UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

DESARROLLO DE UNA PLATAFORMA PARA LA INVESTIGACIÓN DENTRO DE MUNDOS VIRTUALES

T E S I S

QUE PARA OBTENER EL TITULO DE

INGENIERO EN COMPUTACIÓN

P R E S E N T A:

HERRERA ROSALES DAVID ABEL

DIRECTOR DE TESIS:

M. I. RODRIGO TINTOR PÉREZ

CIUDAD UNIVERSITARIA, MÉXICO D.F. MARZO 2011.

Universidad Nacional Autónoma de México
UNAM
Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Agradecimientos

A mis abuelos Felisa y Abel, Paula y Rodolfo, por su cariño y amor incondicional.

A mis papas Nora y Abel, por los valores que me inculcaron, los ejemplos de perseverancia, por dame la oportunidad de tener la mejor educación posible y sobre todo por su amor.

A mis abuelas Felisa y Joela, que siempre recordare por su cariño entrañable.

A mi hermana Erika y prima Cristi, por su apoyo y cariño por siempre.

A mi amigo David De Leon, por su amistad inquebrantable.

A Michael Zyda y Balki Ranganathan de la USC por permitirme participar en el proyecto y su apoyo.

A la Facultad de Ingeniería, con aprecio a todos mis profesores y amigos ingenieros.

A la UNAM, con el orgullo de ser universitario.

Índice

1. Introducción

Una parte crítica de la ciencia social es el desarrollo de nuevas teorías para explicar las interacciones sociales y la formación y usos de las redes sociales, de manera paralela a este problema, se necesita una forma de realizar pruebas a estas teorías para asegurar su validez. Estos problemas se ven obstaculizados por la falta de conceptos operacionales en las interacciones humanas, por la dificultad de llevar a cabo experimentación en grandes poblaciones y la extracción de datos específicos de ellas. La plataforma Cosmopolis se desarrolló para mejorar las pruebas de los modelos teóricos existentes y elaborar nuevas teorías.

Cosmopolis es mundo virtual MMOG que tiene como objetivo estudiar modelos sociales y de comportamiento. Cosmopolis permite la interacción de una gran cantidad de jugadores, además facilita la creación de juegos donde se pueden realizar experimentos controlados.

Para los jugadores, Cosmopolis es un MMOG que consiste en un mundo virtual y juegos de cualquier género. El objetivo es mantener el interés del jugador por la calidad de los juegos y las gráficas. Para los investigadores, Cosmopolis es una plataforma para la implementación y obtención de información de modelos sociales y de comportamiento. Los modelos pueden ser de jugadores específicos, de múltiples jugadores, de personajes virtuales.

La intención de Cosmopolis es ser una plataforma versátil para ser utilizada por otros investigadores interesados en experimentos en ciencias sociales y modelos de comportamiento. La plataforma también funciona como un motor de juegos, ya que permite la creación de diferentes tipos de juegos para la investigación y contienen varios de los subsistemas en un motor de juegos.

El mundo virtual tiene una estructura de 2 niveles: mundo externo y juegos. El mundo externo es una simulación de una ciudad moderna. Los juegos se encuentran dentro del mundo externo, pueden estar completamente aislados, por ejemplo, dentro de un edificio, o pueden estar integrados al mundo externo, por ejemplo, dentro de la ciudad o sus periferias.

1.1 Mundos virtuales como discos de Petri

En el 2006 Castranova propuso que los mundos virtuales podían servir como plataformas para la experimentación con una variedad de individuos, organizaciones y modelos sociales (Castronova, 2006). Además, para demostrar su propuesta realizó pequeños experimentos en el juego *Neverwinter Nights* donde demostró que conceptos del mundo real, como la ley de la oferta y demanda, también son aplicables a mundos virtuales (Castranova, 2008).

El género de los juegos MMOG abrió las puertas a la investigación de modelos de grupos. En la actualidad ya existen una gran cantidad de investigaciones realizada dentro de juegos MMOG, a pesar de que estos juegos no fueron desarrollados para tal propósito. El uso de los juegos MMOG para la investigación se debe a que proporcionan al investigador las siguientes características:

- Ambientes de investigación extensos.
- Una gran cantidad de sujetos de estudio (jugadores); algunos juegos comerciales llegan a tener millones de jugadores.
- Mundos persistentes.
- Interacción social en los mundos virtuales.

En el 2007 el juego *World of Warcraft* se usó como herramienta para la investigación de comportamientos sociales de usuarios que usan avatares ("The Proteus Effect: The Effect of Transformed Self-Representation on Behavior", 2007). Un avatar es una representación virtual de una persona que permite al jugador interactuar dentro del mundo virtual en tiempo real. Los resultados de la investigación sugieren que el comportamiento de los usuarios es modificado dependiendo de la representación visual de su avatar.

El National Research Council de USA publicó en el 2008 un reporte de modelos de comportamiento y simulación, donde se sugiere que los MMOG sean utilizados como plataformas para el desarrollo de modelos de interacción social humana (Zacharias, 2008). En el estudio se destaca que es necesario desarrollar una infraestructura tecnológica para desarrollar, probar y aplicar modelos de comportamientos. Cosmopolis es un MMOG desarrollado para este propósito.

1.2 Videojuegos y Motores de Juegos

Los experimentos dentro de la plataforma Cosmopolis son juegos, es por esto que se considera necesario tener un fundamento en su historia, proceso de desarrollo y tipos de juegos.

En la actualidad los videojuegos no son únicamente considerados herramientas de entretenimiento, también son utilizados por la industria para la investigación, educación o publicidad.

En el 2002 surge la iniciativa de Serious Games, su objetivo es la creación de herramientas para la educación, exploración y administración mediante el diseño de juegos y tecnologías computacionales. Algunos juegos facilitan la educación ya que tienen un contenido de estrategia, administración o historia. Otros juegos se desarrollan con el objetivo de ser terapias para pacientes con problemas de aprendizaje o lesiones cerebrales. Como consecuencia, ha surgido una nueva área de la computación y los videojuegos, que no tiene como único objetivo el desarrollo de software de entretenimiento y es flexible para abarcar múltiples necesidades.

La creación de un videojuego actualmente requiere de un equipo multidisciplinario de profesionales como: artistas, modeladores, animadores, ingenieros, etcétera. La complejidad de los juegos se ha incrementado en paralelo con el desarrollo tecnológico, de manera que se ha incrementado el realismo de los juegos y las diferentes maneras de interactuar con ellos.

El motor de juegos es una herramienta que facilita la creación de juegos y la reutilización del código. Esta herramienta permite crear juegos similares sin necesidad de modificar grandes cantidades de código.

Los juegos pueden ser clasificados en géneros por una serie de elementos que tienen en común. Esta tesis aborda principalmente el género de juegos de MMOG (*Massive Multiplayer Online Games*) ya que representa una oportunidad para estudiar modelos de comportamiento y sociales en comunidades virtuales.

1.3 Definición y problemática

Para mejorar las pruebas de los modelos teóricos y permitir la elaboración de nuevas teorías se busca crear una plataforma con las siguientes características:

- Una plataforma versátil para ser utilizada por otros investigadores interesados en experimentos en ciencias sociales y modelos de comportamiento.
- La creación de ambientes diversos de gran extensión.
- La posibilidad de realizar múltiples experimentos controlados.
- Un soporte a una gran cantidad de sujetos experimentales (jugadores).
- Un registro de eventos y acontecimientos en los experimentos, y dentro de la plataforma.

La plataforma Cosmopolis es un MMOG ya que este género está compuesto por juegos que soportan una gran cantidad de jugadores simultáneos en un mundo virtual persistente de amplias dimensiones, entre otras características.

Cosmopolis permite seleccionar la información que es guardada en una base de datos. La información es referente a la interacción entre personajes y mundo, esto para su posterior análisis e interpretación con los modelos. La información guardada es controlada y configurada por los investigadores. Cosmopolis utiliza diferentes canales de información como: información del juego, cliente, servidor, etc.

Al diseñar el juego se buscó un balance entre el interés del jugador y del investigador: desarrollar algo entretenido para los jugadores y permitir la flexibilidad de incorporar experimentos para los investigadores.

La necesidad de crear diferentes juegos que funcionan como experimentos requiere que la plataforma Cosmopolis funcione de manera similar a un motor de juegos. Debido a la complejidad y extensión del proyecto, se participó en el proyecto Cosmopolis ya iniciado y actualmente en desarrollo en la USC (*University of Southern California*). La duración de la participación en el proyecto fue de aproximadamente 7 meses.

El equipo de desarrolló de Cosmopolis se encuentra integrado principalmente de estudiantes del posgrado de ciencias de la computación, en el capítulo 4 se describirá más detalladamente.

Los objetivos de participar en el proyecto fueron:

- Adquirir mayor experiencia en el proceso de desarrollo de juegos.
- Involucrarse en el desarrollo de juegos MMOG.
- Estudiar las herramientas utilizadas para el proceso de creación de juegos como son: motores de juegos, librerías, patrones de diseño, y lenguajes de programación.

1.4 Estructura del Trabajo

En esta tesis, se comienza por describir la historia de los videojuegos para entender la evolución de los géneros de juegos y las tecnologías involucradas en su desarrollo. Posteriormente se describe en más detalle una herramienta esencial en la construcción de juegos, el motor de juegos. Finalmente, se da una descripción del desarrollo la plataforma Cosmopolis, y mi aportación al proyecto.

1.5 Resultados esperados

La plataforma tendrá una estructura de 2 niveles: mundo externo y juegos. El mundo externo es una simulación de una ciudad moderna y contiene a los juegos.

Los varios juegos se pueden encontrar completamente aislados y en ellos se realizan los experimentos controlados. Se desarrollará un juego de shooter para realizar un primer experimento.

El mundo deberá funcionar con múltiples instancias del mismo juego y con múltiples clientes conectados a un servidor. Los clientes podrán registrarse a los juegos al interactuar con NPCs (*None Player Characters*).

2. Videojuegos

2.1 Definición

Un videojuego es un juego electrónico que tiene interacción con el usuario mediante una interfaz de usuario y genera una respuesta visual mediante un dispositivo de video.

2.2 Historia de los videojuegos

Los años de 1958 y 1961 son de suma importancia ya que se registran los primeros desarrollos de videojuegos.

En 1958 en Brookhaven National Laboratories, Nueva York, el físico Willy Higinbotham desarrolló un tenis de mesa interactivo, ping pong, en un osciloscopio (mostrado en la Figura 2.1).

Figura 2.1. Tenis de Mesa Interactivo en osciloscopio

En el verano de 1961, Digital Equipment donó una computador PDP-1 (*Programmable Data Processor-1*) al MIT. El PDP-1 fué la primera computadora comercial interactiva, tenía una pantalla de rayos catódicos y utilizaba la cinta de papel perforado como medio de almacenamiento primario, se muestra en la Figura 2.2. Steve Russell desarrolla el primer videojuego en esta computadora, Spacewar. Termina un prototipo de Spacewar en 1961 y la versión final en 1962.

Figura 2.2. Computadora PDP-1

Ralph Bear, por algunos considerado como el padre de los videojuegos, en 1966 comienza a desarrollar su idea de hacer juegos para la televisión. Una idea considerada descabellada en la época y no fue apoyada en su comienzo. En 1968 patenta su televisión interactiva.

En 1971 firma un contrato con Magnavox para desarrollar el producto Odyssey. Odyssey era la primera consola de videojuegos que se podía conectar a la televisión, fue limitada a televisiones Magnavox, lo que limitó su mercado. Odyssey utiliza juegos intercambiables que internamente son circuitos impresos, los juegos y la lógica son implementados con DTL (lógica diodo-transistor), se muestra en la Figura 2.3.

Las consolas caseras no tendrían la popularidad inmediata que adquirieron sus similares, las máquinas de videojuegos operadas por monedas, *arcades.*

Figura 2.3. Consola Magnavox Odyssey

En el mismo año, 1971, Nolan Bushnell y Ted Dabney crean Computer Space que fue la primera arcade. Computer Space no utiliza microprocesador, RAM o ROM. El sistema está basado en una

máquina de estados finitos, hecha de circuitos integrados de tecnología TTL (lógica transistor-transistor). Los elementos gráficos están contenidos en arreglos de diodos. La configuración está hecha de hasta tres circuitos impresos interconectados por un *bus* común. El despliegue es *renderizado* en una televisión de tubos de vacío en blanco y negro.

La comercialización de las máquinas arcade se encuentra estrechamente relacionada con las máquinas de *pinball* y otro tipo de juegos "tragamonedas". Los primeros arcade funcionaban en computadoras con el único propósito de correr un juego específico. Estos juegos se distribuían a lugares públicos de diversión, centros comerciales, restaurantes, bares, o salones recreativos.

A medida que las arcade se vuelven más rentables, se incrementa la variedad de juegos y se dispara la cantidad de locales con máquinas arcade. La industria de los arcade comienza su decadencia a mediados de 1982.

Desde 1975 el mercado de las consolas de videojuegos empieza a crecer hasta tomar el lugar de los arcade en la década de los 80´s.

La empresa Atari se forma en 1972, iniciada por Nolan Bushnell y Ted Dabney. Su primer mercado fueron las máquinas arcade, en este mercado tuvieron un éxito inmenso desarrollando juegos como Pong, Breakout, Maneater, entre otros.

En 1975 Atari lanza al mercado su primera consola de juegos, Home Pong. El hardware es hecho específicamente para funcionar con el juego Pong, aun así revive el mercado de consolas caseras que permanecía desapercibido desde la aparición de Odyssey.

En 1976 Coleco lanza su consola Telestar, únicamente para tenis de televisión. Fairchild Camera and Instruments lanza la primer consola con la capacidad de cambiar juegos mediante cartuchos.

Un año después Atari lanza su segunda consola, Video Computer System, también conocida como Atari 2600. La consola vendió 30 millones de unidades, una cifra solo superada hasta 1983 por Nintendo Entretainment System.

Atari definió la industria del entretenimiento por computadora de los años 70's a los 80's.

Durante el periodo de 1977 a 1982 surgen una gran cantidad de consolas y videojuegos, pero las ventas obtenidas por sistemas posteriores al Atari 2600 no fueron extraordinarias. El mercado de videojuegos se considera muerto, nadie quiere invertir ni comercializar en ese momento.

En 1983 se comienza a distribuir el NES, para el asombro de la industria, la cantidad de consolas vendidas marca un nuevo récord y revive el mercado de los videojuegos y consolas. A partir de entonces han surgido una gran cantidad de consolas comerciales, en la actualidad el mercado de las consolas se encuentra dominado principalmente por Nintendo, Microsoft y Sony.

2.3 Géneros de juegos

Aventura

Los juegos de aventura tienen una secuencia basada en una historia. La acción en el juego avanza a medida que se progresa en la historia.

Acción

Son juegos donde el jugador debe reaccionar rápidamente a los cambios que suceden en la pantalla. Los jugadores toman decisiones rápidas y usan los reflejos. Se pueden subclasificar en:

- Peleas
- *Shooters*

Este tipo de géneros requieren menos resolución de problemas que los géneros de aventura, estrategia o puzzles.

RPG's

En esta clasificación los juegos tienen como base el incremento gradual de las habilidades y fuerza de los personajes. Es uno de los géneros donde existe la mayor posibilidad de modificar a los personajes de acuerdo a los gustos del jugador.

Estrategia

Los juegos de estrategia consisten en el uso de recursos limitados para el logro de un determinado objetivo. El manejo de recursos consiste en la decisión de crear cierto tipo de unidades y la forma de ponerlas en acción.

Simulación

Son juegos que tratan de emular las condiciones y operaciones en el mundo real de maquinarias, helicópteros, ciudades, etc.

Deportes

Permite al jugador manejar equipos de un deporte específico. Pueden manejar jugadores, entrenadores o todo el equipo. Estos juegos deben reproducir las reglas y estrategias del deporte.

Casuales

Consisten en juegos tradicionales que son sencillos, cortos y familiares.

Educacionales

Su objetivo es educar mientras entretienen. Los jugadores son por lo general muy jóvenes y se tiene que asegurar que el contenido es adecuado para la edad.

Puzzles

Proporcionan un desafío intelectual. Por lo común los puzzles se utilizan con otros géneros como aventura, rpg's, etc.

MMOG's

Su característica principal es que se juegan completamente en red y las comunidades son gigantescas.

2.4 MMOG

El género de los MMOG está compuesto por juegos que soportan una gran cantidad de jugadores simultáneos en un mundo virtual persistente (continua existente sin importar la conexión del jugador) de amplias dimensiones.

Dentro de esta categoría tenemos subgéneros:

- MMORPG
- MMORTS
- MMOFPS

Los MMOG utilizan servidores para mantener el estado del mundo. Los clientes, jugadores, se conectan al servidor para recibir la información correspondiente al estado del juego y participar en línea.

Este género surge a principios de los años 80's con juegos como Rogue y Dungeon (para la PDP-1), AirWarrior (1986). Juegos como Ultima Online, Ever Quest y Neverwinter Nights popularizaron esté género en los años 90's. World of Warcraft actualmente domina los juegos MMOG, tiene de 11 a 12 millones de subscriptores mensuales y ha generado más de 2.2 billones de dólares en suscripciones.

2.5 Serious games

Los videojuegos han tenido un gran éxito y proliferación desde sus inicios, no hay duda que forman parte de nuestra cultura al igual que los libros, películas, televisión y otros medios de

entretenimiento. Al igual que estos medios, los videojuegos tienen el potencial de ser no solo herramientas de entretenimiento (D. Michael, 2006).

Los *serious games* son juegos cuyo principal objetivo no es el entretenimiento, es la educación, exploración y administración. Esto no quiere decir que los serious games no son, o no deberían ser, entretenidos; al contrario, hacer uso del entretenimiento como herramienta adicional para la enseñanza facilita el aprendizaje.

El uso de medios de entretenimiento para lograr el aprendizaje no está limitado a los videojuegos; en las películas y televisión podemos encontrar documentales; en los libros encontramos lecciones de ética, historia, etcétera.

La iniciativa de Serious Games (Initiative, 2002) es reciente y relativamente nueva para muchas personas. Un término más conocido es edutainmente, o educación por medio del entretenimiento, surge aproximadamente por 1990 con la aparición de las computadoras personales. Por lo general, edutainmente se refiere a la educación que también busca entretener, específicamente a niños; en cambio, serious games abarca una educación más general y de diferentes edades.

El desarrollo de un videojuego es similar al proceso de desarrollo de un serious game. Algunas de las diferencias radican en:

- los diseñadores de juego deben ajustar la diversión dependiendo del producto.
- se debe considerar el uso de equipos de hardware menos sofisticado.
- por lo general son juegos pequeños que requieren menos tiempo e inversión.

Los mercados principales de serious games son:

- Militares: Actualmente la mayor fuente de recursos para desarrollo e innovación de serious games proviene de esta área. Se utilizan para enseñar estrategia y tácticas.
- Gobierno: Existe una gran variedad de serious games desarrollados para este sector, algunos son de entrenamientos para situaciones de estrés como terremotos o incendios.
- Educación: Los serious games permiten el uso de herramientas adicionales que facilitan el aprendizaje en diferentes niveles de educación.
- Corporaciones: Facilitan el entrenamiento y aprendizaje de equipo especializado y procedimientos.
- Salud: Se utilizan para recuperación, rehabilitación y salud mental.

2.6 La producción de un videojuego

El proceso de desarrollo de un videojuego puede durar de días hasta años, igualmente los equipos pueden estar integrados por una o cientos de personas, dependiendo de la complejidad del juego.

A pesar de estas grandes diferencias, en la industria existen una serie de fases bien definidas que se han vuelto estándar para el desarrollo de videojuegos. Estas fases son las siguientes: desarrollo de concepto, preproducción, producción, pruebas alfa y pruebas beta.

2.6.1 Desarrollo de concepto

Esta fase consiste en el decidir de qué se trata el juego y describirlo claramente en texto para que cualquier persona pueda entenderlo, tanto miembros del equipo como posibles inversionistas. Se definen los elementos novedosos del juego, conceptos de arte y creación de historia.

En esta etapa, el equipo de desarrollo es pequeño y puede estar formado por un diseñador de juego, un líder de tecnología, líder de arte y un productor. Se deben completar los siguientes documentos:

- Alto concepto: Es una descripción corta y atractiva del juego.
- Propuesta de juego: Resumen de que se trata el juego, porque será exitoso y como recaudará ingresos.
- Documento de concepto: Documento extenso y muy descriptivo, contiene el alto concepto, género del juego, mercado al que va dirigido, como se juega, características del juego, historia, descripción del mundo donde se lleva a cabo, plataformas de desarrollo, tiempo estimado de desarrollo, presupuesto, análisis de competencia, análisis de riesgos, y equipo de desarrollo.

2.6.2 Preproducción

El objetivo de la preproducción es completar el diseño del juego, establecer el plan de producción y la creación de un prototipo. En resumen, se demuestra que el equipo puede desarrollar el juego y vale la pena hacerlo.

Al igual que en la etapa de desarrollo de concepto, se crean los siguientes documentos para ser usados en la etapa de producción:

- Documento de diseño: Este documento describe exhaustivamente los detalles de todo lo que sucede en el juego. Debe incluir información de cómo se juega, interfaz de usuario, historia, personajes, monstros, inteligencia artificial, armas, escenarios, etcétera. Las descripciones en este documento son los requerimientos para realizar los documentos del plan técnico y el plan de producción de arte.

- Plan de producción de arte: Se completa el “art bible” y el proceso de creación de arte.

El diseñador, líder de arte y artistas de concepto colaboran para llenar el "art bible" que define el estilo del juego y sirve como guía a los artistas que se integren.
También se define el proceso de creación de arte que involucra los pasos que se llevan para pasar de un concepto de arte, a un modelo 3D, texturizarlo, animarlo y agregarlo al juego.

- Plan Técnico: Se describe el plan de pasar el diseño del juego a software. Establece las tareas que realizará el equipo y estima el tiempo para completarlas. El plan técnico define las herramientas que se usarán para desarrollar los juegos (tanto comerciales como propios), hardware y software que se debe adquirir, y descripción técnica de los elementos del juego.

- Plan de desarrollo: Define el calendario de desarrollo, tareas y dependencias, plan de recursos, presupuesto y las principales fechas de entrega (*milestones*).

El resultado de la etapa de preproducción es uno o varios prototipos que capturan la esencia del juego y demuestra que los objetivos son alcanzables.

2.6.3 Producción

En ésta etapa se comienza el desarrollo del juego con las tareas definidas en el plan técnico y con la ayuda de los documentos escritos en preproducción.

2.6.4 Pruebas alfa

Por lo general, las pruebas alfa se realizan sobre un producto que se puede jugar de principio a final. Las pruebas se realizan con jugadores externos al equipo de desarrollo. El objetivo es pulir el juego, quitar *bugs* y eliminar características del juego que no se terminarán a tiempo.

2.6.5 Pruebas beta

El objetivo de estas pruebas es estabilizar el proyecto y eliminar la mayor cantidad de bugs antes de liberar el juego al mercado. Los cambios son mínimos y estos deben a los estándares de los desarrolladores de la consola y los productores.

3. Motores de juegos

3.1 Introducción a los motores de juegos

Un motor de juegos es un conjunto de sistemas de software extensibles y que facilita el desarrollo de un videojuego sin una gran modificación de código. Los motores de juego pueden ser:

- Motores de juego de propósito específico: Se desarrollan específicamente para un género de juegos.
- Motores de juego de propósito general: Abarcan un amplio género de juegos, su desarrollo tiene una mayor complejidad.

Desafortunadamente no existe un motor que pueda desarrollar cualquier tipo de juego, ya que se realizan optimizaciones específicas a los géneros y a las diferentes plataformas de hardware.

El término motor de juegos surge a mediados de los 90's como consecuencia de los juegos de primer persona como el increíblemente popular juego Doom por "id Software". Doom tenía una arquitectura que marca la separación entre componentes de software, recursos gráficos y reglas de juego. Otros desarrolladores podían comprar una licencia de los componentes de software y crear sus propios personajes, arte, interfaces, niveles, etcétera.

El valor de esta separación fue evidente cuando se desean crear juegos similares mediante la modificación del contenido gráfico y realizando un cambio mínimo a los componentes de software. De aquí también surge el término *modding*, donde grupos de jugadores, o desarrolladores independientes, agregan contenido a un juego.

La creación de un motor de juegos es un proceso complicado y largo, en la actualidad existen motores comerciales que venden licencias para permitir crear juegos con componentes de software ya existentes.

3.2 Desarrollo de un motor de juegos

El desarrollo de un motor de juegos es equivalente al desarrollo de un sistema informático. Un motor de juegos bien diseñado tiene las siguientes características (J. Flymnt, 2005; M. Robert, 2006; Sommerville, 2005):

- Mantenibilidad: El motor debe escribirse de tal manera que pueda evolucionar para cumplir con las necesidades de los creadores de juegos. Es inevitable un cambio, debido al uso de nuevas tecnologías y las crecientes expectativas por parte de los jugadores.

- Portabilidad: Involucra la facilidad de usar el motor en diferentes computadoras.

- Confiabilidad: La confiabilidad tiene un gran número de características, incluyendo la fiabilidad, protección y seguridad. El software seguro no debe causar daños en caso de una falla del sistema.

- Eficiencia: El motor no debe hacer que se malgasten los recursos del sistema, como la memoria y ciclos de procesamiento. Por lo tanto, la eficiencia incluye tiempos de respuesta y de procesamiento, utilización de la memoria, etcétera.

- Usabilidad: El motor de juegos debe ser fácil de utilizar para el desarrollo de juegos. Esto significa que debe tener una interfaz de usuario apropiada y una documentación adecuada.

- *Testability*: El motor de juegos debe buscar mantener sus componentes aislados, de esta manera se facilitan las pruebas de software.

Algunos síntomas de un diseño mal elaborado (M. Robert, 2006) son:

- Rigidez: Es la tendencia del software a hacer difícil el proceso de cambio. Un cambió mínimo produce una necesaria cascada de cambios en módulos dependientes. Esto genera una resistencia al cambio y el software se vuelve rígido.

- Fragilidad: Este síntoma está relacionado con la rigidez. Es la tendencia a romper el sistema cada vez que se realiza un cambio. Esto genera que el software se vuelve imposible de mantener; cada solución hace peor el problema, introduciendo nuevos errores.

- Inmovilidad: Es la incapacidad de reutilizar módulos para operaciones similares. El módulo necesario está escrito con demasiadas dependencias y resulta más fácil reescribirlo que utilizarlo.

- Viscosidad: En un proyecto viscoso el diseño del sistema es difícil de mantener. La viscosidad de software surge cuando existen diferentes maneras de realizar un cambio al software. Cuando es más difícil implementar un cambio manteniendo el diseño, que realizarlo mediante uno que no mantiene el diseño, un “hack”, es sencillo tomar la decisión inadecuada.

- Complejidad Innecesaria: La complejidad innecesaria surge cuando se agregan facilidades al código que anticipan cambios en los requerimientos. Esto puede parecer algo positivo,

ya que permite al código ser flexible; sin embargo, el efecto es el opuesto, el código se vuelve más complejo y difícil de entender.

- Repetición Innecesaria: El trabajo de cambiar el sistema se vuelve arduo cuando existe código repetido, además, es un indicador de una falta de abstracción.

- Opacidad: Es la tendencia de un módulo de volverse difícil de entender.

Estos síntomas pueden ser causados por las siguientes razones:

- En un proyecto no ágil, el diseño degrada si los requerimientos cambian.
- Cambios al código que no mantienen la filosofía del diseño inicial.
- Un mal diseño de arquitectura y de los principios de diseño orientados a objetos.
- Un análisis inadecuado de los requerimientos.

Aunque existen muchos modelos para el desarrollo de software como son: cascada, iterativo, ágiles, entre otros, algunas actividades son comunes para todos:

1. Especificación del Software: Se define la funcionalidad y las restricciones de su operación.
2. Diseño del sistema: Establece una arquitectura completa del sistema, identifica y describe las abstracciones fundamentales.
3. Implementación y validación: Se produce el software y se asegura que hace lo que el cliente desea.
4. Evolución del software: El software evoluciona para cubrir las necesidades cambiantes del cliente.

3.2.1 Requerimientos para el motor de juegos

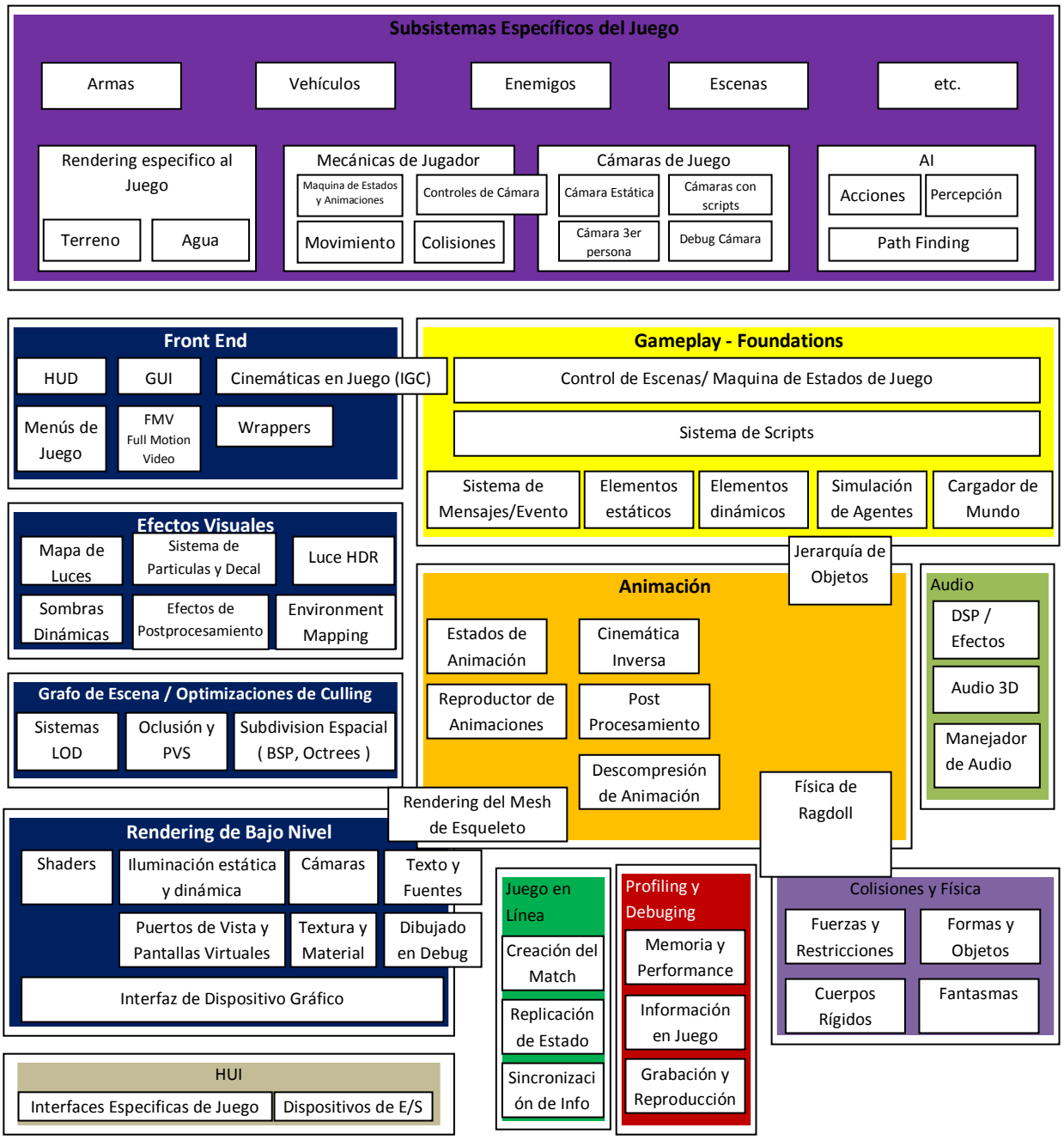
Algunas características que se deben definir antes de comenzar con el diseño del motor de juegos son:

1. Definir los géneros de juegos para los cuales se desarrollará.
2. Características funcionales y no funcionales de los juegos donde se utilizará.
3. Definir las áreas de funcionalidad del motor de juegos, como son: gráficos, audio, red, animaciones, etcétera.

3.2.2 Diseño de Arquitectura

Los grandes sistemas siempre se descomponen en subsistemas que proporciona algún conjunto de servicios relacionados (Sommerville, 2005). Algunos subsistemas del motor de juegos son llamados motores debido a su complejidad y extensión.

La presenta un diseño arquitectónico en capas de un motor propuesto por J. Gregory (J.Gregory, 2009):



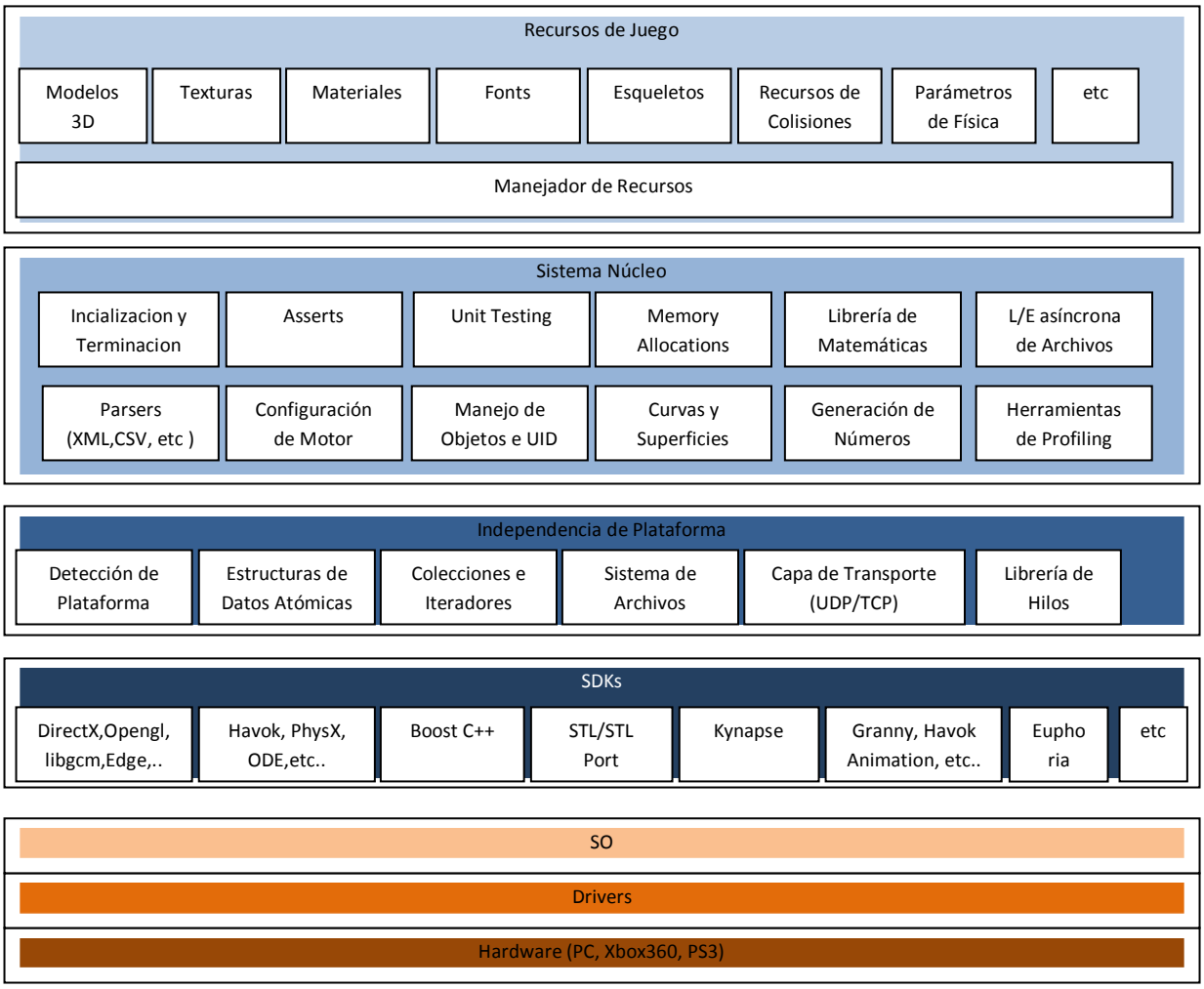


Figura 3.1. Arquitectura en capas de un motor de juegos

3.2.3.1 Hardware objetivo

Representa la computadora o consola destino del juego. Es importante tener en cuenta el dispositivo destino ya que se debe considerar la memoria, procesador, resolución, etcétera.

3.2.3.2 Drivers

Software de bajo nivel proporcionado por el sistema operativo o por el vendedor del hardware. El driver proporciona una abstracción del hardware y proporciona una interfaz para controlarlo.

3.2.3.3 Sistema operativo

En una computadora, el sistema operativo siempre se encuentra presente. Controla la ejecución de múltiples programas (entre ellos los juegos) y la administración de memoria.

En una consola, el sistema operativo es una librería pequeña que se compila dentro del ejecutable del juego. En las consolas, el juego prácticamente es dueño del sistema. Sin embargo, no es el caso en el Playstation 3 ni en el Xbox 360. El sistema operativo puede detener la ejecución del juego o utilizar ciertos recursos.

3.2.3.4 SDKs

La mayoría de los motores de juegos utilizan SDK de terceros y middleware. Un SDK (Software Development Kit) es una colección de librerías, API's y herramientas que le permite a un programador crear aplicaciones para un sistema concreto.

La Figura 3.2 muestra algunos ejemplos:

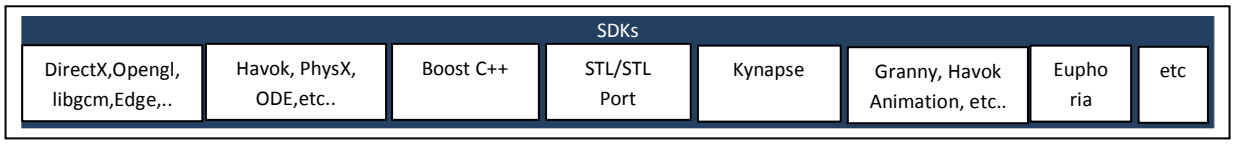


Figura 3.2. SDK's de terceros

Estructuras de Datos y Algoritmos

Como cualquier otro sistema de software, dependemos ampliamente de las estructuras de datos y algoritmos para manipularlos. Algunos ejemplos de librerías son:

- STL. La biblioteca de plantillas estándar implementa muchas estructuras de datos comunes y algoritmos para procesarlas, strings y streams de E/S.
- STLport. Es una versión portable y optimizada de STL.
- Boost. Es una librería con componentes poderosos y algoritmos, diseñada al estilo de STL.

Gráficos

La mayoría de los motores de *rendering* (dibujado a pantalla) se construyen sobre algunas de las siguientes librerías de interfaz de hardware:

- OpenGl fue inicialmente desarrollada por SGI. Es un estándar abierto utilizado en una amplia gama de dispositivos de hardware.
- DirectX es el SDK 3D de Microsoft, principal rival de OpenGL.
- Libgcm es una alternativa OpenGl que fue proporcionada por Sony. Es una interfaz de bajo nivel al hardware gráfico RSX del Playstation 3.

Colisiones y Física

Es un SDK para la detección de colisiones y física de cuerpos rígidos. Algunos de los SDK más conocidos son:

- Havok es muy popular en la industria y es considerado un motor de física y colisiones.
- PhysX es desarrollado por NVIDIA y su uso es gratuito.
- Open Dynamics Engine (ODE) es un SDK de código abierto.

Animación

- Granny incluye exportadores de modelos y animaciones de los paquetes de modelado 3D y animación más comerciales: Maya, 3D Studio Max, etc.
- Havok Animation la compañía desarrolladora del motor de física Havok que facilita la interacción entre animaciones y física.
- Endorphin y Euphoria son paquetes de animación que producen animaciones avanzadas usando modelos biomecánicas del movimiento humano.

Inteligencia Artificial

- Kynapse. La inteligencia artificial era desarrollada específicamente para cada juego, pero, la compañía Kynogon desarrollo Kynapse que contiene bloques de inteligencia artificial como son: búsqueda de caminos, obstrucciones (object avoidance), identificación de vulnerabilidades.

3.2.3.5 Independencia de plataforma

La mayoría de los motores de juegos deben correr en más de una plataforma de hardware, es por esto que la mayoría de los motores de juegos contienen una capa de Independencia de Plataforma.

Aseguramos una independencia de plataforma al remplazar o englobar las llamadas más comunes a funciones de C, llamadas de sistema operativo y otras API's.

Figura 3.3. Capa de independencia de plataforma

3.2.3.6 Sistema Núcleo

Contiene librerías y utilidades de software sobre las cuales se extiende la funcionalidad de nuestro motor, algunas son:

- Los asserts son líneas de código que nos permiten verificar errores lógicos y/o suposiciones que no se cumplen.
- Manejo de Memoria. Permite un uso rápido y eficiente de la liberación y asignación de memoria.
- Librería de Matemáticas. Los juegos usan una gran cantidad de funciones matemáticas. Estas librerías nos proporcionan herramientas como operaciones con vectores, matrices, quaterniones, rotaciones, trigonometría, etc. Además, las librerías pueden estar optimizadas.

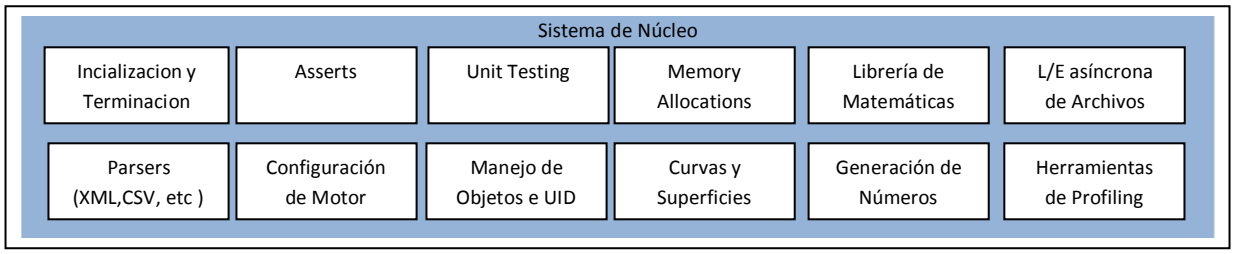


Figura 3.4. Capa de sistema de núcleo

3.2.3.7 Recursos de Juego

El motor de juegos debe ser capaz de administrar y cargar diferentes tipos de recursos como: texturas, modelos, fuentes, audio, video, etcétera.

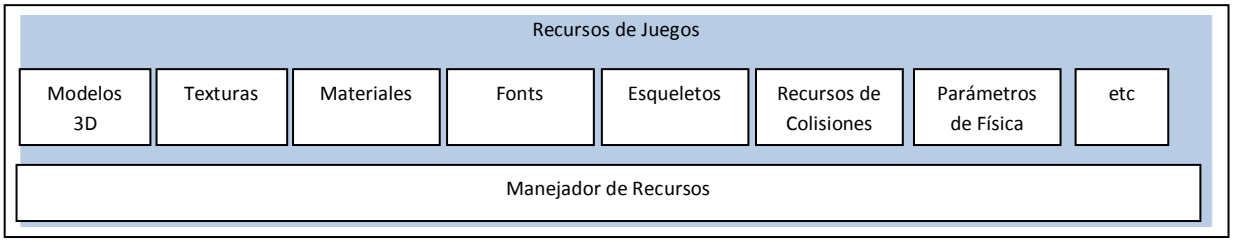


Figura 3.5. Capa de manejo de recursos

El manejador de recursos es el encargado de:

- Debido a que la memoria es limitada, debe asegurarse que los recursos de juego no se repitan y se encuentren presentes en el momento de su uso.
- Proporcionar las interfaces para al acceso a los recursos del juego.
- Envolver las llamadas de acceso al sistema de archivos y proporcionar funciones de alto nivel.
- Manejo de diferentes medios de almacenamiento como discos duros, usb, dvd, archivos en red, etc.

En algunos motores el manejador de recursos es un sistema unificado y centralizado que maneja todo tipos de recursos (Unreal, Ogres Resource Manager). En otros motores el manejador de recursos no existe como un sistema centralizado, al contrario, se conforma de diferentes subsistemas que manejan recursos específicos.

3.2.3.8 Motor de Render

El motor de *render* es uno de los más grandes y complejos componentes en cualquier motor de juegos. El diseño del motor de rendering depende del hardware de gráficos y la librería que se utiliza para interactuar con él.

3.2.3.8.1 Rendering de Bajo Nivel

El objetivo de este componente es dibujar una colección de primitivas los más rápido y visualmente atractivo posible, sin importar que partes de la escena son visibles.

Figura 3.6. Subsitema de rendering de bajo nivel

- Interfaz de Dispositivo Gráfico. Nos facilita la inicialización y selección del dispositivo gráfico, además nos permite la creación de superficies de rendering (depth buffer, color buffer, esténcil buffer, etcétera), entre otras cosas.

Los otros componentes de la capa de rendering realizan un trabajo en cooperación para obtener los datos finales que serán enviados al hardware gráfico, esto incluye triángulos, materiales, textura, matrices de cámara, proyección, etc. Además, se busca mandar esta información a la tarjeta de gráficos de manera eficiente.

3.2.3.8.2 Grafo de Escena / Optimizaciones de Culling

Es un componente del motor de rendering que optimiza la cantidad de primitivas que se mandan al Rendering de Bajo Nivel. Esto se logra al descartar los objetos no visibles (están obstruidos o fuera del área de vista).

Existen diferentes maneras de lograrlo: se puede utilizar un frustrum culling, subdivisión espacial (árbol de BSP, quadtree, octree, kd-tree, etcétera), niveles de detalle LOD.

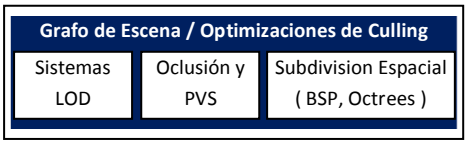


Figura 3.7. Un típico grafo de escena y subdivisión espacial para optimizaciones de culling

3.2.3.8.3 Effectos Visuales

Los actuales motores de juegos soportan una gran variedad de efectos visuales, algunos son:

- Sistemas de partículas para efectos de disparos, humo, agua, etcétera.
- Sistemas de decal para efectos de huellas, hoyos de balas, etcétera.
- Mapas de iluminación y de ambiente para simular el efecto de la iluminación.
- Sombras dinámicas que se pueden logran con shadow mapping, shadow volumes, etcétera.
- Efectos de postprocesamiento de la escena, como blur, bloom, etcétera.

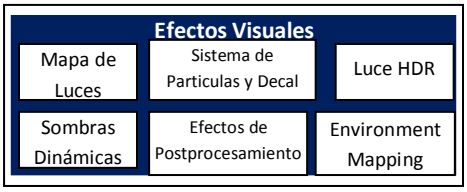


Figura 3.8. Subsistema de efectos visuales

Algunos efectos se pueden implementar dentro del motor de rendering o como un componente adicional que afecta los buffers de salida.

3.2.3.8.4 Front End

La mayoría de los juegos despliega imágenes 2D sobre una escena 3D para mostrar información al jugador. Algunos elementos que se despliegan son:

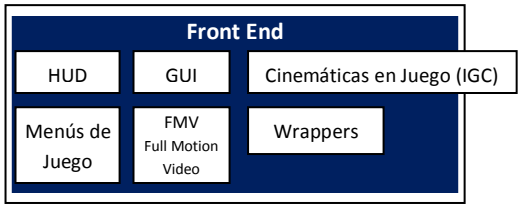


Figura 3.9. Subsistema de Front End

- El HUD (Heads Up Display) muestra información de vidas, armas, puntaje, etcétera.
- Menús en el juego
- Puede tener un GUI (Graphical Users Interface) que permite al jugador modificar inventario, configurar unidades, etcétera.
- IGC's, Cinemáticas en Juego, permiten mostrar animaciones dentro del juego.

3.2.3.9 Herramientas de profiling y debugging

Los juegos son sistemas de tiempo real, es importante contar con herramientas que nos permitan evaluar la eficiencia del sistema para realizar las optimizaciones pertinentes. Es importante evaluar el uso de memoria (heap y stack), tiempo de ejecución del juego (CPU y GPU).

Figura 3.10. Subsistema de profiling y debugging

Algunas herramientas de profiling son:

- CLR Profiler permite ver el uso de memoria heap y stack para sistemas desarrollados con el .NET Framework.
- PIX es un debuger y analizador de performance de CPU y GPU, para DirectX.
- JetBrains es una empresa que desarrolla software para el análisis de memoria y performance para diferentes lenguajes de programación.
- Intel's VTune

La mayoría de los motores incorpora su propio sistema de profiling y debuging. Algunos elementos útiles que vale la pena tener son:

- Mecanismo para evaluar tiempos de ejecución de secciones de código.
- Posibilidad de mostrar en la pantalla las estadísticas de profiling mientras el programa se encuentra en ejecución.
- Guardado de resultados de profiling a archivos de texto o Excel.
- Mecanismos para determinar el uso de memoria por el motor y subsistemas.
- Mecanismos para guardar eventos del juego y la posibilidad de reproducirlos.

3.2.3.10 Colisiones y física

La detección de colisiones es importante para todos los juegos, ya que es la forma de interactuar con el mundo virtual de manera realista. El sistema de física también es llamado motor de física debido a su complejidad y extensión.

Los motores de física por lo común usan dinámica de cuerpos rígidos, ya que únicamente nos encontramos interesados en la cinemática de cuerpos rígidos y las fuerzas que causan el movimiento.

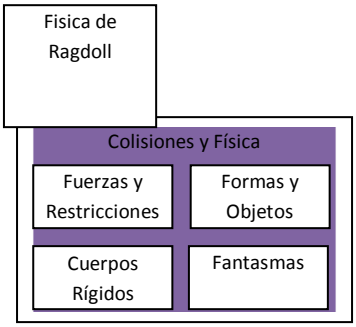


Figura 3.11. Subsistema de colisión y física

En la actualidad pocas empresas desarrollan su propio motor de física, lo común es integrar un SDK al motor, algunos son: Havok, PhysicsX de NVidia.

3.2.3.11 Animaciones

La mayoría de los juegos presentan personajes animados o inanimados que interactúan con el mundo virtual. El uso de animaciones con los personajes da una mayor atracción visual y mantiene el interés del jugador.

Algunos tipos de animaciones utilizadas en juegos son:

- Animaciones por sprites/texturas 2D.
- Animaciones por jerarquías de cuerpos rígidos.

- Animaciones por esqueleto.
- Animaciones de vértices.
- Morphing.

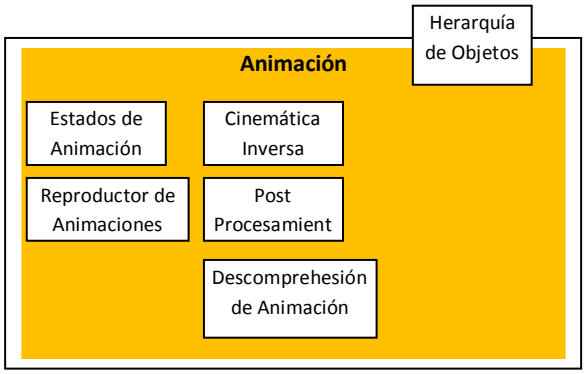


Figura 3.12. Subsistema de animación

3.2.3.12 Dispositivos de Interacción con usuario

Human Interface Device, HID, procesa la entrada por parte del jugador, algunos dispositivos incluyen: teclado, mouse, joystick, controles, otros dispositivos especializados como volantes, wiimote, etcétera.

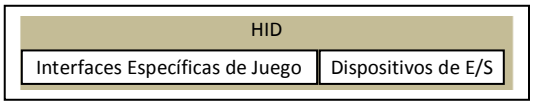


Figura 3.13. Subsistema de interacción con usuario

El subsitema de HID, por lo común, maneja detalles de bajo nivel de los dispositivos y proporciona información de alto nivel como: botones presionados, botones no presionados, interpreta la información de los acelerómetros, información de dirección de joystick, etcétera; además, puede tener lógica para manejar secuencias de botones, presión de múltiples botones, *gestures* (movimientos específicos en touchpads).

3.2.3.13 Audio

El audio es un subsistema que por lo general recibe menor atención en un motor de juegos. Para plataformas de DirectX (PC y Xbox 360) Windows proporciona una herramienta muy útil llamada XACT. Electronic Arts desarrollo un motor de audio llamado SoundR!iot.

Figura 3.14. Subsistema de audio

En caso de que se use un motor de audio ya existente, aún se necesita desarrollo de software para la integración, ajustes y detalles para producir sonido de alta calidad.

3.2.3.14 Multijugador/Red

El uso de multijugadores pude tener un profundo impacto en algunos componentes del motor de juegos. Es importante considerar esta funcionalidad durante el diseño del motor.

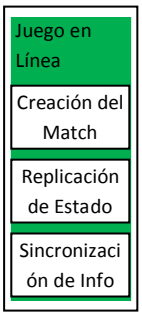


Figura 3.15. Subsistema de juego en red

3.2.3.15 Sistemas de Gameplay - Foundations

El termino gameplay significa la acción que sucede dentro del juego, las reglas del mundo virtual, la mecánica del jugador (sus habilidades) y otras entidades en el mundo, y los objetivos del jugador.

El gameplay por lo común se implementa en el leguaje con que se desarrolla el motor, un lenguaje de scripting o una combinación de ambos. Para separar el código del motor con el juego podemos introducir la capa de **Gamplay – Foundations** que proporciona los componentes sobre los cuáles la lógica específica del juego se puede implementar.

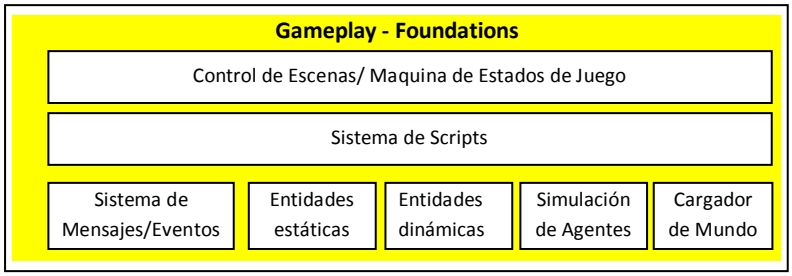


Figura 3.16. Subsistema de Gameplay-Foundations

Algunos componentes de este subsistema son:

- Entidades estáticas y dinámicas. Las entidades del mundo virtual (objetos animados e inanimados) son modelados con la programación orientada a objetos. El modelo de entidades permite tener una colección heterogénea de objetos y facilita su uso mediante polimorfismo.

  Algunas entidades típicas pueden incluir:

  - Entidades estáticas como rocas, edificios, terreno, fondo, etcétera.
  - Entidades dinámicas que utilizan cuerpos rígidos para interactuar con el usuario.
  - Personaje principal (Player Character, PC).
  - Personajes no controlados por el usuario (NPC, Non player characters).
  - Armas
  - Cámaras

- El modelo de objetos define las características de las entidades usadas, algunas son: identificadores únicos para cada objeto, tiempo de vida de los objetos, simulación de los estados de los objetos, si están diseñados con programación orientada a objetos.

- El sistema de Mensajes y Eventos establece como se realiza la comunicación entre objetos. Un objeto puede simplemente llamar una función de otro objeto para mandar el mensaje, en cambio, en un motor que usa un sistema de mensajes y eventos, un objeto crea un nuevo mensaje que posteriormente se entrega al destinatario.

- El sistema de scripts evita la necesidad de recompilar y “re-linkear” cada vez que se hace un cambio a los scripts del juego. Un script puede tener información de la lógica del juego, estados de personajes, descripción de mundos, etcétera.

- El control de escenas permite agregar nuevas escenas como: menú, instrucciones, opciones, etcétera. La máquina de estados facilita la creación de estados para las entidades del juego.

3.2.3.16 Sistemas Específicos del Juego

Los sistemas específicos del juego son diferentes para cada juego, esta es una capa que separa el juego del motor de juegos. En la práctica, esta capa nunca es perfectamente clara ya que algunos elementos se pueden encontrar en los sistemas de gameplay o ya en el juego.

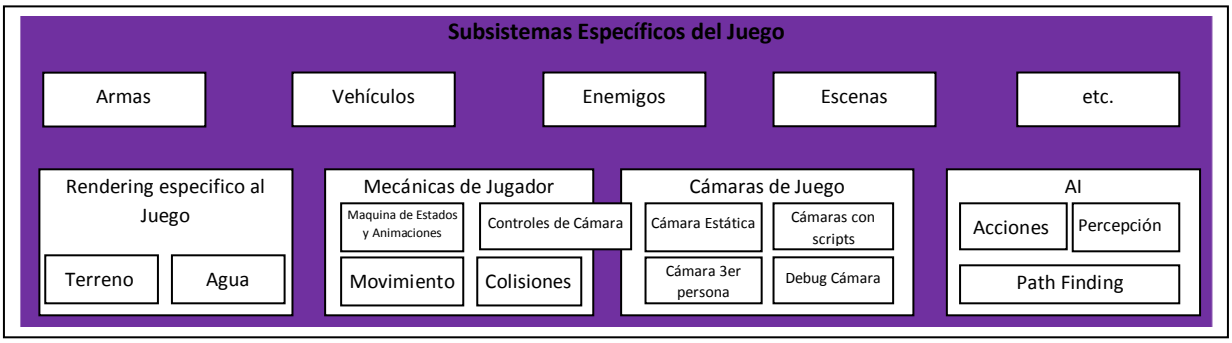


Figura 3.17. Sistema específico de juego.

3.2.3 Proceso de Diseño Orientado a Objetos

Antes de comenzar a desarrollar el motor de juegos, se deben identificar los objetos principales del sistema y desarrollar modelos de diseño basados en el diseño de arquitectura. Los modelos de diseño son el puente entre los requerimientos y la implementación del sistema, deben ser abstractos para que el detalle innecesario no oculte las relaciones entre objetos. Sin embargo, también deben incluir suficiente detalle para que los programadores tomen las decisiones de implementación.

Para desarrollar los modelos e implementarlos es importante hacer uso de los principios de diseño de clases orientadas a objetos y conocer patrones de diseño.

3.2.3.1 Principios de Diseño de Clases Orientadas a Objetos

- Single Responsibility Principle (DeMarco, 1979)

El single responsability principle, también llamado cohesión, define que una clase debe tener una sola razón de cambio. Es importante que una clase tenga una sola responsabilidad, debido a que cada responsabilidad es susceptible a un cambio por modificación de requerimientos.

Si una clase tiene más de una responsabilidad, las responsabilidades se vuelven dependientes y cualquier cambio a alguna de ellas puede afectar a la otra.

- Open Closed Principle (M. Robert, 2006)

Entidades de software (clases, módulos, funciones, etcétera) deben estar abiertas para extensión pero cerradas para modificación. Este principio puede sonar contradictorio; sin embargo, existen técnicas de programación basadas en abstracción, polimorfismo y encapsulamiento que nos permiten lograrlo.

- The Liskov Substitution Principle (Data Abstraction and Hierarchy, 1998)

Las clases derivadas deben poder sustituir a la clase base. Esto quiere decir la clase derivada debe funcionar de manera similar a la clase base y si alguna clase derivada no proporciona la misma funcionalidad que la clase base es mejor crear una nueva clase.

- The Dependency Inversion Principle (M. Robert, 2006)

Es la estrategia de depender de funciones y clases abstractas o interfaces, en vez de concretas. Además, los módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones.

Un modelo inapropiado está representado por la Figura 3.19. La capa de alto nivel *Policy* depende de la capa de bajo nivel *Mechanism*, a su vez, Mechanism depende de la capa de *Utility*. Es inapropiado porque un cambio en las capas bajas requiere modificar todas las capas de mayor nivel.

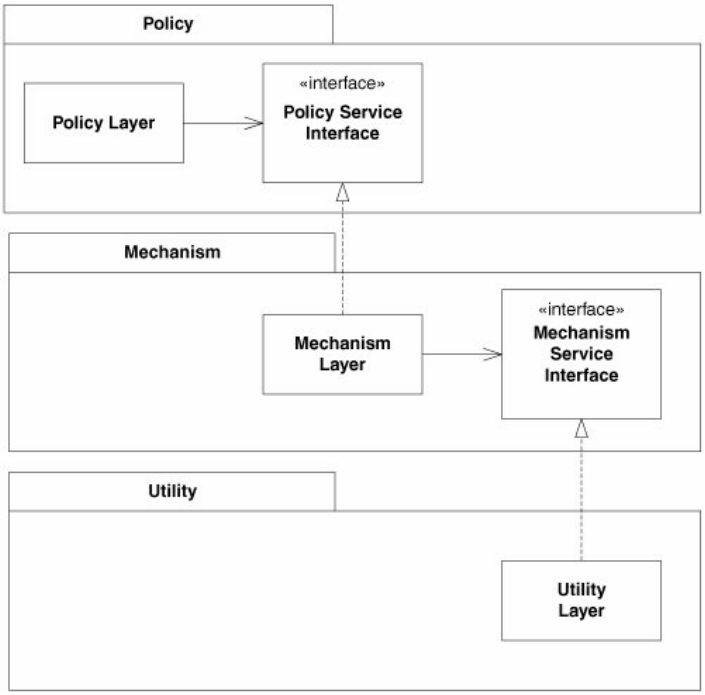


Figura 3.18 Modelo donde se usa el Dependency Inversion Principle

Un modelo que cumple con el Dependecy Inversion Principal está representado por la Figura 3.18. Cada capa de alto nivel declara una interfaz de los servicios que requiere. Las clases de bajo nivel implementan la interfaz y cualquier cambio en éstas no afecta a las capas altas.

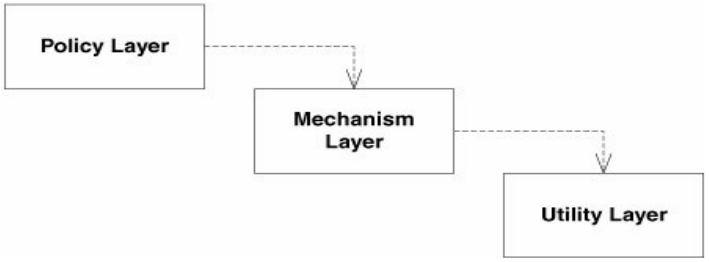


Figura 3.19. Modelo donde las clases de alto nivel depende de las clases de bajo nivel

- The Interface Segregation Principle (M. Robert, 2006)

El *interface segregation principle* asume que existen clases que no pueden cumplir con cohesión, son *fat* interfaces, y se pueden descomponer en grupos de métodos.

Este principio sugiere que los clientes no deben conocer los métodos como una sola clase, en cambio, es conveniente que los clientes conozcan las clases abstractas que tienen interfaces cohesivas, por ejemplo:

La Figura 3.20 muestra que las clases Deposit Transaction, Withdrawal Transaction y Transfer Transaction requieren de la interfaz UI para realizar la transacción adecuada. La interfaz UI es una fat interface y es precisamente lo que ISP nos recomienda evitar. Si creamos una nueva transacción, se deben agregar los métodos a UI y esto afectará a todas las clases de transacción.

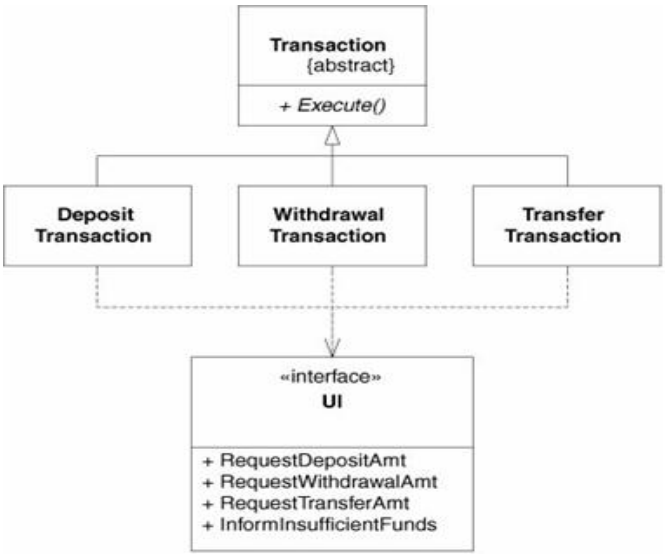


Figura 3.20. Fat interface

Un modelo que cumple con el Interface Segregation Principle se muestra en Figura 3.21. La fat interface UI se separa en interfaces individuales: DepositUI, WithdrawalUI y TransferUI. Además, cada transacción depende de su interfaz específica.

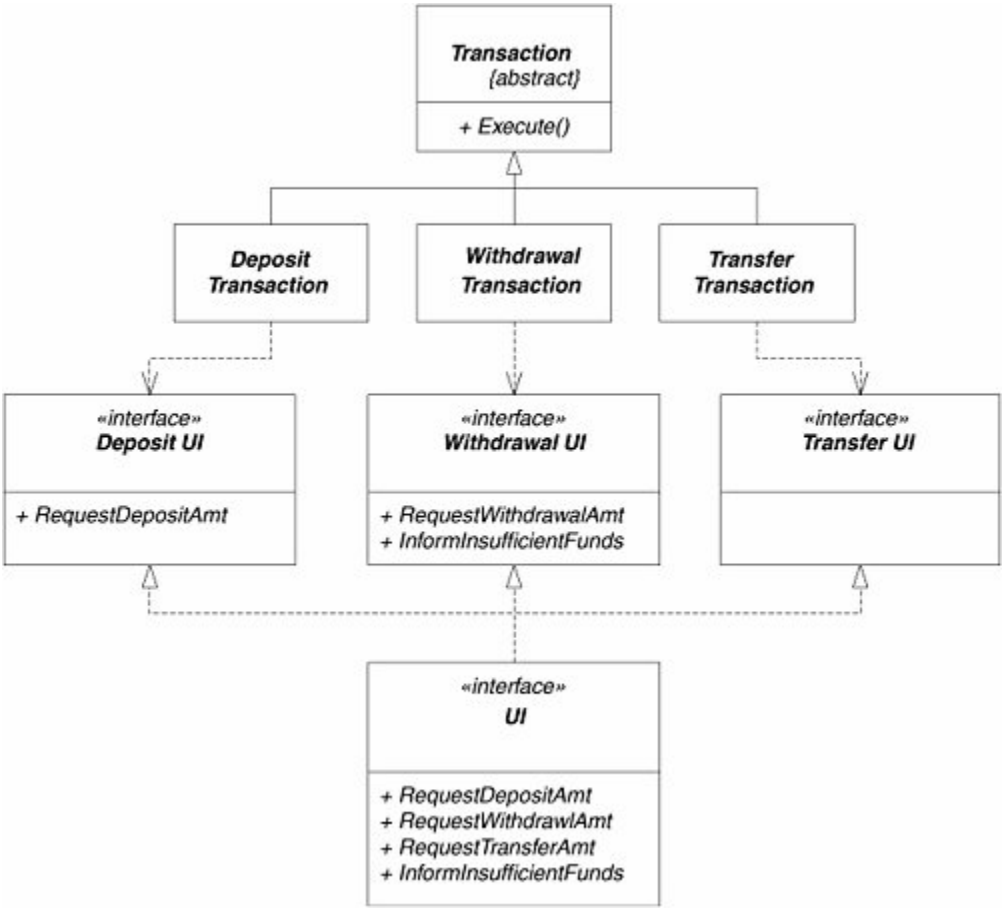


Figura 3.21. Fat interface que cumple con ISP

- Avoid Coupling (J. Flymnt, 2005)

Este principio establece que debemos evitar dependencias con otros elementos.

3.2.3.2 Patrones de Diseño

Los patrones de diseño en el contexto de la ingeniería de software describen una solución elegante a los problemas de diseño de clases orientadas a objetos.

El uso de patrones de diseño permite hacer diseños flexibles, modulares, reusables, confiables y fáciles de utilizar. En el libro de Gamma et. al. (E. Gamma, 1995) se puede encontrar más información sobre los patrones de diseño. Algunos patrones de diseño comúnmente utilizados en el desarrollo de motores de juegos y videojuegos son:

- Abstract Factory

La *abstract factory* Proporciona una interfaz para crear objetos de familias de clases relacionadas o dependientes; sin especificar la clase concreta a utilizar.

Un ejemplo se muestra en la Figura 3.22 es un sistema de sonido basado en (Llopis, 2008) que permite crear diferentes sistemas de sonido, dependiendo del hardware instalado.

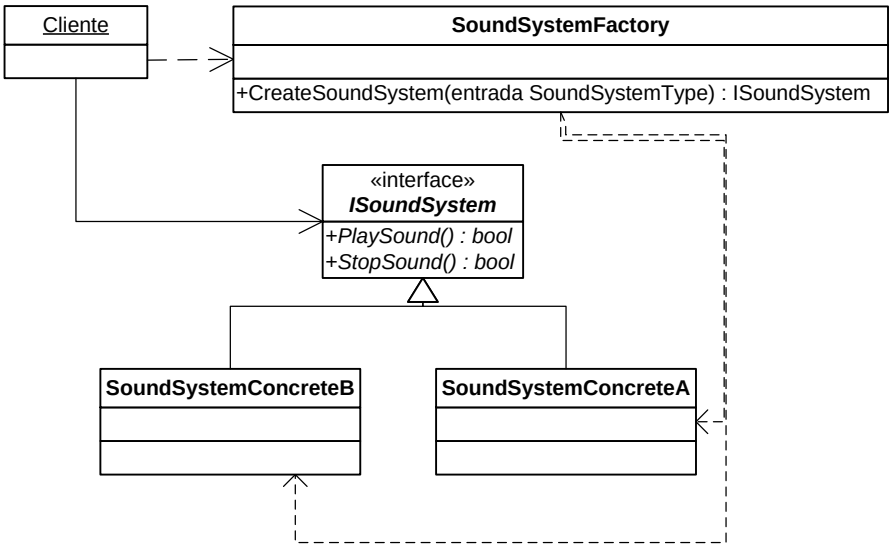


Figura 3.22. Sistema de sonido que usa abstract factory

- Singletone

En el patrón *singleton* aseguramos que una clase tenga una sola instancia, y proporcione un punto de acceso global.

El modelo de la Figura 3.23 muestra una clase que usa el patrón singleton.

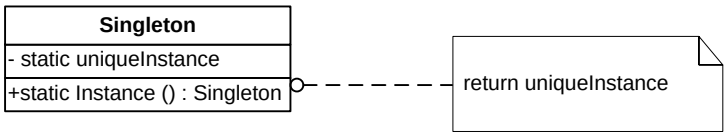


Figura 3.23. Modelo UML de un singleton

- Facade

El patrón *facade* provee una interfaz unificada para un conjunto de interfaces dentro de un subsistema. La Figura 3.24 muestra un facade que define una interfaz de alto nivel para facilitar el uso del subsistema.

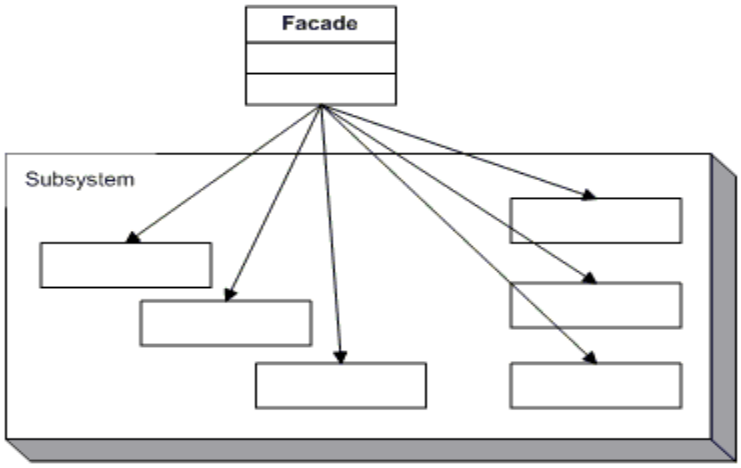


Figura 3.24. Modelo que usa facade

- Observer

En el patrón de *observer* una lista de observers se registran con un sujeto. Cuando el sujeto realiza un cambio a su estado interno, notifica a los observers que se sincronizan con el sujeto en cuestión.

La Figura 3.25 muestra un modelo del patrón observer. La clase sujeto permita agregar o remover observadores. Cuando el sujeto concreto cambia de estado llama a la función Notify() que avisa del cambio a los observadores.

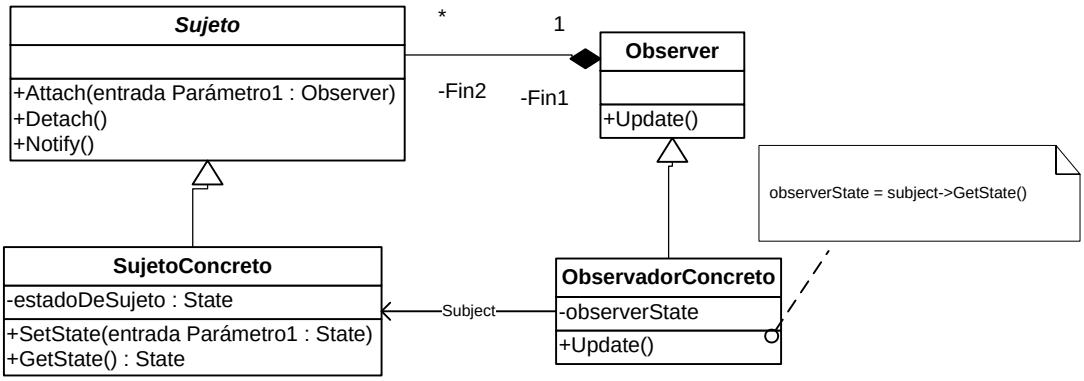


Figura 3.25. Patrón observer

- Composite

Compone a los objetos en una estructura de árbol que representa una jerarquía. Composite permite a los clientes usar objetos individuales como compuestos.

La Figura 3.26 muestra que un Componente puede ser de tipo Composite o Leaf. El tipo Composite puede estar compuesto por uno o más componentes.

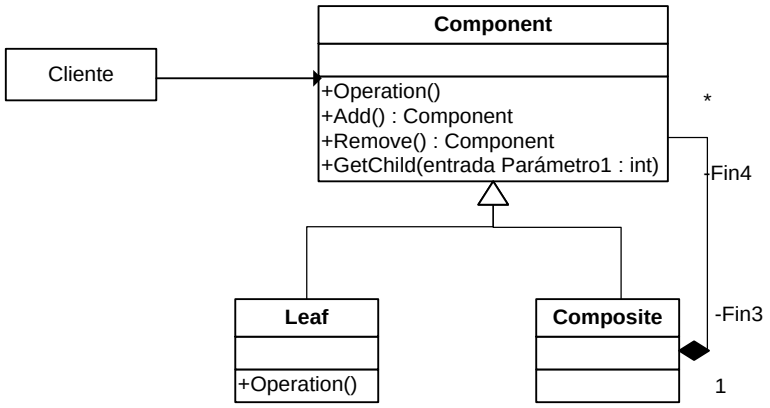


Figura 3.26. Modelo UML del patrón Composite

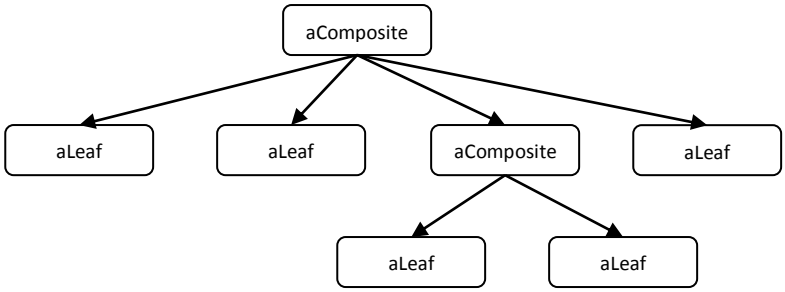


Figura 3.27. Representación de un árbol de componentes

3.3 Tipos de Motores

Anteriormente se dijo que algunos subsistemas del motor de juegos son llamados motores debido a su complejidad y extensión.

En la actualidad existe una gran cantidad motores comerciales y gratuitos, es importante conocer el estado actual de la tecnología y las posibles herramientas que podemos usar para desarrollar nuestro juego o motor de juegos.

3.3.1 Motores Generales

RAGE Engine.- Motor comercial desarrollado por Rockstar San Diego. Se comenzó a desarrollar RAGE (Rockstar Advanced Game Engine) en 2004. Permite utilizar mundos extensos y tiene una compleja IA. Utiliza Natural Motion para las animaciones y para la física usa Bullet.

Se usó en los siguientes juegos: Rockstar Presents Table Tennis, GTA IV + Episodes, Midnight Club: Los Angeles, Red Dead Redemption, L.A. Noire.

CryENGINE.- Motor comercial de desarrollo para las plataformas Xbox, PS3, DX9 y DX10. No utiliza middleware de otros desarrolladores, maneja su propia física, audio y animaciones.

Se usó en los siguientes juegos: Far Cry, Crysis, Crysis Warhead, Crysis 2, Aion: Tower of Eternity

Naughty Dog Game Engine.- Motor comercial desarrollado específicamente para el PS3. Maneja una infinidad de objetos dinámicos con física independiente, interacción ambiente-animación, efectos de iluminación y AI. Tiene transiciones entre cinemáticas y juego casi inidentificable, además de que soporta co-op y multiplayer.

Se usó en los siguientes juegos: Uncharted: Drake's Fortune, Uncharted 2: Among Thieve

Unreal Engine.- Motor comercial desarrollado por Epic Games, su núcleo fue desarrollado en C++ y funciona en las plataformas: Dreamcast, Xbox, Xbox 360, PlayStation 2 y PlayStation 3.

La última versión del Unreal Engine es la UE3 y usa: Microsoft's DirectX 9 para Windows XP, Windows Vista, Windows 7 and Xbox 360; DirectX 10 y DirectX 11 para Windows Vista y Windows 7; OpenGL para Linux, Mac OS X and PlayStation 3. Utiliza PhysX para la física y colisiones.

Se usó en los siguientes juegos: Gears of War, Mass Effect, BioShock, Unreal Tournament, Deus Ex, GRAW, Red Steel, Borderlands, Brothers in Arms, Homefront, Mirror's Edge, Singularity, Rainbow Six: Vegas, etcétera.

Panda3D.- Motor gratuito desarrollado en C++ y puede utilizar Python para agregar complementos. Incluye gráficos, audio, I/O, detección de colisiones y otras herramientas.

Delta3D.- Motor de juegos que integra otros proyectos de código abierto en una API sencillo de utilizar, como son: OpenSceneGraph, Open Dynamics Engine, Cal3D y OpenAL.

3.3.2 Motores Gráficos

id Tech 1.- Conocido como motor de Doom. Creado por John Carmack y fue un motor que revolucionó la industria en su época.

OGRE.- Motor de visualización de gráficos, es uno de los motores de gráficos más prominentes. Soporta las API de gráficos Direct3D y OpenGL, y se ejecuta en plataformas Windows, Linux y Mac. Se desarrolló en C++ y existen muchos complementos que permiten integrar motores de sonido, física, colisiones, red, inteligencia artificial, etcétera.

Irrlicht Engine.- Motor de gráficos 3D escrito en C++. Funciona en diferentes plataformas como son Mac OS X, Linux y Windows además Xbox, PSP, SymbianOS y el iPhone. Soporta OpenGL, DirectX 8 y 9, OpenGL ES. La comunidad desarrollo interfaces para el SDL, iPhone y SymbianOS.

OpenSceneGraph.- Es una herramienta de gráficos 3D. Se desarrolló con OpenGL y C++. Funciona en una variedad de sistemas operativos Windos, Mac OS X, Linux, IRIX, Solaris and FreeBSD.

Aleph One.- Motor de juegos para shooters 3D. Desarrollado por Bungie antes de ser comprador por Microsoft. Se desarrolló con C y su principal plataforma es Mac, Windows y Linux.

Axiom Engine .- Motor de visualización de gráficos en 3D. Desarrollada con C# para ser utilizada con .NET y Mono. Provee una abstracción completa del API 3D, Soporta DirectX y OpenGL, contiene un modelo scene graph y soporta shaders complejos.

3.3.3 Motores de Colisiones y Física

Havok Physics.- Es un motor comercial muy popular que se desarrolló en C/C++. Dependiendo del producto que se desea desarrollar, se puede conseguir la versión gratuita o la versión comercial. La versión actual 7.1 funciona en Xbox y Xbox 360; Wii; Sony's PlayStation 2, PlayStation 3 y PlayStation Portable; Linux; y en Mac OS X. Se usó en los siguientes juegos: Too Human, Alone in the Dark, Assassin's Creed, Bio Shock, Halo, Starcraft 2 y muchos más.

PhysX.- Conocido anteriormente como NovodeX. Actualmente es propiedad de Nvidia y se distribuye de manera gratuita o comercial dependiendo del producto que se desea desarrollar. Funciona en Windows 7, Windows Vista, Windows XP, Mac OS X, Linux, Wii, PlayStation 3, Xbox 360. Permite el acelerar el procesamiento de la física pasando algunos cálculos al GPU, permitiendo al CPU realizar otros cálculos. Se usó en los siguientes juegos: Batman: Arkham Asylum, Mirror's Edge, Tom Clancy's Ghost Recon Advanced Warfighter 2, Unreal Tournament 3, Mafia II y muchos más.

ODE.- Open Dynamics Engine, es un motor de física de código abierto cuya principal funcionalidad es la simulación de cuerpos rígidos y colisiones. Utiliza un API C/C++ y fue desarrollado por Russell Smith, el motor tuvo mucha popularidad en 2005-2006, pero actualmente no se encuentra en desarrollo.

Bullet.- Motor de física de código abierto para objetos 3D, tiene una licencia gratuita zlib. Desarrollado por Erwin Coumans, un ex trabajador de Havok. Se ha utilizado en juegos como:

Grand Theft Auto IV, Madagascar Kartz, Regnum Online, etcétera. Además es utilizado en películas.

Box2D.- Motor de física gratuito para objetos en 2D hecho con C++. Se puede utilizar con los siguiente lenguajes Ada, C++, C#, D, Lisp, Lua, Mercury, Pascal, Perl, Python, Scheme. Se ha utilizado en los siguientes juegos *Crayon Physics Deluxe*, *Rolando*, *Fantastic Contraption*, Incredibots y muchos juegos flash.

Newton Game Dynamics.- Permite realizar la simulación de cuerpos rígidos. Es mucho más específico para detectar la colisión, sin embargo sacrifica velocidad.

3.3.4 Motores de Animación

Cal3D.- Cal3D es una librería de animaciones que permite cargar, reproducir y mezclar animaciones personajes 3D con esqueleto. Se desarrolló con C++ y es independiente de un API de gráficos.

Euphoria.- Euphoria es un motor de animación creado por NaturalMotion basado en *la Dynamic Motion Synthesis* (Síntesis de Movimiento Dinámico), la tecnología genera animaciones sobre la marcha usando una completa simulación del cuerpo, músculos y sistema nervioso. En lugar de utilizar animaciones predefinidas, los personajes, las acciones y reacciones se generan en tiempo real.

Endorphin.- Endorphin es un programa de síntesis de movimiento dinámico desarrollado por Natural Motion. Endorphin puede ser usado para simular física con objetos simples y para crear una animación 3D utilizando "comportamientos", los cuales son una serie de movimientos predeterminados que pueden ser usados en un personaje. El programa hace interactuar las animaciones con el resto del escenario, logrando que no se produzcan penetraciones entre objetos 3D. A diferencia de Euphoria, Endorphin no es un motor, sino que es un software para crear animaciones.

Havok Animation.- Es un motor de animación desarrollado por Havok. Dependiendo del producto que se desea desarrollar, se puede conseguir la versión gratuita o la versión comercial. Facilita la integración de animaciones con Havok Physics.

Granny.- Incluye exportadores de modelos y animaciones de los paquetes de modelado 3D y animación comerciales, como son: Maya, 3D Studio Max, etc.

3.3.5 Motores de Inteligencia Artificial

Kynapse.- La compañía Kynogon desarrollo Kynapse que contiene bloques de inteligencia artificial como son: búsqueda de caminos, detección de obstrucciones (*object avoidance*), identificación de vulnerabilidades, algoritmos de movimiento grupal y trabajo en equipo. Actualmente se encuentra en desarrollo por Autodesk.

Havok AI.- Es un motor comercial de inteligencia artificial desarrollado por Havok que contiene herramientas para: búsqueda de caminos, detección de ambiente, movimiento grupal y facilita la interacción con otras herramientas de Havok.

Navpower.- Motor de inteligencia artificial comercial para las consolas de Xbox 360 y Playstation3. Permite crear y modificar superficies donde se puede caminar, algoritmos de búsqueda de camino, y movimientos grupales.

EKIOne.- Es un motor comercial de inteligencia artificial que permite la creación de NPC´s inteligentes y con emociones, algortimos de planeación y decisión, búsqueda de camino y diferentes modos de percepción del entorno.

3.3.6 Motores de Audio

FMOD.- Es un API para la creación y reproducción de audio interactivo. Soporta una gran cantidad de sistemas operativos y plataformas.

Wwise.- Es un motor de audio que permite la creación, integración y administración de audio. Algunas de sus funcionalidades son: creación de bancos de sonido para juegos, combinar niveles de audio, herramientas de profiling y definición de audio ambiental.

Q3D Interactive.- Este motor se utiliza para sonido 3D, permite posición el sonido en videojuegos y otras aplicaciones de realidad virtual.

4. Desarrollo de Cosmopolis

Hasta ahora se han introducido distintos aspectos teóricos de los videojuegos y los motores de juegos. En este capítulo se describe de manera más detallada el desarrollo de la plataforma Cosmopolis, tomando como base la arquitectura de un motor de juegos descrita en el capítulo anterior. También se describen los juegos que existen dentro de Cosmopolis, especialmente el juego WarPipe.

Cosmopolis se desarrolló usando principalmente XNA y C#, sin embargo, se usaron otras herramientas, tanto comerciales como libres, para acelerar su desarrollo. Posteriormente se verán algunos recursos utilizados para el desarrollo de Cosmopolis.

4.1 Estructura del Equipo de Desarrollo

El equipo de desarrollo se encuentra integrado principalmente por estudiantes del posgrado de ciencias de la computación. Durante mi participación en el proyecto, el trabajo se encontraba divido en las siguientes áreas:

- Líder de proyecto: es el encargado de definir la arquitectura y requerimientos de los juegos y la plataforma.
- Programador de gráficos: es el encargado del desarrolló del motor de render.
- Programador de red: es el encargado del desarrolló de la red y la interacción con la base de datos.
- Programador de juegos: es el encargado del desarrolló de los juegos en la plataforma.
- Modeladores y animadores: estudiantes diseño gráfico, externos a USC se encargaron de esta área.

Los programadores podían trabajar en diferentes áreas dependiendo de las necesidades del proyecto, por ejemplo física, audio, etcétera. Durante mi estancia de 7 meses trabajé en diferentes áreas, debido a que el equipo de desarrollo cambiaba aproximadamente cada 6 meses. Realicé programación de juegos participando en el desarrollo de WarPipe, trabaje en el área de red y posteriormente en el área de gráficos. En el subcapítulo 4.4 se describen mis actividades principales.

4.2 Recursos utilizados en el proyecto

4.2.1 Direct3D

En la presente tesis no se utiliza directamente Direct3D, pero se usa de manera indirecta por medio de XNA y WPF, debido a esto es importante tener una idea básica de la funcionalidad de Direct3D.

Direct3D es uno de los múltiples componentes que contiene la API DirectX de Windows. Es un API para gráficos de bajo nivel que nos permite el dibujado 3D utilizando aceleración de hardware. Direct3D puede ser visto como un mediador entre la aplicación y el hardware gráfico (GPU).

Como se puede ver en la Figura 4.1, existe un paso intermedio entre hardware gráfico y Direct3D, el Kernel Mode Display Driver o HAL (Hardware Abstraction Layer). Direct3D no puede interactuar directamente con el hardware gráfico debido a que existen una gran cantidad de tarjetas gráficas en el mercado, y cada tarjeta tiene diferentes capacidades y formas de implementar acciones. El HAL es el conjunto de código específico al dispositivo, de esta manera Direct3D evita la necesidad de conocer los detalles del dispositivo y su especificación se puede hacer independiente del hardware gráfico (Luna, 2006).

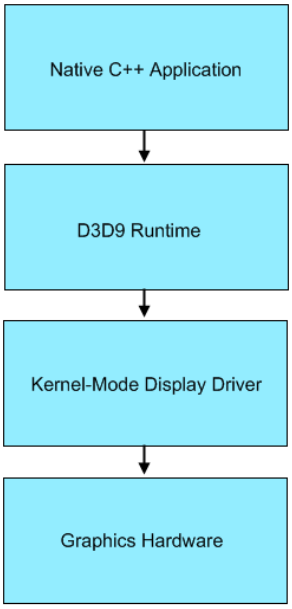


Figura 4.1. Interacción de una aplicación C++ con el hardware gráfico

4.2.2 XNA

Cosmopolis utiliza XNA que es un conjunto de herramientas creadas por Microsoft que facilitan el desarrollo de videojuegos para diferentes plataformas. Se usa el lenguaje C#, el ambiente de desarrollo integrado (*IDE*) Microsoft Visual Studio que es robusto y sencillo de utilizar, y proporciona clases mucho más fáciles de comprender que las antiguas API's de DirectX y Direct3D, por lo tanto simplifica radicalmente el trabajo.

El desarrollo de una aplicación en C# con XNA permite que el desarrollador se centre en los objetivos de la aplicación y deje de lado los problemas habituales de las API's con C++. Por lo tanto, la productividad con C# y XNA es superior y nos facilita no perder el objetivo que estamos buscando para la aplicación.

Esta tecnología está dirigida principalmente a estudiantes y aficionados, sin embargo, se ha demostrados su utilidad en proyectos complejos y de gran extensión. El principal componente de XNA es el *XNA Game Studio* (XGS) que se describirá posteriormente.

4.2.2.1 XNA Game Studio

XNA Game Studio es un IDE, *Integrated Development Environment*, que extiende la funcionalidad de Microsoft Visual Studio para dar soporte al XNA Framework y otras herramientas adicionales. Actualmente, XGS se encuentra en su versión 4.0, sin embargo, en la presente tesis se usa la versión anterior 3.1. XGS facilita el desarrollo de juegos para las plataformas de Windows, Xbox y el Windows Phone sin necesidad de realizar grandes cambios de código.

4.2.2.2 XNA Framework

Un framework es una aplicación parcialmente desarrollada donde el programador no tiene control sobre la estructura base de la aplicación, además un framework provee un conjunto de librerías que se pueden utilizar. El programador debe implementar su propia funcionalidad a los elementos no provistos por el framework.

El XNA Framework contiene una librería de clases diseñadas para el desarrollo de juegos que utilizan managed code, adicionalmente las librerías se basan en Microsoft .NET Framework 2.0 (Microsoft, 2011). Se utiliza un diferente .Net Framework dependiendo de la plataforma a la que

va dirigida la aplicación, en la Figura 4.2 se muestran los frameworks utilizados para cada plataforma.

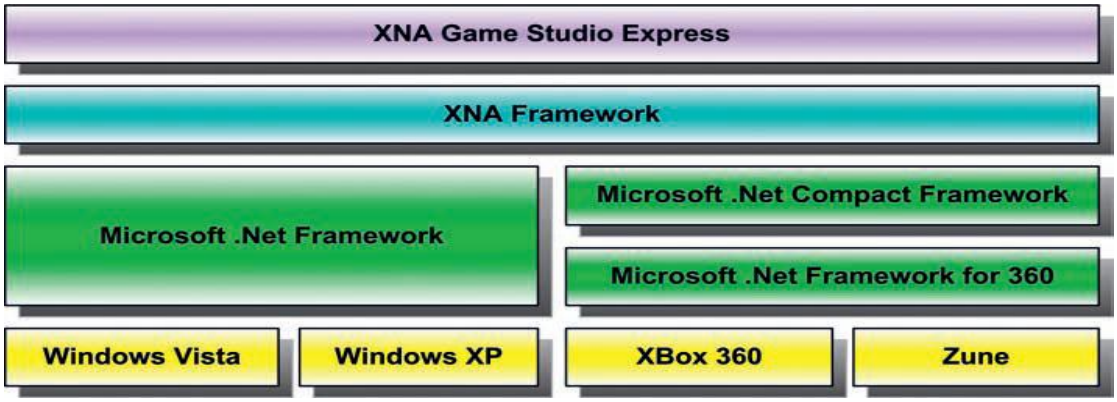


Figura 4.2. Arquitectura XNA

El XNA Framework está formado por 4 capas como se observa en la Figura 4.3.

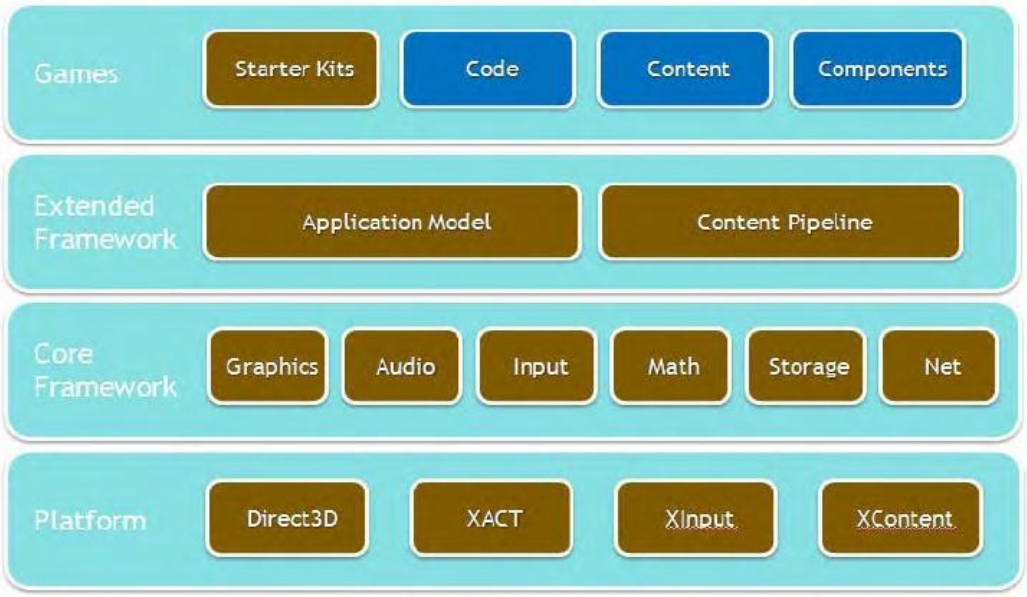


Figura 4.3. Capas del XNA Framework

- Plataforma: Es la capa base que consiste en API's nativas y de bajo nivel. Algunas de las API´s incluyen Direc3D, XACT, XInput y XContent. La capa de plataforma funciona como un *wrapper* de las API's nativas, la Figura 4.4 muestra el proceso de comunicación entre una aplicación de XNA con Direct3D.

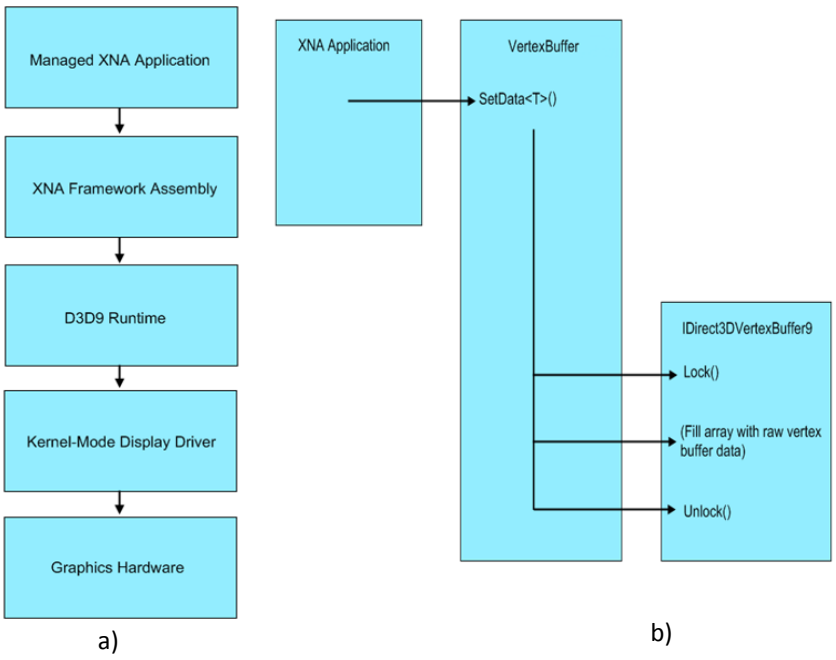


Figura 4.4. a) Interacción de una aplicación XNA con el hardware gráfico. b) Transformación de un método de XNA Framework al API Direct3D

- Core Framework: Es una capa que proporciona un conjunto de librerías comunes al desarrollo de videojuegos. Existen grupos de funcionalidad como son: gráficos, audio, dispositivos de entrada, operaciones matemáticas, almacenamiento y red. Esta capa funciona como los cimientos para capas superiores que extienden la funcionalidad.

  Los grupos de funcionalidad están agrupados dentro de *namespaces* como son:

  - Microsoft.Xna.Framework.Audio: API para el manejo del audio.
  - Microsoft.Xna.Framework.Graphics: API para el rendering 3D y aprovechar la aceleración de hardware.
  - Microsoft.Xna.Framework.Input: Contiene clases para recibir entradas de teclado, mouse y controles Xbox 360.
  - Microsoft.Xna.Framework.Net: Contiene clases para soportar Xbox LIVE, múltiples jugadores y red.

- Extended Framework: El principal objetivo de esta capa es facilitar el desarrollo de juegos.

  Esta capa proporciona al desarrollador un Application Model que ofrece el ciclo básico presente en todos los juegos que consiste en: Inicialización de componentes, carga de archivos, procesamiento de dispositivos de entrada y procesamiento de la lógica del juego, dibujo de la representación del juego y descarga de archivos (Figura 4.5).

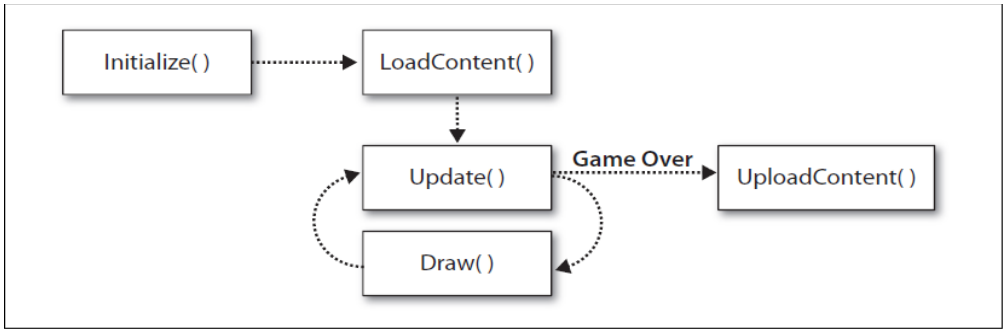


Figura 4.5. Application Model de XNA

El Application Model se muestra de una manera más específica en la Figura 4.6:

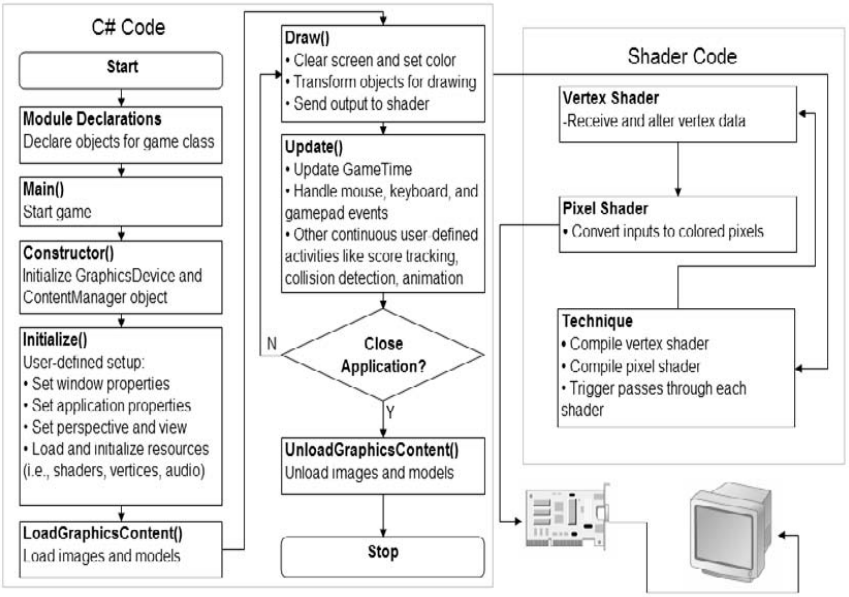


Figura 4.6. Application Model de XNA con aceleración de hardware

Adicionalmente, esta capa provee el *Content Pipeline* que permite procesar diferente formatos de archivos como son: JPEG, TGA, BMP, WAV, MP3, FBX, entre otros, a un nuevo formato optimizado para la plataforma. Si algún formato no se soporta, el desarrollador tiene la posibilidad de extender el content pipeline para soportar el nuevo formato. La Figura 4.7 muestra la secuencia de acciones que realiza el Content Pipeline para convertir un archivo X, FBX y TGA a un nuevo formato optimizado XNB.

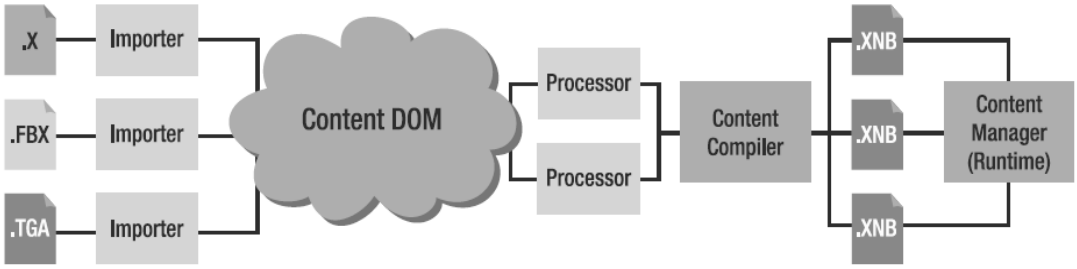


Figura 4.7. XNA Content Pipeline

4.2.4 Havok Physics

Cosmopolis utiliza Havok Physics que es un SDK que fue desarrollada en C/C++ y nos proporciona herramientas para la colisión de objetos en tiempo real y simulación de dinámica de cuerpos rígidos (Havok, 2010). La versión actual 7.1 funciona en Xbox y Xbox 360; Wii; Sony's PlayStation 2, PlayStation 3 y PlayStation Portable; Linux; y en Mac OS X.

Todos los objetos físicos en Havok existen dentro de un mundo que es una instancia de la clase *hkpWorld*. Los objetos físicos son llamados entidades y tienen una clase base *hkpEntity.* Solo existe una entidad proporcionada por Havok: *hkpRigidBody*.

Todas las entidades tienen un *hkpCollidable* que contiene información sobre como realizar una colisión con la entidad. A su vez, los *hkpCollidable* tienen *hkpShape* que define la forma de la entidad.

Havok realiza una simulación varias veces por segundo para crear la impresión de una simulación continua. La Figura 4.8 muestra la secuencia de funciones que se realizan cada frame para obtener una simulación.

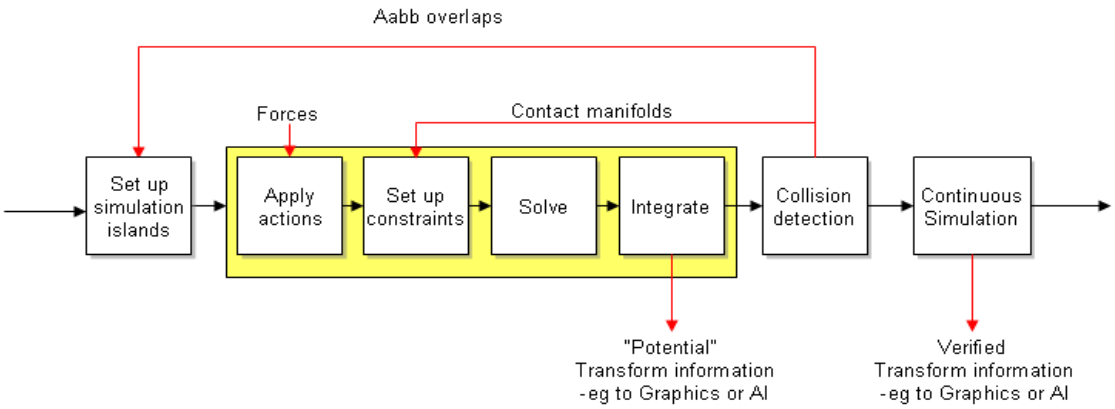


Figura 4.8 Flujo de simulación de Havok

- *Simulation Islands*: Durante una simulación el espacio se particiona en grupos de objetos llamados islas. Todos los objetos de una isla se pueden desactivar (cuando ninguno tiene movimiento) o activar cuando existe una colisión o fuerza.
  Los objetos comparten una isla si existen fuerzas entre ellos o si se aplican las mismas acciones. Los siguientes pasos se realizan por isla.

- *Apply actions*: Se aplican todos las acciones que existen en el mundo, se llama al método applyAction() de cada acción. Las acciones permite controlar el estado de los cuerpos en una simulación.

- *Set up constraints*: Las restricciones de interacción entre objetos se procesan, esto incluye restricciones de contacto para evitar que los objetos penetren y restricciones creadas por el usuario.

- *Solve*: La función de solve calcula el cambio necesario para reducir el error de los constraints. Un ejemplo de una medida de error es la profundidad en que penetra un objeto en otro. Los errores se pueden solucionar moviendo los objetos.

- *Integrate*: El integrador calcula el nuevo estado para cada objeto en la simulación.

- *Collision Detection:* Determina si los objetos están colisionando. La detección de colisión se separa en 3 fases: *broadphase*, que realiza una aproximación rápida para encontrar objetos que podrían estar colisionando; *midphase,* realiza una aproximación más detallada de una colisión potencial; *narrowphase,* determina si los objetos están colisionando.

  Si existe una colisión, se crean agentes de colisión y puntos de contacto de colisión que se usa en la etapa Solve del siguiente frame.

- *Continuos simulation:* En esta etapa se resuelve todos los "*Time Of Impact (TOI)*". Durante cada colisión se crea un nuevo evento TOI, cada TOI pueden generar otros TOI (es recursivo).

Todas las funciones se realizan cuando se manda a llamar la función hkpWorld::stepDeltaTime().

4.2.5 Windows Presentation Foundations

WPF se utiliza en Cosmopolis para el rendering de algunos elementos de la UI, fue introducido en el .NET Framework 3.0 como una mejora a Windows Forms para desarrollar interfaces de usuario en aplicaciones de Windows. Algunas de sus características son: el dibujado se realiza mediante

DirectX; se separa la apariencia y la funcionalidad, la apariencia se describe en XAML's y la funcionalidad se puede escribir en algún lenguaje .NET(C#, VB); su unidad de medida no son pixeles, por lo que se visualiza correctamente en diferentes resoluciones. Sin embargo, el tamaño y complejidad de WPF complican su aprendizaje.

4.2.6 Subversions SVN

Es un software libre usado para el control de versiones y facilitar el uso de archivos compartidos desde distintas computadoras. Todos los archivos se guardan en repositorios, estos pueden ser accedidos por red y se asigna un único número de versión que identifica un estado común de todos los archivos del repositorio en un instante determinado.

El laboratorio Gamepipe de la USC da el servicio de subversiones y solo se requiere un cliente para acceder y descargar el repositorio. En el proyecto se usó Tortoise como cliente SVN.

4.2.7 Herramientas de modelado y diseño

- Maya.-Es un software para la creación de modelos, animaciones, efectos especiales y renders. Los archivos generados en Maya se pueden exportar al formato FBX y pueden ser utilizados en XNA.

- Adobe Photoshop.- Este software se utilizó para la creación y modificación de imágenes 2D.

4.3 Estructura de Cosmopolis

4.3.1 Estructura de la plataforma

La plataforma es un mundo virtual y un motor de juegos, ya que permite la creación de diferentes tipos de juegos para la investigación y contienen varios de los subsistemas descritos en un motor de juegos.

Se utiliza un sistema de control centralizado, la clase *Engine* es la encargada de gestionar la ejecución de los otros subsistemas y permite la comunicación entre ellos (Figura 4.9). La actualización del motor se realiza durante el "ciclo de juego".

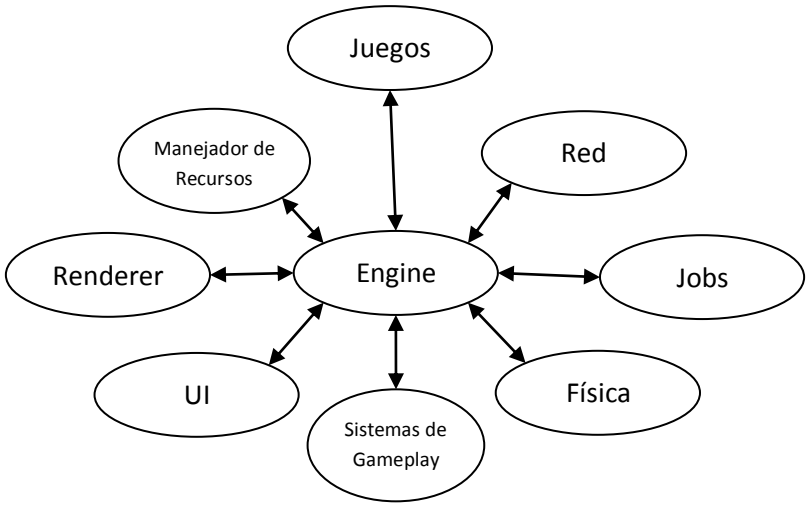


Figura 4.9. Sistema de control centralizado.

El mundo virtual tiene una estructura de 2 niveles: mundo externo y juegos. El mundo externo es una simulación de una ciudad moderna. Los juegos se encuentran dentro del mundo externo, pueden estar completamente aislados, por ejemplo, dentro de un edificio, o pueden estar integrados al mundo externo, por ejemplo, dentro de la ciudad o sus periferias (Figura 4.10).

Figura 4.10. Mundo externo de Cosmopolis

4.3.1.1 El Ciclo de Juego

En el ciclo de juego se realiza la actualización de los componentes del motor. Se busca realizar este ciclo lo más rápido posible para incrementar los fps (frames por segundo) y que el juego se vea fluido.

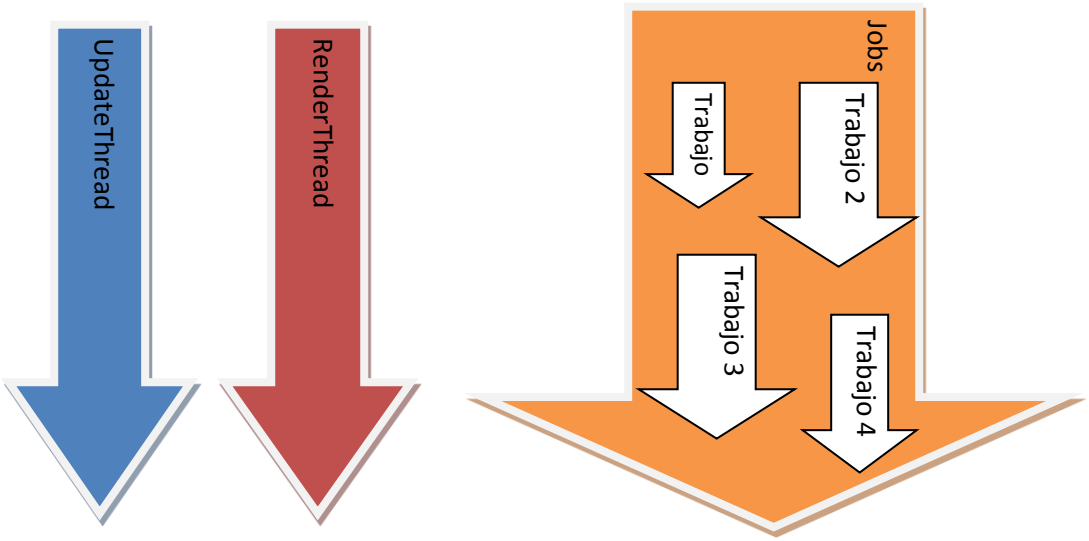


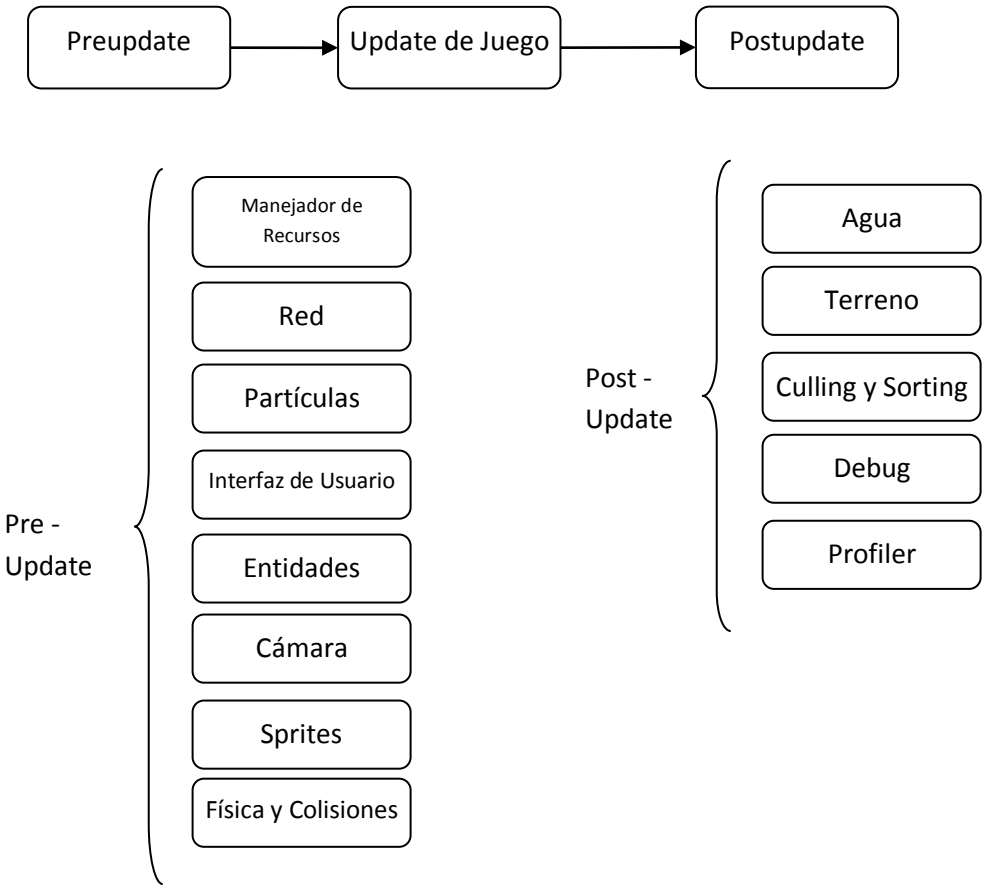
Figura 4.11. Uso de 2 hilos para Render y Update, y un sistema de trabajos.

El motor obtiene paralelismo mediante la creación de hilos por subsistema y el uso de un sistema de trabajos (Figura 4.11).

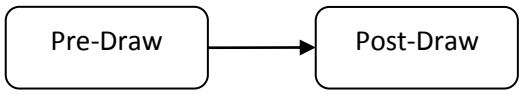
El ciclo de actualización se realiza durante la función de *Update()* de XNA. Para aprovechar las computadoras de varios núcleos, se separa el proceso de rendering y la actualización de la lógica del juego en 2 hilos distintos, a continuación se muestra el código usado para la creación de los hilos:

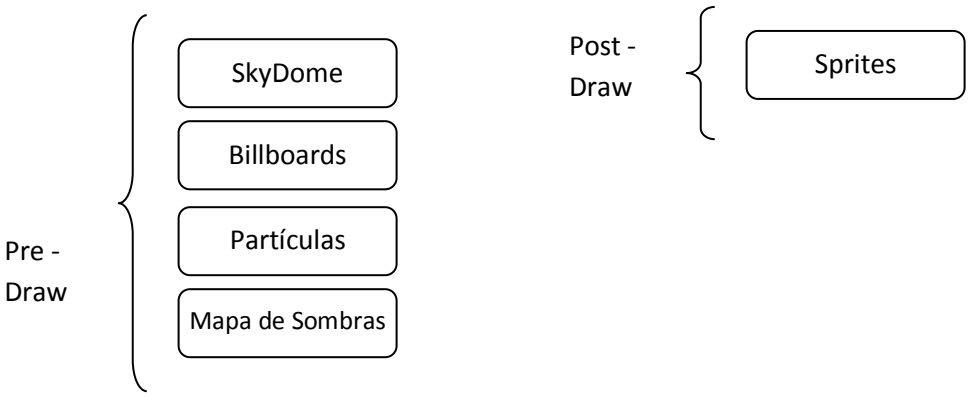
```
protected void InitializeThreading(String updateThreadName, String
drawThreadName)
        {
              .
              .
              .

            updateThread = new Thread(UpdateThread);
            updateThread.IsBackground = true;
            Helper.UpdateThreadName = updateThread.Name = updateThreadName;
            updateThread.CurrentCulture =
System.Globalization.CultureInfo.InvariantCulture;
            updateThread.Start();

            drawThread = new Thread(DrawThread);
            drawThread.IsBackground = true;
            Helper.DrawThreadName = drawThread.Name = drawThreadName;
            drawThread.CurrentCulture =
System.Globalization.CultureInfo.InvariantCulture;
            drawThread.Start();
        }
```

- *UpdateThread*: Actualiza la lógica del juego y procesa las entradas del usuario, se divide en: *preupdate*, update de juego y *postupdate*.



- *DrawThread*: Actualiza todos los subsistemas que se comunican con la tarjeta gráfica para realizar un dibujado a pantalla. El drawThread se divide en: pre-Draw y post-Draw.





La interacción entre los 2 hilos se realiza mediante 2 buffers y se utilizan AutoResetEvents para sincronizar el hilo de update y de render. En los siguientes códigos se muestra la creación de los buffers y el uso de AutoResetEvents.

- Buffers:

```
     drawInputs = new DrawInputs[2] { new DrawInputs(Engine), new
DrawInputs(Engine) };
      drawInputsForUseInUpdate = drawInputs[0];
      drawInputsForUseInDraw = drawInputs[1];
```

- AutoResetEnvents

```
updateStart = new AutoResetEvent(false);
drawStart = new AutoResetEvent(false);
updateEnd = new AutoResetEvent(false);
drawEnd = new AutoResetEvent(false);


protected void UpdateThread()
  {
      while (true)
      {
          updateStart.WaitOne();
          Thread.MemoryBarrier();

          Update();

          Thread.MemoryBarrier();
          updateEnd.Set();
      }
  }

  protected void DrawThread()
  {
      while (true)
      {
          drawStart.WaitOne();
```

```
                Thread.MemoryBarrier();

                Draw();

                Thread.MemoryBarrier();
                drawEnd.Set();
            }
        }
```

El hilo de updateThread actualiza sus cambios y los escribe en el primer buffer, el hilo de render hace una lectura al segundo buffer y manda la información a la tarjeta de gráficos. Una vez que ambos hilos finalizan, los buffers se intercambian y el proceso se repite (Figura 4.12).

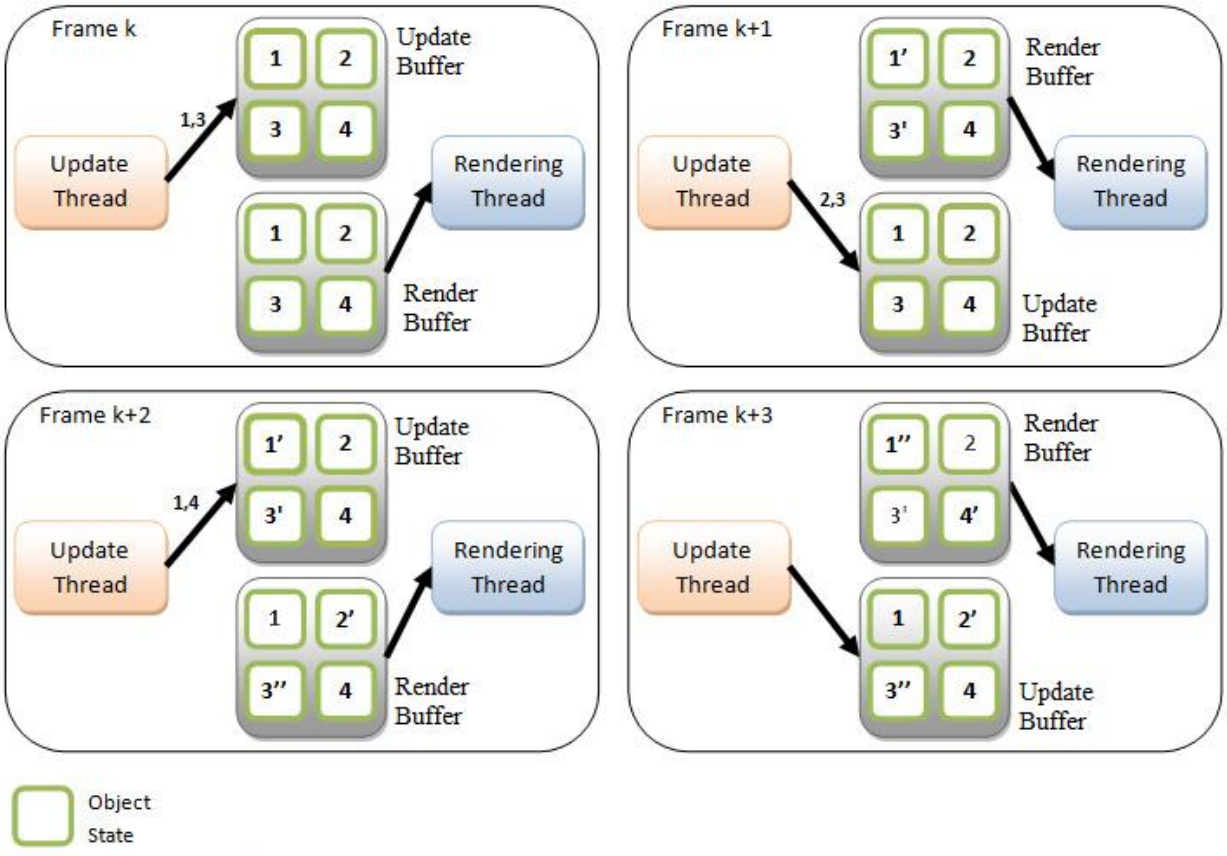


Figura 4.12. Actualización e intercambio de buffers

Los buffers son instancias de la clase DrawInputs (Figura 4.13):

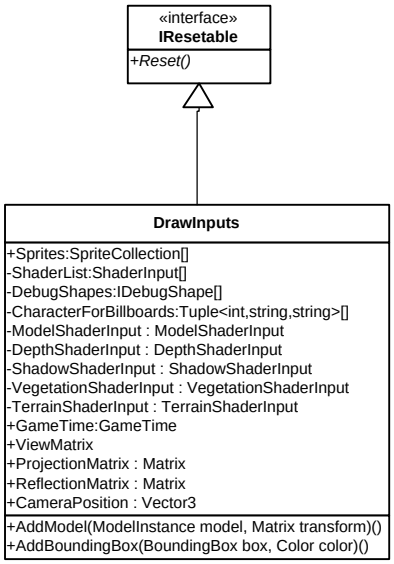


Figura 4.13. Modelo UML de la clase DrawInputs

Se utiliza una arquitectura básica de sistema de trabajos (Figura 4.14) para la administración de acciones que se realizan en paralelo (Gregory, 2010). Las acciones son trabajos independientes que se pueden realizar sin necesidad de sincronización, posteriormente se verá con más detalle este sistema.

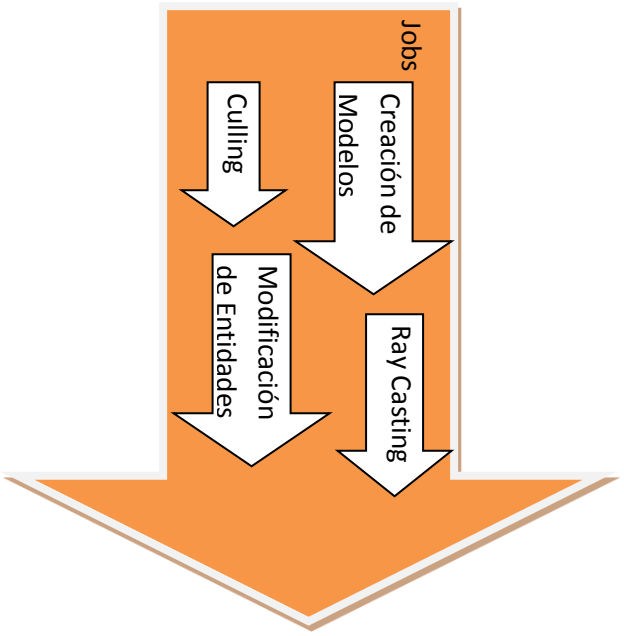


Figura 4.14. Sistema de Trabajos

4.3.1.2 Recursos de Juego

Cosmopolis utiliza diferentes tipos de recursos como: audio, efectos, fuentes, materiales, modelos, texturas.

- Recursos:

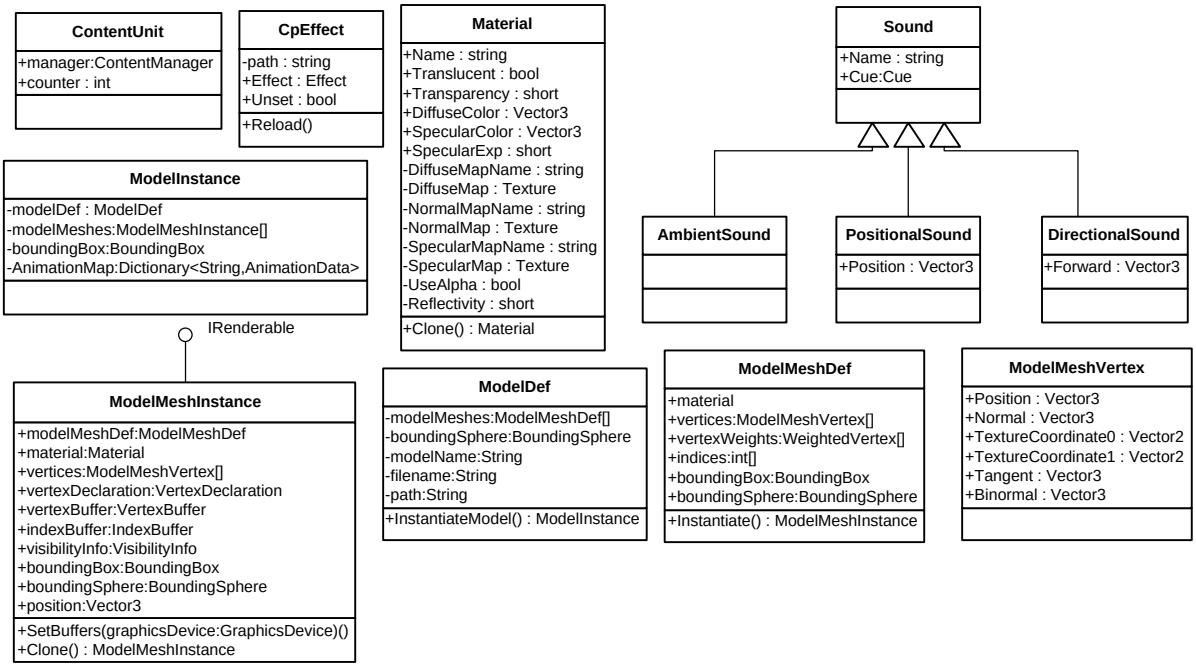


Figura 4.15. Modelos UML de las clases que guardan los recursos

ContentUnit.- Es utilizado por las fuentes, texturas y modelos para mantener el número de objetos que lo están utilizando. Cada *content unit* mantiene un Content Manager para facilitar su destrucción.

CpEffect.- Engloba la clase de *Effect* del framework de xna para facilitar su uso.

Material.- Determina como interactúa la luz con un objeto, modificando el color del vértice/pixel (Luna, 2006) .

Sonido: Se utilizan 3 clases para la creación de diferentes sonidos: AmbientSound, sonido de fondo; Position Sound, tiene una posición que modifica la bocina que reproduce el sonido y su intensidad; Directional Sound, tiene una dirección que modifica la bocina que reproduce el sonido.

ModelDef: Mantiene los datos necesarios para crear una nueva instancia de un modelo.

ModelMeshDef: Un modelo está compuesto por varios Meshes, el ModelMeshDef guarda los datos necesarios para crear una instancia del mesh.

ModelInstance: Representa un modelo que se dibuja en pantalla.

ModelMeshInstance: Representa un mesh que forma parte de un ModelInstace que se dibuja a pantalla.

El manejador de recursos no es un sistema centralizado, está formado por diferentes subsistemas que manejan recursos específicos.

- Manejador de Recursos

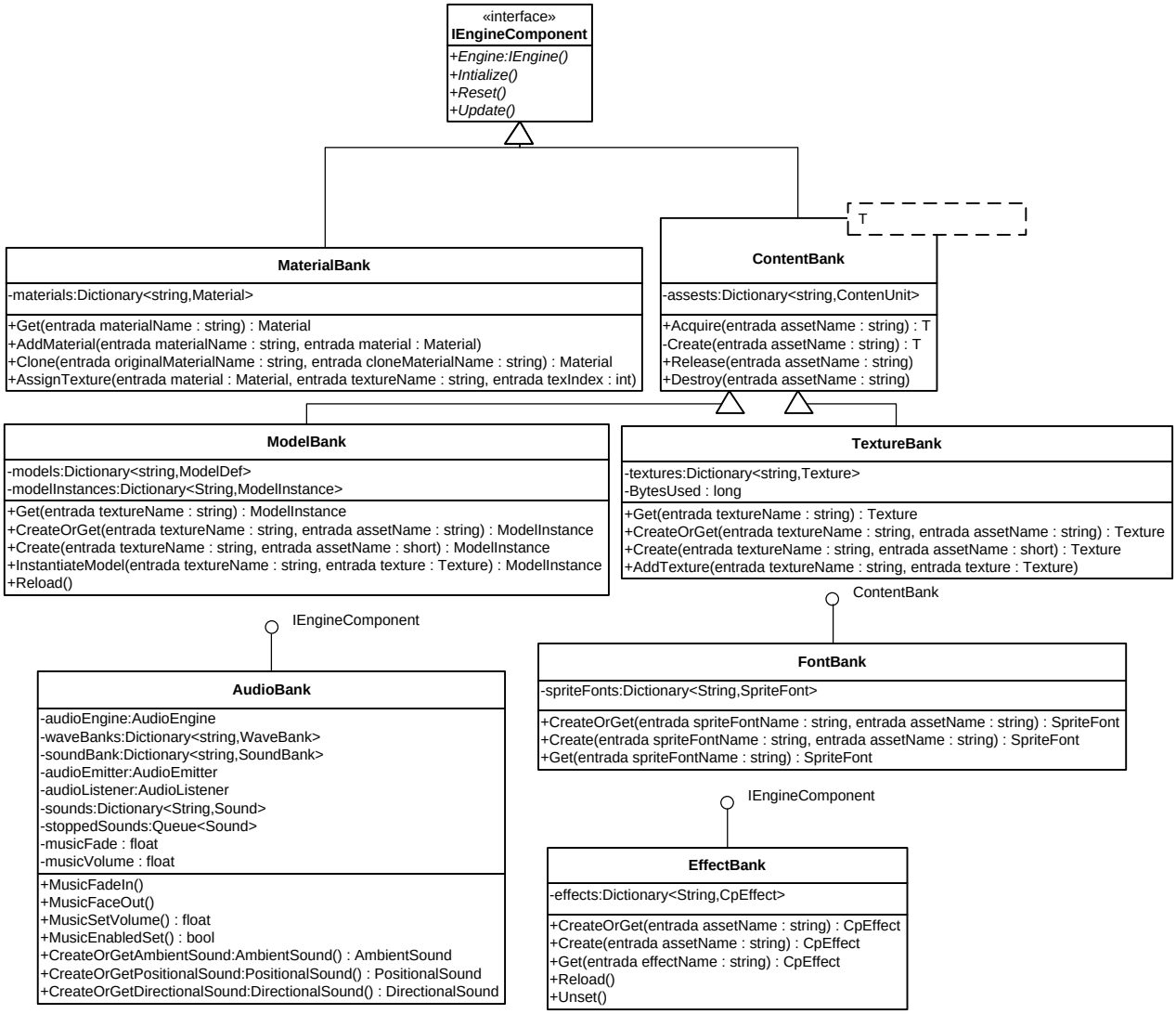


Figura 4.16. Modelos UML de clases que administran los recursos

Content Bank: Clase que administra los content units.

Model Bank: Administra los modelos que se han cargado, sus funciones son:

- Cargar nuevas definiciones de modelos, ModelDef, cuando no existan en memoria.
- Crea una nueva instancia de una ModelDef, ModelInstance, para su render.

FontBank: Administra las fuentes usadas en el juego, SpriteFonts.

TextureBank: Administra todas las imágenes usadas en el juego, Textures.

AudioBank: Se encarga de la administración y reproducciones de todos los elementos de audio.

EffectBank: Administra todos los efectos usados en el juego.

4.3.1.3 Sistema de Trabajos

El motor utiliza un sistema de trabajos que divide los trabajos en 4 categorías:

- Frame Jobs: Trabajos que se deben realizar antes de finalizar el frame actual.
- Sequential Jobs: Trabajos que se deben realizar secuencialmente.
- FreeJobs: Trabajos que se pueden realizar en orden aleatorio.
- UpdateJobs: Trabajos que se realizan y completan antes del pre-update.

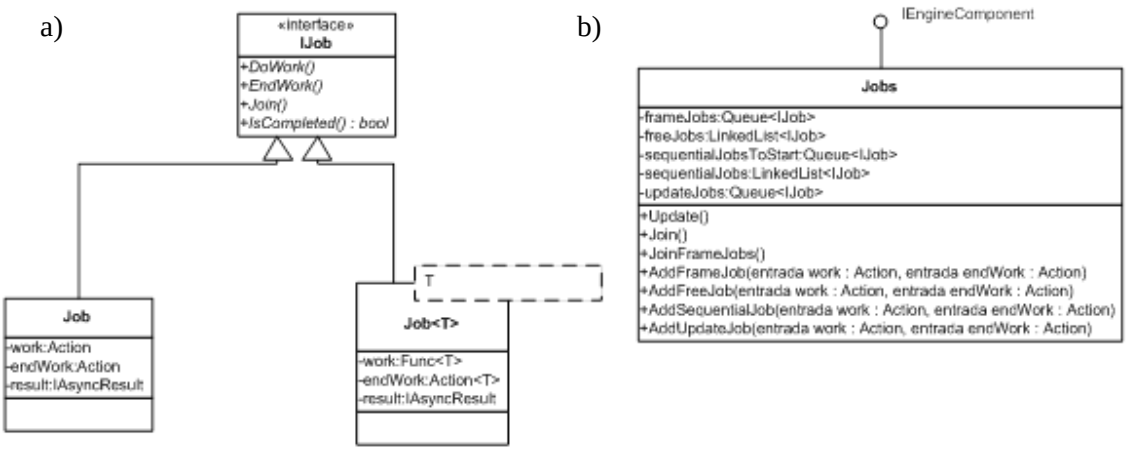


Figura 4.17. a) Modelo UML de los trabajos b) Modelo UML del administrador de trabajos

Los trabajos son objetos que contienen 2 delegados: work, representa el trabajo que se realiza de manera asíncrona; endWork, una vez que el trabajo termina, se llama a este delegado (Figura 4.17).

El manejador de trabajos (Figura 4.17), Jobs, es el encargado de la ejecución asíncrona de los delegados, dependiendo de la categoría del trabajo. Algunas funciones que se usan en el sistema de trabajos son: carga de modelos, culling, modificación de entidades, ray casting, entre otras.

4.3.1.4 Dispositivos de Interacción con usuario

El motor usa los dispositivos de teclado y mouse para interactuar con el jugador. Se envuelven las clases de KeyboardState y MouseState de XNA para extender su funcionalidad.

Las funcionalidades que se agregan son las siguientes:

- Se usan los siguientes estados para las teclas y botones: Up, Down, Released, Pressed.
- Diferenciación entre mayúsculas y minúsculas.
- Mapa de teclas especiales que se forman al presionar Shift.
- Mapa de teclas no usadas.
- Visibilidad de mouse, posición de mouse, posición relativa, scroll relativo.

Además, se utiliza un InputManager (Figura 4.18) para administrar todos los dispositivos y permitir su acceso desde diferentes partes el motor o juego.

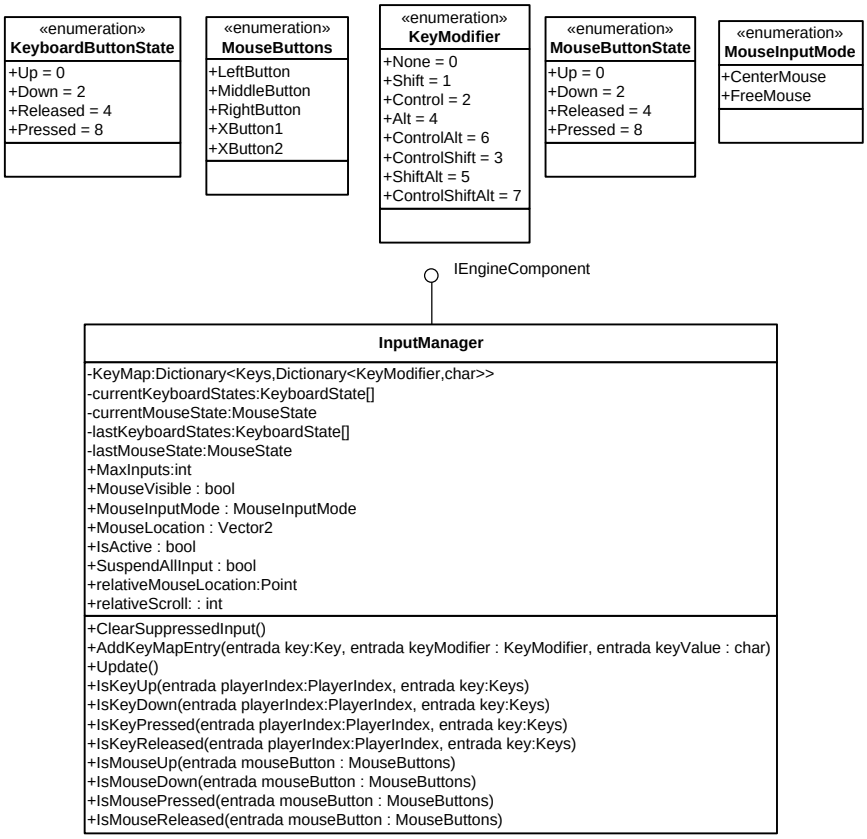


Figura 4.18. Modelos UML de las clases usadas para interactuar con el usuario

4.3.1.5 Profiling y Debuging

El motor utilizan varias herramientas para evaluar la eficiencia del juego, algunas se desarrollaron específicamente para Cosmopolis y otras son herramientas comerciales integradas:

- Se usa un time profiler para evaluar el tiempo de ejecución de una sección específica de código, de esta manera podemos evaluar que secciones de código consume mayor tiempo de CPU y realizar las optimizaciones pertinentes.

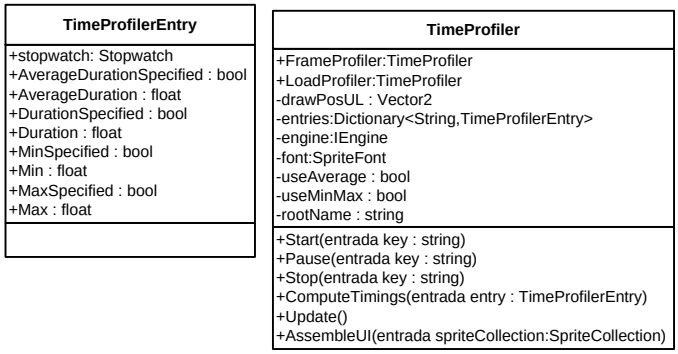


Figura 4.19. Modelo UML de TimeProfilerEntry y TimeProfiler

```
TimeProfiler.FrameProfiler.Start("Draw");

// Codigo a analizar

TimeProfiler.FrameProfiler.Stop("Draw");
```

- Se implemento un contador de frames que muestra la frecuencia deimagenes consecutivas, frames, por segundo (Figura 4.20). Esta herramienta es muy importante ya que un juego que funciona por debajo de los 30fps da el effecto de un retraso a la vista humana.

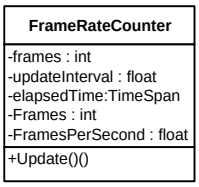


Figura 4.20. Modelo UML de FrameRateCounter

- El uso de memoria se analiza con una herramienta comercial llamada dotTRACE de JetBrains. Muestra la información de memoria heap, stack, garbage collection, entre otras utilidades.

- PIX.- PIX es una herramienta gratuita para análisis de debuging y performance para aplicaciones que usan Direct3D, específicamente nos permite analizar las llamadas que se realizan al GPU (Pettineo, 2010). Podemos usar PIX en nuestro juego de XNA debido a que XNA funciona como un wrapper de Direct3D.

4.3.1.6 Colisiones y Física

Para el manejo de colisiones y física se usa el SDK de Havok Physics. Debido a que havok está escrito en unmanaged C++, necesitamos una manera de interactuar con el unmanaged code (C++) desde managed code (C#), esto se puede realizar de 3 distintas maneras (Scapecode, 2010):

- La primera forma es usar PInvoke, en C# se logra declarando un método como externo e indicar en que dll se puede encontrar.

```
using System;
using System.Runtime.InteropServices;

class MsgBoxTest
{
      [DllImport("user32.dll")]
      static extern int MessageBox (IntPtr hWnd, string text,
      string caption,int type);

      public static void Main()
      {
            MessageBox (IntPtr.Zero,"Please do not press this
            again.", "Attention", 0);
      }
}
```

- La segunda manera es obtener un apuntador a una función usando LoadLibrary/ GetProcAddress/ FreeLibrary y asignar a un delegado el apuntador.
- La tercera es escribir un managed wrapper usando C++/CLI y usar este wrapper para invocar el método.

En la plataforma se usa el tercer método, ya que no solo usamos funciones, también es necesario usar wrappers de clases para la creación de los objetos físicos.

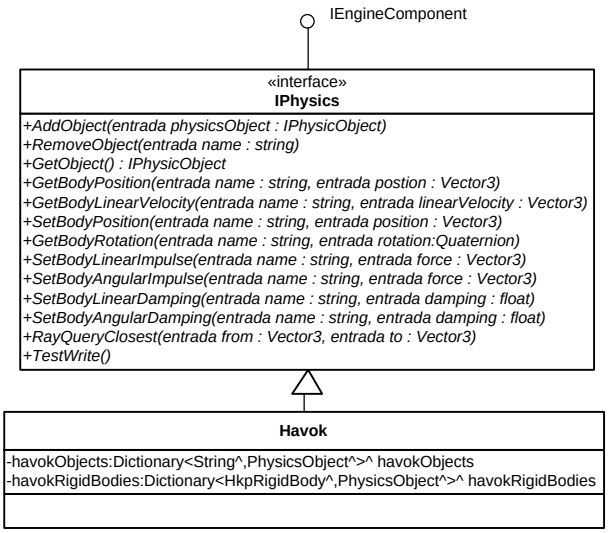


Figura 4.21. Modelo UMLde la clase Havok y la interfaz IPhysiscs

Havok: Este clase (Figura 4.21) funciona como un wrapper para la inicialización, actualización y terminación de Havok Physics. Además, permite acceder y modificar los objetos físicos del mundo.

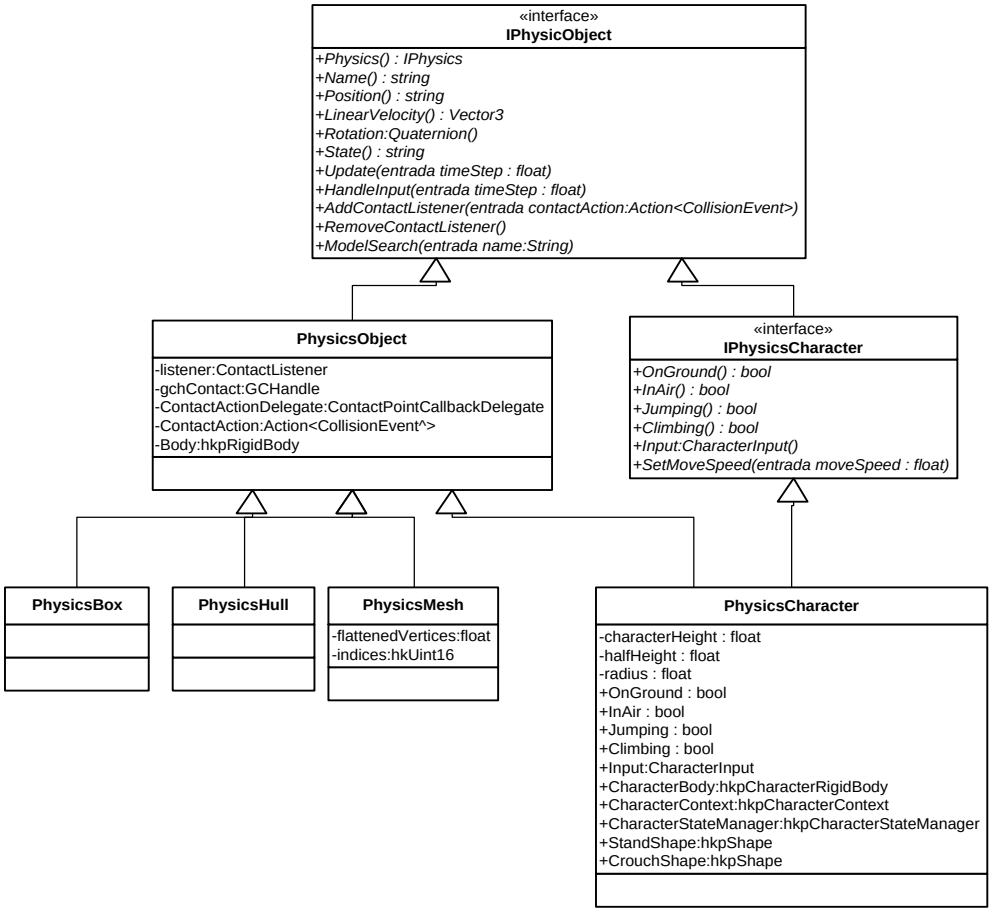


Figura 4.22. Modelo UML de los objetos físicos usados en Cosmopolis

PhysicsObject: Funciona como un wrapper de la entidad hkpRigidBody e incrementa la funcionalidad agregando propiedades como el nombre, estado y funciones en caso de colisión.

PhysicsBox: Facilita la creación de cajas de colisión.

PhysicsHull: Facilita la creación de objetos que contengan los vértices definidos.

PhysicsMesh: Esta clase permite crear objetos 3D que contiene triángulos definidos.

PhysicsCharacter: Permite crear la representación física de los avatares.

El siguiente código muestra la creación de un objeto físico para un personaje:

```
public IPhysicsCharacter PhysicsCharacter { get; private set; }
PhysicsCharacter = new PhysicsCharacter(
GameObjectManager.Engine.Physics , Name, Position);
Engine.Havok.AddObject(PhysicsCharacter);
```

4.3.1.7 Motor de Render

El sistema de render utiliza el buffer DrawInputs para dibujar los cambios generados por el UpdateThread.

En la Figura 4.23 se puede observar que DrawInputs contiene clases que funcionan como las entradas para cada shader: ModelShaderInput, DepthShaderInput, ShadowShaderInput, VegetationShaderInput y TerrainShaderInput.

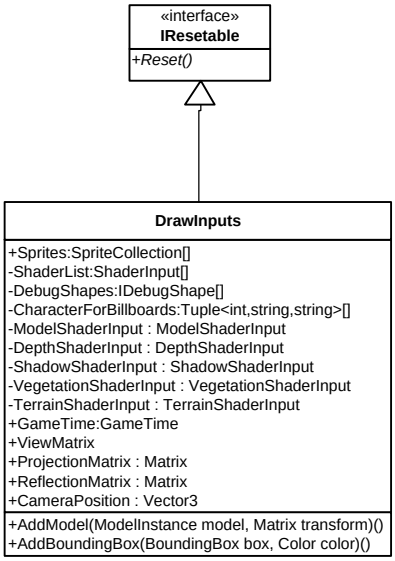


Figura 4.23. DrawInputs es el buffer que guarda los cambios generados en el UpdateThread

Las entradas para cada shader heredan de ShaderInput. En la Figura 4.24 se puede observar la estructura de la clase ShaderInput. Esta clase guarda la información de la cantidad de triángulos, vértices y objetos visibles para cada técnica.

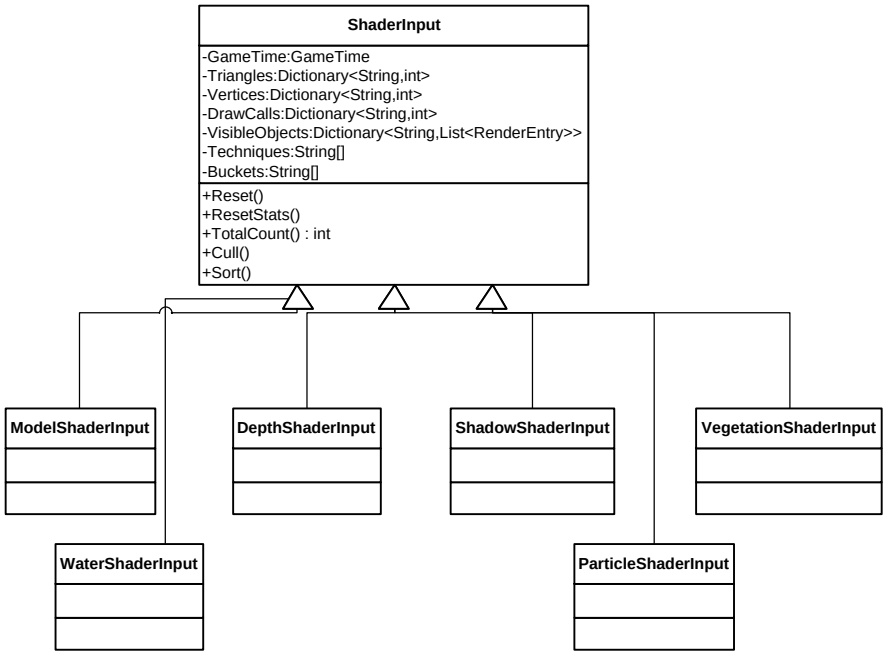


Figura 4.24. Modelo UML de las entradas para los shader

Los diferentes tipos de ShaderInputs guardan información y aplican sus propias técnicas para lograr diferentes efectos, estos son:

ModelShaderInput: Es la clase encargada de guardar los modelos que se dibujan en la pantalla. Utiliza las técnicas: sin textura, translucido, con textura difusa y textura difusa con mapa de normales.

DepthShaderInput y ShadowShaderInpu: Son las clases encargadas de guardar los modelos que reciben o generan sombra.

VegetationShaderInput: Es la clase encargada de guardar los modelos que sirven como vegetación. Unicamente maneja una técnica: Billboards.

TerrainShaderInput: Es la clase encargada de guardar los modelos que sirven como el terreno. Utiliza las técnicas: terreno cercano y terreno distante.

Los shaderInputs guardan una lista de RenderEntrys, la Figura 4.25 muestra la información que contiene cada RenderEntry.

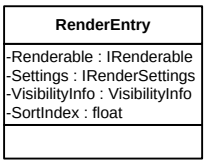


Figura 4.25 Modelo UML del Render Entry

El atributo Renderable es un objeto dibujable que implementa la interfaz IRenderable, se muestra en la Figura 4.26.

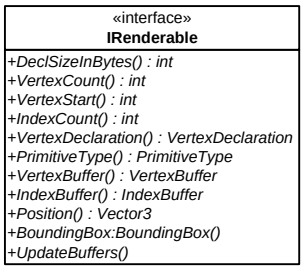


Figura 4.26. Modelo UML de la interfaz IRenderable

El atributo Settings describe algunas características del objeto dibujable, como son: color, material, mapa de iluminación, la Figura 4.27 muestra este modelo y las clases que lo implementan.

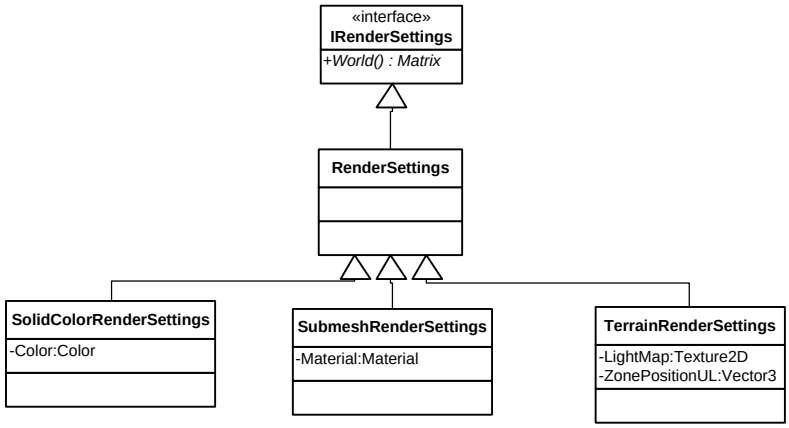


Figura 4.27. Modelo UML de la interfaz IRenderSettings y las clases que lo implementan

El atributo VisbilityInfo guarda la información de si el objetos es visibles o no.

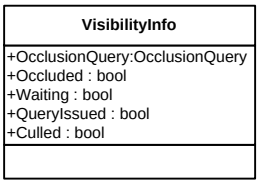


Figura 4.28. Modelo UML de VisibilityInfo

Los shaders son los encargados de mandar las llamadas de render a la tarjeta de gráficos. Cada shader recibe el shaderInput apropiado y se encarga de mandar a dibujar la información del RenderEntry con un efecto específico.

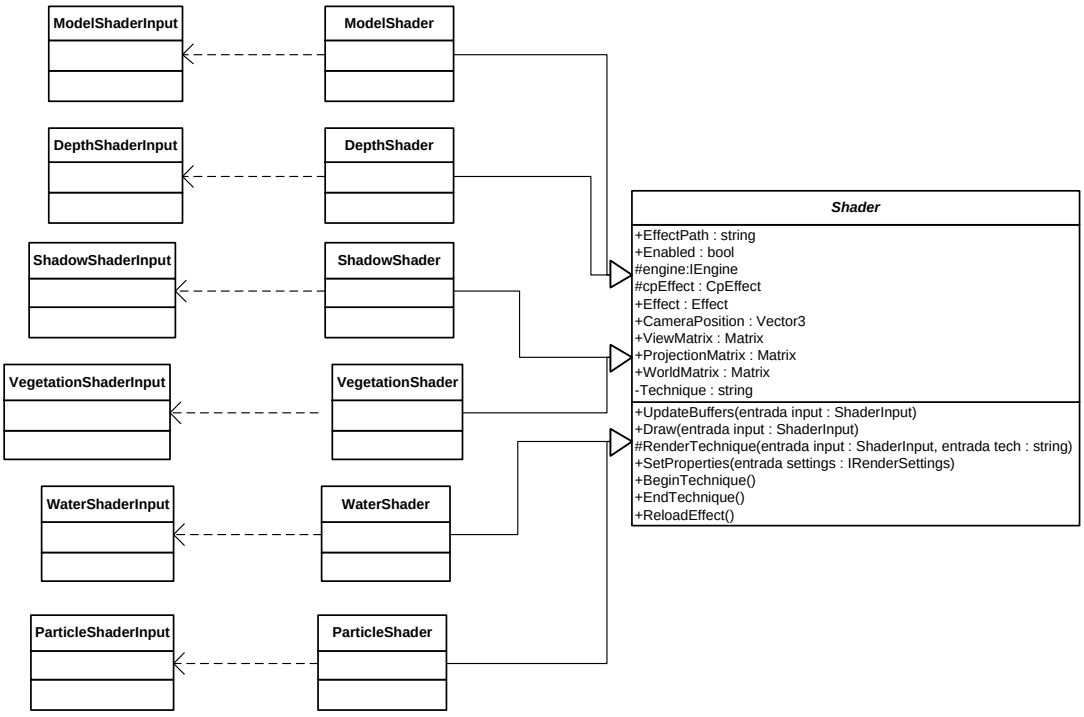


Figura 4.29. Modelo UML de los Shaders

4.3.1.8 Red

La red es uno de los subsistemas más importante de la plataforma por ser un juego MMOG. Se utiliza una arquitectura cliente-servidor (Figura 4.30).

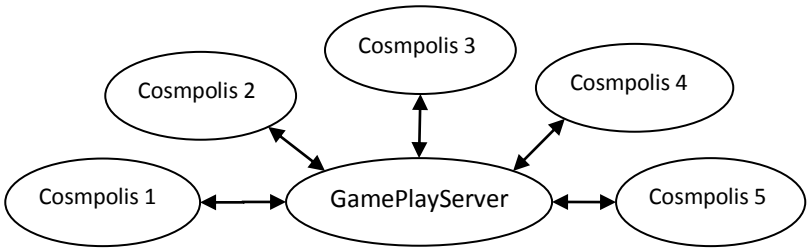


Figura 4.30. Arquitectura cliente servidor

Se desarrollaron dos aplicaciones:

- GamePlayServer: Esta aplicación representa al servidor, sus funciones son: registro de jugadores, recepción de mensaje de jugadores y transferencia de mensajes entre jugadores, procesamiento de mensajes y administración de juegos.

- Cosmopolis: Es la aplicación cliente que proporciona el mundo virtual, el jugador se comunica con el servidor para registrarse, envía información de acciones del jugador al servidor, solicita registro con los juegos, entre otras cosas.

Los mensajes se pueden crear en cualquier subsistema o juego, sin embargo estos se modifican en el subsistema de red para agregar cabeceras. Los mensajes que se crean en los subsistemas o juegos heredan de la siguiente interfaz:

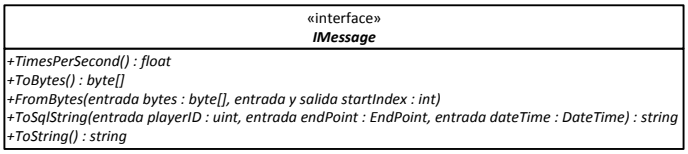


Figura 4.31. Modelo UML de la interfaz IMessage

Cada mensaje agrega información y modifica los métodos: ToBytes, FromBytes, ToSqlString y ToString.

El subsistema de red agrega cabeceras, el mensaje final que es enviado tiene la siguiente estructura, posteriormente se describirá más detalladamente:

Mensaje Final					
Sequence Number	Sequence Channel	Acknowledgment Number	Acknowledgment Channel	EACK	IMessage

- Sequence Number, 32bits – Número de secuencia del paquete.
- Sequence Channel, 32bits – Canal de transmisión asociado a la secuencia.
- Acknowledgment Number, 32bits – Número de confirmación de paquete recibido. 0 hace referencia a que no existen mensajes por confirmar.
- Acknowledgment Channel, 32bits – Canal de transmisión asociado al paquete confirmado.
- EACK, 1bit – Indica si la confirmación es de un paquete que se recibió en orden o en desorden.
- IMessage – Contiene la información del mensaje.

Los mensajes enviados son guardados en la clase SentMessage, en caso de sea necesario reenviarlos.

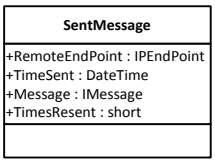


Figura 4.32. Modelo UML de SentMessage

SentMessage			
RemoteEndPoint	TimeSent	Message	TimesResent

- RemoteEndPoint.- Dirección IP destinatario del mensaje.
- TimesSent.- Número de veces enviado por segundo.
- Message.- Los datos del mensaje.
- TimesResent.- El número de veces reenviado

Los mensajes recibidos se guardan en la siguiente clase:

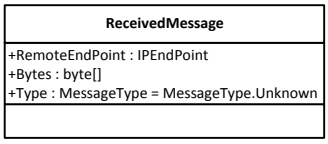


Figura 4.33. Modelo UML de ReceivedMessage

ReceivedMessage		
RemoteEndPoint	bytes	type

- RemoteEndPoint.- Dirección IP del cliente o servidor que envía el mensaje.

- Bytes.- Contiene los datos del mensaje
- Type.- Especifica el tipo de mensaje, dependiendo del tipo será el formato de los datos. En el momento se manejan 96 tipos de mensajes.

Los mensajes son enviados por diferentes canales de comunicación, estos definen la manera en que se debe enviar y procesar un mensaje, pueden ser: ordenados o sin importar el orden, confiable o sin acuse de recibo y el máximo cantidad de veces que se pueden enviar un mensaje por segundo. Algunos canales utilizados son:

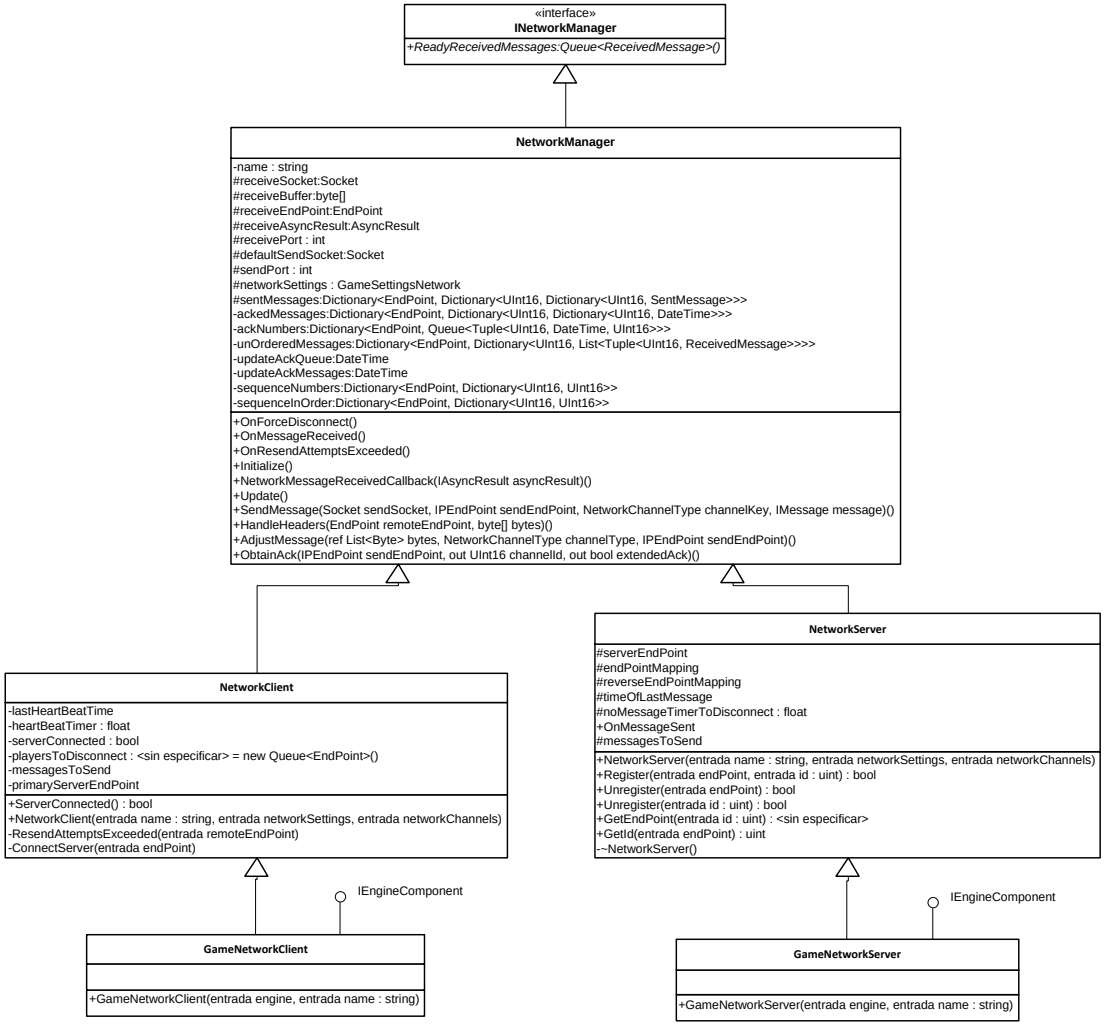
Canal de Comunicación	Ordenado	Confiable	Max. # de envíos de un mensaje p/s
Server_Channel	NO	SI	INF
Reliability_Channel	NO	SI	INF
Client_Channel	NO	SI	INF
Default_Channel	NO	SI	INF
Game_Update_LocalPlayer_Move	NO	NO	8
Game_Update_Npc_Move	NO	NO	20

El código para le creación de un mensaje en algún subsistema o juego es el siguiente:

Cliente: `Engine.Network.AddMessage(Tipo de Canal, IMessage);`

Servidor: `Engine.Network.AddMessage(Tipo de Canal, Dirección Ip de Cliente, IMesage);`

El sistema red es diferente para el cliente y el servidor, sin embargo heredan de clases comunes, a continuación se muestra su modelo:



- NetworkManager.- Esta clase se encarga de la creación de sockets para enviar y recibir mensajes. Además, agrega cabeceras dependiendo del canal de comunicación.

- NetworkClient.- Implementación específica de NetworkManager para el cliente. Se encarga de la conexión y desconexión del servidor.

- NetworkServer.- Implementación específica de NetworkManager para el servidor, implementa funcionalidad para la conexión y desconexión de clientes.

- GameNetworkClient y GameNetworkServer.- Engloban la funcionalidad de red en un componente del motor.

4.3.1.9 Sistemas de Gameplay - Foundations

Cosmpolis proporciona los siguientes componentes sobre los cuáles la lógica específica de los juegos se puede implementar:

- Modelo de objetos para la creación de juegos

Todos los juegos que se quieran incorporar a Cosmopolis deben heredar de Subgame. Es necesario crear un juego para el cliente (Figura 4.34) y otro para el servidor (Figura 4.35), la diferencia son los mensajes que se manejan, su procesamiento y el servidor contiene más métodos y atributos para el cargado y procesamiento del juego.

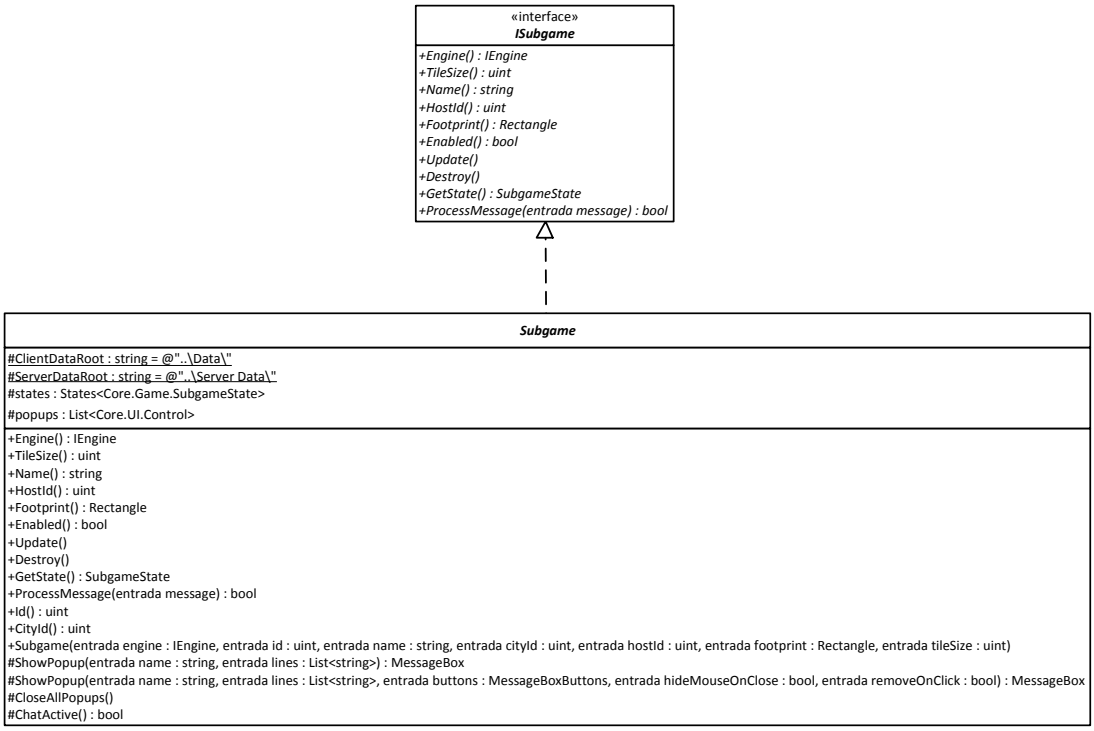


Figura 4.34. Modelos UML de Subgame y ISubgame para cliente

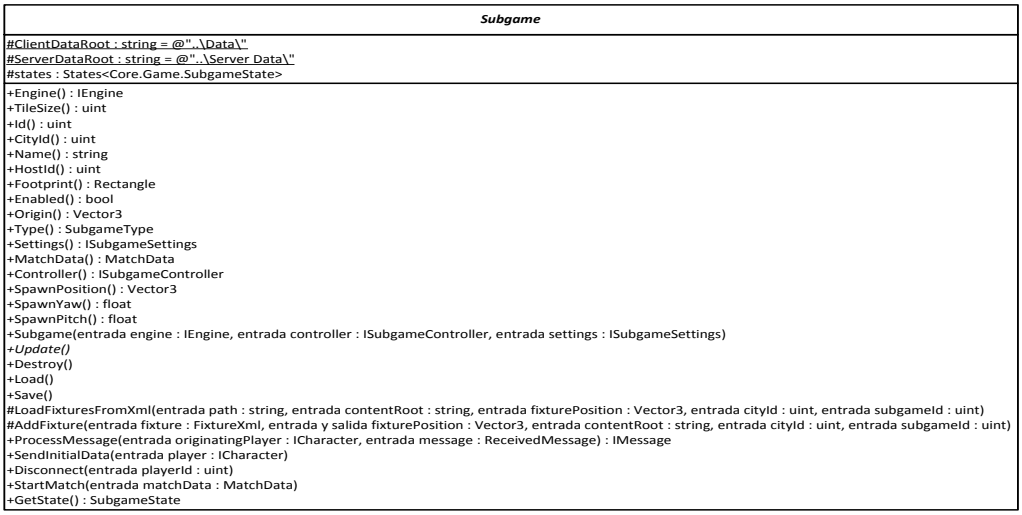


Figura 4.35. Modelos UML de Subgame para servidor

- Administrador de Juegos

El Administrador de juegos es el encargado de la creación, actualización, transferencia de mensajes y destrucción de todos los juegos contenidos en Cosmopolis. El administrador de juegos funciona de manera distante en el cliente que en el servidor.

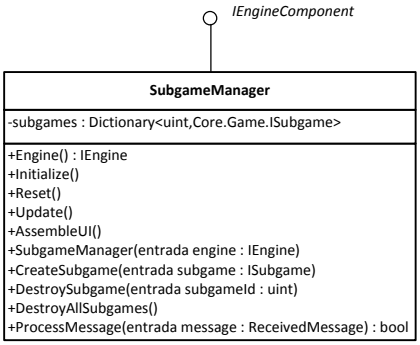


Figura 4.36. Diagrama de clase UML del SubgameManager del Cliente

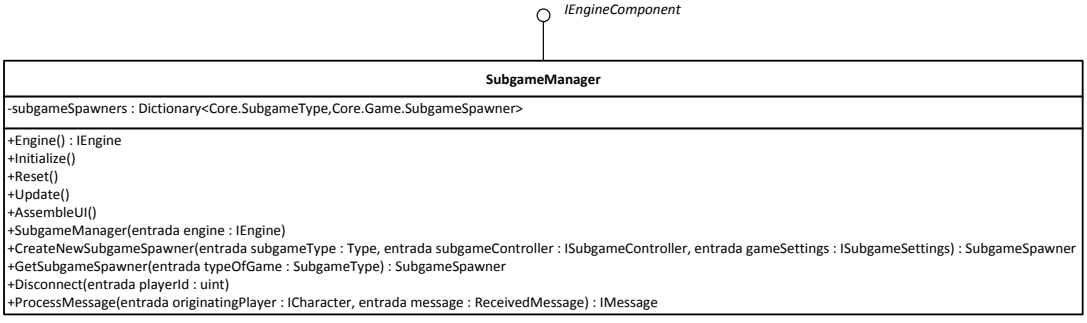


Figura 4.37. Diagrama de clase UML del SubgameManager del Servidor

La Figura 4.36 muestra el administrador de juegos en el cliente, se crea un juego cuando recibe el mensaje del servidor, en el siguiente segmento de código se muestra la recepción de un mensaje para crear un juego *WarPipeSubgame*, la creación de un nuevo trabajo y la creación del juego con el administrador de juegos:

```
case MessageType.WarPipeSubgame:
{
	var WarPipeGameMessage = new
	Subgames.WarPipe.Network.Messages.WarPipeSubgame(message.Byte
	s, ref startIndex);

	Engine.Jobs.AddSequentialJob<WarPipeGame>(
	() => new WarPipeGame(Engine, WarPipeGameMessage, 1)
	(WarPipeSubgame) =>
	{
		Engine.Subgames.CreateSubgame(WarPipeSubgame);
	});
	break;
}
```

La Figura 4.37 muestra el administrador de juegos en el servidor, a diferencia del cliente, en el servidor el administrador de juegos es el encargado de la creación, ejecución y destrucción de SubgameSpawner (Figura 4.38).

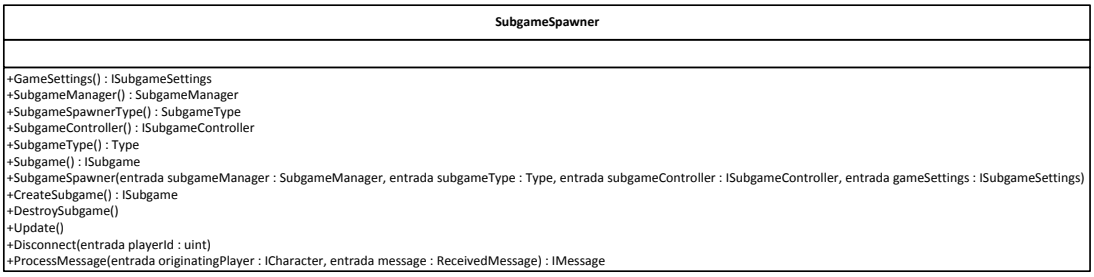


Figura 4.38. Modelo UML de SubgameSpawner

El SubgameSpawner puede crear cualquier tipo de juego, únicamente requiere el tipo de juego y GameSettings que describe las características del juego. Además asigna un ISubgameController que es una entidad específica para el juego que puede iniciar el juego, cambiar de estados, obtener datos del juego, etc.

- Modelos de Entidades

El GameObjectManager es el encargado de mantener la lista de todas las entidades y actualizarlas. La clase GameObjectManager se muestra en la Figura 4.39, esta clase contiene un *Dictionary* que relaciona un identificador con un IGameObject.

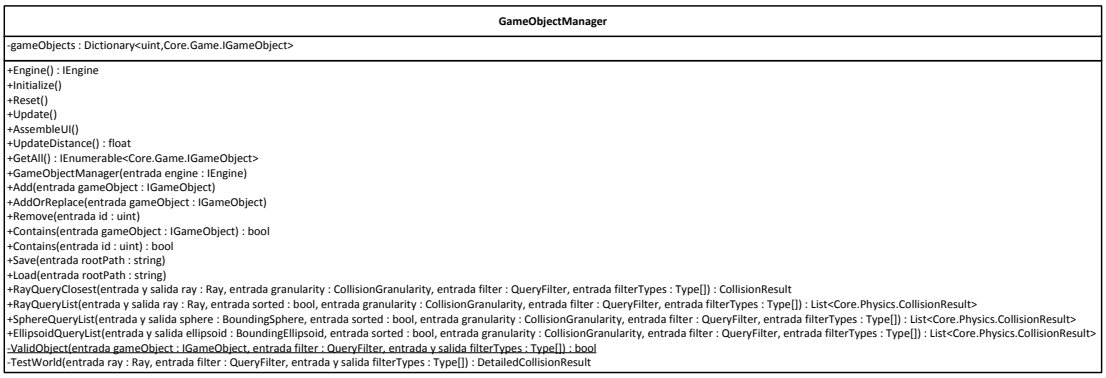


Figura 4.39. Modelo UML de GameObjectManager

IGameObject es la interfaz que heredan todos los objetos del mundo. En la Figura 4.40 se puede observar que varias clases usadas heredan de Character, que a su vez implementa IGameObject, a continuación se describirán algunas entidades:

- Character.- Clase abstracta que contiene la funcionalidad básica para la creación de un personaje en el mundo virtual.
- LocalPlayer.- Extiende la funcionalidad de Character para agregar funciones de interacción con usuario.
- Npc.- Extiende la funcionalidad de Character para crear personajes que funcionan con inteligencia artificial.
- RemotePlayer.- Son avatares controlados por otros jugadores.
- ArenaNpc.- Es un Npc que funciona como controlador de los juegos.

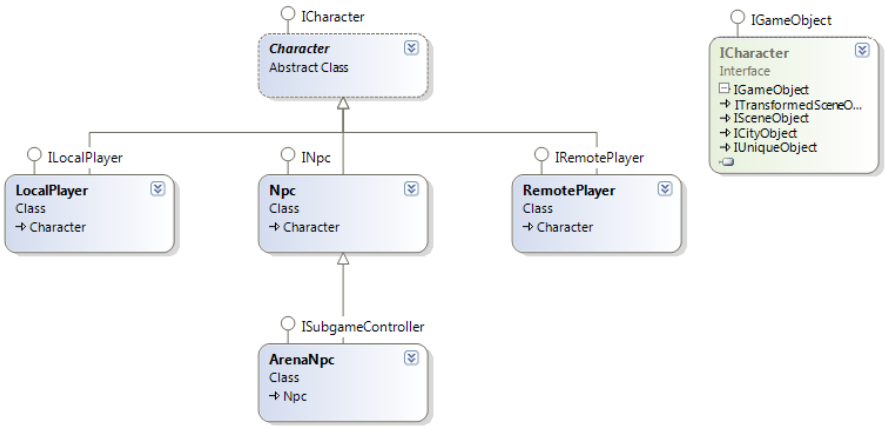


Figura 4.40. Diagrama de clases de entidades

Se usa otro modelo de entidades para la creación de proyectiles, objetos inanimados con posición (*fixture*) y edificios, su diagrama de clases se muestra en la Figura 4.41.

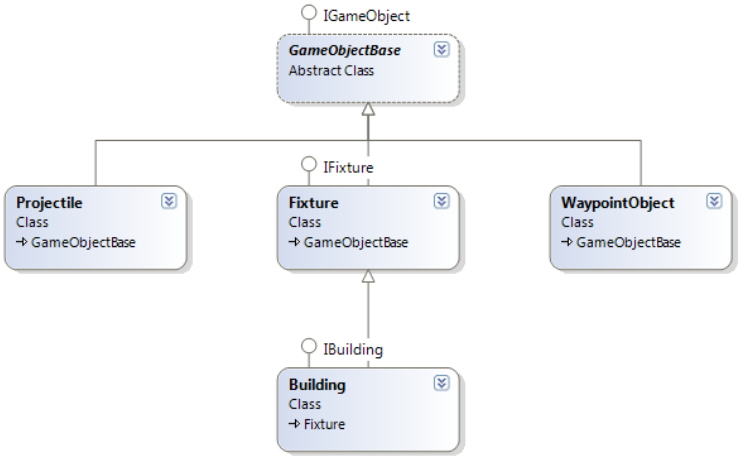


Figura 4.41. Diagrama de clases de entidades

- Definición de mapas mediante XML

Los escenarios de los juegos se definen en archivos XML para facilitar su creación. Cosmopolis puede leer estos archivos y generar el escenario del juego en el mundo virtual. Un archivo de mapa XML debe contener para cada fixture: nombre, nombre de modelo, tipo de modelo, objeto físico para colisión, posición, rotación, escalamiento y una lista de *fixtures* que contenga.

- Máquina de Estados Finitos

Las máquinas de estados finitos permiten describir un sistema en términos de entradas, salidas y estados. Se utiliza una máquina de estados para el control de escenas, estados de entidades y los diferentes estados de un juego.

Se implementó un estado en la clase State representada por la Figura 4.42, al cambiar a un estado se llama la función OnEnter del nuevo estado y la función OnExit del estado anterior. Durante un estado se llama la función OnUpdate para actualizar el estado y OnDraw por si es necesario dibujar algún elemento.

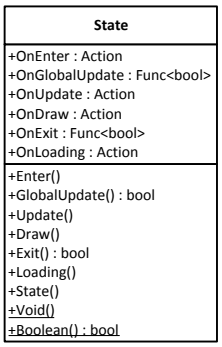


Figura 4.42. Modelo UML de clase State

Si el estado tiene requerimientos de cargado antes de iniciar, se manda a llama la función OnLoading.

Se utiliza un manejador de estados que mantiene una lista de todos los estados, es el encardo de llamar las funciones adecuadas y permite el cambio entre estados Figura 4.43.

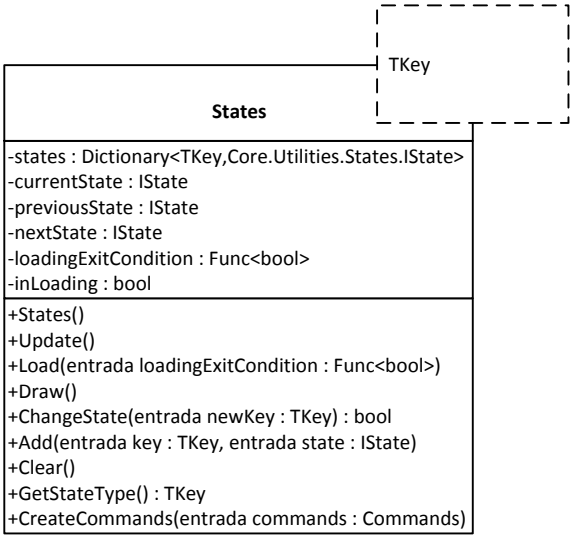


Figura 4.43. Modelo UML del administrador de estados

4.3.2 Juegos

Cosmopolis fue desarrollado de manera que el código facilita la creación de un juego y su incorporación al mundo virtual, actualmente Cosmopolis tienen 3 juegos en desarrollo:

- UNMC: Este juego que se encuentra integrado a la periferias del mundo externo. Involucra la detección y desarme de minas. Se desarrolló para participar en la Imagine Cup 2010.
- Dungeon Run: Juego de acción aventura con fantasía, se encuentra en fase de diseño.

- WarPipe: Juego de acción-shooter multijugador en tercer persona. Se encuentra aislado en un edificio del mundo virtual.

En el presente proyecto, dentro del área de juegos, únicamente se participó en WarPipe.

4.3.3 WarPipe

WarPipe es un juego shooter multijugador con una cámara en tercer o primer persona. Los jugadores del mundo virtual se registran con un NPC para seleccionar su equipo y comenzar a jugar. La Figura 4.44 muestra al avatar interactuando para registrarse al juego, una vez que se selecciona el NPC se muestra una ventana con las opciones de:

- Register for Match.- Permite registrarse a un juego y seleccionar el equipo.
- Withdraw from Match.- Retirarse del juego.
- See Roster.- Ver todos los jugadores registrados
- Leave.- Cerrar la ventana.

Figura 4.44. Registro al juego WarPipe mediante un NPC. Una vez que se interactua con el NPC se muestran las opciones de registro del juego.

Una vez que se han registrado por lo menos 2 jugadores en diferentes equipos, se comienza un temporizador de 30 segundos para comenzar el juego. El juego se realiza dentro de un edificio del mundo virtual; una vez que el juego comienza, los jugadores son transportados desde cualquier parte del mundo virtual al interior del edificio.

El juego de WarPipe tiene una duración y un máximo número de muertes que se definen en el servidor, además se tienen diferentes armas como son: mp5, rifle de francotirador, granada, granada de humo y granada lumínica de aturdimiento. La Figura 4.45 muestra el juego en ejecución.

En este juego se desarrolló el primer experimento de análisis de comportamiento, posteriormente se describirá el experimento.

Figura 4.45. Juego WarPipe en acción

4.4 Actividades

En la estancia de 7 meses en la USC, se trabajó específicamente en algunos subsistemas de Cosmopolis, el juego WarPipe y el primer experimento. A continuación se muestra una lista de las actividades más relevantes que se realizaron.

4.4.1 Juego

1. Agregar crosshair. Se agregó una mira en WarPipe que muestre la posición a la que se está disparando. Además, el color del cursor debe cambiar dependiendo de color del personaje al que se está disparando: color rojo, enemigo; color verde, amigo.

2. Uso de friendly fire. El friendly fire es una opción en el servidor que permite no matar a jugadores del mismo equipo; si el friendly fire se encuentra habilitado, los jugadores puedan atacar y disminuir la vida a jugadores de su propio equipo; si el friendly fire se encuentra deshabilitado, únicamente pueden disminuir la vida de jugadores del equipo contrario.

3. Sincronización de armas, personajes, estadísticas del jugador entre el servidor y los clientes. Se crearon múltiples mensajes para que los clientes tengan la misma información que el servidor, algunos son:

   WarPipeSubgame.- Mensaje que indica la creación de un nuevo juego WarPipe.
   WarPipeTeams.- Mensaje que contiene la información de los jugadores en cada equipo.
   WarPipeChangeTeam.- Mensaje que indica un cambio de equipo.
   WarPipeDeath.- Mensaje que informa de la muerte de un jugador.
   WarPipeKill.- Indica que un jugador mato a otro.
   WarPipeSuicide.- Mensjae que contiene la información de un suicidio de un jugador.
   WarPipeScore.- Mensaje que contiene la información del puntaje de un jugador.
   WarPipeSpectator.- Indica que un jugador es espectador.
   WarPipeSubgameScore.- Contiene la información de todos los jugadores que participaron
   WarPipePlayerScore.- Mensaje que contiene las estadísticas de un jugador.
   WarPipeNewRound.- Mesaje que contiene información del nuevo round.
   WarPipeEndMatch.- Indica que la partida ha terminado.
   WarPipeNewPlayer.- Indica que un jugador ingreso a la partida.
   WarPipeDisconnectPlayer.- Indica que un jugador salió de la partida.
   WarPipeInactive.- Indica que un jugador termino la partida.
   WarPipeTimeRemaining.- Contiene información del tiempo restante de la partida.

   En caso de que un jugador no tenga los prerrequisito para la ejecución de una acción, se utiliza un Trigger. Un Trigger es una acción que se guarda para su posterior ejecución (cuando los prerrequisitos se han cumplido).

4. Guardar información correspondiente al número de muertos, suicidios y puntajes de cada personaje tanto en el servidor como en el cliente.

5. Creación de match y rondas en WarPipe. Se creó una clase WarPipeMatch (Figura 4.46) que guarda la información del número de muertes, rondas y el tiempo máximo de la partida

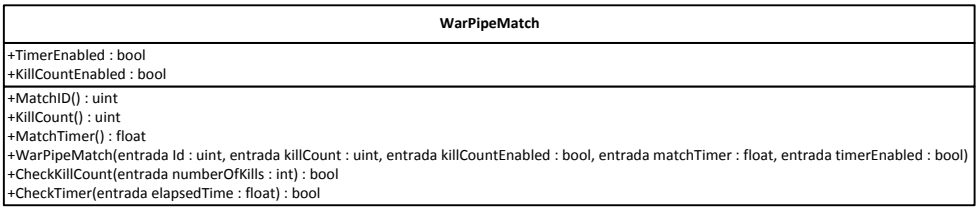


Figura 4.46. Modelo UML de WarPipeMatch

6. Optimización de picking. El picking se utiliza múltiples veces en el juego y subjuegos. Se optimizo de manera que se lanzará un solo rayo en cada frame y se guardan los objetos con colisión.

7. Modo de Espectador. Este modo permite al jugador tener una cámara libre por el mundo o seleccionar un jugador a quien desea seguir automáticamente con la cámara.

8. Cambiar material de personaje dependiendo del color del equipo, a continuación se muestra un segmento de código que realiza el cambio de color para un jugador:

```
private void ChangePlayerColor(String playerTeam, uint playerId)
{
    Character player = Engine.GameObjects.Get<Character>(playerId);

    if (player != null)
    {
       for (int i = 0; i < player.Model.ModelMeshes.Length; i++)
       {
           string postfix = "_" + playerTeam.ToLower();

           var material = Engine.Materials.Get
             (player.Model.ModelDef.GetMaterialName(i) + postfix);

           if (material != null)
           {
              player.Model.ModelMeshes[i].Material = material;
           }
           else
           {
               string assetPath = player.Model.Path +player.Model.ModelDef
      .GetTextureName(i, TextureType.Diffuse) + postfix;

              if (Helper.AssetExists(Engine, assetPath))
              {
                  Material mat =
Engine.Materials.Clone(player.Model.ModelDef.GetMaterialName(i),
player.Model.ModelDef.GetMaterialName(i) + postfix);
                           mat.DiffuseMap =
Engine.Textures.CreateOrGetTexture2D(assetPath);
                           player.Model.ModelMeshes[i].Material = mat;
               }
           }
        }
       }
```

Figura 4.47. Avatares en WarPipe con diferente color dependiendo del equipo

9. Desplegar el nombre, con color correspondiente al equipo, a cada personaje en la parte superior. Se utilizaron SpriteFonts para desplegar texto arriba de un personaje.

10. Estados de Juego WarPipe. Se usó la máquina de estados para crear diferentes estados en el cliente y en el servidor para el juego de warpipe.

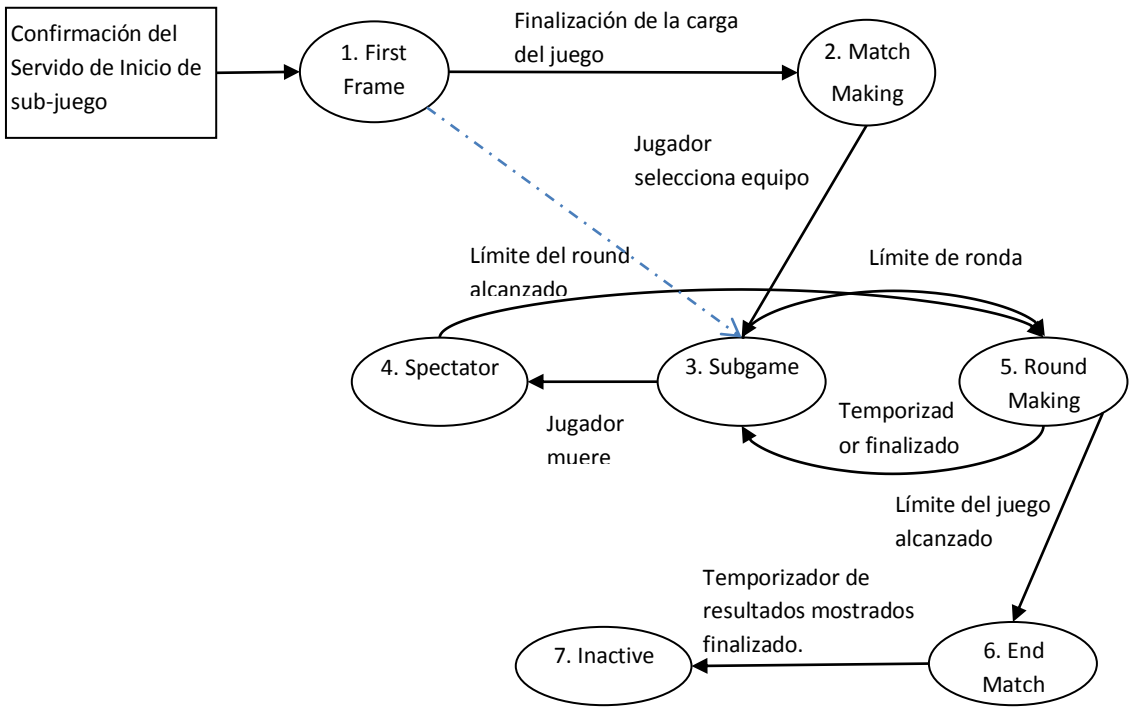


Figura 4.48. Maquina de estados para juego WarPipe en cliente

La Figura 4.48 muestra los estados para el cliente, los estados son:

1. First Frame. El cliente carga los modelos y animaciones del juego.
2. Match Making. El cliente selecciona el equipo que desea integrar.
3. Subgame. Es el estado donde se realiza la lógica y acción del juego.
4. Spectator. El cliente puede recorrer el mundo en cámara libro o siguiendo a algún otro personaje.
5. Round Making. Muestra los resultados de la ronda e inicia un temporizador antes de comenzar la siguiente ronda.
6. End Match. Muestra resultados de juego.
7. Inactive. Espera la destrucción del juego.

El siguiente código muestra la creación de los estados y un ejemplo de estado de WarPipe:

```
private void InitializeGameStates()
        {
            states = new States<SubgameState>();

            states.Add(SubgameState.FirstFrame, new State()
            {
                OnEnter = EnterFirstFrame,
                OnUpdate = UpdateFirstFrame,
                OnExit = ExitFirstFrame,
            });

            states.Add(SubgameState.Matchmaking, new State()
            {
                OnEnter = EnterMatchMaking,
                OnUpdate = UpdateMatchMaking,
                OnExit = ExitMatchMaking,
            });
.
.
.
            states.ChangeState(SubgameState.FirstFrame);
           }


public partial class WarPipeGame
    {
        private Boolean firstFrameLoading = true;

        private void EnterFirstFrame()
        {
             .
             .
        }
        private void UpdateFirstFrame()
        {
             .
             .
        }

        private Boolean ExitFirstFrame()
        {
             .
             .
        }
       }
```

La Figura 4.48 muestra los estados para el cliente, los estados son:

1. Idle. El juego se encuentre en espera de la información de partida.
2. WaitingPlayer. Se transfiere la información de la partida a los clientes.
3. Playing. Se realiza la ejecución del juego.
4. MatchoOver.- Espera a que todos los jugadores en estén en el estado inactivo en el cliente.

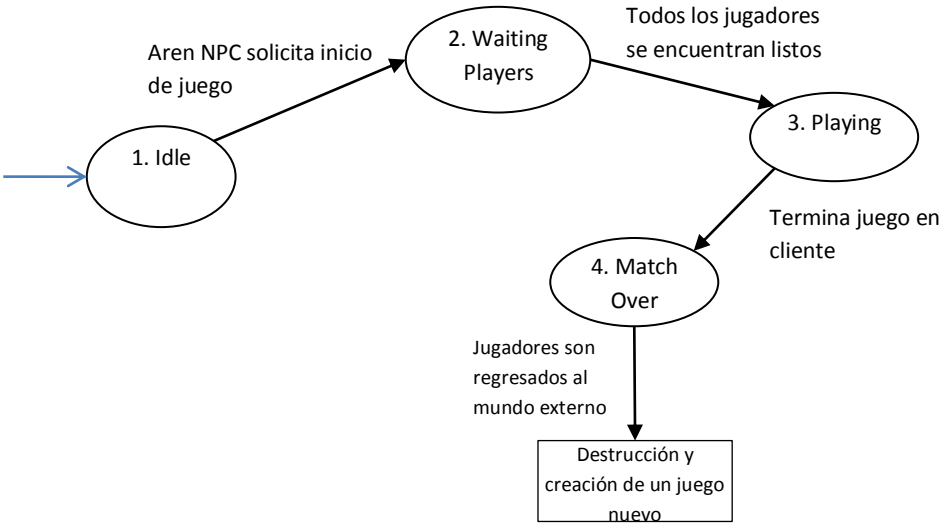


Figura 4.49. Máquina de estados para juego WarPipe en el servidor

11. Se creó una interfaz de usuario para la selección de equipo, los equipos pueden ser azul, amarillo o espectador (Figura 4.50).

Figura 4.50. GUI para la selección de equipo

12. Pantalla de Estadísticas. Se creó una pantalla para mostrar las estadísticas de la ronda y del juego completo (Figura 4.51).

Figura 4.51. GUI que muestra la información de estadísticas de ronda y de juego completo

13. Se agregó texto para mostrar información del jugador como: municiones restantes, equipo actual, vida y puntaje (Figura 4.52).

Figura 4.52. Información de jugador

La interfaz se modificó posteriormente con WPF, la versión final se muestra en la Figura 4.53, el número 1 muestra el tiempo restante, el 2 representa el número de victorias, el 3 es la munición restante y el 4 son las granadas restantes.

Figura 4.53. Versión final de información de jugador

4.4.2 Experimento 1

El objetivo del desarrollo de WarPipe fue realizar un experimento para estudiar el comportamiento de los jugadores en un shooter con cámara de tercera persona en red. Los objetivos que se analizan son:

- Como afectan las restricciones de comunicación a la habilidad de los jugadores en un shooter de 3er persona.
- Como se compara un shooter de tercera persona con un entrenamiento militar.

Se realizaron las siguientes modificaciones a WarPipe para obtener información relevante al experimento:

- Nuevo modo de juego

El modo de juego implementado para el experimento consiste en 7 jugadores que deben buscar y capturar 5 objetos del mundo, llamados McGuffins, en el menor tiempo posible. Los enemigos están limitados a 5 jugadores que restringen sus movimientos a ciertas zonas de WarPipe y las defienden.

Los jugadores son asignados rangos y únicamente se pueden comunicar por texto. Además, los rangos restringen la comunicación, posteriormente se describirá esto.

- Asignación de rangos

La Figura 4.54 muestra la interfaz de usuario donde se puede seleccionar el rango, los tipos de rango se muestran en el siguiente segmento de código:

```
public enum WarPipeRole
{
	NotDefined,
	Leader,
	Lieutenant,
	Soldier
}
```

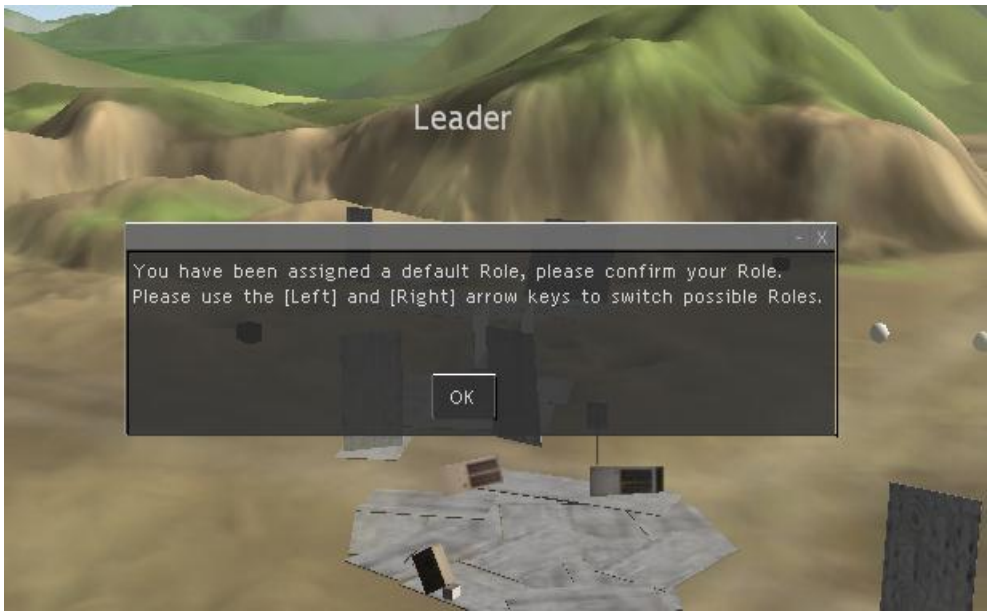


Figura 4.54. GUI para la selección de rangos

- Restricciones de comunicación

Los jugadores se pueden comunicar por proximidad con cualquier otro jugador del mismo equipo. Los jugadores que se comuniquen con otro jugador fuera del área de proximidad estarán restringidos a un patrón de comunicación.

A continuación se muestran los patrones de comunicación:

- Top Down. La comunicación es jerárquica. Se permite la comunicación de mayores rangos a menores (líder se comunica con teniente, teniente con subteniente, subteniente con

soldado, etc…). No se permite la comunicación entre mismas jerarquías, ni de menor a mayor jerarquía ni jerarquías de más de un nivel de diferencia.

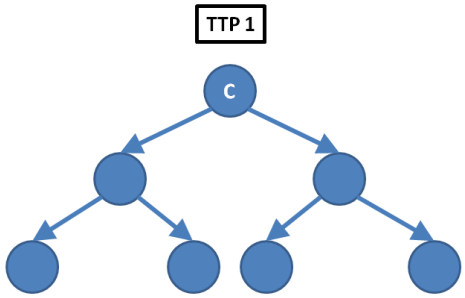


Figura 4.55. Patrón de comunicación Top Down

- Top Down Peer. La comunicación es similar a Top Down, pero también se permite la comunicación con jugadores del mismo rango.

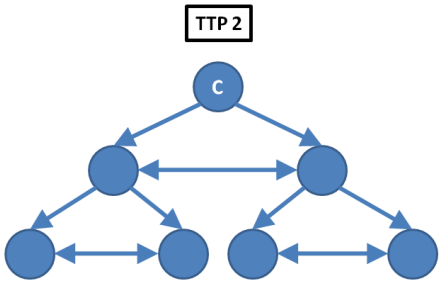


Figura 4.56. Patrón de comunicación Top Down

- All. Permite la comunicación con cualquier jugador.

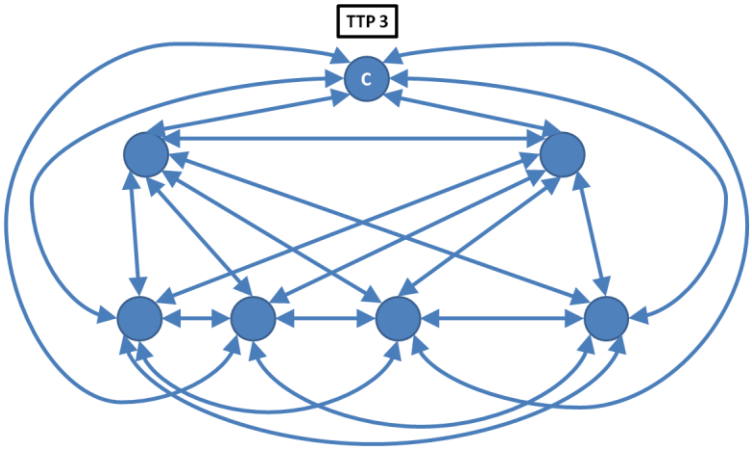


Figura 4.57. Patrón de comunicación All

A continuación se muestra un segmenteo de código de los patrones de comunicación:

```
    public enum WarPipeCommunicationHierarchy
    {
        TD,             // Top Down
        TDP,            // Top Down + Peer-to-peer
        O               // Open
    }

    public class WarPipeCommunications
    {
        public WarPipeCommunicationHierarchy Hierarchy;
        private float proximityRadius;

        public WarPipeCommunications(WarPipeCommunicationHierarchy
communicationHierarchy, float radius)
        {
            Hierarchy = communicationHierarchy;
            proximityRadius = radius;
        }

        public bool sendMessage(WarPipePlayer originatingPlayer,
WarPipePlayer otherPlayer, WarPipeCommunicationType communicationType,
            Vector3 initialPosition, Vector3 sendingPosition,out
CommunicationErroResult errorResult)
        {
            .
            .
            .

            switch (Hierarchy)
            {
                case WarPipeCommunicationHierarchy.TD:
                    result = processTDChat(originatingPlayer,
otherPlayer, communicationType);
                    break;

                case WarPipeCommunicationHierarchy.TDP:
                    result = processTDPChat(originatingPlayer,
otherPlayer, communicationType);
                    break;

                case WarPipeCommunicationHierarchy.O:
                    result = processOChat(originatingPlayer, otherPlayer,
communicationType);
                    break;

                default:
                    result = processOChat(originatingPlayer, otherPlayer,
communicationType);
                    break;
            }

            // Check proximity chat
            if (result == false)
            {
                result = proximityComm(initialPosition, sendingPosition);
```

```
                if (result == false)
                {
                    errorResult =
CommunicationErroResult.HierarchyNotAllowed;
                }
            }

            return result;
        }

        private bool processTDChat(WarPipePlayer originatingPlayer,
WarPipePlayer otherPlayer, WarPipeCommunicationType communicationType)
        {
            .
            .
            .
        }
        private bool processTDPChat(WarPipePlayer originatingPlayer,
WarPipePlayer otherPlayer, WarPipeCommunicationType communicationType)
        {
            .
            .
            .
        }
        private bool processOChat(WarPipePlayer originatingPlayer,
WarPipePlayer otherPlayer, WarPipeCommunicationType communicationType)
        {
            .
            .
            .
        }

        private bool proximityComm(Vector3 initialPosition, Vector3
sendingPosition)
        {
            BoundingSphere commArea = new BoundingSphere(initialPosition,
proximityRadius);
            ContainmentType result = commArea.Contains(sendingPosition);

            if (result == ContainmentType.Contains)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
```

- McGuffins

  Los McGuffins son modelos 3D de cubos que los jugadores deben encontrar para finalizar el experimento. La posición de los McGuffins se define en el servidor, una vez que el cliente encuentra uno en WarPipe, éste cambia de color.

4.4.3 Plataforma

En la plataforma Cosmopolis se trabajó principalmente en el área de red, *gameplay foundations* y el motor de render.

4.4.3.1 Sistema de Confiabilidad

El subsistema de red de Cosmopolis no utiliza las herramientas de red del XNA Framework, en cambio, se utiliza System.Net y System.Net.Sockets del .Net Framework. Las razones por las cuales no se utiliza el XNA Framework son las siguientes:

- Únicamente se pueden hacer pruebas en red local sin suscripción a XNA Live Gold ni a XNA Creator Club.
- Número de jugadores limitado a 32.
- Menor control sobre protocolos y paquetes envidos.

Además, se decidió utilizar UDP en vez de TCP debido a:

- Menor información de cabeceras.
- Animaciones y movimientos de personajes no requieren confirmación de que el mensaje fue recibido.
- La conexión y envió de mensajes con el cliente se debe realizar los más rápido posible.

Sin embargo, al utilizar UDP nos presentamos con el problema de que no estamos seguros si el mensaje se recibió y el orden en que se recibió. Algunos mensajes se tienen que enviar en orden correcto y estar seguros de que se recibieron, por estas razones se implementó un sistema de confiabilidad basado en RUDP.

Reliable UDP (RUDP) es un protocolo simple de capa de transporte, basado en el RFCs 908 desarrollado en Bel Labs. El protocolo RUDP es utilizado cuando el protocolo UDP se vuelve insuficiente debido a la necesidad de enviar paquetes ordenados y con acuse de recibido. Además, el protocolo TCP representa una solución demasiado compleja para el sistema.

El RUDP es un protocolo que puede ser utilizado para enviar paquetes ordenados y confiables (con acuse de recibo). Además, RUDP proporciona elementos para mejorar la calidad del servicio como son: control de congestión, retransmisión, manejo de pérdida de paquetes,

Sus características son:

- Acuse de recibo por parte del cliente de paquetes enviados del servidor al cliente.

- Ventana deslizante y control de congestión, para que el servidor no exceda el ancho de banda disponible.
  - Retransmisión del servidor al cliente en caso de pérdida de paquetes.
  - Envió ordenado de paquetes.

1	2	3	4	5	6	7	8	16 bit
SYN	ACK	EAK	RST	NUL	CHK	TCS	0	Longitud de Cabecera
Numero de Secuencia								Ack number
Checksum								

Figura 4.58. Cabeceras RUDP v2

La Figura 4.58 muestra las cabeceras de RUDP que a continuación se describen:

- Bits de control – Indica que está presente en el paquete:
  - SYN: El bit SYN indica que una secuencia de sincronización está presente. ACK: El bit ACK indica si el número de *acknowledgment* en la cabecera es válido.
  - EACK: El bit EACK indica si un aknowledge extendido está presente. Utilizado para paquetes ordenados o desordenados
  - RST: El bit RST indica si el paquete es un reset
  - NUL: El bit NUL indica si el paquete contiene un segmento nulo
  - CHK: El bit CHK indica si el Checksum contiene información valida o no.
  - TCS: El TCS bit indica si el paquete es un estado de conexión.
  - 0: El valor siempre debe ser cero.
- Longitud de cabecera: Indica donde comienzan los datos en el paquete
- Numero de Secuencia: Número de secuencia del paquete
- Ack number: Esta cabecera indica el número de secuencia del último paquete recibido.
- Checksum: Utilizado para verificar la integridad de los datos

- Sistema Implementado

Se implementó un sistema que agrega información a los mensajes enviados, semejante al protocolo RUDP.

Sequence Number	Sequence Channel	Acknowledgment number	Acknowledge Channel	EACK

- Sequence Number, 32bits – Número de secuencia del paquete
- Sequence Channel, 32bits – Canal de transmisión asociado a la secuencia
- Acknowledgment Number, 32bits – Número de confirmación de paquete recibido. 0 representa un numero inválido
- Acknowledgment Channel, 32bits – Canal de transmisión asociado al paquete confirmado

- EACK, 1bit – Indica si la confirmación es de un paquete que se recibió en orden o en desorden.

- Estructuras de Datos Relevantes

  Almacenamiento de Mensajes Enviados: Se utiliza en caso de que sea necesario reenviar un mensaje, solo se utiliza para canales confiables.

  Almacenamiento de Mensajes Recibidos: Los mensajes se guardan por si se reciben mensajes repetidos, lo mensajes son eliminados después de un tiempo.

  Almacenamiento de Mensajes Desordenados: Para canales ordenados, se guardan los mensajes que se reciben en desorden para su posterior procesamiento.

- Características Adicionales

  Los mensajes que pertenecen a canales confiables son guardados hasta recibir su confirmación. En caso de no recibir confirmación, serán vueltos a enviar en determinado tiempo. En caso de una desconexión del cliente o servidor, todos los mensajes guardados dirigidos al cliente o servidor serán eliminados.

  Si el receptor no envía ningún mensaje, esto implica que no se podrán enviar confirmaciones de mensajes recibidos. Para resolver este problema se realiza lo siguiente:

  a. Temporizador de Transmisión: Se envían mensajes de confirmación si el tiempo de espera ha excedido.
  b. Contador de Retransmisión: Se mandan mensajes de confirmación en caso de que se llegue a un límite de mensajes por confirmar guardados.

4.4.3.2 Sistema de Heart Beat

Un problema que tenemos con UDP es que no es un protocolo orientado a la conexión, esto quiere decir que no sabes cuando el cliente o servidor se desconectan. Para resolver la desconexión de cliente o servidor se implementó un sistema de *heart beat*.

El sistema de heart beat consiste enviar un mensaje de HearBeat cada 30seg. Se utiliza un reloj para tomar el tiempo que ha transcurrido desde el último mensaje recibido (incluyendo el HeartBeat), en caso de sobrepasar el límite de tiempo, se realiza una desconexión del cliente o servidor.

El siguiente segmento de código muestra la desconección de un EndPoint debido a que no se han recibido mensajes en un cierto tiempo:

```
foreach (var endPoint in endPointMapping.Keys.ToList())
        {
                if ((DateTime.Now - timeOfLastMessage[endPoint]).TotalSeconds >
        noMessageTimerToDisconnect)
                {
                        // Disconnect End Point
                        Console.WriteLine("Havent received any messages (including
                Heart Beat) from player: {0}", endPointMapping[endPoint]);
                        Disconnect(endPoint);
                }
        }
```

4.4.3.3 Administración de Juegos

En la sección 4.3.1.9 se describió el administrador que se implementó en Cosmopolis, en esta sección se describirá con más detalle la creación de juegos.

Para el cliente, el servidor envía un mensaje que inicia la creación de un juego, este mensaje debe contener las características del juego, por ejemplo: en el juego WarPipe se recibe un mensaje `MessageType.WarPipeSubgame` y contiene la siguiente información:

- FriendlyFire: Indica si el friendly fire esta habilitado.
- Id: Identificador unico del juego.
- Name: Nombre del juego.
- CityId: Identificador de la ciudad.
- HostId: Identificador del servidor.

Messages::**WarPipeSubgame**
+FriendlyFire : bool -Id : int -Name : string -CityId : int -HostId : int
+TimesPerSecond() : float +WarPipeSubgame() +WarPipeSubgame(entrada id : uint, entrada name : string, entrada cityId : uint, entrada hostId : uint, entrada footprint : Rectangle, entrada friendlyFire : bool) +WarPipeSubgame(entrada bytes : byte[], entrada y salida startIndex : int) +WarPipeSubgame(entrada baseClass : Subgame) +WarPipeSubgame(entrada warpipesubgame : WarPipeSubgame) +ToBytes() : byte[] +FromBytes(entrada bytes : byte[], entrada y salida startIndex : int) +ToSqlString(entrada playerID : uint, entrada endPoint : EndPoint, entrada dateTime : DateTime) : string +ToString() : string

En el servidor siempre está presente un juego pero se encuentra inactivo. Se utiliza System.Reflection que permite la creación de objetos en tiempo de ejecución a partir de su tipo. Para la creación de un juego en el servidor necesitamos: tipo de juego, el controlador del juego y sus propiedades. En el siguiente segmento de código se muestra la creación de un juego:

```
/// <summary>
 /// Creates a new Subgame if the current one is not active or hasnt been created
 /// </summary>
 public ISubgame CreateSubgame()
 {
     if (Helper.Assert(Subgame == null, "Subgame is not null"))
     {
        GameSettings.UpdateIds(SubgameManager.Engine);
        ConstructorInfo subgameConstructor = SubgameType.GetConstructor(new Type[] {
        typeof(IEngine), typeof(ISubgameController), typeof(ISubgameSettings) });

        if (Helper.Assert(subgameConstructor != null, "Couldnt get constructor for
        Subgame "))
        {
                Subgame = (ISubgame)subgameConstructor.Invoke(new Object[] {
                SubgameManager.Engine, SubgameController, GameSettings });
                Helper.Assert(Subgame != null, "Couldnt create Subgame from
                constructor");
         }
         return Subgame;
     }
     return null;
}
```

Las características de un juego se definen en SubgameSettings, en el caso de warpipe WarPipeSettings hereda de SubgameSettings (Figura 4.59) e incluyen la siguiente información:

- FriendlyFire: Indica si el friendly fire esta habilitado.
- KillCount: Número máximo de muertes.
- KillCountEnabled: Si la terminación de partida por número de muertes está habilitada.
- MatchTime: Tiempo máximo de partida.
- TimeIsEnabled: Si el tiempo está habilitado.
- SpawnPositions: Diccionario con los equipos y su lista de puntos de inicio.

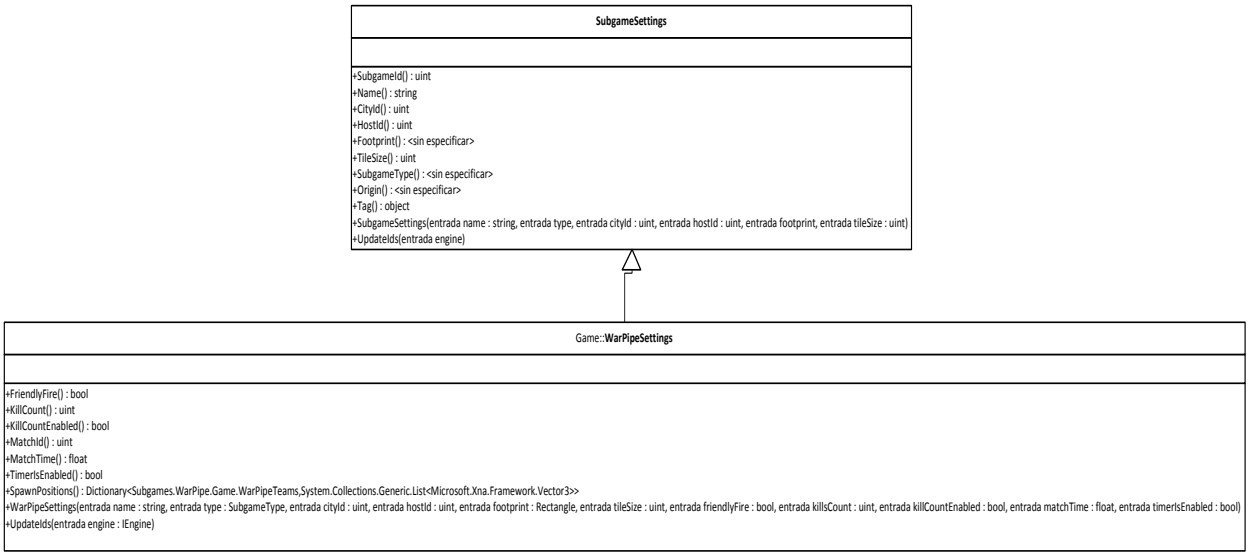


Figura 4.59. Modelo UML de SubgameSettings y WarPipeSettings

4.4.3.4 Sistema de Amigos, Guilds y Party

Es un sistema que permite guardar la información de los amigos, equipo y sociedades a la que el jugador pertenece. El sistema aún no se utiliza pero por el momento la funcionalidad existe para guardar y leer la información de archivos XML.

4.4.3.5 Información 3D

El juego de WarPipe fue el primero juego en tener la necesidad de mostrar el nombre del jugador arriba del avatar. El problema se resolvió mostrando el nombre como una imagen 2D encima de todos los elementos de juego. Sin embargo, se decidió implementar un texto 3D, que tenga una posición en el mundo virtual y por ende se pueda ocultar por los objetos del mundo.

Esta nueva funcionalidad se implementó en la plataforma para que funcione en el mundo externo y en todos los juegos. En el siguiente código muestra como el nombre de los jugadores se dibuja en una imagen 2D:

```
Engine.Graphics.SetRenderTarget(0, billboardTarget[player.Id]);
Engine.Graphics.Clear(ClearOptions.Target | ClearOptions.DepthBuffer,
Color.TransparentWhite, 1, 0);

SpriteBatch.Begin(SpriteBlendMode.AlphaBlend, SpriteSortMode.FrontToBack,
SaveStateMode.SaveState);
var textMeasure = nameFont.MeasureString(player.Name);
SpriteBatch.DrawString(nameFont, player.Item2,
       new Vector2(fontWidth / 2 - textMeasure.X / 2, fontHeight / 2 -
       textMeasure.Y / 2), Color.White);

SpriteBatch.End();
```

Posteriormente, la imagen se utiliza como la textura de un modelo de un plano 3D y se usa como billboard para que el plano siempre se vea por la cámara del juego.

Asignación de la imagen como textura difusa del modelo NameplateCP_{Id}:

```
Material mat = Assets.Common.Shapes.NameplateCP_Model.CreateOrGet
(String.Format("NameplateCP_{0}",player.Id)).ModelMeshes[0].Material;
mat.DiffuseMap = billboardTarget[player.Id].GetTexture();
```

Creación de la matriz de billboard para que el modelo siempre vea hacia la cámara y dibujado de modelo:

```
Matrix billboardMat = Matrix.CreateBillboard(player.Position + new
Vector3(0, offsetY, 0),camPos, Engine.Camera.Up, Engine.Camera.Forward);
Engine.Values.Draw.AddModel(player.ModelBillboard, billboardMat, false,
false);
```

5. Conclusiones

Los avances en el desarrollo de mundos virtuales abrieron las puertas a nuevas formas de investigación, entre ellas los modelos sociales, económicos y de comportamiento. En la actualidad, el uso de mundos virtuales para el estudio estos modelos es escaso, pero en los últimos años se ha demostrado su utilidad.

Los mundos virtuales deben facilitar al investigador la creación de ambientes para la investigación y la obtención de resultados, además, deben mantener el interés del jugador para tener grandes cantidades de sujetos experimentales. El desarrollo de juegos de calidad y el uso de buenas herramientas tecnológicas es indispensable para desarrollar los juegos.

El desarrollo de videojuegos tiene un proceso bien definido en la industria de juegos, cada etapa del proceso genera resultados muy importantes que son utilizados en etapas posteriores. El tener una secuencia de etapas para el desarrollo de un juego, facilita su creación y da consistencia al producto final. En el presente trabajo se adquirió una gran experiencia en el proceso de desarrollo de juegos y en especial, el desarrollo de un motor de juegos y los subsistemas que lo integran.

La plataforma de Cosmopolis es sin duda un proyecto de amplias dimensiones y de mucha complejidad. Cosmopolis no fue desarrollado para un género específico de juego, debido a la necesidad de crear diferentes tipos juegos para satisfacer las necesidades de los investigadores, y la creación de un mundo virtual que funciona con una gran cantidad de jugadores en línea complicaron el desarrollo de esta plataforma. Además, es importante desarrollar código de calidad, que cumpla con los principios de programación orientados a objetos. Este proyecto me permitió mejorar la calidad de mis códigos, conocer más reglas y patrones del desarrollo de software.

Mi aportación en Cosmopolis se realizó en diferentes áreas de la plataforma lo que permitió conocer los diferentes subsistemas utilizados en un motor de juegos y participar en su desarrollo. Se trabajó principalmente en el juego WarPipe, el área de redes, gamplay foundations y el motor de render.

El juego WarPipe fue modificado varias veces para agregar funcionalidad específica a los experimentos y para cumplir con los requisitos (en constante cambio); sin embargo, gracias a la plataforma de Cosmopolis la modificación del juego fue sencilla. Se crearon una gran cantidad de mensajes para sincronizar el cliente y el servidor, además, se implementaron máquinas de estados que permitirán una mayor organización del código y facilitar su entendimiento.

El sistema de red represento un gran reto ya que se implementó un sistema de confiabilidad que funcionaba de manera similar al protocolo RUDP. Este sistema me permitió usar los conocimientos de la materia de Redes de Computadoras y aplicarlos para la resolución de problemas. También me permitió involucrarme en un subsistema de gran importancia en el género de los MMOG. Se

resolvieron problemas como la desconexión abrupta de un cliente o el servidor y manejarla de la mejor manera posible; se trabajó en problemas de tiempos de respuesta y latencia.

En el subsistema del motor de render se implementó información 3D, pero para poder modificar este sistema fue necesario entender su funcionamiento, esto me permitió estudiar más a fondo éste subsistema. El motor de render se actualiza dentro de un hilo, esto permite optimizar el dibujado; además, utiliza una estructura que facilita la creación de nuevos efectos y separa claramente los datos que recibe cada shader.

El trabajo que se realizó en el gamplay-foundations fue el desarrollo del administrador de juegos y sistemas de amigos, equipos y clanes. Mediante el desarrollo del juego WarPipe pude entender algunas necesidades de los juegos e integrar nuevos sistemas a la plataforma Cosmopolis que facilitarán el desarrollo de juegos.

Los subsistemas del motor de juegos pueden llegar a ser extremadamente complejos, debido a esto es posible especializarse en alguno de ellos. En el subsistema de redes se requieren especialistas con técnicas de compresión y seguridad; en el subsistema de motor de render se requiere técnicas de sombras dinámicas para mundos extensos; de igual manera otros subsistemas de la plataforma requieren de gente especializada para resolver problemas cada vez más complejos.

Otra área que requiere mayor trabajo es el gamplay-foundations. A pesar de que Cosmopolis provee herramientas que facilitan la creación de juegos, se necesitan desarrollar herramientas dirigidas a investigadores, ya que, en el estado actual de la plataforma, para crear un juego es casi imprescindible contar con un programador para implementarlo.

La plataforma Cosmopolis aún se encuentra en desarrollo, sin embargo la creciente complejidad del sistema requiere de personas que se especialicen más en cada subsistema. Esta especialización es un paso necesario para entrar a la industria de los juegos, ya que los actuales juegos comerciales son desarrollados por una gran cantidad de programadores (entre otros profesionales) altamente especializados.

6. Referencias

Libros

Bates, B. 2004. *Game Design.* Segunda edición. Thomson Course Technology PTR, 2004.

Carter, Chad. 2009. *Microsoft XNA Game Studio 3.0 Unleashed .* SAMS, 2009.

D. Michael, S. Chen. 2006. *Serious Games: Game That Educate, Train and Inform.* Thomson, 2006.

J. Flymnt, et. al. 2005. *Software Engineering for Game Developers.* Thomson, 2005.

J.Gregory. 2009. *Game Engine Architecture.* A K Peters, 2009.

Joseph Albahari, Ben Albahari. 2010. *C# 4.0 In A Nutshell, The Definitive Reference.* 4. OReilly, 2010.

Kent, Steven L. 2001. *The Ultimate History of Video Games.* Three Rivers Press, 2001.

Luna, Frank D. 2006. *Introduction to 3D Game Programming with DirectX 9.0c, A Shader Approach.* Wordware, 2006.

M. Robert, M. Micah. 2006. *Agile Principles, Patterns, and Practices in C#.* Prentice Hall, 2006.

McGee, Stephen Cawood y Pat. 2009. *XNA Game Studio Creators Guide.* 2nd. McGraw Hill, 2009.

Sommerville, Ian. 2005. *Ingeniería del Software.* Séptima Edición. Addison Wesley, 2005.

Zacharias, Greg L., Jean MacMillan, et. al. 2008. *Behavioral Modeling and Simulation: From Individuals to Societies.* National Academies Press. 2008.

Castronova. 2006. *Synthetic Worlds: The Business and Culture of Online Games.* University of Chicago Press, 2006.

E. Gamma, R. Helm, R. Johnson, J. Vlissides. 1995. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

Páginas de Internet

López, Javi. 2009. XNA Framework 2.0 y 3.0: DirectX y Direct3D en C# para mortales . 2009. http://forum.winjanet.66ghz.com/index.php?topic=135.0.

Microsoft. 2011. XNA Game Studio 4.0. 2011. http://msdn.microsoft.com/en-us/library/bb200104.aspx.

butunclebob. 2010. *Principles of Object Oriented Design.* 2010. http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

Zogrim. 2009. Popular Physics Engines comparison: PhysX, Havok and ODE. 2009. http://physxinfo.com/articles/?page_id=154.

Games, Epic. 2010. Current technology – Unreal Engine 3. 2010. http://www.unreal.com/features.php?ref=technology-overview .

2007. Tecnologias Microsoft. 2007. http://tuyub.wordpress.com/2007/11/11/xna-game-studio/.

Walker, Mitch. 2006. What is the XNA Framework. 2006. http://blogs.msdn.com/b/xna/archive/2006/08/25/724607.aspx.

Pettineo, Matt. 2010. *Pix With XNA Tutorial.* 2010. http://mynameismjp.wordpress.com/samples-tutorials-tools/pix-with-xna-tutorial/.

Scapecode. 2010. *Un-Managed Code.* 2010. http://scapecode.com/category/net/unamanged-code/.

Museum, Computer History. Computer History. http://pdp-1.computerhistory.org/pdp-1/index.php?f=theme&s=1.

Gregory, J. 2010. Multiprocessor Game Loops: Uncharted. 2010. http://www.gameenginebook.com/gfg2010-final.pdf.

Havok. 2010. Havok. 2010. http://www.havok.com/index.php?page=havok-physics.

Initiative, Serious Games. 2002. 2002. http://www.seriousgames.org/.

J. Ward. 2008. http://www.gamecareerguide.com/features/529/what_is_a_game_.php?page=1.

Artículos

Yee, Nick and Jeremy Bailenson. 2007. *"The Proteus Effect: The Effect of Transformed Self-Representation on Behavior".* Human Communication Research 33, págs. 271-290.

Castranova. 2008. *E. A Test of the Law of Demand in a Virtual World: Exploring the Petri Dish Approach to Social Science.* Indiana University. 2008.

Liskov, Barbara. 1998. *Data Abstraction and Hierarchy.* SIGPLAN Notices.

DeMarco, T. 1979. *Structured Analysis and System Specification.* Yourdon Press Computing Series, 1979.

Llopis, N. 2008. Programming with Abstract Interfaces. [aut. libro] M. DeLoura. *Best of Game Programming Gems.* Course Technology, 2008, págs. 41 - 48.

Michael Zyda, Peter Landwehr, Marc Spraragen, Balakrishnan Ranganathan. **2009**. "Designing a Massively Multiplayer Online Game as a Testbed for Social and Behavioral Model Research". SIGGRAPH.

Michael Zyda, Marc Spraragen, Balakrishnan Ranganathan. **2009.** "Testing Behavioral Models with an Online Game". IEE Computer, vol. 42, no. 4, pp. 103-105.

Peter Landwehr, Marc Spraragen, Balakrishnan Ranganathan, Michael Zyda, Carley Kathleen M, Jason Hong. 2010. "Planning a Cosmopolis: Key Features of an MMOG for Social Science Research". *CHI 2010 Workshop: Video Games As Research Instrucments, Atlanta, Georgia: ACM, 2010.*