Proyecto de Grado

Ingeniería en Computación

Trifulca

Desarrollo de un videojuego multijugador de acción

Máximo Martínez

Felipe Otamendi

Febrero 2009

Tutores

Eduardo Fernández

Tomás Laurenzo

Centro de Cálculo - Instituto de Computación

Facultad de Ingeniería - Universidad de la República (UdelaR)

Montevideo - Uruguay

Resumen

El presente informe trata sobre la investigación y desarrollo de un videojuego multijugador de acción llamado Trifulca. El ser multijugador posibilita que varios jugadores puedan participar al mismo tiempo desde distintas computadoras. Por juego de acción se entiende a un juego en el que la interacción ocurre en tiempo real y donde pueden ocurrir múltiples eventos por segundo. La aplicación desarrollada utiliza Internet y las redes locales como medio de comunicación para actualizar el estado del juego en todas las computadoras participantes.

Los principales problemas tratados fueron resolver la comunicación entre los participantes en red contemplando restricciones de tiempo, lograr una presentación visual atractiva haciendo uso de hardware dedicado de última generación, y diseñar la interacción y simulación de los objetos del mundo del juego.

El juego fue implementado en *C++*, usando varias bibliotecas de soporte como *Bullet Physics* para simulaciones físicas, *OpenGL* para acceder al hardware de gráficos y *Lua* para scripting.

El resultado final consistió en un prototipo avanzado del juego, el cual puede ser extendido y mejorado mediante scripting y creando nuevo contenido artístico en formatos estándares.

Índice

INFORME EJECUTIVO **7**

1 Introducción **7**
1.1 Objetivos 8
1.2 Organización del documento 9

2 Estado del arte **11**
2.1 Videojuegos 11
2.2 Gráficos en tiempo real 13
2.2.1 Historia del hardware 13
2.2.2 APIs de gráficos 15
2.2.3 Técnicas 16
Iluminación. 16
Sombras. 17
Animación. 18
2.3 Soporte de red en juegos multijugador 19
2.3.1 Arquitecturas 19
2.3.2 Técnicas 20
Compresión delta. 21
Tablas de strings. 21
Predicción del cliente. 21
Compensación de lag 21
Extrapolación. 22
2.3.3 Protocolos de transporte 23
2.4 Física y fracturas 24
2.4.1 Conceptos 24
2.4.2 Simuladores de física 25
2.4.3 Fracturas 25
2.5 Scripting 26

3 Diseño de Trifulca **29**
3.1 Mecánica del juego 29
3.2 El mundo 29
3.3 Bucle principal 30
3.4 *Pipeline* de Gráficos 31
3.5 Red 33
3.5.1 Capa de plataforma 33
3.5.2 Capa de transporte 33
3.5.3 Capa de sincronización 34
3.5.4 Capa de juego 34
3.6 Lógica 34
3.7 Física 35
3.8 Cámara 36

3.9 Datos 38
3.10 Entrada de usuario 38

4 Implementación.. 39
4.1 Gráficos 39
4.1.1 Geometría 39
4.1.2 Animaciones..40
4.1.3 Materiales y efectos..41
4.1.4 Interpolación..42
4.1.5 Formatos de datos 42
4.2 Interfaz gráfica de usuario..42
4.3 Cámara 42
4.4 Lógica...43
4.4.1 Luchadores...44
4.4.2 Mapa 45
4.5 Red 45
4.5.1 Capa de plataforma...45
4.5.2 Capa de transporte 46
Conexión..46
Mensaje. 47
4.5.3 Capa de sincronización...47
Sincronización del juego. 47
Actualización de estados. 48
Comandos...49
Jugadores..50
4.5.4 Capa de juego 50
Creación e identificación de entidades. 51
Mapas. 51
Luchadores. 51

5 Pruebas.. 53
5.1 Pruebas de jugabilidad 53
5.2 Evaluación de consumo de ancho de banda 53
5.2.1 Subida del servidor variando jugadores..53
5.2.2 Subida de un cliente variando jugadores..54
5.2.3 Subida de un servidor variando clientes 54
5.2.4 Técnicas..55
Compresión delta. 55
Tabla de strings...55
5.2.5 Conclusión...56

6 Conclusiones y trabajo a futuro 57

7 Glosario 59

8 Bibliografía 61

APÉNDICE A – DETALLES DE IMPLEMENTACIÓN DEL SUBSISTEMA DE RED ... **67**

1 Capa de Transporte ... **67**
1.1 Números de secuencia ... 67
1.2 Conexión ... 69
1.2.1 Tipos de paquetes ... 69
1.2.2 Establecimiento de conexión ... 71
1.2.3 Envío de datos ... 74
1.2.4 Finalización de la conexión ... 75
1.2.5 Sincronización ... 76
1.3 Mensajes ... 78

2 Capa de Sincronización ... **80**
2.1 Estructura de estados ... 80
2.2 Técnicas ... 81
2.2.1 Tablas de strings ... 81
2.2.2 Compresión delta ... 82
2.3 Estimación de diferencia entre paquetes ... 82

Informe ejecutivo

1 Introducción

En la actualidad los videojuegos son uno de los principales medios de entretenimiento junto al cine, la televisión y la música. Esto se puede notar en la recaudación de estas industrias. En el 2007 los videojuegos recaudaron 9.5 millardos de dólares en Estados Unidos, mientras que la industria del cine y la música recaudaron 9.6 millardos y 10 millardos respectivamente(1). El crecimiento de la industria de los videojuegos fue mucho mayor al resto, como se puede ver en la Figura 1.

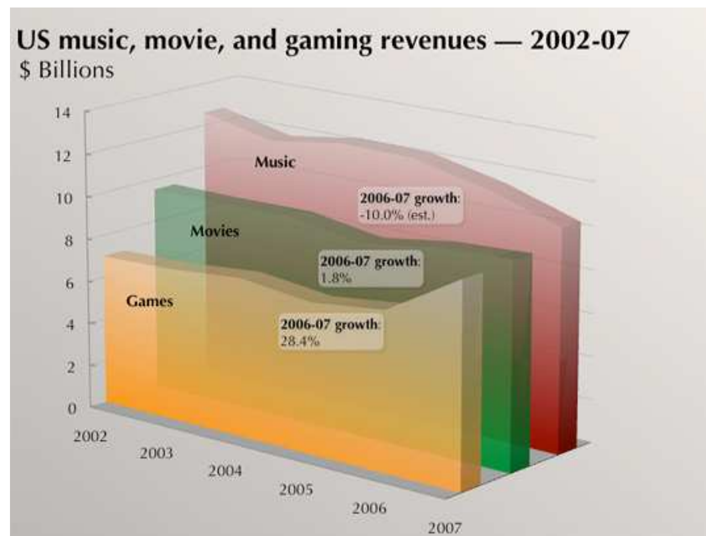

Figura 1 Crecimiento de las industrias de videojuegos, cine y música(1)

En la industria de videojuegos actual existen grandes equipos de desarrollo que crean juegos con enormes valores de producción. Estos equipos impulsan la tecnología tanto en software como en hardware. En software se han visto importantes avances en simulaciones físicas, inteligencia artificial y gráficos en tiempo real. Por el lado del hardware la industria ha impulsado el desarrollo de procesadores dedicados como el de las tarjetas gráficas y los procesadores convencionales de escritorio.

Por otro lado han aparecido pequeños desarrolladores independientes que, con valores de producción mucho menores pero gran creatividad, se han hecho un lugar en la industria. En Uruguay existen varias empresas dedicadas a desarrollar juegos casuales. Estos son juegos que son fáciles de empezar a jugar y que se pueden jugar en sesiones cortas. Los juegos casuales son muy usados para publicitar compañías y productos.

Desarrollar un videojuego puede ser una actividad motivante debido a su desafío técnico y a los resultados que se van obteniendo a lo largo del proceso. La realización en sí involucra arte y programación. Dentro del arte tenemos diseño de gráficos, sonidos, música y de la jugabilidad misma. En programación se abarcan áreas como gráficos en tiempo real, simulación física, inteligencia artificial y

comunicación en red, entre otras. Todas estas áreas pueden converger de forma balanceada en un producto de gran performance, que lleva al límite los aspectos tecnológicos.

Nuestro proyecto se concentró en las áreas de gráficos en tiempo real y comunicación en red. Pero para lograr un producto completo también comprendió lógica, jugabilidad, sonido y física. Para estas últimas áreas se utilizaron varias bibliotecas de apoyo, mientras que para las dos áreas principales se trató de implementar más a bajo nivel para entender sus problemáticas.

El resultado final consistió en un prototipo avanzado del juego, el cual puede ser extendido y mejorado mediante scripting y creando nuevo contenido artístico en formatos estándares. No se llegó a alcanzar la refinación necesaria para un producto liberable.

1.1 Objetivos

El objetivo principal del proyecto de grado es hacer un juego de peleas para PCs en un escenario bidimensional destructible con gráficos tridimensionales, inspirado principalmente en el juego *Super Smash Brothers*(2)(Figura 2) de *Nintendo*. La mecánica del juego es simple: varios jugadores se atacan con el objetivo de lanzar a los otros fuera del escenario. A medida que los jugadores reciben daño, son empujados más lejos por los golpes de otros jugadores, haciendo más fácil que caigan fuera del escenario. El juego es exclusivamente multijugador, soportando pantalla partida y juego en red simultáneamente.

Figura 2 ***Super Smash Bros* por *HAL Laboratory* y *Nintendo*(2)**

Objetivos particulares a cumplir:

- Que sea visualmente atractivo utilizando hardware dedicado de última generación, pero incluyendo un modo básico para hardware antiguo.
- Mantener la jugabilidad frente a conexiones de red con alto tiempo de respuesta.
- Extensibilidad en los luchadores y escenarios a través de formatos estandarizados, scripting y herramientas especializadas.
- Simulación física aplicada a los objetos del mundo.
- Escenario del juego modificable (destruible) por los ataques de los jugadores.
- Que sea divertido.

1.2 Organización del documento

En el siguiente informe ejecutivo se describe la investigación realizada y la solución creada para el proyecto. La sección 2 habla sobre el estado del arte de los problemas que fueron atacados por el proyecto. La sección 3 describe el diseño y arquitectura a alto nivel de todo el sistema. La sección 4 trata sobre la implementación de la solución. En la sección 5 se relatan las pruebas realizadas. Finalmente, en la sección 6 se discuten las conclusiones y posibles mejoras de la solución. En apéndice se anexa información sobre la forma de implementación de la red.

2 Estado del arte

2.1 Videojuegos

Un juego es un ejercicio recreativo sujeto a reglas, en el cual se gana o se pierde(3). Su propósito principal es entretener, pero puede tener finalidades como sensibilizar en diversas temáticas, educar, ser estrategia de mercadotecnia, entre otras(4). Componentes claves de los juegos son los objetivos, retos e interactividad (5). Un videojuego es un juego que involucra interacción con una interfaz de usuario para generar respuestas a través de un dispositivo de pantalla (6).

Los videojuegos se pueden clasificar como en tiempo real, o por turnos. Los juegos en tiempo real son aquellos en los que el tiempo de la simulación del juego avanza independientemente de la interacción con el jugador. En cambio los juegos por turnos esperan a que el jugador realice una acción para avanzar la simulación y los jugadores se turnan para realizar sus acciones. En Trifulca, al ser un juego en tiempo real, todos los jugadores pueden interactuar con el mundo en el mismo momento y el tiempo transcurre en la simulación del mundo independientemente de los jugadores. La simulación de un juego se puede avanzar en intervalos constantes o en los intervalos mínimos permitidos por la computadora donde esté corriendo. Avanzar a intervalos constantes resulta en simulaciones deterministas, cualquier computadora va a tener los mismos resultados después de un tiempo de ejecución. En cambio, si se ejecuta en los intervalos mínimos permitidos, la simulación puede ser más precisa en computadoras más veloces. Cada avance de la simulación se denomina ***frame* lógico**. Al mismo tiempo que se ejecuta la simulación se debe actualizar la imagen en pantalla, esto se puede hacer sincronizado con los *frames* lógicos o independiente de estos. Al evento de redibujado lo llamamos ***frame* de *render***.

La simulación del juego está compuesta por un conjunto de objetos o entidades que contienen información de cómo se representan, comportan e interactúan. Objetos posibles pueden ser árboles, monstruos, niveles y puertas. Para modelar estos objetos en un juego se utilizan principalmente dos técnicas: **jerarquía de clases** y **componentes** (7). La jerarquía de clases tienen como raíz la clase objeto del juego y cada objeto hereda de otro objeto para especializar su funcionalidad (Figura 3).

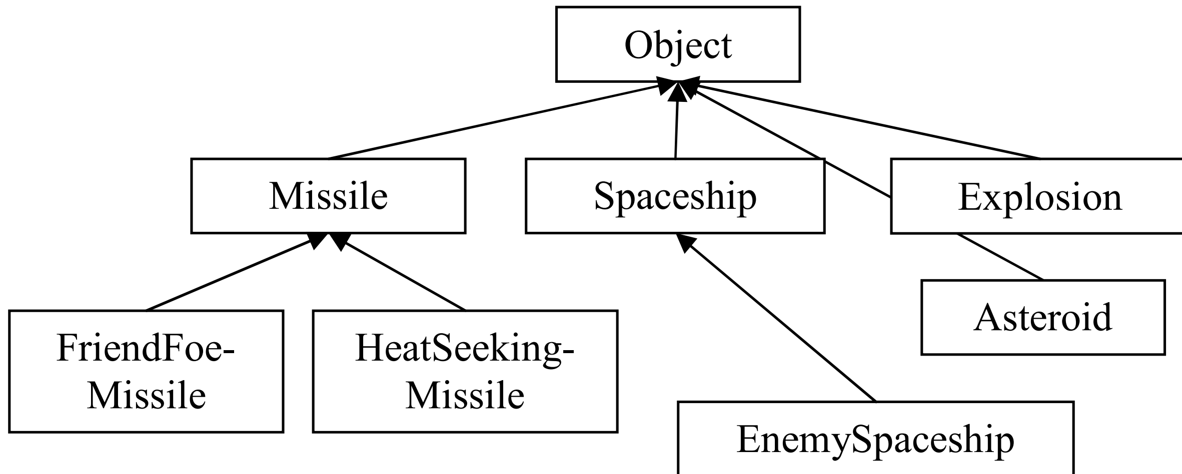

Figura 3   Jerarquía de clases clásica (7).

La jerarquía de clases resulta poco flexible. Si se necesita un objeto que incluya comportamiento de varios objetos, no es posible reutilizar código. Si se agregan interfaces o herencia múltiple se atenúan estos problemas (Figura 4).

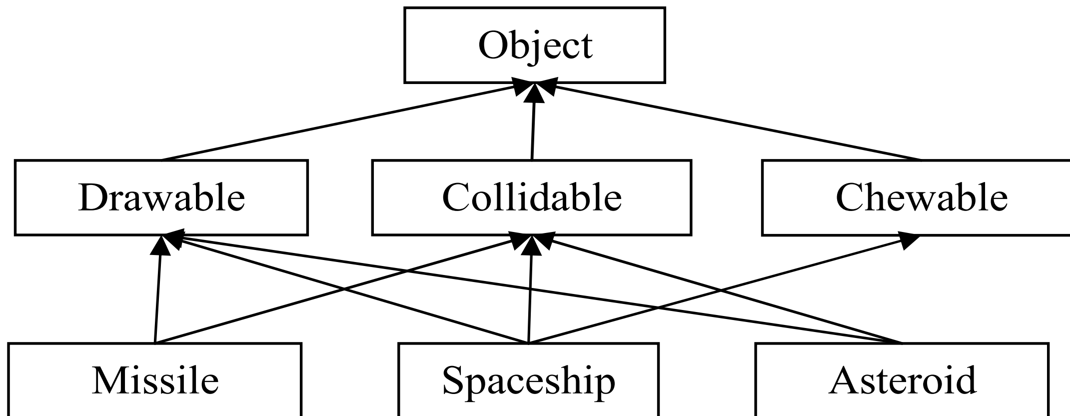

Figura 4   Jerarquía de clases con interfaces(7).

De todas formas las jerarquías de clases resultan inflexibles. Existen muchas combinaciones posibles de objetos, y los diseñadores del juego pueden requerir cambios muchas veces durante el desarrollo. El modelado de objetos de juego por componentes intenta solucionar este problema armando los objetos de juego con componentes, donde cada componente es una pieza auto contenida de lógica (Figura 5). Los componentes que forman un objeto se pueden especificar como datos del juego para permitir rápida prototipación. Para profundizar sobre estos conceptos ver (7).

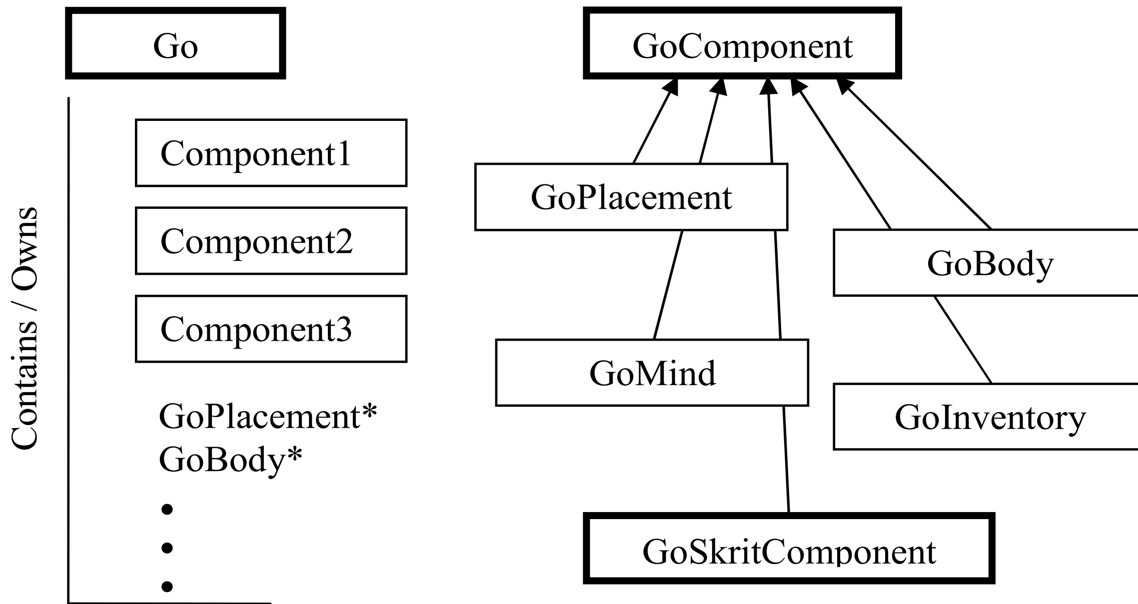

Figura 5 Modelado por componentes(7).

2.2 Gráficos en tiempo real

El avance de los gráficos en tiempo real ha estado ligado al avance del *hardware* dedicado para gráficos. Para hablar del estado actual de esta área primero se discutirá un poco la historia del *hardware* dedicado, luego los *APIs* utilizados para acceder al *hardware* y finalmente se hablará sobre las técnicas que se implementan usando el *hardware* actual. El estado del arte se enfoca en juegos y *hardware* para PC dado que es la plataforma para la que está desarrollado Trifulca.

2.2.1 Historia del hardware

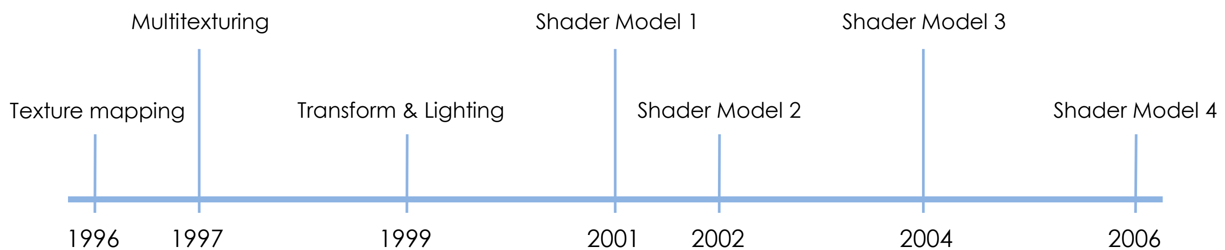

Figura 6 Avances en el *hardware* dedicado de gráficos

Esta sección se basa en la recopilación de las historias de los principales fabricantes de *hardware* gráfico, dado que no se encontró un material bibliográfico que unifique las diferentes visiones históricas.

La primera tarjeta de gráficos dirigida a consumidores apareció en 1996, ésta era la *3dfx Voodoo Graphics*(8). Su única funcionalidad era mapear texturas a polígonos con corrección de perspectiva y filtro bilineal. Esto representó un gran avance en calidad visual (Figura 7), ya que los procesadores de ese momento no eran capaces de hacer los cálculos necesarios para un filtro bilineal en tiempo real.

Figura 7 *Quake* de *id Software*, a la izquierda renderizado por software, a la derecha usando *3dfx Voodoo*(8)

El siguiente avance en hardware fue la inclusión de *Multitexturing* en la tarjeta *Ati Rage Pro*(9). *Multitexturing* permite aplicar varias texturas a un mismo polígono en una sola pasada, pudiendo elegirse varias formas de mezclar las texturas. Una técnica importante que usa *multitexturing* en su implementación es *Lightmapping*. Para esta técnica se pre calcula la iluminación y se guarda en texturas llamadas *Lightmaps*. Luego, en tiempo de ejecución, se aplica la textura original multiplicada por el *Lightmap* (Figura 8). Lo interesante de esta técnica es que la iluminación se pre calcula, facilitando el uso de técnicas costosas (no realizables en tiempo real) para lograr la iluminación. La desventaja principal radica en que el pre cálculo impide modificar la posición y direccionamiento de las luces, y no es aplicable a objetos móviles.

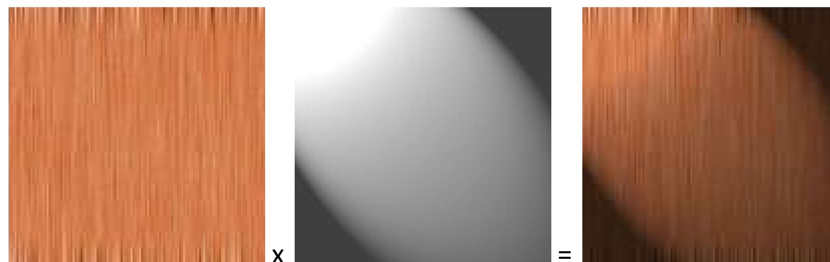

Figura 8 Una textura, multiplicada por un *Lightmap* da la imagen final iluminada (10)

Luego vino la *Nvidia GeForce 256*(11), que realizaba todo el pipeline de transformación e iluminación de vértices por hardware. Esta funcionalidad permitió aumentar la cantidad de polígonos dibujados, aumentando así el nivel de detalle en las imágenes.

Las siguientes generaciones de hardware introdujeron la posibilidad de programar partes del *pipeline* de gráficos. La primera tarjeta gráfica para consumidores programable fue la *Nvidia GeForce 3*. Si bien originalmente la programabilidad de las tarjetas era muy limitada, este avance posibilitó la implementación de nuevas técnicas y la optimización de técnicas ya usadas. En años posteriores fueron apareciendo tarjetas con mayor capacidad de programación. Cada generación de tarjetas gráficas programables se denomina como *Shader Model N*, esta denominación viene del *API Direct3D* de *Microsoft*.

La última generación de tarjetas comenzó con la *Nvidia GeForce 8800*. Esta generación permite programar el pipeline de vértices y el de píxeles, como generaciones anteriores, y también agrega una etapa nueva que permite generar geometría en la *GPU*. Esto se puede utilizar para varias técnicas, como generar pelaje (Figura 9), o subdividir geometría en tiempo real.

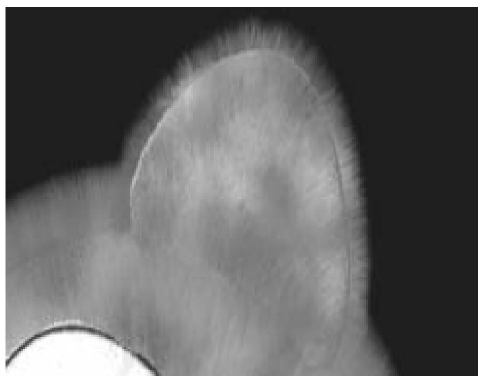

Figura 9 Pelaje renderizado usando la etapa de geometría en *Shader Model 4*(12)

2.2.2 APIs de gráficos

En los PCs hay dos APIs que se pueden usar para acceder al hardware de gráficos: *OpenGL*(13) y *Direct3D*(14). *OpenGL* es un API multiplataforma actualmente desarrollado por un consorcio de empresas. En cambio *Direct3D* es desarrollado por *Microsoft* y solo ejecuta en sistemas operativos *Windows*.

El consorcio que desarrolla *OpenGL* ha demorado bastante en liberar nuevas versiones del estándar, los fabricantes de tarjetas de video compensan esto liberando extensiones propias al API para acceder a nuevas funcionalidades. A su vez, el consorcio estandariza algunas extensiones que eventualmente son incluidas dentro del estándar del API. Por su parte *Microsoft* trabaja junto con los fabricantes de tarjetas de video para liberar nuevas versiones de su API con cada nueva generación del *hardware*. Esto ha llevado a que las generaciones de tarjetas de video sean catalogadas por la versión de *Direct3D* que soportan.

En la actualidad se puede acceder a prácticamente las mismas funcionalidades del *hardware* desde cualquiera de los dos APIs, las diferencias más importantes son: el soporte de los fabricantes de *hardware*, las herramientas disponibles y las plataformas que soportan. *Direct3D* es superior en las dos primeras categorías. Sólo dos fabricantes dedicados a la producción de hardware gráfico, *nVidia* y *Ati*,

tienen soporte de las últimas versiones de *OpenGL*. *Intel*, que es actualmente quien tiene más tarjetas gráficas en el mercado(15), no soporta las últimas versiones de *OpenGL*. *Direct3D* también tiene mejores herramientas disponibles, por ejemplo *nVidia PerfHud*(16) es superior a su equivalente más cercano en *OpenGL*, *gDEBugger*(17). *OpenGL* tiene soporte en más plataformas, pero los desarrolladores de juegos igual tienden a elegir *Direct3D* porque *Windows* es sin lugar a dudas el sistema más instalado en PCs(18).

2.2.3 Técnicas

En esta sección se presentan técnicas de renderizado en tiempo real usualmente utilizadas en videojuegos modernos, y que se consideraron para implementar en Trifulca. Separamos las técnicas en tres grupos: iluminación, sombras y animación.

Iluminación.

Para lograr tanto realismo como visualizaciones artísticamente interesantes, los juegos actuales usan varias técnicas de iluminación. Algunas de estas técnicas son *High dynamic range lighting* (HDRL), *Normal maps* y *Ambient occlusion*.

HDRL(19) es una técnica de renderizado en la que se usa un rango extendido de valores de luz para los cálculos de iluminación. Anteriormente se usaban números entre cero y uno para representar valores de luz, siendo cero la falta total de luz y uno el valor más brillante. Pero esto resulta en errores de redondeos en los cálculos y falta de flexibilidad para representar las luces. Con HDRL se puede tener valores de luz en un rango mucho mayor, pudiendo representarse por ejemplo el valor de luz del sol al mismo tiempo que se representa el de una lamparita. Por más que los cálculos se realizan en un rango mayor, al final es necesario llevar los valores de luz al rango que puede mostrar un monitor. Esto se resuelve simulando lo que hace la pupila humana, o sea cuando los valores de luz visibles son altos se mapea un rango alto al rango mostrable. De esta forma también se puede simular la adaptación del ojo cuando pasa de una zona de baja iluminación a una de alta iluminación.

Otra técnica muy usada son los *Normal maps*, que es una forma de implementar *Bump mapping* (20). Esta técnica logra simular relieve en una superficie mediante la variación de parámetros usados en la iluminación. En particular se varían las normales de la superficie (Figura 10). Las normales representan la orientación de la superficie. Se utilizan en las ecuaciones de iluminación, junto con la dirección hacia la luz, para ponderar la intensidad con que la superficie está iluminada. Usando las normales de una superficie rugosa en la iluminación de una superficie plana se logra visualizar la superficie rugosa a un costo de procesamiento mucho menor aunque se tiene un costo en memoria ya que las normales se guardan en texturas para poder acceder rápidamente.

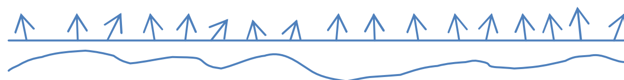

Figura 10 La superficie inferior es simulada por otra plana, con sus normales alteradas.

Ambient occlusion es una técnica de iluminación usada normalmente para renderizado no en tiempo real (Figura 11). En esta técnica se calcula para cada punto de geometría un valor de oclusión respecto del resto de la geometría, estos valores no dependen de ninguna luz, sólo de la geometría. El cálculo del valor de oclusión de un punto se hace disparando rayos desde el punto en todas direcciones y tomando una ponderación de las distancias en las que los rayos colisionan contra el resto de la geometría. En tiempo real se está usando una versión simplificada de esta técnica que calcula una aproximación de la oclusión con los valores finales de profundidad y normal de cada píxel de la imagen(21).

Figura 11 Escena iluminada mediante *Ambient occlusion*(22).

Sombras.

Hay dos técnicas utilizadas actualmente para calcular sombras en tiempo real. *Stencil shadows*(23) (Figura 12) es una técnica aplicada a nivel de la geometría. Para cada objeto y luz se calcula el volumen de sombra que el objeto proyecta en el mundo. Luego, cuando se está renderizando se interseca este volumen con el resto de la geometría del mundo para obtener los píxeles que estén en sombra respecto del objeto. Esta técnica da como resultado sombras con bordes duros, se pasa de estar en sombra a estar en luz sin valores intermedios. Es posible crear sombras con bordes suaves usando *Stencil Shadows* pero es muy costoso.

Figura 12 ***Doom 3*** **usando** ***Stencil shadows*** **para sus sombras(24).**

La otra técnica utilizada para sombras es *Shadow mapping*(25) (Figura 13). Para esta técnica se renderiza el mundo desde el punto de vista de la luz, guardándose la distancia al objeto más cercano en cada punto de la imagen. Luego, cuando se renderiza la imagen desde el punto de vista de la cámara, se compara la distancia de cada píxel a dibujar con la distancia correspondiente guardada en la imagen generada al renderizar desde la luz, si la distancia es mayor entonces el píxel está en sombra. Con *Shadow mapping* es más fácil generar sombras suaves.

Figura 13 GTA4 usando ***shadow maps*** **con bordes suavizados (26).**

Animación.

Las animaciones describen la transformación de un valor en el tiempo. De esta forma se pueden ver como funciones de los reales al valor animado (posiciones de vértices o matrices de transformación). Una técnica común para representar una animación son los *keyframes*. Un *keyframe* es el valor de la función de la animación en determinado instante de tiempo. Para hallar el valor en un instante de tiempo no definido se interpola usando los *keyframes* más cercanos.

Actualmente se utilizan dos métodos para animar personajes en juegos 3D, *morph targets* y animación esquelética. Ambos métodos se basan en *keyframes* para describir la animación. En *morph targets* el valor guardado por *keyframe* es la posición de todos los vértices del modelo animado. Este método se usa para animar expresiones faciales. Para animar los movimientos de un personaje se usa animación

esquelética. Se define un esqueleto asociado a los vértices del modelo. Al mover el esqueleto, los vértices lo siguen. De esta forma se pueden tener varios modelos asociados a una misma animación.

2.3 Soporte de red en juegos multijugador

En esta sección se discutirá el estado actual del soporte multijugador a través de la red comenzando por las topologías y arquitecturas utilizadas en los diferentes tipos de juego, luego las técnicas que se usan para disimular las condiciones de la red y finalmente los diferentes protocolos de transporte utilizados.

2.3.1 Arquitecturas

La arquitectura en la red de un juego refiere a cómo se disponen y conectan las computadoras participantes así como también distribución de la responsabilidad del procesamiento del juego.

La arquitectura más usada en los videojuegos de acción a la fecha es cliente-servidor(27). En esta arquitectura existe un servidor central conectado al resto de las computadoras que serán los clientes y cada cliente se encuentra conectado solamente con el servidor. Esta arquitectura es usada principalmente para que el servidor sea el único que procese los datos del juego.

Como principal alternativa existe la arquitectura punto a punto, en donde cada cliente se encuentra conectado con el resto. La ventaja de la primera sobre la última es que permite que haya un solo árbitro (el servidor) que decide el avance del juego evitando lógica de sincronización extra entre cada par de clientes y posibles errores en la toma de decisiones. Además el único punto con altos requerimientos de ancho de banda es el servidor lo que favorece una configuración en donde éste sea provisto por una empresa que puede costearlos. Para los clientes alcanza con que utilicen subscripciones hogareñas. Otra razón, para el caso en que se juegue a través de Internet, es la facilidad del establecimiento de conexión para clientes de una red que acceda a Internet a través de un NAT debido a que de esta manera el servidor es el único que recibe conexiones entrantes y posee un puerto bien conocido. La ventaja de la arquitectura punto a punto por sobre la cliente-servidor es que el tiempo de respuesta es mucho menor, cada cliente comunica sus intenciones directamente al resto sin intermediarios. Además no existe un único punto de falla. Con respecto a la cantidad de jugadores, la arquitectura punto a punto escala en orden n^2 en la cantidad de conexiones (28) y requiere que cada jugador tenga suficiente ancho de banda para comunicar y recibir datos del resto, siendo entonces aceptable sólo para juegos con pocos jugadores.

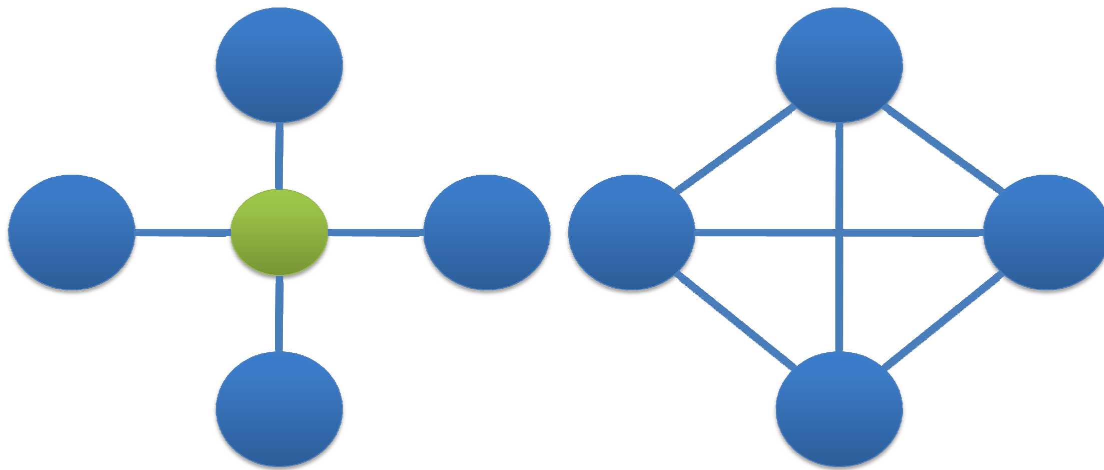

Figura 14 Arquitecturas cliente-servidor (izquierda) y punto a punto (derecha).

Un problema recurrente asociado a las arquitecturas es la confiabilidad de los usuarios. Se ha puesto mucho énfasis en tratar de evitar las trampas en los juegos por red, dado que la competitividad de muchos juegos lleva a que los jugadores utilicen los problemas de seguridad del diseño del juego para ganar.

Existen dos tipos de asignación de responsabilidades:

- Responsabilidad dividida, en donde cada uno de los clientes toma decisiones con respecto a sus acciones y las replica.
- Responsabilidad compartida, en donde se replican los comandos de todos los clientes al resto y cada uno los simula exactamente con la misma lógica.

El primer caso es muy susceptible a trampas de usuarios malintencionados (29) ya que una versión modificada del producto podría mandar mensajes alterados con decisiones que lo favorezcan. Por ejemplo, normalmente un usuario en un cliente dispara, se evalúa si el disparo pegó y en caso afirmativo se replica este hecho, en este caso se permite una situación que el usuario dispare y no de en el blanco pero sin embargo replique el hecho contrario.

Otras arquitecturas multiservidor son usadas principalmente en juegos MMORPG (30) (31) (*Massive Multiplayer Online Role Playing Game*) en los cuales el mundo del juego se puede dividir entre diferentes servidores. Los clientes se mantienen conectados a un servidor mientras estén en la parte del mundo que les corresponde. Cuando el personaje de un cliente conectado a un servidor A se mueve al área correspondiente a un servidor B, el servidor A le informa al servidor B que va a tener un nuevo cliente. Luego se le informa al cliente que debe conectarse al servidor B.

2.3.2 Técnicas

Cómo los recursos de red son muy limitados y además el flujo de datos provisto puede ser inestable se utilizan diversas técnicas para aminorar el impacto de estos factores.

Compresión delta.

Esta técnica es bastante simple y se usa principalmente para ahorro de ancho de banda. Se usa más que nada en arquitecturas de cliente-servidor, en donde el servidor es el único árbitro. El servidor debe enviar únicamente la diferencia del estado del juego a los clientes en vez de enviar un estado completo. Por ejemplo si la posición de un personaje del juego no cambia entre actualizaciones entonces el servidor puede enviar que la posición no cambió en vez de enviar la misma posición.

Esta técnica no sólo es muy usada en juegos (32)(33), sino que también se usa mucho en Internet a través de HTTP(34).

Tablas de strings.

Las tablas de *strings* corresponden a mapeos de *strings* usados frecuentemente, en vez de manejar los *strings* completos se usan identificadores de mucho menor tamaño. Este sistema es usado en *Windows* para la comunicación entre procesos mediante el protocolo DDE(35) (*Dynamic Data Exchange,* Intercambio dinámico de datos). En *Windows* el identificador del *string* se conoce como *Atom*(36).

Predicción del cliente.

La predicción del cliente se usa en arquitecturas cliente-servidor en donde el cliente envía comandos al servidor para ser ejecutados. Sirve para que latencias elevadas de la red no se noten en los tiempos de respuesta de los comandos. Consiste en que el cliente ejecute los comandos con la misma lógica que el servidor al mismo tiempo que se los envía para poder presentar la acción inmediatamente. De cierta manera se puede decir que predice la ejecución del servidor.

El estado que predice el cliente puede ser incorrecto. Esto sucede porque lo que ve el cliente del estado del mundo no es igual a lo que tiene el servidor, lo que ve está atrasado, mientras que el personaje del jugador se encuentra actualizado por la predicción. Puede pasar que un movimiento predicho en el cliente puede tener un resultado diferente en el servidor por interacciones con otros objetos que no se dan en el cliente. Un ejemplo sería que el personaje del cliente atraviese una puerta que en el servidor se encuentre cerrada. En el cliente se vería que el personaje atraviesa la puerta mientras que el servidor no deja atravesarla.

El cliente se da cuenta de estos errores cuando el servidor envía el estado verdadero del personaje. En este momento el cliente debe volver a ejecutar todos los comandos que se ejecutaron desde que comenzó la predicción.

El primer juego en usar predicción del cliente fue el *QuakeWorld*(37). Hoy en día es usada en la gran mayoría de los juegos de acción con soporte de red (38).

Compensación de lag.

Muchos juegos de acción en primera persona tienen lo que se llama disparo instantáneo, esto es que el proyectil que dispara un jugador tiene velocidad infinita e impacta en el objetivo al momento del disparo. Si lo que ve el cliente se encuentra atrasado con respecto al servidor, entonces cuando un cliente dispare a un objetivo puede que el disparo falle porque en el servidor ese objetivo ya se haya

movido. Esto hace que los jugadores tengan que estimar su propio *lag* y apuntar a donde el objetivo está en el servidor y no a donde lo ve. El *Quake 3* reproducía un sonido si el disparo fue efectivo para ayudar a esta estimación(32).

Una manera de evitar completamente esta situación fue propuesta por *VALVE* para el motor *Source*(32) y se denomina compensación de *lag*(32). La compensación de *lag* propone que el cliente envíe el tiempo del estado que se está mostrando (ya sea interpolado o no) al momento de replicar un disparo. De esta manera el servidor puede volver atrás la simulación para ejecutar el disparo y ver si realmente pegó o no con respecto a lo que veía el cliente en ese tiempo.

En la Figura 15 se puede ver un ejemplo en el juego *Counter Strike Source* corriendo en el servidor simulando *lag*. El modelo del contra-terrorista es la posición actual en el servidor, la geometría azul es la simulación con *lag* y el rojo es el estimado.

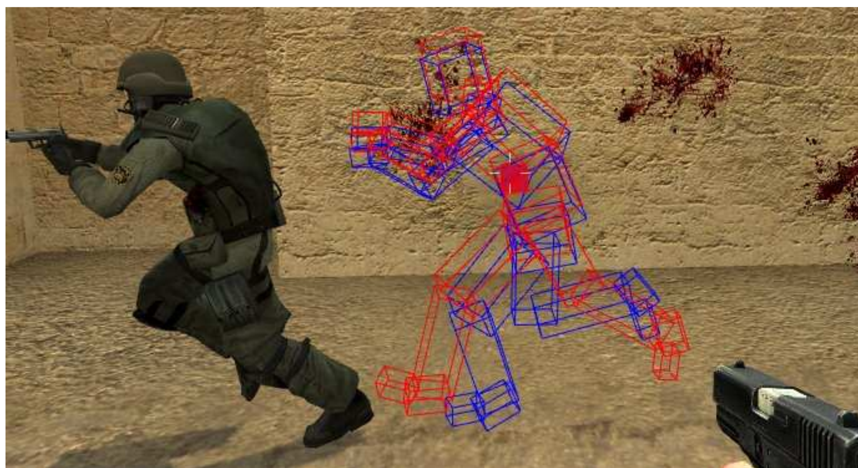

Figura 15 Compensación de lag(32)

Este método puede producir ciertas irregularidades. Por ejemplo un jugador puede resultar herido por más que se encuentre cubierto porque otro le pegó en una posición “vieja”.

Extrapolación.

Durante un juego en red pueden ocurrir situaciones en las cuales un cliente no recibe el estado de un objeto a mostrar en un largo período de tiempo. Esto puede causar saltos en el juego ya que no se tiene que mostrar. Para superar esto se propone generar un nuevo estado extrapolando a partir de los anteriores. Por lo general se puede usar en objetos que tienen física realista, ya que conociendo algunos parámetros como velocidades o aceleraciones, o estimándolos de posiciones anteriores, se puede predecir las posiciones de los objetos simplemente ejecutando la misma simulación que se ejecutó en el servidor. Puede ocurrir que el valor extrapolado difiera del valor estimando por interacciones desconocidas. Para corregir este error de manera suave y continua se puede dirigir el objeto a su posición real mediante fuerzas correctivas o moviéndolo por una guía definida por una función de los puntos inicial y final. Un ejemplo de este último caso puede ser *splines* cúbicos(39), que tomarían en

cuenta no solamente la última posición sino también posiciones anteriores de manera que la corrección no sólo sea continua sino geométricamente continua de nivel 2 (G_2), evitando movimientos bruscos.

La extrapolación también se puede usar para ahorrar ancho de banda. En este caso un cliente predice la posición de un objeto en movimiento y el dueño de este objeto ejecuta la misma lógica para saber qué es lo que lleva predicho el cliente. Como el dueño posee la predicción y el movimiento real del objeto, puede entonces calcular la diferencia entre ambos y cuando detecta que el cliente tiene un error mayor a un cierto límite entonces envía una actualización de estado. Este uso de la extrapolación es conocido como *dead reckoning* y fue diseñado para simulaciones militares(40).

2.3.3 Protocolos de transporte

Existen varios protocolos de transporte para diferentes tipos de aplicaciones. Entre ellos los más destacados y usados son TCP(41) y UDP(42). Sin embargo existe la especificación de otro diseñado para aplicaciones en tiempo real diferentes de video y audio. Este último es denominado DCCP (43).

TCP es un servicio confiable que garantiza el orden y la llegada en el envío de datos. Es muy usado en aplicaciones que requieren envíos sin pérdidas y en dónde el tiempo de espera es secundario. En el caso de redes locales el ancho de banda es por lo general suficientemente grande y el tiempo de respuesta prácticamente inmediato, por lo que TCP no muestra retrasos. En cambio si consideramos Internet, en donde se producen cambios de caminos que provocan diferencias de tiempos, desorden de paquetes o *routers* congestionados, entonces los controles de este protocolo se hacen notar. De acuerdo con la especificación puede ocurrir que nuevos segmentos queden a la espera del reenvío y posterior confirmación de un paquete que fue descartado. Además los algoritmos de control de congestión (44)(45), como comienzo tardío, agregan demoras adicionales al reducir la capacidad de envío y por lo general esperan al fin de la congestión para retomar la capacidad original.

El impacto de estas demoras es notorio en aplicaciones de tiempo real y para un juego es más importante el último estado que estados intermedios.

UDP, en contraste con TCP, es un protocolo mejor-esfuerzo, es decir, no garantiza la llegada o el orden en que llegan los paquetes. Además no provee ningún mecanismo de control de flujo o de congestión. Sin embargo garantiza la corrección de los datos enviados y además no agrega ningún tiempo extra al envío. Se usa por lo general cuando se quiere evitar el establecimiento de una conexión, por ejemplo para transacciones simples, llegando a durar un pedido y una confirmación solamente un *RTT* (*Round Trip Time*, tiempo de ida y vuelta).

Una implementación basada en UDP evita las demoras impuestas por TCP pero puede agregar otras debido a la falta de controles de congestión. En contratos hogareños de Internet por lo general se da una diferencia sustancial entre el ancho de banda brindado para subida de datos y el de bajada, siendo el ancho de banda para el envío bastante bajo, del orden de 128 Kbps en Uruguay(46). Los ISP's (proveedores de Internet) suelen encolar los paquetes en caso de ráfagas para mantener el ancho de banda provisto sin la necesidad de descartar algunos que pueden ser enviados en los intervalos de inactividad. Se usan algoritmos para determinar cuál es el siguiente paquete a enviar de la cola y cuál es

el que debe ser eliminado en caso de que se llene, siendo los más usados en Internet FIFO (*First In – First Out*, primero entra, primero sale) y *DropTail* (descartar el último)(47). Estas elecciones justamente son las que se tratan de evitar en aplicaciones de tiempo real, ya que priorizan paquetes viejos y agregan demoras a los nuevos.

Hoy en día la falta de control de congestión de UDP y la cantidad de aplicaciones que usan este protocolo sin implementar mecanismo alguno es considerada una amenaza para Internet pudiendo provocar un colapso por congestión (48). La dificultad de agregar estos controles en capas sobre UDP o debajo de esta, y de modificar o anexar protocolos existentes como TCP, llevó al diseño de una nueva especificación para un protocolo de transporte denominado DCCP(49).

DCCP intenta suplantar a UDP para las aplicaciones de tiempo real dando las mismas facilidades que UDP con lo que respecta a tiempos de respuesta inmediatos y verificaciones de integridad, y además agrega control de congestión. DCCP fue concebido para aplicaciones en tiempo real como juegos, voz sobre IP y video sobre IP.

2.4 Física y fracturas

Las simulaciones físicas se encuentran en los juegos actuales como parte integral de su jugabilidad o simplemente para ofrecer efectos visuales. Se investigaron algunas bibliotecas de simulación de física para ser incluidas en el juego.

2.4.1 Conceptos

Es necesario introducir algunos conceptos usados en las simulaciones físicas.

Una de las funcionalidades básicas que necesita todo simulador de física es detección de colisiones. Existen dos categorías de detección de **colisiones: discretas y continuas**. La detección de colisiones discreta toma la posición en un momento dado de todos los cuerpos y da como resultado una colisión si dos cuerpos intersecan. En la detección de colisiones continua se toma la posición y velocidad de cada cuerpo y se resuelve en cuanto tiempo ocurrirá la primer colisión. Nunca se llega a que la intersección de dos cuerpos sea un volumen. Las colisiones continuas son más precisas y permiten evitar situaciones en que un objeto traspase a otro como puede suceder en colisiones discretas. Se usa por ejemplo para la pelota en un juego de fútbol, que es un objeto que puede alcanzar altas velocidades y es importante que no traspase otros objetos como los palos de un arco.

Todos los simuladores proveen simulación de **cuerpos rígidos**. Los cuerpos rígidos son cuerpos que ocupan espacio y tienen propiedades físicas, como centro de masa y momento de inercia. La denominación de rígidos está dada porque los cuerpos no sufren deformaciones.

Para formar cuerpos más complejos se unen varios cuerpos rígidos usando **empalmes.** Los empalmes unen varios cuerpos restringiendo sus movimientos relativos. Por ejemplo un empalme puede restringir la rotación de un cuerpo respecto del otro a solo un eje, comportándose así como si fuera una bisagra.

Para facilitar la interacción de personajes manejados por usuarios, algunos simuladores incluyen *character controllers*. Un *character controller* es un cuerpo que colisiona con los demás cuerpos conocidos por el simulador, pero no reacciona con leyes físicas a estas colisiones. Su comportamiento se rige por la lógica del juego y la entrada del usuario.

2.4.2 Simuladores de física

Se estudiaron cuatro bibliotecas de simulación de física para el juego: *Bullet*(50), *ODE*(51), *Newton*(52) y *PhysX*(53).

Bullet es un simulador de física *open-source.* Está implementado en *C++*, con un API orientado a objetos. Soporta tanto colisiones discretas como continuas, simulación de cuerpos rígidos, y empalmes. Si bien tiene colisiones continuas, la simulación de cuerpos rígido se realiza con colisiones discretas. El proyecto fue creado por un ex empleado de *Havok*(54), una empresa dedicada a desarrollar *engines* comerciales, que actualmente trabaja para *Sony*. Por esto cuenta con el apoyo de *Sony* para su desarrollo y está soportado en la consola *Playstation 3* de este fabricante.

ODE es otro simulador *open-source*, soporta simulación de cuerpos rígidos, empalmes y colisiones discretas pero no continuas. Está implementado en *C* y ha sido usado en algunos juegos comerciales. Los desarrolladores detrás de *ODE* crearon su propio algoritmo iterativo para resolver respuestas a colisiones y empalmes. Este algoritmo es más rápido que sus competidores, pero menos robusto(55).

Newton es también un simulador *open-source*, con simulación de cuerpos rígidos, empalmes, colisiones discretas y colisiones continuas. A diferencia de los otros simuladores tiene un algoritmo determinista para resolver respuestas a colisiones y empalmes. Esto hace que sea mucho más robusto aunque menos eficiente.

PhysX es una biblioteca comercial, aunque se puede usar libremente. Tiene soporte de todos los conceptos mencionados, y además agrega simulación de cuerpos deformables. Es el único de los *engines* estudiados que incluye su propio *character controller.*

2.4.3 Fracturas

El problema consiste en generar las fracturas en el escenario y objetos sueltos del mundo a partir de la posición y la fuerza de un impacto. También es necesario poder identificar las fracciones que quedan separadas del objeto original para poder tratarlas como objetos sueltos.

Varios artículos han sido escritos sobre cómo resolver este problema. Todos aplican física de materiales, se modelan los objetos como sólidos para los cuales se resuelven las tensiones internas a partir de las deformaciones causadas por fuerzas externas (ej. colisiones). Desde este punto de vista, simular fracturas es muy similar a simular deformaciones. Las tensiones internas se pueden usar tanto para ver donde el material falla, como donde se deforma.

El modelo físico nos da una función continua de la tensión dentro del objeto. Para poder simularlo en una computadora se discretiza el problema. Los objetos poliédricos se separan en tetraedros y se

resuelven las tensiones en los vértices de los tetraedros. Esto nos da una aproximación por partes de la función continua.

O'Brien (56) y Müller (57) proponen fórmulas equivalentes para calcular la tensión en los vértices. Los cálculos consisten en la resolución de un par de sistemas lineales por vértice de los tetraedros. Las tensiones obtenidas son matrices 3x3 donde los vectores propios corresponden a las direcciones de mayor tensión y los valores propios a la magnitud de la tensión. Si el mayor valor propio es mayor que una constante del material, el material falla en ese punto y se fractura. Luego de que se sabe que un vértice falló, O'Brien separa los tetraedros adyacentes usando el plano perpendicular al vector propio correspondiente al mayor valor propio. En cambio Müller propone partir todos los tetraedros en un cierto radio del vértice que falló. De esta forma Müller acelera la propagación de las fracturas. Müller también propone otras optimizaciones que permiten correr estos algoritmos en tiempo real.

Smith (58) propone una solución diferente a la de Müller y O'Brien. En lugar de utilizar tetraedros, utiliza puntos distribuidos dentro del sólido y unidos por restricciones de distancia. Luego a partir de fuerzas aplicadas a algunos de estos puntos resuelve una distribución de fuerzas en todos los puntos. La principal diferencia con las soluciones de O'Brien y Müller es que no obtiene planos por donde el objeto debe fracturarse, los objetos siempre se rompen en los lugares predeterminados por la distribución de puntos. Por esto la solución de Smith presenta dentados en las fracturas. La solución de Smith parece ser más rápida, aunque los resultados visuales no son tan buenos. Además genera objetos muy dentados que pueden enlentecer el renderizado.

2.5 Scripting

Hoy en día la mayoría de los motores de juegos se basan de una forma u otra en lenguajes de *scripting*, para lograr que la lógica sea fácilmente modificable y extensible sin tener que modificar el motor del juego. Otra razón para usarlos disminuir del tiempo de desarrollo, ya que por lo general son interpretados en vez de compilados y son más simples en el sentido que liberan al programador de tareas como el manejo de memoria.

En el comienzo estos lenguajes eran simplemente archivos de configuración que permitían la definición de algunos parámetros del juego. Por ejemplo en el caso del *Carmageddon*(59) cada vehículo estaba definido por un archivo que dictaba declarativamente parámetros asociados a autos como lo eran velocidad máxima, torque, etc. Otra manera que se usó para permitir la modificación de juegos era simplemente separar la lógica a librerías de carga dinámica, como el caso del motor *ID Tech 2*(60), y, en parte, *ID Tech 3* (61). Con el aumento en la capacidad de procesamiento y la separación del procesamiento de gráficos del procesador principal a uno especializado se pudo avanzar en las simulaciones de física avanzadas y se logró usar lenguajes que corrían completamente interpretados dentro del mismo motor del juego. Estos lenguajes logran simplificar la programación de la lógica y ayudan a que los motores puedan llegar a ser *frameworks* multigénero sobre los cuales se pueden diseñar títulos totalmente diferentes entre sí, como por ejemplo el *Unreal Engine 3*(62), sobre el cual se desarrollan el *Unreal Tournament 3*(63), un juego de acción en primera persona multijugador, y *Mortal Kombat vs DC* (64) , un juego de peleas uno a uno.

Los lenguajes considerados para este desarrollo fueron *Lua*(65), *Python*(66) y *Squirrel*(67).

Lua es un lenguaje que comenzó para la creación de archivos de configuración y fue extendido hasta ser un lenguaje de programación. Es dinámico, multi-paradigma y débilmente tipado. Provee mecanismos de meta-programación para extender el lenguaje, facilitando, por ejemplo manejar orientación a objetos. Tiene un *garbage collector* incremental para el manejo de memoria(68). La implementación de la librería *C* no ocupa más de 150KB haciendo que sea muy liviano en comparación con las implementaciones de otros lenguajes. Este lenguaje es muy usado para videojuegos(69). Algunos ejemplos de juegos que usan *Lua* son: *Far Cry, Crysis, Sim City 4, World of Warcraft* y *S.T.A.L.K.E.R.: Shadow of Chernobyl*(70).

Python es un lenguaje dinámico, orientado a objetos y débilmente tipado usado por muchas aplicaciones para agregar *scripting*. El mismo tiene una amplia lista de bibliotecas para simplificar la programación. *Python* basa su manejo de memoria en conteo de referencias y además incluye un *garbage collector* en caso de que sea necesario el uso de ciclos. *Python* es muy usado para la extensión de herramientas, pero el interprete es considerado lento (71) (en comparación con *Lua*) para su uso en juegos.

Squirrel fue diseñado por desarrolladores del *Far Cry* para suplantar a *Lua*(72). El lenguaje es también dinámico pero impone la orientación a objetos. Su sintaxis es muy similar a la de C/C++ ya que sus desarrolladores argumentan que la mayoría de los programadores usa lenguajes con sintaxis parecida. Además incluye un manejo de memoria que mezcla conteo de referencias y *garbage collection* para evitar las ráfagas de consumo de CPU que se podían ver en *Lua* cuando se ejecutaba el *garbage collector*.

3 Diseño de Trifulca

En este capítulo se explica el diseño de Trifulca. Se comienza por describir la mecánica del juego y el modelo de datos, llamado el mundo. Luego se explica qué ocurre en un pase de ejecución del juego y finalmente se introducen los otros subsistemas: gráficos, red, lógica, física, cámara, datos y entrada de usuario.

3.1 Mecánica del juego

La mecánica del juego es simple: varios jugadores se atacan con el objetivo de lanzar a los otros fuera del escenario. Cada jugador tiene tres vidas y un contador de daño. El contador de daño se incrementa con cada golpe recibido, distintos tipos de golpe incrementan este contador en cantidades diferentes. A medida que el contador crece el jugador es empujado más lejos por los golpes de los otros jugadores. Cuando un jugador cae fuera del escenario pierde una vida, una vez se le acaban sale del juego, siendo el ganador el último jugador que quede en el mundo

3.2 El mundo

El mundo en Trifulca constituye el estado del juego. Dicho mundo se encuentra compuesto por un conjunto de entidades, conformando cada una de ellas, una representación determinada para un objeto del mundo. Estas entidades poseen las siguientes propiedades:

- **Transformación:** Representa la posición en el mundo. Es una matriz cuatro por cuatro por lo que puede representar traslaciones, escalados y rotaciones en un mundo tridimensional.
- **Representación visual:** Son los datos utilizados en conjunto con la transformación y animación para dibujar la entidad en pantalla.
- **Cuerpo físico:** Es el cuerpo tridimensional utilizado para colisiones y simulación física.
- **Lógica:** Es el comportamiento de la entidad frente a la interacción con el mundo y con el jugador.
- **Animaciones:** Son funciones de otras propiedades de la entidad en el tiempo. Se utilizan para animar la representación visual, y de esta forma lograr que, por ejemplo, una entidad jugador camine. También se utilizan para representar cambios de propiedades lógicas en el tiempo. Por ejemplo si se anima la propiedad de daño efectuado en colisión, junto con la animación visual de una patada, se logra el efecto lógico y visual de una patada.

Las entidades están organizadas en un árbol. Esto facilita su creación y destrucción cuando una entidad está compuesta por varias entidades.

3.3 Bucle principal

El bucle principal es la secuencia de instrucciones que se ejecuta continuamente durante el juego. Es importante conocerlo para entender cómo interactúan todos los componentes. Se puede decir que el bucle principal es quien orquesta el desarrollo del juego.

La secuencia de pasos seguida por una ejecución del bucle es la siguiente:

1. **Leer entrada de jugadores:** se lee el estado de los dispositivos de entrada (teclado y joysticks) y se traduce a entrada para la lógica.

2. **Sincronizar clientes y servidor:** si el juego está haciendo de servidor, recibe la entrada de los clientes y envía el nuevo estado de vuelta. Si es un cliente envía entrada y recibe estado.

3. **Avanzar lógica y física:** si pasó un tiempo mayor al de un *frame* lógico desde la última actualización (el tiempo de un *frame* lógico es constante), entonces se calculan un nuevo estado de la lógica y la física.

4. **Interpolar estados:** los estados calculados que se obtienen de la lógica o la red se guardan en una cola indicando en qué tiempo de juego ocurren. La imagen que se muestra en pantalla está atrasada con respecto a la lógica, de esta forma se pueden interpolar los últimos n estados a la hora de dibujar y lograr un movimiento mucho más suave (Figura 16).

5. **Calcular animaciones:** se calculan las transformaciones de las animaciones correspondientes al actual estado interpolado.

6. **Mover la cámara:** se posiciona la cámara teniendo en cuenta la posición actual de todos los jugadores.

7. **Dibujar:** se dibuja el estado interpolado en pantalla. Como se dibuja en todas las pasadas del bucle, una pasada del bucle coincide con un *frame de render.*

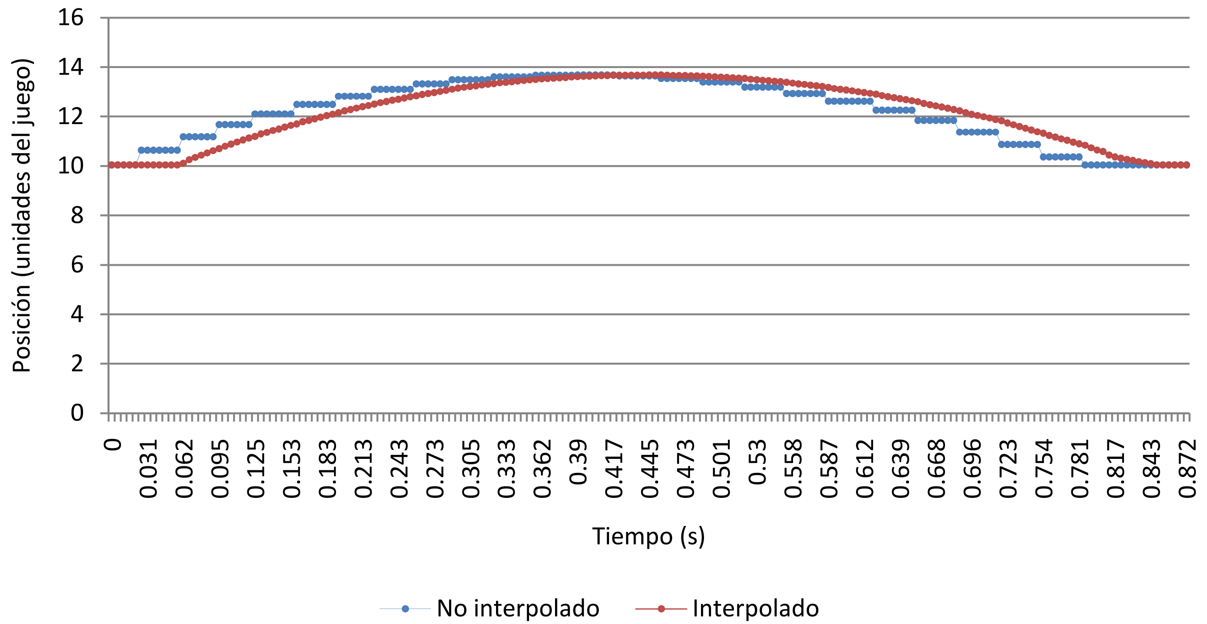

Figura 16 Resultado de la interpolación

En la Figura 16, el movimiento representado es el salto de un jugador. Al interpolar se logra un movimiento suave, mientras que si se usa el último estado se tienen saltos en la posición. Si estos saltos ocurrieran a intervalos regulares no habría problema, pero pequeñas variaciones en estos intervalos resultan en un movimiento oscilante. El corrimiento en la gráfica del movimiento interpolado es resultado del retraso con el que se realiza para poder tener dos o más estados entre los que interpolar. Esta gráfica fue tomada con el juego configurado para tener dos *frames* lógicos de retraso a 30 *frames* lógicos por segundo, esto es un retraso de 1/15 segundos.

3.4 *Pipeline* de Gráficos

El *pipeline* de gráficos (Figura 17) es el responsable de traducir el estado actual del mundo a una imagen en pantalla. Del estado del mundo al *pipeline* gráfico le interesan solamente las entidades que contienen geometría, *Overlays* o luces. Cargando de forma adecuada los datos de estas entidades, más los datos de la cámara, en la GPU se logra obtener la imagen deseada en pantalla.

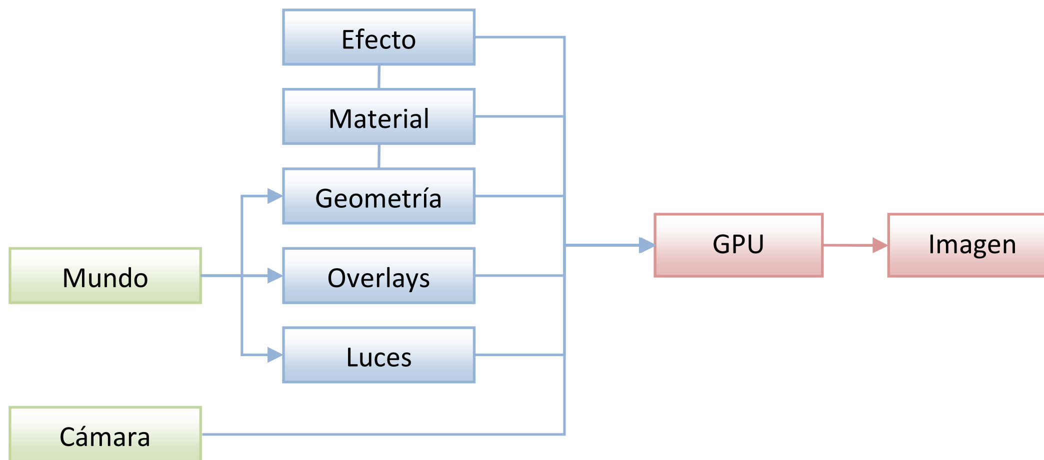

Figura 17 Los elementos principales del *pipeline* gráfico.

La geometría de una entidad es una lista de vértices que describen un conjunto de triángulos. Cada vértice está constituido por un conjunto de atributos, como su posición, vector normal y coordenadas de textura. Todos estos datos describen un objeto tridimensional que representa la forma de la entidad. Para poder dibujar una geometría es necesario también conocer los materiales asociados.

Una geometría se divide en sub-geometrías, cada una de estas tiene asociado un material usado para colorear la proyección de la geometría en pantalla. A su vez, cada material está asociado a un efecto. Los efectos describen el proceso ejecutado para colorear, mientras que los materiales establecen valores en parámetros de su efecto asociado. Los parámetros de un efecto pueden ser vectores, colores, matrices o texturas.

Por ejemplo, Trifulca usa en mucha de la geometría del mundo normal maps, un efecto que simula relieve donde en realidad la geometría es plana (Figura 18). Pero en cada uso el material puede definir establecer distintos parámetros para lograr distintos resultados.

Figura 18 Normal maps usados en dos materiales con distintas texturas.

Las luces también son entradas de los efectos. Trifulca soporta luces puntuales y direccionales. Cuando los efectos lo requieren, los datos de las luces actuales del mundo se pasan como parámetros.

Además de los parámetros definidos en los materiales y las luces, los efectos tienen de entrada datos dinámicos como la posición en el mundo del píxel que se está coloreando y los atributos interpolados de los vértices del triángulo al cual pertenece el píxel a colorear.

La mayoría de las entidades pertenecientes al mundo son dibujadas como geometrías del mundo tridimensional. También es necesario poder dibujar elementos bidimensionales, como menús, el puntaje actual del juego e información de *debug*. A estos elementos bidimensionales, que siempre se dibujan sobre el mundo tridimensional, se les llama *Overlays*. Los *Overlays* definen una región en la pantalla y pueden ser usados para dibujar tanto texto (Figura 19) como materiales de los antes. La principal diferencia con las geometrías es que no es necesario proyectar desde tres dimensiones a dos, los *Overlays* ya son bidimensionales.

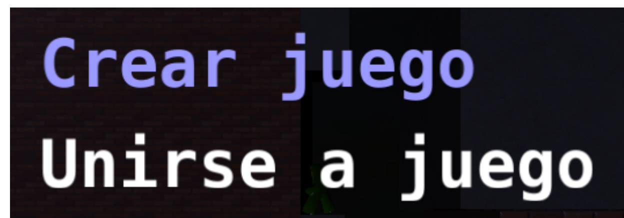

Figura 19 Se usan *Overlays* para dibujar los texto del menú y la capa transparente de atrás.

3.5 Red

El subsistema de red es el encargado de sincronizar el estado de las computadoras que se encuentran conectadas al juego.

La arquitectura de red elegida para esta sincronización es cliente-servidor porque es más simple de implementar y por los requerimientos de ancho de banda de los clientes. En Trifulca el servidor es el único árbitro del juego y los clientes envían comandos generados a partir de la entrada de usuario.

El diseño del subsistema de red es una arquitectura en capas en donde cada capa interactúa sólo con las capas superior e inferior. De esta manera se pueden modificar y extender cada capa por separado de manera que sea transparente para el resto.

A continuación se brinda una breve descripción acerca de cada capa.

3.5.1 Capa de plataforma

Esta capa se encarga de abstraer la plataforma usada. Encapsula el API de acceso a red de manera de no crear una dependencia con un sistema particular.

3.5.2 Capa de transporte

La capa de transporte provee un servicio con las siguientes propiedades:

- Orientado a conexión. Implica que debe haber un intercambio de información para establecer una conexión previo al envío de datos entre *endpoints*.

- No confiable. Este servicio no garantiza que un paquete enviado llegue al destino.
- Con garantía de orden. Se garantiza que un paquete no pueda llegar en diferente orden en que fue enviado.
- De comunicación bidireccional. Ambos *endpoints* de una conexión pueden enviar datos.

Esta capa brinda además un control de flujo simple mediante el establecimiento de un máximo de ancho de banda para ambos puntos de la conexión.

3.5.3 Capa de sincronización

El propósito principal de esta capa es actualizar el estado del juego y replicar los comandos enviados por los clientes. Replica ciegamente el estado del juego sin conocer su desarrollo.

3.5.4 Capa de juego

Esta capa determina las reglas de juego, cuándo se termina una partida y quién gana. Interpreta el estado recibido de la capa de sincronización separando las entidades con respecto a su tipo, si son mapas, jugadores u otros objetos y también asociándolas a cada jugador.

3.6 Lógica

El subsistema de lógica es el encargado de generar a partir de los comandos y el estado anterior el nuevo estado del juego.

Cada entidad del juego puede tener asociado un *script* de lógica el cual definirá como se comportará e interactuará con respecto al resto de las entidades. Este *script* puede acceder a diferentes funcionalidades del motor como lo son:

- Mover entidades cinemáticas.
- Consultar valores de cualquier entidad: nombre, posición, hijos, tipo de entidad (si es luchador o parte del mapa, cinemática, dinámica o rígida) y conocer la animación en la que se encuentra.
- Colisiones que ocurrieron en el turno, y de estas el punto de colisión y el vector de penetración.
- Agregar o quitar entidades hijas.
- Cambiar la animación de una entidad.
- Proporcionar y recibir daño de otras entidades.
- Crear y actualizar variables asociadas a las entidades que pueden ser replicadas por la red.
- Realizar colisiones continuas sobre el mundo, recibiendo una lista de colisiones con el punto de colisión y la distancia recorrida hasta llegar a este punto.

Las funcionalidades que se proveen a los *scripts* son las mínimas necesarias para lograr la lógica requerida en el juego.

3.7 Física

Dado que el juego es, desde el punto de vista lógico, bidimensional, el movimiento de los jugadores está limitado a un plano. Sin embargo se permiten diversos objetos que se pueden desplazar en todo el espacio y que pueden interactuar con ellos. Ejemplos de estos objetos en el caso del juego son cajas de madera que los luchadores pueden patear.

Por lo general los motores físicos manejan una sola opción, o bien son únicamente tridimensionales o bidimensionales. Esto llevó a la decisión de usar un motor tridimensional y adaptarlo a los requerimientos planteados. El motor elegido provee detección de colisiones 3D y dinámicas de cuerpos rígidos.

Para la limitación del movimiento del jugador surgieron dos opciones.

- Usar un cuerpo completamente dinámico como representante de la entidad en el motor, o sea, que el movimiento sea en realidad llevado por el motor mientras que el control es dado mediante la aplicación de impulsos y agregarle una restricción de movimiento. Esta opción permite que la programación sea más fácil, ya que el motor se encarga de la fluidez de los movimientos y de la respuesta de las colisiones. Por otra parte es más restrictiva ya que muchas veces se quiere tener más control y permitir movimientos que son físicamente imposibles, como aceleraciones infinitas al girar o detenciones abruptas. Además puede llegar a ser inestable, ya que con valores elevados de velocidad los errores de cálculo llevan a romper las restricciones que mantienen el movimiento deseado.

- Usar un cuerpo cinemático, es decir, un cuerpo para el cual la posición está definida por fuera del motor sin este poder modificarla. En este caso todo tipo de movimiento debe de ser manejado en su totalidad por la lógica del juego o una capa intermedia de movimientos predefinidos. Además es necesario proveer respuesta a las colisiones tanto con otros objetos dinámicos como cinemáticos, que, si no se manejan con cuidado puede lograr que los objetos que colisionan terminen “temblando” o incluso intersecados. Sin embargo, este método permite la libertad planteada en el punto anterior, que llevaría a una mejor jugabilidad.

La opción elegida fue la segunda. Como la naturaleza del juego es una mezcla entre plataformas y peleas, los movimientos que se quieren lograr deben tener una respuesta rápida de control, que requiere cierto irrealismo. Los cuerpos cinemáticos son los que brindan dicha opción.

Los mapas están definidos tanto por cuerpos dinámicos como por cuerpos estáticos. Los cuerpos estáticos son aquellos que nunca se mueven, ya sea por lógica del juego o simulación. Los cuerpos cinemáticos pueden cumplir la función de los cuerpos estáticos si la lógica no los mueve, pero la certeza de que un cuerpo no se va a mover permite al simulador de física realizar optimizaciones de performance.

La detección de colisiones se encuentra delegada en su totalidad al motor de física. Para la respuesta a las colisiones se implementan dos métodos principales que son *collide and slide* (colisionar y deslizar) y

collide and bounce (colisionar y rebotar)(73), permitiendo además que sea fácil agregar nuevos métodos. En el caso de las colisiones de los cuerpos dinámicos con los jugadores la respuesta la da el motor.

Como se usa una biblioteca de simulación de física externa, es necesario sincronizar las entidades del juego con las entidades manejadas por el simulador. El subsistema de física se encarga de hacer esta sincronización. En cada *frame* busca nuevas entidades en el mundo del juego para agregar a la simulación física, y borra del simulador las entidades que ya no están en el mundo. El subsistema de física también abstrae el uso del simulador permitiendo suplantarlo por otro sin impactar en los demás subsistemas.

3.8 Cámara

La función de la cámara es mostrar las entidades que son de interés para el jugador. Debido a que en Trifulca el juego ocurre en el entorno de un plano, la cámara debe mostrar una región de este entorno que incluye a las entidades que son de interés para los jugadores (Figura 20).

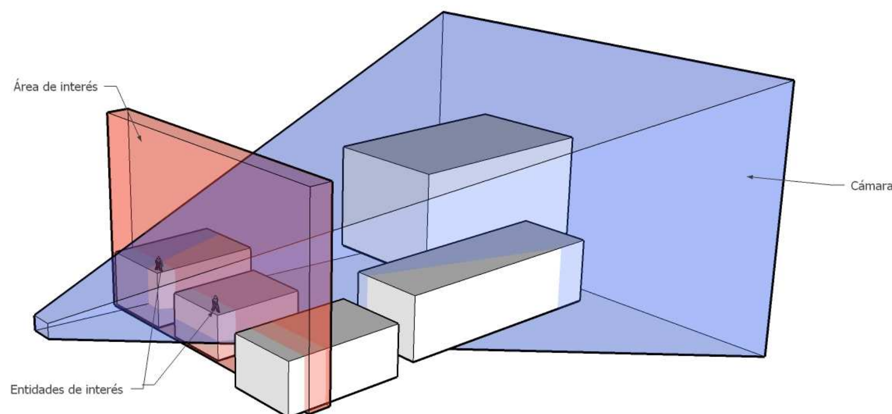

Figura 20 Representación de la cámara, la pirámide azul es el campo de vista, la región roja es donde ocurre la acción del juego.

En particular, la región mostrada por la cámara debe mostrar a todos los jugadores que estén jugando en una misma computadora. Existen dos soluciones clásicas para esto. Una es hacer que la cámara muestre a todos los jugadores (Figura 21), la otra es partir la pantalla en regiones y dedicar cada región a un jugador (Figura 22). Con la segunda opción si dos jugadores se encuentran cerca se desperdicia espacio de pantalla porque se muestra casi lo mismo en las dos regiones. En cambio si se intenta mostrar a todos los jugadores dentro de una misma cámara, es necesario alejar mucho el punto de vista, perdiéndose detalle y mostrando grandes regiones que no son de interés.

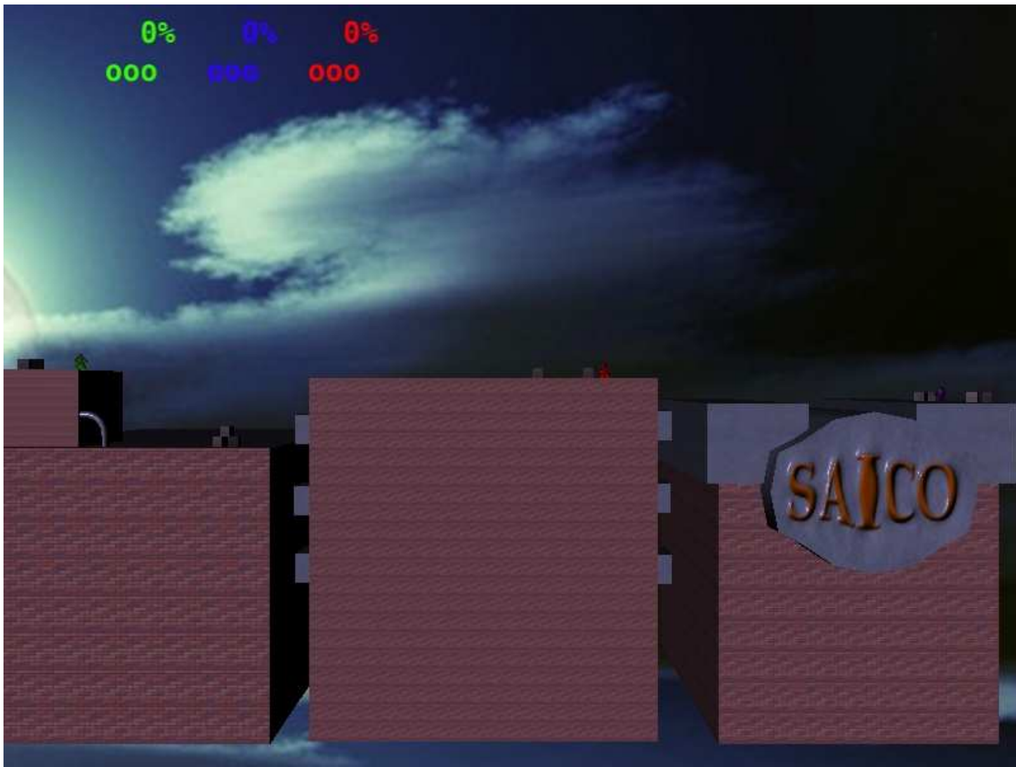

Figura 21 Trifulca sin pantalla partida.

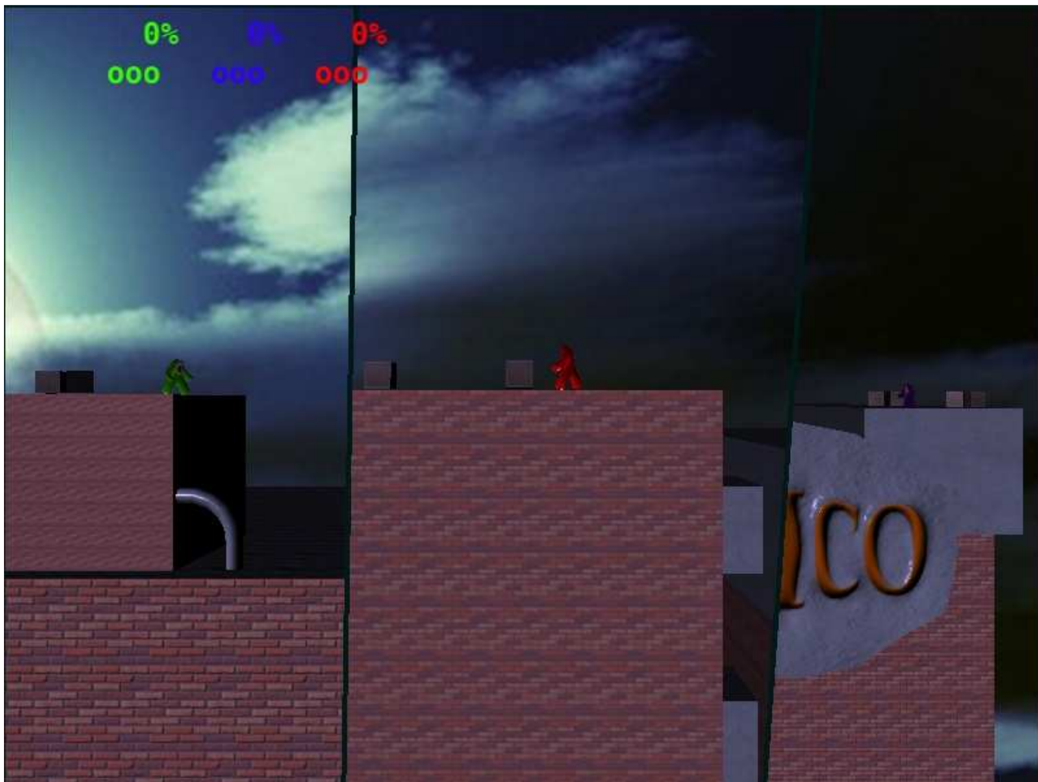

Figura 22 Trifulca usando pantalla partida con Voronoi.

Se propuso usar un enfoque hibrido para Trifulca. Cuando los jugadores están cerca la cámara es una sola, cuando se alejan la cámara se separa. Se encontró un solo juego que usa este enfoque, el *Dragon Ball Z Shin Butouden*(74), pero lo hace para solo dos jugadores. Se buscó una solución que permitiera como mínimo cuatro jugadores en una misma pantalla.

La solución propuesta utiliza diagramas de Voronoi para dividir el espacio en las regiones que se encuentran más cercanas a cada jugador. Luego se unen las regiones de los jugadores que están cerca y se divide la pantalla en las regiones que quedan. Esto logra una solución con varios beneficios:

- Escala a cualquier cantidad de jugadores.
- La división de la pantalla muestra la posición relativa de los jugadores en el mundo. Esto es, si un jugador se encuentra arriba y a la derecha de otro en el juego, la división de pantallas también lo va a mostrar arriba y a la derecha.
- Le da un valor estético interesante al juego.

3.9 Datos

Se diseñó un subsistema que abstrae la carga de datos al juego, buscando obtener beneficios para los programadores y creadores de contenido.

Este subsistema recarga los datos en el disco duro automáticamente cuando cambian, ahorrando tiempo durante la edición de datos, dado que no es necesario reiniciar el juego para ver el resultado dentro del mismo. También realiza carga tardía de los datos, se retrasa la carga hasta que realmente se requieren. De esta forma se distribuye el tiempo de carga cuando el juego ya está corriendo, simultáneamente disminuyéndose el tiempo inicial de carga. Por último cuenta con un sistema de archivos virtual. Los datos se cargan indicando identificadores llamados URIs (ya que usan la sintaxis de URIs definida por la IETF(75)), estos luego se mapean a archivos en disco para su carga. Esta funcionalidad permite tener todos los datos comprimidos en un solo archivo, para reducir espacio y tiempo de carga.

3.10 Entrada de usuario

El subsistema de entrada de usuario se encarga de tomar la entrada de diversos dispositivos, como teclados y joysticks, y la convierte en estados de un conjunto de botones internos del juego. La lógica usa esta representación abstracta de la entrada para tomar control de las entidades de los jugadores. Se definieron diez botones, cuatro de direcciones y seis para otras acciones, pero en el juego final solo se usaron dos de los botones de acciones. La conversión de la entrada a botones es simple en el caso de entradas discretas como un teclado, directamente se mapean teclas a botones. En el caso de un joystick es necesario definir en qué eje, y a partir de qué valor, el joystick activa un botón.

4 Implementación

En el presente capítulo se explica a grandes rasgos la implementación de los subsistemas más importantes de Trifulca. Estos son: gráficos, interfaz gráfica de usuario, cámara, lógica y red.

4.1 Gráficos

La implementación del código de gráficos abarca varios subsistemas independientes, cada uno de estos realiza tareas asociadas a una etapa del pipeline descripto en la sección 3.4 y visualizado en la Figura 17.

4.1.1 Geometría

La geometría a dibujar se representa con una sopa de triángulos. Cada triángulo está compuesto por tres vértices, donde cada vértice es un conjunto de atributos. Los vértices pueden ser compartidos por varios triángulos, para esto cada geometría tiene un vector común de vértices y los triángulos están definidos como tres índices del vector de vértices. Los atributos posibles para un vértice son:

- Posición: es el único atributo que siempre es requerido, representa la posición del vértice en el mundo.
- Normal: representa la orientación de la superficie en ese punto, se utiliza para iluminación.
- Color y Color secundario: color del vértice, normalmente no se usa porque se colorea con texturas.
- Coordenadas de textura: las texturas son imágenes que se mapean sobre la superficie de una geometría. Las coordenadas de textura describen como se realiza este mapeo. Pueden haber varios conjuntos de coordenadas de texturas.
- Peso e índice de huesos: usados para la animación esquelética explicada en la siguiente sección.
- Tangente y binormal: usados para realizar *normal mapping*. La tangente es un vector unitario contenido en el plano tangente a la geometría, y que es coplanar con la normal y uno de los ejes de coordenadas de texturas. La binormal (a veces llamada bitangente) es un vector unitario perpendicular a la tangente y la normal.

La sopa de triángulos se divide en secciones para aplicar materiales, cada sección tiene su propio material (Figura 23).

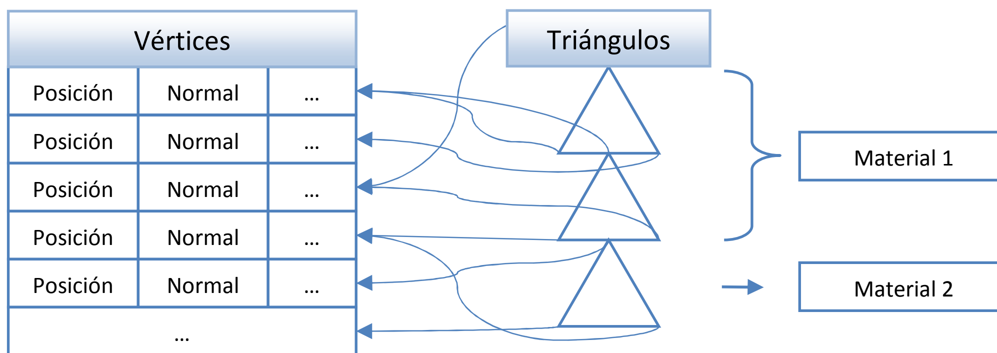

Figura 23 Estructura de la geometría.

Toda la información de una geometría es guardada en la memoria de la tarjeta de video para ser renderizada rápidamente. Podría llegar a ser necesario modificar la geometría en la CPU, y en este caso sería más conveniente tener la geometría en memoria del sistema, pero esto no se dio en el juego. La única geometría que es modificada para renderizarse son los personajes con sus animaciones, pero las animaciones son aplicadas enteramente en la GPU.

4.1.2 Animaciones

Para animar a los personajes se utilizó animación esquelética. Esto significa que existe un árbol de huesos, a las que la geometría está asociada y cuando los huesos se mueven la geometría se mueve con ellos. Cada hueso es una transformación, y cada una de ellas se define en el espacio de la transformación del hueso padre (Figura 24).

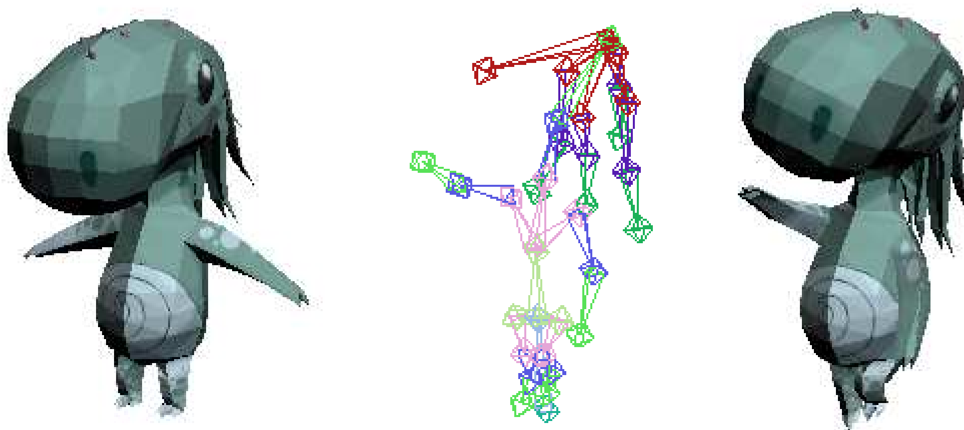

Figura 24 A la izquierda el modelo en su *bind pose* (esta es la posición en la que es creado por el modelador), en el centro el esqueleto posicionado, y a la derecha el modelo con la posición del esqueleto

Una animación es entonces una secuencia de *keyframes*, donde cada *keyframe* contiene la transformación de los huesos en ese tiempo.

Al proceso de asociar la geometría y los huesos se denomina *skinning*. La asociación misma se llama *skin*. La *skin* asocia cada vértice del modelo a un conjunto de huesos, donde cada hueso tiene un factor de peso sobre el vértice. Además de las transformaciones en los *keyframe*, cada hueso tiene una transformación que lleva los vértices de su posición original a la *bind pose*, esto es la posición en la que se basan las transformaciones de los *keyframes*. La transformación final aplicada a un vértice es la suma ponderada de las transformaciones de los huesos asociados, donde la transformación de un hueso es la transformación a la *bind pose* compuesta con la transformación del tiempo actual en la animación.

Las transformaciones de los *keyframes* se interpolan linealmente para obtener la transformación en un tiempo dado. Usar un algoritmo de interpolación mejor que lineal no hace diferencia a la vista dado que los tiempos entre *keyframes* son bastante cortos.

Para transformar los vértices eficientemente, se calculan las transformaciones de cada hueso en la CPU y luego se transfieren a la GPU para que los vértices se transformen usando un *Vertex Shader*. El *Vertex Shader* recibe el conjunto de huesos asociados a un vértice como un atributo del mismo, luego pondera las transformaciones de los huesos y finalmente transforma el vértice con la suma ponderada de las transformaciones. Hacer estas transformaciones en la GPU es eficiente tanto en procesamiento como en espacio: la GPU es mucho más eficiente que la CPU para este tipo de cálculos, además al realizar la transformación en el pipeline de la GPU, se evita tener una copia del modelo donde guardar los vértices antes de transferirlos a la GPU, y por último se evita tener que transferir los vértices a la GPU.

4.1.3 Materiales y efectos

Los efectos describen el proceso ejecutado para colorear, mientras que los materiales establecen valores en parámetros de un efecto que tienen asociado. Los parámetros de un efecto pueden ser:

- Vectores: usados para pasar las posiciones de las luces.
- Colores: usados para colores de brillos en los personajes y de luces.
- Matrices: transformaciones como las usadas para la animación.
- Texturas: arrays de hasta tres dimensiones de colores. Se usan para definir los colores sobre una geometría o para agregar detalle en el relieve de una geometría.
- Samplers: son las funciones usadas para acceder a las texturas. Especifica si se debe interpolar puntos no enteros al acceder a las texturas, y si se debe aplicar alguna transformación a las coordenadas antes de acceder a la textura.

Los efectos se definen en *CgFX*(76), un lenguaje creado por *nVidia*(77) para este propósito. *CgFX* permite especificar varias técnicas, donde cada técnica establece una forma de configurar la GPU (seleccionar *Shader*, texturas, etc.) para lograr el efecto deseado. El Renderer de Trifulca se puede configurar para que use la técnica de mayor detalle que la máquina actual soporta o para que siempre use la técnica de menor detalle de forma de lograr mayor rendimiento. Todos los efectos de Trifulca tienen una técnica básica, que logra el efecto usando solo cualidades encontradas en tarjetas de video de generaciones no

programables. Algunos de los efectos también poseen una técnica que utiliza *Shaders* para lograr un efecto más llamativo a la vista.

4.1.4 Interpolación

Como fue mencionado en la sección sobre el bucle principal, las posiciones de las entidades dibujadas siempre son una interpolación de las posiciones de dos *frames* lógicos. En caso de ser el cliente en un juego por red, las diferencias de tiempo entre *frames* lógicos conocidos pueden ser bastante grandes, por esto una interpolación lineal de las matrices de transformaciones no tiene buenos resultados visuales. Se decidió interpolar por separado la rotación y la traslación. Para la traslación se usa interpolación lineal, mientras que para la rotación se usa una interpolación lineal esférica (*Slerp*, por su sigla en inglés) (78). *Slerp* interpreta las rotaciones como puntos sobre una esfera y luego interpola por el camino más corto entre estos puntos a velocidad angular constante.

4.1.5 Formatos de datos

Se utilizó *COLLADA* (79) como formato de datos de donde se cargan las geometrías, animaciones, materiales y efectos para los gráficos. *COLLADA* es un formato estándar que soporta todos los tipos de datos mencionados y cuenta con soporte en varios programas de modelado 3D muy usados como *3D Studio Max*, *Maya* y *XSI*.

4.2 Interfaz gráfica de usuario

Se implementó un pequeño subsistema de interfaz gráfica de usuario para mostrar las opciones de inicio de juego y configuración. Este subsistema utiliza la funcionalidad de dibujar *Overlays* del subsistema de gráficos para mostrar listas de opciones e implementa navegación por mouse, teclado y joystick. Por cada opción se guarda un puntero a la función que se debe llamar cuando es seleccionada. A su vez las listas de opciones están organizadas como un árbol donde el menú principal es la raíz y cuando se elige una opción se muestra la lista de opciones de esa rama. En cada nodo se permite volver al nodo padre y en el nodo raíz se permite volver al juego en curso.

4.3 Cámara

Como se mencionó anteriormente, la cámara de Trifulca parte la pantalla dinámicamente dependiendo de la posición de los jugadores. Para lograr esto utiliza diagramas de Voronoi. En esta sección se describe cómo se llega de la posición de los jugadores a un conjunto de vistas del mundo y a una partición de la pantalla.

Se entiende por vista al punto de origen y a la orientación desde donde se renderiza el mundo. Normalmente se le llamaría cámara a esto, pero se decidió llamarle cámara al conjunto de vistas, y al algoritmo para obtener las vistas.

Dada la posición actual de los personajes en pantalla, el algoritmo de la cámara encuentra, de forma determinista, un conjunto de vistas. Este conjunto de vistas no es el que se usa en ese instante para renderizar, sino que se toma como objetivo a alcanzar a partir de la última vista usada para renderizar.

Entonces, en cada instante de tiempo se tiene un conjunto de vistas actuales (las usadas para renderizar) y otro conjunto de vistas objetivo. Cuando se avanza un paso de tiempo, se modifica la vista actual para que se acerque a la vista objetivo.

El algoritmo para obtener las vistas objetivo es el siguiente:

1. Se calcula el diagrama de Voronoi para las posiciones de los personajes en el plano donde se mueven.
2. Se agrupan los personajes de forma que si la distancia entre dos personajes es menor que una distancia configurable, entonces están en el mismo grupo.
3. Para cada grupo de personajes del paso 2 se fusionan sus regiones de Voronoi en una sola región. Estás regiones se escalan al tamaño de la pantalla para obtener la partición.
4. Cada región del paso 3 se escala al mínimo posible con la restricción de que sea visible un radio configurable alrededor de cada personaje.
5. Para cada región del paso 4 se obtiene una vista que se ajusta al rectángulo que limita la región. Este conjunto de vistas son las vistas objetivo.

Para encontrar el diagrama de Voronoi se intentó implementar el algoritmo de Voronoi de Fortune (80)(81)(82), pero la implementación lograda era poco robusta, no se obtenían resultados aceptables para muchos casos límite como puntos alineados o regiones infinitas. Se decidió implementar un algoritmo más simple: para cada punto se intersecan los semiplanos limitados por la mediatriz entre ese punto y cada uno de los otros puntos. Este algoritmo es bastante más ineficiente que Fortune, es $\mathrm{O}(\mathrm{n}^3)$ en lugar de $\mathrm{O}(\mathrm{n}\log\mathrm{n})$ (donde n es la cantidad de puntos). Como el algoritmo se usa para los personajes de los jugadores que están jugando en una misma computadora (la cámara no sigue a los que están jugando en otras computadoras por red), n no llega a ser suficientemente grande para que importe.

4.4 Lógica

En Trifulca se decidió usar *Lua* como lenguaje de scripting. *Lua* es el lenguaje de mayor adopción en el mundo de los videojuegos por su poco peso con respecto al tamaño final de la librería dentro del ejecutable. Además es más eficiente que la mayoría de sus semejantes dinámicos y contiene un *garbage collector* incremental desde la versión 5.1. Es un lenguaje fácil de usar por ser débilmente tipado, y simple para los requerimientos del motor.

El subsistema de lógica del juego se ocupa de correr todos los *scripts* asociados con cada uno de los participantes de la partida. Previo a cada corrida de la lógica ocurre una corrida del simulador de física y colisiones para que mueva los objetos que no son controlados por la lógica y además obtener las colisiones que ocurrieron debido a esta corrida. En cada paso lógico este subsistema llama a cada uno de los objetos que contienen un *script* que ejecutar pasándoles el conjunto de colisiones que les corresponde, el tiempo de duración al paso lógico y una representación de la entidad. El orden en que se ejecutan los *scripts* de las entidades no está predefinido y no se puede depender del mismo.

4.4.1 Luchadores

Los luchadores en Trifulca están representados con una cápsula como cuerpo físico. Esta representación es muy usada para *character controllers* debido a que su forma regular permite movimientos suaves de respuesta de colisiones. La cápsula está asociada al torso de la entidad en el esqueleto, tiene una altura que va desde los pies hasta cubrir la cabeza, y un diámetro que incluye todo el cuerpo. Esta cápsula es la única que colisiona con las componentes estáticas del mapa para permitir esta suavidad en los movimientos. En una primera instancia se probó usar un *convex hull* de cada grupo de puntos de cada hueso del esqueleto para así tener una representación física más fiel a la gráfica. Esta representación probó ser muy inestable con el movimiento causado por las animaciones en cada hueso, haciendo que se vean tembleques cuando un luchador se posaba o apoyaba sobre otro cuerpo. Para poder crear los golpes se asociaron capsulas sobre las extremidades (Figura 25) que tienen un diámetro y un largo algo mayor que las extremidades en sí. Estas extremidades exageradas se obtuvieron por ensayo y error para que los golpes se "sientan" bien. Las extremidades no colisionan con otros cuerpos dinámicos, y las colisiones con los cuerpos cinemáticos (otros luchadores) se manejan en la lógica de golpe con la reacción correspondiente.

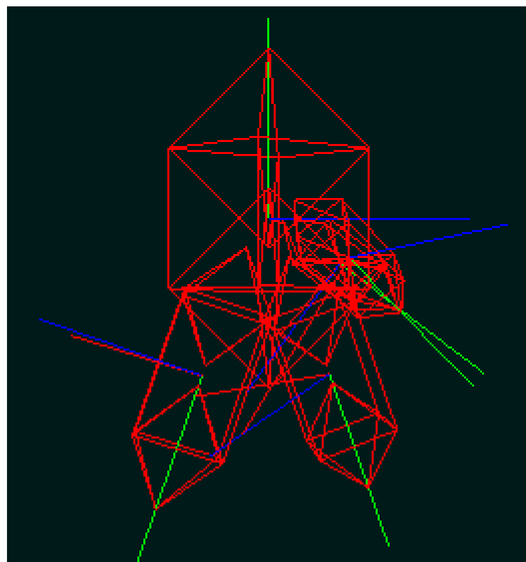

Figura 25 Representación física de un luchador en Trifulca

El movimiento de las entidades está dado por los comandos que se reciben. Se mantiene una velocidad en cada eje de movimiento y los comandos agregan aceleraciones a estas velocidades. Se mantiene una aceleración constante hacia abajo que proporciona la gravedad del juego. En caso de un salto se agrega un pequeño impulso en el eje vertical con dirección arriba y se deja que la gravedad proporcione el movimiento proyectil.

Para identificar si una entidad se puede considerar apoyada sobre el piso, es decir, sobre una superficie no muy inclinada, se mueve continuamente la cápsula principal de la entidad una distancia muy pequeña hacia abajo y se verifica si colisiona con alguna superficie.

Se usan los algoritmos implementados de *collide and slide* y *collide and bounce* para los movimientos de los luchadores. Con el movimiento normal de un luchador cuando está en un estado que no es golpeado se usa *collide and slide*. Este algoritmo permite, junto con la representación de la cápsula, que la entidad pueda subir escaleras y además, junto con la gravedad, que sea costoso subir colinas y se vaya más rápido al bajarlas. También se siente más realista al colisionar con otras entidades debido a la manera en la que resbalan una con otra sin "repelerse". Cuando un luchador es golpeado se usa *collide and bounce* para hacer que rebote contra el escenario. Como los luchadores al recibir daño son cada vez desplazados más lejos, entonces rebotan más veces y a mayor velocidad.

Para representar la energía en Trifulca se usa un valor porcentual que determina la velocidad a la que es desplazado. Al recibir un golpe aumenta este valor implicando que cada vez se moverá más lejos. Al caerse del mundo, determinado por el límite inferior, se le restará una vida y perderá cuando pierda las 3 vidas asignadas al principio del juego.

4.4.2 Mapa

El mapa está representado por un conjunto de entidades, donde cada una tiene un cuerpo físico convexo. El juego soporta cargar cuerpos físicos cóncavos, los cuales son partidos en sub cuerpos convexos en tiempo de ejecución. Pero esto no resulta en buena performance, dado que el algoritmo de división en cuerpos convexos puede generar muchos más cuerpos de los necesarios, lo cual impacta en la velocidad de la resolución de colisiones. Existen algunos componentes de los mapas que son dinámicos y otros estáticos. Las entidades dinámicas están completamente controladas por la física y no tienen script asociados, e interactúan con las entidades cinemáticas cuando estas las golpean.

4.5 Red

A continuación se describirán detalles del soporte de red vistos desde el punto de vista de cada capa del diseño.

4.5.1 Capa de plataforma

Esta capa es equivalente a la capa de sistema del *Quake 3*(83) *y* la capa de plataforma de *TNL*(84). Brinda una interfaz mínima para encapsular un transporte *mejor-esfuerzo*. Requiere una implementación para cada sistema operativo ya que cada uno tiene pequeñas diferencias.

Para simplificar la inicialización y el manejo de recursos para toda la comunicación se utiliza un único socket UDP, el cual además es no bloqueante, ya que si lo fuera sería necesario levantar un hilo dedicado a recibir y encolar paquetes, que lo único que logra es agregar complejidad a la implementación. Así entonces la llegada de paquetes se verifica asincrónicamente una vez por cada paso lógico en el mismo hilo de ejecución que el bucle principal del juego. La ventaja de usar varios puertos radicaría en tener una cola de paquetes por cada cliente, pero sólo es útil en caso de que se esperen desbordamientos.

El juego se implementó sobre UDP ya que se requerían respuestas rápidas que TCP no puede brindar. Lo ideal hubiera sido usar DCCP, dado que tiene las mismas facilidades que UDP en lo que respecta a

tiempos de respuesta inmediatos y verificaciones de integridad, y además agrega control de congestión. Pero existen muy pocas implementaciones de este protocolo, recién en el *kernel* de *Linux* versión 2.6.14 es que se encuentra finalizada en un sistema operativo de uso doméstico, existiendo algunos parches para versiones de 2.4, y una implementación preliminar en *FreeBSD*(85).

Esta capa no busca proveer un transporte confiable, en caso de ser necesario el mismo debería ser implementado en una capa superior.

4.5.2 Capa de transporte

La capa de transporte provee un servicio orientado a conexión no confiable, sin garantía de orden, de comunicación bidireccional, que puede ser extendida para brindar confiabilidad. Está orientado a paquetes y provee mecanismos para evitar conexiones medio abiertas y ataques simples mediante re-sincronizaciones. Esta implementación es una versión simplificada de lo propuesto por DCCP.

Esta capa brinda además un control de flujo simple mediante el establecimiento de un máximo de ancho de banda para ambos puntos de la conexión.

Conexión.

Para una partida de un juego es necesario intercambiar características específicas para su inicialización (como lo son mapa, tipo de juego, etc.) y además mantener cierto estado asociado a la comunicación entre cada uno de los clientes y el servidor. Este estado es creado y mantenido por la conexión.

Una conexión se establece activamente por el cliente, siendo necesario que el servidor se encuentre esperando pasivamente conexión. Esto se produce sobre un protocolo no confiable por lo que se usó un mecanismo de negociación basado en números de secuencia muy similar al definido en DCCP. El mismo intenta prevenir conexiones erróneas en las que un *endpoint* quede en un estado apto para el intercambio de datos y el otro no.

La conexión está definida por un par de identificadores formados por una dirección IP y un puerto UDP. Para que un cliente se conecte a un servidor, el mismo necesita conocer el par que lo identifica, por esta razón al servir un juego es necesario que el servidor defina el puerto UDP que va a usar y lo dé a conocer. Al cliente le es asignado uno elegido por el sistema operativo. Para ayudar al establecimiento de una conexión se provee de un servicio de búsqueda en red local de juegos. Este servicio no puede ser extendido a Internet sin la necesidad de un intermediario dedicado.

Una vez establecida la conexión, el protocolo usa mecanismos para mantenerla viva ante inactividad causada por razones como la espera de carga de recursos por parte del servidor o de un cliente. Es importante la inclusión de estos mecanismos para encontrar lo antes posible una conexión medio abierta, debido a una caída en alguno de los *endpoints* o alguna otra falla y poder, por ejemplo, avisar en cada cliente de la desconexión del servidor.

Es común que en conexiones a través de Internet ocurran ráfagas de pérdida de paquetes causadas por diversos motivos, como el congestionamiento en algún *router* perteneciente al camino. En conexiones con secuencias estas ráfagas pueden llevar a que queden en un estado en el cual los *endpoints* no

pueden identificar paquetes correspondientes al enlace actual. Estos estados se caracterizan por desfasajes entre lo que conoce un *endpoint* del estado del otro y el verdadero estado. Para evitar esto se implementó un mecanismo de sincronización que permite que los *endpoints* de la conexión reconozcan desfasajes e intercambien la información de estado.

Para el caso particular de este juego, no tiene sentido que un *endpoint* siga conectado mientras el otro sigue enviando datos, por lo que se usó un mecanismo de desconexión simétrico.

Los estados de una conexión, así como el pasaje entre ellos y los tipos de paquetes usados se detallan en el Apéndice A.

Mensaje.

La capa de red ofrece una estructura de datos a capas superiores que permite la escritura de un mensaje en forma secuencial. Los métodos que se proveen para escribir sobre un mensaje se aseguran que los datos estén alineados y ocupen el menor espacio posible. Para facilitar la implementación además se permite de manera limitada escrituras fuera de orden y una implementación simple de transacciones similares a las de bases de datos. Se profundiza sobre las funcionalidades de un mensaje en el Apéndice A.

4.5.3 Capa de sincronización

El propósito principal de esta capa es actualizar el estado del juego, representado por las entidades, desde el servidor al cliente y también publicar los comandos obtenidos en el cliente hacia el servidor.

En esta capa se define que es lo que se transmite de las diferentes entidades del juego así como también el momento en el tiempo en que se generarán y replicarán estas actualizaciones. Además se implementan las técnicas de tabla de strings y compresión delta que se detallan en el Apéndice A.

Sincronización del juego.

El hecho que el servidor sea el único árbitro del juego lleva a que el mundo que maneja es el que contiene los datos reales del juego. Al mismo tiempo, cada cliente necesita tener uno local para que los diferentes subsistemas puedan interactuar con él de manera de presentarlo al usuario. Esta versión local por lo general es un estado antiguo del servidor. Fue necesario entonces definir mecanismos para que durante el curso de un juego los grafos de los diferentes clientes y el servidor se mantengan en sincronía tanto como fuese posible, de manera que el usuario no perciba las diferencias que tiene su estado local con lo que es el verdadero estado del juego.

La actualización del grafo de un cliente puede lograrse de diversas maneras. Dos opciones comunes son:

- Enviar las actualizaciones en sí, o sea, instrucciones de cambio predefinidas, como lo son “saltar” y “disparar”, las cuales provocarían un estado esperado en el cliente. Básicamente sería replicar los comandos recibidos por el servidor para que se ejecuten en el mismo orden en que son enviados por el cliente.

- Enviar el estado actual a cada cliente para que sobre-escriba al que tiene en el momento de la llegada.

La primera alternativa puede llegar a ser muy eficiente con respecto al consumo de recursos, ya que en vez de enviar el estado nuevo correspondiente a varias entidades se puede enviar un comando simple que produzca el mismo efecto. La gran desventaja que tiene este tipo de actualización es que es muy poco robusta respecto a pérdidas. El estado de un juego es el resultado de la ejecución de diversos comandos en un orden particular. Con la pérdida de uno solo de estos comandos el resultado puede ser totalmente diferente del esperado por el servidor, y no se podría implementar de esta manera a menos que se tenga un mecanismo de recuperación. Este mecanismo debería tener asociado el envío de estados absolutos, lo que requeriría implementar a su vez el segundo método mencionado. Otra opción sería el uso de confiabilidad en el envío de estos comandos, que lleva a un incremento de las latencias en conexiones por Internet como fue mencionado antes. Por estas razones se decidió implementar la segunda técnica.

Actualización de estados.

Para que los clientes muestren una representación del mundo, cada subsistema requiere que se le actualicen ciertos datos de la entidad. Por ejemplo, la cámara requiere la posición de la entidad actualizada y el *Renderer* además de la posición requiere la animación actual de la entidad. El conjunto de estos datos para todas las entidades se denomina estado y se profundiza sobre su estructura en el Apéndice A.

Para actualizar los estados es necesario que por lo menos la red conozca cuales son los valores relevantes para el resto o que el resto de los subsistemas definan cuales les interesa replicar hacia los clientes. Esto implica que exista un acoplamiento entre el subsistema de red y el resto de los subsistemas, es necesario que el subsistema de red conozca que datos debe enviar de cada subsistema o que cada subsistema le pase esos datos a la red. Se decidió implementar una responsabilidad compartida para dar solución a este problema. De esta manera existe un conjunto de atributos que la red sabe que debe enviar para el funcionamiento de todos los subsistemas y además aquellos que sean altamente modificables, como por ejemplo la lógica a través de scripts, pueden definir atributos extra e indicarle al subsistema de red si son replicables o no.

Los estados en Trifulca deben estar contenidos en un único mensaje y no se diseñó ningún método de fragmentación. Proveer fragmentación en esta capa o la inferior provocaría que se descarten muchos paquetes por la pérdida de uno sólo. Por ejemplo, si consideramos una actualización de estado cuyo tamaño ocupa más que el *payload* de un paquete entonces se requiere el envío de más mensajes para esta actualización. Si se pierde uno de los paquetes, es necesario descartar el resto que compone el estado, a menos que se implemente algún sistema de recuperación y reenvío. En vez de implementar un sistema de fragmentación se juntaron los estados de las entidades que quepan y se guardó cuales fueron enviadas de manera de poder enviar el resto en actualizaciones posteriores. Esta decisión tiene como ventaja que la pérdida de un mensaje no lleva a la pérdida de la actualización de todo el mundo sino que solamente de un grupo reducido de entidades. Como consecuencia, ayuda a que las entidades se mantengan lo más actualizadas posibles, evitando que crezca la diferencia entre lo que conoce el

cliente y el estado del juego, que podría disminuir la eficacia de la compresión delta. La desventaja de este método es que en el cliente cada entidad no necesariamente está en sincronía con el resto, ya que puede haber entidades que estén más actualizadas que otras.

El estado del juego varía únicamente con cada paso lógico, por lo que tiene un tiempo de generación asociado. Un cliente necesita este tiempo para poder mostrar dos estados con una diferencia de tiempo aproximada a la de generación. Si se muestran cuando se reciben, pueden ocurrir aceleraciones en los movimientos causados por las condiciones de la red. Por esta razón cada estado incluye su tiempo de generación.

Las actualizaciones de estado no necesariamente se deben realizar al momento que se generan. La precisión del paso lógico puede ser mayor que la necesaria para obtener una calidad aceptable, haciendo posible el salteo de estados para lograr disminuir el consumo de recursos. En Trifulca se puede modificar la cantidad de estados por segundo (SPS, *states per second*) que son enviados por default, para poder regular la cantidad de estados que envía el servidor. Esta propiedad junto con el control de ancho de banda que provee la capa inferior permite controlar la subida del servidor de manera de no superar las limitaciones impuestas y así no incurrir en retardos innecesarios.

Como las actualizaciones a los clientes se envían en intervalos mayores a los generados por la lógica, la interpolación generada por el *Renderer* evita que los movimientos de las entidades se vean cortados cuando los estados que dibuja son producto de una actualización por red. La interpolación necesita la generación de un estado nuevo para poder recién dibujar el estado que se generó en el último paso lógico. Como en el servidor los pasos lógicos tienen un tiempo definido y no es posible que se pierdan estados, entonces se puede lograr una interpolación continua en la cual no se vean saltos abruptos. Pero en el cliente no se tiene certeza que la diferencia de tiempo entre el último estado recibido y el próximo sea un valor fijo. Para manejar este problema se decidió que el cliente se sitúe atrás en el tiempo con respecto al servidor, técnica que es usada en el *Quake 3*(83) y en el motor *Source*(32). Este atraso permite que pérdidas de estado no lleven a que el juego quede parado por no tener qué dibujar.

Comandos.

Los comandos en Trifulca están compuestos por las acciones de cada jugador y deben ser ejecutadas en el servidor por parte de la lógica. Fue necesario definir el momento de ejecución de los comandos con respecto a su llegada. Las alternativas consideradas impactan en cómo la lógica debe manejar las entradas. Estas alternativas eran:

- Ejecutar los comandos a medida que llegan
- Asociarlos al *frame* de ejecución en el cual se generó, y ejecutarlos en ese *frame*.

Asociar cada comando a un *frame* implica que el servidor antes de ejecutar un paso lógico debe esperar a que lleguen los comandos de cada cliente hasta un cierto límite de tiempo para intentar no perder ningún comando. Esto provoca que el retraso del servidor con respecto a la ejecución del paso sea igual al retraso del cliente con peor conexión, o del límite de espera impuesto en el servidor. Además, el

límite de espera es la latencia máxima que acepta de las conexiones ya que no ejecutará nunca los comandos de un cliente que demoren más que ese tiempo en llegar al servidor.

La otra alternativa es mucho más simple, los comandos son ejecutados en el *frame* que llegan, sin importar el *frame* del cliente en que fueron generados. El impacto que tiene esta opción sobre el resultado de la ejecución del comando, comparado con lo esperado por el jugador, es mucho mayor que la otra alternativa. Un ejemplo de esto es cuando se produce un cambio en las condiciones de la red, que tiene como consecuencia que se reduzcan las latencias. En el momento del cambio en el servidor llegan más comandos de los esperados para ejecutar en el mismo *frame*, si decide ejecutar el último para brindar mejores respuestas al cliente entonces se pierde un comando intermedio. Para evitar esto, se pueden ejecutar todos los comandos que llegaron, pero la lógica ejecutaría los comandos sin ejecutar nunca un paso de la simulación física. Como en este desarrollo el motor de física considera los últimos estados de los cuerpos cinemáticos para sus estimaciones, si se ejecutan dos o más comandos que mantienen un movimiento constante entre dos pasos de simulación física, el motor puede estimar aceleraciones de mayor valor que la real.

En el caso de Trifulca se decidió usar la segunda técnica debido a que es más dinámica para la cantidad de jugadores que se espera soportar. Con una cantidad considerable de clientes conectados, las demoras de cada jugador provocarían un atraso general del juego haciendo que el mismo se vea lento.

Para poder soportar pérdidas sin implementar garantía, se repite el envío de cada comando. Esta opción se consideró porque es importante que un comando llegue al servidor ya que la pérdida del mismo no se puede interpolar como los estados. Además el reenvío de los comandos requiere muy poco ancho de banda y genera confirmaciones de los estados, que, como se vio anteriormente, ayuda a técnicas de ahorro de ancho de banda como compresión delta y tablas de strings.

Jugadores.

Como la replicación de entidades es una comunicación de servidor a cliente, fue necesario establecer un método para que el cliente le indique al servidor que debe agregar una entidad que la lógica debe manejar con ciertos comandos. Para esto se ideó un mensaje de pedido de jugadores que es enviado desde el cliente al servidor. Un mensaje puede contener varios pedidos de jugadores. Cada jugador corresponde con una entidad con la URI del recurso que la representa y el identificador de comandos que usará el cliente localmente. Cuando un servidor recibe este tipo de mensajes por parte de un cliente debe crear la entidad correspondiente avisando a la capa superior para indicar el hecho y dar lugar a un rechazo. En caso de que la entidad ya fuera creada se descarta el pedido. El servidor adjunta al comienzo de un mensaje de estado la confirmación de las entidades creadas junto con sus identificadores de manera que el cliente deje de enviar los pedidos de creación.

4.5.4 Capa de juego

La capa de juego determina como se deben crear las entidades de acuerdo a los diferentes pedidos de creación, y qué hacer con ellas cuando se reciben a través de la red. Es la encargada de asignar los identificadores globales de las entidades de manera de poder distinguirlas a través de la red. Determina como deben ser las reglas de juego, es decir, quien gana y cuando lo hace.

Creación e identificación de entidades.

Las entidades son creadas en esta capa únicamente del lado del servidor. La capa de sincronización solamente envía las entidades que ya están creadas, y en el cliente avisa si se creó o destruyó una entidad.

Los identificadores de entidades son únicos ya que son usados por la red para reconocerlas y actualizarlas. Se decidió darle la responsabilidad a esta capa para la asignación de identificadores, de manera de poder usarlos para los diferentes tipos de entidades que manejará el juego, de esta manera se puede separar el conocimiento del tipo de entidad de la sincronización de las mismas.

Mapas.

Los mapas del juego están formados por varias entidades con diferentes comportamientos. Es necesario poder conocer si estas entidades se deben actualizar continuamente, debido a que son dinámicas y su estado varía con el tiempo, o sólo se deben crear y nunca más ser modificadas por ser estáticas. Para ahorrar en el envío entidades que forman un mapa, el servidor envía únicamente la entidad padre del mapa. Cuando en el cliente se recibe esta entidad, carga y numera todas las restantes del mapa exactamente igual que en el servidor.

Luchadores.

Los luchadores son creados a partir del pedido de creación del cliente al agregarse un jugador. La lógica le asigna a la entidad un identificador del rango correspondiente a los luchadores, y un identificador de entrada de usuario global en el servidor. Este identificador se usará para convertir los comandos que llegan de la red a entradas de usuario, de la misma manera en que se hace con los eventos de periféricos (teclados, dispositivos de juego, etc.).

5 Pruebas

5.1 Pruebas de jugabilidad

No se realizó ninguna prueba formal sobre aspectos de jugabilidad y lo entretenido del juego. Pero sí se realizaron varias partidas con cinco personas, todas con experiencia en videojuegos.

Las partidas se desarrollaron de las siguientes formas:

- Sobre Internet uno contra uno.
- Dos máquinas en red local de a cuatro, dos en cada máquina.
- Tres jugadores en la misma máquina, sin juego por red.

El consenso general es que es el juego es divertido, los controles son buenos, pero le falta pulido y contenido.

En los juegos por Internet la queja principal era la demora entre un comando y su ejecución, que se arreglaría con predicción en el cliente.

Uno de los participantes en las pruebas expresó gran alegría por ganar varias partidas contra los desarrolladores, resultado que consideramos muy positivo.

5.2 Evaluación de consumo de ancho de banda

Se evaluó el consumo de ancho de banda del juego en situaciones de juegos reales y la eficacia de la compresión delta y de las tablas de strings.

Para el ancho de banda usando las técnicas se tomaron las siguientes medidas:

- Consumo subida del servidor cuando varían los jugadores.
- Consumo de subida de un cliente cuando varían los jugadores conectados desde ese cliente.
- Consumo de subida del servidor cuando varía la cantidad de clientes.

Todas las medidas se tomaron en redes locales con conexión inalámbrica. Se usó el mapa diseñado que tiene 13 cajas que son entidades dinámicas, más los luchadores. Las partidas duraron entre uno y cinco minutos.

La cantidad de estados por segundo que se envía (SPS) es 15, valor común para el juego. Los comandos por segundo (CPS) 120 que representan el máximo.

Todas las mediciones se tomaron en el mismo código del juego sin herramientas externas.

5.2.1 Subida del servidor variando jugadores

En este caso se probó un servidor con un solo cliente conectado, el servidor tenía un jugador y el cliente varió entre uno, dos y tres jugadores. Se midió la subida del servidor.

Cantidad de jugadores en el cliente	Consumo promedio (B/s)	Consumo mínimo (B/s)	Consumo máximo (B/s)	Desviación Estándar (B/s)
1	1840.822	770	3372	497.0709
2	1840.689	934	3050	515.7446
3	2367.077	846	3252	522.3175

Tabla 1 Subida del servidor variando la cantidad de jugadores.

Como la experiencia se realizó en una partida real, se puede ver en la Tabla 1 que los datos son muy similares a pesar de la variación en la cantidad de jugadores. El crecimiento parece ser muy bajo en comparación con el consumo básico y la desviación estándar. Esta última puede ser causada por la compresión delta, ya que al probarse en una situación real, los luchadores no pasaron todo el tiempo moviéndose al igual que las cajas.

Se puede ver que el máximo de consumo no supera los 128Kb de subida de las conexiones encontradas en Uruguay.

5.2.2 Subida de un cliente variando jugadores

El caso de prueba es el mismo que para el punto anterior. Se midió la subida del cliente.

Cantidad de jugadores en el cliente	Consumo promedio (B/s)	Consumo mínimo (B/s)	Consumo máximo (B/s)	Desviación Estándar (B/s)
1	2514.328	2500	2520	9.031593
2	3160.417	2625	4025	240.0178
3	3768.529	3360	4020	144.1109

Tabla 2 Subida de un cliente cuando varía la cantidad de jugadores.

Como se puede ver en la Tabla 2 la variación en los comandos es mucho menor que en los estados, porque no se aplica ningún tipo de técnica en el envío. El crecimiento se nota, no entra en la desviación estándar, pero igual es bajo en comparación con el consumo base.

Al igual que con los estados los valores se encuentran entro los vistos por las conexiones uruguayas.

5.2.3 Subida de un servidor variando clientes

En este caso se intenta aumentar la cantidad de clientes manteniendo la cantidad de luchadores en cuatro. Se midió la subida del servidor.

Cantidad de clientes conectados al servidor	Consumo promedio (B/s)	Consumo mínimo (B/s)	Consumo máximo (B/s)	Desviación Estándar (B/s)
1	1840.822	770	3372	497.0709
2	4668.168	2616	5927	763.1497
3	5712.433	4171	8650	846.9091

Tabla 3 Subida de un servidor con varios clientes.

En la Tabla 3 es notorio el crecimiento de la subida con la cantidad de clientes, esto es porque el servidor envía el mismo estado a cada cliente.

5.2.4 Técnicas

Compresión delta.

Para verificar la eficacia de la compresión delta, se tomaron mediciones de un juego con un cliente y un servidor con un luchador, cada uno primero con esta técnica habilitada y luego deshabilitada. Se midió la subida del servidor.

	Consumo promedio (B/s)	Consumo mínimo (B/s)	Consumo máximo (B/s)	Desviación Estándar (B/s)
Con compresión delta	1840.822	770	3372	497.0709
Sin compresión delta	8516.609	7931	9018	210.8307

Tabla 4 Comparación de la subida de un servidor con y sin compresión delta.

La Tabla 4 muestra la ganancia del uso de compresión delta. En este caso todos los consumos aumentaron cuando se deshabilitó esta funcionalidad. Se puede verificar también que parte de la desviación estándar de la Tabla 3 corresponde a la compresión delta.

Tabla de strings.

Estas medidas se tomaron en dos juegos con un cliente y un servidor con un luchador cada uno. Uno de los juegos usa tabla de *strings* y otro no. En este caso se desactivó la compresión delta en ambos casos para que su alta desviación no interfiera en los resultados.

	Consumo promedio (B/s)	Consumo mínimo (B/s)	Consumo máximo (B/s)	Desviación Estándar (B/s)
Con tabla de strings	8516.609	7931	9018	210.8307
Sin tabla de strings	12745.02	11550	13491	367.5106

Tabla 5 Comparación de la subida de un servidor con y sin tabla de strings.

Según los resultados de la Tabla 5 el consumo sin usar tabla de *strings* es muy grande, pero hay que notar que este juego en particular usa *strings* muy largos para cualquier animación.

5.2.5 Conclusión

El único caso en que se superó el límite de conexión de 128Kb como valor máximo fue con las dos técnicas desactivadas, en estas condiciones el juego puede correr en conexiones uruguayas. El mayor crecimiento se nota en el aumento de la cantidad de clientes conectados a un servidor, pero esto era de esperarse con la arquitectura cliente-servidor.

6 Conclusiones y trabajo a futuro

Se cumplieron casi todos los objetivos planteados en cuanto a las funcionalidades mínimas requeridas, obteniéndose un prototipo avanzado del juego. Esto es:

- Se implementó un motor gráfico que provee las funcionalidades básicas para dibujar objetos 3D, animaciones esqueléticas, y materiales usando *Shaders*.
- Se desarrolló un sistema de comunicación sobre red que permite que varios usuarios interactúen estando en diferentes computadoras. Esta implementación tiene un bajo consumo de ancho de banda que permite jugar usando conexiones de Internet hogareñas.
- Se logró un producto fácilmente modificable por medio de scripts de lógica en un lenguaje dinámico de alto nivel, y contenido artístico en formatos estándares que son *COLLADA* y PNG.
- Como producto final se obtuvo un juego entretenido.

Para llegar a los objetivos planteados fue necesario desarrollar además las siguientes funcionalidades:

- Un manejador de recursos que permite cargarlos a demanda sin necesidad de recargar el juego. Esto fue necesario para poder acortar los tiempos entre las pruebas. Fue de vital importancia para el diseño y el ajuste de los valores usados por la lógica, los cuales requieren muchas iteraciones hasta lograr el efecto deseado.
- Una consola de comandos y de manejo de variables generales del juego. Esto facilitó probar funcionalidades del mismo, al poder invocarlas por medio de un llamado y permitió la estimación en tiempo de ejecución de diferentes parámetros del juego.
- Una interfaz de usuario, que brinda las funcionalidades básicas para inicializar partidas y modificar las configuraciones dentro del juego. Fue necesario su desarrollo para tener la posibilidad de que jueguen usuarios con pocos conocimientos computacionales.

Con respecto al diseño del juego hubiera sido una mejor alternativa usar componentes dinámicos(86) para las entidades, en vez de incluir lo necesario para cada módulo estáticamente en la clase que representa la entidad. Un diseño orientado a componentes hubiera significado una mayor separación de responsabilidades en la entidad, evitando que sea sobrecargada de datos no relacionados y logrando que pueda ser utilizada para más propósitos.

La elección de *C++* como lenguaje tuvo como ventaja la existencia de muchas más bibliotecas auxiliares generales y especializadas en juegos debido a su longeva existencia y gran aceptación en el medio. A pesar de esto se considera que podría haber sido una buena elección usar un lenguaje de más alto nivel como *C#* o *Python*. La razón por la que *C++* sigue siendo el lenguaje de elección en la industria reside en la ganancia en performance que se puede obtener gracias al mayor control del bajo nivel que brinda. Algunas de las funcionalidades provistas por los lenguajes a más alto nivel se pueden implementar en

C++ como bibliotecas. Un ejemplo de éstas, que fue usado durante todo el desarrollo de Trifulca, es *Boost*. *Boost* provee funcionalidades como manejo del ciclo de vida de la memoria, y punteros a funciones miembro que aumentan la productividad de la misma manera que los lenguajes mencionados. Sin embargo, el intrincado diseño de esta biblioteca puede llevar a que un simple error en su uso sea muy difícil de depurar.

La elección de *COLLADA* como formato de datos para los gráficos no resultó satisfactoria. Por más que cuenta con soporte en muchos programas de modelado, las implementaciones son malas, incompletas e incompatibles. Fue necesario implementar herramientas que preparan los archivos *COLLADA* exportados por un programa para importarlos en otro y en el juego. Debido a esto la creación de mapas para el juego resulta bastante trabajosa, es necesario utilizar dos programas comerciales más la herramienta de preparación para crear un nuevo mapa.

Dada la dificultad de crear nuevo contenido, queda como trabajo a futuro mejorar las herramientas, implementando nuevos exportadores específicos para Trifulca. Con mejores herramientas será más fácil crear nuevos escenarios y luchadores, los cuales son necesarios para tener un juego completo.

En los gráficos falta implementar varios efectos para mejorar la visualización del juego. Algunos que sería bueno implementar son sombras, partículas y oclusión de ambiente. Otro aspecto relacionado con gráficos y física que queda como trabajo a futuro es el mundo destructible. Las técnicas investigadas eran demasiado complejas y queda pendiente probar técnicas más simples.

Para lograr eliminar la principal consecuencia de las latencias de la red, sería importante implementar la técnica de predicción del lado del cliente. En conexiones con altas latencias, la respuesta del juego vista por el usuario no era instantánea. La implementación de esta técnica hubiese impactado en la lógica del juego, debido a que gran parte de los movimientos deben ser fácilmente predecibles, además de que se tendría que considerar la manera de corregir errores de predicción. Estos cambios requerían un rediseño de la lógica del juego que escapaba a los requerimientos.

Otra técnica de red que faltó implementar es la extrapolación. El resultado de ésta es menos importante que la de predicción del cliente porque no impacta en la respuesta del jugador sino en el resto de los objetos y solo se usa cuando se altera la periodicidad de las actualizaciones.

Por último, para poder armar partidas a través de internet sería conveniente un sistema que permita coordinarlas. Esto implicaría un servidor central en el que se registran jugadores que pueden conversar y proponerse jugar, sin la necesidad de manejar direcciones IP.

7 Glosario

- *API*: acrónimo de *application programming* interface. Es el conjunto de funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.
- *Collide and bounce*: método de respuesta de colisiones entre un objeto dinámico y uno estático que consiste en aplicar la energía remanente de la colisión para que el objeto dinámico rebote en la superficie del objeto estático.
- *Collide and slide*: método de respuesta de colisiones entre un objeto dinámico y uno estático que consiste en usar la energía remanente de la colisión para deslizar el objeto dinámico sobre la superficie del objeto estático.
- Congestión: En una red refiere al cuando un nodo recibe más paquetes de los que puede mandar.
- *Convex hull*: el menor cuerpo convexo que contiene a un conjunto de puntos.
- DCCP: acrónimo de *Datagram Congestion Control Protocol*. Protocolo de red de capa de transporte orientado a mensajes y diseñado para aplicaciones en tiempo real.
- *Endpoint*: es uno de los participantes en una conexión de capa de transporte.
- *Engine*: en videojuegos refiere a un sistema de software que ayuda en su creación y desarrollo. Puede incluir bibliotecas y herramientas de creación de contenido. En la mayoría de los casos abarca todo el desarrollo, pero existen engines especializados para gráficos, física y otras áreas.
- *Exponential backoff*: Método de cálculo de la tasa de reintentos de un evento en el cual el tiempo entre cada reintento se aumenta de manera multiplicativa.
- *Keyframe*: el valor de una función de animación en determinado instante de tiempo.
- *Lag*: abreviación de *loading*, en juegos refiere a retrasos por una demora en las comunicaciones
- *Multitexturing*: referido a tarjetas de video es la capacidad de poder renderizar una superficie con varias texturas combinadas sobre ésta.
- *NAT*: acrónimo de *Network Address Translation*. Proceso de traducción de direcciones de red entre una red local y una internet.
- *Payload*: es el contenido de datos que un paquete permite llevar.
- Renderizar: El proceso que genera una imagen a partir de la descripción de un modelo en datos de más alto nivel como geometría, texturas y *Shaders*.

- *Router*: computadora especializada en enrutar paquetes de red.
- *RTT*: Acrónimo de *Round Trip Time*, tiempo de ida y vuelta.
- *Shader*: programa que corre en una *GPU* para procesar vértices, píxeles o geometría.
- *TCP*: acrónimo de *Transmission Control Protocol*. Protocolo de red de alta confiabilidad.
- *UDP*: acrónimo de *User Datagram Protocol*. Protocolo *mejor esfuerzo* de capa de transporte.

8 Bibliografía

1. *Growth of gaming in 2007 far outpaces movies, music.* [En línea] http://arstechnica.com/news.ars/post/20080124-growth-of-gaming-in-2007-far-outpaces-movies-music.html.

2. Smash Smash Bros. [En línea] http://www.youtube.com/watch?v=EgJKLUTVPY4.

3. *Diccionario de la lengua española.* [En línea] http://buscon.rae.es/draeI/SrvltConsulta?TIPO_BUS=3&LEMA=juego.

4. Juegos Serios. [En línea] http://portal.educar.org/multimediamachine/blog/juegosseriosseriousgames.

5. *Game - Wikipedia.* [En línea] [Citado el: 8 de Febrero de 2009.] http://en.wikipedia.org/wiki/Game.

6. Video game - Wikipedia. [En línea] http://en.wikipedia.org/wiki/Video_game.

7. *A Data-Driven Game Object System.* **Bilas, Scott.** 2002.

8. The 3dfx history. [En línea] http://www.3dfx.ch/project/timeline/page_one_eng.htm.

9. *Comparison of ATI graphics processing units.* [En línea] http://en.wikipedia.org/wiki/Comparison_of_ATI_graphics_processing_units.

10. *Multitexture and the Quake 3 graphics engine.* [En línea] [Citado el: 18 de Abr de 2009.] http://www.bigpanda.com/trinity/article1.html.

11. *Comparison of Nvidia graphics processing units.* [En línea] http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units.

12. **Tariq, Sarah.** DirectX 10 Effects. [En línea] 2006. http://developer.download.nvidia.com/presentations/2006/siggraph/dx10-effects-siggraph-06.pdf.

13. OpenGL. [En línea] http://www.opengl.org/.

14. Direct3D. [En línea] http://msdn.microsoft.com/en-us/directx/default.aspx.

15. PC Graphics Shipments Q1 2007. [En línea] http://www.jonpeddie.com/about/press/2007/FLQ107.shtml.

16. nVidia PerfHUD. [En línea] http://developer.nvidia.com/object/nvperfhud_home.html.

17. gDEBugger. [En línea] http://www.gremedy.com/.

18. Matt Writes Blog. [En línea] http://dezignstuff.com/blog/?p=146.

19. *Real-time High Dynamic Range Texture Mapping.* **Jonathan Cohen, Chris Tchou, Tim Hawkins, Paul Debevec.** 2001.

20. *Simulation of Wrinkled Surfaces.* **Blinn, James F.** 1978.

21. *Finding Next Gen – CryEngine 2.* **Mittring, Martin.** 2007.

22. *Custom Portfolio - Krishnamurti M. Costa.* [En línea] [Citado el: 19 de Abr de 2009.] http://www.antropus.com/private/custom_portfolio/home.htm.

23. **Kwoon, Hun Yen.** The Theory of Stencil Shadow Volumes. [En línea] http://www.gamedev.net/reference/articles/article1873.asp.

24. *oZone3D.Net Tutorials - Stencil Shadow Volumes.* [En línea] [Citado el: 19 de Abr de 2009.] http://www.ozone3d.net/tutorials/stencil_shadow_volumes.php.

25. **Tomas Akenine-Möller, Eric Haines, Naty Hoffman.** Real-time Rendering.

26. *Rockstar Games: Grand Theft Auto IV.* [En línea] [Citado el: 19 de Abr de 2009.] http://www.rockstargames.com/IV/.

27. **Jenkins, Kevin.** Designing Secure, Flexible, and High Performance Game Network Architectures. *Gamasutra.* [En línea] 6 de Diciembre de 2004. http://www.gamasutra.com/view/feature/2174/designing_secure_flexible_and_.php.

28. 1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond. *Gamasutra.* [En línea] http://www.gamasutra.com/view/feature/3094/1500_archers_on_a_288_network_.php.

29. **Gray, Mike O'Brien & Gaile.** Game Cheats and Cheat Prevention. *ArenaNet.* [En línea] http://www.arena.net/articles/mikearticle040802.php.

30. *A Distributed Architecture for MMORPG.* **Tzanov, Marios Assiotis and Velin.**

31. *A Grid-enabled Multi-server Network Game Architecture.* **Tianqi Wang, Cho-Li Wang, Francis Lau.**

32. Source Multiplayer Networking. *The Valve Developer Community.* [En línea] http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking.

33. The Quake3 Networking Model. *Book of hook.* [En línea] http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/Quake3Networking#TheQuake3NetworkingModel.

34. **Mogul, J.** *Delta encoding in HTTP.* s.l. : Network Working Group, 2002. RFC3229.

35. Dynamic Data Exchange. *Microsoft Developer Network.* [En línea] http://msdn.microsoft.com/en-us/library/ms648711(VS.85).aspx.

36. Atoms. *Microsoft Developer Network.* [En línea] http://msdn.microsoft.com/en-us/library/ms648708(VS.85).aspx.

37. QuakeWorld. [En línea] http://www.quakeworld.net/.

38. **Raymond, Hal.** Half-Life 1.1.0.0. *HAQSAU'S LAB.* [En línea] http://rocketland.planetquake.gamespy.com/haqsau/hl1100.shtml.

39. **Caldwell, Nick.** Defeating lag with cubic splines. *Gamasutra.* [En línea] http://www.gamedev.net/reference/programming/features/cubicsplines/page2.asp.

40. **Aronson, Jesse.** Dead Reckoning: Latency Hiding for Networked Games. *Gamasutra.* [En línea] http://www.gamasutra.com/features/19970919/aronson_01.htm.

41. **Postel, J.** *Transmission Control Protocol.* 1981. RFC 793.

42. —. *User Datagram Protocol.* 1980. RFC 768.

43. **E. Kohler, M. Handley, S. Floyd.** *Datagram Congestion Control Protocol (DCCP).* 2006. RFC 4340.

44. **Stevens, W.** *TCP Slow Start, Congestion Avoidance, Fast Retransmit.* 1997. RFC 2001.

45. **M. Allman, V. Paxson, W. Stevens.** *TCP Congestion Control.* 1999. RFC 2581.

46. Tabla de planes y precios de ADSL. *Antel.* [En línea] [Citado el: 17 de Marzo de 2009.] http://www.antel.com.uy/portal/hgxpp001.aspx?2,424,2431,O,S,0,MNU;E;479;3;486;18;MNU.

47. **Marsan, M. Ajmone.** Introduction. *Quality of Service in Multiservice IP Networks.* s.l. : Springer, 2003.

48. **B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, L. Zhang.** *Recommendations on Queue Management and Congestion Avoidance in the Internet.* 1998. RFC 2309.

49. **S. Floyd, M. Handley, E. Kohler.** *Problem Statement for the Datagram Congestion Control Protocol (DCCP).* 2006. RFC 4336.

50. Bullet physics. [En línea] http://www.bulletphysics.com/.

51. ODE. [En línea] http://www.ode.org/.

52. Newton Game Dynamics. [En línea] http://www.newtondynamics.com/.

53. Nvidia PhysX. [En línea] http://www.nvidia.com/object/nvidia_physx.html.

54. Havok. [En línea] http://www.havok.com/.

55. Solvers in bullet. [En línea] http://www.bulletphysics.com/Bullet/phpBB3/viewtopic.php?t=1883.

56. *Graphical Modeling and Animation of Brittle Fracture.* **O'Brien y Hodgins.** 1999.

57. *Real-Time Simulation of Deformation and Fracture of Stiff Materials.* **Müller, y otros.** 2001.

58. *Fast and Controllable Simulation of the Shattering of Brittle Objects.* **Smith, Witkin y Baraff.** 2000.

59. *The Spray Shop.* [En línea] http://www.thesprayshop.net/tutorials/tut_car_physics.shtml.

60. *id Software: Technology.* [En línea] http://www.idsoftware.com/business/idtech2/.

61. *id Software: Technology.* [En línea] http://www.idsoftware.com/business/idtech3/.

62. Unreal Engine 3. *Unreal Technology.* [En línea] http://www.unrealtechnology.com/technology.php.

63. Unreal Torunament 3. [En línea] http://www.unrealtournament3.com/.

64. **Ransom-Wiley, James.** Mortal Kombat 8 inspired by Gears of War, using Unreal Engine 3. *Joystiq.* [En línea] Febrero de 2007. http://www.joystiq.com/2007/02/05/mortal-kombat-8-inspired-by-gears-of-war-using-unreal-engine-3/.

65. Lua. [En línea] http://www.lua.org/.

66. *Python Programming Language -- Official Website.* [En línea] http://www.python.org/.

67. *Squirrel The programming lenguage.* [En línea] http://squirrel-lang.org/.

68. Lua: Garbage Collection Tutorial. [En línea] [Citado el: 12 de May de 2009.] http://lua-users.org/wiki/GarbageCollectionTutorial.

69. **Burns, Jon.** *Lua in the Gaming Industry Roundtable Repor.* 2004.

70. Lua Uses. [En línea] [Citado el: 12 de May de 2009.]

71. **Wrenholt, Erik.** Ruby, Io, PHP, Python, Lua, Java, Perl, Applescript, TCL, ELisp, Javascript, OCaml, Ghostscript, and C Fractal Benchmark. *Timestretch.* [En línea] http://www.timestretch.com/FractalBenchmark.html.

72. Squirrel FAQ. [En línea] [Citado el: 12 de May de 2009.]

73. Collision detection & Response. [En línea] http://www.peroxide.dk/download/tutorials/tut10/pxdtut10.html.

74. Dragon Ball Z Shin Butouden gameplay. [En línea] http://www.youtube.com/watch?v=LoBk6y8ZEM0.

75. Uniform Resource Identifier (URI): Generic Syntax. [En línea] http://tools.ietf.org/html/rfc3986.

76. Cg Toolkit. [En línea] http://developer.nvidia.com/object/cg_toolkit.html.

77. nVidia - Visual technologies. [En línea] http://www.nvidia.com/.

78. Slerp. [En línea] http://en.wikipedia.org/wiki/Slerp.

79. COLLADA. [En línea] http://www.khronos.org/collada/.

80. *A Sweepline Algorithm for Voronoi Diagrams.* **Fortune, Steven.**

81. **de Berg, van Kreveld, Overmars, Schwarzkopf.** Capítulo 7: Voronoi Diagrams. *Computational geometry: algorithms and applications.* 2000.

82. *An Efficient Implementation of Fortune's Plane-Sweep Algorithm for Voronoi Diagrams.* **Wong, Müller.**

83. **Carmack, John.** Quake III: Arena Source Code. [En línea] http://www.idsoftware.com/business/techdownloads/.

84. Torque Network Library Documentation. *Torque Network Library.* [En línea] http://opentnl.sourceforge.net/doxydocs/.

85. DCCP. [En línea] [Citado el: 12 de May de 2009.] http://read.cs.ucla.edu/dccp/.

86. A Data-Driven Game Object System. [En línea] [Citado el: 8 de Enero de 2009.] http://www.drizzle.com/~scottb/gdc/game-objects_files/frame.htm.

87. mturoute.exe - Debug the MTU values between you and a host. [En línea] [Citado el: 13 de Feb de 2009.] http://www.elifulkerson.com/projects/mturoute.php.

88. **P754, IEEE Task.** *IEEE 754-1985, Standard for Binary Floating-Point Arithmetic.* 1985. IEEE 754-1985.

Apéndice A – Detalles de implementación del subsistema de red

En este apéndice se describe en forma detallada algunos aspectos particulares de la implementación de red.

Sobre la capa de transporte se detallará el uso y la validación de los números de secuencia. Además se describirá el protocolo de transporte a partir de sus estados y las transiciones entre ellos. Finalmente se habla sobre los métodos de escritura del mensaje.

De la capa de sincronización se describe la estructura de un estado, y los detalles de implementación de las técnicas de ahorro de ancho de banda.

1 Capa de Transporte

1.1 Números de secuencia

Para poder mantener la conexión, validar paquetes y además proveer la notificación, cada paquete de datos tiene su propio identificador de secuencia de 32 bits, el cual es incrementado con aritmética modular. Cada paquete tiene su número de secuencia que se incrementa con cada envío, la única condición en la que se pueden repetir los números de secuencia es cuando se generan duplicados de red.

Para validar los números de secuencia de una conexión y evitar conflictos con conexiones anteriores abiertas, se implementó una validación basada en ventanas similar a la que propone DCCP. Cada paquete que envía un *endpoint* a través de una conexión contiene la aceptación del último paquete recibido por el otro *endpoint,* que lo usará para actualizar sus ventanas. Se usan dos ventanas para validar los paquetes entrantes, la de secuencias y la de aceptaciones. La ventana de secuencias representa los paquetes que espera recibir la conexión, mientras que la ventana de aceptaciones los enviados por la misma.

Una ventana de validación de la secuencia está formada por el primer p y último u número de secuencia que son aceptados por un *endpoint*. Para que un paquete con un número de secuencia s sea aceptado tiene que cumplir que:

$$p \leq s < u$$

La ventana de aceptaciones tiene como límite superior ua al último paquete enviado que todavía no fue aceptado y ese número menos el tamaño de ventana de aceptación como límite inferior pa. La validación que se hace sobre un número de aceptación a es:

$$pa < a \leq ua.$$

Para ilustrar la ventana de aceptación supongamos que nos encontramos en el estado de la Figura 26. Se han enviado nueve paquetes, para cuatro de estos se recibieron aceptaciones. Los paquetes 3 y 4 no recibieron aceptaciones, pero, como ya quedaron por fuera de la ventana, si se recibieran aceptaciones serían ignoradas. El sistema espera aceptaciones para los estados 6, 7 y 8.

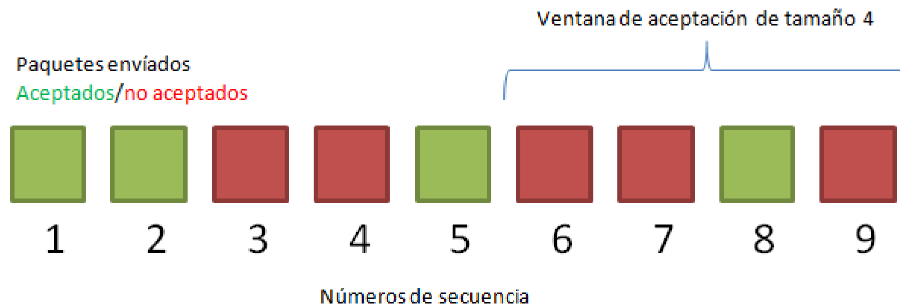

Figura 26 Secuencia de paquetes enviados.

El largo de la ventana de secuencia es igual a la cantidad de paquetes que puede aceptar un receptor. Debe ser lo suficientemente largo como para aceptar los paquetes que el emisor tiene en vuelo y no perder la sincronización en caso de una racha corta de pérdidas. También debe ser corto como para llevar al mínimo la probabilidad de colisiones con medias-conexiones y paquetes viejos. El largo de la ventana de aceptación del emisor es igual al largo de la ventana de secuencia del receptor.

La especificación de DCCP recomienda un largo igual a cinco veces la cantidad de paquetes que se esperan mandar en un *RTT.* En caso de variaciones en la cantidad esperada es recomendable que el emisor avise del cambio al receptor. En nuestro caso, para la cantidad de paquetes que se prevé mandar, se calcula una cota superior para cada conexión en base a los valores de cantidad de actualizaciones de estado por segundo (SPS) y el *RTT* en el servidor. Para el caso del cliente se estima con la cantidad de comandos por segundo (CPS).

$$l = 5 * \left\lceil \frac{SPS}{RTT} \right\rceil$$

Este valor es una cota superior, ya que las limitaciones de ancho de banda pueden llevar a que el valor real sea mucho menor. En esta implementación se asume un valor constate de RTT y el emisor envía una actualización al receptor del largo de la ventana cuando detecta que varió el valor de SPS o CPS.

El límite inferior de la ventana de secuencia es exactamente el último número de secuencia recibido, a diferencia de DCCP en donde se permiten un rango pequeño de paquetes anteriores al último. Esta decisión implica que el protocolo no acepta paquetes fuera de orden. Al recibir un nuevo número de secuencia válido el límite inferior se actualiza a este número invalidando cualquier paquete cuyo número de secuencia estuviese entre el anterior valor del límite y el nuevo.

El límite superior se actualiza cada vez que surge un cambio en el largo de la ventana o cuando se recibe y acepta un paquete.

Al ser el protocolo no confiable, puede pasar que después de una ráfaga de pérdida de paquetes se pierda la sincronización y los números de secuencia enviados por el emisor no validen contra la ventana del receptor. En este caso el receptor descarta el paquete recibido y comienza una secuencia de sincronización de manera de restablecer los valores de secuencia.

TCP y DCCP insisten en que el identificador inicial de los paquetes de una conexión debe ser elegido de manera de asegurar que no haya paquetes en vuelo de una conexión anterior que puedan calificar como válidos. Para esto proponen métodos como no comenzar una conexión desde un *endpoint* A, a un *endpoint* B determinado, hasta pasado dos veces el tiempo de vida de un segmento o paquete, en caso de que A recién se levante y no tenga conocimiento de los últimos números de secuencia manejados con B. Otro caso como el *Quake 3*, toma como valor inicial de cualquier conexión 0 y la única validación que hace es que el paquete tenga una secuencia superior al último recibido(83), pudiéndose darse el caso de que dos intentos de conexiones seguidos en un corto tiempo generen una situación en la que no se puede reconocer un paquete retrasado por la red. Para nuestro caso simplemente se toma un valor inicial aleatorio. Este método tiene una probabilidad de colisión suficientemente baja para los requerimientos de un juego.

1.2 Conexión

Una conexión en Trifulca puede encontrarse en uno de los siguientes estados:

- CLOSED: este estado se usa para indicar que no existe conexión. No existe ningún tipo de estructura asociada a una conexión en este estado.
- LISTEN: cuando un servidor decide comenzar a recibir paquetes para poder abrir una conexión se pasa al estado LISTEN. Todavía no existe conexión ya que no se conoce el otro *endpoint*.
- ACCEPTING: una conexión del lado del servidor se encuentra en este estado cuando se recibe un pedido de conexión de un cliente. En este momento se crean las estructuras que identifican la conexión.
- PENDING: un cliente se encuentra en este estado luego de hacer un pedido de conexión al servidor, que está esperando su confirmación.
- OPEN: en este estado la conexión está lista para el intercambio de datos.
- CLOSING: es un estado usado al comenzar una desconexión. Para evitar conexiones medio abiertas se espera aceptación de la desconexión, y se reenvía el paquete de desconexión, por un tiempo determinado.

1.2.1 Tipos de paquetes

Los tipos de paquetes que se pueden enviar en los diferentes estados durante el desarrollo de la conexión son los siguientes:

- CONNECT: lo envía un cliente cuando quiere abrir una conexión con un servidor. Este paquete contiene el primer número de secuencia elegido por el cliente.
- ACCEPT: es enviado por un servidor ante el pedido de conexión de un cliente. Este paquete contiene el primer número de secuencia elegido por el servidor y una aceptación del CONNECT del cliente.
- ACK: se envía para confirmar la llegada de datos. Se usa para mantener la conexión viva cuando no hay paquetes que mandar o para confirmar un ACCEPT.
- DATA: es un paquete que contiene datos en el *payload*. Los mismos son pasados a la capa superior. Este además contiene la aceptación del último paquete recibido. El tamaño máximo de un paquete DATA es de 1492 bytes incluyendo el cabezal. Este valor es el máximo MTU medido con mturoute(87) en una conexión a Internet hogareña.
- CLOSE: es enviado cuando uno de los *endpoints* decide finalizar la conexión. Este paquete incluye la aceptación del último paquete recibido.
- RESET: es enviado en respuesta a un pedido de desconexión o en caso de que se reciba un tipo de paquete no esperado.
- SYNC: es un pedido de sincronización, sirve para iniciar una secuencia de sincronización ante la llegada de un paquete con secuencia o aceptación inválida.
- SYNCACK: es la respuesta a un pedido de sincronización.

Cada paquete enviado incrementa en uno el número de secuencia, no existen reenvíos reales.

El pasaje entre estos estados se describe a continuación. El pasaje común se encuentra representado por el diagrama de estados de la Figura 27. En este diagrama las transiciones se muestran en pares RECIBIDO/ENVIADO donde RECIBIDO puede ser un paquete de otro *endpoint* o pedidos de la capa superior.

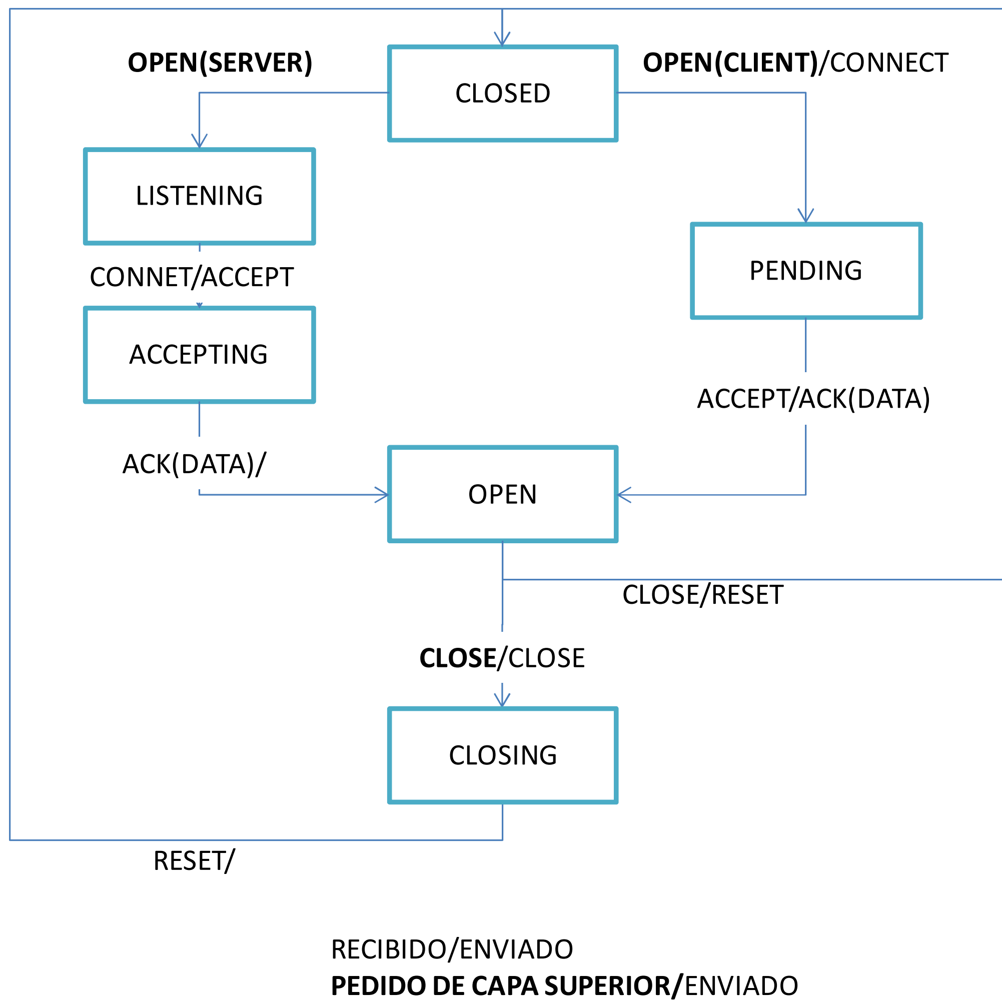

Figura 27 Transiciones entre estados de conexión.

1.2.2 Establecimiento de conexión

El método para establecer una conexión es equivalente a los utilizados por DCCP y TCP: se lo conoce como *three-way-handshake.*

El flujo común para establecer una conexión, como se muestra en la Figura 28, es el siguiente:

- Un *endpoint* se encuentra en el estado LISTEN esperando un pedido de conexión. Este *endpoint* será el servidor.
- Otro *endpoint* envía un paquete CONNECT al servidor con el número de secuencia elegido y se queda en el estado PENDING.
- El servidor envía un ACCEPT al cliente indicando que acepta la conexión e informa el número de secuencia que eligió. Queda en el estado ACCEPTING en espera de una confirmación del cliente.
- El cliente, al recibir el ACCEPT, pasa al estado OPEN y envía un ACK.
- El servidor recibe el ACK y pasa al estado OPEN.

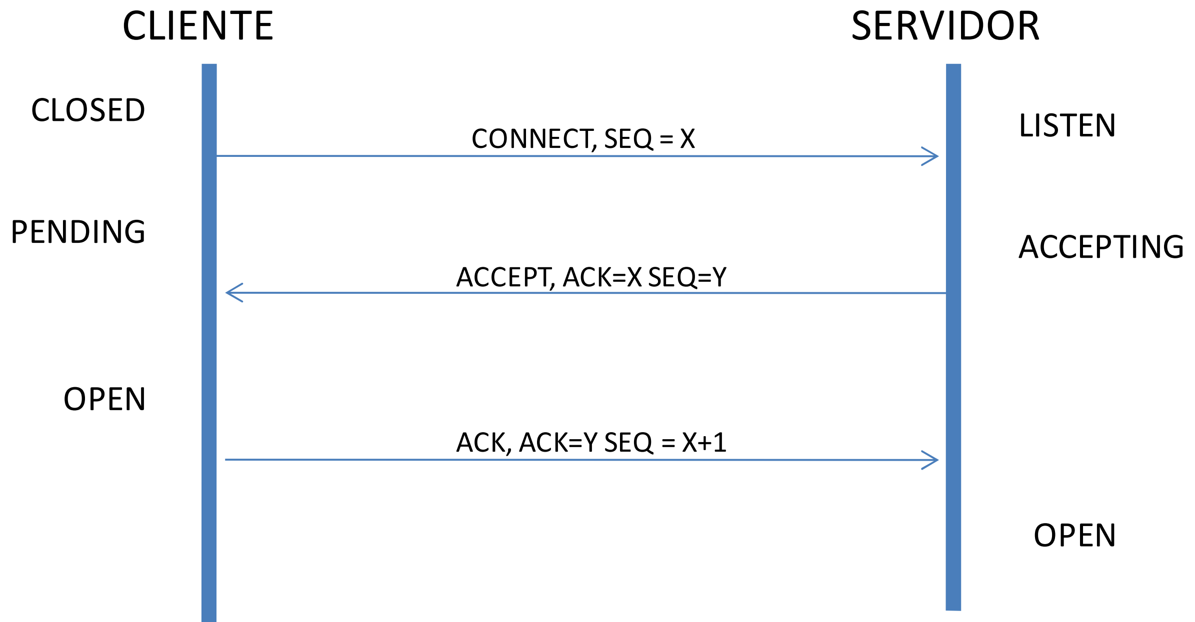

Figura 28 Flujo común al establecer una conexión.

Cuando el servidor se encuentra en el estado LISTEN, no valida el número de secuencia de los pedidos de conexión. El único paquete que considera válido es el de tipo CONNECT y ante cualquier otro paquete envía un RESET. Esto es debido a que otro paquete indica una conexión medio abierta que debe ser finalizada. El cliente reenvía el pedido de conexión con *exponential backoff*, el primer pedido se reenvía comenzando al segundo de haber enviado el primer CONNECT si no recibió otro pedido de conexión. Se reenvía hasta un máximo de cinco veces, luego del cual se envía un RESET por si el servidor recibió alguno de los CONNECT, se pasa a estado CLOSED y se avisa a la capa superior.

En el estado ACCEPTING si recibe un pedido de conexión con secuencia válida y se reenvía el ACCEPT correspondiente. Si el cliente recibe un segundo ACCEPT válido lo descarta. El servidor no reenvía ACCEPT a menos que reciba un CONNECT válido, espera a que el cliente reenvíe el pedido de conexión como puede verse en la Figura 29. Si el servidor no recibe una respuesta después de medio minuto luego del último pedido de conexión da la conexión por terminada, envía un RESET y pasa al estado LISTEN.

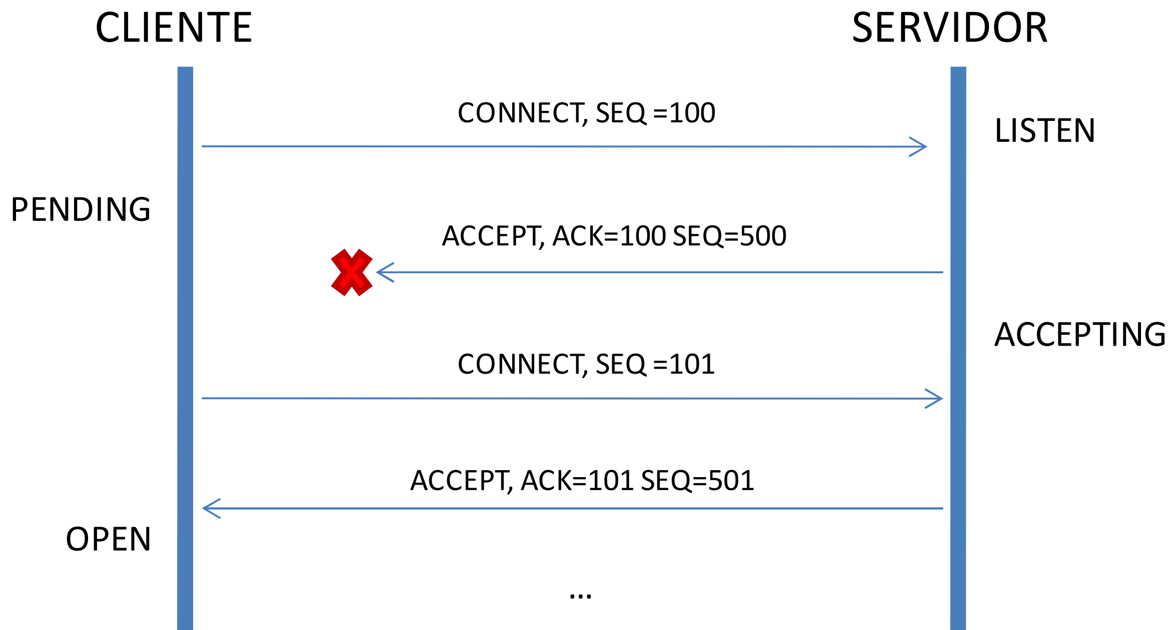

Figura 29 Reenvío de ACCEPT desde el servidor.

El servidor al recibir el ACK de parte del cliente pasa al estado OPEN y puede comenzar a enviar datos. La confirmación que acepta el servidor puede ser cualquier paquete de datos ya que como se mencionó antes estos incluyen una aceptación. Así se concreta la conexión y los datos son pasados a la capa superior. Este comportamiento puede verse en la Figura 30.

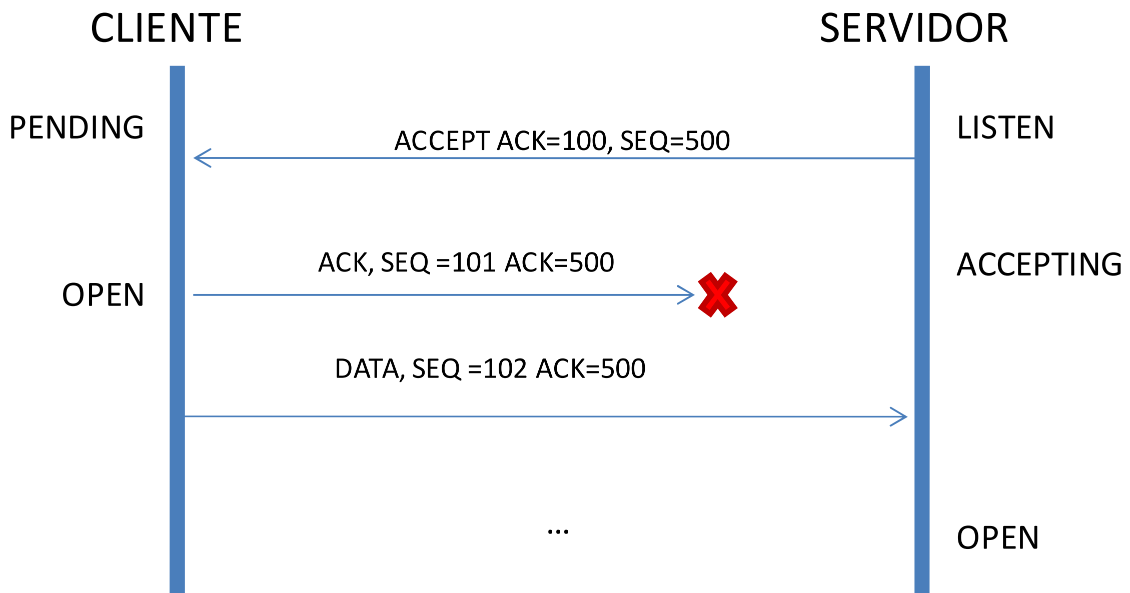

Figura 30 El servidor puede reconocer un paquete de datos como aceptación de un ACCEPT

1.2.3 Envío de datos

El envío de datos generalmente se realiza enviando paquetes DATA que confirman estados anteriores como se puede ver en la Figura 31.

Figura 31 Secuencia de paquetes de datos.

El envío de datos entre *endpoints* puede comenzar únicamente cuando el servidor llega al estado OPEN y el cliente al estado PENDING u OPEN. Por cada paquete de datos enviado se incrementa el número de secuencia y no existen reenvíos de datos. La capa superior debería reenviar el dato pedido si así lo requiriera pero esto es independiente de esta capa, que los tomaría como dos paquetes diferentes.

La validación de números de secuencia se hace como fue indicado en la sección de *números de secuencia.* En el estado OPEN los únicos paquetes válidos son CLOSE, DATA, SYNC y ACK.

Se mantiene la conexión mediante reenvío de acks. Al cabo de diez segundos si no se envió un paquete, ya sea DATA, SYNC o ACK, se envía un ACK para mantener viva la conexión como se puede ver en la Figura 32.

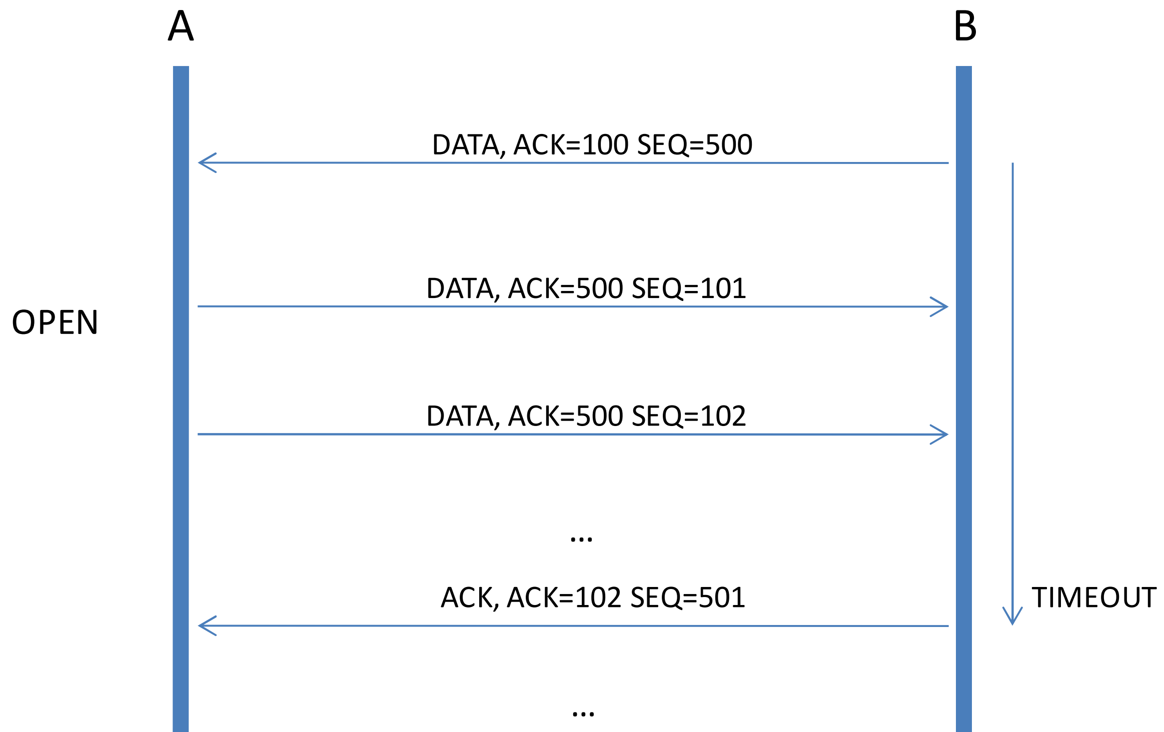

Figura 32 Mensaje ACK para mantener viva la conexión.

En caso de no recibir ningún paquete esperado del otro *endpoint* por un período de medio minuto se cierra la conexión por inactividad y se envía un RESET indicando el hecho.

1.2.4 Finalización de la conexión

La desconexión puede ocurrir de dos maneras: iniciada por la capa superior, o causada por errores o condiciones inesperadas.

En el caso de una finalización normal se siguen los siguientes pasos que se muestran en la Figura 33:

- Un *endpoint* A en estado OPEN envía una solicitud de desconexión CLOSE y pasa al estado CLOSING.
- El *endpoint* B que también se encuentra en estado OPEN al recibir el CLOSE envía un RESET y pasa al estado CLOSED.
- El *endpoint* A al recibir el RESET pasa al estado CLOSED.

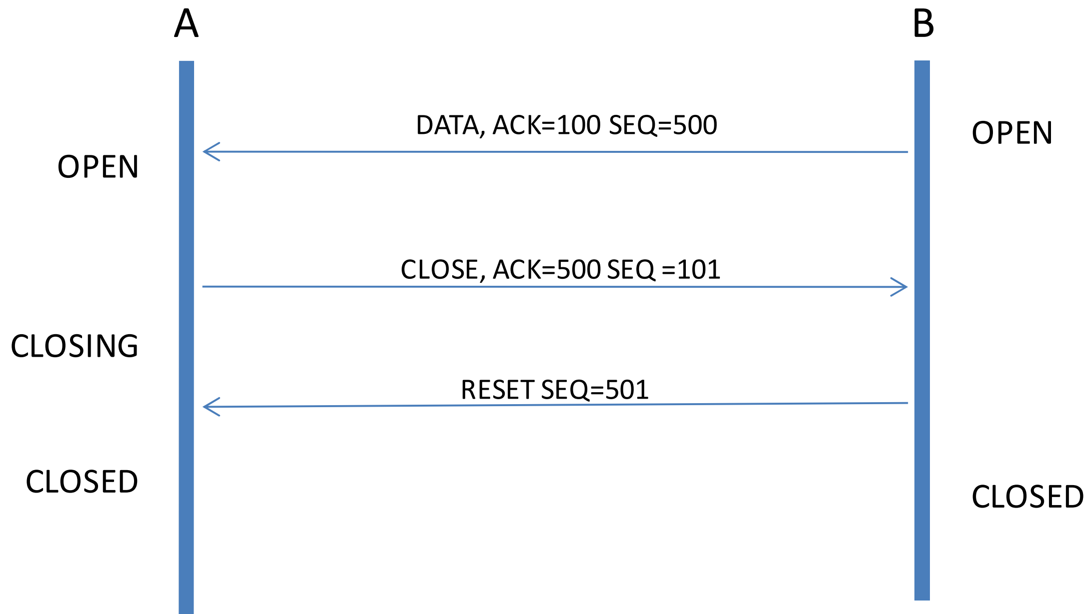

Figura 33 Finalización normal de una conexión.

En el estado CLOSING si se recibe un paquete de tipo DATA con número de secuencia y aceptación válidos entonces se reenvía el mismo CLOSE.

El CLOSE es reenviado con *exponential backoff* comenzando desde un segundo y ejecutándose hasta tres veces, luego se envía un RESET y se pasa al estado CLOSED.

No existe un estado similar a TIME-WAIT de TCP y DCCP al recibir un CLOSE y pasar al estado CLOSED. El estado TIME-WAIT se usa para que se descarten todos los paquetes que puedan estar en vuelo de una conexión cerrada y que no colisionen con los de una conexión nueva. La probabilidad de que se seleccione un número de secuencia igual al de paquetes que puedan estar en vuelo es muy baja ya que son seleccionados al azar y el protocolo está diseñado exclusivamente para este juego.

La finalización abrupta de una conexión sucede cuando se recibe un paquete incorrecto de acuerdo al estado en el que se encuentra el *endpoint*, pero con una secuencia válida. También puede suceder cuando por una conexión inactiva.

Una conexión al recibir un RESET válido debe finalizar la conexión inmediatamente.

1.2.5 Sincronización

La sincronización es el método utilizado para restablecer las ventanas de aceptación en caso de que ocurran ráfagas de pérdidas. Las sincronizaciones ocurren en los estados en los que alguna vez se estuvo sincronizado, que son todos menos LISTEN, CLOSED y PENDING, cuando los números de secuencia no son válidos.

La secuencia de una sincronización es la siguiente:

- A recibe un paquete luego de que se hayan perdido muchos, al no validar su secuencia envía un SYNC
- B al recibir un SYNC verifica que sea válido de acuerdo a lo último que recibió de A, si es válido envía un SYNACK
- A al recibir el SYNACK actualiza su ventana.

Un ejemplo de sincronización causado por varias pérdidas se muestra en la Figura 34.

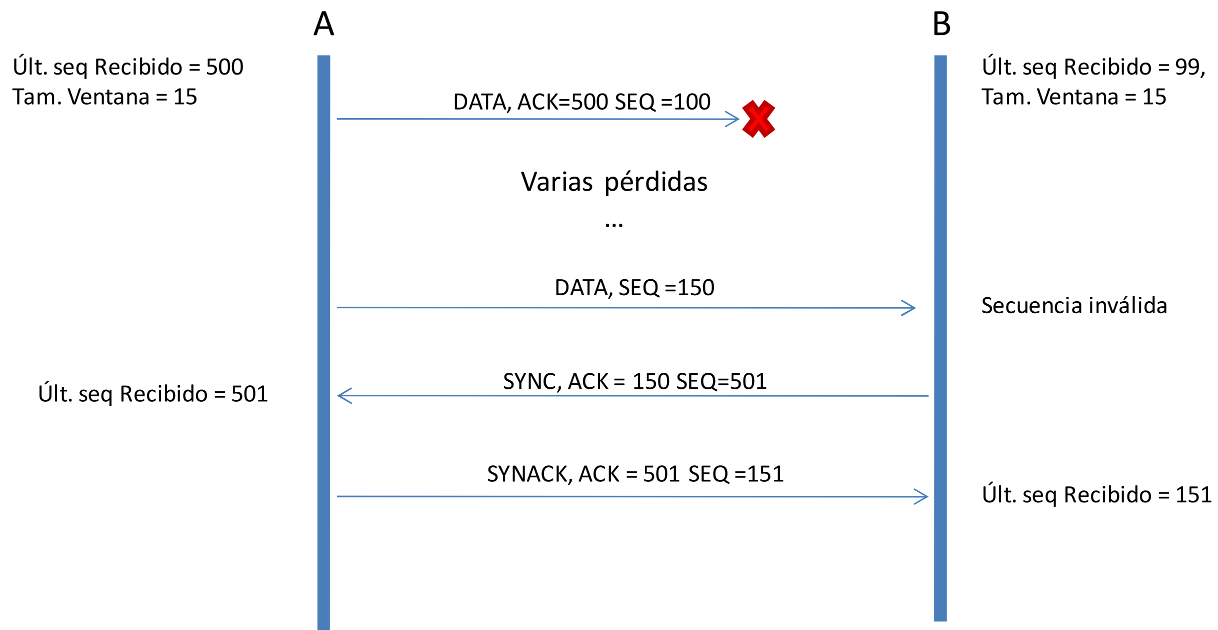

Figura 34   Re sincronización de números de secuencia por ráfaga de pérdidas.

Otro ejemplo es cuando llega un paquete viejo o a través del intento de un atacante como se puede ver en la Figura 35.

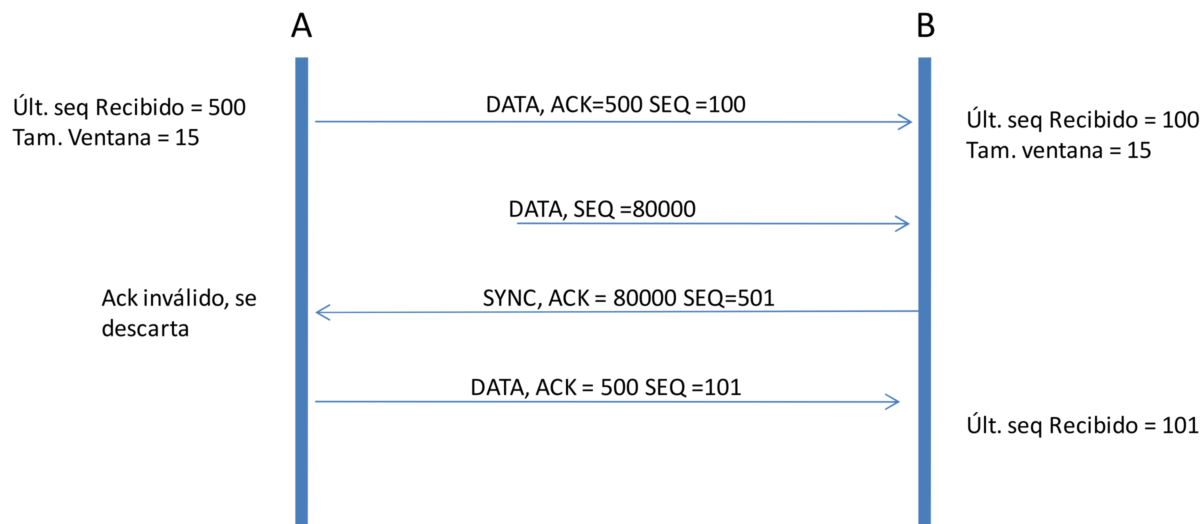

Figura 35 Re sincronización por paquete viejo o intento de ataque.

El SYNC enviado tiene un número de aceptación igual al número de secuencia recibido inválido, de manera que A pueda detectar que el SYNC es respuesta al dato que envió. El SYNACK, de la misma manera, envía la aceptación del SYNC que recibió.

Cuando se recibe y valida el SYNACK entonces se consideran los estados válidos.

Un *endpoint* al recibir un SYNACK valida el número de aceptación y la ventana de aceptación, también que haya sido enviado un SYNC. Si el paquete no cumple estas condiciones entonces es descartado.

Si el SYNACK recibido es válido entonces se actualiza la ventana de secuencia y se considera que ambos *endpoints* se encuentran nuevamente sincronizados.

1.3 Mensajes

El mensaje tiene un tamaño limitado y se corresponde con el espacio asignado de carga del paquete DATA de la capa de red.

Como fue mencionado anteriormente la escritura se hace como un *stream* de datos. Pero a pesar de que es suficiente en la mayoría de los casos, existen otros en donde es necesario escribir los datos fuera de orden. Por este motivo se implementó un sistema de marcas, en el cual se permite la reserva de la posición relativa y espacio de un valor para poder ser escrito luego de ser procesados otros datos. Este método es útil, por ejemplo, debido a que muchas tiras de datos escritas por capas superiores se representan primero escribiendo el tamaño y luego la tira. Esto es porque muchas veces en el juego no se sabe a priori el tamaño de una cadena a mandar. Así este método facilita la lectura de la cadena, debido a que se puede reservar la memoria necesaria para leer sus entradas al leer el largo y además evita una entrada especial de finalización de cadena que puede ser no trivial en tipos de datos complejos.

Por los requerimientos de ancho de banda era necesario poder escribir los datos en la forma más compacta posible. Por esta razón se definieron operaciones de escritura de tipos de datos básicos con diversos tamaños. Se permite la escritura de enteros con y sin signo especificando el tamaño de los mismos en bits, valores de punto flotante de simple o media precisión, booleanos y cadenas de caracteres.

- Los enteros son de 32 bits y lo que se escriben son los n bits menos significativos siendo n la cantidad de bits elegida. La representación de los enteros es complemento a dos, es decir, los valores positivos son iguales a su representación binaria mientras que los negativos son los que tienen el bit más significativo en uno (el i-ésimo bit).
- Los valores de punto flotante se permiten escribir con precisión simple o media precisión. La precisión simple es la usada comúnmente, se usa el estándar IEEE 754 (88) debido a que es la implementada en las arquitecturas objetivo. Para el caso de la media precisión se usa la mitad de la mantisa dejando igual el exponente y el signo. Este formato permite una precisión aceptable para el mundo del juego.
- Las cadenas de caracteres se imprimen en formato ANSI carácter a carácter usando el carácter nulo (ó 0) para representar la finalización del mismo de manera de permitir cadenas de largos arbitrarios.
- Para valores booleanos, que son muy usados debido a la técnica *compresión delta*, se utiliza un solo bit. Se representa el verdadero con 1 y el falso con 0.

La escritura de los valores no está alineada, es decir, cada valor es escrito exactamente desde el bit siguiente al último bit ocupado por el valor anterior. La lectura y escritura son, entonces, más lentas y complicadas de implementar, mantener y extender. Estos problemas son opacados por el ahorro de espacio que se gana en el mensaje, comparado con una escritura en donde los valores están alineados, por lo menos, al mínimo representable por las arquitecturas que se manejan. La ganancia es notoria en este tipo de desarrollo ya que es muy común que en los juegos los valores que se manejen estén acotados en rangos muy pequeños como las vidas, una barra de energía discreta, el estado de una animación o identificadores de tipo. En la Figura 36 se puede ver como se escriben el entero 4, la cadena de caracteres “AB”, los valores *verdadero* y *falso booleanos,* y el valor 15 en punto flotante de precisión simple.

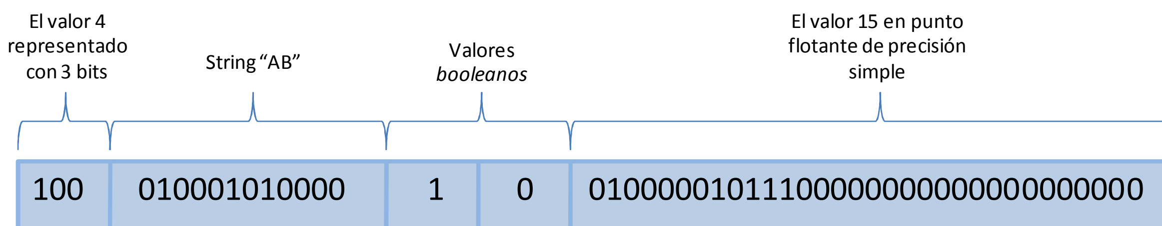

Figura 36   Varios tipos de datos empaquetados en un mensaje.

Se implementó además un sistema de *rollback* (vuelta atrás) que consiste en dejar una marca en el mensaje de modo que la escritura de un conjunto de datos fuertemente relacionados pueda ser vista

como una transacción. Lo que permite esto al momento de desarrollo, entre otras cosas, es eliminar las verificaciones de tamaño máximo de mensaje en cada escritura, dejando una única verificación en el último valor escrito del conjunto de datos a mandar. En caso de que el último valor no pueda ser escrito se puede descartar el conjunto entero. Así el espacio restante puede ser utilizado por otros datos más pequeños.

2 Capa de Sincronización

2.1 Estructura de estados

Los valores mínimos para la representación por parte del cliente de un mundo son la transformación, la animación y la URI de la geometría que la representa.

La transformación es una matriz de dimensiones 4x4 de punto flotante en la cual la matriz 4x3 izquierda contiene la rotación y el escalado mientras que el vector columna restante la traslación. La última fila de la transformación entera es usada únicamente para simplificar cálculos siendo todos sus valores constantes. Por esta razón en una actualización de estado no se incluyen estos valores sino que son asumidos al armar la transformación cuando es recibida. Como el juego no utiliza escalado la matriz de 3x3 que queda para rotación y escala contiene información irrelevante para la transmisión. Debido a esto el estado de una entidad no incluye la matriz completa sino que extrae la rotación a un vector 4x1 que se conoce como *cuaternión*. El vector de traslación de tres entradas se incluye entero en la actualización ya que no existe información redundante en el mismo y se puede considerar importante en su totalidad. De esta manera se envían solamente siete valores de punto flotante como estado posicional de una entidad. Por lo general se espera que las pantallas estén acotadas a un cubo bastante pequeño comparado con el rango de representación de punto flotante de la arquitectura elegida.

Como la precisión de la representación es mucho mayor que la necesaria se usó media precisión para todos estos valores. Se puede notar que dependiendo del tipo de objeto la transformación no es usada completamente. Por ejemplo los objetos dinámicos de la pantalla se mueven en un espacio tridimensional mientras que los luchadores sólo en dos dimensiones. Se podría ahorrar más ancho de banda haciendo que la red pueda diferenciar entre estos dos casos, pero limitaría la extensibilidad ya que requeriría modificar la red cada vez que se agrega un tipo de entidad nuevo. Otra opción es hacer que el que conoce el movimiento y las limitaciones del mismo, que es la lógica, defina como es necesario replicar la transformación dándole a la red métodos de serialización. Este método es usado por el motor *Source* que termina integrando conocimientos del funcionamiento de la red al módulo que es visible a los desarrolladores de modificaciones(32).

Las animaciones incluyen como estado relevante la URI que la identifica, el tiempo en que se encuentra la animación, la velocidad a la que corre la animación y si debe reiniciar o no una vez finalizada. Como el tiempo que lleva corriendo la animación es un valor relativo con respecto al tiempo de ejecución del juego entonces se puede obviar. Este tiempo siempre puede ser calculado del lado del cliente a partir del tiempo en que comenzó a correr la animación y el tiempo que se usa para el dibujado. La animación

es una URI por lo que se representa como una cadena de caracteres y la velocidad un valor de punto flotante de media precisión.

En Trifulca la geometría de una entidad está identificada por la URI del recurso de donde se carga, el mismo es enviado al momento de la creación de una entidad para su carga por parte del cliente. Toda modificación sobre la geometría está basada en la animación y puede ser reproducida usando la actualización de ésta.

Los datos que no se conocen de la entidad están definidos en una estructura genérica que puede contener diferentes tipos de datos entre enteros, punto flotante y cadenas de caracteres. Los mismos los pueden definir los diferentes módulos pero deben indicarle a la red si debe enviarlos o no y además, si tienen rango para el caso de los enteros, o si necesitan compresión en el caso de los valores de punto flotante. Un ejemplo de estos valores son la energía y las vidas restantes de los luchadores. En el caso del Quake3 la red no conoce ningún atributo específicamente si no que los trata a todos como cadenas de bytes. Esto evita ciertas características de compresión que son intrínsecas de los tipos de datos.

2.2 Técnicas

2.2.1 Tablas de strings

Como las animaciones y las entidades están representadas por su URI completo, es común que en las actualizaciones de estado se incluyan muchas cadenas de caracteres que ocupan mucho más que el resto de los tipos de datos. La compresión delta ayuda a que se envíen muchas menos cadenas debido a que, por ejemplo, la animación actual se envía únicamente cuando cambia. De todos modos ocurren muchas actualizaciones de animaciones durante el curso del juego y esto deriva en un consumo elevado de ancho de banda.

Para limitar el impacto de las cadenas de caracteres se usa la técnica de tablas de strings. En esta implementación al enviar una cadena por primera vez se asocia la misma a un entero que se envía junto con la cadena. El receptor al recibir el mensaje con la cadena y su identificador la guarda localmente. El emisor al encontrarse con que tiene que enviar la cadena nuevamente puede enviar únicamente el identificador de la cadena. Cuando el cliente lo recibe puede recuperar la cadena de su mapa local. Se podrían definir previamente todas las cadenas que se pretende manejar durante un juego de manera de no tener que enviar ninguna (esto sin contar otro tipo de comunicación como puede ser chat), pero esto limitaría la facilidad que ofrece el motor del juego con respecto a la definición de recursos. Además el primer envío es despreciable comparado con la cantidad de datos que se enviarán durante el transcurso de una partida.

En el juego se tienen tablas con mayor capacidad que la cantidad los *strings* que se espera mandar durante un juego, considerando las animaciones y las definiciones de los mapas. En caso de que se llene el buffer de cadenas se usa una estrategia FIFO para la elección de la entrada a reemplazar. Se podrían considerar estrategias que consideren valores más usados, pero esto requeriría guardar la información y mantenerla ordenada. Como no se espera superar el máximo de *strings* se decidió mantener la implementación simple.

2.2.2 Compresión delta

La técnica que logra el mayor ahorro de ancho de banda en la distribución de estados es la compresión delta. Ésta requiere que el servidor guarde los últimos n estados enviados a cada cliente para poder comparar cuales cambiaron, y que el cliente confirme la llegada de estos. Así el servidor puede conocer desde qué estado comprimir. El cliente también debe mantener los últimos n estados que recibió del servidor para lograr reconstruir el paquete a partir de los valores anteriores y los nuevos. Cabe destacar que no alcanza con que el cliente mantenga sólo el último estado recibido por si el servidor envía más estados de los que confirma un cliente. Por ejemplo, si el servidor envía un estado E_1 y se tiene un RTT de 200 milisegundos, entonces puede que la confirmación del mismo no llegue antes de que se envíe el estado siguiente E_2, que será delta comprimido desde el mismo estado confirmado que E_1. Si el cliente se queda únicamente con el último estado, entonces puede ser que el estado E_2 sea armado en el cliente desde el estado E_1 pudiendo provocar valores incorrectos. En trifulca se mantienen suficientes estados como para manejar una ráfaga de hasta cuatro segundos de pérdida de paquetes recuperando un estado confirmado por el cliente. Una vez superado este límite, se debe enviar los estados completos, situación en la cual el cliente debe dar de baja todas sus entidades y crear las que aparecen en el nuevo estado recibido. En el cliente se mantienen los estados que llegaron desde el último estado sobre el cual se aplica la compresión delta.

2.3 Estimación de diferencia entre paquetes.

En Trifulca se estima la diferencia de tiempo entre los paquetes que llegan tratando de lograr un valor cercano a la diferencia más grande notada en el último tiempo. Si los tiempos de respuesta son uniformes, se intenta reducir esta medida para que se acerque lo más posible al valor esperado de la diferencia, con la restricción de que se debe mantener cerca del límite inferior impuesto por la desviación estándar. Esta estimación es para que retrasos predecibles en la actualización no provoquen saltos visibles en el dibujado. La estimación del tiempo se hace basándose en la estimación anterior y la diferencia entre los dos últimos estados recibidos, para poder mantenerla continua con respecto al *frame* actual. La fórmula usada es similar a las usadas en diferentes implementaciones (Quake 3(83), TNL(84)) y es

$$t_n = t_{n-1} * a + \Delta t * (1 - a) + \beta$$

Con t_n como la nueva diferencia de tiempo, t_{n-1} la diferencia de tiempo calculada para la llegada del paquete anterior, Δt es la diferencia entre t_n y t_{n-1} y a es un valor grande cuando Δt es mayor a 0 (o sea la diferencia de tiempo actual es mayor que la anterior) y chico en caso contrario. La discrepancia entre los valores de a tomados es debido que se quiere lograr que si la diferencia en tiempos aumenta entonces el juego se adapte rápido, mientras que si la diferencia es chica entonces el juego se vaya acercando lentamente a la diferencia promedio.

Además de la diferencia de tiempo de llegada de los paquetes, se agregó una constante de tiempo más a la diferencia de tiempo calculada para determinar el momento de dibujado de los estados, de manera de poder controlar conexiones con cambios muy abruptos en su confiabilidad y tiempos de respuesta. Este es un parámetro denominado *packetDelay.*