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).

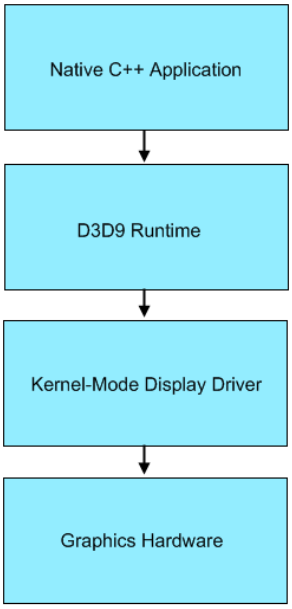


Figura 4.1. Interacción de una aplicación C++ con el hardware gráfico

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.

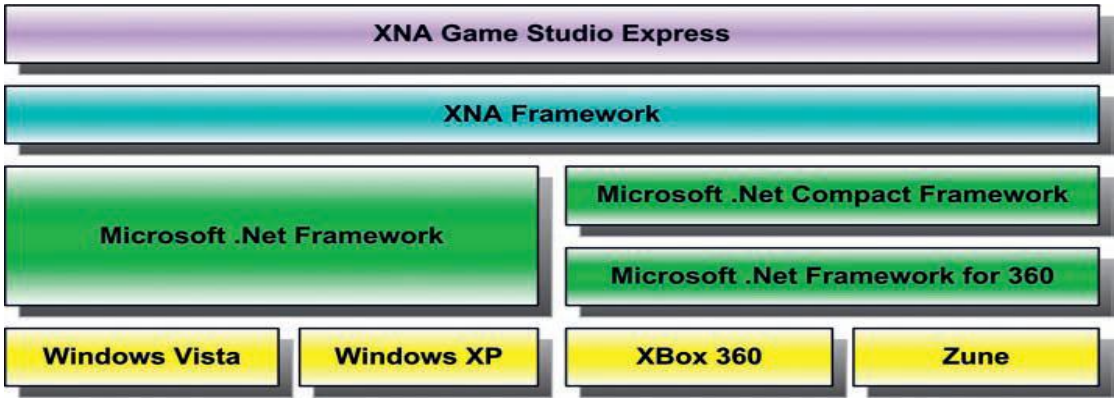


Figura 4.2. Arquitectura XNA

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

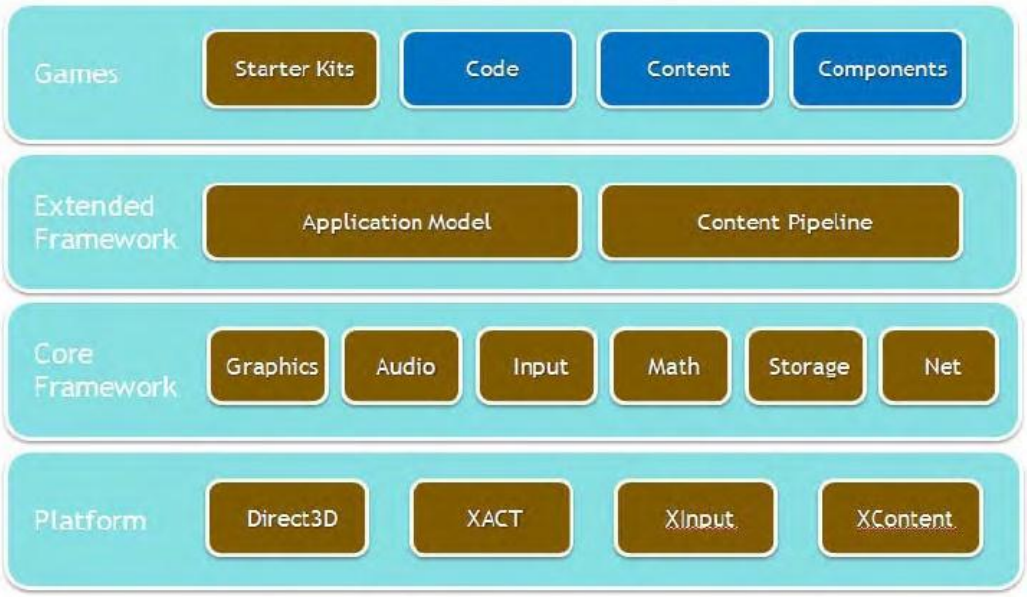


Figura 4.3. Capas del XNA Framework

- 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.

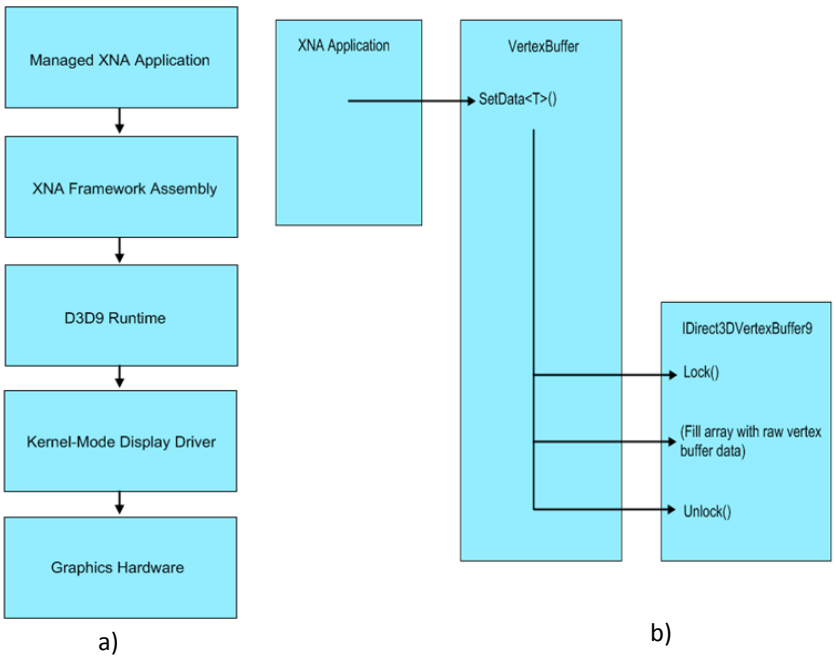


Figura 4.4. a) Interacción de una aplicación XNA con el hardware gráfico. b) Transformación de un método de XNA Framework al API Direct3D

- 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).

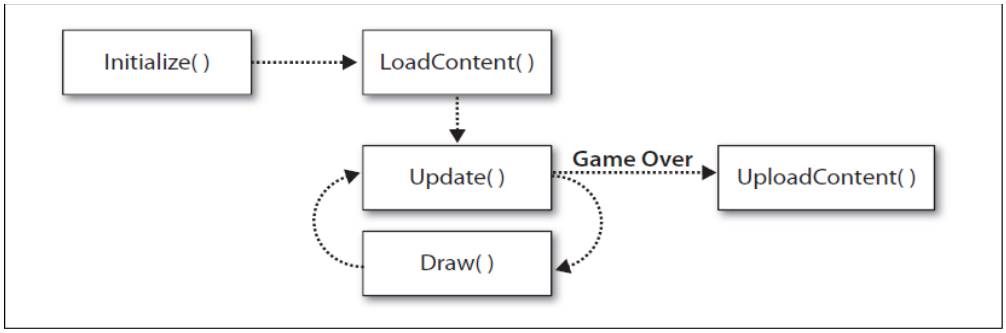


Figura 4.5. Application Model de XNA

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

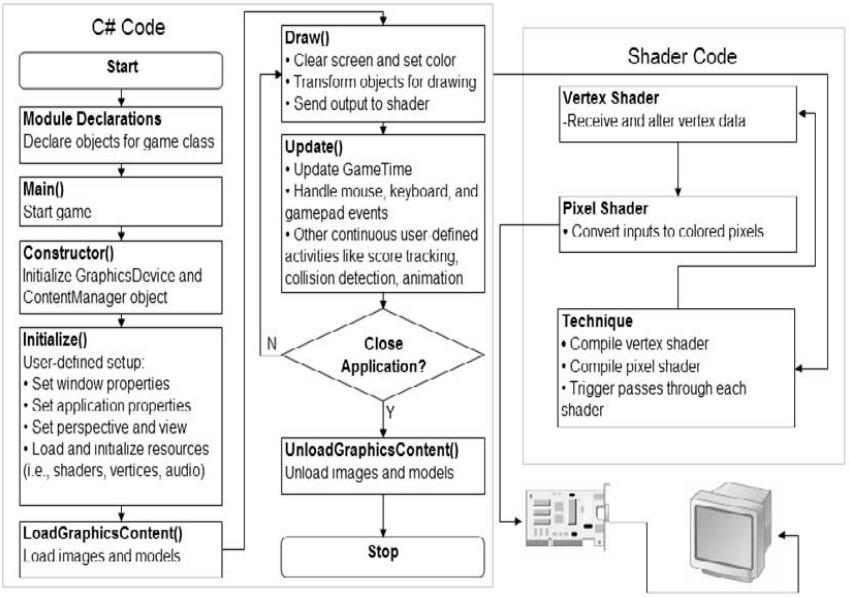


Figura 4.6. Application Model de XNA con aceleración de hardware

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.

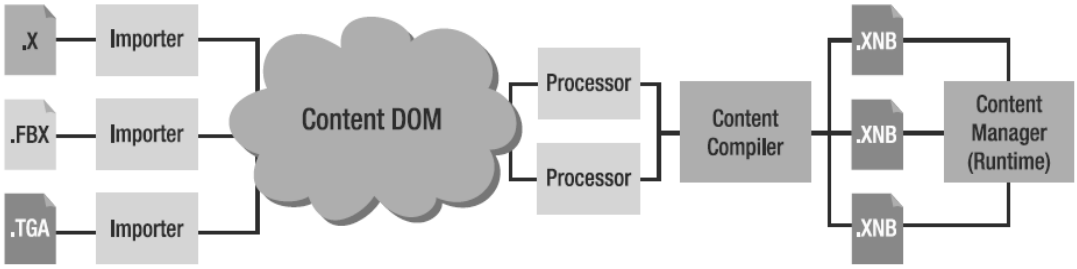


Figura 4.7. XNA Content Pipeline

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.

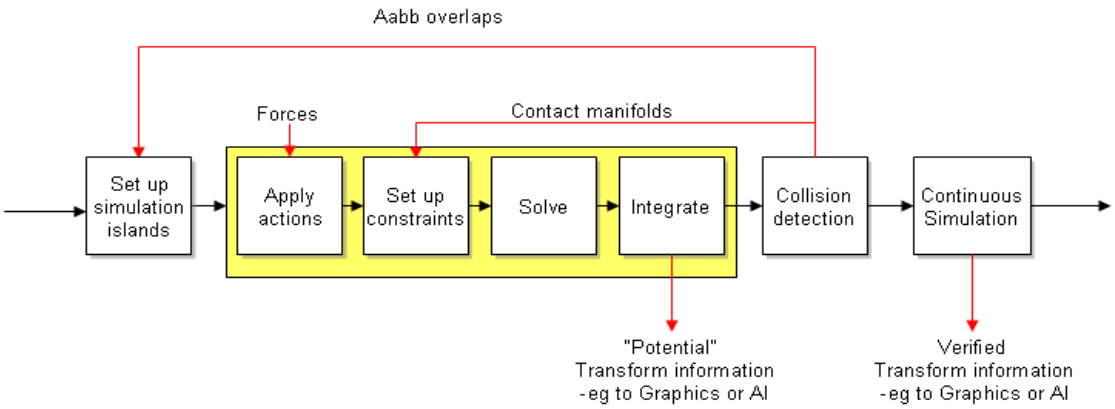


Figura 4.8 Flujo de simulación de Havok

- *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".

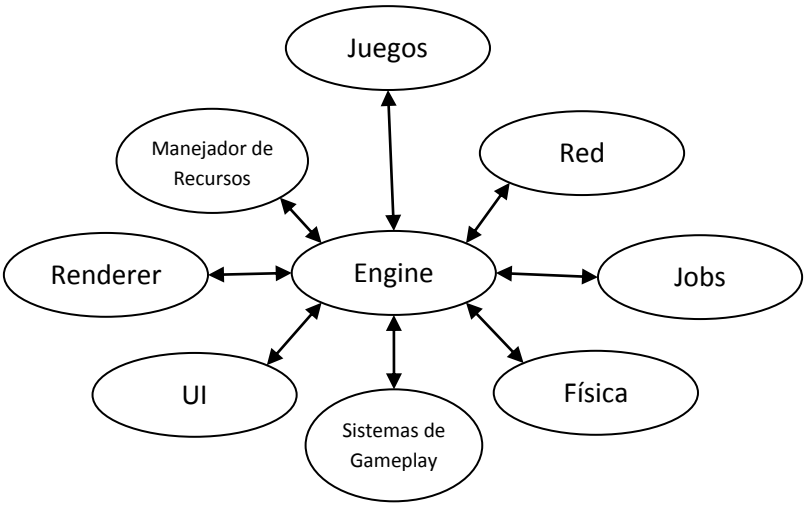


Figura 4.9. Sistema de control centralizado.

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.

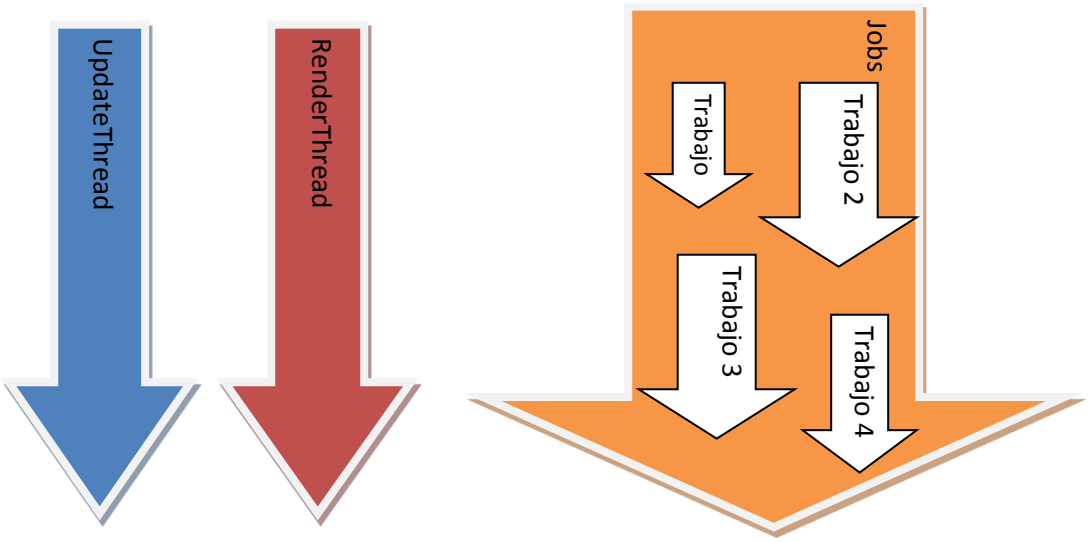


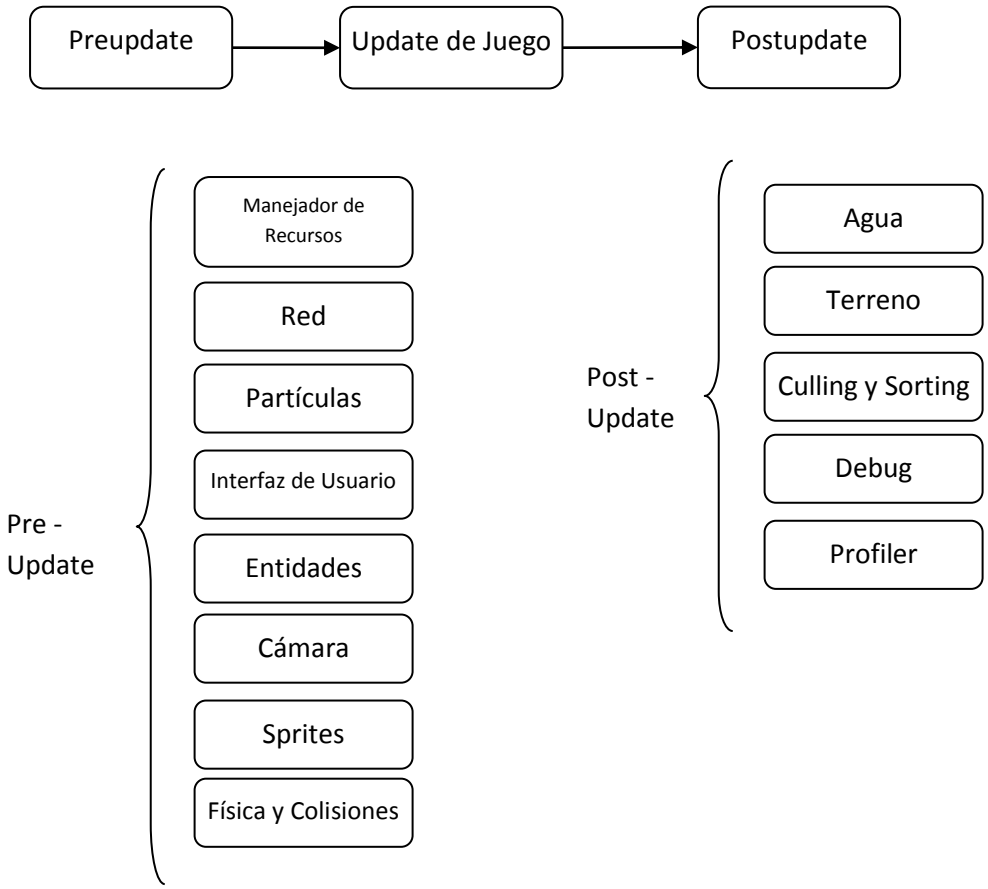
Figura 4.11. Uso de 2 hilos para Render y Update, y un sistema de trabajos.

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

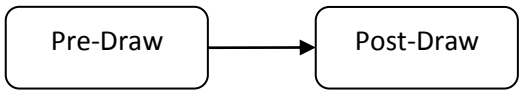
El ciclo de actualización se realiza durante la función de *Update()* de XNA. Para aprovechar las computadoras de varios núcleos, se separa el proceso de rendering y la actualización de la lógica del juego en 2 hilos distintos, a continuación se muestra el código usado para la creación de los hilos:

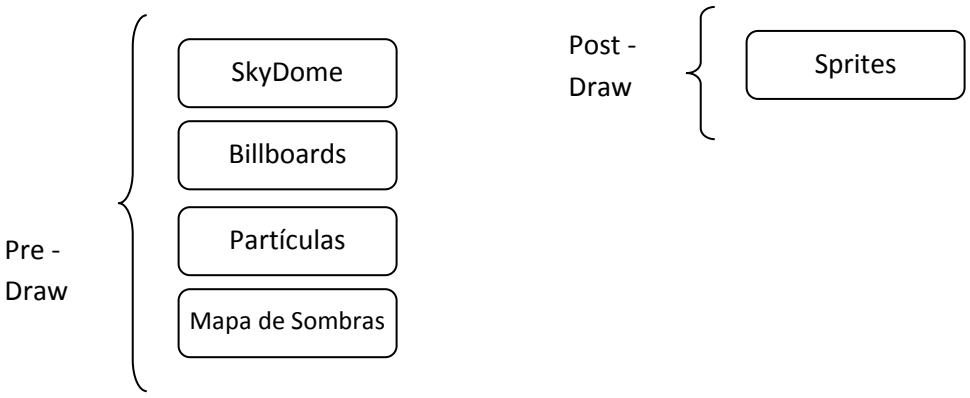
```
protected void InitializeThreading(String updateThreadName, String
drawThreadName)
        {
              .
              .
              .

            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).

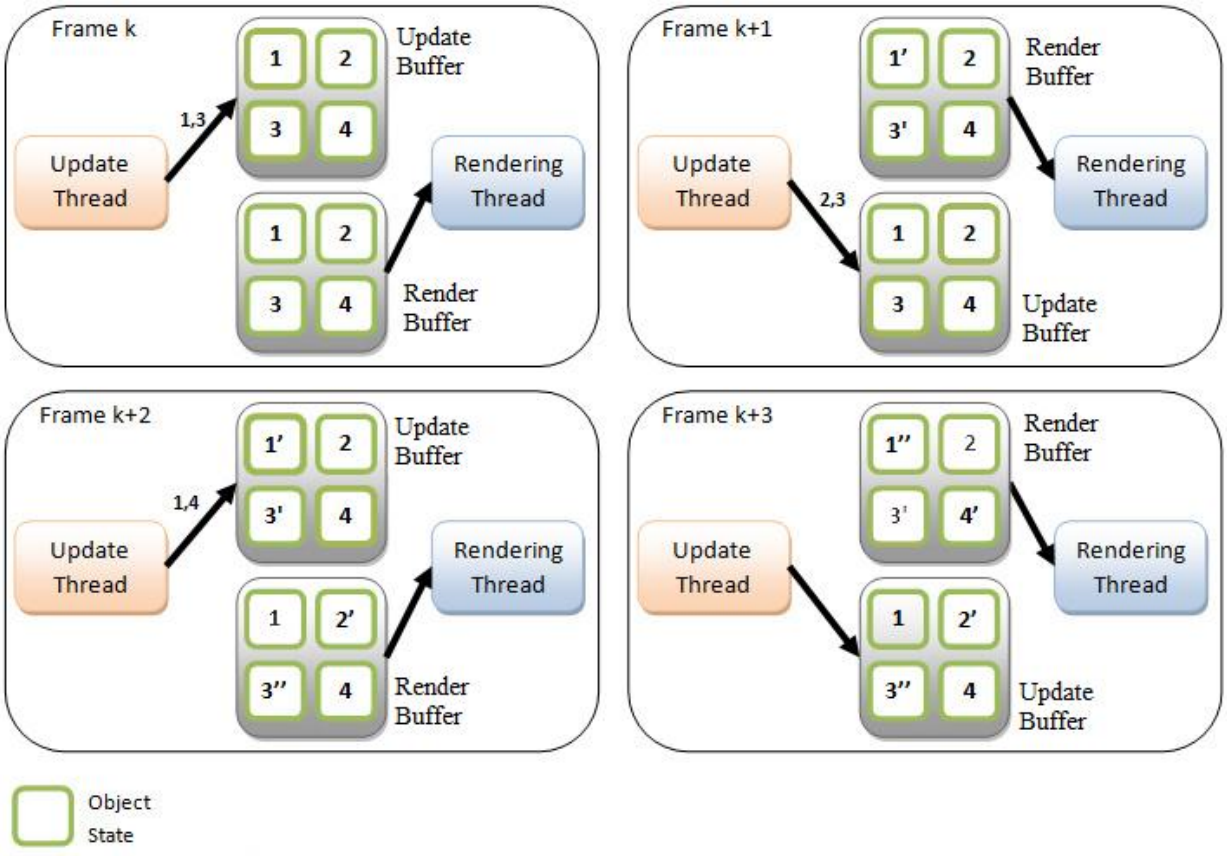


Figura 4.12. Actualización e intercambio de buffers

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

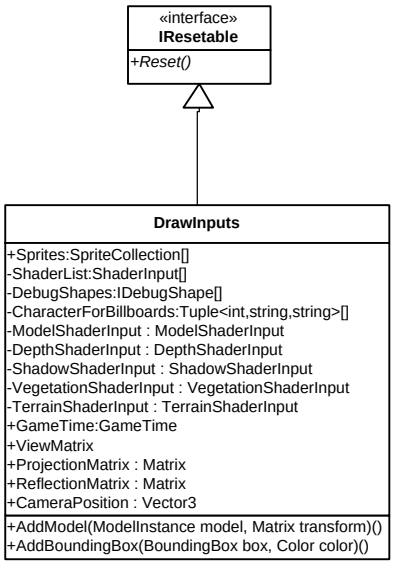


Figura 4.13. Modelo UML de la clase DrawInputs

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.

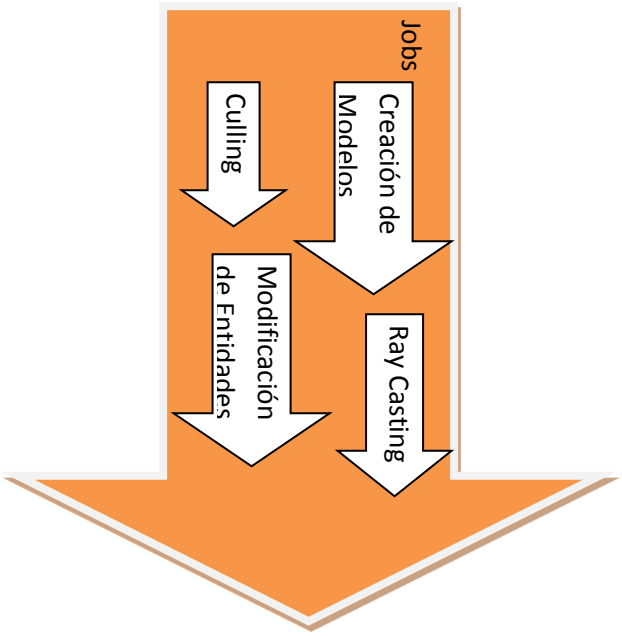


Figura 4.14. Sistema de Trabajos

4.3.1.2 Recursos de Juego

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

- Recursos:

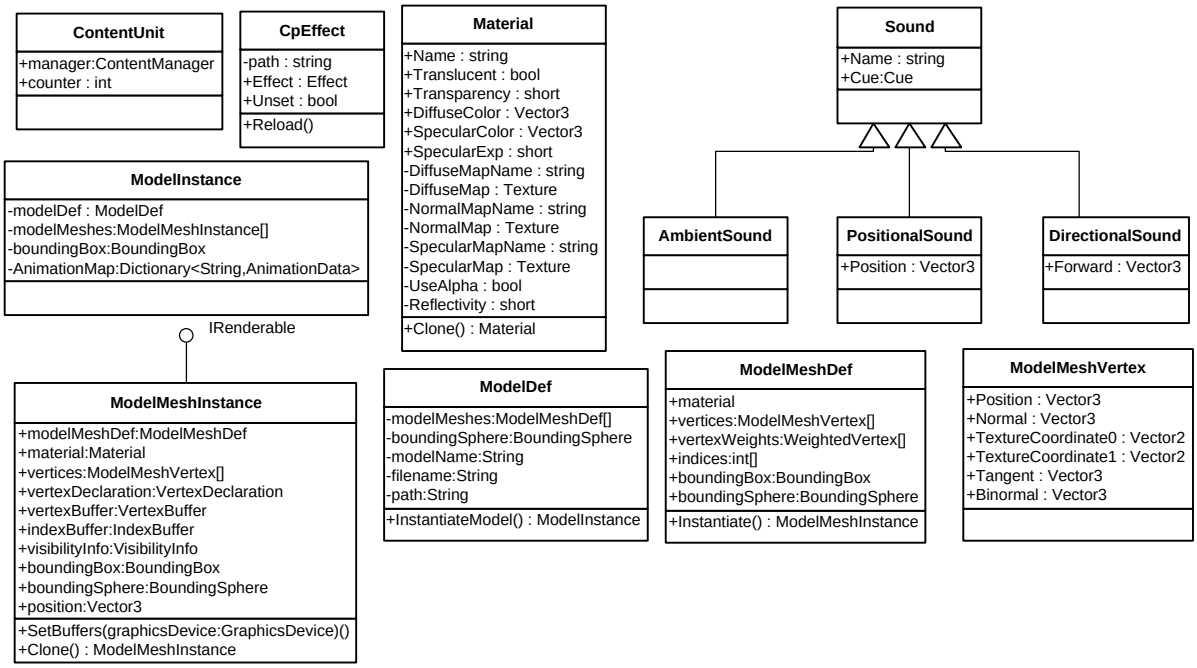


Figura 4.15. Modelos UML de las clases que guardan los recursos

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

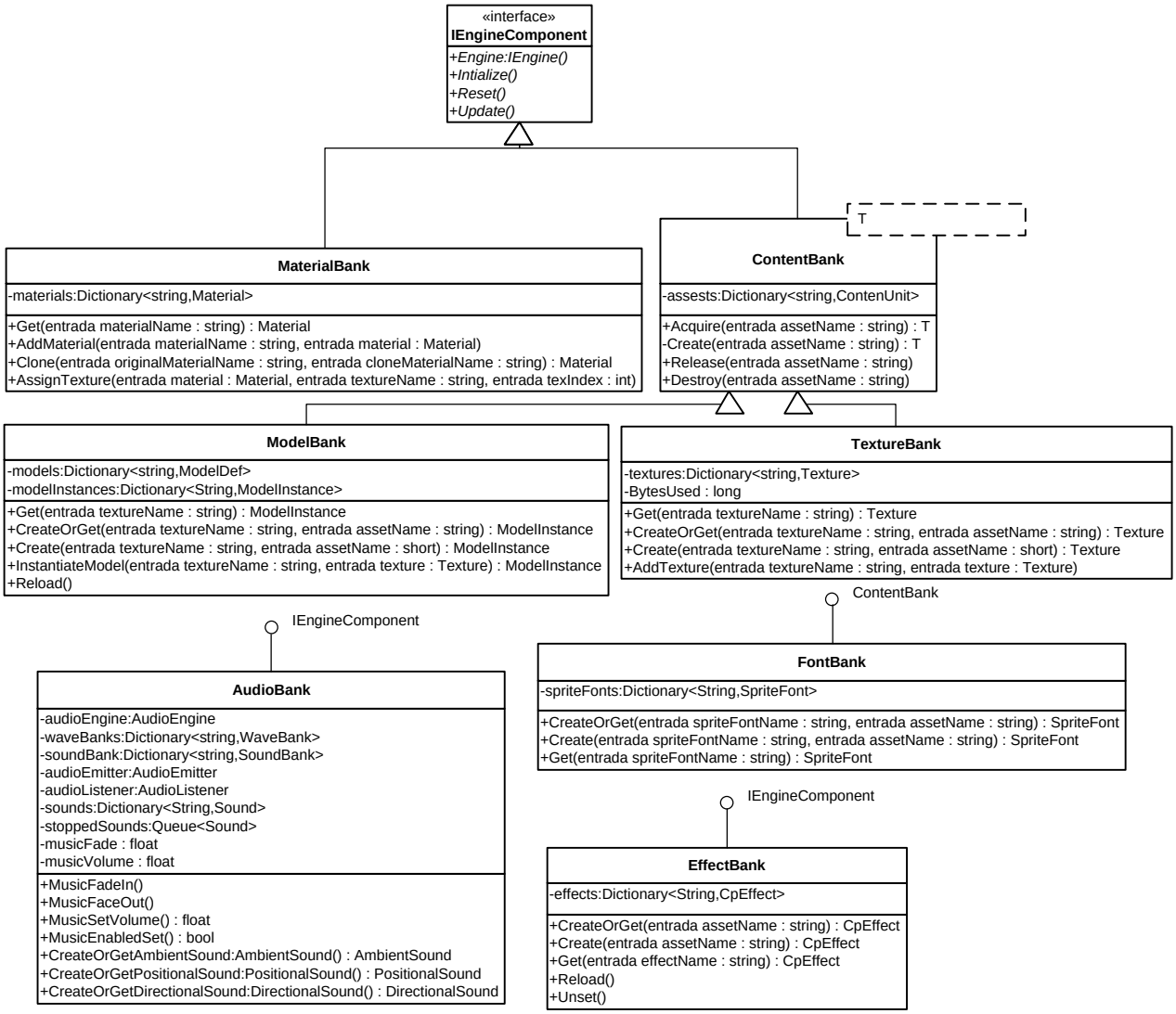


Figura 4.16. Modelos UML de clases que administran los recursos

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.

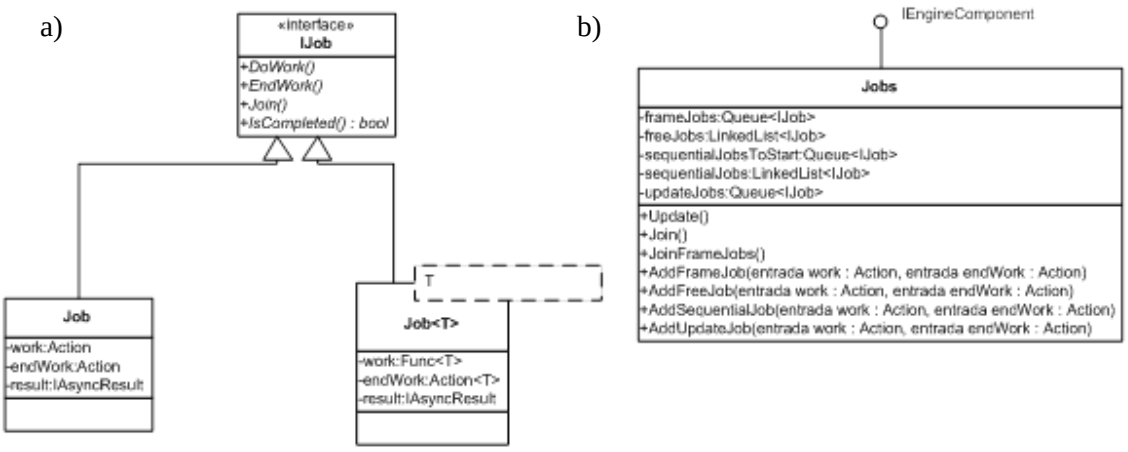


Figura 4.17. a) Modelo UML de los trabajos b) Modelo UML del administrador de trabajos

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.

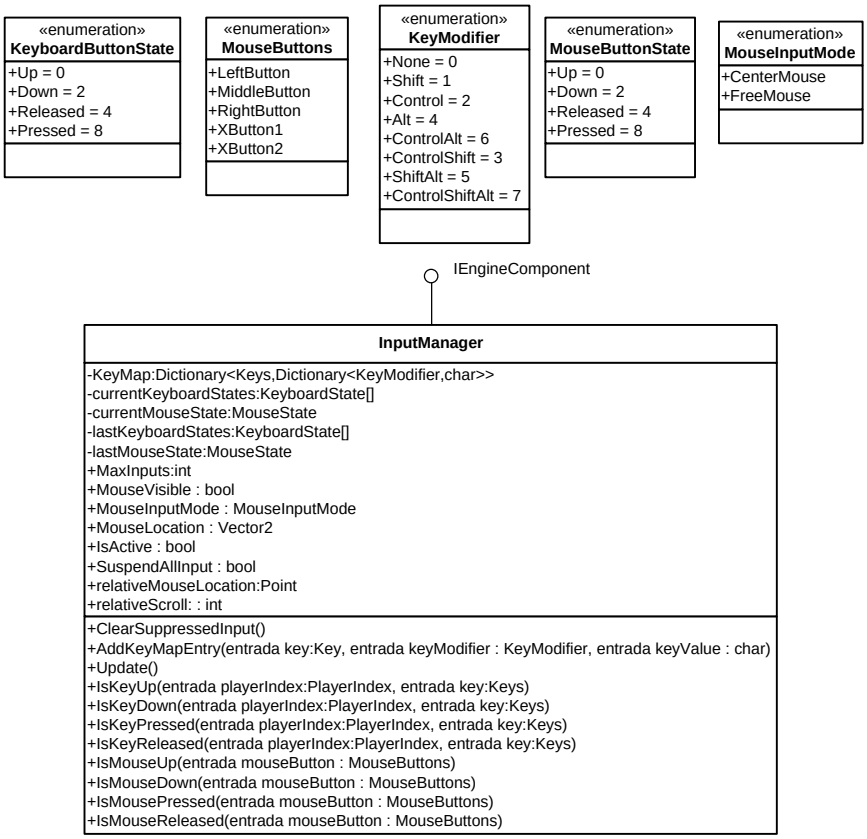


Figura 4.18. Modelos UML de las clases usadas para interactuar con el usuario

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.

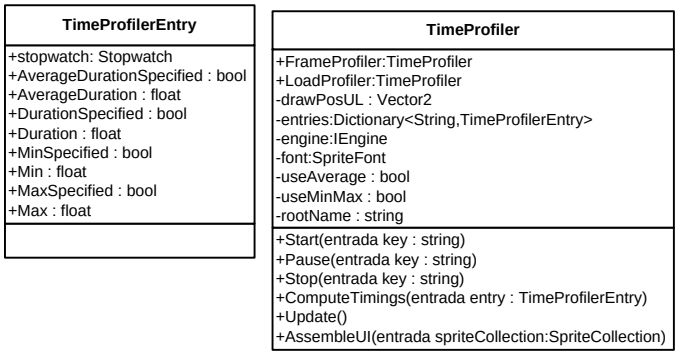


Figura 4.19. Modelo UML de TimeProfilerEntry y TimeProfiler

```
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.

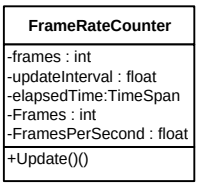


Figura 4.20. Modelo UML de FrameRateCounter

- 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.

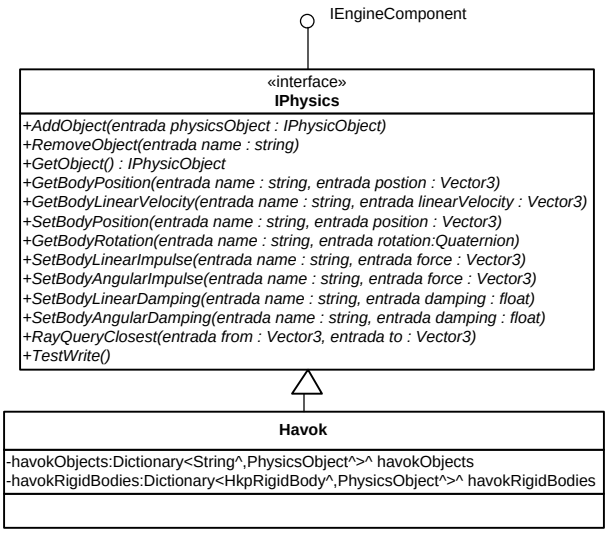


Figura 4.21. Modelo UMLde la clase Havok y la interfaz IPhysiscs

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.

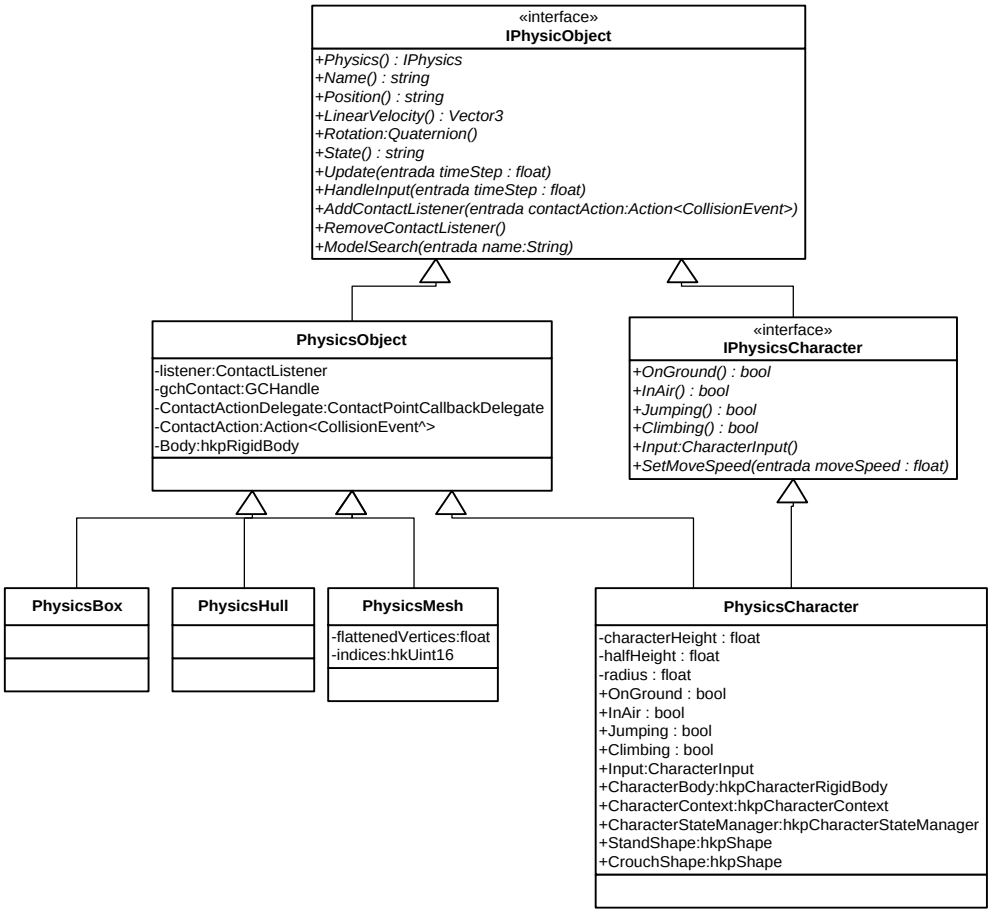


Figura 4.22. Modelo UML de los objetos físicos usados en Cosmopolis

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.

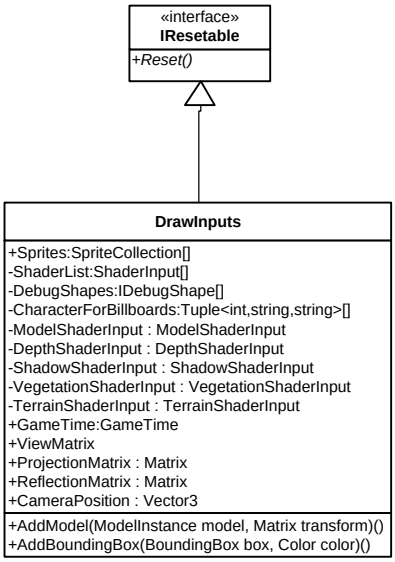


Figura 4.23. DrawInputs es el buffer que guarda los cambios generados en el UpdateThread

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.

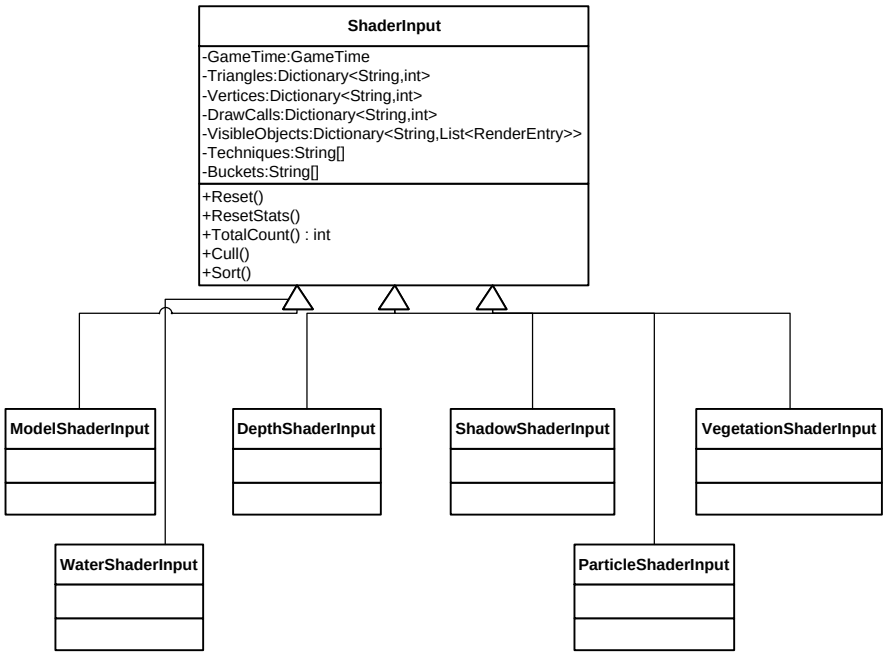


Figura 4.24. Modelo UML de las entradas para los shader

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.

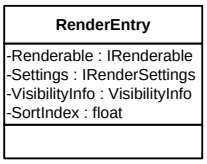


Figura 4.25 Modelo UML del Render Entry

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

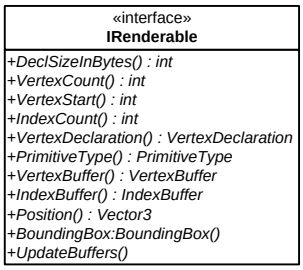


Figura 4.26. Modelo UML de la interfaz IRenderable

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.

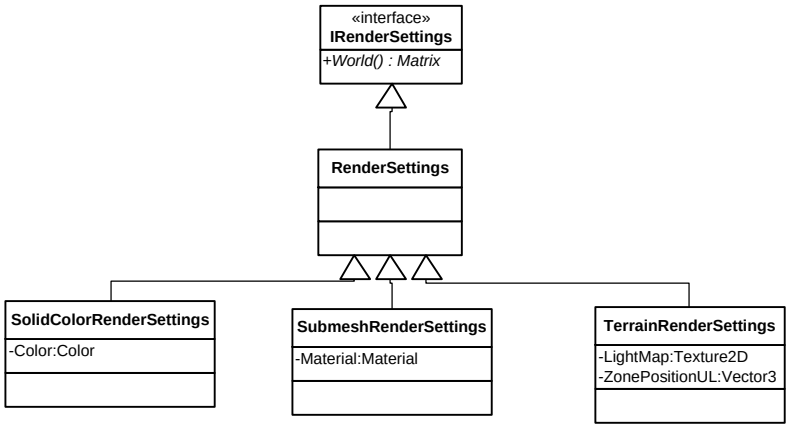


Figura 4.27. Modelo UML de la interfaz IRenderSettings y las clases que lo implementan

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

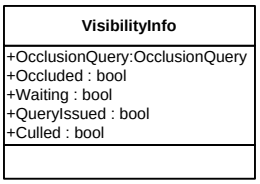


Figura 4.28. Modelo UML de VisibilityInfo

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.

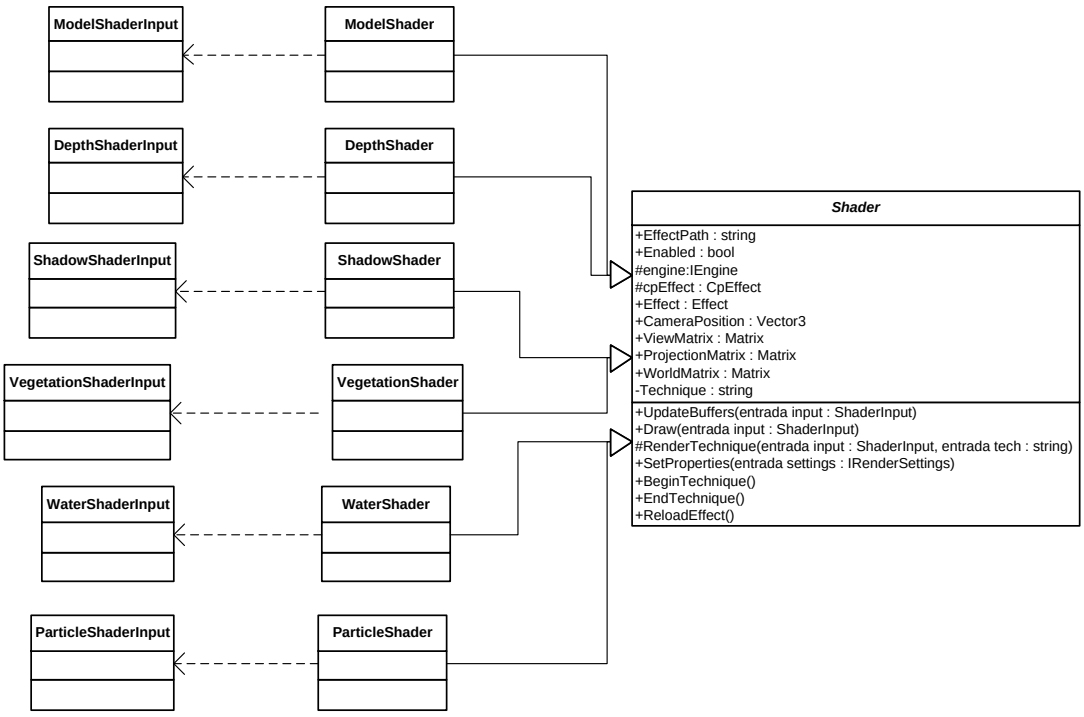


Figura 4.29. Modelo UML de los Shaders

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).

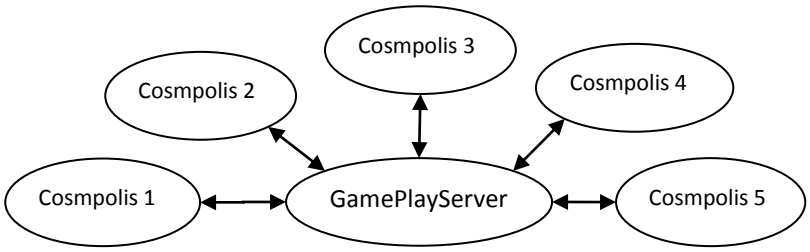


Figura 4.30. Arquitectura cliente servidor

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:

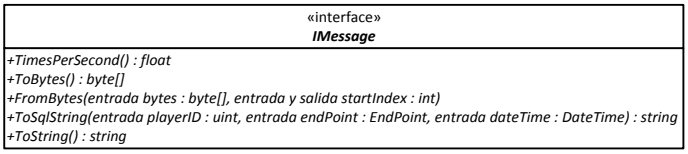


Figura 4.31. Modelo UML de la interfaz IMessage

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.

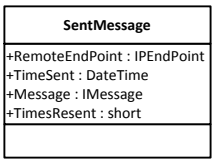


Figura 4.32. Modelo UML de SentMessage

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:

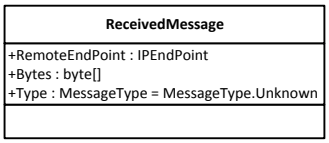


Figura 4.33. Modelo UML de ReceivedMessage

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:

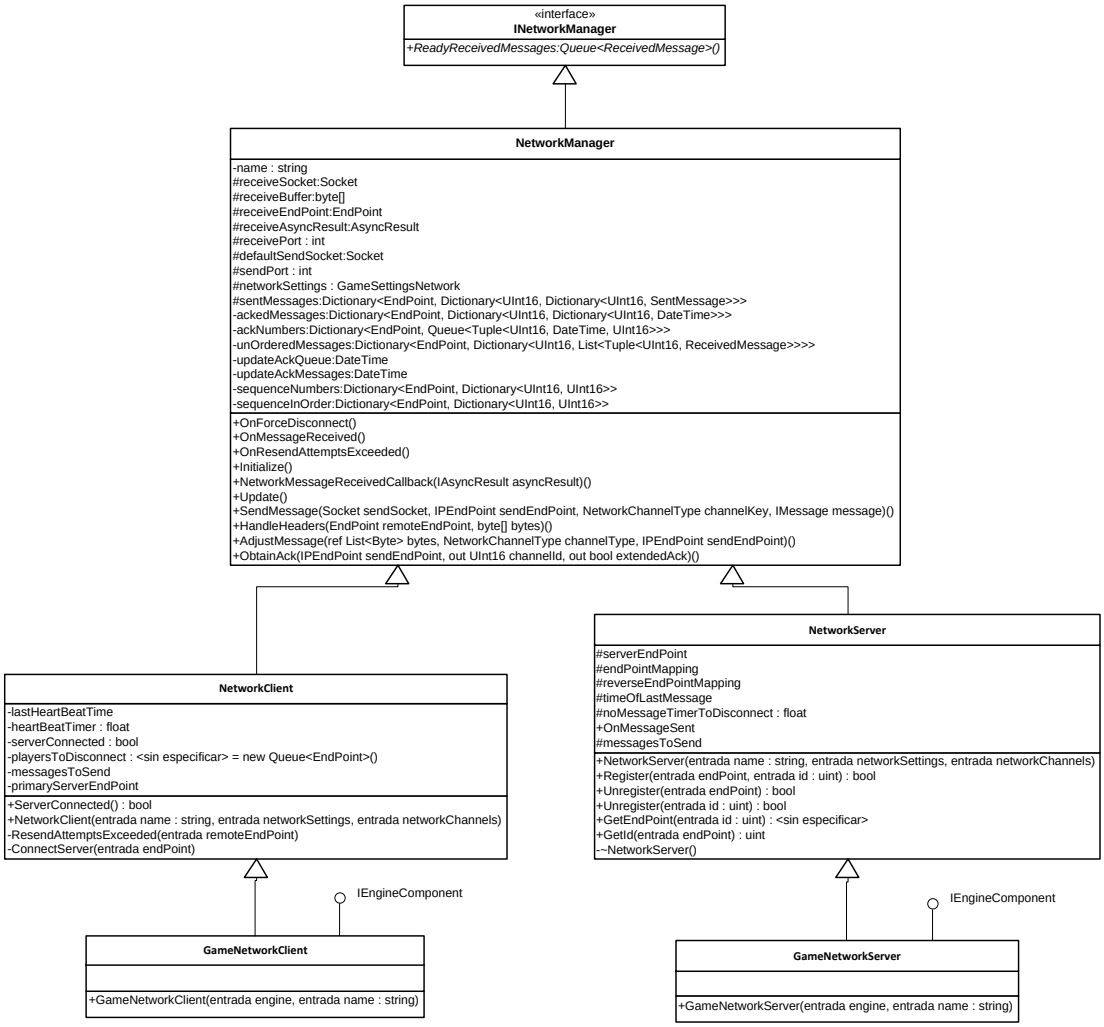
Canal de Comunicación	Ordenado	Confiable	Max. # de envíos de un mensaje p/s
Server_Channel	NO	SI	INF
Reliability_Channel	NO	SI	INF
Client_Channel	NO	SI	INF
Default_Channel	NO	SI	INF
Game_Update_LocalPlayer_Move	NO	NO	8
Game_Update_Npc_Move	NO	NO	20

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.

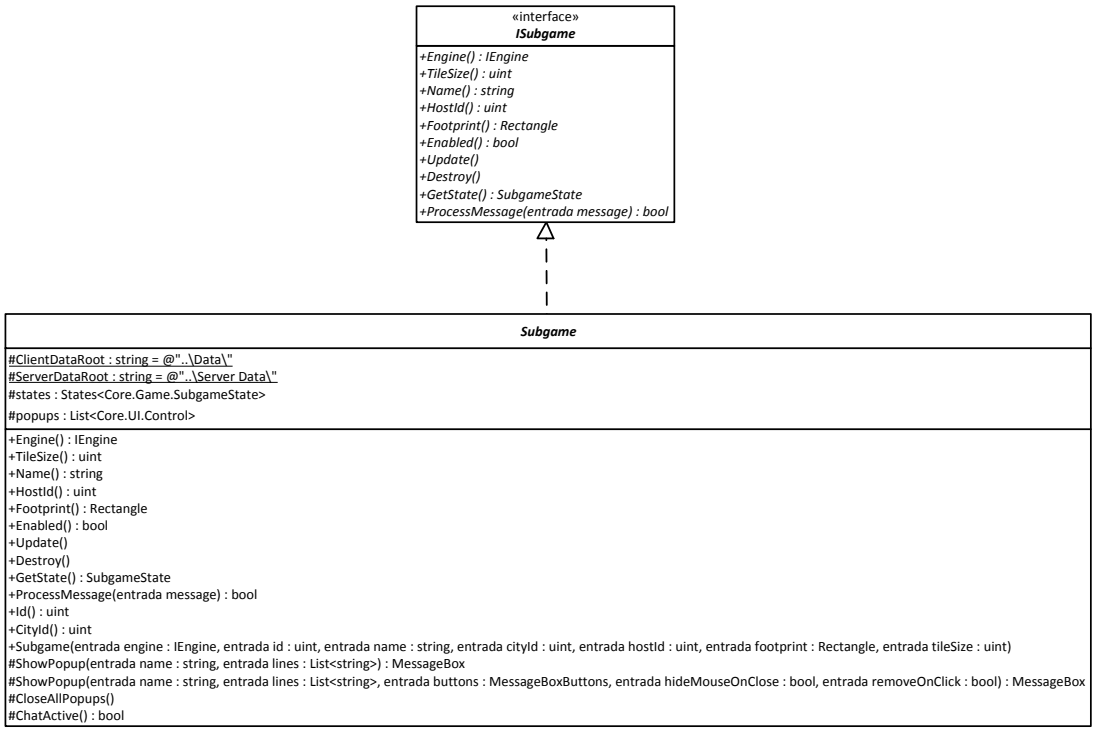


Figura 4.34. Modelos UML de Subgame y ISubgame para cliente

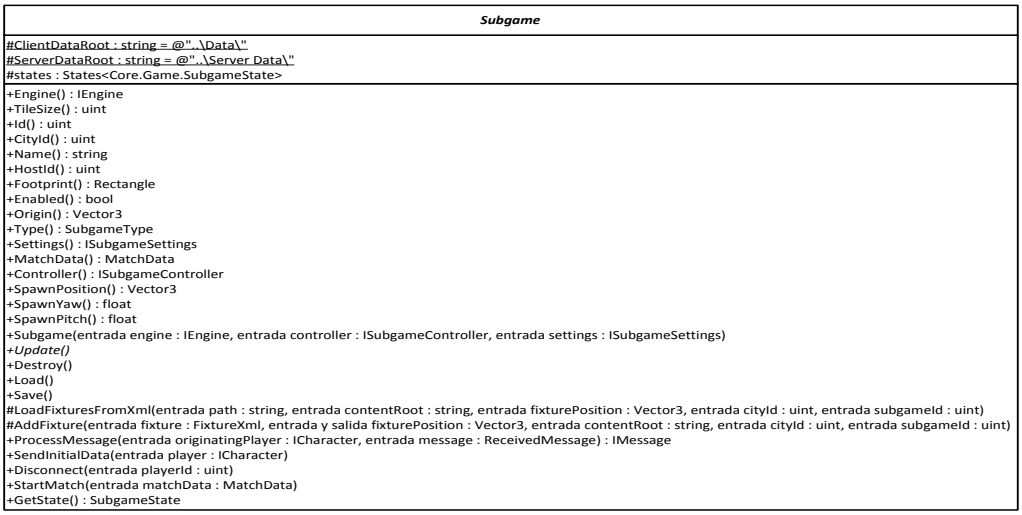


Figura 4.35. Modelos UML de Subgame para servidor

- 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.

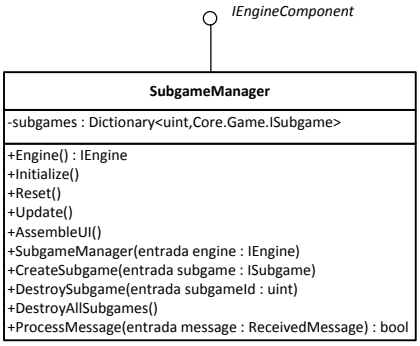


Figura 4.36. Diagrama de clase UML del SubgameManager del Cliente

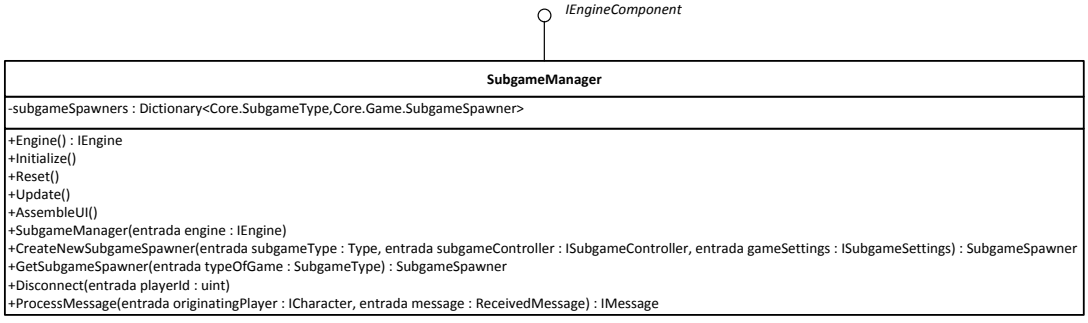


Figura 4.37. Diagrama de clase UML del SubgameManager del Servidor

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).

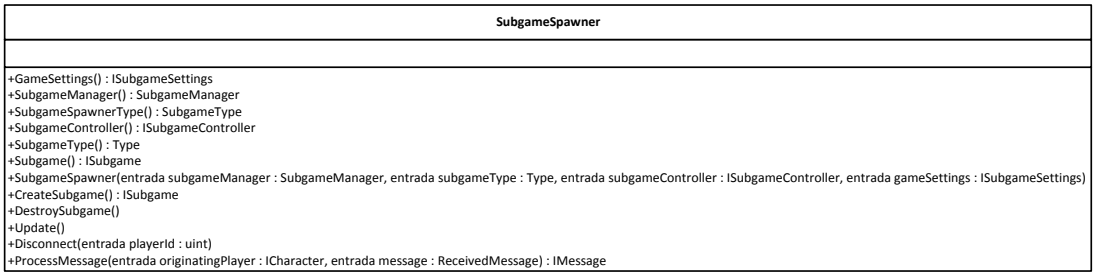


Figura 4.38. Modelo UML de SubgameSpawner

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.

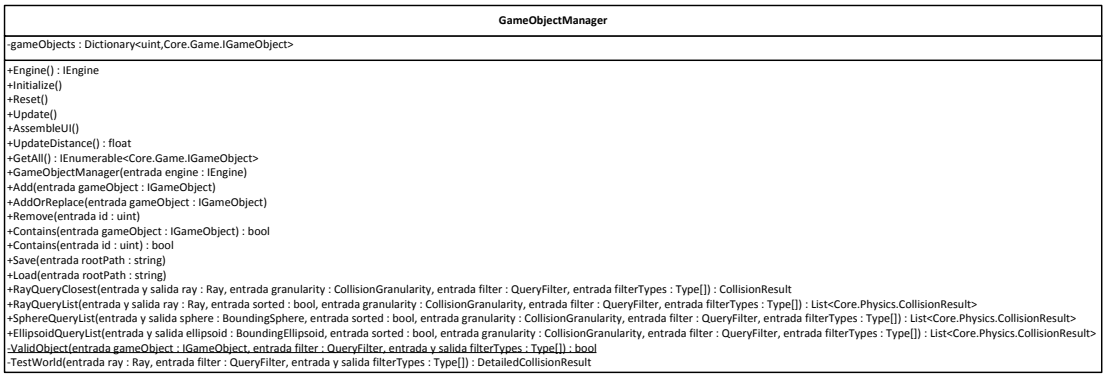


Figura 4.39. Modelo UML de GameObjectManager

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.

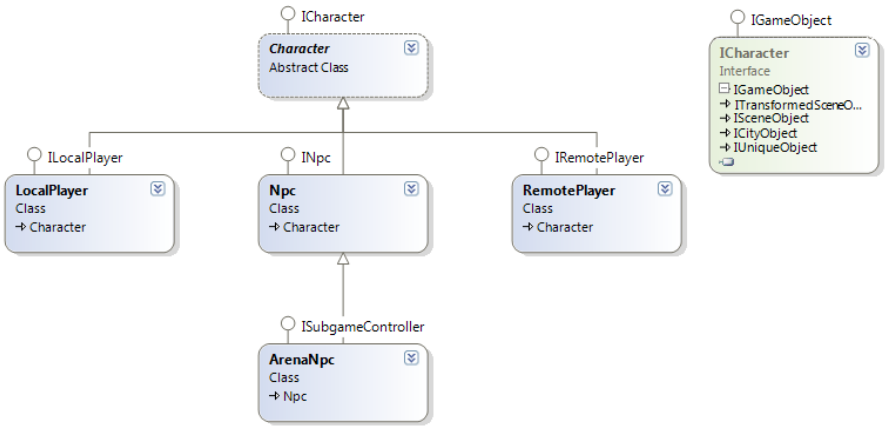


Figura 4.40. Diagrama de clases de entidades

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.

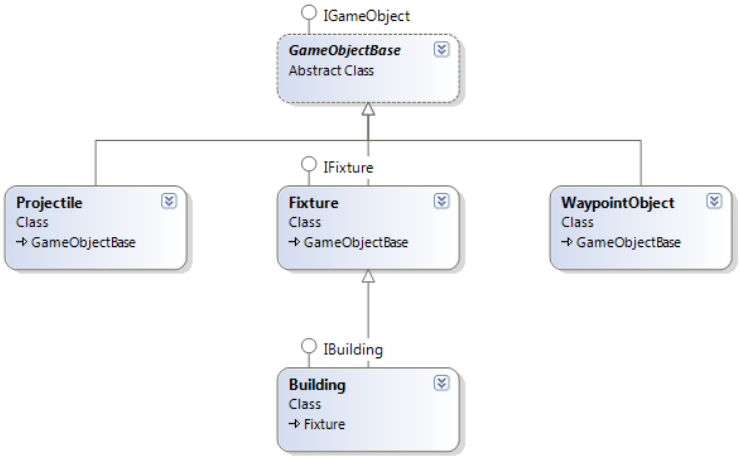


Figura 4.41. Diagrama de clases de entidades

- 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.

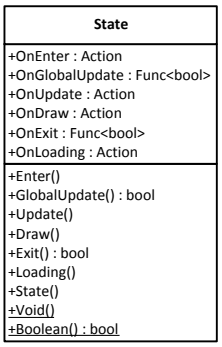


Figura 4.42. Modelo UML de clase State

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.

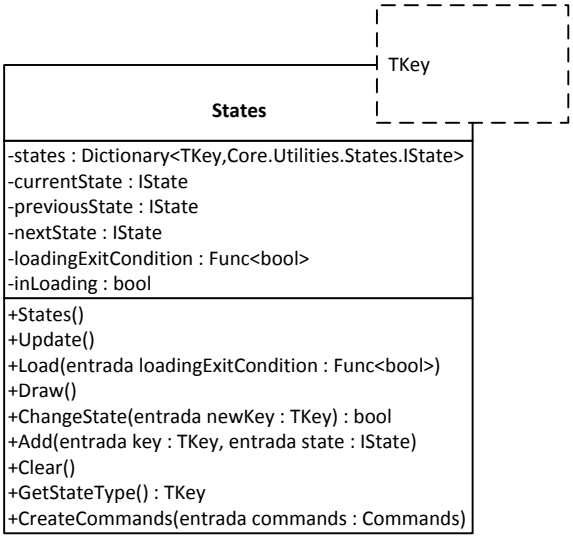


Figura 4.43. Modelo UML del administrador de estados

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

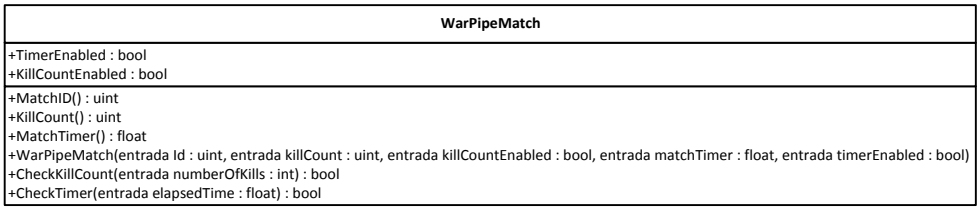


Figura 4.46. Modelo UML de WarPipeMatch

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.

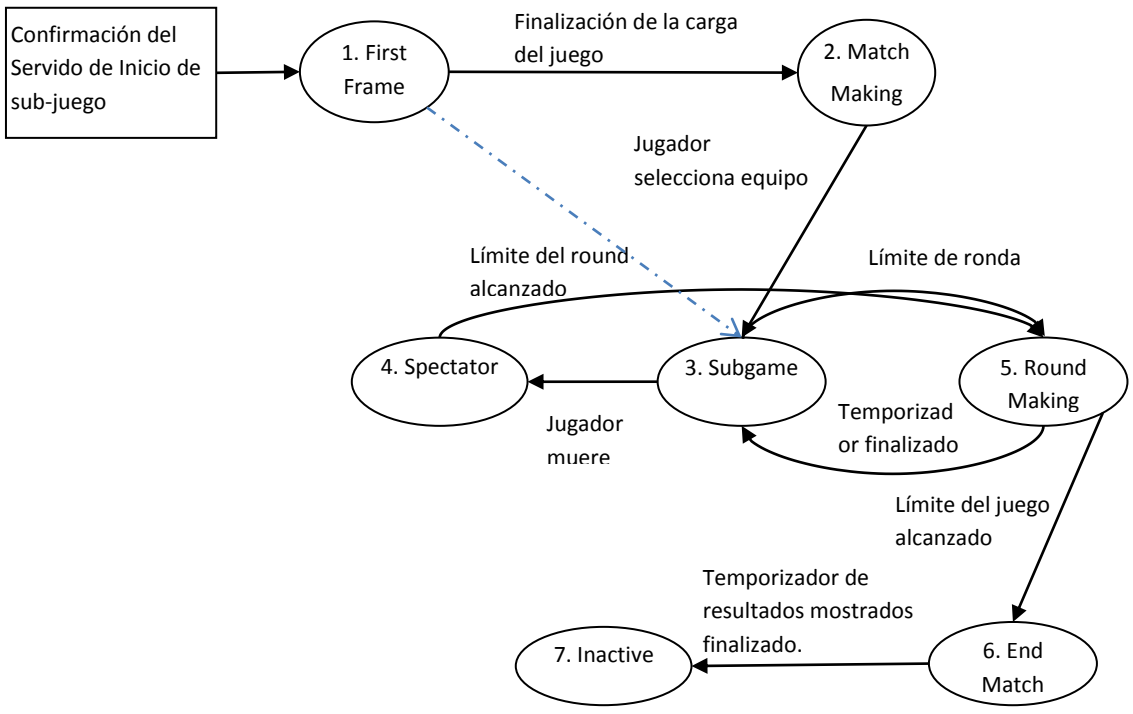


Figura 4.48. Maquina de estados para juego WarPipe en cliente

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.

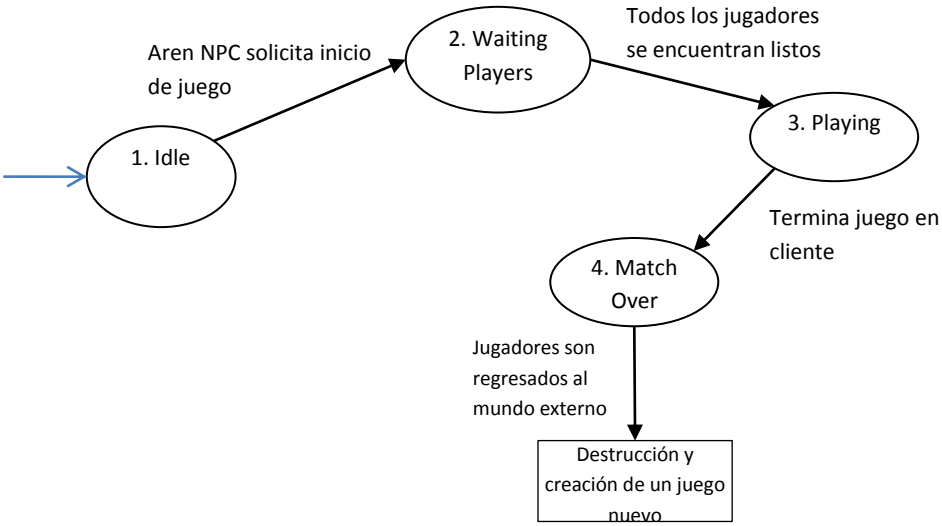


Figura 4.49. Máquina de estados para juego WarPipe en el servidor

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
}
```

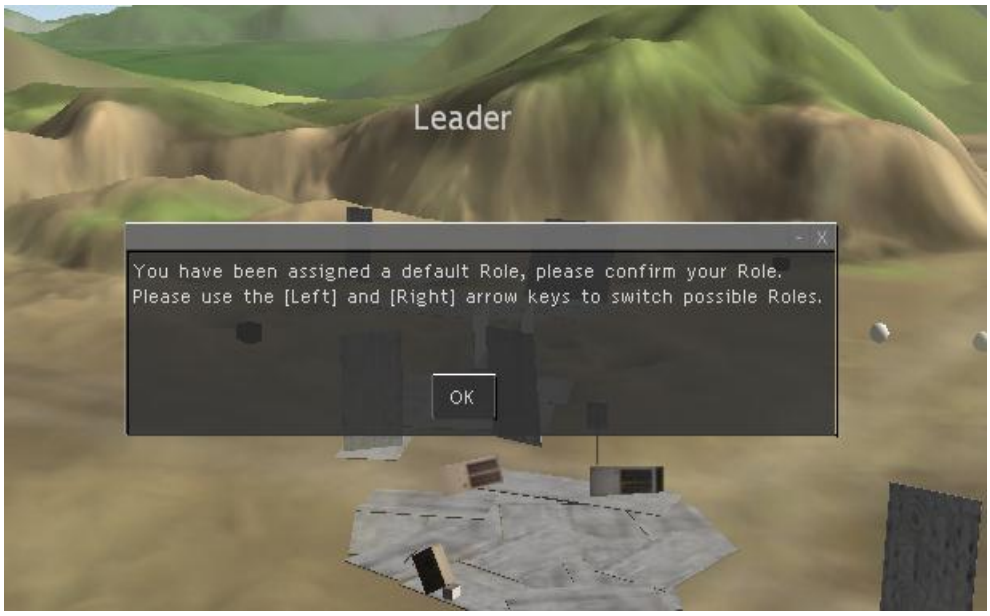


Figura 4.54. GUI para la selección de rangos

- 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.

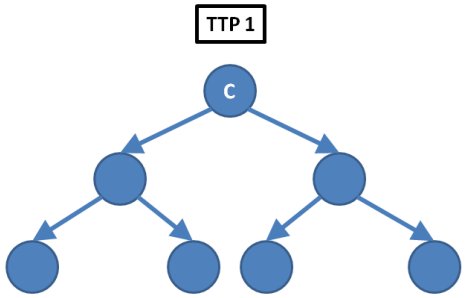


Figura 4.55. Patrón de comunicación Top Down

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

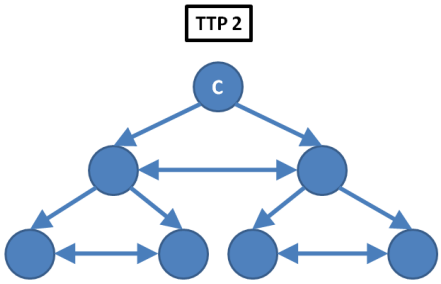


Figura 4.56. Patrón de comunicación Top Down

- All. Permite la comunicación con cualquier jugador.

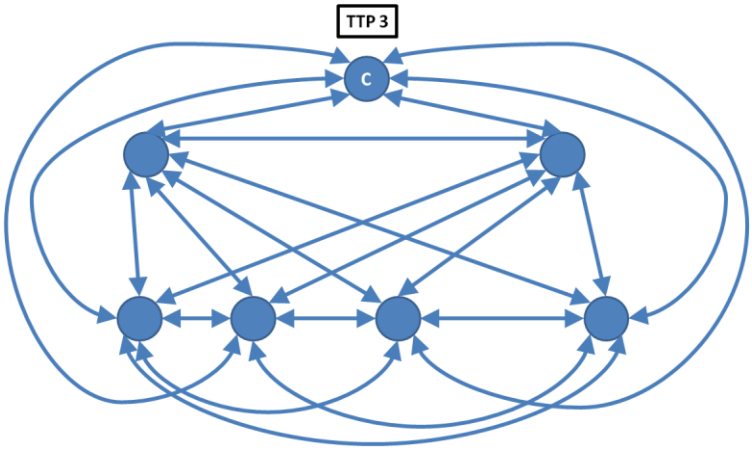


Figura 4.57. Patrón de comunicación All

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:

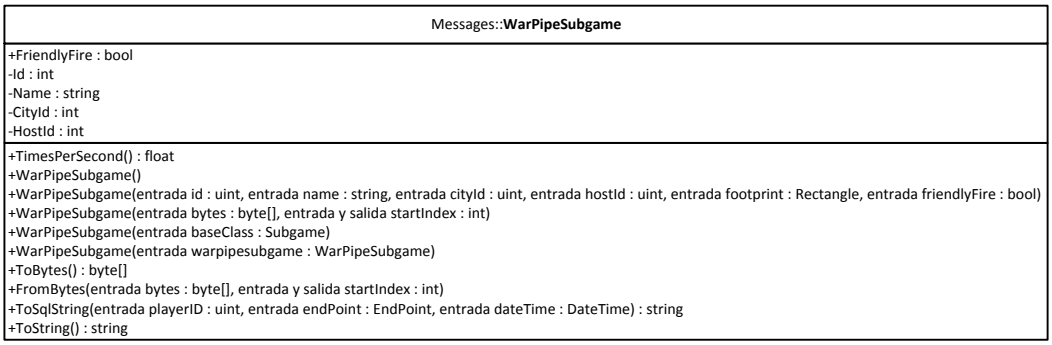
```
foreach (var endPoint in endPointMapping.Keys.ToList())
        {
                if ((DateTime.Now - timeOfLastMessage[endPoint]).TotalSeconds >
        noMessageTimerToDisconnect)
                {
                        // Disconnect End Point
                        Console.WriteLine("Havent 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.



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.

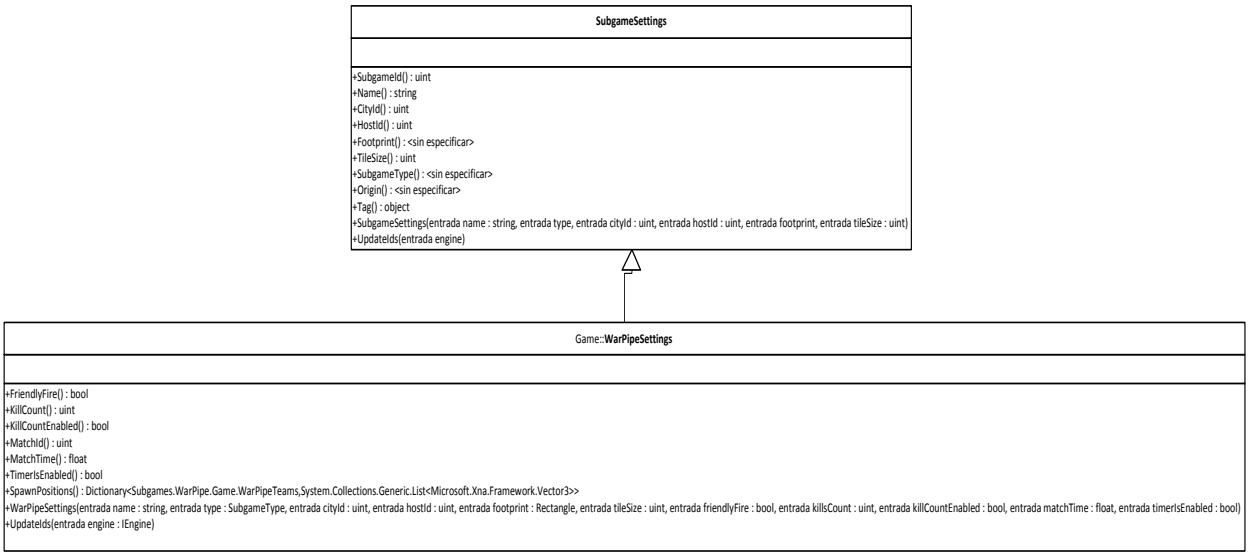


Figura 4.59. Modelo UML de SubgameSettings y WarPipeSettings

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);
```