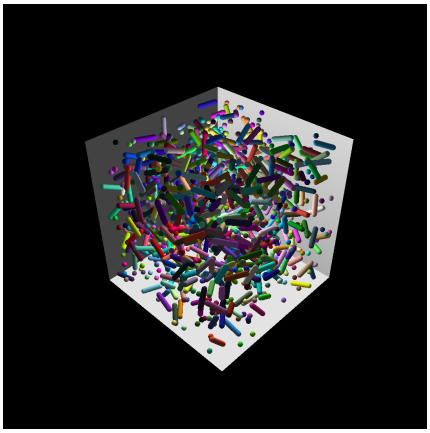
Para comparar los algoritmos, se decidió ejecutarlos en escenas de diferentes cantidades de objetos con una configuración similar: cápsulas y esferas moviéndose encerradas en una caja, en iguales cantidades. Todas las cápsulas y esferas tienen las mismas dimensiones: las esferas tienen un radio de 0.3 y las cápsulas un radio de 0.3 y un largo de 2. Para crear las escenas se generaron grillas tridimensionales equiespaciadas y cuadradas, con cápsulas y esferas intercaladas, las cuales contienen el triple de celdas que las cantidades finales de objetos de la escenas. Luego, se remueven objetos de las grillas aleatoriamente hasta obtener la cantidad final de objetos con el objetivo de lograr un balance adecuado entre la cantidad de movimiento y la cantidad de colisiones en la escena. Las cajas en las escenas son ligeramente más grandes que las grillas de objetos que encierran y todas las cápsulas y esferas tienen una velocidad inicial aleatoria. La Figura 5.2 muestra ejemplos de algunas de las escenas utilizadas y ejemplos individuales de los objetos con las proporciones recién descritas.

Se comenzará analizando, para cada algoritmo, la cantidad de chequeos de BPCD (b) y la cantidad de chequeos de NPCD (o).

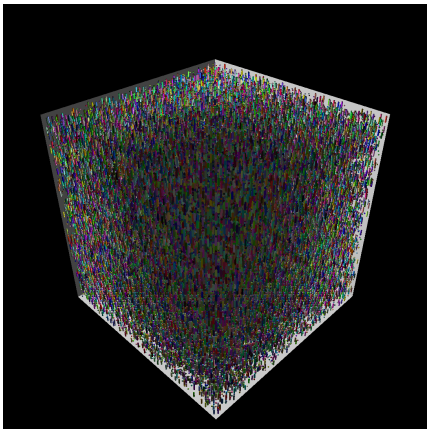
En cuanto a la cantidad de chequeos de BPCD, si no se utiliza ningún algoritmo en esta etapa evidentemente este valor será 0. En caso de utilizar como BPCD uno de los dos algoritmos de fuerza bruta, que chequean todos los pares posibles de volúmenes acotantes, el valor de b será siempre $(n^2-n)/2$



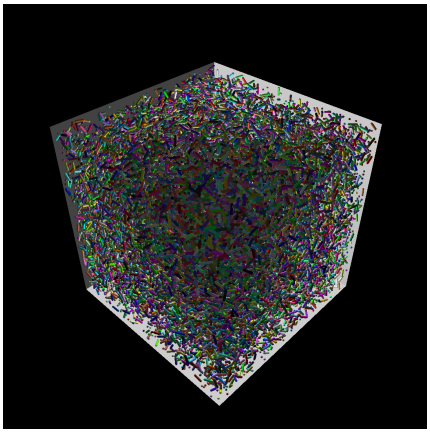
(a) 1×10^3 objetos, posiciones iniciales



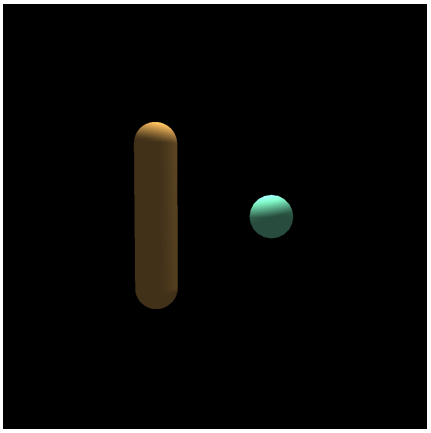
(b) 1×10^3 objetos, en movimiento



(c) $1 \times 10^{4.5}$ objetos, posiciones iniciales



(d) $1 \times 10^{4.5}$ objetos, en movimiento



(e) Una cápsula y una esfera con las dimensiones utilizadas

Figura 5.2: Escenas y objetos utilizados en la evaluación. Las cajas tienen seis caras, pero tres son transparentes para poder ver la simulación.

(se resta n porque no se chequean los volúmenes contra sí mismos y se divide entre dos para remover duplicados). Por otro lado, como se mencionó antes, para los dos algoritmos que utilizan SAP no se puede calcular la cantidad de chequeos, pero aún así la finalidad de todo algoritmo de BPCD es descartar colisiones del conjunto total de colisiones posibles. Se mostrará entonces la cantidad total de pares posibles de objetos para cada escena, $(n^2-n)/2$, con el fin visualizar qué tan eficaces son los algoritmos descartando colisiones de este conjunto. En las gráficas este valor tiene el nombre *Chequeos BPCD*, ya que corresponde a la cantidad inicial de pares de objetos que un algoritmo de BPCD debe refinar. El otro valor no obtenido de las BPCD que aparece en las gráficas es la cantidad real de colisiones en la escena, que al igual que el anterior es independiente de los algoritmos pero permite evaluarlos mejor.

Las pruebas sobre los algoritmos fueron realizadas aumentando progresivamente la cantidad de objetos hasta que la duración de la ejecución de la escena fuera mayor a una hora. Debido a esto, en las gráficas las lineas de algunos algoritmos (los menos eficientes) no tienen valores para algunas de las cantidades de objetos más grandes que se utilizaron.

En la Figura 5.3 se pueden ver los resultados obtenidos. La cantidad de colisiones es siempre el valor más bajo, ya que en todos los casos los chequeos que se hacen corresponden a las colisiones que efectivamente se dan y además algunas que no. Las dos gráficas con valores más altos son la de los chequeos de BPCD, y los chequeos de NPCD al no utilizar BPCD, que son iguales. Esto tiene sentido ya que ambos casos representan todos los posibles pares de colisiones.

En el medio están las gráficas de cantidad de chequeos de NPCD para los diferentes algoritmos de BPCD. Los algoritmos menos eficaces en descartar colisiones son SAP y fuerza bruta con AABBs. De hecho, sus valores son exactamente iguales en todas las pruebas, lo cual tiene sentido porque ambos algoritmos descartan únicamente aquellos pares de objetos cuyas AABBs colisionan. El siguiente algoritmo más eficaz es la fuerza bruta con OBBs. Como se vio en la Sección 2.3.1, los OBBs son mejores aproximando objetos que las AABBs, por lo que también es esperable que descarten más colisiones. Finalmente, el algoritmo que descarta más colisiones en total es SAP + OBBs, lo cual se debe a que hay colisiones descartadas por las AABBs que las OBBs no descartan y viceversa. SAP + OBBs es el algoritmo más eficaz por ser el único entre los evaluados que descarta ambos tipos.

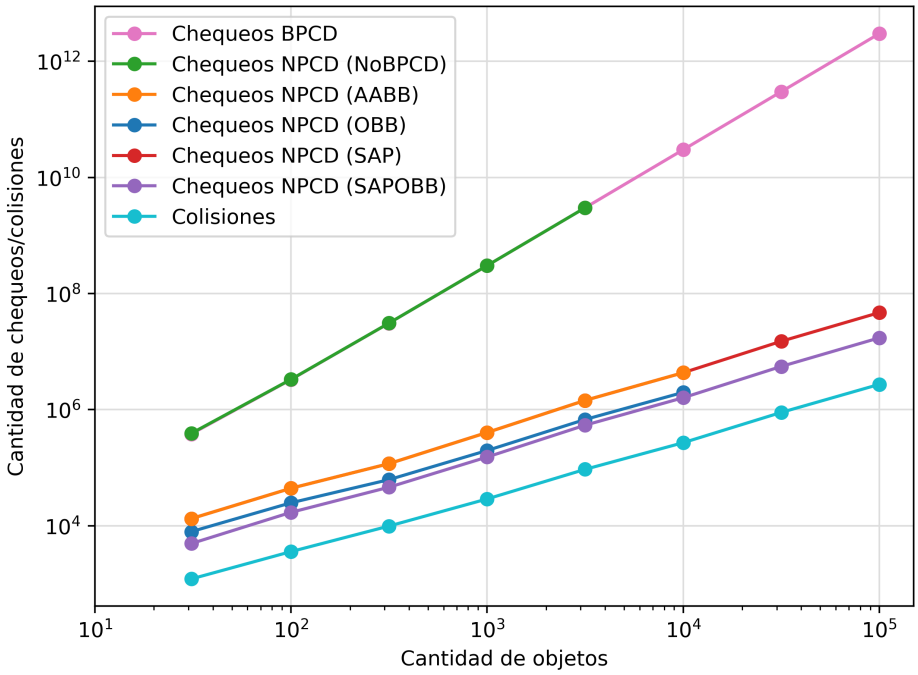


Figura 5.3: Cantidad de colisiones y chequeos de BPCD y NPCD, para los diferentes algoritmos de BPCD variando la cantidad de objetos.

Por otro lado, en la Figura 5.4 se muestran los tiempos totales de BPCD y NPCD para los diferentes algoritmos. Sobre el tiempo de los BPCD, se puede observar que los algoritmos que más demoran son las fuerzas brutas, y que la que utiliza OBBs demora un poco más que la de AABBs. También se puede ver que a pesar de que SAP + OBBs realiza una etapa más que SAP, el aumento en tiempo de ejecución del BPCD es muy poco. En cuanto a los tiempos de NPCD, como es de esperarse, no utilizar algoritmo de BPCD resulta en los valores más altos. Los tiempos de NPCD que obtienen SAP y la fuerza bruta con AABBs son muy similares, y esto se debe a que, como se mencionó antes, estos algoritmos resultan en los mismos chequeos de NPCD. Finalmente, al utilizar la fuerza bruta con OBBs, como se descartan más posibles colisiones el tiempo de NPCD se reduce más, y al hacer la combinación de SAP + OBBs se descartan aún más colisiones y el tiempo es el menor de todos.

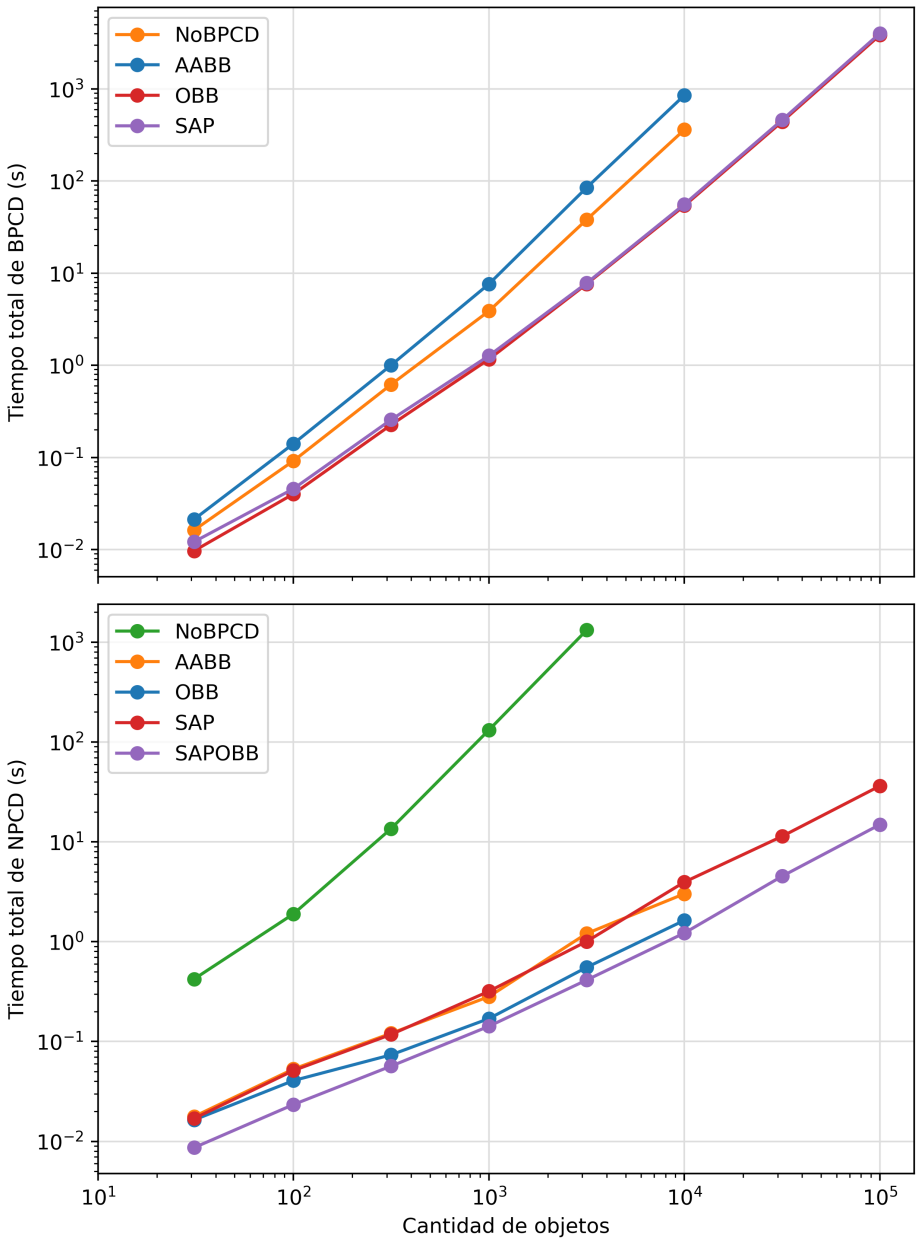


Figura 5.4: Tiempos totales de BPCD y NPCD, para los diferentes algoritmos de BPCD variando la cantidad de objetos.

Para concluir la comparación se analizará el tiempo total de ejecución de la detección y respuesta a las colisiones. En la Figura 5.5 se puede ver la gráfica con los resultados obtenidos. Como se esperaba, no utilizar un algoritmo de BPCD resulta en el peor tiempo de ejecución en todos los casos. En un segundo nivel están los algoritmos de fuerza bruta, donde se da el salto más grande en términos de rendimiento, y la utilización de AABBs resulta más eficiente que la de OBBs. A pesar de que al utilizar OBBs se reduce más la cantidad de chequeos de NPCD (ya que estos aproximan mejor al objeto), la reducción no es suficientemente grande como para justificar la complejidad de los chequeos entre OBBs. Finalmente, los algoritmos de SAP y SAP + OBBs son los más eficientes, obteniendo resultados muy parecidos. Se entrará más en detalle sobre cómo se comparan estos dos últimos a continuación.

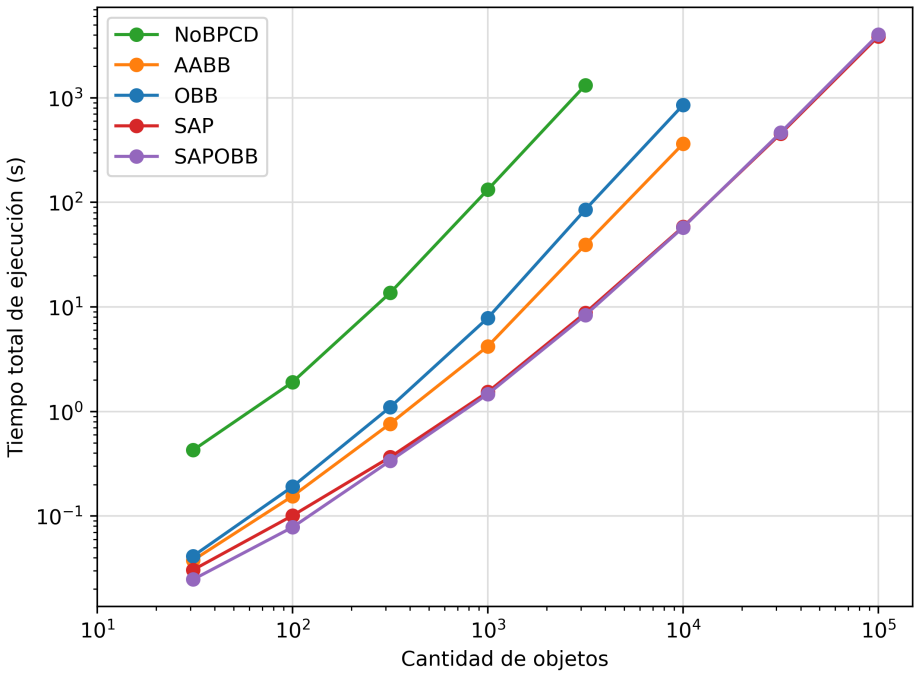


Figura 5.5: Tiempos totales de ejecución para los diferentes algoritmos de BPCD, variando la cantidad de objetos

Algoritmo de SAP + OBBs

En la Sección 2.3.1, al hablar de las AABBs, se explicó que este volumen acotante no es bueno para aproximar objetos alargados. Como las OBBs sí logran una buena aproximación en estos casos, interesa evaluar el desempeño del algoritmo SAP + OBBs variando el largo de los objetos que componen una escena. Para tener una idea de qué tan bien funciona la etapa del algoritmo que utiliza las OBBs, se comparará a este contra el algoritmo SAP básico (que utiliza únicamente AABBs). También se analizará cómo se distribuye el tiempo de detección de colisiones entre la etapa de SAP, la de OBBs, y la de NPCD.

Se diseñaron escenas similares a las utilizadas hasta ahora para evaluar, pero sin esferas, con únicamente cápsulas chocando dentro de una caja. Dentro de cada una de las cinco escenas que se evaluarán, todas las cápsulas son de igual tamaño, pero entre las diferentes escenas la razón entre el diámetro de las cápsulas y su largo varía. Como ejemplo, una razón de $2:1$ significa que el largo de la cápsula (incluyendo las semiesferas) es igual al doble de su diámetro, por lo que una mayor razón significa que la cápsula es más alargada. Se espera que al aumentar esta razón, el algoritmo de SAP + OBBs tenga un mejor desempeño en comparación al algoritmo SAP. La Figura 5.6 muestra las cápsulas utilizadas en las diferentes escenas.

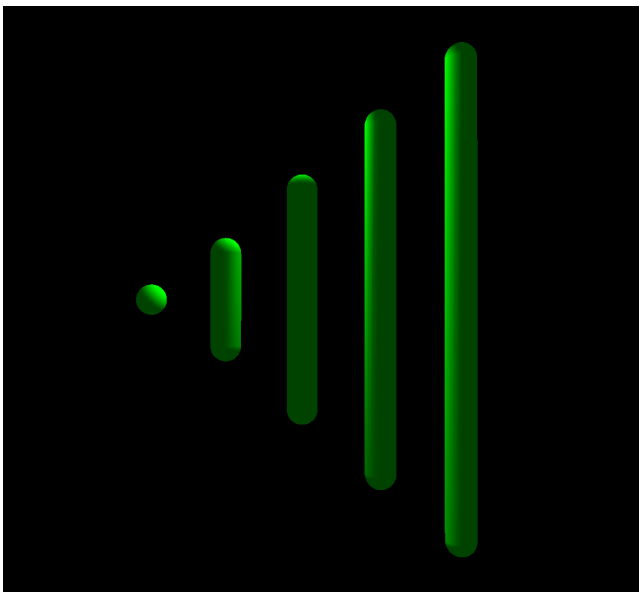


Figura 5.6: Las distintas proporciones de cápsulas utilizadas para las pruebas entre SAP y SAP + OBBs: 1:1, 4:1, 8:1, 12:1 y 16:1.

En las escenas utilizadas para esta evaluación, las cápsulas se encuentran posicionadas de igual forma que en las escenas usadas para la comparación de los algoritmos de BPCD, pero cuentan con más espacio entre ellas, lo cual fue necesario para lograr un mejor movimiento en el caso de las cápsulas más alargadas. La cantidad de objetos es constante en todas estas escenas, por lo que la grilla que forman y la caja que las contiene tienen proporciones similares a las de las propias cápsulas.

Es interesante analizar cómo se relaciona la ecuación de SAP (ecuación 5.2) con la de SAP + OBBs (ecuación 5.3) para comparar estos dos algoritmos. Dado que el resultado de cada ecuación es el tiempo total de la detección de colisiones, para que el algoritmo de SAP + OBBs sea más rápido que aplicar SAP, debe cumplirse la desigualdad 5.4. Como en ambos algoritmos la escena es la misma, el costo promedio de chequear objetos (O) es igual. Además, como el algoritmos SAP es lo primero que se ejecuta en los dos casos, su costo (C) es el mismo.

$$C + o_1 O > C + bB + o_2 O \tag{5.4}$$

Teniendo en cuenta que en ambos casos el algoritmo SAP descartará las mismas colisiones, la etapa siguiente a aplicarlo hará los mismos chequeos en ambos casos. Se puede ver entonces que la cantidad de chequeos de NPCD en el caso de usar solo SAP (o_1), es igual a la cantidad de chequeos de OBBs al usar SAP + OBBs (b). Igualando estos dos valores y cancelando en ambos lados de la desigualdad 5.4 el costo del SAP, se llega a la desigualdad 5.5.

$$o_1 O > o_1 B + o_2 O \tag{5.5}$$

Finalmente, reordenando los términos se llega a que SAP + OBBs es más eficiente que SAP si se cumple la desigualdad 5.6.

$$\frac{O}{B} > \frac{o_1}{o_1 - o_2} \tag{5.6}$$

Se calculó anteriormente que el tiempo de NPCD entre cápsulas (O) es de 546ns, y que el chequeo entre OBBs (B) se hace en 25ns. Sabiendo esto, para que la utilización de SAP + OBBs valga la pena, el lado derecho de la desigualdad debe ser menor a $546ns/25ns = 21.84$. Más adelante se

analizarán algunos resultados en base a esto.

La gráfica de la Figura 5.7 muestra una comparación entre el tiempo de detección de colisiones del algoritmo SAP y el del algoritmo SAP + OBBs, para diferentes largos de cápsulas. Dado que lo que interesa es ver cómo se comparan estos dos algoritmos, se normalizaron los valores de forma que el tiempo total de ejecución del SAP siempre valga 1.

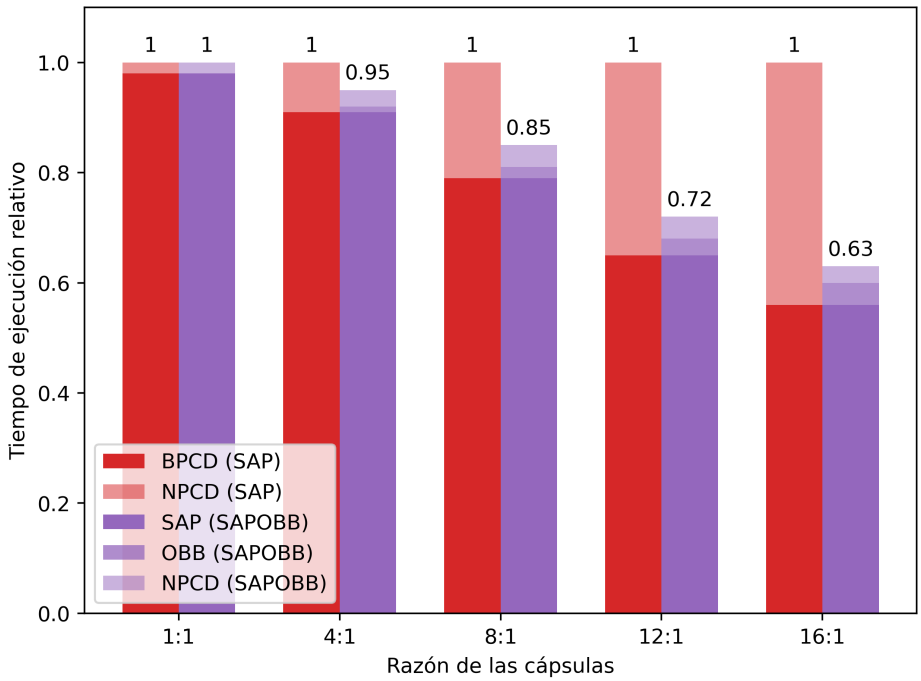


Figura 5.7: Comparación del tiempo de detección de colisiones de los algoritmos SAP y SAP + OBBs, variando el largo de los objetos de la escena.

Se puede ver que ambos algoritmos tienen un desempeño casi igual cuando la razón es 1 : 1, donde las cápsulas se degeneran a esferas. En este caso, como las AABBs aproximan muy bien a las esferas, SAP descarta una gran cantidad de colisiones y por lo tanto el tiempo que insume la fuerza bruta con OBBs es despreciable. Para todos los otros casos SAP + OBBs resulta más eficiente. Se cumple que siempre que aumenta la razón, mejora el desempeño de SAP + OBBs con respecto al de SAP, llegando a ejecutarse en alrededor de un 63 % del tiempo de que insume SAP al manejar objetos con una razón de 16 : 1.

Es interesante notar también que a medida que aumenta la razón de los objetos, aumenta el tiempo de la etapa de NPCD en el caso de SAP, y el de la etapa de OBBs en el caso de SAP + OBBs. Esto tiene sentido, ya que

al aumentar la razón, las AABBs que utiliza SAP aproximarán al objeto de forma menos precisa, por lo que SAP descartará menos colisiones y la etapa que le sigue (sea la de NPCD o la fuerza bruta con OBBs), tendrá más pares de posibles colisiones para chequear. Para entender mejor esto, en la gráfica de la Figura 5.8 se muestra la cantidad de chequeos de NPCD que se realizan para ambos algoritmos en las diferentes escenas.

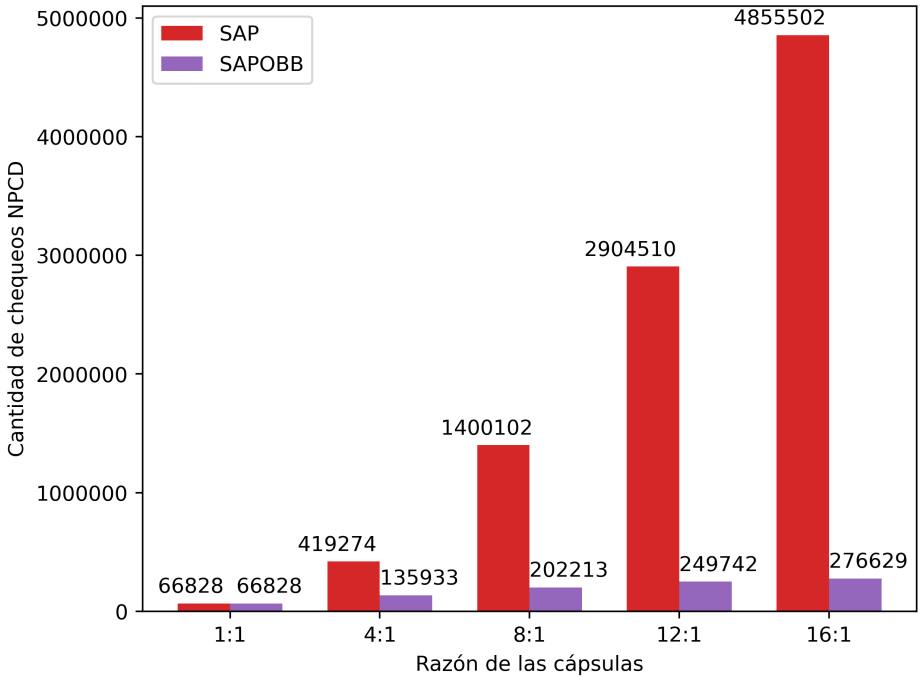


Figura 5.8: Cantidad de chequeos de NPCD para los algoritmos SAP y SAP + OBBs, variando el largo de los objetos de la escena.

Al tener una razón de 1 : 1, como las cápsulas pasan a ser esferas, las AABBs son iguales a las OBBs, por lo que las OBBs no descartan ninguna colisión extra y entonces ambos algoritmos resultan en los mismos chequeos de NPCD. A medida que la razón crece, los chequeos de NPCD realizados por SAP aumentan a una velocidad mucho mayor que para SAP + OBBs, llegando a hacer 17 veces más chequeos en el caso de las cápsulas con una razón de 16 : 1.

Analizando la desigualdad 5.6 que se presentó antes, en el caso de la razón 1 : 1 tanto la cantidad de chequeos de NPCD de SAP como la de OBBs valen 66828. Se puede ver en la desigualdad 5.7 que el resultado obtenido al sustituir (infinito o indeterminado) no es menor a 21.84, por lo que la desigualdad no se cumple. Esto tiene sentido, ya que es imposible en este

caso que SAP + OBBs sea más eficiente que SAP, porque no descarta ninguna colisión extra.

$$21.84 \not> \frac{66828}{0} = \frac{66828}{66828 - 66828} \tag{5.7}$$

Por otro lado, en el caso de la razón 4 : 1 la cantidad de chequeos NPCD de SAP es 419274, mientras que la de SAP + OBBs es 135933. En este caso, como se vio antes, el tiempo de SAP + OBBs es menor al de SAP, y la desigualdad se cumple:

$$21.84 > 1.48 \approx \frac{419274}{419274 - 135933} \tag{5.8}$$

Por último, es importante aclarar que las pruebas realizadas no son representativas de todos los escenarios posibles a los que se puede enfrentar un motor de física. Sin embargo, en base a los resultados obtenidos la combinación de SAP y fuerza bruta con OBBs parece superior a utilizar únicamente SAP, ya que en el peor de los casos los dos algoritmos tienen un desempeño muy similar, mientras que en casos favorables SAP + OBBs logra ganancias muy significativas. También se debe aclarar que las cápsulas son uno de los cuerpos cuya detección de colisiones es más simple, por lo que para otros tipos de objetos más complejos el impacto de la etapa NPCD sería mayor y las diferencias encontradas también lo serían.

5.2. Desempeño de Fingsics

Para analizar el desempeño de Fingsics, se comenzará analizando el impacto de cada una de las etapas del ciclo del motor en el tiempo de procesamiento. Dado que el algoritmo SAP + OBBs obtuvo muy buenos resultados, se lo utilizará como BPCD para el resto de la experimentación.

Las gráficas de la Figura 5.9 muestran qué proporción del tiempo de procesamiento de cada cuadro insume cada etapa del ciclo de detección de colisiones, respuesta a las colisiones, y renderizado. Se utilizó una escena con esferas y cápsulas como las descritas antes, de 10000 objetos. La primera gráfica corresponde a una ejecución en tiempo real, mientras que la segunda muestra una reproducción de los datos grabados de la misma escena. Como ya se mencionó, reproducir una escena grabada no ejecuta ninguna tarea relacionada al manejo de colisiones, sino que solamente lee las posiciones

almacenadas y actualiza los objetos en base a estas en cada cuadro.

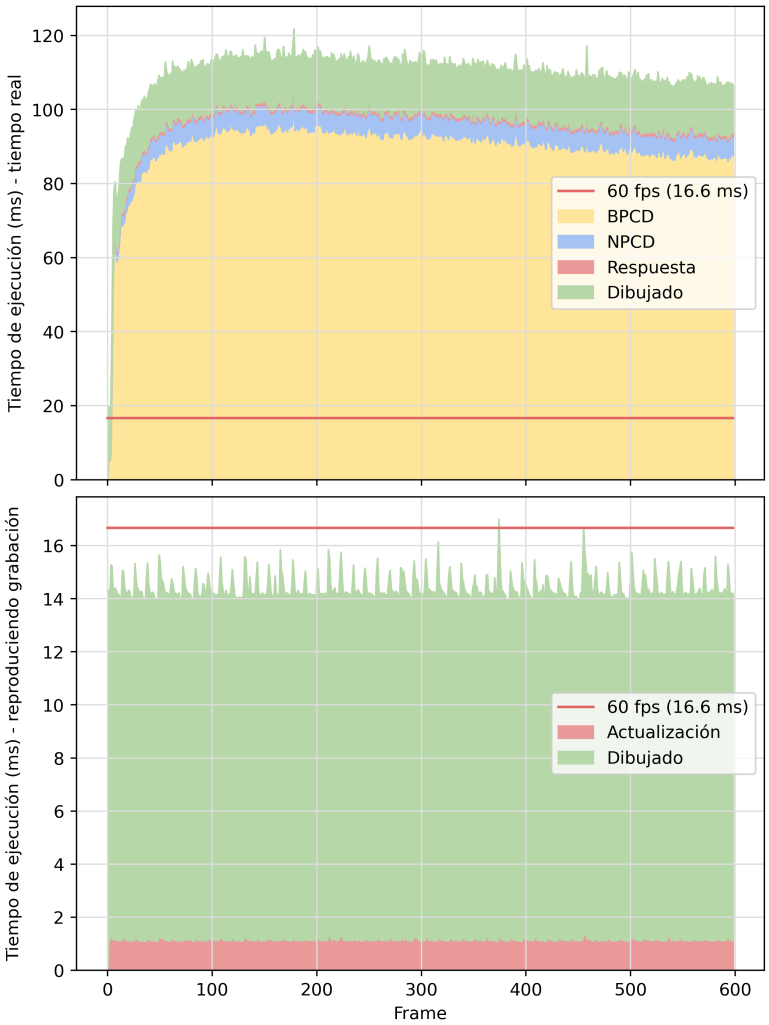


Figura 5.9: Tiempo de ejecución de cada etapa del procesamiento de los cuadros de una escena de 1000 objetos, para una ejecución en tiempo real y la reproducción de una grabación. Los valores por encima de la línea roja no pueden ser ejecutados a 60fps.

Se puede observar que el tiempo total de procesamiento de todos los cuadros para la reproducción en tiempo real excede los 16.6 milisegundos, por lo que en este caso la ejecución no pudo realizarse a 60fps. Mientras tanto, al reproducir la escena grabada, el tiempo sí es menor que este valor, y se logra mantener de forma estable 60fps.

Se puede ver en la primera gráfica que el tiempo correspondiente a la detección de las colisiones es el más alto, tomando unos 90 milisegundos de los 115 que lleva procesar cada cuadro. Dentro de la detección de colisiones, la mayor cantidad del tiempo corresponde a la ejecución de la etapa de

BPCD, y esto tiene sentido ya que como se mencionó antes, el algoritmo de SAP + OBBs es costoso pero permite descartar muchas colisiones y por lo tanto alivianar el NPCD.

Para esta cantidad de objetos el tiempo de dibujado es también relativamente alto, alrededor de 15 milisegundos. Dicho costo depende de qué tantos polígonos se utilicen para dibujar a las cápsulas y las esferas. Esto es configurable en Fingsics, a través de la propiedad *OBJECT_DEFINITION*, y en este caso se usó un valor de 8 (que significa que cada objeto curvo se divide en polígonos utilizando 8 latitudes y 8 longitudes, y entonces contiene aproximadamente 64 polígonos).

La respuesta a las colisiones es casi despreciable, lo cual tiene sentido, ya que solo tiene que recorrer el conjunto de colisiones (mientras que la detección recorre todas las posibles colisiones, y el dibujado todos los objetos).

Finalmente, en la segunda gráfica se puede ver que al reproducir una escena casi todo el tiempo de procesamiento corresponde a dibujar los objetos, y una parte mucho menor a actualizar sus posiciones.

Como segunda prueba, para estudiar cuántos objetos puede soportar Fingsics manteniendo 60fps de forma estable, se recabaron los datos mostrados en las gráficas de la Figura 5.10.

Igual que antes, la primera corresponde a una ejecución en tiempo real mientras que la segunda es la reproducción de una grabación. En la primera se puede observar que la escena con 1000 objetos es la más grande para la cual se logra renderizar en tiempo real todos sus cuadros en menos de 16.6 milisegundos. En la segunda gráfica se ve que al reproducir una grabación se pueden soportar más objetos, siendo la escena de 10000 objetos la más grande que se logra ejecutar a 60fps.

Para escenas de más objetos, se cuenta también con la posibilidad de renderizarlas en formato video. En estos casos, videos de 10 segundos a 60fps de escenas de hasta 100000 objetos pueden procesarse en menos de dos horas.

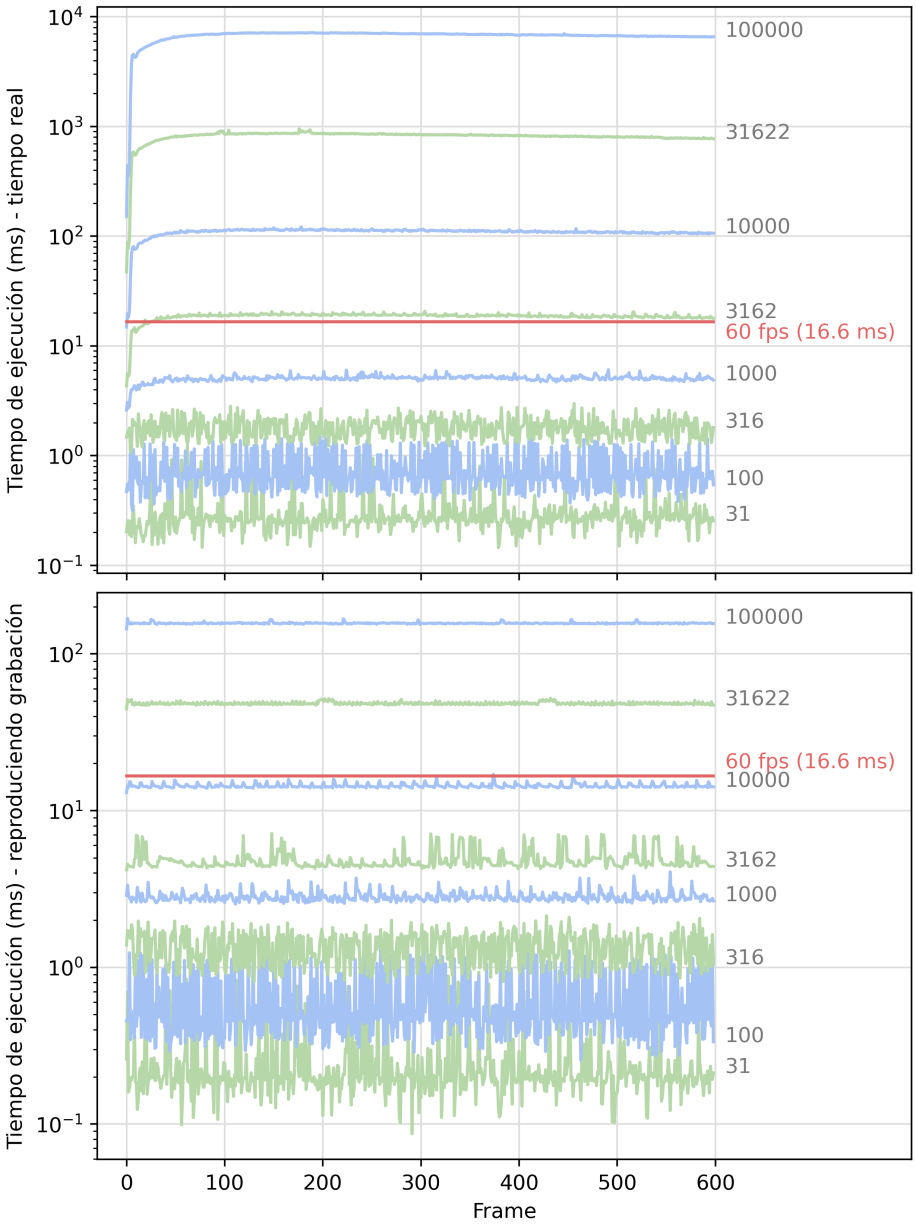


Figura 5.10: Tiempo de procesamiento de cada cuadro en escenas de diferentes cantidades de objetos, para ejecuciones en tiempo real y reproducciones de grabaciones.

5.3. Comparación con Bullet

Como se mencionó en la Sección 2.6, Bullet es un motor físico de código abierto ampliamente utilizado en videojuegos y otras aplicaciones, por lo que es un buen punto de comparación para los resultados obtenidos en Fingsics.

Si bien las interfaces que Bullet ofrece son muy distintas a las de Fingsics, modificando su demo se logró configurar una escena muy similar a las que se utilizaron en las secciones anteriores: esferas y cápsulas encerradas en una caja. Más específicamente, se replicó la escena con $10^{3.5}$ objetos, utilizando proporciones similares entre los objetos y el contenedor. También se desactivaron las fuerzas de gravedad y de rozamiento en todos los objetos de la escena y se utilizaron los mismos coeficientes de restitución que en las pruebas de las secciones anteriores. En la Figura 5.11 se muestra la escena creada.

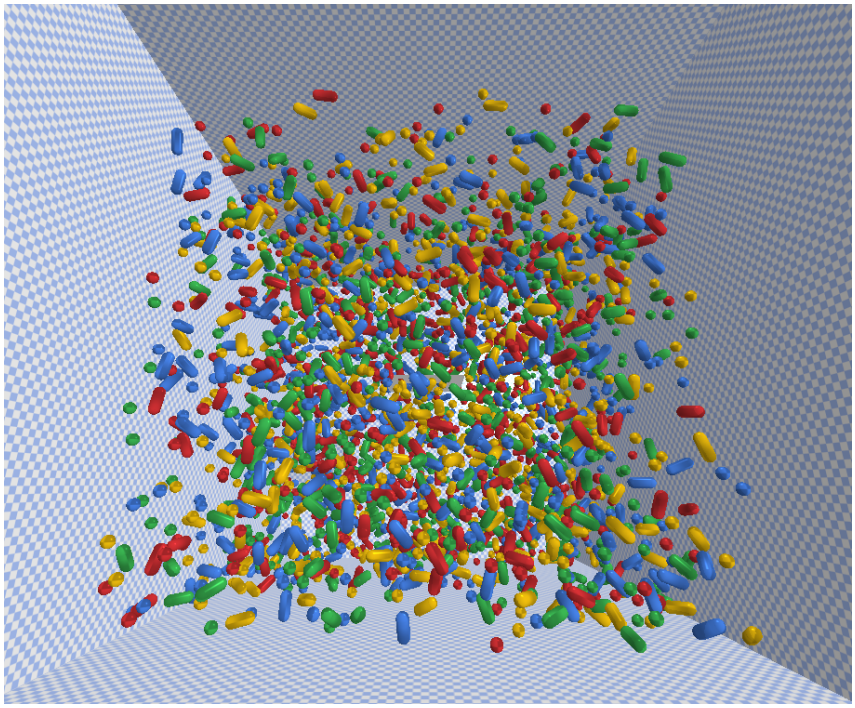


Figura 5.11: Escena usada para comparar con Bullet.

Se ejecutó esta escena en el mismo hardware que las escenas de las secciones anteriores, utilizando una opción de Bullet que permite la ejecución en un único hilo, para que la comparación sea justa. Los tiempos tiempos

de ejecución por cuadro reportados por Bullet fluctúan entre 10 y 12 milisegundos. Como se vio antes en la Figura 5.10, en el caso Fingsics los tiempos se encuentran entre 18 y 20 milisegundos para la misma escena.

Considerando la madurez y el gran porte de Bullet como proyecto, obtener tiempos de ejecución de magnitudes relativamente similares es un buen resultado. Sin embargo, hay que aclarar que la escena utilizada solo evalúa una pequeña parte de las posibilidades que Bullet ofrece en cuanto a manejo de colisiones. Además, incluso en este caso el manejo de colisiones de Bullet es superior al de Fingsics, ya que incluye funcionalidades como el manejo de contactos entre superficies en reposo.

Capítulo 6

Conclusiones y trabajo a futuro

En este último capitulo se presentan las conclusiones del proyecto y la investigación relativa al mismo, junto a algunas ideas que no fueron implementadas pero podría ser interesante desarrollar en un futuro.

6.1. Conclusiones

El enfoque de este trabajo estuvo en la implementación y estudio de diferentes técnicas para la detección de colisiones en escenas con grandes cantidades de objetos, pero también se abordaron algunas temáticas relacionadas como la respuesta a las colisiones y la renderización de una escena en tiempo real.

Se implementó Fingsics, un motor de simulación de colisiones para objetos rígidos, que permite ejecutar escenas en tiempo real así como grabarlas, tanto en formato video como en un formato propio que permite su reproducción dentro del programa. El motor cuenta con cuatro posibles algoritmos de BPCD: fuerza bruta con AABBs, fuerza bruta con OBBs, SAP y SAP + OBBs. Como objetos soporta cápsulas, esferas y tiles, y se implementaron los algoritmos de NPCD correspondientes a cada par posible de estos objetos, exceptuando el par Tile-Tile. Para la respuesta a las colisiones se implementó el enfoque basado en impulsos.

Se pudo ver que en el programa los tiempos promedio de detección de colisiones entre objetos (NPCDs) son siempre mayores o iguales a los correspondientes a volúmenes acotantes. La detección entre AABBs toma 12*ns* y la de OBBs 25*ns*, mientras que los NPCDs toman entre 25*ns* (Esfera-

Tile) y 1733*ns* (Cápsula-Tile).

Experimentando con los algoritmos de BPCD se encontró que, en comparación a no utilizar ningún algoritmo, usar un algoritmo de fuerza bruta con algún tipo de volumen acotante supone un salto considerable en el rendimiento de la detección de colisiones. Se vio que a pesar de que la fuerza bruta con OBBs permite descartar más chequeos de NPCD que la de AABBs, en las escenas utilizadas esto no fue suficiente para justificar el alto costo de detección de colisiones para este volumen acotante.

A su vez, usar un algoritmo más elaborado como SAP o SAP + OBBs mejora aún más el desempeño, también de forma considerable. Analizando más a fondo estos dos últimos algoritmos, se pudo ver que si bien tienen un desempeño similar en escenas con objetos variados, al manejar objetos alargados el algoritmo SAP + OBBs funciona bastante mejor, tomando hasta solo un 63 % del tiempo en el caso de objetos con una razón de 16 : 1.

En cuanto a la respuesta a las colisiones y al renderizado, se vio que sus tiempos son considerablemente bajos con respecto al de detección de colisiones, principalmente el de la respuesta a las colisiones.

También se analizó el rendimiento de Fingsics en términos generales, haciendo énfasis en qué cantidad máxima de objetos puede soportar a una velocidad de 60fps en hardware moderno. En el caso de la ejecución en tiempo real, utilizando el algoritmo de BPCD para el cual se obtuvieron los mejores resultados (SAP + OBBs) se soportan más de 1.000 objetos manteniendo 60fps de forma estable. Si se graba la escena y se reproduce dentro del programa (y por lo tanto no se tiene que ejecutar la detección y respuesta a las colisiones), se pueden manejar hasta 10.000 objetos a 60fps. Por otro lado, si se quiere ver escenas con más objetos, se puede también renderizar la simulación en formato video, donde 600 cuadros de escenas de hasta 100.000 objetos toman menos de dos horas en procesarse.

Finalmente, se comparó el rendimiento de Fingsics con el de Bullet, otro motor de colisiones ampliamente utilizado. Los resultados obtenidos muestran que además de ser más completo, Bullet es un motor más rápido, pero que las diferencias en este sentido son relativamente pequeñas.

6.2. Trabajo a futuro

A lo largo del proyecto surgieron ideas que serían interesantes de implementar y evaluar, así como funcionalidades que por restricciones de tiempo se decidió no desarrollar.

Sería interesante implementar grillas como algoritmo de BPCD, y evaluar cómo se compara su desempeño al de los algoritmos ya implementados, así como agregar funcionalidades que permitan ampliar la variedad de escenas que Fingsics permite simular. Esto incluye el algoritmo de NPCD para el par Tile-Tile (para poder utilizarlas como objetos no estáticos), soportar más variedad de cuerpos rígidos, o incluso agregar objetos articulados y flexibles. Otras posibles mejoras son las relacionadas al algoritmo de respuesta a las colisiones: manejar correctamente las colisiones simultáneas y los contactos de reposo, y agregar otras funcionalidades como la fuerza de rozamiento entre objetos. Estas últimas mejoras lograrían que las simulaciones sean más realistas.

Glosario

AABB - *Axis Aligned Bounding Box* es un tipo de volumen acotante que tiene forma de prisma de base rectangular y cumple que sus aristas son paralelas a los ejes de coordenadas.
API - *Application Programming Inferface* o interfaz de programación de aplicaciones, es un conjunto de definiciones y protocolos (por ejemplo funciones y procedimientos) ofrecidas por cierta biblioteca con el fin de que sean consumidas por otro software.
BPCD - *Broad Phase Collision Detection* es la primera etapa de la detección de colisiones, en la cual se busca realizar un primer filtro del conjunto de todas las colisiones posibles, descartando algunas (usualmente en base a sus volúemenes acotantes).
Bullet - Biblioteca de manejo de colisiones que ha sido utilizada en el desarrollo de videojuegos, efectos especiales y robótica, y cuyo código es público.
Cápsula - Cuerpo rígido que es similar a un cilindro pero en lugar de dos bases tiene en cada uno de sus extremos una media esfera de igual radio que el cilindro.
Coeficiente de restitución - Medida de la proporción de energía cinética que se conserva tras el choque entre dos partículas o cuerpos.
Coherencia temporal - Propiedad que, cuando se cumple, expresa que los objetos sufren modificaciones menores en sus posiciones y velocidades entre un cuadro y el siguiente.
Cuadro - O *frame* en inglés, es una imagen (o los datos para representarla) en una serie de imágenes consecutivas que componen, por ejemplo, un video. También se usa este término para referirse a los pasos en una simulación física de tiempo discreto en los cuales se aplica el manejo de colisiones.
Cuerpo rígido - Cuerpo que cumple que la distancia entre todas las partículas que lo componen es constante (no se deforma, a diferencia de una tela por ejemplo).

CUDA - *Compute Unified Device Architecture* es una plataforma de computación en paralelo desarrollada por NVIDIA cuyo objetivo es facilitar el uso de GPUs para la computación de propósito general.
FPS - *Frames per second* o *fotogramas por segundo* es la cantidad de imágenes que se muestran por cada segundo de video o simulación.
GPU - *Graphic Processing Unit* o unidad de procesamiento gráfico, es un procesador de datos cuya principal funcionalidad es el procesamiento de gráficos y las operaciones de punto flotante.
Línea base - Un mínimo o punto de referencia que se desea superar en futuras comparaciones.
Malla Poligonal - Es una colección de ejes, vértices y caras que conforman un poliedro.
MPCD - *Mid Phase Collision Detection* es la segunda etapa de la detección de colisiones. Toma como entrada la salida de la BPCD, y busca obtener información sobre qué partes de los objetos considerados podrían estar colisionando. Su salida es la entrada de la etapa de NPCD, y a veces es omitida (en cuyo caso la salida de la BPCD es la entrada de la NPCD).
MultiSAP - Algoritmo multihilo de BPCD basado en SAP.
NPCD - *Narrow Phase Collision Detection* es la tercera y última etapa de la detección de colisiones. Toma como entrada la salida de la MPCD (o la de la BPCD si no hay MPCD), y determina qué pares de objetos colisionan entre sí.
Objeto estático - Objeto al cual las fuerzas no le hacen efecto (se puede interpretar como un objeto de masa infinita).
OpenGL - API multilenguaje y multiplataforma para la renderización de gráficos 2D y 3D.
PhysX - Motor físico en tiempo real, de código abierto y desarrollado por NVIDIA.
Politopo - Generalización a cualquier dimensión de un polígono (de dos dimensiones) o de un poliedro (de tres dimensiones). Es un objeto geométrico con caras planas.
Ray tracing - Conocido como *trazado de rayos* en español, es una técnica de modelado de luz para la iluminación de imágenes generadas por computadoras.
Renderizar - Proceso de generar una imagen a partir de un modelo 2D o 3D (también llamado escena).

Sweep-And-Prune (SAP) - Algoritmo de BPCD que es eficiente gracias a que aprovecha la coherencia temporal y utiliza AABBs como volúmenes acotantes.
Tensor de inercia - Caracterización de la inercia rotacional de un cuerpo rígido. En el caso de cuerpos bastante estudiados, como esferas y cápsulas, existen fórmulas para calcularlos en base a sus dimensiones.
Tile - *Baldosa* o *azulejo* en español, es un cuerpo rígido de forma rectangular, es decir que el largo de una de sus tres dimensiones es despreciable con respecto al de las otras dos.
Volumen acotante - Es un objeto que contiene completamente a otro (o a otro conjunto de objetos). Suelen ser simples y son muy utilizados ya que realizar cálculos con ellos (por ejemplo determinar si dos de ellos colisionan entre sí) es más simple que hacerlo con los objetos complejos que contienen. En el caso de la detección de colisiones, si dos volúmenes acotantes no colisionan, se puede afirmar que los cuerpos que contienen tampoco lo hacen, pero si sí colisionan, es posible que los cuerpos contenidos también lo hagan o que no.
Wireframe - Representación gráfica de cuerpos en la cual sólo se muestran sus aristas y no sus caras.

Bibliografía

1. Caulfield B. NVIDIA Unveils GeForce RTX, World's First Real-Time Ray Tracing GPUs. https://blogs.nvidia.com/blog/2018/08/20/gamescom-rtx-turing-real-time-ray-tracing/. 2018. Último acceso: Noviembre del 2021
2. Millington I. Game Physics Engine Development: How to Build a Robust Commercial-Grade Physics Engine for Your Game. USA: CRC Press, Inc., 2007
3. Ericson C. Real-Time Collision Detection. USA: CRC Press, Inc., 2004
4. Nealen A, Müller M, Keiser R, Boxerman E y Carlson M. Physically Based Deformable Models in Computer Graphics. Comput. Graph. Forum 2006; 25:809-36. DOI: 10.1111/j.1467-8659.2006.01000.x
5. Teschner M, Kimmerle S, Heidelberger B, Zachmann G, Raghupathi L, Fuhrmann A, Cani MP, Faure F, Thalmann N, Straßer W y Volino P. Collision Detection for Deformable Objects. Comput. Graph. Forum 2005; 24:61-81. DOI: 10.1111/j.1467-8659.2005.00829.x
6. Akenine-Mller T, Haines E y Hoffman N. Real-Time Rendering. 4th. USA: A. K. Peters, Ltd., 2018
7. Weghorst H, Hooper G y Greenberg DP. Improved Computational Methods for Ray Tracing. ACM Trans. Graph. 1984; 3:52-69. DOI: 10.1145/357332.357335
8. O'Rourke J. Finding minimal enclosing boxes. International Journal of Computer & Information Sciences 1985; 14. DOI: 10.1007/BF00991005
9. Gottschalk S. Collision Queries using Oriented Bounding Boxes. Tesis doct. Department of Computer Science, University of North Carolina, 2000

10. Larsson T y Kallber L. Fast Computation of Tight-Fitting Oriented Bounding Boxes. *Game Engine Gems 2*. Ed. por A K Peters L. CRC Press, 2011 :3-19

11. Gottschalk S, Lin M y Manocha D. OBBTree: A Hierarchical Structure for Rapid Interference Detection. Computer Graphics 1997. DOI: 10.1145/237170.237244

12. Baraff D. Dynamic Simulation of Non-Penetrating Rigid Bodies. Tesis doct. Cornell University, 1992

13. Lin MC. Effcient Collision Detection for Animation and Robotics. Tesis doct. University of California, 1993

14. Terdiman P. Sweep-and-prune. http://www.codercorner.com/SAP.pdf. 2007. Último acceso: Noviembre del 2021

15. Teschner M, Heidelberger B, Müller M, Pomeranets D y Gross M. Optimized Spatial Hashing for Collision Detection of Deformable Objects. VMV'03: Proceedings of the Vision, Modeling, Visualization 2003; 3

16. Pouchol M, Ahmad A, Crespin B y Terraz O. A Hierarchical Hashing Scheme for Nearest Neighbor Search and Broad-Phase Collision Detection. J. Graphics, GPU, & Game Tools 2009; 14:45-59. DOI: 10.1080/2151237X.2009.10129281

17. Klosowski J, Held M, Mitchell J, Sowizral H y Zikan K. Efficient collision detection using bounding volume hierarchies of k-DOPs. IEEE Transactions on Visualization and Computer Graphics 1998; 4:21-36. DOI: 10.1109/2945.675649

18. Gilbert E, Johnson D y Keerthi S. A fast procedure for computing the distance between complex objects in three-dimensional space. IEEE Journal on Robotics and Automation 1988; 4:193-203. DOI: 10.1109/56.2083

19. Hasegawa S, Fujii N, Koike Y y Sato M. Real-Time Rigid Body Simulation Based on Volumetric Penalty Method. 2003 :326-32. DOI: 10.1109/HAPTIC.2003.1191304

20. Vella C. Gravitas : an extensible physics engine framework using object-oriented and design pattern-driven software architecture principles. Tesis de mtría. University of Malta, 2008

21. Mirtich B y Canny J. Impulse-Based Simulation of Rigid Bodies. I3D '95. Monterey, California, USA: Association for Computing Machinery, 1995. DOI: 10.1145/199404.199436

22. Mirtich B. Impulse-based Dynamic Simulation. 2002

23. Cottle Pang S. The Linear Complementarity Problem. Boston, MA: Academic Press, Inc., 1992

24. Baraff D. Fast Contact Force Computation for Nonpenetrating Rigid Bodies. SIGGRAPH '94. New York, NY, USA: Association for Computing Machinery, 1994 :23-34. DOI: 10.1145/192161.192168

25. Erleben K. Numerical Methods for Linear Complementarity Problems in Physics-Based Animation. Vol. 7. 2013. DOI: 10.1145/2504435.2504443

26. Stepień J. Physics-Based Animation of Articulated Rigid Body Systems for Virtual Environments. Tesis doct. 2013 :9-11

27. Tassa Y. Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX. 2015. DOI: 10.1109/ICRA.2015.7139807

28. Nvidia. PhysX 4.1 SDK Guide. https://gameworksdocs.nvidia.com/PhysX/4.1/documentation/physxguide/Manual/RigidBodyCollision.html#broad-phase-algorithms. 2008–2021. Último acceso: Noviembre del 2021

29. Terdiman P. Box pruning revisited. http://www.codercorner.com/blog/?p=1978. 2018. Último acceso: Noviembre del 2021

30. Coumans E y Bai Y. PyBullet, a Python module for physics simulation for games, robotics and machine learning. http://pybullet.org. 2016–2021. Último acceso: Noviembre del 2021

31. Linietsky J. Godot 3.0 switches to Bullet for physics. https://godotengine.org/article/godot-30-switches-bullet-3-physics. 2017. Último acceso: Noviembre del 2021

32. Closest points between two lines. Mathematics Stack Exchange. https://math.stackexchange.com/q/1993990. Último acceso: Noviembre del 2021

ANEXOS

Anexo A

Instalación y uso

Los archivos necesarios para ejecutar Fingsics en Windows x64 se pueden encontrar en la carpeta *binaries* del siguiente repositorio:

https://github.com/fingsics/Fingsics

En dicha carpeta se encuentra el archivo binario *Fingsics.exe* que permite ejecutar el programa, el archivo de configuración *config.txt*, una carpeta llamada *testing* que contiene los resultados esperados para la funcionalidad de test, y una carpeta llamada *scenes* con diversas escenas que pueden ser ejecutadas.

Es importante tener en cuenta que para que el programa funcione correctamente no se debe cambiar el nombre de ningún archivo ni carpeta, y que todas las escenas comienzan en pausa, por lo que para que la simulación comience hay que apretar la tecla *P*.

En este anexo se explica cómo usar el programa, los diferentes modos que tiene, cómo configurarlo a través del archivo *config.txt*, y cómo crear nuevas escenas.

A.1. Modos de ejecución

En Fingsics existen los siguientes cuatro modos de ejecucion:

1. DEFAULT: Es el modo principal, permite ejecutar escenas en tiempo real generando grabaciones y reportes. Las salidas de este modo se generan en una carpeta llamada *output*. Dentro de esta carpeta, para cada ejecución que tenga una salida se crea una carpeta con el nombre

de la escena y la fecha y hora de ejecución, en la cual se puede encontrar el reporte y los archivos de grabación resultantes de la ejecución (si en la configuración se indicó que se generen dichos archivos).

2. REPLAY: Permite reproducir una escena grabada.
3. BENCHMARK: Permite ejecutar una escena varias veces sin renderizarla, generando reportes con información sobre las ejecuciones. Las salidas de este modo se generan en la carpeta *benchmarks*.
4. TEST: Se implementó con fines de desarrollo, genera reportes de varias escenas que luego pueden ser comparados con los resultados esperados. Las salidas de este modo se generan en la carpeta *testing*.

A.2. Archivo de configuración

Mediante este archivo se puede modificar el funcionamiento de Fingsics. En el archivo que se encuentra en el repositorio antes nombrado se pueden ver ejemplos de todas las opciones, que son las siguientes:

- **SCENE_FILE_NAME:** Aquí se debe indicar el nombre del archivo XML de la escena, sin su extensión. Dicho archivo debe encontrarse dentro de la carpeta *scenes*.
- **FPS_CAP:** Cantidad de aplicaciones de la física (detección y respuesta a las colisiones) por segundo de simulación. Si el hardware permite que se puedan ejecutar más iteraciones que estas por segundo, el proceso de Fingsics se duerme de forma que se respete este máximo.
- **FULLSCREEN:** Booleano que indica si el programa debe ejecutarse en pantalla completa o no.
- **WINDOWED_WIDTH:** Resolución horizontal para la ejecución en modo ventana.
- **WINDOWED_HEIGHT:** Resolución vertical para la ejecución en modo ventana.
- **LOG:** Booleano que indica si se debe generar como salida un archivo CSV con datos sobre la ejecución.
- **RECORD_SCENE:** Booleano que indica si se debe grabar la escena en el formato utilizado por Fingsics.
- **RECORD_VIDEO:** Booleano que indica si se debe grabar la escena en formato video.

- **DRAW_AXES:** Booleano que indica si se deben dibujar los ejes de coordenadas en la simulación.
- **OBJECT_DEFINITION:** Entero que permite modificar con cuánta precisión se dibujan los objetos. Un mayor valor hace que los objetos se dibujen mejor pero el desempeño decaiga. Se recomienda un valor entre 5 y 20.
- **SHOW_FPS:** Booleano que indica si se deben dibujar los FPS actuales en la simulación.
- **BROAD_PHASE:** Permite indicar el algoritmo de BPCD a utilizar. Las opciones son *NONE*, *AABB*, *OBB*, *SAP* y *SAPOBB*.
- **RUN_MODE:** Permite indicar el modo de ejecución. Las opciones son *DEFAULT*, *TEST*, *BENCHMARK* y *REPLAY*.
- **NUM_RUNS_FOR_BENCHMARK:** Cantidad de veces que se ejecutan las escenas cuando se utiliza el modo *BENCHMARK*.
- **SCENE_REPLAY_FOLDER_NAME:** Nombre de la carpeta autogenerada dentro de la carpeta *output* que contiene la grabación de la escena que se desea reproducir. Se debe indicar para utilizar el modo REPLAY.
- **STOP_AT_FRAME:** Número de frame en el cual la simulación termina. Si se ingresa -1, la simulación continuará hasta que se finalice manualmente.

A.3. Controles

Los modos de ejecución *TEST* y *BENCHMARK*, como no dibujan la escena, no tienen ninguna interfaz ni controles (simplemente se abre una ventana en negro mientras ejecutan).

El modo *DEFAULT* cuenta con los siguientes controles:

- **Modo de la cámara:** Con la tecla *C* se puede cambiar el modo de la cámara entre el modo libre y el modo centrado. En el modo libre la cámara se puede mover y apuntar hacia cualquier lado, mientras que en el modo centrado la cámara siempre apunta al origen de coordenadas.
- **Cámara centrada:** Utilizando las teclas *S* y *W* se puede modificar la distancia de la cámara al origen, y apretando el segundo click del mouse y moviéndolo se puede rotar la cámara con respecto al origen.

- **Cámara libre:** Utilizando las teclas *W*, *A*, *S*, *D*, *LCTRL* y *SHIFT* se puede mover la cámara, y apretando el segundo click del mouse y moviéndolo cambiar hacia donde apunta.
- **Dibujado de volúmenes acotantes:** Con la teclas *I* y *O* se puede habilitar/deshabilitar el dibujado de las AABBs y las OBBs, respectivamente.
- **Controlar la simulación:** Con la tecla *P* se puede pausar y reanudar la simulación (todas las simulaciones arrancan en pausa), y con la tecla *M* activar la cámara lenta. Para finalizar la ejecución se puede cerrar la ventana o presionar la tecla *Q* o *ESC*.

Además, en el modo *REPLAY* se puede adelantar y atrasar el tiempo en la simulación en diferentes intervalos, utilizando las flechas del teclado, el punto y la coma.

A.4. Escenas

El nombre de la escena a ejecutar se indica en el archivo de configuración. Si se desea implementar una nueva escena, se debe crear un nuevo archivo XML dentro de la carpeta *scenes* siguiendo el mismo formato de las escenas existentes. El Listado A.1 muestra como ejemplo una escena simple con un objeto de cada tipo.

Listado A.1 Ejemplo de escena en formato XML

```
<?xml version="1.0" encoding="utf-8"?>
<config>
    <camera rad="171" pitch="-50" yaw="58"/>
    <objects>
        <tile pos="0,0,5" length="3" width="3" static="true" />
        <sphere pos="0,5,0" radius="1.5" />
        <capsule pos="5,0,0" radius="0.5" length="4" />
    </objects>
</config>
```

La configuración inicial de la cámara es opcional, y en caso de definirse apunta al centro de coordenadas con el radio (*rad*), ángulo vertical (*pitch*) y ángulo horizontal (*yaw*) indicados. Para las esferas se debe definir su radio (*radius*), para las cápsulas su radio (*radius*) y largo (*length*), y para las tiles su largo (*length*) y ancho (*width*).

Además, para todos los objetos se pueden definir las siguientes propiedades:

- **pos:** Posición inicial del objeto, dada por un vector de tres dimensiones.
- **vel:** Velocidad inicial del objeto, dada por un vector de tres dimensiones.
- **acceleration:** Aceleración del objeto, dada por un vector de tres dimensiones.
- **ang:** Ángulo inicial del objeto, dado por un vector de tres dimensiones, donde cada coordenada representa la rotación inicial con respecto al eje correspondiente a esa coordenada.
- **angVel:** Velocidad angular inicial del objeto, dado por un vector de tres dimensiones, donde cada coordenada representa la velocidad angular inicial con respecto al eje correspondiente a esa coordenada.
- **mass:** Masa del objeto, dada por un número.
- **elasticityCoef:** Coeficiente de elasticidad del objeto, dado por un número.
- **static:** Booleano que indica si el objeto es estático o no.
- **draw:** Booleano que indica si el objeto debe dibujarse o no.
- **color:** Color del objeto, dado por un vector de tres dimensiones en formato RGB.

Anexo B

Pseudocódigos de algoritmos

En este anexo se presentan los pseudocódigos del algoritmo *Sweep-And-Prune* y los algoritmos de *Narrow Phase Collision Detection* implementados.

B.1. SAP

El listado B.1 muestra los pasos realizados en una iteración del algoritmo SAP. La función `obtenerColisiones` es la función principal de la iteración, y recorre los objetos actualizando cada valor de sus AABBs. También hay que aclarar que las estructuras del algoritmo deben estar inicializadas correctamente. En el caso de los arreglos o listas con los valores de los puntos de las AABBs esto implica estar ordenados, y en el caso de la colección de colisiones significa que debe contener todas y únicamente las colisiones de la iteración anterior. Además, dependiendo de decisiones sobre las estructuras utilizadas puede ser necesario realizar algún paso extra que no está reflejado en el pseudocódigo. Por ejemplo, si se utilizan arreglos para los ejes es necesario mantener actualizadas las referencias en las AABBs a sus valores en los ejes mientras se los intercambia con otros.

Listado B.1 Pseudocódigo de una iteración de SAP

```
1
2  puntosEjeX = [2 * objetos.size()]
3  puntosEjeY = [2 * objetos.size()]
4  puntosEjeZ = [2 * objetos.size()]
5  coleccionColisiones = {}
6
7  // Una vez modificadas las posiciones de los objetos, devuelve las colisiones de sus AABBs
8  function obtenerColisiones(objetos):
9      for objeto in objetos
10         for puntoAABB in objeto.aabb
11             puntoAABB.valor = objeto.nuevoValorParaPunto(puntoAABB)
12             actualizarPuntoAABB(puntoAABB.eje, puntoAABB)
13         end for
14     end for
15     return coleccionColisiones
16 end obtenerColisiones
17
18 // Encontrar el nuevo indice para el punto y moverlo a ese lugar
19 function actualizarPuntoAABB(eje, puntoAABB):
20     indice = puntoAABB.indice
21
22     // Mover a la derecha
23     while puntoAABB > eje[indice + 1]
24         manejarIntercambio(eje[indice + 1], puntoAABB)
25         intercambiar(puntoAABB, eje[indice + 1])
26         indice = indice + 1
27     end while
28
29     // Mover a la izquierda
30     while puntoAABB < eje[indice - 1]
31         manejarIntercambio(puntoAABB, eje[indice - 1])
32         intercambiar(puntoAABB, eje[indice - 1])
33         indice = indice - 1
34     end while
35 end actualizarPuntoAABB
36
37 // Verificar si al mover un punto se da una nueva colision o deja de haber una colision
38 function manejarIntercambio(nuevoMenor, nuevoMayor):
39     if nuevoMenor.esMax and nuevoMayor.esMin
40         coleccionColisiones.remover(<nuevoMenor.objeto.id, nuevoMayor.objeto.id>)
41     else if nuevoMenor.esMin and nuevoMayor.esMax
42             and intersecanEnTodosLosEjes(nuevoMenor.objeto, nuevoMayor.objeto)
43         coleccionColisiones.agregar(<nuevoMenor.objeto.id, nuevoMayor.objeto.id>)
44     end if
45 end manejarIntercambio
```

B.2. Algoritmos de NPCD

A continuación se muestran los pseudocódigos de los algoritmos de *Narrow Phase Collision Detection* implementados, a excepción del de Cápsula-Tile debido a su complejidad. No se incluye la parte de determinación de las colisiones, es decir que las funciones solo detectan si hay una colisión o no, y no hacen los cálculos para obtener información de la colisión.

Esfera-Esfera

Listado B.2 Algoritmo de NPCD para las primitivas Esfera-Esfera

```
function esferaEsfera(esfera1, esfera2):
  return distancia(esfera1.centro, esfera2.centro) < esfera1.radio + esfera2.radio
end esferaEsfera
```

Esfera-Cápsula

Listado B.3 Algoritmo de NPCD para las primitivas Esfera-Cápsula

```
function esferaCilindro(esfera, cilindro):
  proyeccion = proyectar(esfera.centro, capsula.eje)
  estaAdentro = distancia(proyeccion, capsula.centro) < capsula.largo / 2
  return estaAdentro and distancia(proyeccion, esfera.centro) < esfera.radio + capsula.radio
end esferaCilindro

function esferaCapsula(esfera, capsula):
  if esferaCilindro(esfera, capsula.cilindro)
    return true
  end if

  if esferaEsfera(esfera, capsula.esfera1)
    return true
  end if

  if esferaEsfera(esfera, capsula.esfera2)
    return true
  end if

  return false
end esferaCapsula
```

Cápsula-Cápsula

Listado B.4 Algoritmo de NPCD para las primitivas Cápsula-Cápsula

```
1  function capsulaCapsula(capsula1, capsula2):
2    puntoCapsula1, puntoCapsula2 = puntosMasCercanos(capsula1.eje, capsula2.eje)
3    if distancia(puntoCapsula1, puntoCapsula2) > capsula1.radio + capsula2.radio
4      return false
5    end if
6
7    punto1EstaAdentroDelCilindro1 = distancia(puntoCapsula1, capsula1.centro) < capsula1.largo / 2
8    punto2EstaAdentroDelCilindro2 = distancia(puntoCapsula2, capsula2.centro) < capsula2.largo / 2
9
10   if punto1EstaAdentroDelCilindro1 and punto2EstaAdentroDelCilindro2
11     return true
12   end if
13
14   esfera1 = obtenerEsferaDelLadoDelPunto(capsula1, puntoCapsula1)
15   esfera2 = obtenerEsferaDelLadoDelPunto(capsula2, puntoCapsula2)
16
17   if esferaCilindro(esfera1, capsula2.cilindro)
18     return true
19   end if
20
21   if esferaCilindro(esfera2, capsula1.cilindro)
22     return true
23   end if
24
25   if esferaEsfera(esfera1, esfera2)
26     return true
27   end if
28
29   return false
30 end capsulaCapsula
```

Esfera-Tile

Listado B.5 Algoritmo de NPCD para las primitivas Esfera-Tile

```
1  function esferaTile(esfera, tile):
2    distancia = distancia(esfera.centro, tile.plano)
3    if distancia < esfera.radio
4       return false
5    end if
6
7    proyeccion = proyectar(esfera.centro, tile.plano)
8    cuadrante = obtenerCuadrante(proyeccion, tile)
9
10   if cuadrante == 1
11     return true
12   end if
13   elseif cuadrante in [2,4,6,8]
14     lineaTile = obtenerLinea(cuadrante)
15     return distancia(esfera.centro, lineaTile) < esfera.radio
16   end elseif
17   else
18     esquinaTile = obtenerEsquina(cuadrante)
19     return distancia(esfera.centro, esquinaTile) < esfera.radio
20   end else
21 end esferaTile
```