



ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

Titulación:

INGENIERO SUPERIOR EN INFORMÁTICA

Título del proyecto:

Navegación con Wiimote en entorno 3D mediante XNA

Alumno: Francisco Javier Labiano Jáuregui

Tutor: Eduardo Alfaro Larragueta

Pamplona, 15 de Noviembre de 2011

AGRADECIMIENTOS

Quiero agradecer a todas aquellas personas que han estado a mi lado durante la elaboración del proyecto.

A Teresa por aguantar estoicamente todas mis pruebas y explicaciones a veces incompresibles hasta para mí.

A mi tutor, Edu, por ofrecerse a aguantarme con este proyecto y por no poner más problemas de los estrictamente necesarios.

Quería también agradecer al "jefe" de XNA Iñaki Ayúcar, por sus consejos (el primero y más importante que me decantara por XNA frente a otras herramientas o Frameworks) y por las dudas resueltas.

A Raúl y otros compañeros de trabajo por mostrar interés en este proyecto.

También agradezco a mi familia por su interés en que todo me fuese sin ningún problema, y haciéndome saber que en cualquier momento y por cualquier razón siempre estarían a mi lado.

Agradezco también a todos mis amigos por hacerme saber que en cualquier momento tengo la ocasión de contar con ellos para reír y olvidar esos problemas que pudiese haber tendido.

Gracias a todos.

ÍNDICE

INTRODUCCIÓN.....	8
1.1 PROPÓSITO DEL PROYECTO	9
1.2 DESCRIPCIÓN DEL PROYECTO.....	9
1.3 PERFIL DE USUARIO	10
1.4 REQUISITOS	10
1.4.1. <i>Requisitos Generales</i>	10
1.4.2.- <i>Requisitos de hardware y de sistema</i>	10
1.5.- SOLUCIÓN PROPUESTA.....	11
1.5.1.- <i>Lenguaje de Programación</i>	11
1.5.2.- <i>Librería WiimoteLib</i>	12
FRAMEWORKS DE DESARROLLO Y GAME ENGINES: HISTORIA Y POSIBILIDADES ACTUALES	13
2.1 HISTORIA DE LOS GAME ENGINES	14
XNA	21
3.1 INTRODUCCIÓN A MICROSOFT XNA	22
3.2 ESTRUCTURA BÁSICA	24
3.3 EJEMPLOS	30
3.4 XNA INTERNAMENTE	34
3.4.1 <i>Content pipeline</i>	34
3.4.2 <i>Render pipeline</i>	37
IMPLEMENTACIÓN DE LA APLICACIÓN.....	43
4.1 PRIMEROS PASOS	44
4.2 DESGRANANDO LOS COMPONENTES	44
4.2.1 <i>Primera aplicación 3D</i>	45
4.2.2 <i>Nuevos objetivos (reconocimiento y selección)</i>	61
4.3 SIMULANDO UN MUNDO REAL.....	68
4.3.1 <i>Introducción a la física en 3D</i>	68
4.3.2 <i>Motores de física</i>	69
4.3.3 <i>Alternativas utilizadas e implementación final</i>	69
4.3.4 <i>Introducción a la iluminación en 3D</i>	82
4.3.5 <i>SkyBox</i>	92
4.3.6 <i>Postprocesamiento</i>	97
4.3.6 <i>Partículas</i>	119
4.4 NAVEGACIÓN.....	124
4.5 DIAGRAMA DE CLASES.....	128
CONCLUSIONES	130
7.1 CONCLUSIONES DEL PROYECTO	131
7.2 CONCLUSIONES PERSONALES	131
7.3 LÍNEAS FUTURAS	132
7.4 CONTENIDO DEL DVD	132
BIBLIOGRAFÍA	133

8.1 LIBROS.....	134
8.2 ENLACES WEB.....	134
ANEXOS	136
DIAGRAMA DE CLASES XNA.....	137

Lista de figuras

Figura 2.1.1 - (Space Rogue) Utilizaba un engine del mismo nombre	14
Figura 2.2.2 - (Ultima underworld) Utilizaba un engine del mismo nombre	14
Figura 2.2.3 - (Doom) id Tech 1 Engine	15
Figura 2.2.4 - (Crysis) CryEngine	16
Figura 2.2.5 - (Unreal Tournament 3) Unreal Engine 3	16
Figura 2.2.6 - (Assassin's Creed) Anvil/Scimitar Engine	17
Figura 2.2.7 - Unreal Engine 4	17
Figura 2.2.8 - Unreal Engine 4	18
Figura 2.2.9 - Unreal Engine 4	18
Figura 2.2.10 - CryEngine 3	19
Figura 3.2.1 - Nuestra primera aplicación	29
Figura 3.3.1 - Plataformas 2D	30
Figura 3.3.2 - Racing Game (Menú)	31
Figura 3.3.3 - Racing Game (Selección coche)	31
Figura 3.3.4 - Racing Game (Pantalla juego)	32
Figura 3.3.5 - Robot Game desarrollado por Zepetto (Menú).....	32
Figura 3.3.6 - Robot Game desarrollado por Zepetto (Juego)	33
Figura 3.3.7 - Robot Game desarrollado por Zepetto (Juego)	33
Figura 3.4.1 - Componentes XNA	34
Figura 3.4.2 - XNA Framework Content pipeline	34
Figura 3.4.3 - Importer de Visual Studio	35
Figura 3.4.4 - Parámetros del Processor	36
Figura 3.4.5 - Pipeline de renderizado	37
Figura 3.4.6 - Procesos del componente Geometry Processing	38
Figura 3.4.7 - Componente Píxel Rendering	39
Figura 4.1.1 - Sistemas de coordenadas	45
Figura 4.2.2 - Estructura de la aplicación	45
Figura 4.2.1 - Carga de modelo 3D y navegación con cámara libre	46
Figura 4.2.3 - Valores del constructor	54
Figura 4.2.4 - Actualización del target de la cámara	55
Figura 4.2.5 - Campo de visión	56
Figura 4.2.6 - Movimientos posibles en los diferentes ejes de coordenadas	56
Figura 4.2.7 - Vector perpendicular (producto vectorial)	59
Figura 4.2.8 - Modelo tras acercarnos a él con la cámara	60
Figura 4.2.9 - RayCasting	62
Figura 4.2.10 - Esferas sin que interseccione el rayo	65
Figura 4.2.11 - Esferas tras la intersección del rayo con una de ellas	66
Figura 4.2.12 - Posición inicial de la caja	67
Figura 4.2.13 - Posición final de la caja	68
Figura 4.3.1 - Clases que implementan la física	70
Figura 4.3.2 - Modelos cargados	73
Figura 4.3.3 - Se lanza la caja	77
Figura 4.3.4 - La caja choca con el resto	77
Figura 4.3.5 - Menú desplegado	78
Figura 4.3.6 - Se carga un tronco	78
Figura 4.3.7 - Se carga una caja	79

<i>Figura 4.3.8 - Esfera iluminada solo por luz ambiental</i>	83
<i>Figura 4.3.9 - Luz difusa</i>	83
<i>Figura 4.3.10 - Esfera iluminada solo por luz difusa</i>	84
<i>Figura 4.3.11 - Esfera con iluminación difusa y ambiental</i>	84
<i>Figura 4.3.12 - Esfera iluminada por luz difusa, ambiental y especular</i>	85
<i>Figura 4.3.13 - Editor SunBurn</i>	85
<i>Figura 4.3.14 - Iluminación SunBurn</i>	90
<i>Figura 4.3.15 - Iluminación SunBurn</i>	90
<i>Figura 4.3.16 - Iluminación SunBurn</i>	91
<i>Figura 4.3.17 - Sin iluminación SunBurn</i>	91
<i>Figura 4.3.18 - Skybox</i>	93
<i>Figura 4.3.19 - Skybox sin objetos en escena</i>	95
<i>Figura 4.3.20 - Skybox con objetos en escena</i>	96
<i>Figura 4.3.21 - Skybox con objetos en escena</i>	96
<i>Figura 4.3.22 - Lens Flare</i>	97
<i>Figura 4.3.23 - Lens flare</i>	98
<i>Figura 4.3.24 - Efecto Bloom</i>	99
<i>Figura 4.3.25 - Profundidad de campo</i>	100
<i>Figura 4.3.26 - Desenfoque de movimiento</i>	100
<i>Figura 4.3.27 - Desenfoque de movimiento</i>	101
<i>Figura 4.3.28 - Efecto desaturado</i>	102
<i>Figura 4.3.29 - Efecto saturado</i>	102
<i>Figura 4.3.30 - Efecto borroso</i>	103
<i>Figura 4.3.31 - AO</i>	103
<i>Figura 4.3.32 - Sin efecto Bloom</i>	107
<i>Figura 4.3.33 - Efecto Bloom sutil</i>	107
<i>Figura 4.3.34 - Efecto Bloom más pronunciado</i>	108
<i>Figura 4.3.35 - Triangulo rasterizado</i>	109
<i>Figura 4.3.36 - Esquema HLSL</i>	109
<i>Figura 4.3.37 - Content</i>	112
<i>Figura 4.3.38 - Shader de luz ambiental</i>	114
<i>Figura 4.3.39 - Iluminación difusa</i>	117
<i>Figura 4.3.40 - Partículas</i>	123
<i>Figura 4.3.41 - Partículas</i>	124
<i>Figura 4.5.1 - Diagrama de clases</i>	129
<i>Figura Anexo 1 - Storage</i>	137
<i>Figura Anexo 2 - Audio</i>	138
<i>Figura Anexo 3 - Content</i>	139
<i>Figura Anexo 4 - Content.Pipeline</i>	140
<i>Figura Anexo 5 - Content.Pipeline.Audio</i>	141
<i>Figura Anexo 6 - Content.Pipeline.Graphics</i>	142
<i>Figura Anexo 7 - Content.Pipeline.Graphics(Textures)</i>	143
<i>Figura Anexo 8 - Content.Pipeline.Processors</i>	144
<i>Figura Anexo 9 - Content.Pipeline.Serialization.Compiler</i>	145
<i>Figura Anexo 10 - Content.Pipeline.Serialization.Intermediate</i>	146
<i>Figura Anexo 11 - Content.Pipeline.Tasks</i>	147
<i>Figura Anexo 12 - Content.GameServices</i>	148
<i>Figura Anexo 13 - Framework</i>	149
<i>Figura Anexo 14 - Framework.Graphics</i>	150
<i>Figura Anexo 15 - Framework.Graphics (Effects)</i>	151

<i>Figura Anexo 16 - Framework.Graphics (Misc)</i>	152
<i>Figura Anexo 17 - Framework.Graphics (Models&Sprites)</i>	153
<i>Figura Anexo 18 - Framework.Graphics (RenderState)</i>	154
<i>Figura Anexo 19 - Framework.Graphics (RenderTarget, Textures)</i>	155
<i>Figura Anexo 20 - Framework.Graphics.PackedVector</i>	156
<i>Figura Anexo 21 - Framework.Input</i>	157
<i>Figura Anexo 22 - Framework.Media</i>	158
<i>Figura Anexo 23 - Framework.Net</i>	159

Capítulo 1

INTRODUCCIÓN

En este capítulo introductorio se va a explicar el propósito de este proyecto fin de carrera. Consiste en la realización de un estudio y posterior implementación de una aplicación 3D manejada en tiempo real mediante el dispositivo Wiimote de Nintendo y a través del Framework de programación 3D XNA de Microsoft. Se comentará el proceso seguido para realizarlo.

1.1 Propósito del proyecto

El objeto del proyecto es el desarrollo de una aplicación que permita la navegación en un entorno virtual 3D mediante el control remoto de Wii (Wiimote+Nunchuck). Para ello se ha utilizado el Framework XNA 3.1 de Microsoft.

Actualmente las aplicaciones 3D constituyen una de las apuestas con más futuro y aplicación práctica en el mundo de la informática. Además, la incorporación al mercado de la videoconsola de Nintendo Wii proporcionó un periférico (Wiimote) barato y potente para el desarrollo de aplicaciones basadas en acelerómetros.

Además, Microsoft desarrolla el Framework XNA 3.1, que permite la creación de aplicaciones y juegos, tanto 2D como 3D, de una forma sencilla y asequible a la gran mayoría de desarrolladores.

Por tanto, la aplicación desarrollada en el proyecto ha consistido en una aplicación XNA que permita navegar, mover y añadir elementos a un entorno tridimensional creado previamente en un software 3D (3D Studio Max). A su vez se pretende investigar en los diferentes tipos de componentes que un juego o aplicación de estas características puede tener (partículas, física, iluminación, etc.)

1.2 Descripción del proyecto

En una primera fase se han estudiado los diferentes tipos de herramientas disponibles para la creación de aplicaciones 3D.

Una vez decantados por un Framework, en este caso XNA en su versión 3.1, se han estudiado las necesidades de la aplicación (motor de iluminación, motor físico, partículas, etc.). Por último, y tras varias implementaciones, se han aplicado estos conocimientos en una aplicación final.

Las etapas de desarrollo del proyecto han sido seis:

- 1.- Estudio. Esta tarea se ha realizado a lo largo de todo el proyecto.
- 2.- Fase de Análisis. Ha consistido en analizar detalladamente los gustos de navegación de los usuarios generales para mejorar en todo lo posible la experiencia de navegación y que ésta sea lo más cómoda y eficaz posible.
- 3.- Fase de Diseño. En esta fase se ha realizado el diseño de la aplicación.
- 4.- Fase de Implementación. En esta fase primero se han creado una serie de versiones beta de la aplicación con funcionalidad limitada, para a continuación desarrollar la aplicación final.

5.- Integración y pruebas. Se trata de la puesta a punto de la herramienta.

6.- Documentación. Se trata de documentar todo el proceso. Esta tarea concluye con la realización de este documento.

1.3 Perfil de usuario

Este proyecto se ha realizado con la intención de que pueda servir como ayuda a aquellos usuarios que quieran iniciarse en el mundo de la programación 3D.

También puede ir destinado a un amplio grupo de personas que tengan cierta experiencia en el uso de este Framework y quieran mejorar o añadir conocimientos en ciertos aspectos.

Por último se destaca el uso de esta aplicación como puro entretenimiento.

1.4 Requisitos

En este apartado se mostrará los requisitos mediante los cuales se ha hecho el análisis y diseño de la aplicación software. Se explicarán los requisitos generales y, a continuación, los requisitos por categorías.

1.4.1. Requisitos Generales

Cualquier usuario que emplee la aplicación deberá poder usarla con facilidad, para ello la aplicación deberá ser intuitiva en su uso y disponer de un manual en el que se muestre las distintas funcionalidades del mando en la aplicación.

1.4.2.- Requisitos de hardware y de sistema

Los requisitos de sistema operativo son los siguientes:

- Windows XP
 - Home Edition
 - Professional
 - Media Center Edition
 - Tablet PC Edition

- Windows Vista
 - Home Basic
 - Home Premium
 - Business
 - Enterprise
 - Ultimate

- Windows 7
 - Home Basic
 - Home Premium
 - Professional
 - Enterprise
 - Ultimate

Los requisitos hardware son:

- Tarjeta grafica que soporte, como mínimo, el modelo de shaders 1.1 y DirectX 9.0c. Se recomienda el uso de una tarjeta que soporte el modelo 2.0.

- 256 MB de memoria RAM, aunque se recomienda tener entre 512 MB y 1 GB como mínimo.

Otro de los requisitos necesario y obligatorio será que disponga de Bluetooth. Esta aplicación funciona a través del mando Wiimote que se comunica con el equipo a través de este protocolo. A día de hoy, existen en el mercado muchos adaptadores Bluetooth a través de USB a un precio más que módico. Además muchos de los equipo lo llevan integrado.

1.5.- Solución propuesta

1.5.1.- Lenguaje de Programación

Para el desarrollo de este proyecto se empleará el lenguaje Visual C# .NET, y más concretamente la última versión de este lenguaje existente en el mercado al inicio del proyecto, con Microsoft .NET Framework 3.5. Este lenguaje es utilizado dentro del entorno de desarrollo Visual Studio 2008. La versión del Framework XNA utilizada ha sido la 3.1. Posteriormente se lanzó la versión 4.0 que incluye algunas mejoras.

La elección del lenguaje de programación C# está fundamentada en que XNA se basa en él. Además, dada mi experiencia con dicho lenguaje, me resultaba mucho más cómodo que otras alternativas.

1.5.2.- Librería WiimoteLib

Es una librería de dominio público desarrollada por Brian Peek en Visual C# .NET que permite obtener todos los datos enviados por el Wiimote de una forma clara y precisa.

Esta librería se encuentra actualmente en la versión estable 1.7.0.0 y existe una versión beta con soporte para el WiimotePlus.

Estos son algunos ejemplos de código de cómo se emplea esta librería:

- La instanciación y conexión del aparato se realizará de la siguiente manera:

```
Wiimote wmc;  
wmc = new Wiimote();  
wmc.Connect();
```

- Para conocer el estado del mando deberemos instanciar dicho estado y ver los valores de los diferentes botones, acelerómetros, etc:

```
WiimoteState wiiMoteState;  
wiiMoteState = new WiimoteState();  
wiiMoteState.ButtonState.A  
wiiMoteState.ButtonState.B  
wiiMoteState.ButtonState.Home  
...  
wiiMoteState.AccelState.RawValues.X
```

Además del mando, podremos acceder a los complementos que tenga unidos a él. En este caso el Nunchuck:

```
wiiMoteState.NunchukState.RawJoystick.Y
```

Estos son solo unos pequeños ejemplos del gran número de métodos y variables que contiene la librería. Como ya hemos comentado la librería permite acceder a todos los dispositivos de la Wii (Balance, Nunchuck, Wiimote, etc.)

Más adelante veremos cual ha sido el uso del Wiimote en la aplicación.

Capítulo 2

Frameworks de desarrollo y game Engines: Historia y posibilidades actuales

En este capítulo vamos a ver la historia de los game engines de desarrollo de juegos y aplicaciones 3D así como las diferentes alternativas que existen en el mercado actual.

2.1 Historia de los game engines

Antes de hablar de los diferentes frameworks de desarrollo y game engines existentes en el mercado, vamos a ver la evolución de los videojuegos a lo largo de la historia.

En primera instancia hablaremos de los game engines. Y es que antes de que aparecieran, los juegos se creaban exclusivamente para una máquina hardware en concreto. Es decir, si creábamos un juego para la Comodore, este valía exclusivamente para esa máquina y el código del juego era difícilmente utilizable en otros juegos.



Figura 2.1.1 - (Space Rogue) Utilizaba un engine del mismo nombre



Figura 2.2.2 - (Ultima underworld) Utilizaba un engine del mismo nombre

La primera generación de *engines* gráficos o renderizadores (precursores de los *engines* actuales) fueron *BRender* de *Argonaut Software*, *Renderware* de *Criterion Software* y *RenderMorphics* de *Reality Lab*.

El término *game engine* nació a mediados de los 90 en pleno auge de los juegos 3D (especialmente los shooters en primera persona (FPS)). Con el éxito de juegos como *Doom* o *Quake*, muchos desarrolladores adquirieron la licencia de porciones de código de dichos juegos para diseñar sus propios gráficos, personajes, armas, niveles, etc.



Figura 2.2.3 - (*Doom*) id *Tech 1 Engine*

Con el tiempo y el desarrollo de dichos motores, se vio incluso que el desarrollo y posterior venta de éstos era más rentable que la creación de juegos en sí. Y es que el valor de la licencia para un juego comercial puede rondar entre los 10.000 dólares americanos y los millones de dólares.

El juego *Lineage II* es un ejemplo de este sistema de creación de juegos ya que está basado en el motor *Unreal*. Y es que cada vez más empresas se deciden por comprar el motor en vez de crearlo ya que, a pesar del coste inicial, acorta los plazos de desarrollo en un mundo cada vez más competitivo. Además los motores cada vez son más potentes e incorporan más funcionalidades (detección de colisiones, física, iluminación, partículas, etc.).

En las siguientes figuras vemos algunos ejemplos de los *engines* más modernos y comprobamos el salto de calidad que se ha producido en pocos años.



Figura 2.2.4 - (Crysis) CryEngine



Figura 2.2.5 - (Unreal Tournament 3) Unreal Engine 3



Figura 2.2.6 - (Assassin's Creed) Anvil/Scimitar Engine

El futuro de los engines sin duda promete grandes avances sobre todo a raíz de las últimas imágenes de dos de los motores más famosos hoy en día: CryEngine3 y Unreal Engine 4. El primero estará disponible en breve y se podrá ver en juegos como Crysis 2. En cuanto al Unreal Engine 4, uno de sus responsables ha comentado recientemente que verá la luz entre 2012 y 2018.



Figura 2.2.7 - Unreal Engine 4



Figura 2.2.8 - Unreal Engine 4



Figura 2.2.9 - Unreal Engine 4



Figura 2.2.10 - CryEngine 3

Sin duda, las imágenes son espectaculares. Hay que tener en cuenta que estamos hablando siempre de tiempo real, lo que sin duda es increíble viendo las imágenes. Evidentemente aún estamos lejos de ver estos niveles en nuestros ordenadores y consolas, principalmente porque esa calidad gráfica implica una elevadísima capacidad de cálculo, lejos aún de los ordenadores y consolas actuales, amén de una capacidad de almacenaje muy grande.

El desarrollo de estas tecnologías lo que han permitido es que cada vez más desarrolladores aficionados se decidan a crear su propio juego, aumentando la variedad existente en el mercado hasta unos límites difícilmente imaginados hace solo unos pocos años. Así hemos visto como proliferan cada vez más los famosos markets que permiten a los usuarios de ordenador, smartphones, etc. descargarse juegos (de pago o gratuitos) desarrollados en muchas ocasiones por una o dos personas.

La lista de engines[1] es interminable pero basta algunos ejemplos para ver las posibilidades que tenemos:

- **idTech1:** Motor con que se crearon juegos como Doom II
- **idTech4:** Motor utilizado para crear los juegos Quake 4 y Doom III
- **Blender:** Motor proporcionado por el programa de modelado 3D

[1] http://en.wikipedia.org/wiki/List_of_game_engines

Claro que estamos hablando de game engines y aunque, como hemos dicho, los game engines actuales permiten crear un juego entero (con física, iluminación, etc.) siempre están englobados en un determinado engine. Es decir, no nos permite salirnos de sus características.

Si quisiéramos crear por ejemplo una iluminación o efecto que el engine no nos permite no podríamos. Por eso tendríamos que diferenciar entre game engine y framework. Un framework de desarrollo de juegos es un producto basado en una tecnología que nos permite crear juegos o aplicaciones basándonos en dicha tecnología. Aparentemente la definición se parece pero no es lo mismo. Con un framework de desarrollo basado en DirectX (como puede ser XNA) podríamos crear nuestro propio game engine con el que podríamos crear juegos. Es decir, el framework accede a un nivel más bajo de programación, en este caso directamente a DirectX, OpenGL, etc. por lo que nos permite tantas posibilidades como nuestra tarjeta gráfica acepte.

Algunos ejemplos de frameworks basados en DirectX son:

- XNA
- SlimDX
- OpenTK

Otros ejemplos de framework:

- GameJS (Framework JavaScript para crear juegos 2D)
- lotu (Framework para crear juegos en Ruby)
- Pygames (Framework para crear juegos en Python)
- PulpCore (Framework para crear juegos en Java)

Capítulo 3

XNA

En este capítulo vamos introducir los conceptos básicos sobre el Framework de programación XNA y sobre las diferentes alternativas que existen en el mercado.

3.1 Introducción a Microsoft XNA

Microsoft XNA es un conjunto de herramientas con un entorno de ejecución que facilita el desarrollo de aplicaciones y juegos tanto en 2D como en 3D para ordenador, Xbox 360 y Zune. Se lanzó con la intención de evitar el llamado "código repetitivo" tan frecuente en estos ámbitos.

Incluye un conjunto de bibliotecas de clases, específicos para el desarrollo de juegos, para promover la reutilización de código máximo en las diferentes plataformas. Es decir, que si tenemos un código para ordenador, es muy fácilmente extensible a Xbox 360.

Es importante recalcar que XNA **no** es un motor de videojuegos y **tampoco** es un creador de juegos estilo *drag&drop*. Decimos que no es un motor de videojuegos porque no incluye motor físico, motor de iluminación, detección de colisiones, etc. Y es que si queremos estas funcionalidades deberemos implementarlas o utilizar librerías ya creadas. Por otra parte decimos que no es un creador de juegos estilo *drag&drop* porque no tendremos interfaz gráfico para crear nuestra aplicación. Lo que sí existen son engines basados en XNA. Algunos ejemplos son:

- SunBurn Game Engine
- TorqueX
- Visual3D
- Blade3D
- Axiom3D
- Flat Red Ball
- Thrust

Dos de las herramientas más importantes que ofrece XNA son:

- **XNA Framework**

Basado en la implementación nativa de .NET Compact Framework 2.0 para el desarrollo de la Xbox 360 y .NET Framework 2.0 en Windows, incluye un conjunto de bibliotecas de clases, específicos para el desarrollo de juegos, para promover la reutilización de código máximo a través de plataformas de destino. Está disponible para Windows XP, Windows Vista y Xbox 360. Las aplicaciones podrán ejecutarse en cualquier plataforma que admite el XNA Framework con mínima o ninguna modificación.

Los juegos pueden escribirse en cualquier lenguaje compatible con .NET, pero oficialmente se admite sólo C#.

Además integra con una serie de herramientas, tales como la plataforma XACT, para ayudar en la creación de contenido.

- XNA Build (Content Pipeline):

XNA Build proporciona una serie de herramientas para la gestión de contenidos. Está pensado para la definición, mantenimiento, depuración y optimización del pipeline de contenidos.

- **XNA Game Studio**

XNA Game Studio es un entorno de desarrollo integrado (IDE) para el desarrollo de juegos. Existen cinco versiones actualmente:

- **XNA Game Studio Professional:**

Versión planificada del IDE XNA dirigida a profesionales del desarrollo de juegos. Proporciona herramientas para la administración del proyecto, tales como la gestión de activos, seguimiento de defectos, automatización de proyectos y listas de artículos de trabajo, son algo automatizadas por XNA Studio. Ya no está bajo desarrollo activo.

- **XNA Game Studio Express:**

Versión destinada a los estudiantes, aficionados y desarrolladores independientes. Está disponible como descarga gratuita. Proporciona lo básico para el rápido desarrollo de juegos típicos, tales como los basados en plataformas, estrategia en tiempo real y shooters en primera persona.

- **XNA Game Studio 2.0:**

Versión mejorada lanzada el 13 de Diciembre de 2007. Incluía una API de red usando Xbox Live en Windows y Xbox 360 y una mejora en el manejo de dispositivos.

- **XNA Game Studio 3.0:**

Versión lanzada en Septiembre de 2008. Se incluyó compatibilidad con C# 3.0, LINQ y la mayoría de las versiones de Visual Studio 2008. Además se incluyó soporte para el dispositivo Zune.

- **XNA Game Studio 3.1:**

Versión anunciada el 24 de Marzo de 2009. Incluía una API de audio revisada y una nueva API de video.

- **XNA Game Studio 4.0:**

Versión que permite desarrollar juegos y aplicaciones para Windows Phone 7.

3.2 Estructura básica

La estructura de código que sigue una aplicación XNA es siempre la misma. La idea es muy simple. En primer lugar se inicializan las variables, a continuación se carga el contenido (modelos, sprites, texturas, etc.) que vayamos a necesitar. Una vez cargado el contenido iniciaremos un bucle infinito que en primer lugar comprueba que objetos pueden haber cambiado respecto al frame anterior y a continuación se dibuja la escena en su estado actual. Finalmente si salimos del bucle se descarga o destruye el contenido antes de salir del programa.

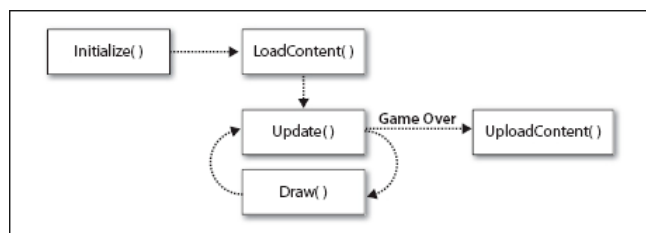


Figura - Estructura básica de una aplicación XNA

Este sería un ejemplo básico:

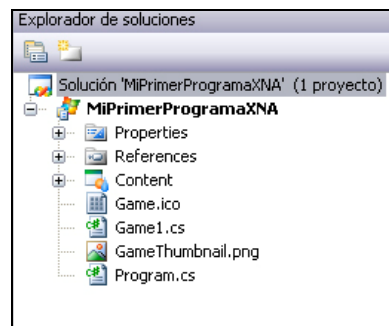


Figura - Estructura de la solución

Donde la entrada del programa sería la clase `Programa` que tiene la siguiente estructura:

```
using System;

namespace MiPrimerProgramaXNA
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main(string[] args)
        {
            using (Game1 game = new Game1())
            {
                game.Window.Title = "Mi primera aplicación XNA";
                game.Run();
            }
        }
    }
}
```

Y la clase principal (`Game1`) la siguiente estructura:

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace MiPrimerProgramaXNA
{
    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

        /// <summary>
        /// Allows the game to perform any initialization it needs to
        before starting to run.
    }
}
```

```
/// This is where it can query for any required services and
load any non-graphic
/// related content. Calling base.Initialize will enumerate
through any components
/// and initialize them as well.
/// </summary>
protected override void Initialize()
{
    // TODO: Add your initialization logic here

    base.Initialize();
}

/// <summary>
/// LoadContent will be called once per game and is the place
to load
/// all of your content.
/// </summary>
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw
textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);

    // TODO: use this.Content to load your game content here
}

/// <summary>
/// UnloadContent will be called once per game and is the
place to unload
/// all content.
/// </summary>
protected override void UnloadContent()
{
    // TODO: Unload any non ContentManager content here
}

/// <summary>
/// Allows the game to run logic such as updating the world,
/// checking for collisions, gathering input, and playing
audio.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing
values.</param>
protected override void Update(GameTime gameTime)
{
    // Allows the game to exit
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
ButtonState.Pressed)
        this.Exit();

    // TODO: Add your update logic here

    base.Update(gameTime);
}

/// <summary>
/// This is called when the game should draw itself.
/// </summary>
```

```
/// <param name="gameTime">Provides a snapshot of timing
values.</param>
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    // TODO: Add your drawing code here

    base.Draw(gameTime);
}
}
```

Veamos las diferentes partes:

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    SpriteBatch spriteBatch;
    GraphicsDeviceManager graphics;

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";
    }
}
```

Lo primero que observamos es que la clase principal (Game1 en este caso) hereda de `Microsoft.Xna.Framework.Game`. Esto nos indica que es la clase principal. Como veremos más adelante se pueden definir otras clases que heredan de `Microsoft.Xna.Framework.GameComponent` y que, como su propio nombre indica, son partes o componentes de la aplicación.

A continuación vemos que se declaran dos variables. La variable de tipo `SpriteBatch` sirve para mostrar textos, imágenes, etc. por pantalla. No es vital para el funcionamiento de la aplicación. Sin embargo la siguiente variable (la de tipo `GraphicsDeviceManager`) sí que es fundamental para que todo funcione correctamente. Y es que la variable `graphics`, en este caso, es el puente entre nuestra aplicación XNA y la tarjeta gráfica del ordenador/XBOX/Windows Phone 7 o Zune. Más concretamente con la unidad de procesamiento gráfico (GPU) de dicha tarjeta. Por tanto, todo lo que se quiera mostrar por pantalla, obligatoriamente, va a pasar por este objeto.

Las siguientes líneas de código lo que hacen es inicializar dicha variable `graphics` y la última le dice a la aplicación cual va a ser la carpeta de contenidos (Content). Esta carpeta de contenidos es donde se guardarán los modelos, texturas, fuentes, etc. que se usen.

```
protected override void Initialize()
{
    // TODO: Add your initialization logic here
    base.Initialize();
}
```

El método Initialize() se usa para, como su propio nombre indica, inicializar las variables y otros objetos de nuestra aplicación.

```
protected override void LoadContent()  
{  
    // Create a new SpriteBatch, which can be used to draw textures.  
    spriteBatch = new SpriteBatch(GraphicsDevice);  
  
    // TODO: use this.Content to load your game content here  
}
```

El método LoadContent() se usa para cargar el contenido gráfico (imágenes, modelos, sonidos, etc.)

A partir de aquí, la aplicación entra en un bucle infinito. Este bucle se conoce como *game loop*.

Este bucle, como hemos comentado antes, se compone de dos métodos (Update y Draw)

```
protected override void Update(GameTime gameTime)  
{  
    // Allows the game to exit  
    if(GamePad.GetState(PlayerIndex.One).Buttons.Back==  
    ButtonState.Pressed)  
        this.Exit();  
  
    // TODO: Add your update logic here  
  
    base.Update(gameTime);  
}
```

El método Update() se encarga de gestionar toda la lógica de la aplicación (choques, aumentos de variables, comprobaciones de botones pulsados, etc.) y de modificar los valores apropiados. En este ejemplo comprueba si hemos pulsado el botón *Back* del mando que tengamos conectado al PC. Si ha sido pulsado invocamos el método Exit() de la aplicación.

```
protected override void Draw(GameTime gameTime)  
{  
    GraphicsDevice.Clear(Color.CornflowerBlue);  
  
    // TODO: Add your drawing code here  
  
    base.Draw(gameTime);  
}
```

El método Draw se encarga, como su propio nombre indica, de dibujar en pantalla. También se encarga, como en este caso, de limpiar todo lo dibujado en pantalla.

```
protected override void UnloadContent()  
{  
    // TODO: Unload any non ContentManager content here  
}
```

Por último, el método `UnloadContent()` se usa para descargar y destruir los distintos componentes. Por defecto la tecnología .NET ejecuta este método automáticamente cuando se sale de la aplicación, pero se puede llamar o en cualquier momento en el que queramos descargar algún componente cargado en memoria (modelos, sonidos, etc.).

Se observa también que en los métodos `Initialize`, `Update` y `Draw` se está llamando a dicho método pero de la clase `base` y con el parámetro `gameTime`. Esta llamada lo que hace es una propagación hacia el resto de clases y `GameComponents` que existen en la aplicación.

Si ejecutamos este código la salida será la que observamos en la siguiente figura:

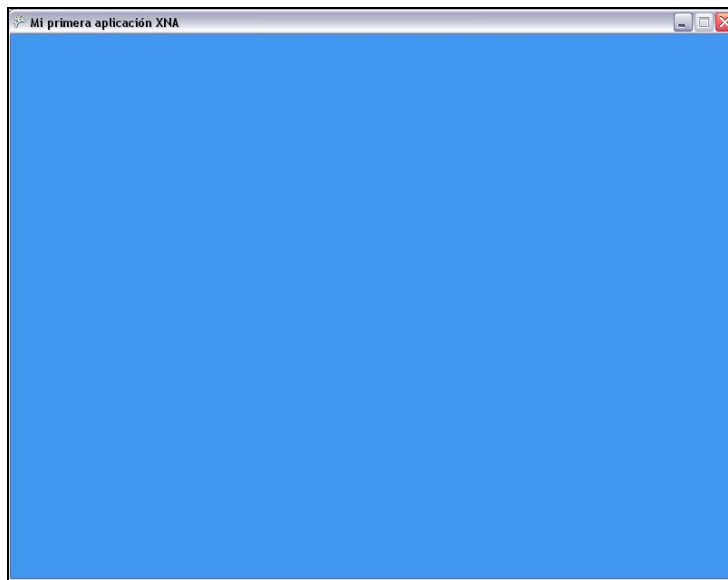


Figura 3.2.1 - Nuestra primera aplicación

No es precisamente espectacular pero es nuestra primera aplicación XNA.

3.3 Ejemplos

Evidentemente lo que más nos puede llegar a llamar la atención, cuando valoramos el uso de las diferentes herramientas para la creación de un juego, son los ejemplos de lo que la herramienta puede llegar a hacer.

Veamos pues que es lo que XNA nos ofrece con una serie de ejemplos: **2D**

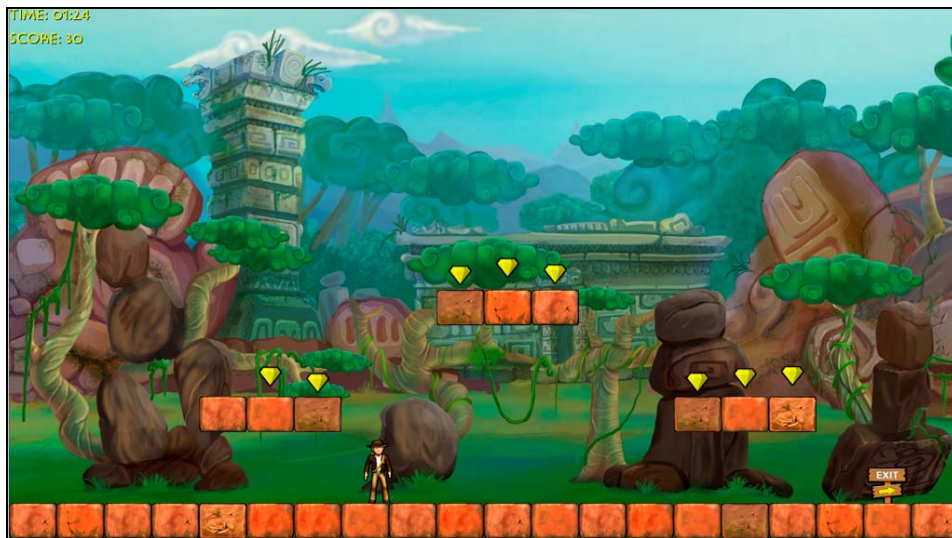


Figura 3.3.1 - Plataformas 2D

3D

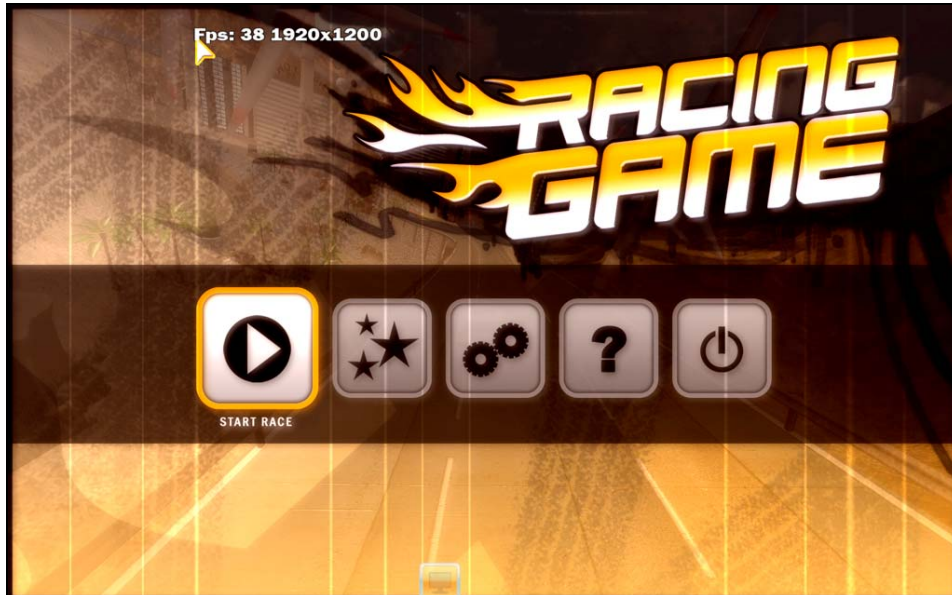


Figura 3.3.2 - Racing Game (Menú)



Figura 3.3.3 - Racing Game (Selección coche)



Figura 3.3.4 - Racing Game (Pantalla juego)



Figura 3.3.5 - Robot Game desarrollado por Zepetto (Menú)



Figura 3.3.6 - Robot Game desarrollado por Zepetto (Juego)



Figura 3.3.7 - Robot Game desarrollado por Zepetto (Juego)

Como vemos se pueden lograr auténticas maravillas pero, lógicamente, a mayor complejidad mayor número de horas de desarrollo o de desarrolladores. Aunque XNA facilita mucho la creación de juegos, tampoco la hace milagrosa.

3.4 XNA Internamente

Internamente XNA se compone de muchas clases agrupadas en las librerías que vemos en la siguiente figura.

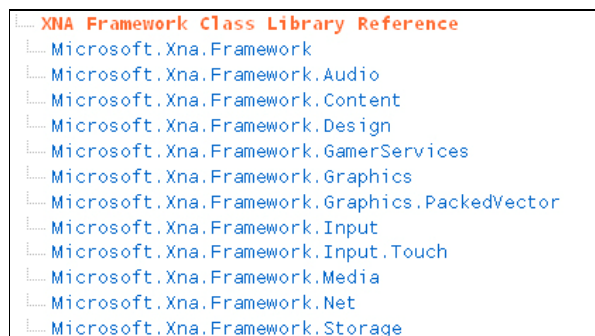


Figura 3.4.1 - Componentes XNA

En el apéndice 1 podemos ver el diagrama de clases completo del Framework XNA. No vamos a analizar todos los componentes de XNA pero sí que vamos a hablar de dos de los *pipelines* principales. El *pipeline* se podría traducir como tubo de procesos o simplemente como el proceso que se ejecuta para conseguir una determinada acción. Vamos a hablar en primer lugar del *Content pipeline*, es decir el proceso que se sigue internamente para cargar contenido.

3.4.1 Content pipeline

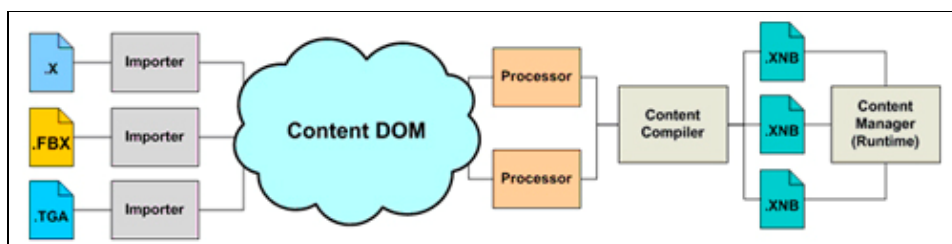


Figura 3.4.2 - XNA Framework Content pipeline

Uno de los mayores problemas existentes en la creación de juegos es la gestión del contenido. Existen muchos problemas: exportar los modelos a un formato soportado por el framework, encontrar herramientas para procesar el contenido y utilizar dichas herramientas para crear contenido para tu juego. Además debemos cargar el contenido y hacer todo lo necesario para que dicho contenido se vea bien. Esto supone una enorme carga de trabajo.

Con el *content pipeline* de XNA este proceso es diferente (y más sencillo). El contenido se maneja desde el propio Visual Studio, por tanto, al igual que añadimos código podemos añadir contenido. Esto permite mantener todo ordenado y en una única solución. Además a cada contenido podremos decirle qué *Importer* y qué *Processor* va a usar.

Como vemos en la figura anterior existen varios componentes. Para empezar, cuando añadimos el contenido debemos elegir un *Importer*. Éste es responsable de obtener la información y normalizarla, es decir, obtiene la información según viene del programa de creación (3D Studio, Blender, Photoshop, etc.) y la normaliza para poder ser usada en nuestro juego. Esto tiene una gran ventaja: ya no estamos atados a la aplicación de creación sino que ahora podemos hacer el contenido en el programa que queramos porque el *Importer* es el mismo para todos. Es decir, que si se crea un modelo .FBX en 3D Studio y otro en Blender, el *Importer* utilizado será simplemente el de carga de modelos propio de XNA (no habrá uno para Blender y otro para 3D Studio).

Cuando añadimos el contenido debemos seleccionar un *Importer*. En la siguiente figura vemos un ejemplo de la asignación del *Importer*. Seleccionamos el *Importer* de XNA Framework, en este caso para ficheros .FBX.

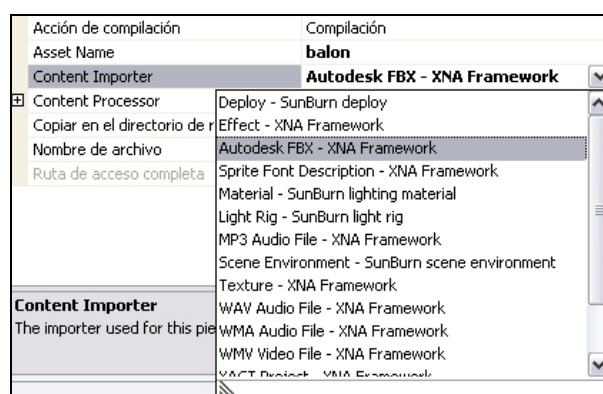


Figura 3.4.3 - Importer de Visual Studio

A pesar de que es independiente de en qué programa lo hayamos creado, el contenido sí que tiene que ser de determinada extensión. Veamos qué tipos de fichero soporta XNA:

- Modelos 3D: .FBX y .X
- Imágenes 2D: .DDS, .BMP, .JPG, .PNG y .TGA
- Materiales: .FX
- Audio: .XAP, .WAV, .WMA y .MP3

Una vez normalizada la información del contenido, dicha información estará almacenada en una colección de clases a la que coloquialmente se le conoce como DOM (*Data Object Model*). La información se guarda fuertemente tipada, es decir que

siempre será igual venga de donde venga el contenido. Esto es importante ya que al *Processor* le debe llegar la información perfectamente.

Hablemos pues del *Processor*. Éste es el encargado de obtener la información del DOM y crear un objeto que pueda ser usado en *ContentManager*. El *content pipeline* dispone de varios *Processors* ya implementados (**Model** para modelos, **Effect** para los materiales del modelo y **Texture2D** para sprites o para las texturas del modelo).

El *Processor* nos permite modificar una serie de parámetros que afectarán al rendimiento de nuestra aplicación. En la siguiente figura vemos un ejemplo de alguno de estos parámetros. En este caso los que vemos a **True** son parámetros para que optimizar modelos.



Propiedades	
box.fbx Propiedades del archivo	
Generate Tangent Frames	False
Model Material Names	False
Optimize Mesh Bounds	True
Optimize Model	True
Resize Textures to Power of Two	True
Scale	1
Swap Winding Order	False
Texture Format	DxtCompressed
X Axis Rotation	0
XSI Support	False

Figura 3.4.4 - Parámetros del Processor

En principio el *Processor* nos evita tener que preocuparnos, por ejemplo, de los buffers de vértices, pero puede que queramos una mayor flexibilidad a la hora de crear nuestros objetos. Veamos un ejemplo:

Imaginemos que queremos hacer un juego de carreras. Tendríamos nuestros modelos de coches, el terreno, edificios, etc. Pero ahora pensemos en la carretera, pensemos que queremos tener checkpoints, la salida, la llegada, etc. ¿deberíamos tener un modelo para cada trozo de pista donde exista un punto de estos? La respuesta es obvia, no. Lo que nos interesaría es crear nuestro propio *Processor* que consuma toda esta información y que como salida tenga un objeto *Pista*.

Los *Processors* han sido diseñados para ser escritos de una forma fácil además de ser altamente reutilizables. Como hemos dicho el *Importer* normaliza la información y ahora es cuando vemos la importancia de esto ya que podemos crear nuestro *Processor* sin preocuparnos de si la información que nos llega es de un fichero *.X*, *.FBX*, *.BMP*, etc. sino que siempre es la misma.

Por último, necesitamos una forma de usar toda esta información en nuestro juego. De esto se va a encargar el *ContentManager*. Este componente está diseñado para cargar contenidos de forma rápida, sin mucho código. Se encarga de mirar que

todos los subcontenidos estén cargados también. Como veremos más adelante, cargar contenido es tan sencillo como:

```
ContentManager Content = new ContentManager(GameServices);
modelo = Content.Load<Model>("modelo");
```

Con formato: Inglés (Reino Unido)

3.4.2 Render pipeline

Por otra parte tenemos el *pipeline* de renderizado. XNA realiza una llamada al pipeline de renderizado de DirectX. Este *pipeline* tiene la siguiente estructura:

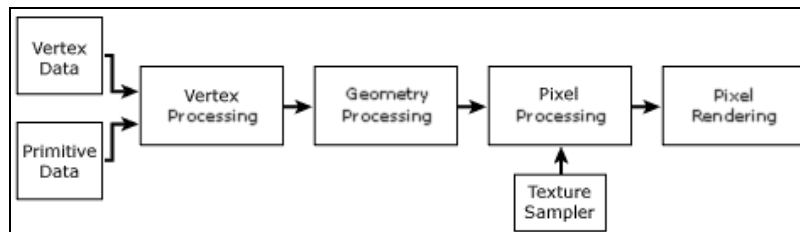


Figura 3.4.5 - Pipeline de renderizado

Vertex Data: Almacena los vértices del modelo sin transformar. Este componente contiene la clase `VertexDeclaration` con la que podemos acceder a la información de dichos vértices (Color, coordenadas de texturas, posición, etc.).

Primitive Data: Todas las primitivas (incluyendo líneas, triángulos, polígonos, etc.) se referencian en este componente a través de buffers de indexados. La ventaja de los buffers indexados es que permiten no tener información redundante (ya que muchas veces diferentes primitivas comparten un mismo índice).

Vertex Processing: En este componente, como su propio nombre indica, es donde se procesan los vértices. Aplicando los shaders de vértices de un `Effect` transformamos los vértices almacenados en el buffer de vértices. Usando las matrices de vista, proyección, etc. asignadas a dicho `Effect` podemos también transformar los vértices. Es decir, aquí es donde se usan nuestros efectos HLSL.

Geometry Processing: Este componente gestiona los siguientes procesos:

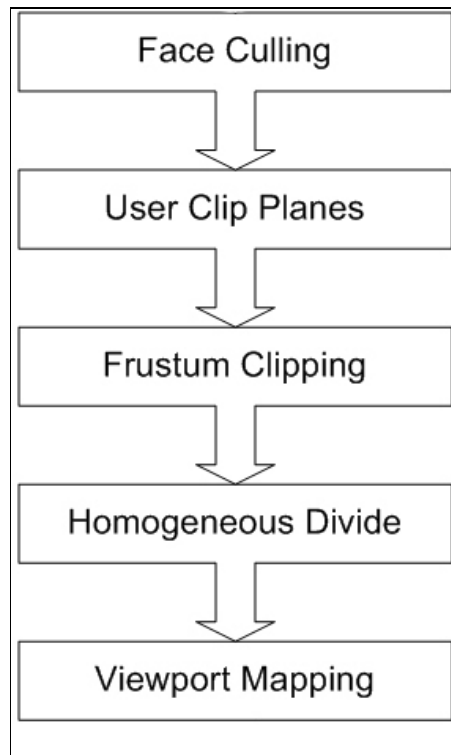


Figura 3.4.6 - Procesos del componente Geometry Processing

- *Face Culling*: Se eliminan de la pantalla los triángulos que no miran hacia la cámara (el ángulo formado por el vector normal del triángulo y el vector lookAt de la cámara es mayor de 90°).
- *User Clip Planes*: Si el usuario ha definido planos de corte, aquí es donde se realiza el test y todos los triángulos que están por detrás de estos planos son borrados de la escena.
- *Frustum Clipping*: Clipping es el proceso por el cual los triángulos que no aparecen dentro de la visión de la cámara son eliminados de la pantalla.
- *Homogeneous Divide*: La información x, y, z de cada vértice es transformada a non-homogeneous space antes de ser enviada al *rasterizer* (rasterizador). Los espacios homogéneos tienen que ver con el modo en que un punto tridimensional se proyecta en un espacio bidimensional.
- *Viewport Mapping*: Se lleva cabo el proceso de asignación de píxeles de pantalla a cada triángulo.

Texture Sampler: En este componente se aplica a las texturas el filtrado por nivel de detalle. Estas texturas serán usadas por el componente Pixel shader de un

Effect. Incluye además la clase TextureAddressMode de cada coordenada de cada textura y la clase TextureFilter que se usan para redimensionar la textura para que se ajuste al objeto de la pantalla.

Píxel Processing: En este componente el Pixel shader de un **Effect** usa los datos de la geometría para combinar los vértices de entrada con los datos de las texturas además de las ecuaciones de iluminación obteniendo así el color de salida del píxel.

Píxel Rendering: Este componente realiza el proceso final de modificar el color del píxel con alfas además de otros procesos. Todos los píxeles resultantes son los que vemos por pantalla. Aquí vemos más en detalle el proceso que se produce en este componente.

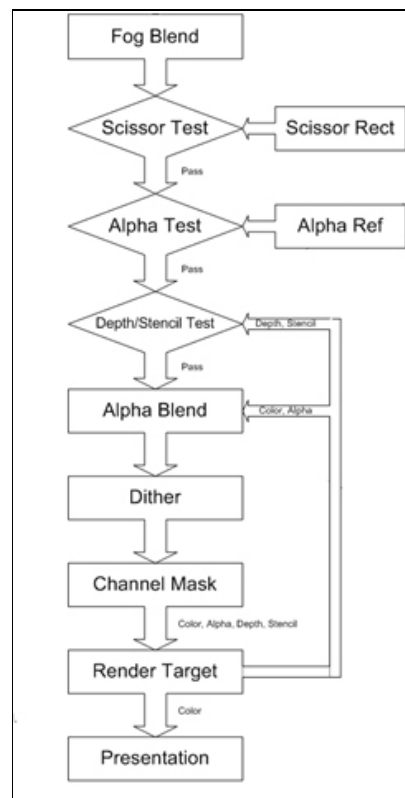


Figura 3.4.7 - Componente Píxel Rendering

- **Fog Blend:** La API nos proporciona la funcionalidad de aplicar niebla a la escena y es aquí donde se realiza el test.
- **Scissor Test:** También podemos definir un rectángulo de la pantalla (Scissor Rect) he indicar al Device que sólo se renderize esa porción del buffer. El test en el que se determina si los píxeles están dentro o fuera de dicho rectángulo se realiza aquí.

- *Alpha Test*: Se realiza una comprobación para evaluar mediante los valores del canal alpha si cumplen o no una condición marcada por el Alpha Ref.
- *Depth/Stencil Test*: Se llevan a cabo los test de Depth (profundidad), se actualiza el depth buffer con la profundidad del píxel si este será visible. También se aplica el Stencil Test (plantilla) para ver si el píxel afectará al color final del píxel en pantalla.
- *Alpha Blend*: En él se realizan las mezclas de colores que hacen posible el dibujado de objetos semitransparentes.
- *Dither*: Utiliza un algoritmo de interpolación de colores para combinar los píxeles adyacentes y así conseguir una gama de colores más consistente.
- *Channel mask*: Se puede escribir solo en los canales que deseemos (`RenderState.ColorWriteChannels`).
- *RenderTarget*: Se colocan los píxeles en el Render Target.
- *Presentation*: El RenderTarget es presentado por pantalla en el monitor.

Con estos componentes tenemos todo el proceso de renderizado de una aplicación en XNA. Veamos cual es exactamente dicho proceso y las modificaciones que se pueden hacer.

El pipeline gráfico (en 3D) utiliza un dispositivo gráfico (GraphicsDevice) para cargar recursos y renderizar la escena 3D usando un efecto (Effect)

En principio necesitamos esto para empezar:

- Matrices de world, view y projection.
- Un buffer de vértices (Vertex Buffer).
- Un efecto (Effect) que aplica las matrices world, view y projection a dicho buffer.

Con formato: Francés (Francia)

Con esto podríamos empezar una aplicación sencilla pero XNA nos permite además:

- Manipular los vértices creados.
- Crear efectos propios.
- Aplicar texturas.

- Mejorar el rendimiento con diferentes técnicas de gestión de buffers (por ejemplo convirtiendo el buffer de vértices a uno del mismo tipo pero indexado)

XNA utiliza el llamado un pipeline de shader totalmente programable. Requiere un tarjeta gráfica capaz de renderizar Shader Model 1.1 aunque se recomienda 2.0 como mínimo. En este pipeline los shaders y efectos se usan para aplicar transformaciones, texturas, iluminación y materiales. Sin embargo, para empezar, XNA provee de la clase `BasicEffect` que contiene las operaciones más básicas.

El proceso básico es este:

1. El `GraphicsDeviceManager` inicializa el `GraphicsDevice` antes de la llamada a `Game.Initialize`. Y precisamente antes de dicha llamada se pueden cambiar las propiedades del `GraphicsDevice` como `PreferredBackBufferHeight`, `PreferredBackBufferWidth`, etc. en el `GraphicsDeviceManager` en nuestro constructor. Además se podrán configurar más opciones.
2. Cuando se realiza la llamada a `Game.Initialize`, el `GraphicsDeviceManager` crea y configura el `GraphicsDevice`. Entonces se puede acceder a sus propiedades (back buffer, depth/stencil buffer, viewport, render states, etc.) en el método `Initialize`. Hay que tener en cuenta que después de la llamada a `Game.Initialize`, los cambios en `PresentationParameters` del `GraphicsDevice` no tendrán efecto hasta que llamemos al método `GraphicsDeviceManager.ApplyChanges()`. El resto de cambios (render states, etc.) tendrán efecto inmediato.

Como hemos visto el pipeline gráfico de XNA utiliza los llamados recursos. Los recursos son una parte vital de XNA. Son una colección de datos almacenada en memoria que pueden ser accedidos por la GPU o la CPU. Algunos ejemplos son los render targets, vertex buffers, index buffers y las texturas.

Anteriormente se ha hablado de los buffers de vértices pero ahora se verán más en profundidad. Los buffers de vértices contienen una lista de vértices 3D que serán enviado en stream al dispositivo gráfico (`GraphicsDevice`). Cada vértice contiene datos no solo de la coordenada 3D sino también información de descripción de dicho vértice; como puede ser la normal del vértice, el color, la coordenada de la textura, etc. Tenemos también varias clases para obtener esta información; como son `VertexPositionColor`, `VertexPositionColorTexture`, `VertexPositionNormalTexture`, `VertexPositionTexture`.

Como hemos dicho anteriormente estos buffers pueden estar o no indexados. Si no está indexado los vértices son almacenados en el orden en el que van a ser renderizados. Pero como ya hemos comentado muchos objetos pueden tener el mismo vértice en común por lo que estaríamos redundando la información. Por contra los buffers indexados permiten ordenar los vértices teniéndolos así solamente una vez.

Para renderizar un buffer no indexado hay que llamar al método `GraphicsDevice.DrawPrimitives` o a `GraphicsDevice.DrawUserPrimitives`. Para hacerlo con uno indexado debemos llamar al método `GraphicsDevice.DrawIndexedPrimitives` o a `GraphicsDevice.DrawUserIndexedPrimitives`.

Otro recurso son las texturas. Las texturas (a nivel básico) son una colección estructurada de datos. Estos datos están compuestos por subrecursos organizados en arrays. La textura se puede aplicar usando la propiedad `Texture` de la clase `BasicEffect` o se puede crear un efecto propio para aplicar las texturas.

Este sería el proceso básico que sigue un elemento desde que se carga hasta que se renderiza. Con esto ya tenemos una ligera idea de como funciona el *pipeline* gráfico del framework XNA. Veamos ahora cuales han sido los pasos para la implementación de la aplicación final.

Capítulo 4

IMPLEMENTACIÓN DE LA APLICACIÓN

En este capítulo vamos a tratar la implementación de la aplicación final que ha sido creada con XNA. Describiremos los pasos seguidos para llegar al resultado final, así como los intentos previos.

4.1 Primeros pasos

A la hora de crear una aplicación de estas características se tiene que tener claro qué es lo que se quiere hacer para después ver cómo hacerlo.

La aplicación debía cumplir una serie de requisitos:

- **Entorno 3D:** Aunque la jugabilidad y utilidad de las aplicaciones no depende exclusivamente de la calidad de sus gráficos, sí se quería conseguir un nivel de calidad aceptable para conseguir una mayor inmersión en la escena y para probar las posibilidades que ofrece XNA como entorno de desarrollo 3D.
- **Fácil manejo:** Para un correcto uso de la aplicación se exige un fácil manejo de la misma. Por eso se proporcionará la ayuda necesaria para aprender el manejo.
- **Inmersión en el mundo creado:** A parte del, ya mencionado, uso de las tres dimensiones para crear un mundo más real, se pretendía utilizar una serie de recursos que mejoraran dicha inmersión (partículas, sonidos, etc.).
- **Física creíble:** Otro de los puntos claves en la aplicación era el estudio de las diferentes posibilidades para recrear un comportamiento físico creíble.
- **Iluminación avanzada:** Por último se pretendía también conseguir una iluminación bonita y lo más parecida a la realidad.

4.2 Desgranando los componentes

Si bien la creación de juegos y aplicaciones 2D no conlleva excesivas complicaciones desde el punto de vista espacial, las aplicaciones 3D sin duda suponen un reto mayor. Añadir una tercera dimensión tiene muchas implicaciones (espacios, escalas, matrices, etc.).

Por tanto para realizar una implementación eficaz de la aplicación, se necesitaba desgranar todas y cada una de las funcionalidades que íbamos a implementar y ver todas las soluciones posibles; desde las iniciales y más básicas hasta las más complejas.

El primer paso era aprender a comprender qué supone un mundo 3D (más complejidad y cálculo), qué matemática hace falta para manejarlo (multiplicación de vectores, matrices, etc.) y qué componentes hay que desarrollar para navegar por él (cámaras libres, fijas, pivotantes, etc.).

Para entender el mundo 3D hay que tener en cuenta como vamos a orientarnos y cuales van a ser las 3 dimensiones. Y es que existen dos formas de representar el espacio tridimensional. Las que tradicionalmente se llaman *sistema de la mano izquierda* y *sistema de la mano derecha*. La siguiente figura ilustra dichos sistemas:

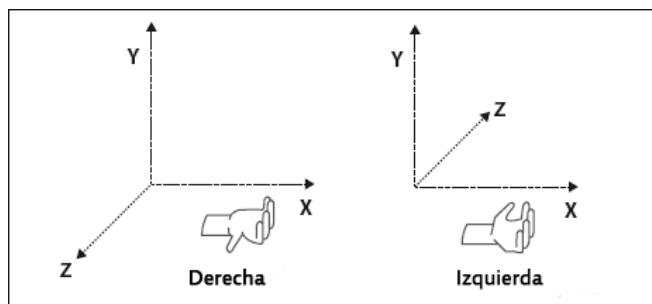


Figura 4.1.1 - Sistemas de coordenadas

El sistema que nos interesa en nuestro caso es el de la derecha ya que XNA lo usa. Lo cual quiere decir que cuando estamos mirando con la cámara hacia el origen la coordenada positiva de X irá hacia nuestra derecha, la de la Y hacia arriba y la de la Z hacia nosotros,

4.2.1 Primera aplicación 3D

Así pues el primer paso fue crear una aplicación que simplemente cargara un modelo en pantalla y que a través de una cámara (manejada mediante teclado y ratón) nos permitiera navegar a su alrededor. Para ello, primero había que cargar el modelo y a continuación crear la cámara. En este caso la estructura de la aplicación es la siguiente:

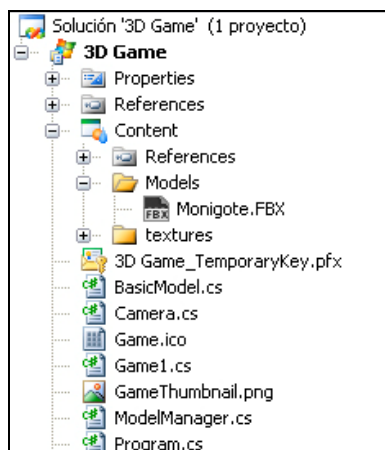


Figura 4.2.2 - Estructura de la aplicación

Y el resultado es el que se muestra en la figura:

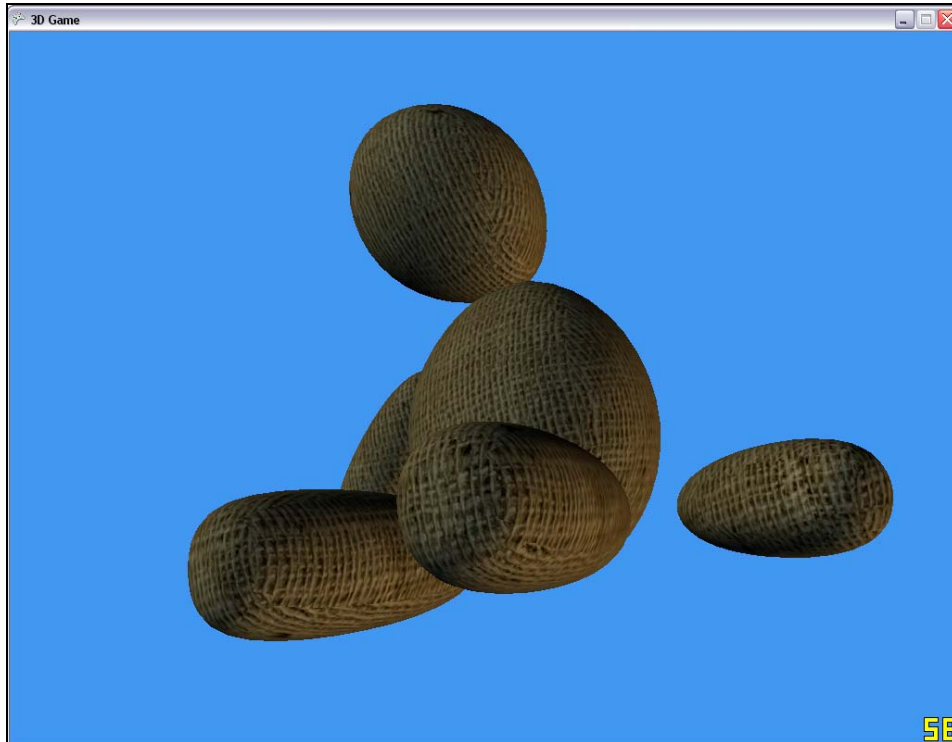


Figura 4.2.1 - Carga de modelo 3D y navegación con cámara libre

Se observan varios elementos nuevos respecto a la aplicación base. Lo primero que llama la atención es el archivo `Monigote.FBX`. Este archivo es el modelo 3D. XNA acepta tanto archivos `.FBX`, como `.X`. Para poder exportar nuestros modelos desde 3D Studio Max o cualquier otro programa de modelado 3D, necesitamos el plugin correspondiente. En el caso de 3D Studio Max, en las versiones más modernas, el plugin viene integrado en la propia aplicación. Más adelante se explicará mejor el proceso o *pipeline* que deben seguir los modelos que integrarán nuestras escenas.

A continuación llama la atención las dos nuevas clases incluidas en el proyecto (`BasicModel` y `ModelManager`). Aunque el nombre es bastante intuitivo respecto a su función vamos a explicarlas en detalle.

BasicModel.cs

Esta clase va a ser nuestra clase base a la hora de gestionar modelos.

```
class BasicModel
{
    public Model model { get; protected set; }
    public Matrix world = Matrix.Identity;
}
```

```

public BasicModel(Model m)
{
    model = m;
}

public virtual void Update()
{
    //Base class does nothing here
}

public void Draw(Camera camera)
{
    Matrix[] transforms = new Matrix[model.Bones.Count];
    model.CopyAbsoluteBoneTransformsTo(transforms);

    foreach (ModelMesh mesh in model.Meshes)
    {
        foreach (BasicEffect be in mesh.Effects)
        {
            be.EnableDefaultLighting();
            be.Projection = camera.projection;
            be.View = camera.view;
            be.World = GetWorld() * mesh.ParentBone.Transform;
        }

        mesh.Draw();
    }
}

public virtual Matrix GetWorld()
{
    return world;
}

```

Al igual que anteriormente vamos a ver el código en detalle:

```

public Model model { get; protected set; }
public Matrix world = Matrix.Identity;

```

Con formato: Español (alfab. internacional)

La variable `model` de tipo `Model` es la que se usará para representar los modelos 3D en memoria. Como veremos se le asignará un valor pasado como parámetro en el constructor.

Por otro lado la variable `world` de tipo `Matrix` es la que representará el "mundo" de cada modelo en particular. Esta matriz representa dónde se debe dibujar el modelo y cómo debe ser rotado, escalado, etc.

```

public void Draw(Camera camera)
{
    Matrix[] transforms = new Matrix[model.Bones.Count];
    model.CopyAbsoluteBoneTransformsTo(transforms);

    foreach (ModelMesh mesh in model.Meshes)
    {
        foreach (BasicEffect be in mesh.Effects)

```

```
        {
            be.EnableDefaultLighting();
            be.Projection = camera.projection;
            be.View = camera.view;
            be.World = GetWorld() * mesh.ParentBone.Transform;
        }

        mesh.Draw();
    }
}

public virtual Matrix GetWorld()
{
    return world;
}
```

En este bloque tenemos varios puntos importantes. En primero lugar definimos un array de matrices (`Matrix[]`) para guardar las transformaciones de cada parte del modelo (en el caso de que tenga diferentes partes). El guardado en dicho array de matrices se hace con la llamada al método `CopyAbsoluteBoneTransformsTo`. Por último lo que hacemos es entrar en un bucle que recorre todas las mallas (meshes) del modelo y en cada malla se definen las propiedades de todos sus efectos (en este caso `BasicEffect`). Más adelante hablaremos de los diferentes efectos y del lenguaje utilizado para programar dichos. En este bucle le definimos una iluminación por defecto, la matriz de proyección y de vista de nuestra cámara y por último el mundo en el que estará dicho efecto. Al multiplicar el mundo por la transformación de nuestra malla nos aseguramos de que el modelo va a ser dibujo donde le corresponde.

Al final se llama al método `Draw()` de la malla (mesh). Si no lo hiciéramos no se mostraría nada en pantalla.

Teniendo ya la clase base para dibujar nuestros modelos, lo más natural es crear una clase que nos permita manejar dichos modelos de una forma sencilla. De esto se encarga la clase `ModelManager`.

ModelManager.cs

```
public class ModelManager : DrawableGameComponent
{
    List<BasicModel> models = new List<BasicModel>();
    Matrix x;
    Matrix y;
    Matrix z;

    public ModelManager(Game game)
        : base(game)
    {
    }
    protected override void LoadContent()
    {
        models.Add(new
BasicModel(Game.Content.Load<Model>(@"models\Monigote"));

        base.LoadContent();
    }
}
```



```
public override void Initialize()
{
    base.Initialize();
}

public override void Update(GameTime gameTime)
{
    if (Keyboard.GetState().IsKeyDown(Keys.Q))
        models.Add(new
BasicModel(Game.Content.Load<Model>(@"models\Monigote")));
    if (Keyboard.GetState().IsKeyDown(Keys.Down))
    {
        x = Matrix.CreateTranslation(new Vector3(1, 0, 0));
        models[0].world = models[0].world * x;
    }
    if (Keyboard.GetState().IsKeyDown(Keys.Up))
    {
        y = Matrix.CreateTranslation(new Vector3(0, 0, 1));
        models[0].world = models[0].world * y;
    }
    if (Keyboard.GetState().IsKeyDown(Keys.Left))
    {
        z = Matrix.CreateTranslation(new Vector3(0, 1, 0));
        models[0].world = models[0].world * z;
    }

    // Loop through all models and call Update
    for (int i = 0; i < models.Count; ++i)
    {
        models[i].Update();
    }

    base.Update(gameTime);
}

public override void Draw(GameTime gameTime)
{
    // Loop through and draw each model
    foreach (BasicModel bm in models)
    {
        bm.Draw(((Game1)Game).camera);
    }
    base.Draw(gameTime);
}
```

En primer lugar tenemos la declaración de cuatro variables

```
List<BasicModel> models = new List<BasicModel>();
Matrix x;
Matrix y;
Matrix z;
```

La primera de ellas es una lista de `BasicModel` para un mejor manejo de dichos modelos. El resto de variables nos sirven para hacer pruebas y observar mejor el comportamiento del mundo (world) de cada objeto de la escena.

```
protected override void LoadContent()
{
    models.Add(new
BasicModel(Game.Content.Load<Model>(@"models\Monigote")));

    base.LoadContent();
}
```

Cargar un modelo es tan sencillo como añadir un nuevo `BasicModel` a la lista con el modelo que queremos cargar.

```
if (Keyboard.GetState().IsKeyDown(Keys.Q))
    models.Add(new
BasicModel(Game.Content.Load<Model>(@"models\Monigote")));
if (Keyboard.GetState().IsKeyDown(Keys.Down))
{
    x = Matrix.CreateTranslation(new Vector3(1, 0, 0));
    models[0].world = models[0].world * x;
}
....

for (int i = 0; i < models.Count; ++i)
{
    models[i].Update();
}
```

Vemos que, al principio, se hacen dos comprobaciones de qué teclas se han pulsado en el teclado. En caso de que la tecla pulsada sea una Q, se añadirá un nuevo objeto. Si la tecla pulsada es la flecha hacia abajo en primer lugar asignaremos un valor a la matriz "x" declarada inicialmente mediante el método estático `Matrix.CreateTranslation` que lo que hace es crear una nueva matriz de traslación a partir de un vector de 3 posiciones u otros parámetros en sus diferentes sobrecargas. El siguiente paso es multiplicar mundo del primer modelo de nuestra lista por dicha matriz de traslación y reasignárselo a sí mismo. Con esto conseguimos que el objeto se mueva una posición hacia las X.

Por último se llama a cada método `Update` de los modelos. En este caso el método `Update()` de nuestra clase base no hace nada pero si lo hiciera sería necesario llamar a dicho método.

```
public override void Draw(GameTime gameTime)
{
    // Loop through and draw each model
    foreach (BasicModel bm in models)
    {
        bm.Draw(((Game1)Game).camera);
    }
    base.Draw(gameTime);
}
```

Para finalizar la explicación de esta clase vemos que se llama a cada método `Draw()` de nuestros modelos pasándole la cámara.

Lo siguiente que hay que implementar es la clase Camera.cs. Esta clase será la que nos permita navegar por el entorno 3D. Y es que es básico crear la cámara y decir la posición hacia la que va a apuntar, ya que solo los objetos que la cámara "ve" son los que se verán en la pantalla. Por ello, primero, vamos a introducir la teoría de cámaras y luego veremos la implementación de dicha teoría.

En una cámara tenemos dos matrices básicas: la matriz *view* o vista y la matriz *projection* o de proyección. La primera de ellas nos da información de dónde se encuentra la cámara en nuestro mundo, en qué dirección apunta y cuál es su orientación. La de proyección mantiene información del ángulo de visión, la distancia hasta la que podemos ver, etc. Esta matriz representa la transformación del mundo 3D en el plano 2D de nuestra pantalla.

Ahora que tenemos la teoría podemos continuar con la práctica. Aquí tenemos nuestra clase Camera.

Camera.cs

```
public class Camera : Microsoft.Xna.Framework.GameComponent
{
    public Matrix view { get; protected set; }
    public Matrix projection { get; protected set; }
    public Matrix world { get; protected set; }

    // Vectores
    public Vector3 cameraPosition { get; protected set; }
    Vector3 cameraDirection;
    Vector3 cameraUp;
    MouseState prevMouseState;
    float speed = 1f;

    public Camera(Game game, Vector3 pos, Vector3 target, Vector3
up)
        : base(game)
    {
        // Build camera view matrix
        cameraPosition = pos;
        cameraDirection = target - pos;
        cameraDirection.Normalize();
        cameraUp = up;
        CreateLookAt();
        projection =
Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
(float)Game.Window.ClientBounds.Width /
(float)Game.Window.ClientBounds.Height, 1, 3000);
    }

    /// <summary>
    /// Allows the game component to perform any initialization it
needs to before starting
    /// to run. This is where it can query for any required
services and load content.
    /// </summary>
    public override void Initialize()
    {
        // TODO: Add your initialization code here
    }
}
```

```

        // Set mouse position and do initial get state
        Mouse.SetPosition(Game.Window.ClientBounds.Width /
2,Game.Window.ClientBounds.Height / 2);
        prevState = Mouse.GetState();

        base.Initialize();
    }

    /// <summary>
    /// Allows the game component to update itself.
    /// </summary>
    /// <param name="gameTime">Provides a snapshot of timing
values.</param>
    public override void Update(GameTime gameTime)
    {
        // TODO: Add your update code here

        if (Keyboard.GetState().IsKeyDown(Keys.W))
            cameraPosition += cameraDirection * speed;
        if (Keyboard.GetState().IsKeyDown(Keys.S))
            cameraPosition -= cameraDirection * speed;

        if (Keyboard.GetState().IsKeyDown(Keys.A))
            cameraPosition += Vector3.Cross(cameraUp,
cameraDirection) * speed;
        if (Keyboard.GetState().IsKeyDown(Keys.D))
            cameraPosition -= Vector3.Cross(cameraUp,
cameraDirection) * speed;

        // Roll
        if (Mouse.GetState().MiddleButton ==
Microsoft.Xna.Framework.Input.ButtonState.Pressed)
        {
            cameraUp = Vector3.Transform(cameraUp,
Matrix.CreateFromAxisAngle(cameraDirection,
MathHelper.PiOver4 / 45));
        }
        if (Mouse.GetState().RightButton ==
Microsoft.Xna.Framework.Input.ButtonState.Pressed)
        {
            cameraUp = Vector3.Transform(cameraUp,
Matrix.CreateFromAxisAngle(cameraDirection,
-MathHelper.PiOver4 / 45));
        }

        // Yaw rotation
        cameraDirection = Vector3.Transform(cameraDirection,
Matrix.CreateFromAxisAngle(cameraUp, (-MathHelper.PiOver4
/ 150) *
(Mouse.GetState().X - prevState.X)));

        // Pitch rotation
        cameraDirection = Vector3.Transform(cameraDirection,
Matrix.CreateFromAxisAngle(Vector3.Cross(cameraUp,
cameraDirection),
(MathHelper.PiOver4 / 100) *
(Mouse.GetState().Y - prevState.Y)));
        cameraUp = Vector3.Transform(cameraUp,
Matrix.CreateFromAxisAngle(Vector3.Cross(cameraUp,
cameraDirection),

```

```
(MathHelper.PiOver4 / 100) *  
(Mouse.GetState().Y - prevMouseState.Y));  
  
prevMouseState = Mouse.GetState();  
CreateLookAt();  
  
base.Update(gameTime);  
}  
  
private void CreateLookAt()  
{  
    //posicion,objetivo,direccion  
    view = Matrix.CreateLookAt(cameraPosition, cameraPosition  
+ cameraDirection, cameraUp);  
}  
}
```

```
public Matrix view { get; protected set; }  
public Matrix projection { get; protected set; }  
public Matrix world { get; protected set; }  
  
// Vectores  
public Vector3 cameraPosition { get; protected set; }  
Vector3 cameraDirection;  
Vector3 cameraUp;  
MouseState prevMouseState;  
float speed = 1f;
```

Lo primero que vemos son las variables tipo `Matrix` que definirán las matrices que necesitamos para nuestra cámara (`view` y `projection`) y nuestro mundo (`world`). En segundo lugar vemos 3 vectores (`cameraPosition`, `cameraDirection` y `cameraUp`) que usaremos más adelante para crear la matriz `view`. Por último tenemos la variable `prevMouseState` para conocer el estado del ratón y la variable `speed` para definir la velocidad del movimiento de la cámara.

Observamos también que tanto las matrices `view`, `projection` y `world` como el vector `cameraPosition` son atributos públicos. Esto nos sirve para obtener el valor o asignarlo desde otras clases. Es muy importante ya que otras partes de nuestra aplicación pueden necesitar conocer las matrices de proyección o de vista en determinados momentos.

```

up) public Camera(Game game, Vector3 pos, Vector3 target, Vector3
    : base(game)
    {
        cameraPosition = pos;
        cameraDirection = target - pos;
        cameraDirection.Normalize();
        cameraUp = up;
        CreateLookAt();
        projection =
Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
(float)Game.Window.ClientBounds.Width /
(float)Game.Window.ClientBounds.Height, 1, 3000);
    }

```

La siguiente parte del código corresponde al constructor de la clase. Como vemos la posición, el objetivo y el vector up se pasan como parámetro. El vector dirección de la cámara lo obtendremos haciendo la resta simple del objetivo menos la posición. A continuación normalizaremos el vector dirección para que sea más manejable y por último crearemos la matriz de vista en la llamada al método `CreateLookAt()`; (que ahora veremos) y la matriz de proyección con el método estático `Matrix.CreatePerspectiveFieldOfView`.

En la siguiente figura vemos más claramente los valores del constructor:

Nombre	Valor
pos	{X:0 Y:0 Z:50}
target	{X:0 Y:0 Z:0}
up	{X:0 Y:1 Z:0}
target - pos	{X:0 Y:0 Z:-50}
cameraDirection	{X:0 Y:0 Z:-1}

Figura 4.2.3 - Valores del constructor

La cámara, por tanto, estará en la posición $\{X:0,Y:0,Z:50\}$ apuntando a la posición $\{X:0,Y:0,Z:0\}$ y su dirección será el vector $\{X:0,Y:0,Z:-50\}$ que tras normalizarlo se transformará en el vector $\{X:0,Y:0,Z:-1\}$

```

private void CreateLookAt()
{
    //posicion,objetivo,direccion
    view = Matrix.CreateLookAt(cameraPosition, cameraPosition
+ cameraDirection, cameraUp);
}

```

El método `CreateLookAt()` lo que va a hacer es llamar a su vez al método estático de `CreateLookAt()` de la clase `Matrix` pasándole como parámetros la posición de la cámara, el objetivo y el vector up. Vemos que en el segundo parámetro (el correspondiente al objetivo) no le pasamos la variable `target` del constructor. Esto es debido a que dicha variable es fija y no se modifica en el resto de programa y como veremos la matriz de vista tiene que cambiar continuamente para actualizarse con los

nuevos valores de posición y dirección. Por ello, el parámetro que le pasamos es la suma vectorial de la posición y la dirección. En esta figura lo vemos más claro con un ejemplo:

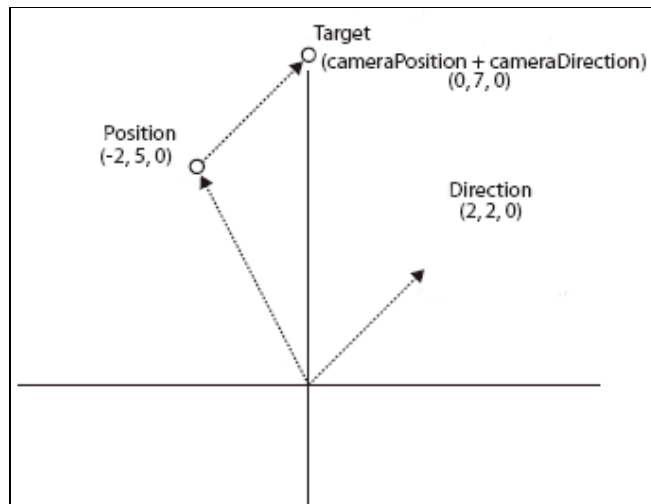


Figura 4.2.4 - Actualización del target de la cámara

Por último creamos la matriz de proyección con el método estático `CreatePerspectiveFieldOfView ()` de la clase `Matrix`. Este método tiene como parámetros:

- *fieldOfView*: Representa el ángulo de visión de la cámara en radianes. Normalmente se usan 45 grados ($\pi/4$).
- *aspectRatio*: El aspect ratio o relación de aspecto es la proporción entre la anchura y la altura y se calcula dividiendo la anchura por la altura de la imagen visible en la pantalla. Los formatos más comunes son 4:3 y 16:9.
- *nearPlaneDistance*: Define lo cerca que un objeto puede estar de la cámara antes de que deje de verse.
- *farPlaneDistance*: Define lo lejos que un objeto puede estar de la cámara antes de que deje de verse.

Con esto se crea el llamado *campo de visión* o *viewing frustum* el cual define un prisma o zona en la que los objetos de la escena se verán. En la siguiente figura se ilustra:

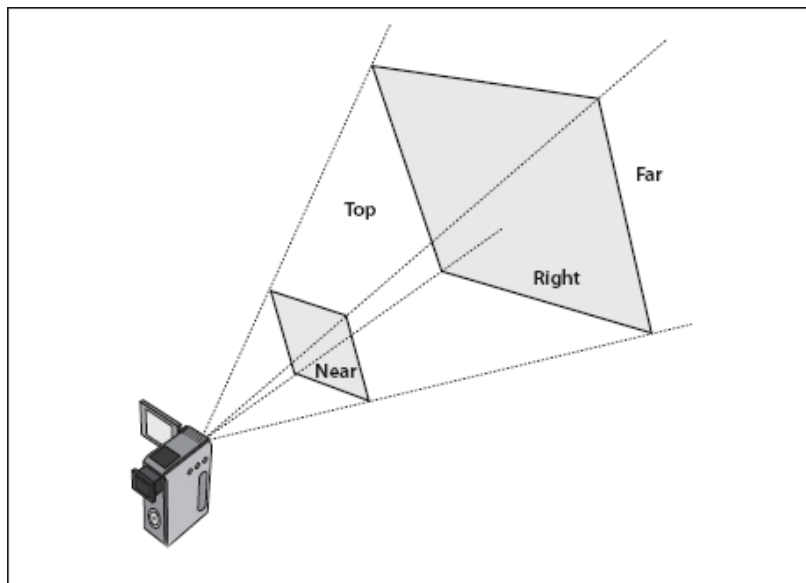


Figura 4.2.5 - Campo de visión

El campo de visión sería el prisma formado por los planos *Near* y *Far*.

Con esta lógica ya tendríamos una cámara implementada, y podríamos ver los objetos que carguemos delante suya. Sin embargo, y como es normal, queremos que nuestra cámara se mueva por el espacio. Para ello lo único que tendremos que hacer es definir en nuestro método `update()` las transformaciones necesarias para poder manejar la cámara.

Para ver las transformaciones aplicadas, primero deberemos conocerlas. Son tres y se les conoce como *Yaw*, *Pitch* y *Roll*. En la siguiente figura vemos un ejemplo:

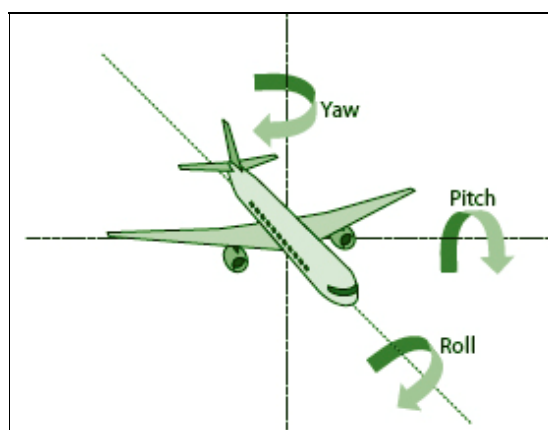


Figura 4.2.6 - Movimientos posibles en los diferentes ejes de coordenadas

Se podría decir que el movimiento *Yaw* es el que se produce en el eje de las Y, el *Pitch* en el eje de las X y el *Roll* en el eje de las Z. Sin embargo esto no es del todo cierto, y es que asociar dichos movimientos a los ejes puede llevar a confusión. Imaginemos que el avión de la figura gira 90° hacia su derecha (giro *Yaw*). Si realizamos entonces un giro *Roll* lo estará haciendo en un eje distinto al que lo haría si no lo hubiéramos girando en primera instancia. Por ello es mejor pensar en giros en función de los vectores de nuestra cámara. Así pues el movimiento *Yaw* estaría asociado a nuestro vector up, el movimiento *Roll* al vector dirección y el movimiento *Pitch* al vector producto vectorial entre el up y el dirección que más adelante se explicará.

Veamos como transformar estos movimientos en código:

```
public override void Update(GameTime gameTime)
{
    // TODO: Add your update code here

    if (Keyboard.GetState().IsKeyDown(Keys.W))
        cameraPosition += cameraDirection * speed;
    if (Keyboard.GetState().IsKeyDown(Keys.S))
        cameraPosition -= cameraDirection * speed;

    if (Keyboard.GetState().IsKeyDown(Keys.A))
        cameraPosition += Vector3.Cross(cameraUp,
cameraDirection) * speed;
    if (Keyboard.GetState().IsKeyDown(Keys.D))
        cameraPosition -= Vector3.Cross(cameraUp,
cameraDirection) * speed;

    // Roll
    if (Mouse.GetState().MiddleButton ==
Microsoft.Xna.Framework.Input.ButtonState.Pressed)
    {
        cameraUp = Vector3.Transform(cameraUp,
Matrix.CreateFromAxisAngle(cameraDirection,
MathHelper.PiOver4 / 45));
    }
    if (Mouse.GetState().RightButton ==
Microsoft.Xna.Framework.Input.ButtonState.Pressed)
    {
        cameraUp = Vector3.Transform(cameraUp,
Matrix.CreateFromAxisAngle(cameraDirection,
-MathHelper.PiOver4 / 45));
    }

    // Yaw rotation
    cameraDirection = Vector3.Transform(cameraDirection,
Matrix.CreateFromAxisAngle(cameraUp, (-MathHelper.PiOver4
/ 150) *
(Mouse.GetState().X - prevMouseState.X)));

    // Pitch rotation
    cameraDirection = Vector3.Transform(cameraDirection,
Matrix.CreateFromAxisAngle(Vector3.Cross(cameraUp,
cameraDirection),
(MathHelper.PiOver4 / 100) *
```

```
(Mouse.GetState().Y - prevMouseState.Y));  
cameraUp = Vector3.Transform(cameraUp,  
Matrix.CreateFromAxisAngle(Vector3.Cross(cameraUp,  
cameraDirection),  
(MathHelper.PiOver4 / 100) *  
(Mouse.GetState().Y - prevMouseState.Y)));  
  
prevMouseState = Mouse.GetState();  
CreateLookAt();  
  
base.Update(gameTime);  
}
```

```
if (Keyboard.GetState().IsKeyDown(Keys.W))  
    cameraPosition += cameraDirection * speed;  
if (Keyboard.GetState().IsKeyDown(Keys.S))  
    cameraPosition -= cameraDirection * speed;  
  
if (Keyboard.GetState().IsKeyDown(Keys.A))  
    cameraPosition += Vector3.Cross(cameraUp,  
cameraDirection) * speed;  
if (Keyboard.GetState().IsKeyDown(Keys.D))  
    cameraPosition -= Vector3.Cross(cameraUp,  
cameraDirection) * speed;
```

Esta primera parte no se refiere a las rotaciones en sí, sino que gestionará el movimiento tanto hacia delante como hacia atrás y hacia los lados.

Para implementar el movimiento hacia delante y hacia detrás simplemente debemos sumar o restar a la posición actual de la cámara su dirección por una velocidad fija. Como vemos es muy sencillo ya que disponemos de un vector (el de dirección) que está en la misma posición hacia donde queremos movernos. Haciendo un símil sería como si el vector dirección fuera una vía de un tren y para movernos hacia delante o hacia atrás deberíamos seguir dicha dirección con cierta velocidad.

Para los movimientos laterales, sin embargo, no tenemos un vector propio por el que guiarnos. Para obtener dicho vector deberemos recurrir a la matemática vectorial y en este caso al producto vectorial (*cross product*) que dados dos vectores nos va a dar su vector perpendicular. En este caso los vectores que le pasamos al método `Cross` de la clase `Vector3` son el vector Up y el vector dirección de la cámara. Veámoslo gráficamente:

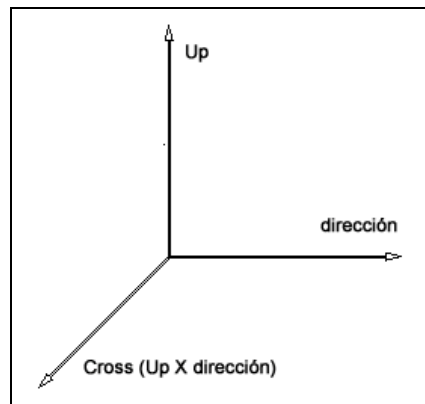


Figura 4.2.7 - Vector perpendicular (producto vectorial)

```

// Roll
if (Mouse.GetState().MiddleButton ==
Microsoft.Xna.Framework.Input.ButtonState.Pressed)
{
    cameraUp = Vector3.Transform(cameraUp,
Matrix.CreateFromAxisAngle(cameraDirection,
MathHelper.PiOver4 / 45));
}
if (Mouse.GetState().RightButton ==
Microsoft.Xna.Framework.Input.ButtonState.Pressed)
{
    cameraUp = Vector3.Transform(cameraUp,
Matrix.CreateFromAxisAngle(cameraDirection,
-MathHelper.PiOver4 / 45));
}

```

Aunque el movimiento *Roll* no siempre tiene sentido en una aplicación o juego 3D se ha implementado a modo de prueba. En este caso cuando pulsemos el botón central del ratón hará el giro positivo y cuando pulsemos el botón derecho lo hará de forma negativa.

Para ello utilizamos la llamada al método `Transform` de la clase `Vector3` que nos aplicará la transformación definida en el segundo parámetro al vector del primer parámetro. El vector será el vector up que es el que queremos cambiar y la transformación se obtendrá con la llamada al método `CreateFromAxisAngle` que nos generará una rotación (la que le decimos en el segundo parámetro) a través del vector que le indicamos como primer parámetro.

```

// Yaw rotation
cameraDirection = Vector3.Transform(cameraDirection,
Matrix.CreateFromAxisAngle(cameraUp, (-MathHelper.PiOver4
/ 150) *
(Mouse.GetState().X - prevMouseState.X)));

```

Como vemos la filosofía del giro *Yaw* es la misma que la del *Roll*. La única diferencia es el cambio de los vectores a modificar y que el ángulo de giro depende también de la posición del ratón (en el eje de las X) actual y anterior.

```
// Pitch rotation
cameraDirection = Vector3.Transform(cameraDirection,
Matrix.CreateFromAxisAngle(Vector3.Cross(cameraUp,
cameraDirection),
(MathHelper.PiOver4 / 100) *
(Mouse.GetState().Y - prevMouseState.Y)));

cameraUp = Vector3.Transform(cameraUp,
Matrix.CreateFromAxisAngle(Vector3.Cross(cameraUp,
cameraDirection),
(MathHelper.PiOver4 / 100) *
(Mouse.GetState().Y - prevMouseState.Y)));
```

En el caso del movimiento *Pitch*, el comportamiento es un poco diferente ya que no solo se modifica el vector perpendicular al vector dirección de la cámara (el que hemos comentado sería el producto vectorial entre dicho vector y el vector up), sino que también hay que modificar el vector up. El giro también dependerá del ratón; en este caso del eje Y.

Con esto tenemos una aplicación capaz de permitirnos un movimiento a través del espacio 3D con la carga simple de un modelo. En la siguiente figura se muestra el modelo tras acercarnos a él mediante la cámara.

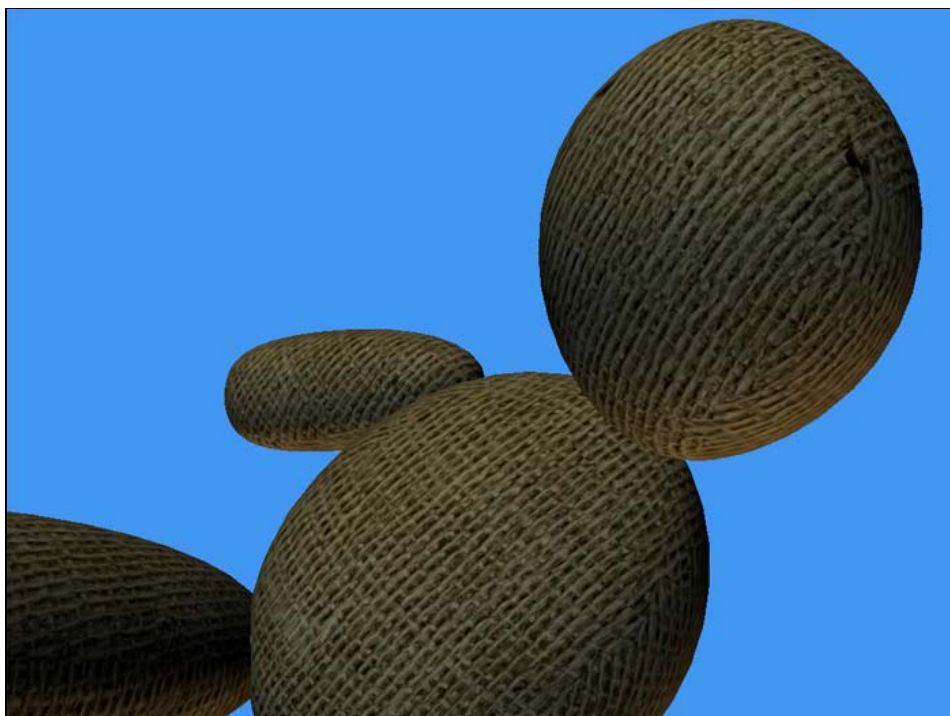


Figura 4.2.8 - Modelo tras acercarnos a él con la cámara

4.2.2 Nuevos objetivos (reconocimiento y selección)

Tras conseguir la carga de un modelo simple y la posterior navegación por la escena, se plantean dos cuestiones a resolver:

- Cargar varios modelos y saber cual estoy seleccionando con el ratón
- Mover el modelo a una posición deseada

Por tanto parece claro que el siguiente paso a investigar tiene mucho que ver con el reconocimiento de modelos.

El primer paso es saber cuando el ratón pasa por encima de un objeto. En una aplicación 2D es muy sencillo ya que al tener simplemente X e Y con saber si el objeto está entre dichas dimensiones ya lo tendríamos. Pero necesitamos saber si el puntero (que es un objeto bidimensional) está sobre el objeto (que está en un espacio tridimensional). Aparentemente parece un problema complejo, pero XNA proporciona un método que nos facilitará esta tarea. El método es el conocido como `Unproject` y lo que va a hacer es conseguir darnos las coordenadas del espacio tridimensional que tendría nuestro ratón. Sin embargo, debemos plantearnos hasta qué punto ya que imaginemos que estamos en la posición $\{X:10,Y:20\}$ y que en un espacio tridimensional esto equivale a la posición $\{X:0.8,Y:50,Z?\}$. Habrá que ver como se conoce la "profundidad" a la que se encuentra el objeto, y además saber si el ratón está interseccionando con el objeto o dicho objeto se encuentra mucho más lejos. Para saber si intersecciona con un objeto habrá que utilizar más elementos. Y estos elementos son los rayos o en XNA, la clase `Ray`.

El método de reconocimiento de objetos en XNA, y en la mayoría de aplicaciones 3D, se basa en el denominado *Raycasting* o *casteo de rayos*. Este método es tan simple como "lanzar" rayos desde un punto, en una determinada dirección y con un límite de alcance máximo. En el momento en el que un rayo intersecciona ya sabemos que hemos localizado un objeto, y para saber cual simplemente se llama al método que emite rayos por cada parte o malla de un modelo. Como ya sabemos si interseccionamos sabemos en qué modelo ha sido y por tanto podemos hacer las operaciones pertinentes. Puede parecer un método muy ineficiente, pero es uno de los más utilizados y no supone un coste excesivo de recursos.

Vamos a ver un ejemplo gráfico para verlo más fácilmente:

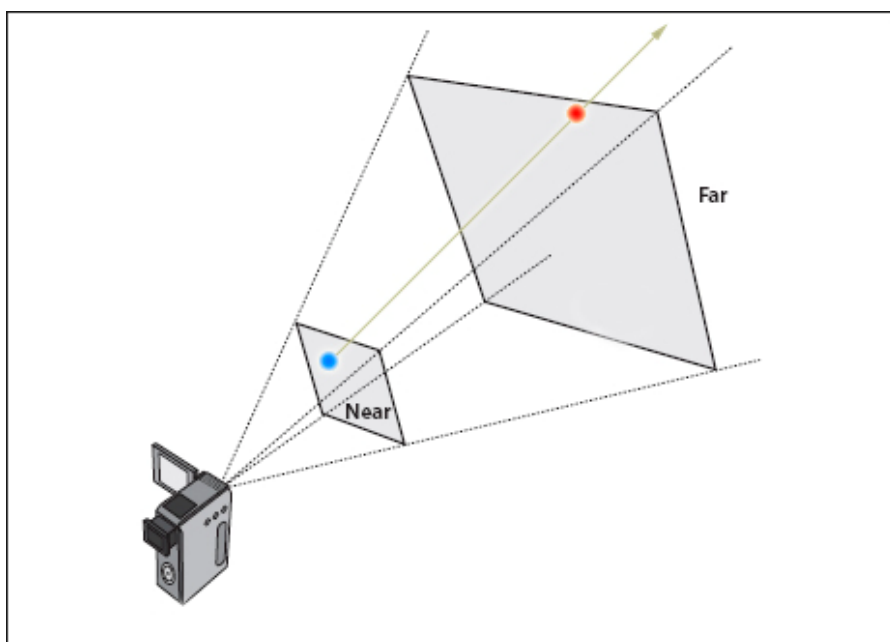


Figura 4.2.9 - RayCasting

Como se ve la figura es la misma que la usada al explicar el campo de visión de la cámara, y es que comparten muchos conceptos, el principal el de punto cercano y lejano. Y es que lo primero que se debe hacer es definir los límites del rayo, es decir, cuál va a ser la posición Z más cercana a nuestra cámara y cuál la más lejana. Por defecto, y normalizado, se suele tomar la coordenada Z igual a 0 como punto más cercano a la cámara y el punto 1 como más alejado. Esto es el equivalente (en un vector normalizado) a decir "todo" el eje, o dicho de otra forma, el rayo va a llegar hasta el "infinito", pero podríamos definir un límite en la distancia (por si no queremos coger objetos muy alejados o muy cercanos). Los puntos X e Y serán los del ratón en ese momento.

A continuación se calculará el punto en el espacio tridimensional más cercano a la cámara y el más lejano. Para esto se utiliza el método `Unproject` mencionado anteriormente. Una vez se tiene el punto más lejano posible y el más cercano se resta uno al otro para obtener el vector dirección que debe seguir el rayo.

Por último, con el punto más cercano posible y con la dirección ya se puede proyectar el rayo desde dicho punto de origen y ver si hay algún objeto que choque con dicho rayo.

A continuación se explicará, mediante código, un ejemplo implementado.

```
Ray cursorRay = CalculateCursorRay(this.camera.projection,  
this.camera.view);  
  
// go through all of the models...
```

```

for (int i = 0; i < models.Length ; ++i)
{
    if (RayIntersectsModel(cursorRay, models[i],
        models[i].GetWorld(),
        models[i].transforms,gameTime)
        )
    {
        Vector2 textPosition =
        new Vector2(Mouse.GetState().X,Mouse.GetState().Y-60);

        Vector2 stringCenter =
        spriteFont.MeasureString(nombresModelos[i]) / 2;

        Vector2 shadowOffset = new Vector2(1, 1);
        spriteBatch.DrawString(spriteFont,
        nombresModelos[i],textPosition + shadowOffset, Color.Black,
        0.0f,stringCenter, 1.0f,SpriteEffects.None, 0.0f);

        spriteBatch.DrawString(spriteFont, nombresModelos[i],
        textPosition, Color.White, 0.0f,
        stringCenter, 1.0f, SpriteEffects.None, 0.0f);
    }
}

```

Este trozo de código irá en nuestro método Update y de lo que se encarga es de crear el rayo y de comprobar si choca con alguno de los objetos. Pero vamos a ver primero las funciones auxiliares que utiliza:

```

public Ray (Vector2 mpos, Matrix projectionMatrix, Matrix
viewMatrix)
{
    Vector3 nearSource = new Vector3(mpos, 0f);
    Vector3 farSource = new Vector3(mpos, 1f);

    Vector3 nearPoint =
    GraphicsDevice.Viewport.Unproject(nearSource,
        projectionMatrix, viewMatrix, Matrix.Identity);

    Vector3 farPoint =
    GraphicsDevice.Viewport.Unproject(farSource,
        projectionMatrix, viewMatrix, Matrix.Identity);

    Vector3 direction = farPoint - nearPoint;
    direction.Normalize();

    return new Ray(nearPoint, direction);
}

```

El método CalculateCursorRay, como su propio nombre indica, es el encargado de calcular el rayo que sale del cursor. Para eso realizamos los pasos indicados con anterioridad:

```

Vector3 nearSource = new Vector3(mpos, 0f);
Vector3 farSource = new Vector3(mpos, 1f);

```

Primero definimos los límites del rayo.

```

Vector3 nearPoint =
GraphicsDevice.Viewport.Unproject(nearSource,
projectionMatrix, viewMatrix, Matrix.Identity);

Vector3 farPoint =
GraphicsDevice.Viewport.Unproject(farSource,
projectionMatrix, viewMatrix, Matrix.Identity);

```

A continuación calculamos el punto más cercano y el más lejano en nuestro mundo 3D. Para ello usamos el método `Unproject` pasándole como parámetros el punto (primero el inicial y luego el final), la matriz de proyección, la matriz de vista y por último nuestro mundo, que en este caso es la matriz identidad.

```

Vector3 direction = farPoint - nearPoint;
direction.Normalize();

return new Ray(nearPoint, direction);

```

Por último calculamos la dirección del vector, lo normalizamos y creamos un nuevo rayo, que devolveremos, que partirá del punto más cercano a la cámara con la dirección calculada.

```

private bool RayIntersectsModel(Ray ray, Modelo3D modelo, Matrix
worldTransform, Matrix[] absoluteBoneTransforms, GameTime
gameTime)
{
    Matrix world;
    BoundingSphere sphere;

    foreach (ModelMesh mesh in modelo.model.Meshes)
    {
        world = absoluteBoneTransforms[mesh.ParentBone.Index]
* worldTransform;
        sphere = TransformBoundingSphere(mesh.BoundingSphere,
world);

        if (sphere.Intersects(ray) != null)
        {
            BoundingSphereRenderer.Render(sphere, gDevice,
camera.view, camera.projection, Color.Black);
            return true;
        }
    }
    return false;
}

```

El método `RayIntersectsModel` también es bastante descriptivo en cuanto a su función, que no es otra que ver si un rayo intersecciona con un modelo dado. Para ello recorre todas las mallas que tiene el modelo que le pasamos como parámetro. Por cada malla calcula su "mundo" y a continuación transforma su `BoundingSphere` acorde a ese mundo. Cuando hablamos de `BoundingSphere` nos referimos a la clase de dicho nombre, propia de XNA, que representa una esfera existente alrededor de cada malla de un objeto y que precisamente se usa para el cálculo de colisiones y reconocimiento de objetos. Además de la clase `BoundingSphere` existe la clase `BoundingBox`.

Por último, y una vez calculada la esfera, veremos si esa esfera intersecciona con el rayo con el método `Intersects` propio de XNA, en cuyo caso renderizaremos la `BoundingBoxSphere`.

Lo vemos en las siguientes figuras:

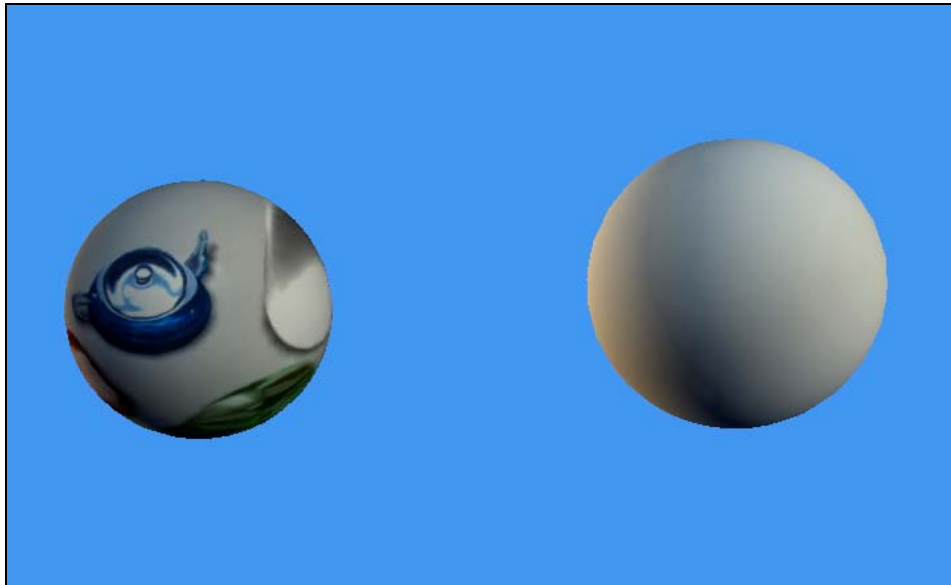


Figura 4.2.10 - Esferas sin que interseccione el rayo

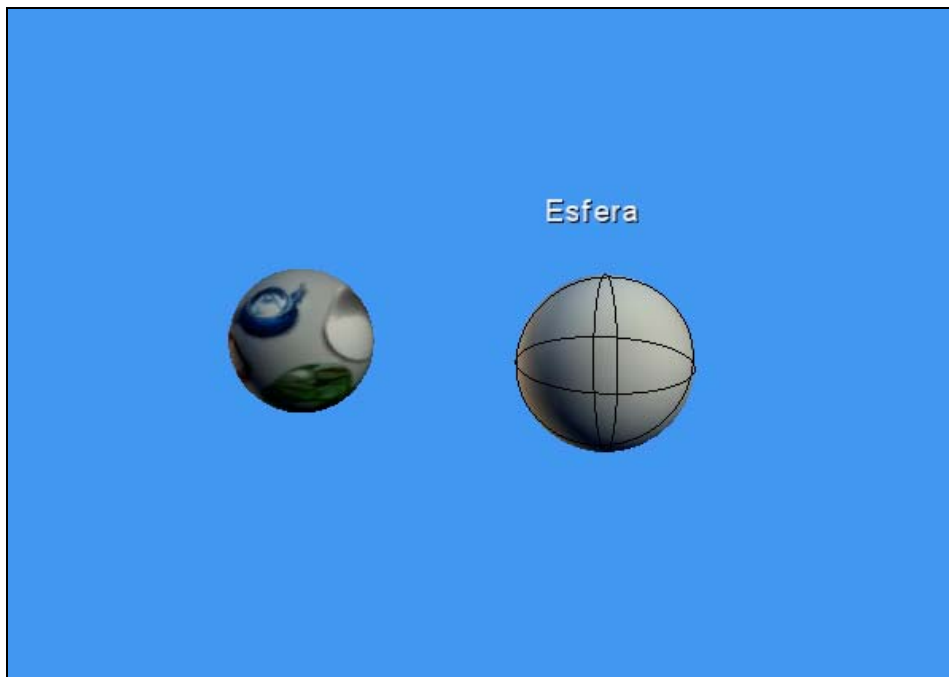


Figura 4.2.11 - Esferas tras la intersección del rayo con una de ellas

Ahora ya podemos reconocer sobre que objeto está nuestro ratón. El siguiente paso será mover el objeto. Para mover el objeto añadiremos el código que se muestra a continuación:

```

if (mouseState.LeftButton == ButtonState.Pressed)
{
    //Plano de intersección
    Plane p = new Plane(camera.mCameraPosition, 0);

    //Vectores ratón
    Vector2 ompos = new Vector2(prevMouseState.X,
prevMouseState.Y);
    Vector2 mpos = new Vector2(mouseState.X, mouseState.Y);

    Ray nmray = CalculateCursorRay(mpos, camera.projection,
camera.view);
    Ray omray = CalculateCursorRay(ompos, camera.projection,
camera.view);

    Vector3 opos = omray.Position + 20 * omray.Direction;
    Vector3 npos = nmray.Position + 20 * nmray.Direction;
    npos -= opos;
    rigidbodies.GetBody(i).Pos += npos;
    rigidbodies.GetBody(i).Vel = Vector3.Zero;
}

```

```

//Plano de intersección
Plane p = new Plane(camera.mCameraPosition, 0);

```

```
//Vectores ratón
Vector2 ompos = new Vector2(prevMouseState.X, prevMouseState.Y);
Vector2 mpos = new Vector2(mouseState.X, mouseState.Y);
```

En primer lugar vamos a crear un plano en la posición de la cámara. Después guardaremos la posición anterior del ratón y la actual.

```
Ray nmray = CalculateCursorRay(mpos, camera.projection,
camera.view);
Ray omray = CalculateCursorRay(ompos, camera.projection,
camera.view);
```

Con esas posiciones ya guardadas calculamos los rayos para ambas dos.

```
Vector3 opos = omray.Position + 20 * omray.Direction;
Vector3 npos = nmray.Position + 20 * nmray.Direction;
npos -= opos;
rigidbodies.GetBody(i).Pos += npos;
rigidbodies.GetBody(i).Vel = Vector3.Zero;
```

Lo siguiente que tenemos que hacer es calcular tanto la posición anterior como actual de los rayos. Para eso simplemente hacemos la suma vectorial de la posición del rayo y de la dirección del mismo multiplicada por el valor que deseemos (a más valor más rápido se moverá el objeto). Por último le restamos la posición vieja a la nueva y a la posición del objeto le restamos dicha posición nueva obtenida y esa será nuestra nueva posición. También vamos a hacer que la velocidad del objeto mientras lo tenemos sujeto sea cero.



Figura 4.2.12 - Posición inicial de la caja

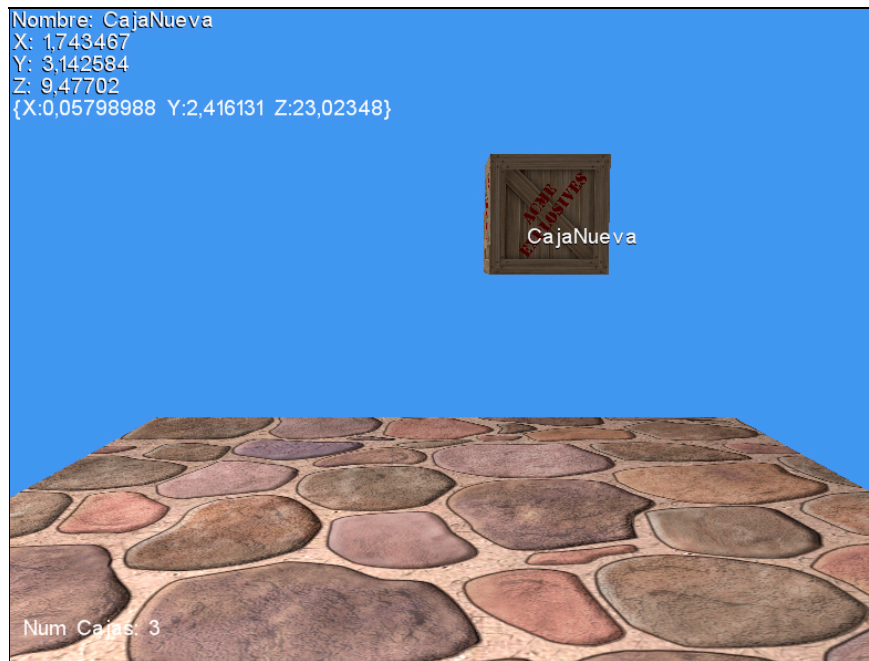


Figura 4.2.13 - Posición final de la caja

4.3 Simulando un mundo real

Con todos estos elementos ya seríamos muy capaces de crear un juego más o menos adictivo. Al fin y al cabo ya tenemos lo básico, carga de modelos, reconocimiento de objetos, selección de dichos objetos e interacción entre ellos. Sin embargo, el objetivo era tratar de hacer que nuestro entorno 3D fuera lo más realista posible, y para ello necesitamos algo vital... **la física**.

4.3.1 Introducción a la física en 3D

El concepto de física en un mundo 3D es algo muy sencillo de imaginar. Al fin y al cabo, de lo que se trata cuando creamos una física realista en un mundo tridimensional, es simple y llanamente de simular lo más fielmente posible el mundo real, nuestro mundo.

Claro que lo que dicho así parece muy sencillo, en la práctica es muy complejo. Si pensamos en la cantidad de interacciones físicas que se producen en nuestro día a día nos daremos cuenta de lo complejo que puede ser simularlas.

El ejemplo más sencillo es la gravedad. En la tierra la gravedad es de $9,81 \text{ m/s}^2$ lo cual traducido a un mundo de tres dimensiones equivaldría a decir que todos los objetos de dicho mundo se ven afectados en el eje Y (generalmente) por dicha

velocidad. Pero claro, esto sin tener en cuenta el rozamiento del aire, la variabilidad del viento, etc. Vemos que la cosa se complica.

Otra de las teorías que se implementan a menudo en los juegos es la teoría de cuerpos rígidos. Esta teoría se encarga de calcular el comportamiento de los cuerpos ante un choque, ya sea con un objeto estático o con un objeto dinámico. Es decir, trata de averiguar qué le pasa a un objeto cuando choca con otros objetos del espacio (velocidad resultante, dirección, velocidad rotacional, etc.).

Algo a tener muy en cuenta cuando abordamos la física de un juego es la complejidad computacional que conlleva. Calcular la física no es "gratis" y menos a partir de cierto número de polígonos.

4.3.2 Motores de física

Existe, dentro de XNA, una gran variedad de motores físicos. Los hay de todo tipo: 2D, 3D, gratuitos, de código libre, de pago, etc. Aquí vemos algunos:

2D

- Farseer Physics
- Jello Physics
- Box2Dx

3D

- JigLibX
- BulletX
- BEPUphysics
- Matali Physics
- DigitalRune Physics

4.3.3 Alternativas utilizadas e implementación final

A la hora de implementar la física en nuestra aplicación 3D se plantearon varias posibilidades.

4.3.3.1 Implementación propia

La primera idea consistía en implementar la física por mi cuenta. El primer paso en cualquier implementación física, sin duda, es la gravedad. Para ello se utilizaron las fórmulas básicas de la gravedad.

El objeto se aceleraba según la gravedad pero, aquí cabe plantearse qué pasaría cuando llegara al suelo, es decir, cómo rebotaría y en qué dirección lo haría. Sin duda, con la teoría de la gravedad no nos valdría. Tendríamos que implementar *algo* más, y ese *algo* más era la teoría de cuerpos rígidos.

Llegados a este punto, se llegó a una conclusión lógica, la implementación de una física realista implicaría un tiempo de desarrollo mucho mayor del estimado, y por tanto quedaría fuera del alcance del proyecto. Por ello, se estudiaron otras posibilidades para la implementación de dicha lógica.

4.3.3.2 Primera aproximación realista

Como hemos visto la implementación de la física, especialmente la física de cuerpos rígidos, implica mucho tiempo de programación, que se queda fuera del alcance del proyecto. Por ello se investigó una implementación ya desarrollada en C++ y C# y basada en el trabajo *Nonconvex Rigid Bodies with Stacking* por Eran Guendelman, Robert Bridson y Ronald Fedkiw que simulaba dicha física de cuerpos rígidos. Se incorporó al proyecto con resultados que analizaremos. Veamos en qué consiste.

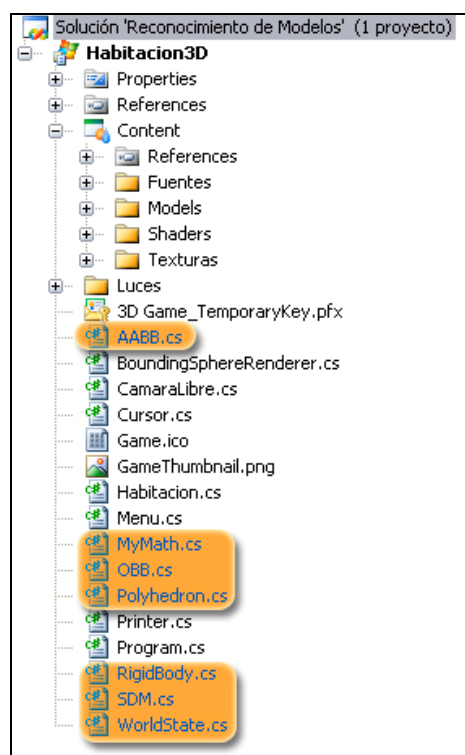


Figura 4.3.1 - Clases que implementan la física

Lo primero que vemos es que al proyecto creado se añaden siete clases que implementarán nuestra física. Dentro de estas clases son tres las que más importancia tienen:

Polyhedron.cs

Esta clase va a ser la encargada de "averiguar" qué tipo de objeto le estamos pasando. Es decir, su número de aristas, vértices, caras, centro de masa, tensor de inercia, etc.

RigidBody.cs

Esta clase contendrá toda la información del poliedro obtenido. Es decir, nos dirá si el objeto está parado, qué velocidad lineal tiene, su velocidad rotacional, etc.

WorldState.cs

Como su nombre indica, la clase `WorldState` será la encargada de "gestionar" los `RigidBodys` que tengamos en nuestra aplicación. Aquí tendremos una lista de `RigidBodys` a la cual podremos añadir nuevos objetos o eliminarlos (según queramos que se les aplique la física o no). Además definiremos todas las propiedades de este "motor" físico, como pueden ser número de colisiones, frontera de los objetos, etc.

El funcionamiento, a grandes rasgos, de este pequeño motor físico es el siguiente:

1. Se cargan los modelos:

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(graphics.GraphicsDevice);
    spriteBatchFont = Content.Load<SpriteFont>(@"Fuentes\Arial 14");

    modelBall = Content.Load<Model>(@"Models\ball");
    modelBox = Content.Load<Model>(@"Models\box");
    ...

    crateTexture = Content.Load<Texture2D>(@"Texturas\crate");
    ...

    if (escena == 1)
    {
        PyramideScene();
    }
    else
    {
        PruebaScene();
    }
    worldState = new WorldState(rigidbodies, 2, 2, true, true,
true);
    ...
}
```

En esta carga vemos que primero que cargan los modelos y luego se crea una variable de tipo `WorldState` a la cual le pasamos la lista de `RigidBodys` que creamos de la siguiente forma:

```
public void PruebaScene()
{
    platformType = new RigidBodyType("PlatformType",
@"..\..\models\platform.x", modelPlatform);
    boxType = new RigidBodyType("BoxType",
@"..\..\models\box.x", modelBox);
    smallboxType = new RigidBodyType("SmallBoxType",
@"..\..\models\small_box.x", modelSmallbox);
    modelBallType = new RigidBodyType("modelBall",
@"..\..\models\ball.x", modelBall);

    rigidbodies.AddBody(new RigidBody(
        "Suelo", //name
        platformType, //type
        new Vector3(0, -5, 0), //pos
        new Vector3(0, 0, 0), //vel
        new Quaternion(0, 0, 0, 1), //orientation
        new Vector3(0, 0, 0), // rotation
        .2f, //bounciness
        .5f, //friction
        float.PositiveInfinity //density
    ));

    rigidbodies.AddBody(new RigidBody(
        "Bola",
        modelBallType,
        new Vector3(0, 40, 0),
        new Vector3(0, 0, 0),
        new Quaternion(0, 0, 0, 1),
        new Vector3(0, 0, 0),
        .2f,
        .5f,
        1.0f
    ));
}
```

Aquí creamos primero el tipo `RigidBodyType` de cada modelo y luego añadimos a la lista de `RigidBodys` (`rigidbodies`) los modelos que queramos. En este caso la plataforma y una pelota. Podemos verlo en la siguiente figura:

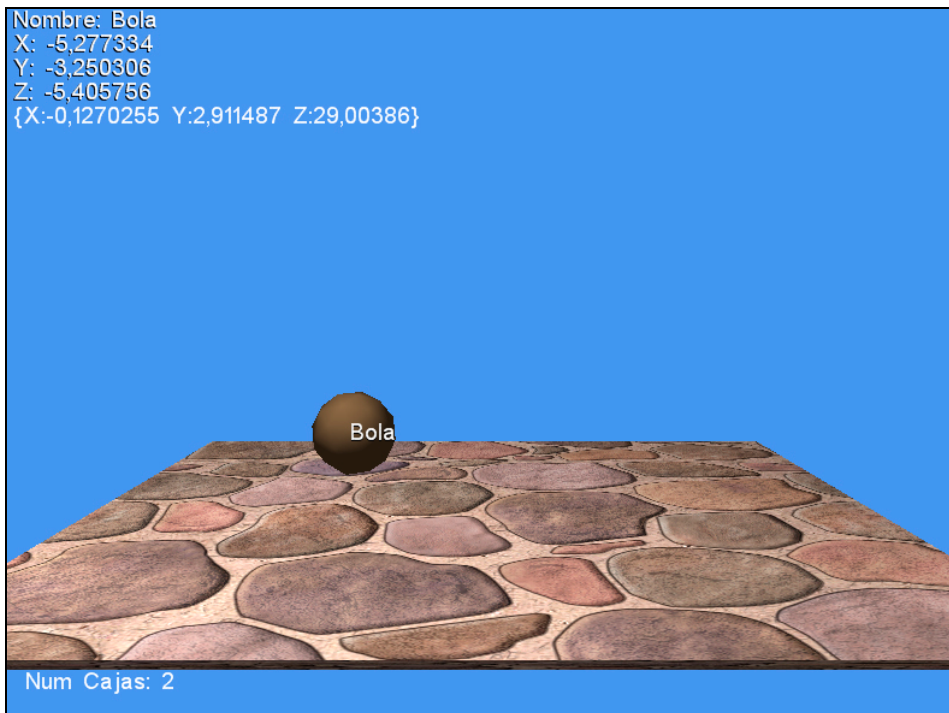


Figura 4.3.2 - Modelos cargados

A la hora de crear los objetos se definirán varias propiedades iniciales:

- Nombre: Nombre del objeto.
- Tipo: Tipo de objeto, que previamente hemos cargado.
- Posición: Posición inicial del objeto en el mundo 3D.
- Velocidad: Velocidad lineal inicial.
- Orientación: Orientación inicial .
- Rotación: Rotación inicial.
- Rebote: Definirá la cantidad de capacidad de rebote que tendrá el objeto.
- Fricción: Definirá la cantidad de fricción que ejercerá el objeto sobre otros.
- Densidad: Densidad del objeto. Como vemos a la plataforma le damos una densidad infinita (`float.PositiveInfinity`). Con esto conseguimos que la plataforma no se vea afectada por la gravedad. Si no fuera así la plataforma caería al vacío como el resto de objetos.

2. Se calculan las propiedades del objeto:

```
public RigidbodyType(string name, string meshFileName, Model  
model)  
{  
    this.name = name;  
    this.meshFileName = meshFileName;
```

```

    this.model = model;
    mesh = model.Meshes[0];

    polyhedron = new Polyhedron(mesh, name);

    Vector3 centerOfMass;
    polyhedron.ComputeMassProperties(1.0f, out volume, out
centerOfMass, out inertiaTensor);

    ...

    //sdm = new BruteForceSDM(polyhedron);
    //sdm = new GridSDM(polyhedron, 0.1f);
    sdm = new FasterSDM(polyhedron, 0.1f);
    ...
}

```

En primer lugar obtenemos el poliedro resultante de nuestro modelo

```

public Polyhedron(ModelMesh mesh, String name)
{
    if (true)
    {
        int NumberVertices = mesh.VertexBuffer.SizeInBytes /
mesh.MeshParts[0].VertexStride;
        int NumberFaces =
mesh.MeshParts[0].PrimitiveCount;

        ...
        verts = new ArrayList();
        for (int i = 0; i < NumberVertices; i++)
        {
            Vertex v = new Vertex();
            ...
            verts.Add(v);
        }
        faces = new ArrayList();
        for (int i = 0; i < NumberFaces; i++)
        {
            Face f = new Face();
            ...
            faces.Add(f);
        }
    }

    Initialize();
}

```

Con esto obtenemos los vértices ([Vertex](#)) y caras ([Face](#)) de nuestro modelo.

Una vez tenemos los vértices y las caras podemos computar las propiedades del objeto con la llamada al método `ComputeMassProperties(1.0f, out volume, out centerOfMass, out inertiaTensor)`, que vemos que nos devuelve el volumen, el centro de masa y el tensor de inercia. Estos elementos son indispensables para saber como se comportará cada objeto ante la fuerza de la gravedad y ante un choque con otros objetos.

3. Actualizar nuestro "mundo":

```

if (isRunning)
{
    int num_steps = 1;
    for (int i = 0; i < num_steps; i++)
    {
        worldState.Update(1.0f * elapsed / num_steps);
    }
}

```

Lo primero que observamos es que existe la variable `isRunning` que cuando sea verdadera hará que actúe la física.

```

public WorldState(myRigidBodyList bodies, int
num_collision_steps, int num_contact_steps, bool shock_propagation,
bool edge_intersection, bool collisions_before_velocity_update)
{
    this.bodies = bodies;
    this.num_collision_steps = num_collision_steps;
    this.num_contact_steps = num_contact_steps;
    this.shock_propagation = shock_propagation;
    this.edge_intersection = edge_intersection;
    this.collisions_before_velocity_update =
collisions_before_velocity_update;

    const int MaxBodies = 100;
    ...
    this.bodyPair = new RigidBody[2];
    this.collisionPairPoolCount = 0;
    this.collisionPairPool = new
CollisionPair[MaxCollisionPairPoolCount];
    for (int i = 0; i < collisionPairPool.Length; i++)
        collisionPairPool[i] = new CollisionPair();

    contactQueue = new Queue();
    bodiesAtLevel = new ArrayList[MaxContactLevels];
    collisionsAtLevel = new ArrayList[MaxContactLevels];
    for (int i = 0; i < MaxContactLevels; i++)
    {
        bodiesAtLevel[i] = new ArrayList();
        collisionsAtLevel[i] = new ArrayList();
    }
}

```

Vemos que en el constructor de la clase `WorldState` se definen una serie de variables que afectarán a la física y al rendimiento según usemos unos u otros valores.

```

public void Update(float dt)
{
    if (dt == 0 || bodies == null) return;
    for (int i = 0; i < bodies.GetNumBodies(); i++)
    {
        bodies.GetBody(i).RebuildMatrices();
    }

    if (collisions_before_velocity_update)
    {

```

```
        ProcessAllCollisions(dt);
    }

    for (int i = 0; i < bodies.GetNumBodies(); i++)
    {
        bodies.GetBody(i).UpdateVel(dt);
    }

    if (!collisions_before_velocity_update)
ProcessAllCollisions(dt);
        ProcessAllContacts(dt);

    for (int i = 0; i < bodies.GetNumBodies(); i++)
    {
        bodies.GetBody(i).UpdatePos(dt);
    }
}
```

Por último la llamada al método `Update` hará que se revisen todas las colisiones y que en función de ellas se actualice la velocidad y la posición de los objetos que contenga el array de `RigidBody`s (en este caso `bodies` del tipo `myRigidBodyList`).

Sin duda tener un array facilita mucho las cosas ya que nos permite añadir objetos a los que la física afecta, o incluso, si queremos, eliminar objetos a los que la física no afectará más.

Para añadir un objeto, es tan sencillo como hacer lo siguiente:

```
if (keyboardState.IsKeyDown(Keys.Z))
{
    if (bodyToAdd == null)
        bodyToAdd = new RigidBody(
            "BoxThrow",
            smallboxType,
            camera.mCameraPosition,
            camera.cameraDirection * 80.0f,
            new Quaternion(0, 0, 0, 0),
            new Vector3(0, 0, 0),
            .2f,
            .5f,
            1.0f
        );
}
```

En este caso si pulsamos la tecla `Z` una caja saldrá volando desde la posición de la cámara en la dirección de su vector dirección y con una velocidad resultante de multiplicar dicho vector dirección por 80. En las siguientes figuras vemos el ejemplo.

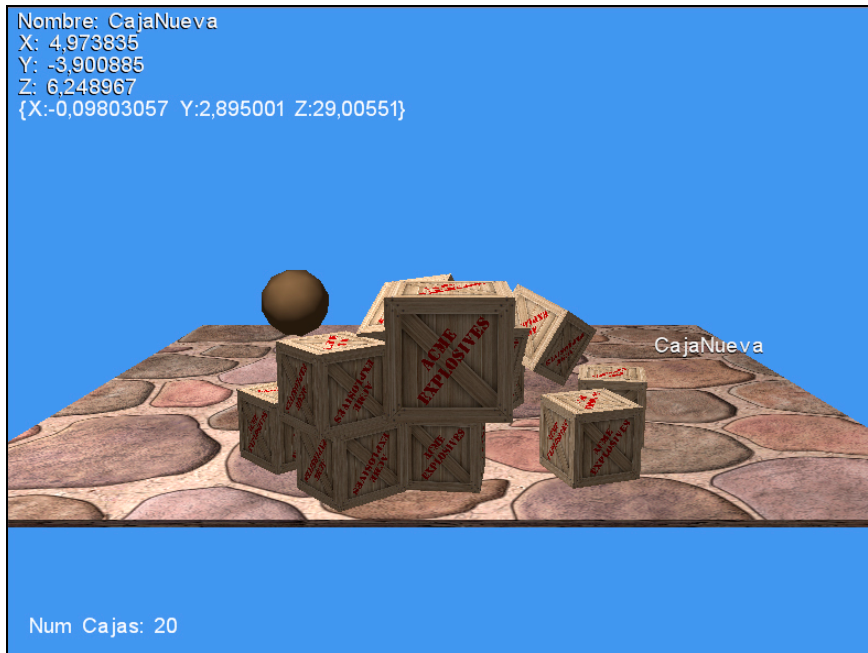


Figura 4.3.3 - Se lanza la caja

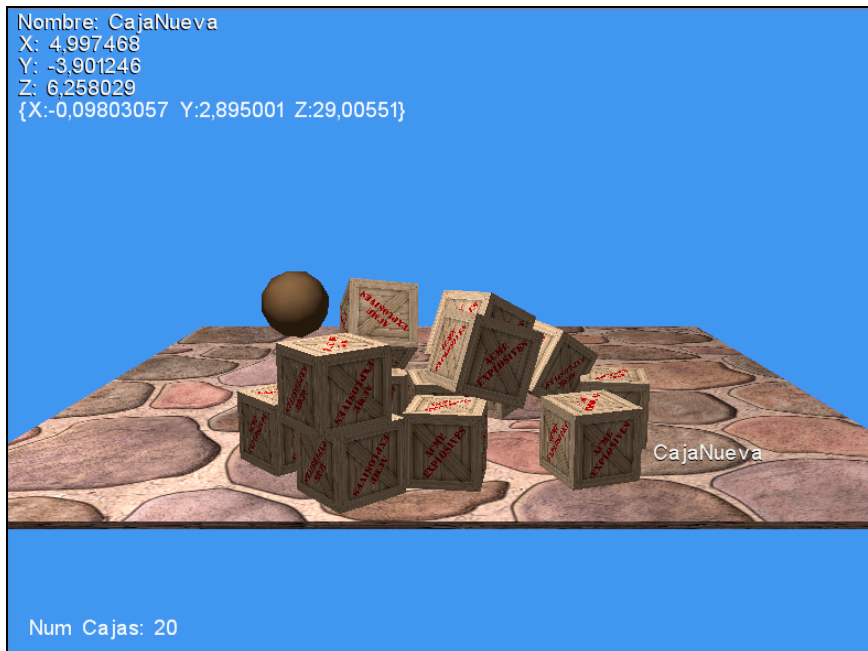


Figura 4.3.4 - La caja choca con el resto

Otra forma de cargar modelos es a través de un menú (clase `Printer`) implementado a partir de imágenes y la interacción que estas hacen con el ratón. XNA nos permite detectar colisiones en dos dimensiones y por tanto podemos ver en qué parte del menú se ha hecho clic. En las siguientes figuras vemos la carga de dos modelos a través de este menú:

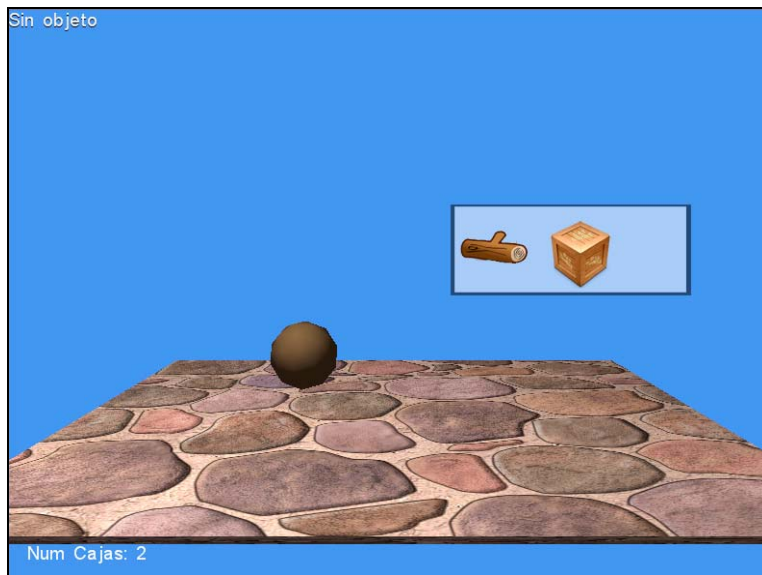


Figura 4.3.5 - Menú desplegado

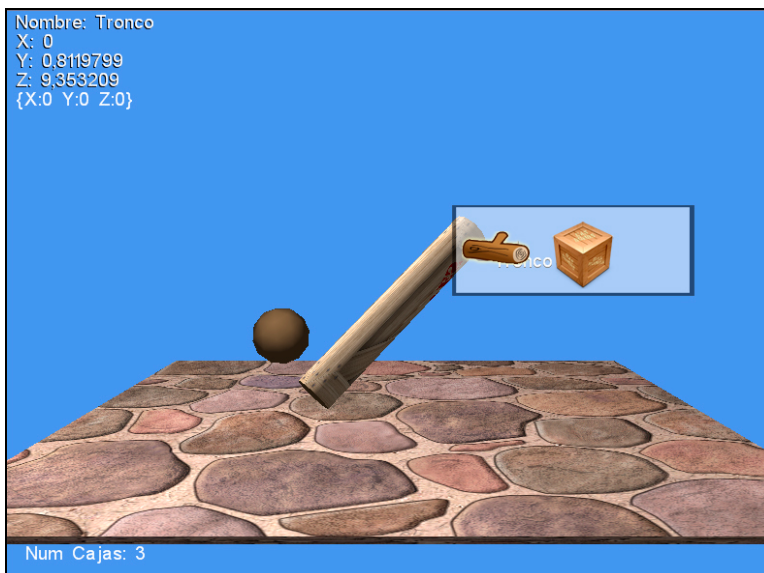


Figura 4.3.6 - Se carga un tronco

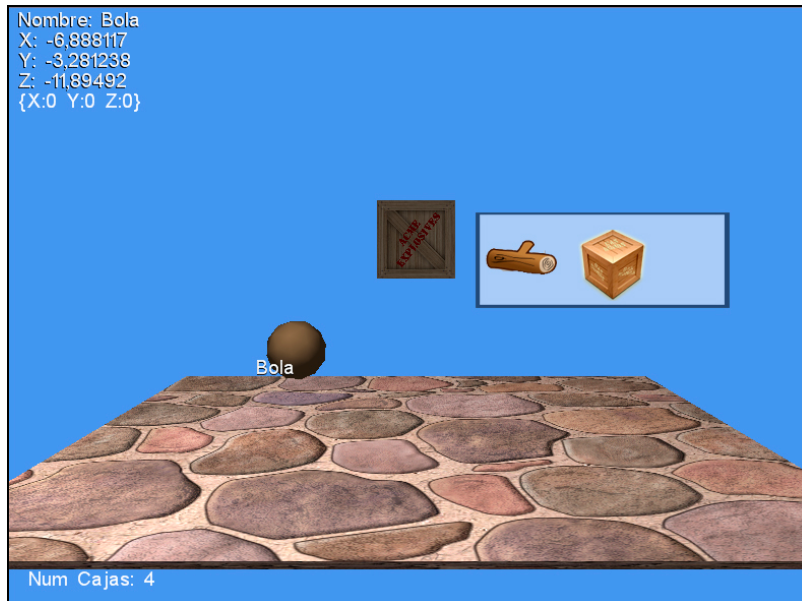


Figura 4.3.7 - Se carga una caja

Con esta implementación se simularía una física bastante realista. Sin embargo existen una serie de problemas:

- Límite de polígonos: El motor no permite una cantidad elevada de polígonos.
- Colisiones imperfectas: Se observan colisiones poco realistas entre objetos.
- Objetos traspasados: En ocasiones los objetos se traspasan antes de aplicarse la colisión que debería aparecer.
- Velocidades lineales: El motor es poco permisivo con las velocidades lineales altas. Si el objeto tiene demasiada velocidad traspasa los objetos.

Por esto, para la aplicación final se optó por usar un motor físico más depurado y testeado. En este caso se eligió el motor gratuito (y posteriormente de código libre) BEPU Physics.

4.3.3.3 Motor físico BEPU Physics

En el manejo motor físico BEPU Physics debemos destacar la existencia de dos variables:

Space.cs

La clase `Space` es la encargada de almacenar todos los objetos que tendrá nuestra escena. Sería el equivalente al array de objetos que hemos definido en la prueba anterior. Además podemos definir propiedades de nuestra física; como por ejemplo la gravedad, precisión de las colisiones, etc.

Entity.cs

La clase `Entity` es la equivalente a nuestra clase `RigidBody` del ejemplo anterior. Contendrá las propiedades (masa, volumen, etc.) del modelo que carguemos.

Como vemos el funcionamiento del motor físico BEPU Physics es similar a la aproximación implementada anteriormente. Sin embargo, este es un motor muy depurado, lo que le da una eficiencia muy grande. Además está en continua mejora por parte de su creador.

Veamos cual ha sido el modo de proceder para utilizar este motor.

```
using BEPUphysics;  
using BEPUphysics.Entities;  
using BEPUphysics.DataStructures;  
using BEPUphysics.ResourceManagement;  
using BEPUphysics.Constraints;
```

Lo primero que hay que hacer, evidentemente, es cargar la librería del motor.

```
Space space = new Space();  
List<Actor> WorldObjects = new List<Actor>();
```

A continuación declaramos la lista de actores de nuestro mundo. La clase `Actor` simplemente contendrá la entidad que le pasemos por parámetro junto a un valor de escala. Además creará una variable necesaria para la iluminación que más adelante veremos. También declaramos la variable `space`.

```
space.SimulationSettings.MotionUpdate.Gravity = new Vector3(0, -  
28f, 0);
```


A continuación asignamos la gravedad.

```
// Añadimos multithreading
if (Environment.ProcessorCount > 1)
{
    for (int i = 0; i < Environment.ProcessorCount; i++)
    {
        space.ThreadManager.AddThread();
    }
    space.UseMultithreadedUpdate = true;
}

// Número de iteraciones en la simulación física (A menor
// número más rendimiento pero se pierde precisión en el cálculo físico)
space.SimulationSettings.CollisionResponse.Iterations = 4;
```

El siguiente paso es habilitar el multithreading, y es que si el número de procesadores es mayor que uno crearemos un hilo de ejecución por cada procesador. Además asignaremos el número de iteraciones en las colisiones.

```
box = Content.Load<Model>("Models/box");
caja = Content.Load<Model>("Models/box2");
sphere = Content.Load<Model>("Models/balon");

//Añadimos la metrallera
metralleraEntity = new Box(camara.mCameraPosition, 1,1,1);
metralleraActor = new Actor(metrallera, metralleraEntity,
Vector3.One);
WorldObjects.Add(metralleraActor);

//Añadimos todos los objetos que hemos creado
(WorldObjects) al ObjectManager(SunBurn) y al Space(BEPU)
for (int i = 0; i < WorldObjects.Count; i++)
{
    //Lo añadimos al motor físico BEPU
    space.Add(WorldObjects[i].Entity);
}
```

Una vez habilitado cargamos los modelos. Después de cargarlos tenemos que crear la forma geométrica que más se amolde a nuestro modelo. En este caso creamos una caja simple de dimensiones 1x1x1 y con esta caja ya podemos crear nuestro Actor y añadirlo a la lista de Actores (WorldObjects) creada. A continuación se recorre esa lista y se van añadiendo los objetos a la variable space.

```
for (int i = 0; i < WorldObjects.Count; i++)
{
    WorldObjects[i].Update();
}
```

Por último, en el método update, actualizamos nuestra lista de objetos.

Con este flujo de código estamos cargando los objetos dinámicos de nuestra escena, pero... ¿y si queremos cargar un objeto estático?

Para ello utilizamos el siguiente código. Lo que hace esta serie de métodos es calcular los vértices, aristas, etc. del modelo en cuestión y crear la malla a partir de dicha información.

```
BEPUpysics.StaticTriangleGroup.StaticTriangleGroupVertex[]
verticesPista;
int[] indicesPista;

//Obtenemos la información del modelo (vértices e índices)
StaticTriangleGroup.GetVerticesAndIndicesFromModel(pistaCompleta
, out verticesPista, out indicesPista);

//Creamos la malla
TriangleMesh triMeshPista = new TriangleMesh(verticesPista,
indicesPista);

//Transformamos la malla en una StaticTriangleGroup para poder
añadirla al espacio

StaticTriangleGroup staticTriGroupPista = new
StaticTriangleGroup(triMeshPista);

//Escalamos la malla según el tamaño de nuestro mundo
staticTriGroupPista.WorldMatrix =
Matrix.CreateScale(worldScale);
space.Add(staticTriGroupPista);
```

Como vemos, es muy sencillo utilizar este motor físico y la potencia que nos brinda es realmente asombrosa. Como más adelante observaremos, en la parte de pruebas de rendimiento, cuanto mayor son los cálculos mayor es la pérdida de rendimiento; pero hay que tener muy claro que el cálculo en **tiempo real** de la física de cuerpos rígidos es muy compleja y requiere mucha capacidad de cálculo.

Otro aspecto a tener muy en cuenta en la simulación de un entorno realista era **la iluminación**.

4.3.4 Introducción a la iluminación en 3D

En la iluminación existen varios tipos de luces. Antes de adentrarnos en el motor utilizado para iluminar la escena vamos a ver las tres más importantes y como se definen.

Luz ambiental

La luz ambiental es una luz básica que no tiene dirección. La formula típica es:

$$L = \text{Intensidad} \times \text{Color}$$

Donde L es la luz, *Intensidad* es la intensidad de la luz (entre 0 y 1) y *Color* es el color de la luz ambiental. Este color puede ser un valor fijo, un parámetro o una textura.

Cuando un objeto como el de la figura está iluminado solo por luz ambiental se ve en 2 dimensiones ya que no tenemos sensación de volumen.



Figura 4.3.8 - Esfera iluminada solo por luz ambiental

Luz difusa

La luz difusa sigue la siguiente fórmula:

$$L = \text{Intensidad} \times \text{Color} + \text{IntensidadDifusa} \times \text{ColorDifusa} \times N.L$$

Como podemos ver seguimos utilizando el algoritmo de iluminación ambiental pero tenemos dos nuevas variables que indican la intensidad y el color de la luz difusa y dos vectores (N y L) que describirán la dirección de la luz L comparada con la normal de la superficie N.

Dicho así puede sonar un poco raro pero pensemos que la iluminación difusa es un valor que nos indica cuanto refleja la luz una superficie. Lógicamente la luz reflejada será mayor y más visible cuanto menor sea el ángulo entre la normal N y la dirección de la luz L como vemos en la figura.

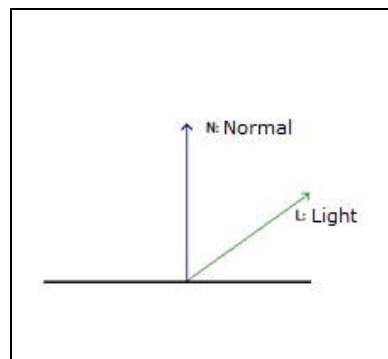


Figura 4.3.9 - Luz difusa

Si L está paralelo a N la luz se reflejará al máximo, si está paralelo a la superficie se reflejará al mínimo. Por tanto debemos calcular ese ángulo entre L y N. Para utilizaremos el producto escalar o producto punto:

$$N.L = |N| \times |L| \times \cos(\alpha)$$

Donde $|N|$ es la magnitud del vector N , lo mismo para L y $\cos(\alpha)$ es el ángulo entre los vectores.

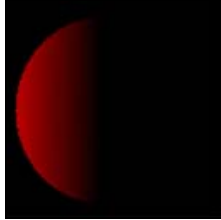


Figura 4.3.10 - Esfera iluminada solo por luz difusa

Si combinamos ambas luces tendremos un efecto más realista de nuestra esfera 3D.

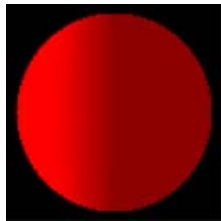


Figura 4.3.11 - Esfera con iluminación difusa y ambiental

Luz especular

La luz especular nos servirá para simular como de reflectante es un objeto. La fórmula de dicha luz es:

$$L = \text{Intensidad} \times \text{Color} + \text{IntensidadDifusa} \times \text{ColorDifusa} \times N \cdot L + \text{EspecularIntensidad} \times \text{EspecularColor} \times (R \cdot V)^n$$

Donde

$$R = 2 \times (N \cdot L) \times N - L$$

Y donde V representa el llamado vector "vista" que es simplemente el vector que va desde la posición de la cámara hasta el target de dicha cámara.



Figura 4.3.12 - Esfera iluminada por luz difusa, ambiental y especular

4.3.4.1 Motor de iluminación SunBurn

Al igual que en la física, y tras investigar soluciones propias, se decidió utilizar un motor de iluminación avanzado que nos diera una calidad superior en los resultados, tanto visualmente como en rendimiento.

Para ello, se optó por el motor de iluminación SunBurn de SynapseGaming. Se implementó la solución utilizando el framework 1.3.2.8, aunque para la creación de escenarios se utilizó el editor que trae la versión Indie de dicho motor de iluminación. En la siguiente figura podemos ver un ejemplo del editor gráfico que nos permite tanto añadir objetos como definir puntos de luz, etc.

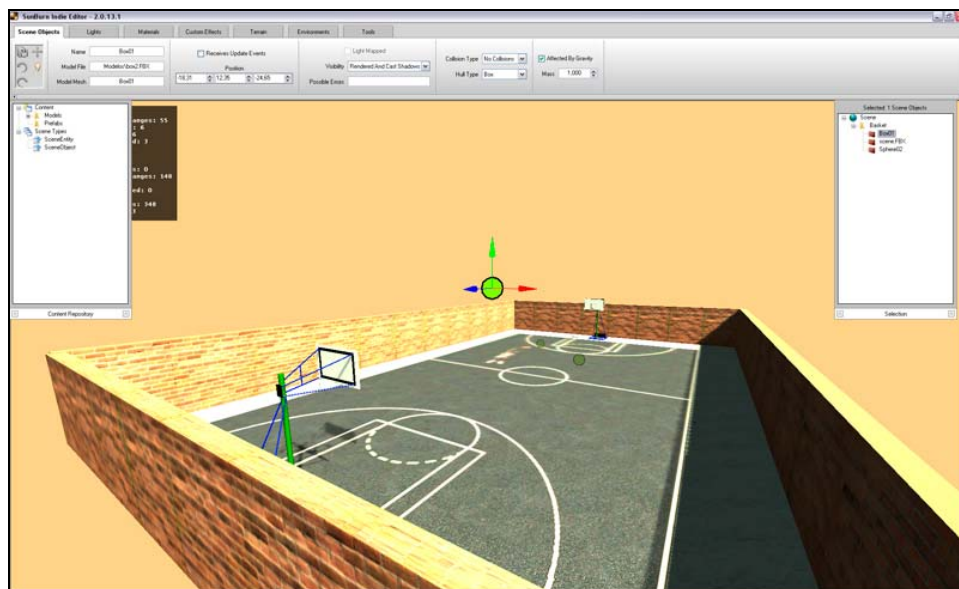


Figura 4.3.13 - Editor SunBurn

Con este editor nos ahorramos mucho tiempo sobre todo en el posicionamiento de los objetos y también en la iluminación de la escena. Al fin y al cabo puede llegar a ser desesperante colocar todo lo que queremos en nuestra escena basándonos solo en puntos tridimensionales.

Este es el proceso que hay que seguir para usar la iluminación avanzada proporcionada por SunBurn:

1. Cargamos las librerías necesarias:

```
using SynapseGaming.LightingSystem.Core;  
using SynapseGaming.LightingSystem.Lights;  
using SynapseGaming.LightingSystem.Rendering;  
using SynapseGaming.LightingSystem.Shadows;  
using SynapseGaming.LightingSystem.Processors;
```

2. Declaramos las variables necesarias:

```
LightingSystemManager lightingSystemManager;  
SceneInterface sceneInterface;  
SceneState sceneState;  
SceneEnvironment environment;  
LightingSystemPreferences preferences;  
SplashScreenGameComponent splashScreenGameComponent;  
LightRig rig;
```

Las vamos a ir viendo a medida que aparezcan.

3. Definimos la SplashScreen (Pantalla de presentación):

```
splashScreenGameComponent = new SplashScreenGameComponent(this,  
graphics);  
Components.Add(splashScreenGameComponent);
```

4. Definimos el manager general de luces y la variable sceneState:

```
lightingSystemManager = new LightingSystemManager(Services);  
sceneState = new SceneState();
```

El manager general de luces es el encargado de manejar todos el sistema de luces.

Por otra parte tenemos la variable sceneState. Esta variable es la encargada de dibujar todo lo que esté entre su Begin() y su End(). Es muy similar a la clase [SpriteBatch](#). Luego veremos su uso.

5. Inicializamos la variable sceneInterface y definimos las preferencias a la hora de renderizar las luces, sombras, etc.

```
sceneInterface = new SceneInterface(graphics);  
sceneInterface.CreateDefaultManagers(false, false, true);
```

```

//Cargamos las preferencias de un archivo de configuración
o directamente por código
preferences = new LightingSystemPreferences();
#if !XBOX
    if (File.Exists(userPreferencesFile))
        preferences.LoadFromFile(userPreferencesFile);
    else
#endif
    {
        preferences.EffectDetail = DetailPreference.High;
        preferences.MaxAnisotropy = 4;
        preferences.PostProcessingDetail =
DetailPreference.High;
        preferences.ShadowDetail = DetailPreference.High;
        preferences.ShadowQuality = 6.0f;
        preferences.TextureQuality = DetailPreference.High;
        preferences.TextureSampling =
SamplingPreference.Trilinear;
    }

```

La clase `SceneInterface` actúa como un proveedor de servicios, que contiene todos los managers de la escena. También actúa como un contenedor de componentes, donde una llamada a los métodos de los managers (como `BeginFrameRendering`, `Unload`, etc.) es automáticamente propagada a todos los managers contenidos en él.

Este diseño permite a los managers se añadidos como componentes modulares, pudiendo ser fácilmente añadidos, eliminados o reemplazados con diferentes configuraciones o implementaciones. Esto nos da por tanto una modularidad muy interesante.

El siguiente paso es comprobar si existe un fichero de preferencias de usuario y si no es así definir las a mano. En este caso no tenemos dicho fichero así que los parámetros se definirán por código. Entre dichos parámetros podemos ver la calidad de los efectos, el detalle del postprocesado, la calidad y detalle de las sombras o de las texturas entre otras. Vemos que se está definiendo los parámetros a mucha calidad. Si deseamos mejorar el rendimiento no tenemos más que cambiarlos.

6. Cargamos cada objeto en el manager de luces:

```

for (int i = 0; i < WorldObjects.Count; i++)
{
    //Lo añadimos al motor de iluminación
    sceneInterface.ObjectManager.Submit
    (WorldObjects[i].SceneObject);
}

```

Cada modelo que tenemos vamos a añadirlo al manager de objetos (`ObjectManager`) contenido en el `sceneInterface`. Vemos que lo que cargamos no es el `Actor` en sí sino la clase `SceneObject` contenida en dicho modelo. Esta clase pertenece a la librería `SynapseGaming.LightingSystem.Rendering` y se crea en la propia clase `Actor` como vemos a continuación.

```

public Actor(Model model, Entity entity, Vector3 scale)
{

```

```
SceneObject = new SceneObject(model);
...
```

7. Cargamos la configuración de las luces a partir de un fichero XML:

```
rig = Content.Load<LightRig>("Escena/light_rig");
sceneInterface.LightManager.Submit(rig);
```

Así se estructura este fichero:

```
<root>
  <LightRig>
    <LightGroups>
      <GroupList>
        <item_0>
          <LightGroup>
            <Name>Escenario</Name>
            <ShadowType>None</ShadowType>
            <Position>
              <Vector3>
                <X>0</X>
                <Y>0</Y>
                <Z>0</Z>
              </Vector3>
            </Position>
            <Radius>0</Radius>
            <ShadowQuality>0.5</ShadowQuality>
            <ShadowPrimaryBias>1</ShadowPrimaryBias>
            <ShadowSecondaryBias>0.2</ShadowSecondaryBias>
            <ShadowPerSurfaceLOD>True</ShadowPerSurfaceLOD>
            <ShadowRenderLightsTogether>
              False
            </ShadowRenderLightsTogether>
            <ShadowGroup>False</ShadowGroup>
            <Lights>
              <LightList>
                <item_0>
                  <AmbientLight>
                    <Name>Ambient Lighting</Name>
                    <Enabled>True</Enabled>
                    <DiffuseColor>
                      <Vector3>
                        <X>0.8</X>
                        <Y>0.9803922</Y>
                        <Z>0.9882353</Z>
                      </Vector3>
                    </DiffuseColor>
                    <Intensity>0.2</Intensity>
                  </AmbientLight>
                </item_0>
              </LightList>
            </Lights>
            ...
          </LightGroup>
        </item_0>
      </GroupList>
    </LightGroups>
  </LightRig>
</root>
```


El XML define los grupos de luces y dentro de cada grupo tenemos subgrupos diferentes y cada uno de ellos contiene diferentes luces.

8. Cargamos la configuración del entorno y aplicamos las preferencias que hemos definido anteriormente:

```
environment = Content.Load<SceneEnvironment>
( "Escena/scene_environment " );

sceneInterface.ApplyPreferences( preferences );
```

El fichero scene_enviroment.env sigue una estructura similar al fichero de configuración de la iluminación.

```
<root>
  <SceneEnvironment>
    <VisibleDistance>288</VisibleDistance>
    <FogEnabled>False</FogEnabled>
    <FogColor>
      <Vector3>
        <X>0.7098039</X>
        <Y>0.8784314</Y>
        <Z>1</Z>
      </Vector3>
    </FogColor>
    <FogStartDistance>230</FogStartDistance>
    <FogEndDistance>250</FogEndDistance>
    <ShadowFadeStartDistance>200</ShadowFadeStartDistance>
    <ShadowFadeEndDistance>250</ShadowFadeEndDistance>
    <ShadowCasterDistance>250</ShadowCasterDistance>
    <BloomAmount>2.76</BloomAmount>
    <BloomThreshold>0.94</BloomThreshold>
    <DynamicRangeEnabled>True</DynamicRangeEnabled>
    <ExposureAmount>1.25</ExposureAmount>
    <DynamicRangeTransitionMaxScale>4.5
    </DynamicRangeTransitionMaxScale>
    <DynamicRangeTransitionMinScale>0.5
    </DynamicRangeTransitionMinScale>
    <DynamicRangeTransitionTime>0.5
    </DynamicRangeTransitionTime>
    <DynamicRangeSaturationAmount>1
    </DynamicRangeSaturationAmount>
    <DynamicRangeDarkenAmount>0.27</DynamicRangeDarkenAmount>
    <DynamicRangeCinematicAmount>0.3
    </DynamicRangeCinematicAmount>
    <Gravity>1</Gravity>
    <EditorIconScale>6</EditorIconScale>
    <EditorMoveScale>100</EditorMoveScale>
    <EditorRotationScale>4</EditorRotationScale>
    <EditorDefaultObjectScale>0.05</EditorDefaultObjectScale>
  </SceneEnvironment>
  <classes>
    ...
  </classes>
</root>
```

Aquí definiremos aspectos como el color de la niebla (fog), a qué distancia empieza, etc.

En las siguientes figuras podemos ver ejemplos de como queda la aplicación tras iluminar la escena con SunBurn:



Figura 4.3.14 - Iluminación SunBurn



Figura 4.3.15 - Iluminación SunBurn



Figura 4.3.16 - Iluminación SunBurn

Podemos observar la mejora respecto a la siguiente imagen que fue obtenida sin la iluminación avanzada.

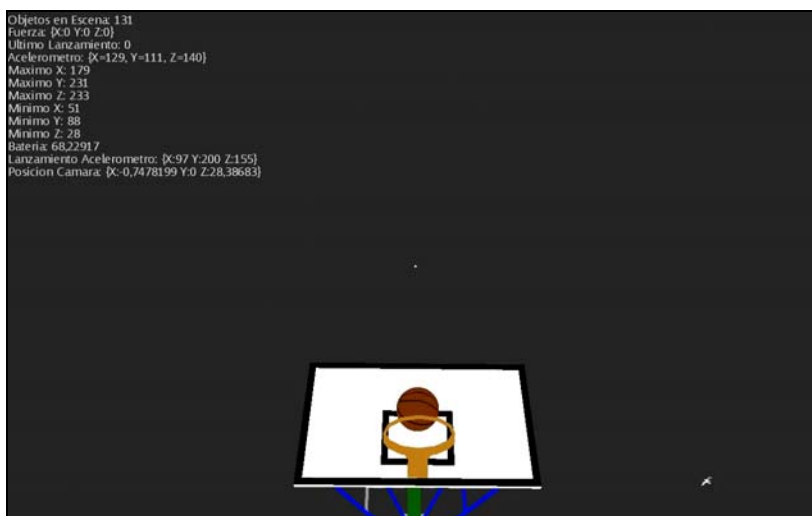


Figura 4.3.17 – Sin iluminación SunBurn

Sunburn nos permite cambiar la iluminación por código. Vemos un ejemplo:

```
foreach (LightGroup lightGroup in rig.LightGroups)
{
    foreach (ILight light in lightGroup.Lights)
    {
        if (light is DirectionalLight)
        {
            DirectionalLight dirLight=light as DirectionalLight;

            if (dirLight.Name.Equals("Foco") && oscuro)
            {
                dirLight.Intensity = 1f;
                dirLight.Enabled = true;
            }
        }
        rig.CommitChanges();
    }
}
```

La estructura como vemos sigue la forma del fichero con el *rig* de iluminación pudiendo elegir los grupos de luces, las luces individuales, etc. Luego podremos cambiar la intensidad, habilitar/deshabilitar las luces, etc. Por último llamamos al método `rig.CommitChanges()` para confirmar los cambios realizados.

4.3.5 SkyBox

Cuando nos movemos en un mundo tridimensional tenemos que tener en cuenta que el cielo se mueve a nuestro alrededor. Debemos tratar de simular un mundo que nos englobe y dado que modelar una escena muy grande supone un gasto grande de recursos la técnica más utilizada es la conocida como SkyBox. El Skybox no es más que, como su propio nombre indica, una caja que simula el cielo y el suelo. Esta caja contiene una textura de seis caras.



Figura 4.3.18 - Skybox

Esta textura se creará con el programa de nvidia nvdxt a partir de 6 imágenes. El resultado será un fichero con formate .dds.

El uso en programa será el siguiente:

```
public Skybox(string skyboxTexture, ContentManager Content)
{
    skyBox = Content.Load<Model>("SkyBox/SkyBoxModel");
    skyBoxTexture = Content.Load<TextureCube>(skyboxTexture);
    skyBoxEffect = Content.Load<Effect>("SkyBox/Skybox");
}
```

Vemos que los elementos que usamos son tres:

1. El modelo de la caja.
2. La textura generada anteriormente.
3. El efecto que actualizará la posición de la textura.

Para dibujar la escena se usará el siguiente método Draw:

```
public void Draw(Matrix view, Matrix projection, Vector3
cameraPosition)
{
    foreach (EffectPass pass in
        skyBoxEffect.CurrentTechnique.Passes)
    {
        foreach (ModelMesh mesh in skyBox.Meshes)
        {
            foreach (ModelMeshPart part in mesh.MeshParts)
            {
                part.Effect = skyBoxEffect;
                part.Effect.Parameters["World"].SetValue(
                    Matrix.CreateScale(size) *
                    Matrix.CreateTranslation(cameraPosition));
                part.Effect.Parameters["View"].SetValue(view);

                part.Effect.Parameters["Projection"].SetValue(projection);

                part.Effect.Parameters["SkyBoxTexture"].SetValue(skyBoxTexture);

                part.Effect.Parameters["CameraPosition"].SetValue(cameraPosition);
            }
            mesh.Draw();
        }
    }
}
```

El resultado sin ningún otro elemento en la escena es el siguiente:



Figura 4.3.19 - Skybox sin objetos en escena

Y así con la escena ya cargada:



Figura 4.3.20 - Skybox con objetos en escena

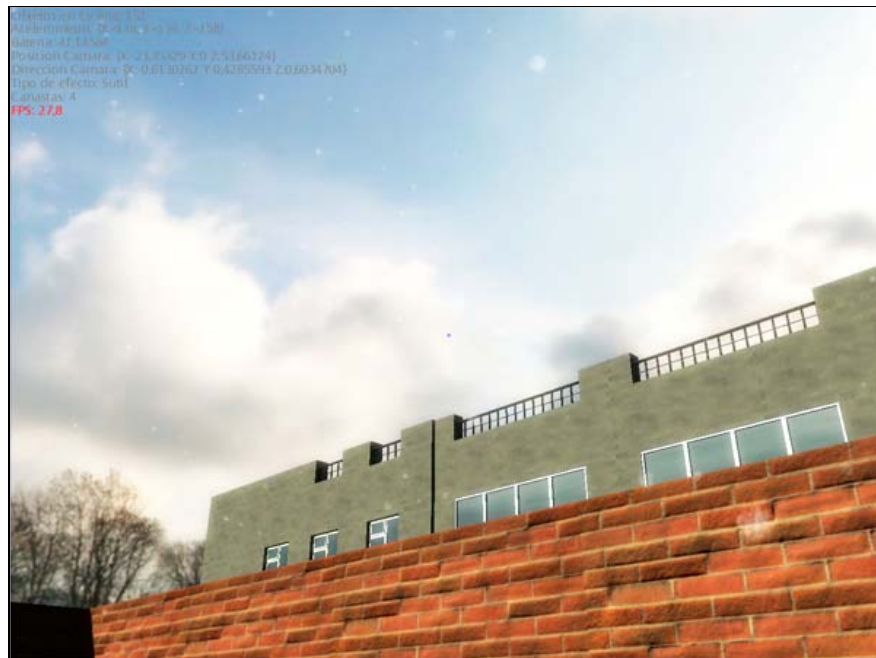


Figura 4.3.21 - Skybox con objetos en escena

Como vemos, hemos hablado de efectos, y es que estos efectos son clave en el llamado postprocesamiento que pasaremos a explicar.

4.3.6 Postprocesamiento

El postprocesamiento es una técnica de render utilizada especialmente en videojuegos para mejorar el aspecto visual de dicho videojuego. Lo que se pretende con estos efectos es conseguir un mayor realismo, o incluso el efecto contrario, simular una situación extrema o fuera de lo normal. Existen múltiples efectos que se pueden aplicar. Veamos algunos:

- **Lens Flare:** El Lens Flare es el efecto que trata de simular una luz puntual (principalmente solar) y los efectos de reflejos que tendría sobre una cámara. En la siguiente figura podemos ver un ejemplo:



Figura 4.3.22 – Lens Flare

El efecto no solo muestra la fuente de luz puntual, que en este caso es el sol, sino que trata de simular los efectos que tendría sobre los objetos que le rodean. Así vemos que existe un halo de luz difuminada en el monte que baña dicha luz solar. Además, como vemos en la siguiente figura, se genera una serie de efectos visuales alrededor del rayo central en forma de círculos.



Figura 4.3.23 – Lens flare

- **Viñeteo:** El viñeteo consiste en crear unas esquinas negras alrededor de la imagen. Con esto conseguimos que la visión se centre en lo que tenemos justo enfrente de la pantalla, obviando posibles objetos de alrededor.
- **Bloom:** El Bloom es un efecto que simula el comportamiento de las superficies de los objetos ante una iluminación muy fuerte. Este efecto no solo se usa para simular un mundo real sino que se puede exagerar para generar ciertas sensaciones (sueños, mundo irreal, etc.)

En la siguiente figura vemos un ejemplo de un efecto Bloom exagerado. Como vemos, en la imagen, también existe un viñeteo en las esquinas.



Figura 4.3.24 – Efecto Bloom

Vemos que ante un exceso de iluminación muy elevado las superficies brillan mucho.

- **Profundidad de campo (Depth Of Field ó DOF):** La profundidad de campo es un efecto que en nuestro día a día vemos constantemente pero nunca reparamos en ello. Cuando enfocamos la visión en un punto concreto automáticamente lo que le rodea se desenfoca. A mayor cercanía del objeto mayor es el desenfoque generado. En el mundo 3D este efecto está muy extendido ya que aporta un gran realismo (siempre que se use con medida) a la escena.

En la siguiente figura observamos una escena con profundidad de campo:



Figura 4.3.25 – Profundidad de campo

- **Desenfoque de movimiento (Motion Blur):** El desenfoque de movimiento es otro de esos efectos en los que apenas reparamos en nuestra vida diaria pero que siempre está presente. Si hacemos una prueba y giramos la cabeza de un lado a otro (como en una negación) veremos que lo que observamos se desenfoca y se difumina. Cuanto más rápido lo hagamos mayor será el efecto. Este efecto está muy extendido en juegos con mucho movimiento (shooters, plataformas 3D, etc.). Veamos un par de ejemplos:



Figura 4.3.26 – Desenfoque de movimiento

Como vemos, al hacer un giro se enfoca el área directamente alrededor del punto de mira del arma y el resto se desenfoca. El desenfoco que se aplica no es el mismo que en el DOF sino que se trata de un desenfoco de movimiento, que precisamente hace eso darnos la sensación de movimiento; proporcionando a los juegos un aspecto más vivo y de mayor acción y velocidad. En la figura 4.3.23 en la que veíamos el efecto Lens Flare, vemos también el desenfoco de movimiento que nos proporciona esa sensación de velocidad. Y aquí vemos otro ejemplo de desenfoco de movimiento en un juego de carreras:



Figura 4.3.27 – Desenfoco de movimiento

- **Desaturado/Saturado/Borroso:** Existen una serie de efectos que se aplican al color de la imagen. Podemos desde hacer que se vea todo borroso hasta saturar el color por encima de los niveles normales o desaturarlos. Aquí vemos algunos ejemplos:

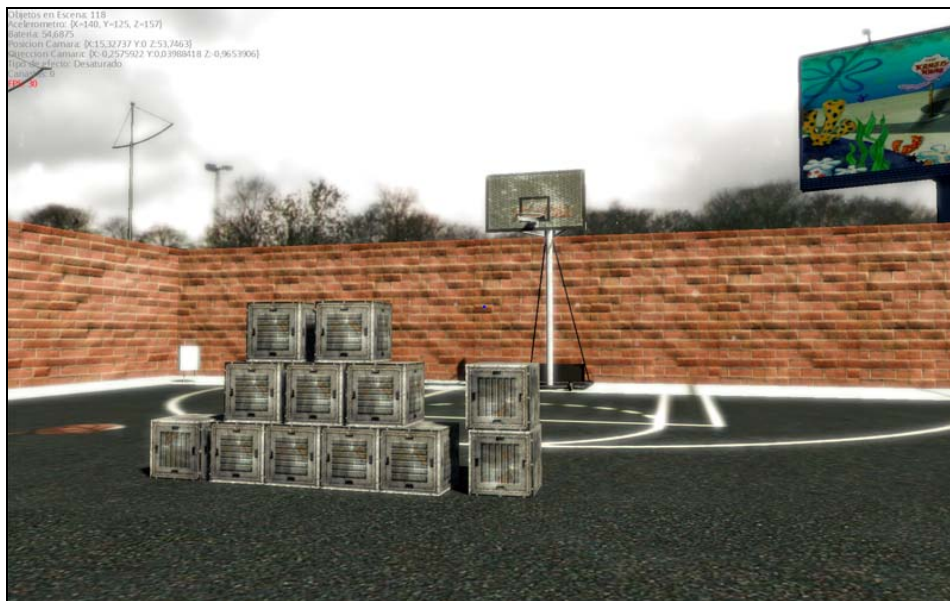


Figura 4.3.28 - Efecto desaturado

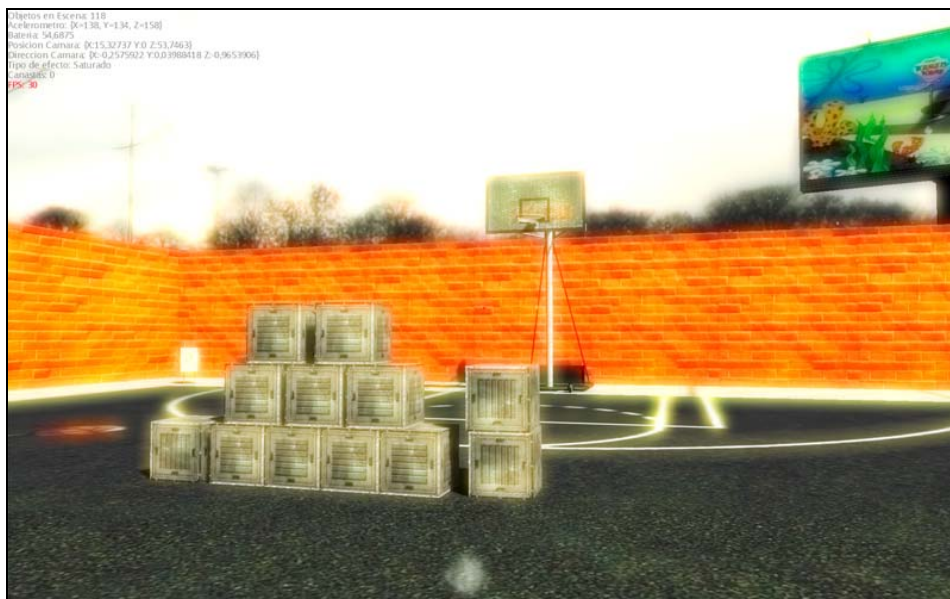


Figura 4.3.29 - Efecto saturado



Figura 4.3.30 - Efecto borroso

- **Oclusión Ambiente (Ambient Occlusion ó AO):** La oclusión ambiente se usa para dar un mayor realismo a las sombras de la escena. Generan una serie de sombras alrededor de los objetos. Veamos un ejemplo:

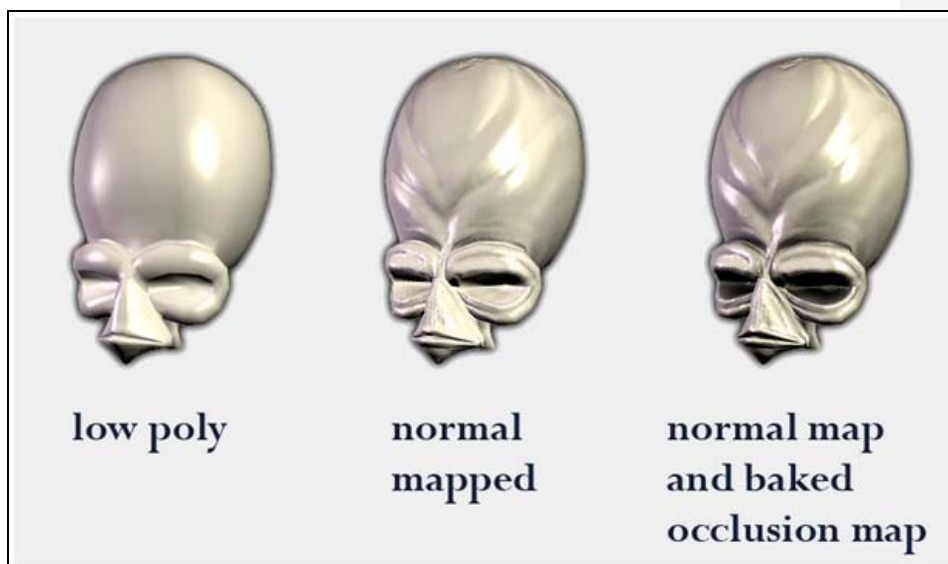


Figura 4.3.31 - AO

En primera instancia está el modelo en baja poligonización, el siguiente modelo muestra el modelo tras aplicarle el mapa de normales (que simula un modelo de muchos más polígonos sobre el mismo polígono de baja poligonización) y por último tenemos el modelo tras aplicar el Ambient Occlusion. Tras aplicar dicho efecto se generan las sombras entre los pliegues del objeto, lo que aporta un mayor realismo a dicho modelo.

Toda esta serie de efectos de postprocesado, además de muchos otros existentes, se pueden aplicar simultáneamente en una misma escena. Siempre tendremos que tener en cuenta que estos efectos suponen un aumento del cálculo y por tanto un rendimiento más bajo de la aplicación.

Una vez vistos los diferentes efectos de postprocesado que existen veremos cuales se han aplicado en nuestra aplicación y cómo gestiona XNA dichos efectos.

Como hemos explicado, el procesamiento es una técnica que simplemente genera un efecto sobre la salida normal. Es decir, que obtiene la imagen de salida y aplica el efecto deseado antes del render final.

Veamos un ejemplo de la implementación del efecto Bloom utilizado en nuestra aplicación:

Dentro de nuestra aplicación tenemos una clase llamada `BloomComponent`. Esta clase es la encargada de hacer el renderizado.

En primer lugar se cargan los efectos:

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    bloomExtractEffect =
    Game.Content.Load<Effect>( "Effects\\BloomExtract" );
    bloomCombineEffect =
    Game.Content.Load<Effect>( "Effects\\BloomCombine" );
    gaussianBlurEffect =
    Game.Content.Load<Effect>( "Effects\\GaussianBlur" );
}
```

Existen 3 efectos:

1. **BloomExtract**: Se encarga de obtener los píxeles más iluminados que serán en los que se aplique el efecto Bloom.
2. **BloomCombine**: Su labor es calcular la unión del efecto bloom calculado anteriormente con la imagen existente en la salida.
3. **GaussianBlur**: Por último este efecto se encarga de crear un desenfoque gaussiano.


```
PresentationParameters pp =  
GraphicsDevice.PresentationParameters;  
  
int width = pp.BackBufferWidth;  
int height = pp.BackBufferHeight;  
  
SurfaceFormat format = pp.BackBufferFormat;
```

A continuación se obtienen los `PresentationParameters` para conocer la anchura, altura y formato del `BackBuffer`.

```
resolveTarget = new ResolveTexture2D(GraphicsDevice, width,  
height, 1, format);
```

El siguiente paso va a ser crear una nueva textura a la cual pasaremos el valor del `BackBuffer`.

```
width /= 2;  
height /= 2;  
  
renderTarget1 = new RenderTarget2D(GraphicsDevice, width,  
height, 1, format);  
renderTarget2 = new RenderTarget2D(GraphicsDevice, width,  
height, 1, format);  
}
```

Por último crearemos dos `RenderTarget2D` para el procesamiento del efecto Bloom. Tendrán la mitad del tamaño que la salida original para minimizar los costes. Además, dado que vamos a desenfocar ligeramente la salida, no se perderá mucha calidad.

Una vez preparados los efectos, el siguiente paso es dibujarlos. Para ello en el método `Draw` tendremos el siguiente código:

```
public override void Draw(GameTime gameTime)  
{  
    GraphicsDevice.ResolveBackBuffer(resolveTarget);
```

En primer lugar vamos a generar (o resolver) la textura creada anteriormente con el contenido del `BackBuffer`. Podremos usarla así de entrada de nuestro postprocesado.

La siguiente parte se divide en 4 pasos:

```
bloomExtractEffect.Parameters["BloomThreshold"].SetValue(  
    Settings.BloomThreshold);  
  
DrawFullscreenQuad(resolveTarget, renderTarget1,  
    bloomExtractEffect,  
    IntermediateBuffer.PreBloom);
```

El primero consiste en dibujar la escena en el primer renderTarget usando el shader inicial (BloomExtract) que obtiene los píxeles más brillantes de la imagen.

```
        SetBlurEffectParameters(1.0f / (float)renderTarget1.Width,  
0);  
  
        DrawFullscreenQuad(renderTarget1.GetTexture(),  
renderTarget2, gaussianBlurEffect, IntermediateBuffer.BlurredHorizontally);
```

El segundo paso dibuja del renderTarget 1 al renderTarget 2 usando el shader que aplica el desenfoque gaussiano en horizontal.

```
        SetBlurEffectParameters(0, 1.0f /  
(float)renderTarget1.Height);  
  
        DrawFullscreenQuad(renderTarget2.GetTexture(),  
renderTarget1,  
gaussianBlurEffect,  
IntermediateBuffer.BlurredBothWays);
```

El tercer paso dibuja del renderTarget2 al renderTarget1 usando el desenfoque gaussiano en vertical.

```
        GraphicsDevice.SetRenderTarget(0, null);  
  
        EffectParameterCollection parameters =  
bloomCombineEffect.Parameters;  
  
parameters["BloomIntensity"].SetValue(Settings.BloomIntensity);  
parameters["BaseIntensity"].SetValue(Settings.BaseIntensity);  
parameters["BloomSaturation"].SetValue(Settings.BloomSaturation);  
parameters["BaseSaturation"].SetValue(Settings.BaseSaturation);  
  
        GraphicsDevice.Textures[1] = resolveTarget;  
  
        Viewport viewport = GraphicsDevice.Viewport;  
  
        DrawFullscreenQuad(renderTarget1.GetTexture(),  
viewport.Width, viewport.Height,  
bloomCombineEffect,  
IntermediateBuffer.FinalResult);  
    }
```

Por último se dibuja el renderTarget 1 con la escena original utilizando el último shader que combina ambas entradas para crear el efecto Bloom en la salida.

Veamos un ejemplo de la salida con los diferentes tipos de efecto Bloom y sin efecto.



Figura 4.3.32 – Sin efecto Bloom



Figura 4.3.33 – Efecto Bloom sutil



Figura 4.3.34 – Efecto Bloom más pronunciado

En la primera imagen vemos que no se produce el efecto Bloom ya que está desactivado. En la segunda por el contrario ya se empiezan a apreciar partes de los modelos más iluminadas. Vemos que la escena es más luminosa. En la última escena se observa claramente el efecto, en este caso sería un tanto exagerado para una imagen normal. Se podría usar para simular estados de ánimo o situaciones concretas. Por ejemplo, un efecto Bloom exagerado como el de la imagen (o más) podría simular perfectamente una escena que transcurra en un sueño.

Ahora que hemos visto como se renderiza este efecto de postprocesado vamos a ver una pequeña introducción a los shaders en sí, es decir, al lenguaje HLSL que es el utilizado en este caso.

4.3.5.1 HLSL

Lo primero es la teoría:

HLSL ó High Level Shader Language es un lenguaje desarrollado por Microsoft para su uso sobre aplicaciones Direct3D. Sus componentes son principalmente dos:

- **Vertex Shader:** Se ejecuta para cada vértice y es el responsable de transformar el vértice del objeto para poder verlo, generar las coordenadas de las texturas y calcular los coeficientes de iluminación (tangente del vértice, coeficientes de iluminación, binormales, etc.). Cuando un grupo de vértices (generalmente los tres que forman cada triángulo) pasan por este componente, su posición de salida es interpolada para formar píxeles dentro de su área. A esto lo conocemos como rasterización. En la siguiente figura vemos el ejemplo.

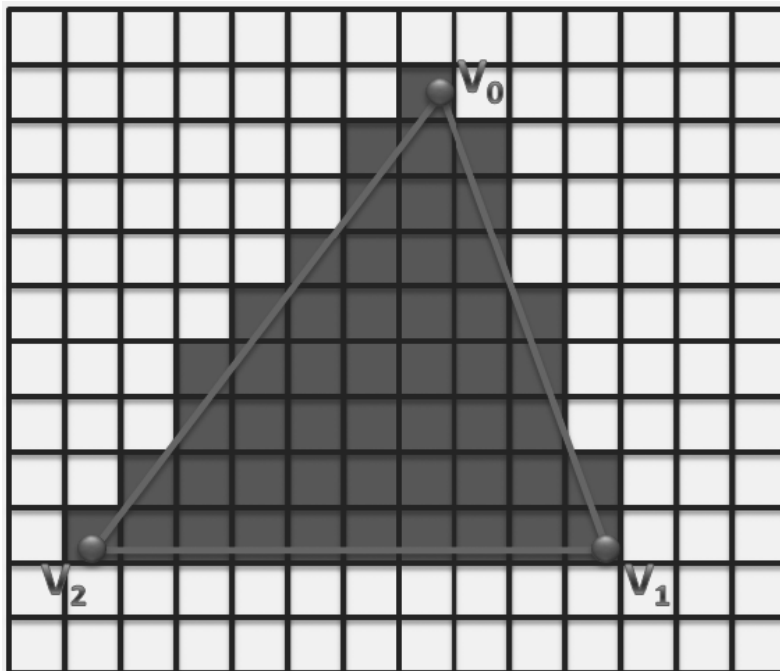


Figura 4.3.35 - Triangulo rasterizado

- **Pixel shader:** En este componente se calcula el color resultante para la pantalla.

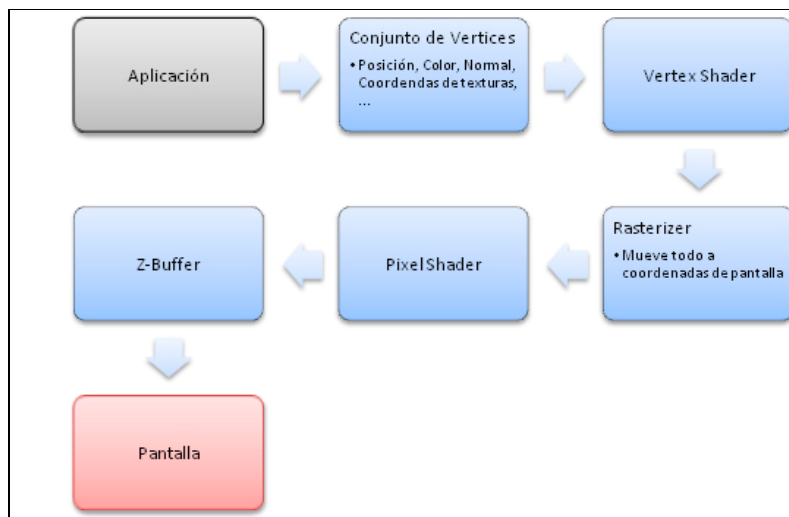


Figura 4.3.36 - Esquema HLSL

Como vemos en la figura de la aplicación pasamos al Vertex Shader, a continuación se rasteriza la información y se le pasa al Pixel shader, por último pasamos al Z-Buffer (ó FrameBuffer) y de ahí a la pantalla.

Al igual que en otros lenguajes, en HLSL tenemos una serie de tipos de datos. Aquí vemos algunos ejemplos:

Tipos de datos

bool: true/false
int: 32-bit integer
half: 16bit integer
float: 32bit float
double: 64bit double

Vectores

float3
vector
float2

Matrices

float3x3: 3x3 matrix float
float2x2: 2x2 matrix float

Además se pueden usar estructuras como if-else, for, while, do, etc. Y disponemos también de funciones propias para realizar funciones como el seno, coseno, producto vectorial, normalizar, etc.

En XNA los shaders se guardarán en ficheros .fx también conocidos como **Effect Files**. Estos ficheros son de gran utilidad ya que nos permiten guardar variables globales, funciones, estructuras, el vertex shader, el pixel shader, texturas y diferentes técnicas y pases. Un shader puede tener una o más **Técnicas (Techniques)**. Cada técnica puede tener un nombre único y contendrá uno o más **Pasos (Passes)**. Desde la aplicación podemos seleccionar la técnica en el shader que queremos usar asignándoselo a la propiedad `CurrentTechnique` de la clase `Effect`.

```
effect.CurrentTechnique = effect.Techniques["AmbientLight"];
```

En este caso le estamos diciendo al efecto que use la técnica `AmbientLight`.

El siguiente ejemplo tiene una técnica y un pase.

```
technique Shader
{
    pass P0
    {
        VertexShader = compile vs_2_0 VS();
        PixelShader = compile ps_2_0 PS();
    }
}
```

Este es otro ejemplo usando una técnica con dos pases:

```

technique Shader
{
    pass P0
    {
        VertexShader = compile vs_2_0 VS();
        PixelShader = compile ps_2_0 PS();
    }
    pass P1
    {
        VertexShader = compile vs_2_0 VS_Other();
        PixelShader = compile ps_2_0 PS_Other();
    }
}

```

Y por último este ejemplo contiene dos técnicas y dos pases (uno cada una):

```

technique Shader_11
{
    pass P0
    {
        VertexShader = compile vs_2_0 VS();
        PixelShader = compile ps_2_0 PS();
    }
}
technique Shader_2a
{
    pass P0
    {
        VertexShader = compile vs_2_0 VS2();
        PixelShader = compile ps_2_a PS2();
    }
}

```

En el primer ejemplo, vemos que la técnica tiene dos funciones, una para el pixel shader y otra para el vertex shader:

```

VertexShader = compile vs_2_0 VS2();
PixelShader = compile ps_2_0 PS2();

```

Con estas funciones indicamos que se va a usar VS2() como vertex shader y PS2() como pixel shader y por tanto soportaremos shaders del modelo 1.1 y superiores. Esto nos permite aplicar diferentes shaders en función de la capacidad de nuestra tarjeta gráfica y de su GPU. En XNA 4.0 se ha eliminado el soporte a modelos bajos ya que están muy anticuados para las tarjetas actuales.

Para implementar un shader en XNA debemos seguir estos pasos:

1. Programar el shader.
2. Añadir el fichero .fx con el shader al Content del proyecto.

Asset Name	Shader
Build Action	Compile
Content Importer	Effect - XNA Framework
Content Processor	Effect - XNA Framework
Copy to Output Director	Do not copy
File Name	Shader.fx
Full Path	F:\Dark Codex Studios\de_5

Figura 4.3.37- Content

3. Crear una instancia de la clase `Effect`:

```
Effect effect;
```

4. Inicializar dicha instancia:

```
effect = Content.Load<Effect>("Shader");
```

5. Empezar el shader y seleccionar la técnica que queremos usar:

```
effect.CurrentTechnique = effect.Techniques["AmbientLight"];
foreach (EffectPass pass in effect.CurrentTechnique.Passes)
{
    //0 indica el primer paso
    effect.CurrentTechnique.Passes[0].Apply();
}
```

6. Pasar los diferentes parámetros al shader:

```
effect.Parameters["matWorldViewProj"].SetValue( worldMatrix *
viewMatrix * projMatrix);
```

Donde `matWorldViewProj` es una variable definida en el shader (`float4x4 matWorldViewProj`) y que resultará de multiplicar las matrices `worldMatrix * viewMatrix * projMatrix`.

El método `SetValue` manda un valor al shader mientras que `GetValue<Type>` recoge un valor del shader, donde `<Type>` es un tipo de dato (por ejemplo, `GetValueInt16`).

7. Dibujar la escena:

Por último vamos a ver un ejemplo de shader para crear una luz ambiental. Recordemos que la fórmula típica de la luz ambiental es:

$$L = \text{Intensidad} \times \text{Color}$$

Lo primero que necesitamos es una matriz que represente el mundo. Para ello declaramos la siguiente variable:

```
float4x4 matWorldViewProj;
```

La declaramos al inicio del shader como una variable global.

A continuación necesitaremos saber que valores pasara el vertex shader al pixel shader. Para ello crearemos una estructura que llamaremos OUT.

```
struct OUT  
{  
    float4 Pos: POSITION;  
};
```

Esta estructura contendrá una variable del tipo float4 con el nombre Pos. El nombre POSITION indica a la GPU que registro colocar en este valor. Este es simplemente un contenedor alojado en la GPU que contiene datos. La GPU contiene diferentes registros para poner datos de posición, normales, coordenadas de texturas, etc. y cuando se define una variable que pasaremos al pixel shader debemos decidir en que lugar de la GPU se guardará ese dato.

Ahora veamos el pixel shader:

```
OUT VertexShaderFunction( float4 Pos: POSITION )  
{  
    OUT Out = (OUT) 0;  
    Out.Pos = mul(Pos, matWorldViewProj);  
    return Out;  
}
```

Creamos un vertex shader del tipo OUT que hemos definido anteriormente. El parámetro de entrada es Pos de tipo float4. Esta es la posición del vértice definido en la aplicación.

A continuación creamos una instancia de la estructura de tipo OUT y la llamamos Out. Esta estructura se rellenará y devolverá en la función para el procesamiento posterior. La variable de entrada Pos no tiene un procesamiento así que necesita ser multiplicada con la variable matWorldViewProj para que se vea bien en pantalla. En resumidas cuentas, por ahora lo único que hacemos es coger el parámetro de entrada y multiplicarlo por la matriz matWorldViewProj.

Ahora es el turno del pixel shader. Declaramos un parámetro de salida de tipo float4 que se guardará en el registro de tipo COLOR de la GPU. En el pixel shader es donde resolveremos el algoritmo de la luz ambiental.

```
float4 PixelShaderFunction() : COLOR
{
    float Ai = 0.8f;
    float4 Ac = float4(0.075, 0.075, 0.2, 1.0);
    return Ai * Ac;
}
```

Como veremos luego, usaremos el modelo 1.1 para calcular el color de cada píxel. La variable A_i de tipo float será la intensidad de la luz ambiental y A_c definirá el color de dicha luz. Vemos que el algoritmo es muy simple.

Por último definiremos la técnica y el paso y le diremos como hay que compilarlos y qué funciones son las que definen el vertex shader y el pixel shader.

```
technique AmbientLight
{
    pass P0
    {
        VertexShader = compile vs_1_1 VertexShaderFunction();
        PixelShader = compile ps_1_1 PixelShaderFunction();
    }
}
```

El resultado final es el que podemos ver en la siguiente figura:

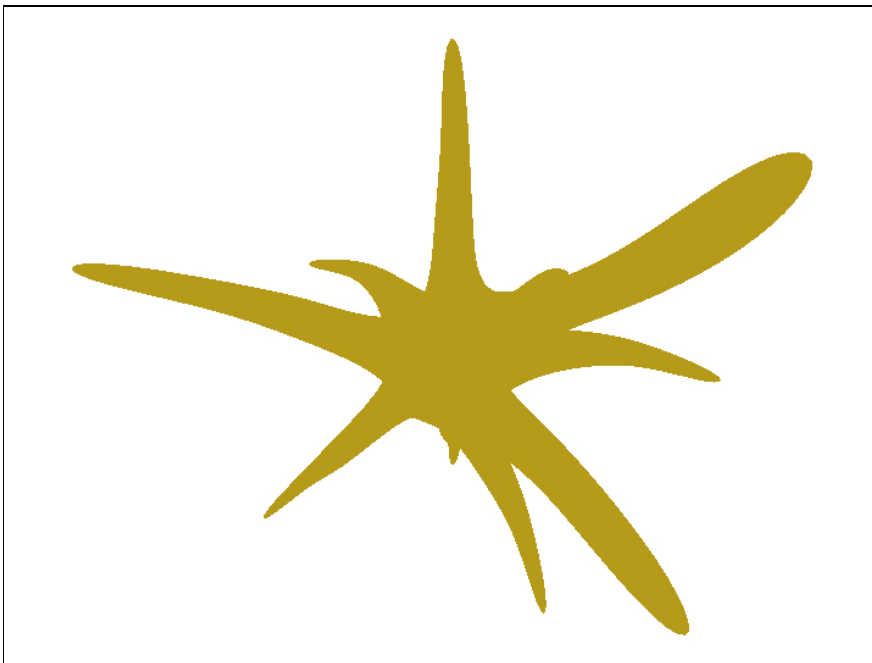


Figura 4.3.38 - Shader de luz ambiental

Como vemos el ejemplo no es precisamente espectacular, ya que el algoritmo que hemos visto no es muy complejo. Veamos pues cómo mejorar esto con el algoritmo de la luz difusa que hemos visto anteriormente y que, recordemos, seguía la siguiente fórmula:

$$L = \text{Intensidad} \times \text{Color} + \text{IntensidadDifusa} \times \text{ColorDifusa} \times N.L$$

Pasemos pues a la implementación del shader. En primer lugar definimos las variables globales:

```
float4x4 matWorldViewProj;
float4x4 matWorld;
float4 vLightDirection;
```

Seguimos manteniendo la variable `matWorldViewProj`, del ejemplo anterior, pero vamos a añadir dos variables más. La matriz `matWorld` que nos ayudará a calcular la normal de cada vértice respecto al mundo actual y la variable `vLightDirection` que nos marcará la dirección de la luz. Tendremos también que definir la estructura `OUT` para el vertex shader.

```
struct OUT
{
    float4 Pos: POSITION;
    float3 L: TEXCOORD0;
    float3 N: TEXCOORD1;
};
```

Guardaremos, como antes, la posición en una variable de tipo `float4` y los vectores `L` y `N` en registros de tipo `TEXCOORDn` que puede ser usado para cualquier valor y como aún no usamos texturas podemos usarlos para guardar dichos vectores.

El vertex shader quedaría así:

```
OUT VertexShader( float4 Pos: POSITION, float3 N: NORMAL )
{
    OUT Out = (OUT) 0;
    Out.Pos = mul(Pos, matWorldViewProj);
    Out.L = normalize(vLightDirection);
    Out.N = normalize(mul(N, matWorld));
    return Out;
}
```

Obtenemos la posición y la multiplicamos igual que antes. Además normalizamos la dirección de la luz que le pasamos y por último calculamos el vector `N` como la multiplicación por la matriz `matWorld` y la normalización de dicha multiplicación.

Por programa pasaremos las variables así:

```
Vector4 vLightDirection = new Vector4(0.0f, 0.0f, 1.0f, 1.0f);  
  
effect.Parameters["matWorldViewProj"].SetValue(worldMatrix *  
viewMatrix * projMatrix);  
effect.Parameters["matInverseWorld"].SetValue(worldInverse);  
effect.Parameters["vLightDirection"].SetValue(vLightDirection);
```

El pixel shader tendrá la siguiente forma:

```
float4 PixelShader(float3 L: TEXCOORD0, float3 N: TEXCOORD1) :  
COLOR  
{  
    float Ai = 0.8f;  
    float4 Ac = float4(0.075, 0.075, 0.2, 1.0);  
    float Di = 1.0f;  
    float4 Dc = float4(1.0, 1.0, 1.0, 1.0);  
    return Ai * Ac + Di * Dc * saturate(dot(L, N));  
}
```

La función saturate realiza la siguiente comprobación:

```
if x < 0 result = 0  
else if x > result = 1  
else result = x
```

La técnica es la misma que antes:

```
technique AmbientLight  
{  
    pass P0  
    {  
        VertexShader = compile vs_1_1 VertexShaderFunction();  
        PixelShader = compile ps_1_1 PixelShaderFunction();  
    }  
}
```

Y este es el resultado final:

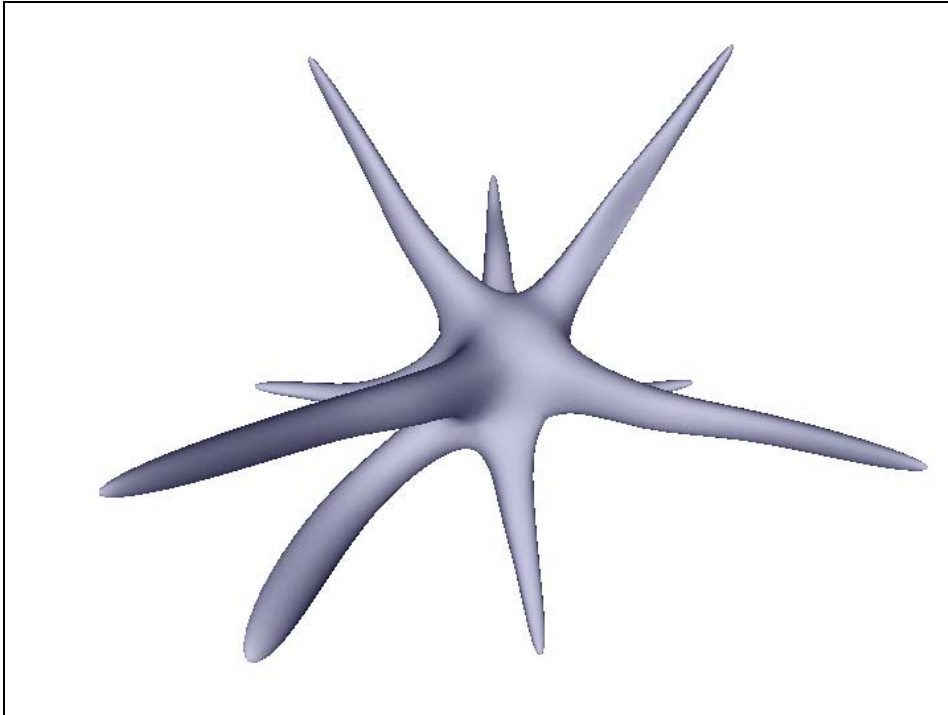


Figura 4.3.39 - Iluminación difusa

Aquí ya se puede observar como existen sombras en las zonas menos iluminadas.

Por último veremos un ejemplo práctico con la aplicación. En este caso veremos como se obtienen los píxeles más luminosos de la escena para calcular el Bloom.

En primer lugar declaramos las variables globales:

```
sampler TextureSampler : register(s0);  
float BloomThreshold;
```

Declaramos `TextureSampler` como una variable de tipo `sampler` y que contendrá una muestra de la textura. A continuación definimos la constante `BloomThreshold` que indicará el umbral elegido para la iluminación (a partir de ese umbral se considerará luminoso o no).

En este caso no tenemos vertex shader y el pixel shader tiene la siguiente forma:

```
float4 PixelShader(float2 texCoord : TEXCOORD0) : COLOR0  
{
```

```
float4 c = tex2D(TextureSampler, texCoord);
return saturate((c-BloomThreshold)/(1- BloomThreshold));
}
```

Al pixel shader se le pasa la variable `texCoord` que contendrá la posición del píxel actual a procesar. Con esa coordenada obtenemos el color de la imagen en dicha posición. Por último ajustamos dicho color para mantener solamente aquellos que están por encima del umbral. Es decir, la idea es crear una *máscara* que contenga solamente los píxeles más brillantes de la imagen a los cuales aplicaremos después el Bloom combinando esta *máscara* con la imagen original.

Por la aplicación le pasaremos el `BloomThreshold` así:

```
bloomExtractEffect.Parameters["BloomThreshold"].SetValue(
    Settings.BloomThreshold);
```

Y dicho valor lo definiremos así para los diferentes tipos de Bloom:

```
//      Name      Thresh Blur Bloom Base BloomSat BaseSat
new BloomSettings("Por defecto",0.25f, 4, 1.25f,1, 1, 1),
new BloomSettings("Suave", 0, 3, 1, 1, 1, 1),
new BloomSettings("Desaturado", 0.5f, 2, 2, 1, 0, 1),
new BloomSettings("Saturado", 0.25f, 4, 2, 1, 2, 0),
new BloomSettings("Borroso", 0, 2, 1, 0.1f,1, 1),
new BloomSettings("Sutil", 0.5f, 2, 1, 1, 1, 1),
```

La técnica se compilará con el modelo 2.0.

```
technique BloomExtract
{
    pass Pass1
    {
        PixelShader = compile ps_2_0 PixelShader();
    }
}
```

Una vez tenemos calculado este shader podemos aplicarlo para calcular el Bloom resultante:

Al igual que antes definimos las variables globales que, en este caso, serán la imagen original y la imagen obtenido con el shader anterior. Además tendremos la intensidad de la imagen original y de la del preprocesado así como la saturación de ambas.

```
sampler BloomSampler : register(s0);
sampler BaseSampler : register(s1);

float BloomIntensity;
float BaseIntensity;

float BloomSaturation;
float BaseSaturation;
```

Vamos a definir una función auxiliar cambiar la saturación de un color.

```
float4 AdjustSaturation(float4 color, float saturation)
{
    float grey = dot(color, float3(0.3, 0.59, 0.11));
    return lerp(grey, color, saturation);
}
```

Y el pixel shader:

```
float4 PixelShader(float2 texCoord : TEXCOORD0) : COLOR0
{
    float4 bloom = tex2D(BloomSampler, texCoord);
    float4 base = tex2D(BaseSampler, texCoord);

    bloom = AdjustSaturation(bloom, BloomSaturation) * BloomIntensity;
    base = AdjustSaturation(base, BaseSaturation) * BaseIntensity;

    base *= (1 - saturate(bloom));

    return base + bloom;
}
```

El proceso consiste en obtener el color de la imagen original y la de la *máscara*. Una vez las tenemos, modificamos la saturación en función de las variables globales. Por último oscurecemos la imagen original en aquellas zonas que están muy iluminadas y que por tanto van a tener mucho Bloom. Esto lo hacemos para evitar que la imagen se *queme*. El valor de salida será la suma de ambos colores.

4.3.6 Partículas

El último aspecto a tener en cuenta si queremos simular un entorno realista es la gestión de partículas. En nuestra aplicación, la escena de sitúa en un entorno nevado, por ello se han utilizado partículas para simular la nieve cayendo. Además, y dado que tenemos la posibilidad de disparar, los impactos de la bala producen un humo simulado con dichas partículas.

La generación de partículas en XNA sigue una lógica relativamente sencilla. Se trata de crear un generador de partículas que se carga de crear instancias de objetos con una velocidad, orientación, tamaño, etc. y emitir más o menos partículas. Dichas partículas no son más que imágenes (generalmente .PNG).

Para el proyecto se ha usado una librería ya creada que se llama DSPF.

Poner en marcha un sistema de partículas con DPSF es muy sencillo. Estos son los pasos a seguir:

1. Declaramos el sistema de partículas y lo inicializamos vacío:

```
SnowParticleSystem snParticleSystem = null;
```

2. Llamamos al constructor pasándole el Game:

```
snParticleSystem = new SnowParticleSystem(this);
```

3. Realizamos la llamada al método AutoInitialize para que se inicie en cuanto arranque el juego:

```
snParticleSystem.AutoInitialize(this.GraphicsDevice,  
this.Content, null);
```

4. Definimos las matrices de vista, proyección y world:

```
snParticleSystem.SetWorldViewProjectionMatrices(  
Matrix.Identity, camara.view, camara.projection);
```

5. Llamamos al método Draw del sistema de partículas:

```
snParticleSystem.Draw();
```

6. Por último cuando se sale de la aplicación el sistema de partículas se destruye:

```
snParticleSystem.Destroy();
```

Por su parte la clase `SnowParticleSystem` tiene la siguiente forma:

```
class SnowParticleSystem : DefaultPointSpriteParticleSystem  
{  
    /// <summary>  
    /// Constructor  
    /// </summary>  
    public SnowParticleSystem(Game cGame) : base(cGame) { }  
  
    public Vector3 mcMaxWindForce = new Vector3(50, 10, 0);  
}
```

Se define la fuerza máxima del viento.

```
protected override void SetEffectParameters()  
{  
    base.SetEffectParameters();  
    Effect.Parameters["xColorBlendAmount"].SetValue(1.0f);  
}
```


Se le indica a la clase que no debe utilizar el componente Color de la textura a la hora de dibujarla (creando así el efecto de fondo transparente):

```
public override void AutoInitialize(GraphicsDevice
cGraphicsDevice, ContentManager cContentManager, SpriteBatch
cSpriteBatch)
{
    InitializePointSpriteParticleSystem(cGraphicsDevice,
cContentManager, 30000, 50000, UpdateVertexProperties,
"Texturas/Cloud");

    LoadSnowEvents();
    Emitter.ParticlesPerSecond = 120;
    Name = "Snow";
}
```

En el método AutoInitialize definimos la cantidad de partículas que se alojarán en memoria, la textura que generaremos, los eventos que ocurrirán con las partículas y el número de partículas generadas por segundo.

```
public void InitializeParticleSnow(DefaultPointSpriteParticle
cParticle)
{
    Vector3 sPosition = Emitter.PositionData.Position;
    sPosition.Y = 100;
    sPosition.X += RandomNumber.Next(-80, 80);
    sPosition.Z += RandomNumber.Next(-60, 180);

    cParticle.Lifetime = 0.0f;

    cParticle.Position = sPosition;
    cParticle.Size = RandomNumber.Next(1, 2);
    cParticle.Color = DPSFHelper.LerpColor(new Color(255, 255,
255, 50), new Color(255, 255, 255, 255), RandomNumber.NextFloat());
    cParticle.Rotation = RandomNumber.Between(0,
MathHelper.TwoPi);

    cParticle.Velocity = new Vector3(RandomNumber.Next(-10,
3), RandomNumber.Next(-15, -5), RandomNumber.Next(-10, 10));
    cParticle.Acceleration = Vector3.Zero;
    cParticle.RotationalVelocity = RandomNumber.Between(-
MathHelper.PiOver2, MathHelper.PiOver2);
}
```

Definiremos la posición desde donde empezarán las partículas (con un valor fijo en las Y y un random entre dos posiciones para las X y las Z). Además definiremos el tiempo de vida de cada partícula, su tamaño, el color, velocidad, etc.

```
public void LoadSnowEvents()
{
    ParticleInitializationFunction = InitializeParticleSnow;

    ParticleEvents.RemoveAllEvents();
    ParticleEvents.AddEveryTimeEvent(
UpdateParticlePositionUsingExternalForce, 100);
    ParticleEvents.AddEveryTimeEvent(
UpdateParticlePositionAndVelocityUsingAcceleration, 500);
}
```

```

ParticleEvents.AddEveryTimeEvent(
UpdateParticleRotationUsingRotationalVelocity, 500);
ParticleEvents.AddEveryTimeEvent(
UpdateParticleDieOnceGroundIsHit, 1000);
ParticleEvents.AddEveryTimeEvent(
UpdateParticleChangeVelocityAndRotation);
}

```

Definimos los eventos que se van a producir (rotaciones, posiciones, etc.) en las partículas a lo largo del tiempo.

```

protected void
UpdateParticleDieOnceGroundIsHit(DefaultPointSpriteParticle cParticle,
float fElapsedTimeInSeconds)
{
    if (cParticle.Position.Y < -5)
    {
        cParticle.Lifetime = 1.0f;
    }
}

```

Miramos si la partícula está por debajo de un límite para destruirla.

```

protected void
UpdateParticleChangeVelocityAndRotation(DefaultPointSpriteParticle
cParticle, float fElapsedTimeInSeconds)
{
    if (RandomNumber.Next(0, 100) == 50)
    {
        Vector3 sChange = new Vector3(RandomNumber.Next(-3,
3), RandomNumber.Next(-3, 3), RandomNumber.Next(-3, 3));
        cParticle.Velocity += sChange;
        cParticle.RotationalVelocity += RandomNumber.Between(-
MathHelper.PiOver4, MathHelper.PiOver4);
    }
}

protected void
UpdateParticleAddWindForce(DefaultPointSpriteParticle cParticle, float
fElapsedTimeInSeconds)
{
    cParticle.ExternalForce = Vector3.Lerp(Vector3.Zero,
mcMaxWindForce, RandomNumber.Between(0.1f, 1.0f));
}

protected void
UpdateParticleRemoveWindForce(DefaultPointSpriteParticle cParticle,
float fElapsedTimeInSeconds)
{
    cParticle.ExternalForce = Vector3.Zero;
}

```

Implementación de las funciones de los eventos.

```

public void AddWindForce()
{
    this.ParticleEvents.AddOneTimeEvent(UpdateParticleAddWindForce);
}

```

```
public void RemoveWindForce()  
{  
    this.ParticleEvents.AddOneTimeEvent(UpdateParticleRemoveWindForc  
e);  
}
```

Funciones para añadir o eliminar viento.

Veamos un ejemplo de varias partículas actuando a la vez.



Figura 4.3.40 - Partículas

En este caso diferenciamos tres tipos de partículas:

1. Con los recuadros blancos vemos la nieve.
2. Con el recuadro rojo vemos la partícula del fuego de la metralleta.
3. Con el cuadro negro vemos tres tipos de partículas que a continuación ampliamos:

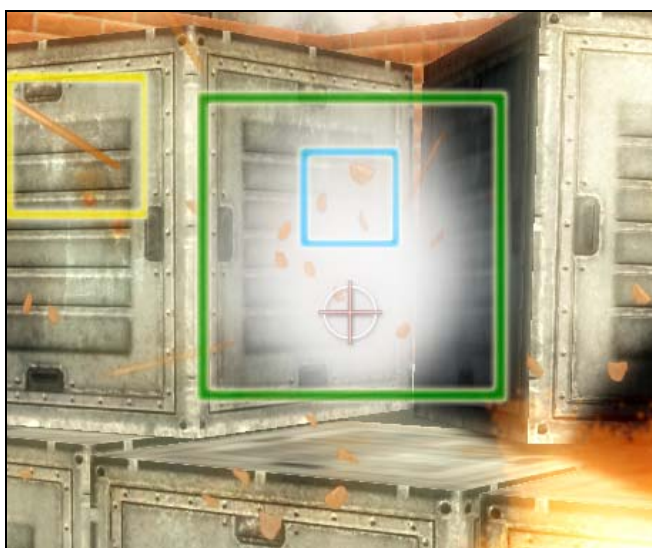


Figura 4.3.41 - Partículas

- En el cuadro amarillo vemos las chispas del disparo.
- En el cuadro azul vemos la simulación de trozos de metal volando.
- En el cuadro verde vemos el humo generado al chocar la bala contra la caja.

4.4 Navegación

Como hemos comentado al principio, la navegación en nuestro entorno virtual va a ser a través del mando de la Wii, más conocido como Wiimote, unido a su complemento Nunchuck.

Como también hemos dicho, el manejo de este dispositivo se realizará a través de la librería creada por Brian Peek.

La secuencia de uso de la librería es muy sencilla:

1. Añadimos la librería:

```
using WiimoteLib;
```

2. Declaramos una variable de tipo `Wiimote` y dos de tipo `WiimoteState` (una con el estado actual del Wiimote y otra con el último estado guardado):

```
//Variables Wii  
Wiimote wmc;  
WiimoteState wiiMoteState;  
WiimoteState lastWiiMoteState;
```

3. Creamos una estructura llamada `boton`. Esta estructura nos servirá para guardar información de los botones pulsados:

```
//Botones Wii
public struct boton
{
    public string nom;
    public bool estado;
    public int pulsaciones;
}

boton btnA;
...
```

4. En el constructor instanciaremos la variable `wmc`. Además conectaremos el Wiimote con la llamada al método `Connect()` y por último instanciaremos las dos variables con el estado del Wiimote:

```
//Creamos la instancia de nuestro mando Wii, lo conectamos y
obtenemos su estado
wmc = new Wiimote();
wmc.Connect();
wiiMoteState = new WiimoteState();
lastWiiMoteState = new WiimoteState();
```

Además comprobaremos si el Nunchuck está conectado y podremos definir qué LEDs queremos que se enciendan del mando.

```
if (wm.WiimoteState.ExtensionType.ToString() == "Nunchuk")
{
    wm.SetReportType( InputReport.IRExtensionAccel,
        IRSensitivity.Maximum, true);
}
else
{
    wm.SetReportType( InputReport.IRAccel,
        IRSensitivity.Maximum, true);
}

wm.SetLEDs(true, false, false, false);
```

5. A partir de aquí ya tenemos todo lo necesario para controlar el Wiimote. Veamos algunos ejemplos:

Control de botones Wiimote

```
//Lógica de botones. Controlamos las pulsaciones de cada uno de
ellos
if (wiiMoteState.ButtonState.A)
{
    btnA.estado = true;
    btnA.pulsaciones = btnA.pulsaciones + 1;
}
...
```

En este caso estamos comprobando qué botones pulsamos. Es decir, que para acceder a los botones pulsados simplemente comprobamos la propiedad `wiiMoteState.ButtonState.Botón_a_comprobar` y nos devolverá un booleano como resultado.

La lógica de botones se ha implementado ya que, como recordamos, estamos programando en un bucle infinito y cuando pulsamos, por ejemplo, el botón A para lanzar un balón de baloncesto tenemos que generar solo un balón. Sin embargo, si miramos solamente que el botón pulsado sea el A y no miramos que las pulsaciones sean mayores de 1, generaríamos un balón por cada frame.

```
if (!wiiMoteState.ButtonState.A && btnA.pulsaciones>=1 &&
!metrallletaOn)
{...}
```

Control de botones Nunchuck

```
if (wsAct.NunchukState.C)
{
    if (wsAct.NunchukState.RawJoystick.Y > 128)
        cameraPosition += cameraDirection * speed / 5;
    ...
}
```

Al igual que el wiimote, el Nunchuck puede ser controlado mediante esta librería y podemos acceder a él también a través de la clase `WiimoteState`.

Control de acelerómetro

```
sphereEntity.LinearVelocity = new
Vector3(camara.cameraDirection.X *
wiiMoteState.AccelState.RawValues.X / distancia,
camara.cameraDirection.Y * wiiMoteState.AccelState.RawValues.Y /
distancia, camara.cameraDirection.Z *
wiiMoteState.AccelState.RawValues.Z / distancia);
```

A través de la propiedad `AccelState` y de `RawValues` podemos acceder a los valores del acelerómetro. Es decir, podremos saber cuál ha sido la aceleración en los distintos ejes.

Dentro de la aplicación usaremos el valor del acelerómetro para asignar la velocidad lineal al balón de baloncesto. Así en función de la aceleración el balón saldrá con mayor o menor ángulo y velocidad.

Control de vibración

El wiimote tiene la opción de vibrar. Para activarlo simplemente ejecutamos la siguiente orden:

```
wmc.SetRumble(true);
```

Y para desactivarlo:

```
wmc.SetRumble(false);
```

El funcionamiento de la navegación es muy sencillo. Simplemente tendremos definidos dos movimientos:

1. **Movimientos Strafe/avance/retroceso de la cámara:** El movimiento strafe es el que realiza la cámara a izquierda y derecha pero siempre manteniendo el punto de vista en el frente de la cámara, es decir que cuando movemos a izquierda o derecha el joystick, la cámara no gira sino que simplemente cambia su posición. El avance es el movimiento hacia delante y el retroceso hacia atrás.

Para poder movernos, lo primero que debemos hacer es mantener pulsado el botón C del Nunchuck y a continuación mover el joystick de dicho Nunchuck.

```
if (wsAct.NunchukState.C)
{
    if (wsAct.NunchukState.RawJoystick.Y > 128)
        cameraPosition += cameraDirection * speed / 5;
    if (wsAct.NunchukState.RawJoystick.Y < 120)
        cameraPosition -= cameraDirection * speed / 5;

    if (wsAct.NunchukState.RawJoystick.X < 128)
        cameraPosition += Vector3.Cross(cameraUp,
            cameraDirection) * speed / 5;
    if (wsAct.NunchukState.RawJoystick.X > 120)
        cameraPosition -= Vector3.Cross(cameraUp,
            cameraDirection) * speed / 5;
}
```

Las operaciones del movimiento son las que ya hemos definido antes.

2. **Giro de la cámara:** El movimiento de giro es aquel que hace que miremos hacia donde queremos.

Para realizar el giro tenemos que mantener pulsado el botón B y a continuación girar alrededor con el wiimote.

```
if (wsAct.ButtonState.B)
{
    cameraDirection = Vector3.Transform(cameraDirection,
        Matrix.CreateFromAxisAngle(cameraUp, (-MathHelper.PiOver4 /
        20) * (wsAct.AccelState.Values.X)));
    cameraDirection = Vector3.Transform(cameraDirection,
        Matrix.CreateFromAxisAngle(Vector3.Cross(cameraUp,
        cameraDirection), (MathHelper.PiOver4 / 100) *
        (wsAct.AccelState.Values.Y) * speed * 1.75f));

    // El vector Up siempre se mantiene
    cameraUp = new Vector3(0, 1, 0);
}
```

Al igual que antes las operaciones son las mismas que cuando teníamos el ratón para girar la cámara. La diferencia es que aquí giramos en función del acelerómetro, es decir, que cuanto más rápido giremos más rápido girará la cámara.

4.5 Diagrama de clases

Con todos estos componentes ya definidos e integrados en nuestra aplicación, el diagrama de clases queda definido así:

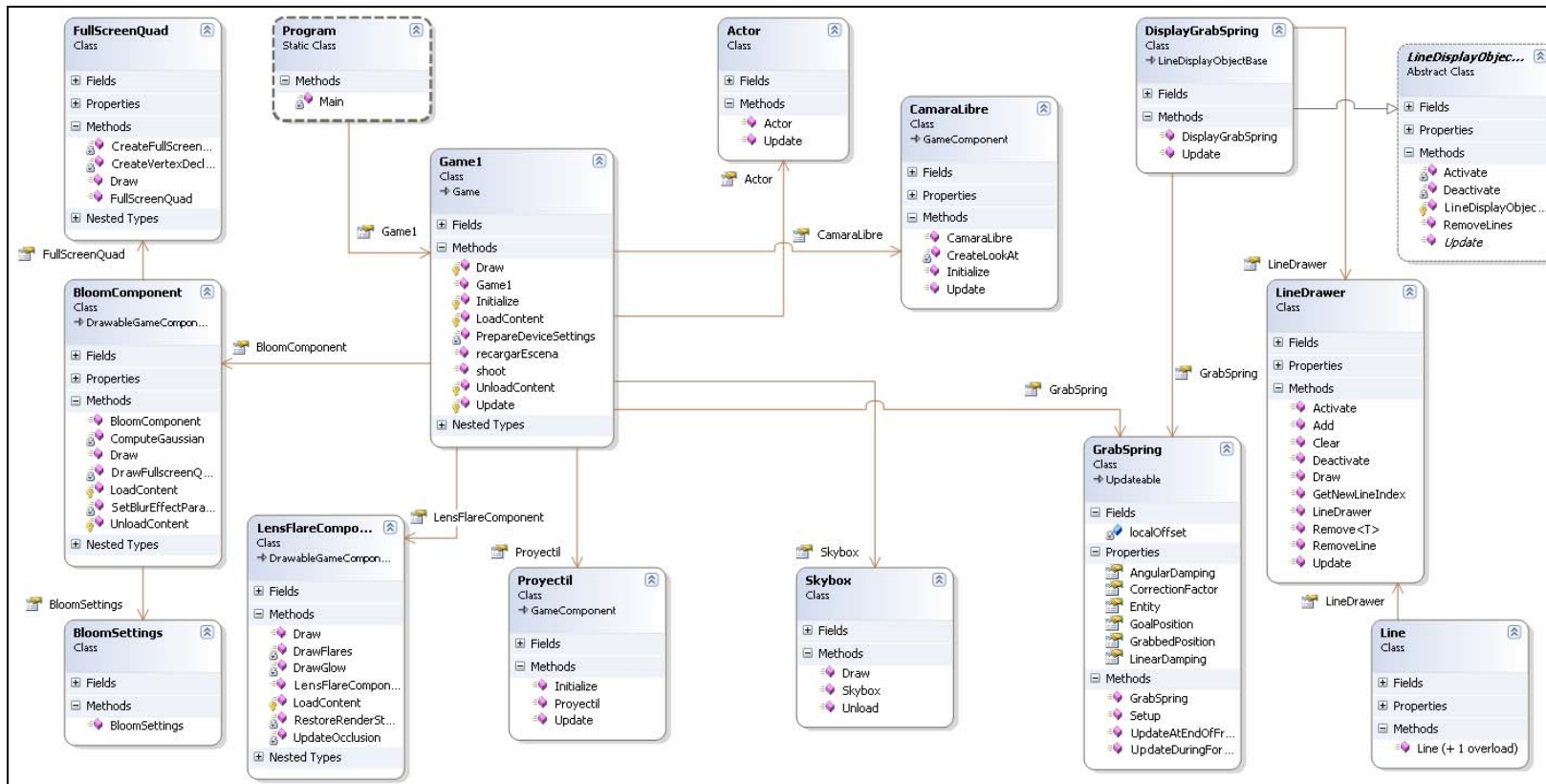


Figura 4.5.1 - Diagrama de clases

Capítulo 7

CONCLUSIONES

Este capítulo está dedicado a las conclusiones obtenidas del proyecto fin de carrera. Por un lado trata las conclusiones del producto creado, y por otro las valoraciones personales del autor. Posteriormente se estudian posibles líneas de desarrollo o continuación del proyecto presente, y finalmente, detalla el contenido del CD.

7.1 Conclusiones del proyecto

Sin duda el mundo de los videojuegos está cambiando, la irrupción de los smartphones, la especialización y potenciación de las videoconsolas de última generación y los adelantos en computación gráfica están haciendo que cada vez sean más los desarrolladores que se adentran en este mundo. Ya no hace falta tener un presupuesto enorme y un gran grupo de desarrolladores para crear un juego. El modelo de mercado ha cambiado y las herramientas disponibles en el mercado aumentan y son más sencillas de utilizar.

XNA se enmarca en este escenario cada vez más exigente y más poblado. Y sin duda da la talla ya que, a pesar de que gráficamente no haya que tener mucha calidad para que un juego triunfe, XNA nos proporciona un método de creación de juegos muy potente y con una curva de aprendizaje muy buena.

Además de esto se ha tenido en cuenta que los modos de navegación en interfaces y entornos informáticos están cambiando. El ratón sigue siendo el rey pero la irrupción de la videoconsola Wii así como recientes dispositivos como el Kinect para la XBox 360 nos permiten dilucidar un futuro donde la navegación será mucho más interactiva, rápida y eficaz.

Basándonos en XNA y el Wiimote se ha creado una aplicación con la que se ha pretendido investigar las posibilidades de dichas tecnologías así como su posible aplicación en juegos u otro tipo de aplicaciones.

Sin duda XNA provee una nueva forma de crear aplicaciones basadas en DirectX. No solo nos permite crear aplicaciones 2D y 3D de una forma sencilla sino que soporta múltiples plataformas, entre las que destacan el ordenador y XBox.

7.2 Conclusiones personales

Los objetivos planteados inicialmente cuando comencé el proyecto eran, básicamente el aprendizaje de las técnicas de creación de aplicaciones 3D. El mundo de los juegos siempre me ha parecido muy interesante, a la vez que complicado.

Al ser capaz de crear una aplicación que combina todos los aspectos que inicialmente tenía pensados aplicar, considero que se ha alcanzado el primero de los objetivos planteados.

Además, personalmente, este proyecto me ha introducido en este mundo apasionante de la creación de videojuegos. Sin duda seguiré creando nuevos juegos e investigando nuevas técnicas game engines y frameworks.

7.3 Líneas futuras

La primera línea futura sería la creación de un juego específico ya que actualmente la aplicación no tiene una finalidad específica.

Otra de las líneas futuras es la interacción con otros dispositivos. Uno de los dispositivos con más posibilidades futuras es el Kinect de Microsoft. Kinect ha supuesto una revolución como lo fue el Wiimote de Nintendo. Su capacidad de reconocer todo el cuerpo del jugador junto al SDK recién lanzado por el equipo de XNA prometen un aluvión de aplicaciones y nuevas experiencias de usuario.

7.4 Contenido del DVD

1. Solución Final → Contiene tanto la solución de la aplicación como el ejecutable.
2. Otras implementaciones → Contiene otras soluciones creadas a lo largo del proyecto.
3. Memoria → Contiene el archivo con la memoria.
4. Presentación → Contiene el archivo de la presentación.

Capítulo 8

BIBLIOGRAFÍA

Este capítulo está dedicado a recopilar todas las fuentes consultadas durante la creación del proyecto fin de carrera. Se recopilarán tanto libros como enlaces web.

8.1 Libros

Título: **Beginning XNA 3.0 Game Programming
From Novice to Professional**

Autores: Bruno Pereira Evangelista , Alexandre Santos Lobao , Riemer Grootjans y Antonio Leal de Farias

Enlace: <http://www.apress.com/9781430218173>

Título: **Learning XNA 3.0
XNA 3.0 Game Development for the PC, Xbox 360, and Zune**

Autor: Aaron Reed

Enlace: <http://shop.oreilly.com/product/9780596521967.do>

8.2 Enlaces Web

- Motor de iluminación SunBurn y game engine: <http://www.synapsegaming.com/>
- Motor fisico BEPU Physics: <http://bepu.squarespace.com/>
- Página principal de XNA: <http://create.msdn.com/en-US/>
- Página de descarga de XNA 3.1:
<http://www.microsoft.com/download/en/details.aspx?id=39>
- Game engine Unreal Development Kit: <http://www.udk.com/>
- Game engin CryEngine: <http://mycryengine.com/>
- Motor fisico Farseer Physics: <http://farseerphysics.codeplex.com/>
- Motor fisico Jello Physics:
<http://walaber.com/index.php?action=showitem&id=16>
- Motor fisico Box2Dx: <http://code.google.com/p/box2dx/>
- Motor fisico JigLibX: <http://www.codeplex.com/JigLibX>
- Motor fisico BulletX:
<http://www.codeplex.com/xnadevru/Wiki/View.aspx?title=Managed%20Bullet%20Physics%20Library>
- Motor fisico BEPUphysics: <http://www.bepu-games.com/BEPUphysics/>

- Motor físico Matali Physics: <http://www.mataliphysics.com/>
- Motor físico DigitalRune Physics:
<http://www.digitalrune.com/Products/Physics.aspx>

ANEXOS

Anexo I

Diagrama de clases XNA

Estos son los diferentes diagramas de clases de los componentes de XNA:

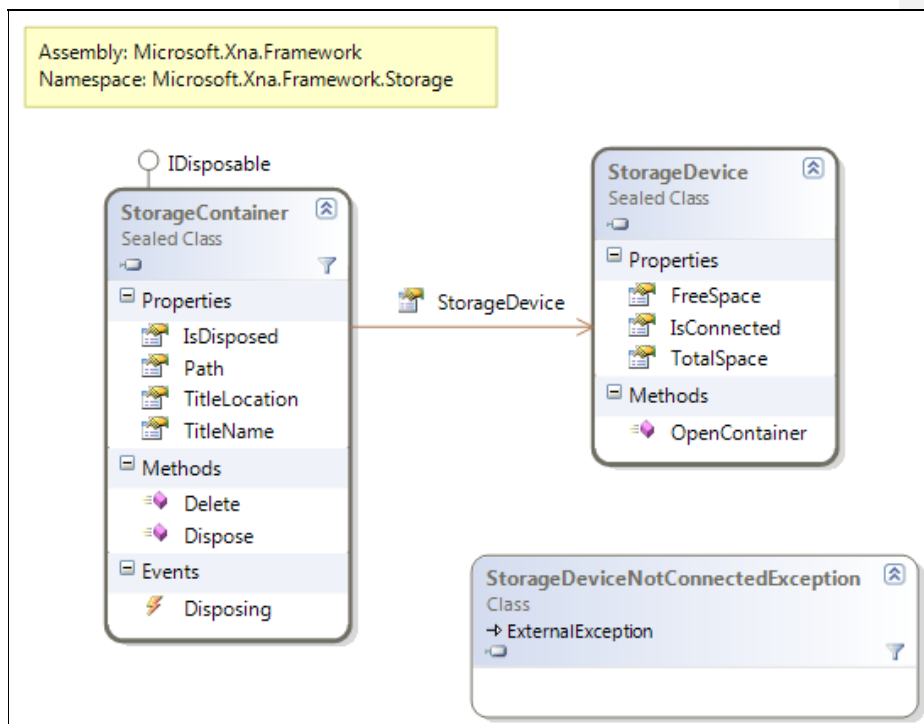


Figura Anexo 1 - Storage

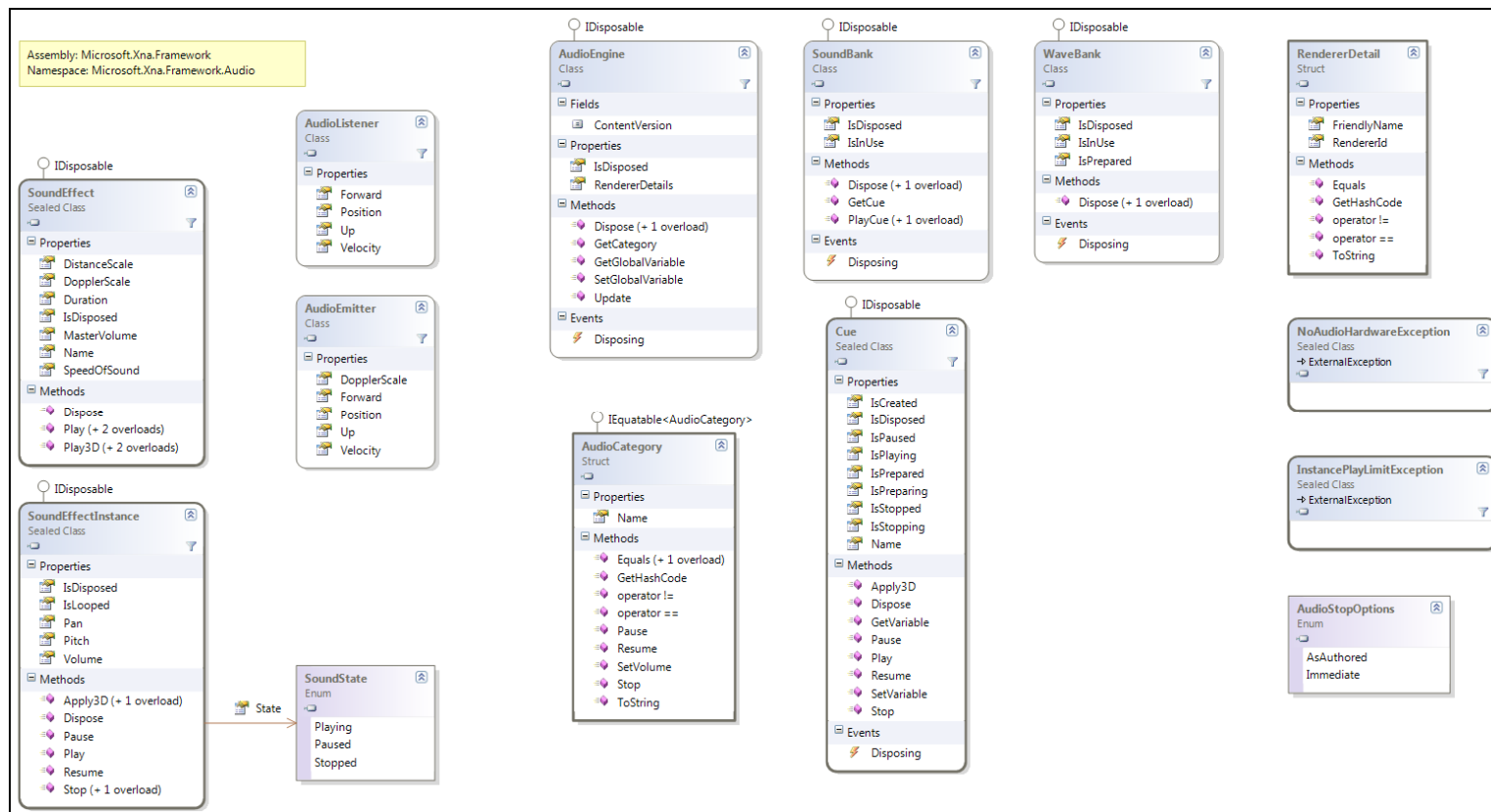


Figura Anexo 2 - Audio

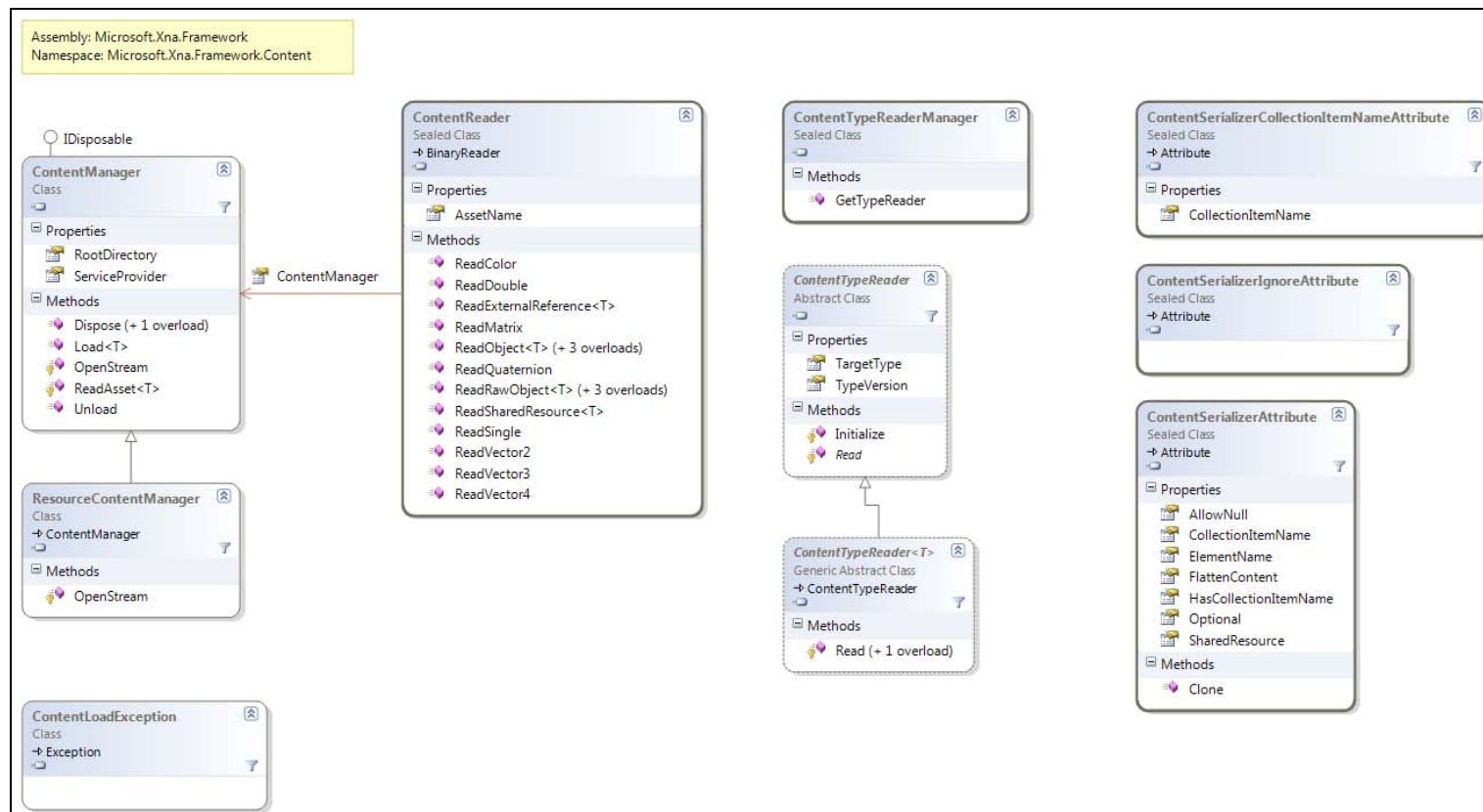


Figura Anexo 3 - Content

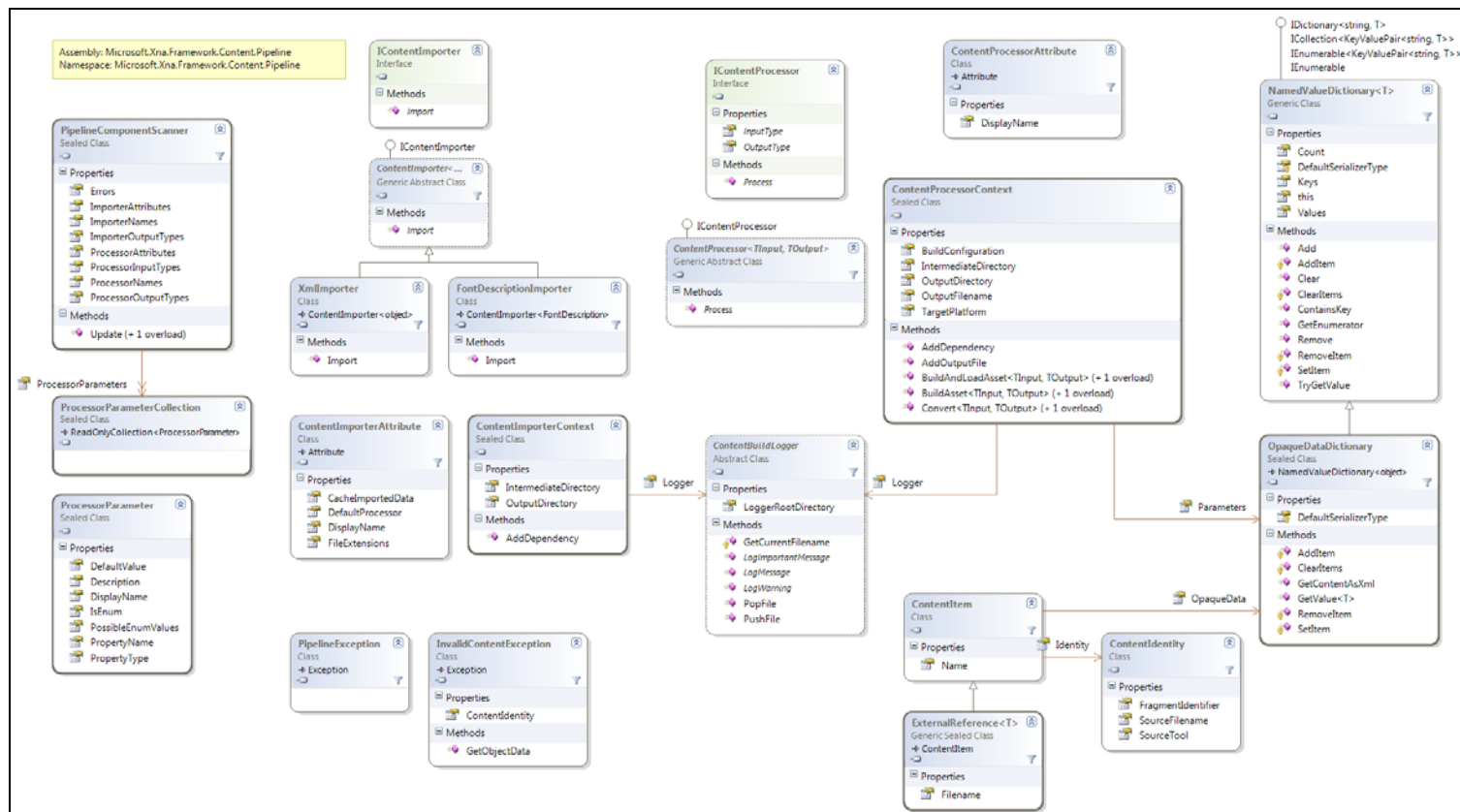


Figura Anexo 4 - Content.Pipeline

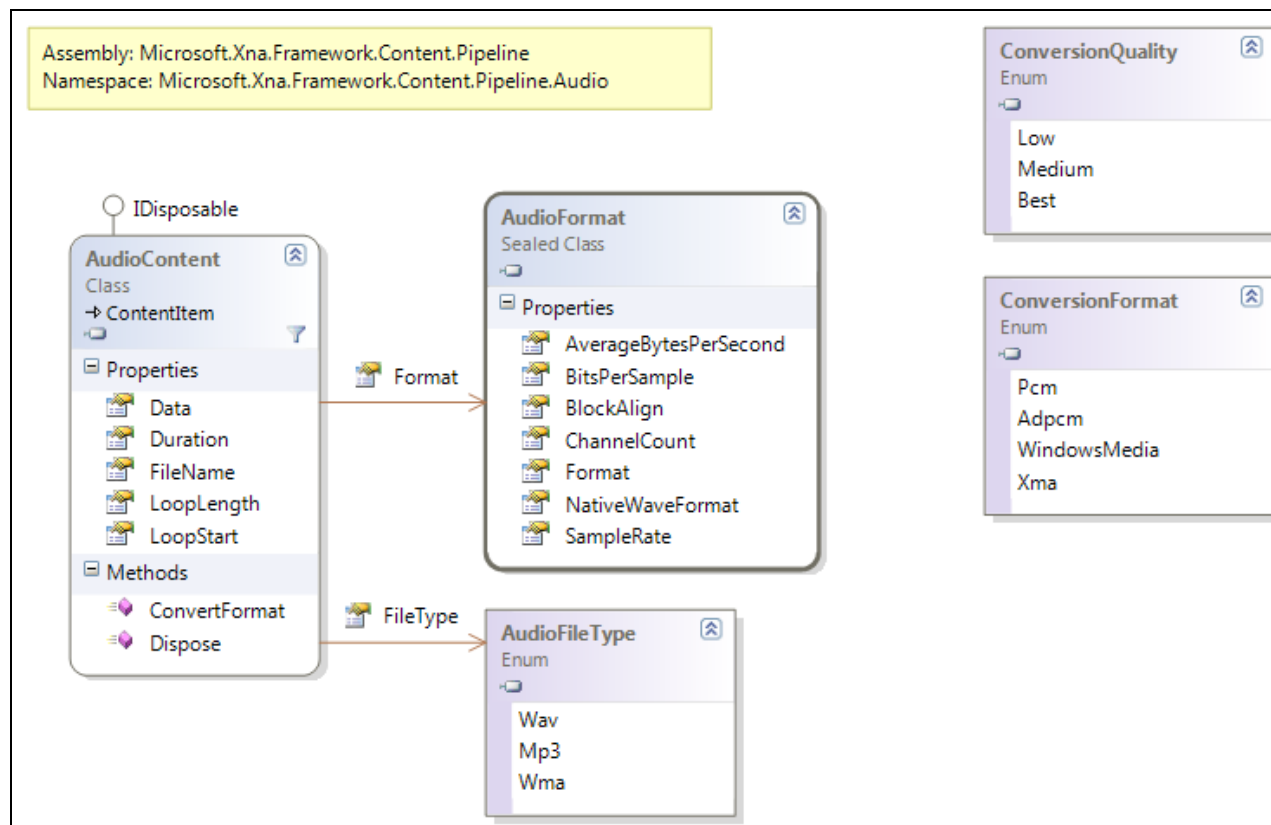


Figura Anexo 5 - Content.Pipeline.Audio

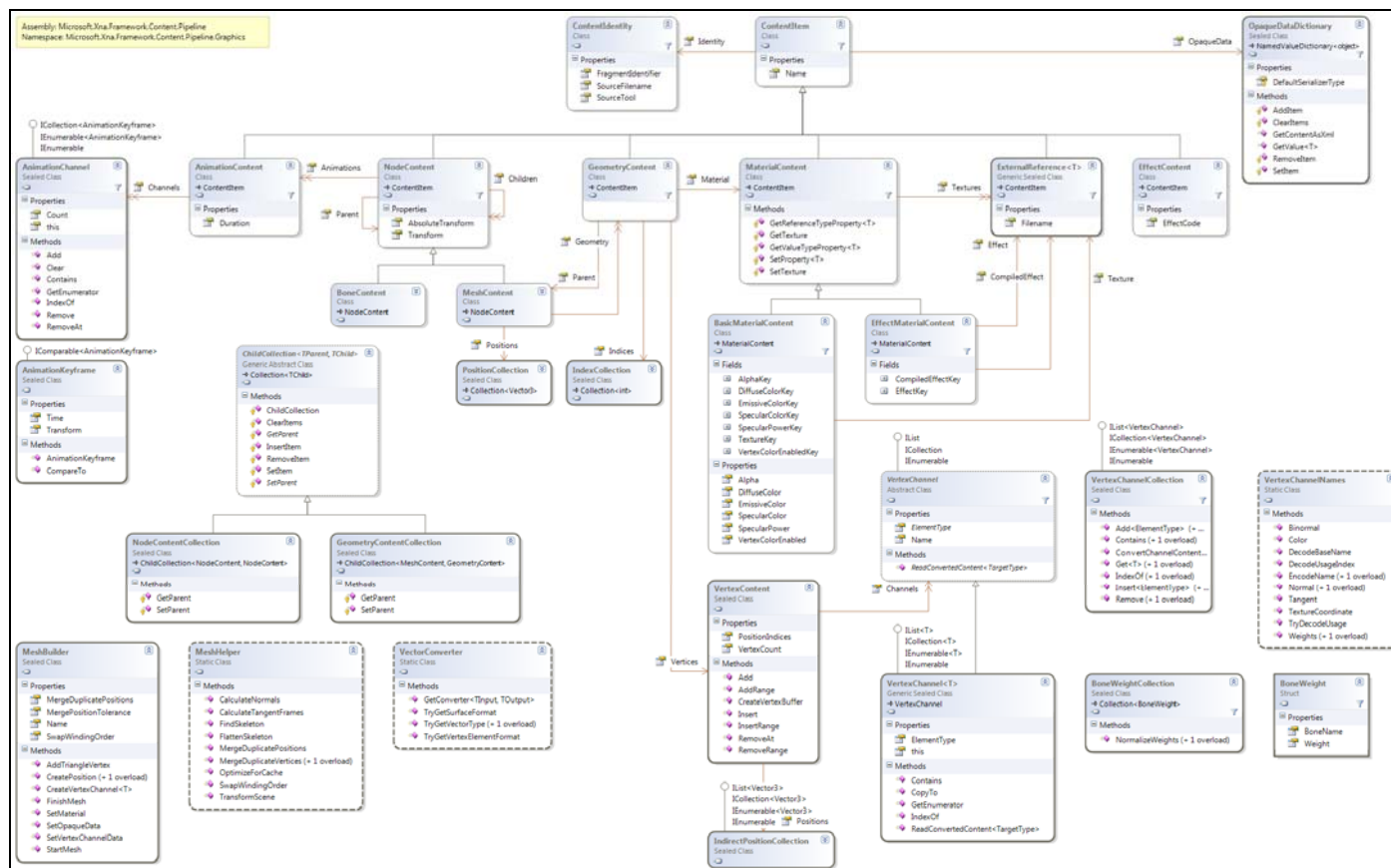


Figura Anexo 6 - Content.Pipeline.Graphics

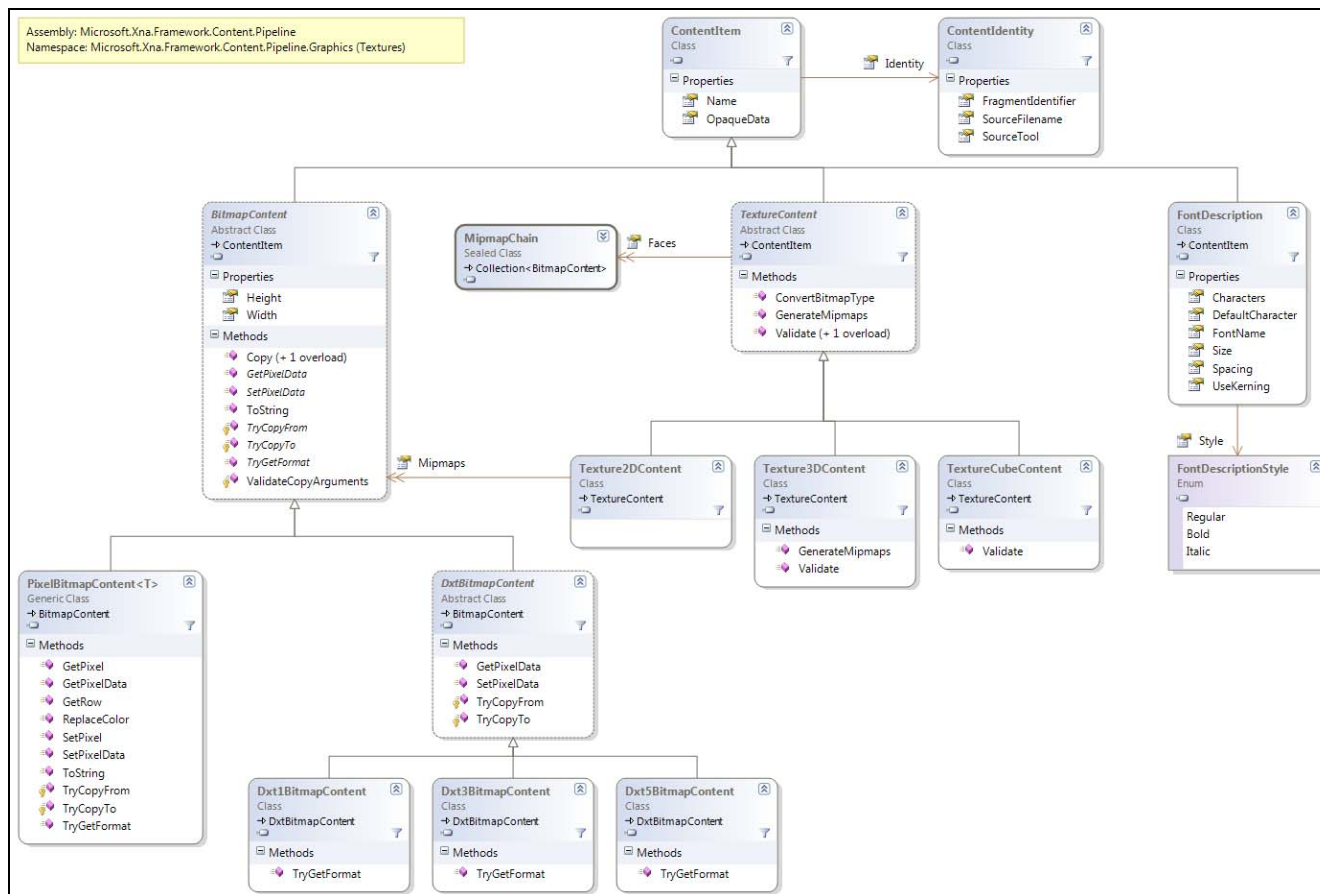


Figura Anexo 7 - Content.Pipeline.Graphics(Textures)

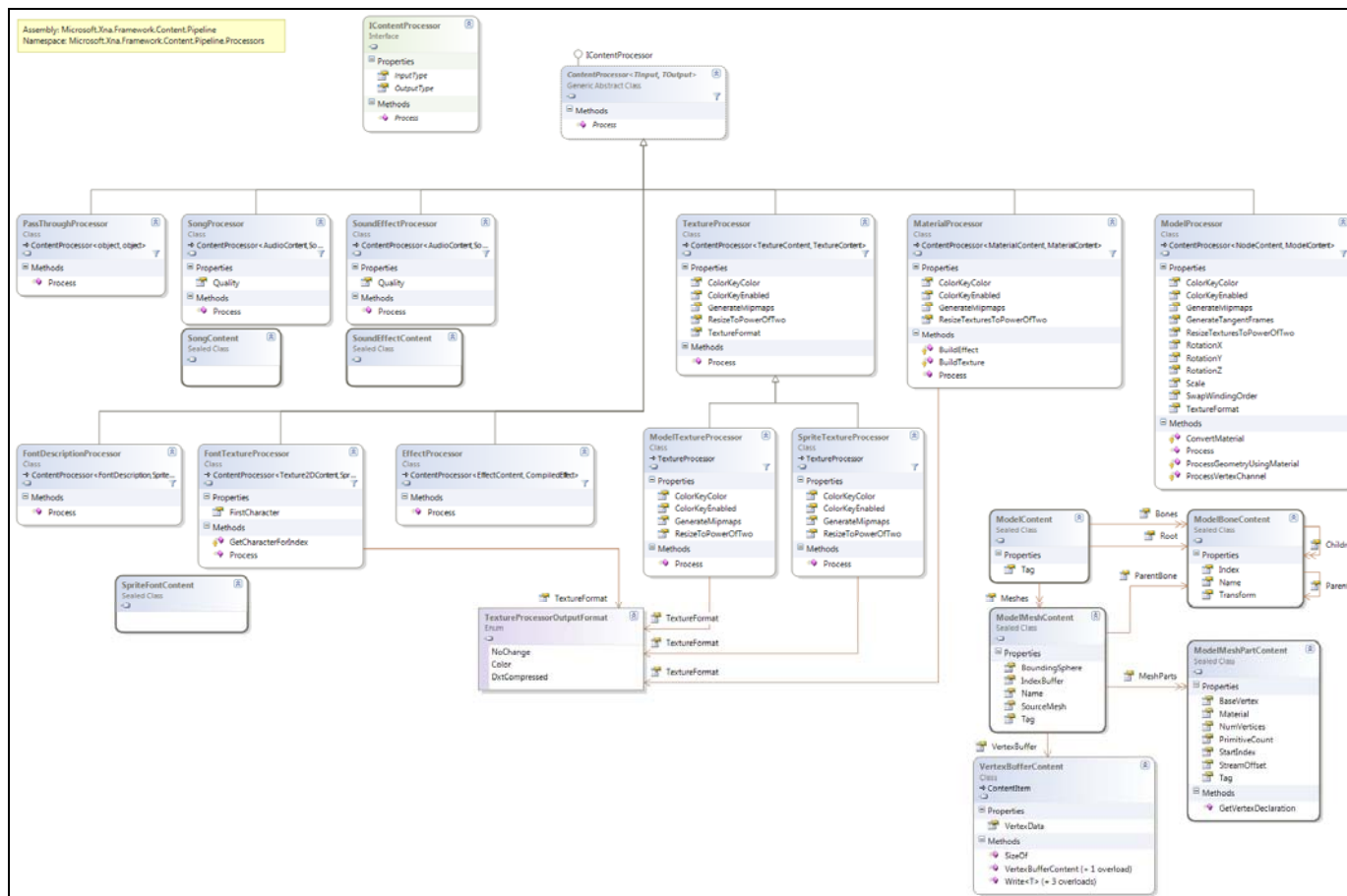


Figura Anexo 8 - Content.Pipeline.Processors

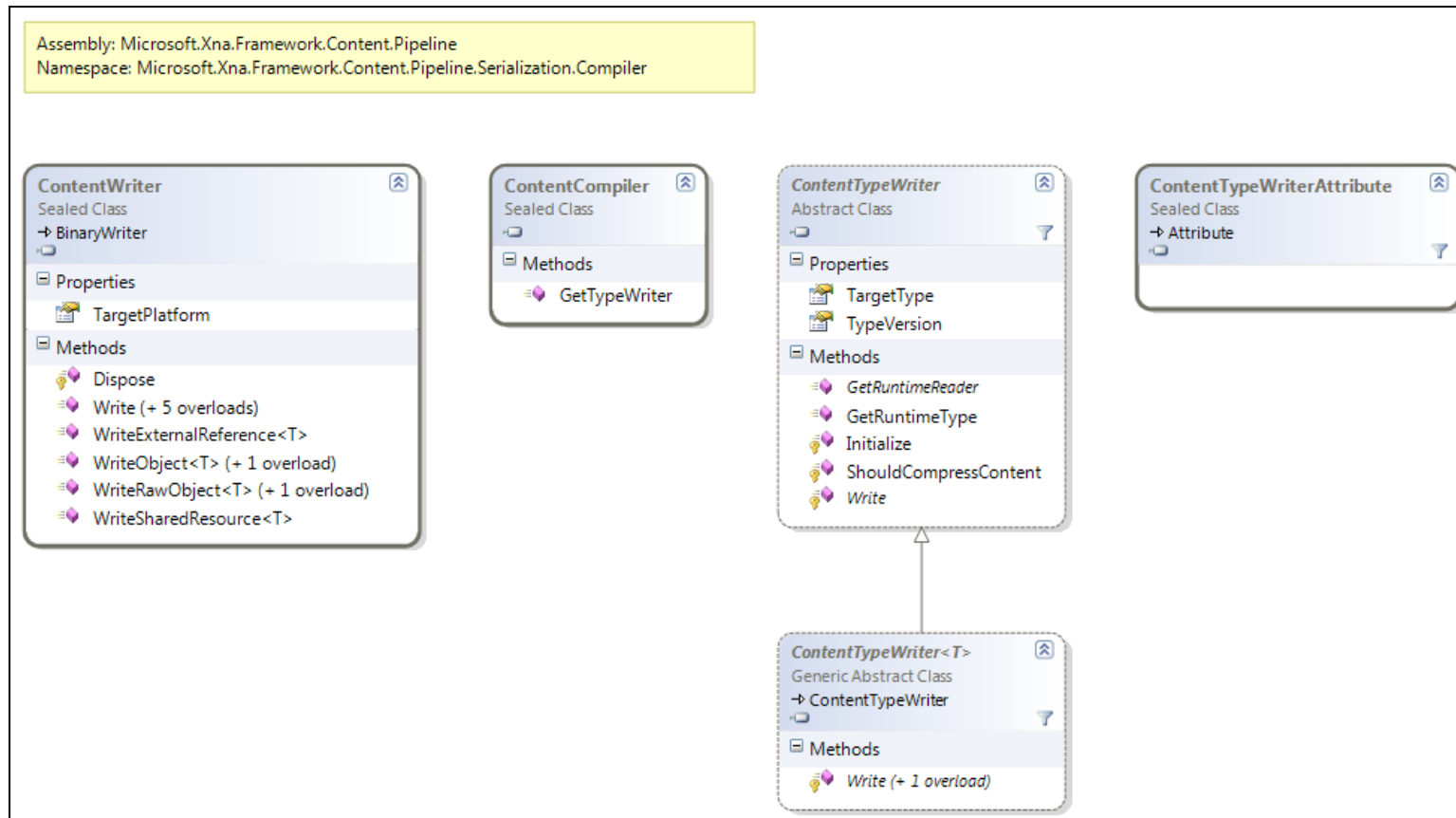


Figura Anexo 9 - Content.Pipeline.Serialization.Compiler

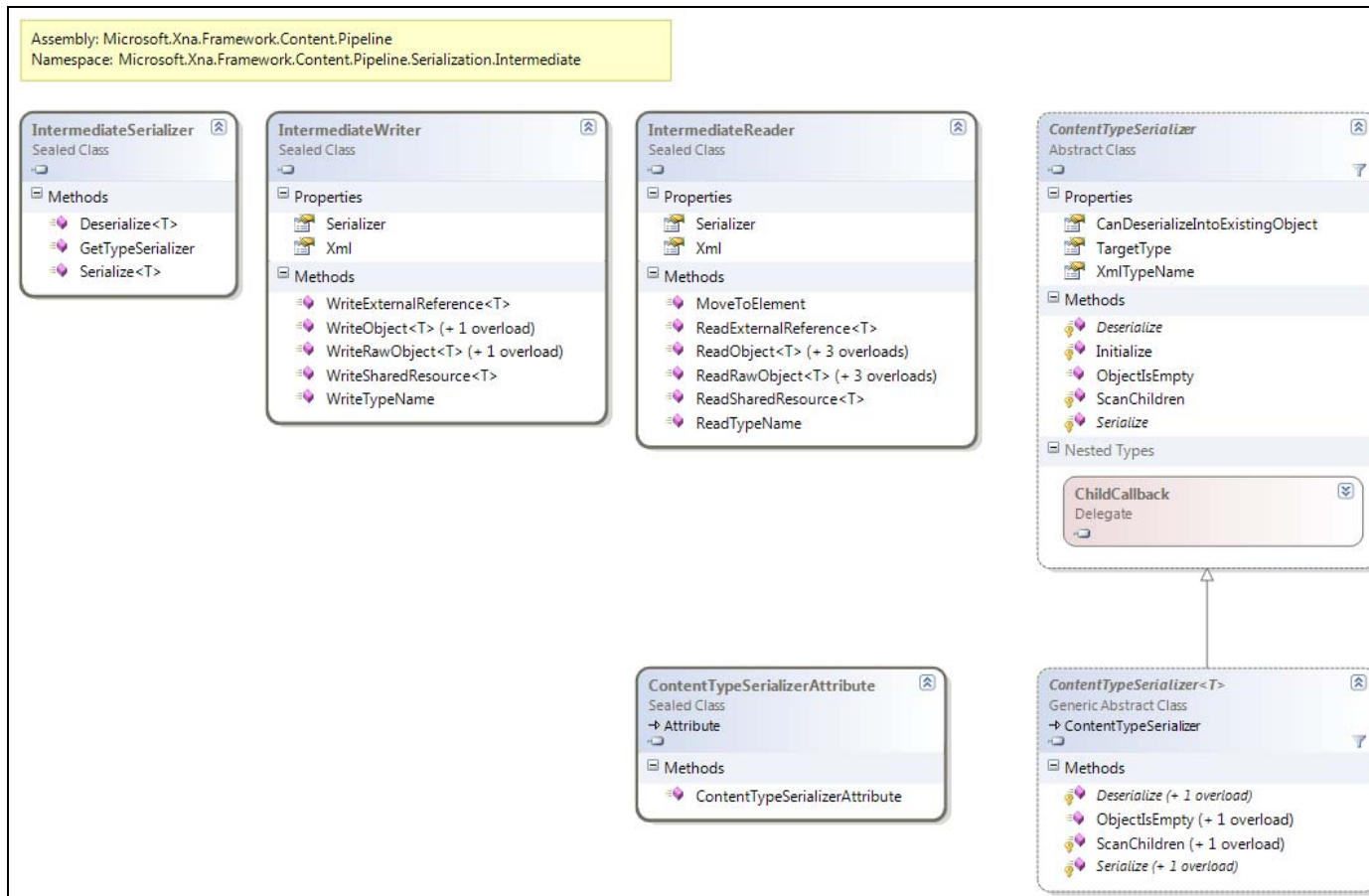
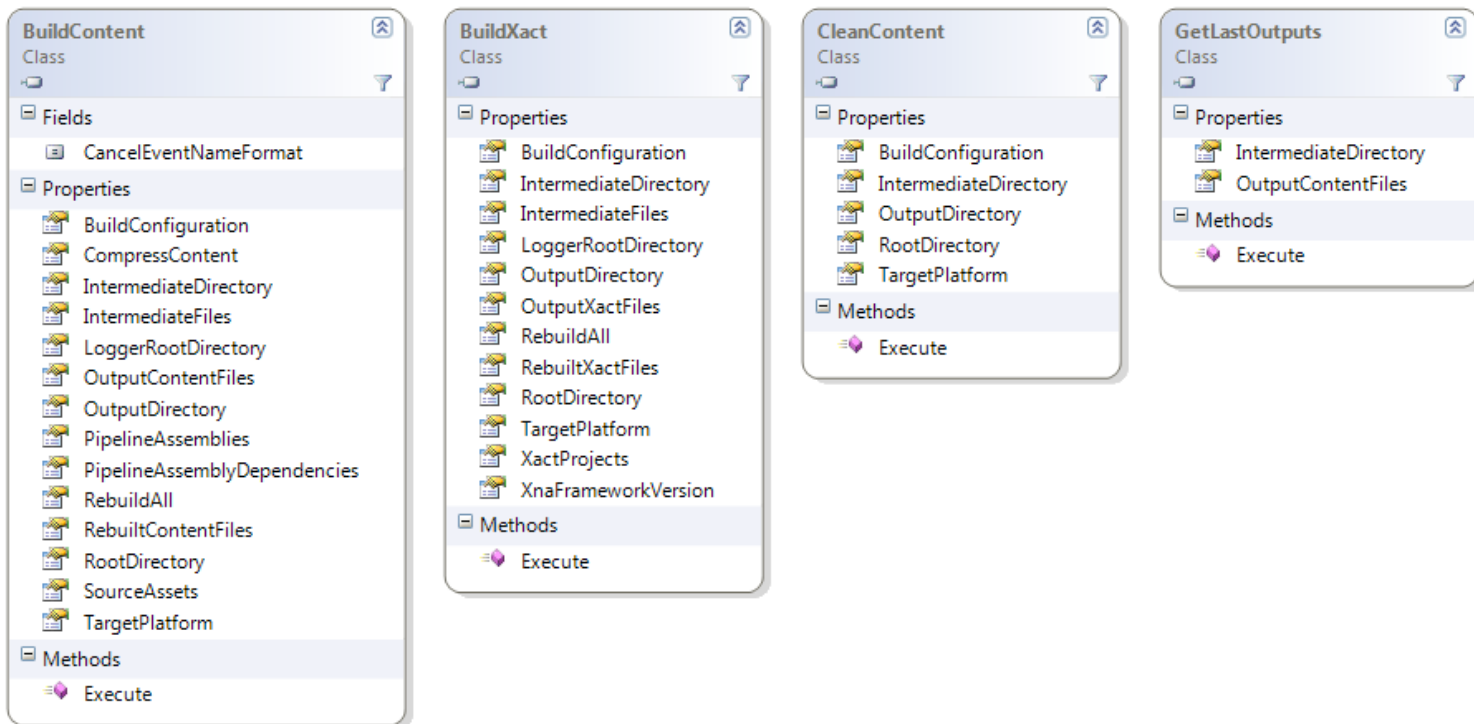


Figura Anexo 10 - Content.Pipeline.Serialization.Intermediate

Assembly: Microsoft.Xna.Framework.Content.Pipeline
Namespace: Microsoft.Xna.Framework.Content.Pipeline.Tasks



- BuildContent** (Class)
 - Fields
 - CancelEventNameFormat
 - Properties
 - BuildConfiguration
 - CompressContent
 - IntermediateDirectory
 - IntermediateFiles
 - LoggerRootDirectory
 - OutputContentFiles
 - OutputDirectory
 - PipelineAssemblies
 - PipelineAssemblyDependencies
 - RebuildAll
 - RebuiltContentFiles
 - RootDirectory
 - SourceAssets
 - TargetPlatform
 - Methods
 - Execute
- BuildXact** (Class)
 - Properties
 - BuildConfiguration
 - IntermediateDirectory
 - IntermediateFiles
 - LoggerRootDirectory
 - OutputDirectory
 - OutputXactFiles
 - RebuildAll
 - RebuiltXactFiles
 - RootDirectory
 - TargetPlatform
 - XactProjects
 - XnaFrameworkVersion
 - Methods
 - Execute
- CleanContent** (Class)
 - Properties
 - BuildConfiguration
 - IntermediateDirectory
 - OutputDirectory
 - RootDirectory
 - TargetPlatform
 - Methods
 - Execute
- GetLastOutputs** (Class)
 - Properties
 - IntermediateDirectory
 - OutputContentFiles
 - Methods
 - Execute

Figura Anexo 11 - Content.Pipeline.Tasks

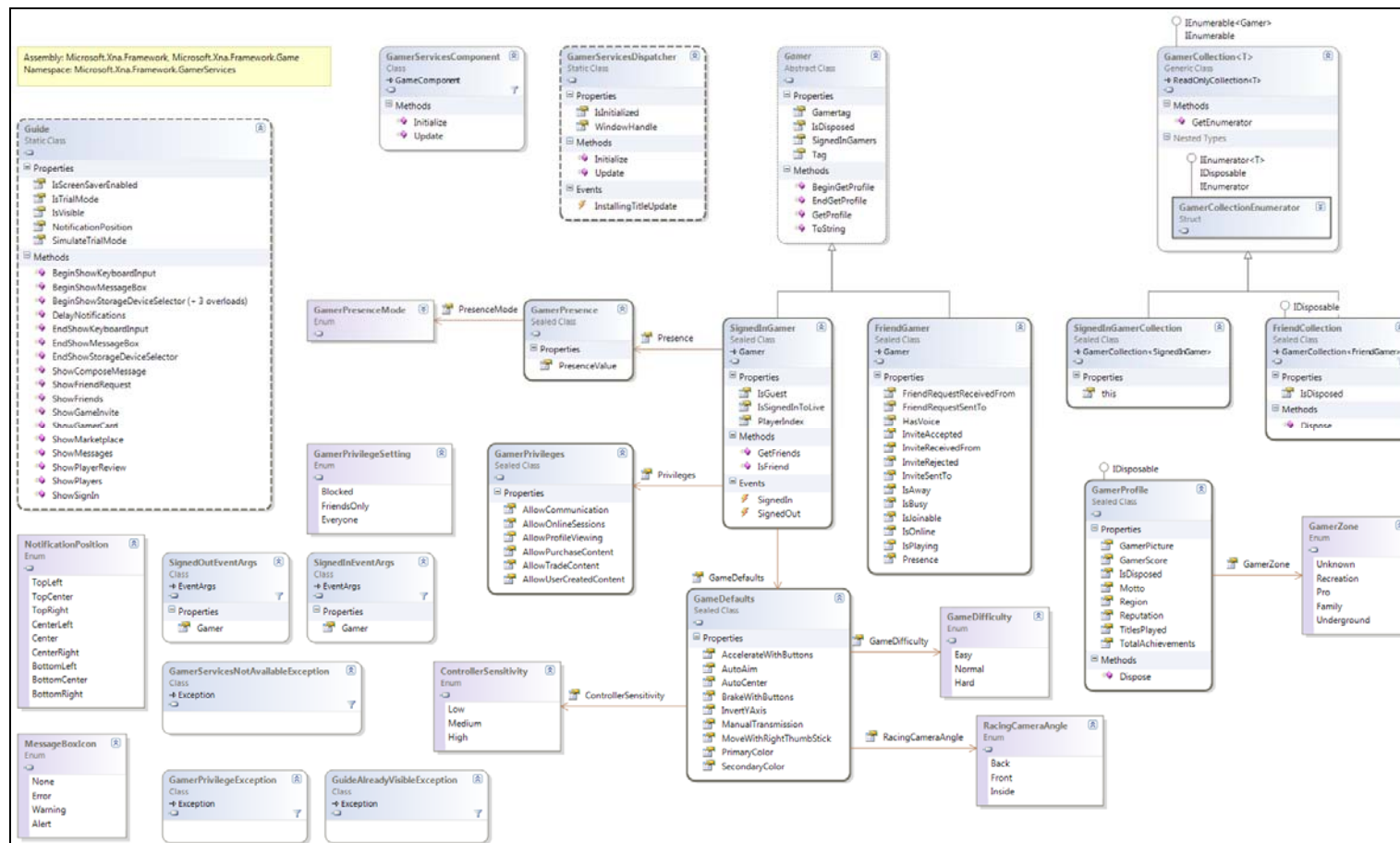


Figura Anexo 12 - Content.GameServices

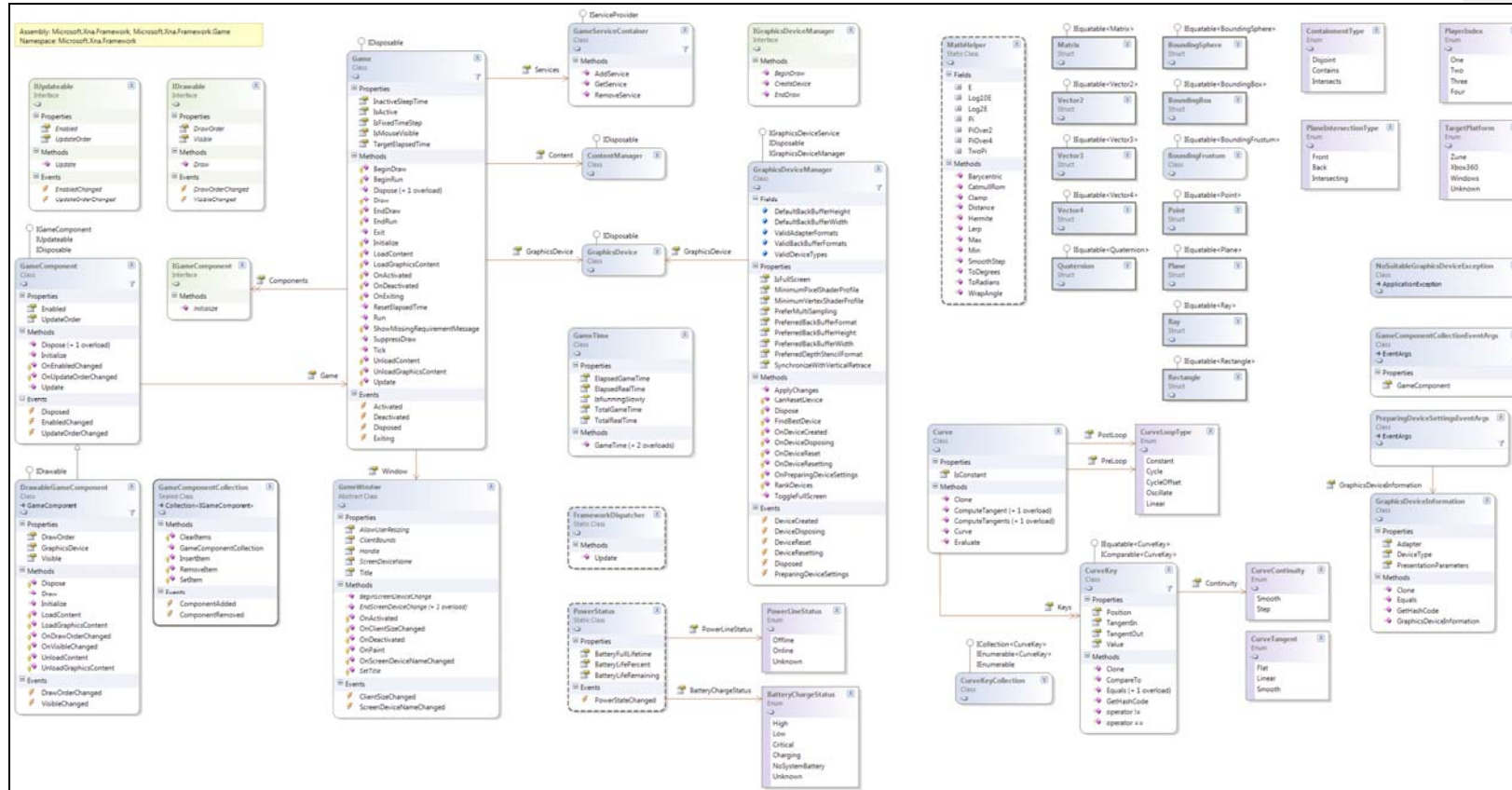


Figura Anexo 13 - Framework

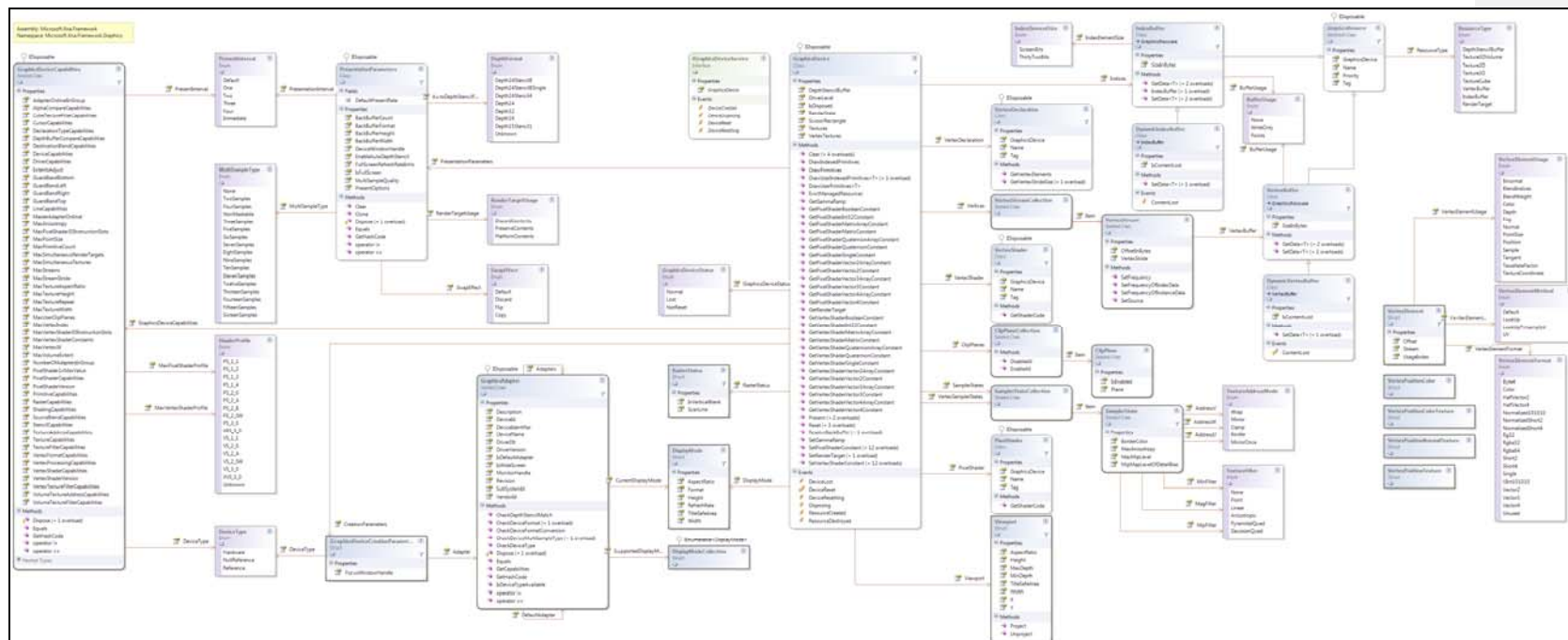


Figura Anexo 14 - Framework.Graphics

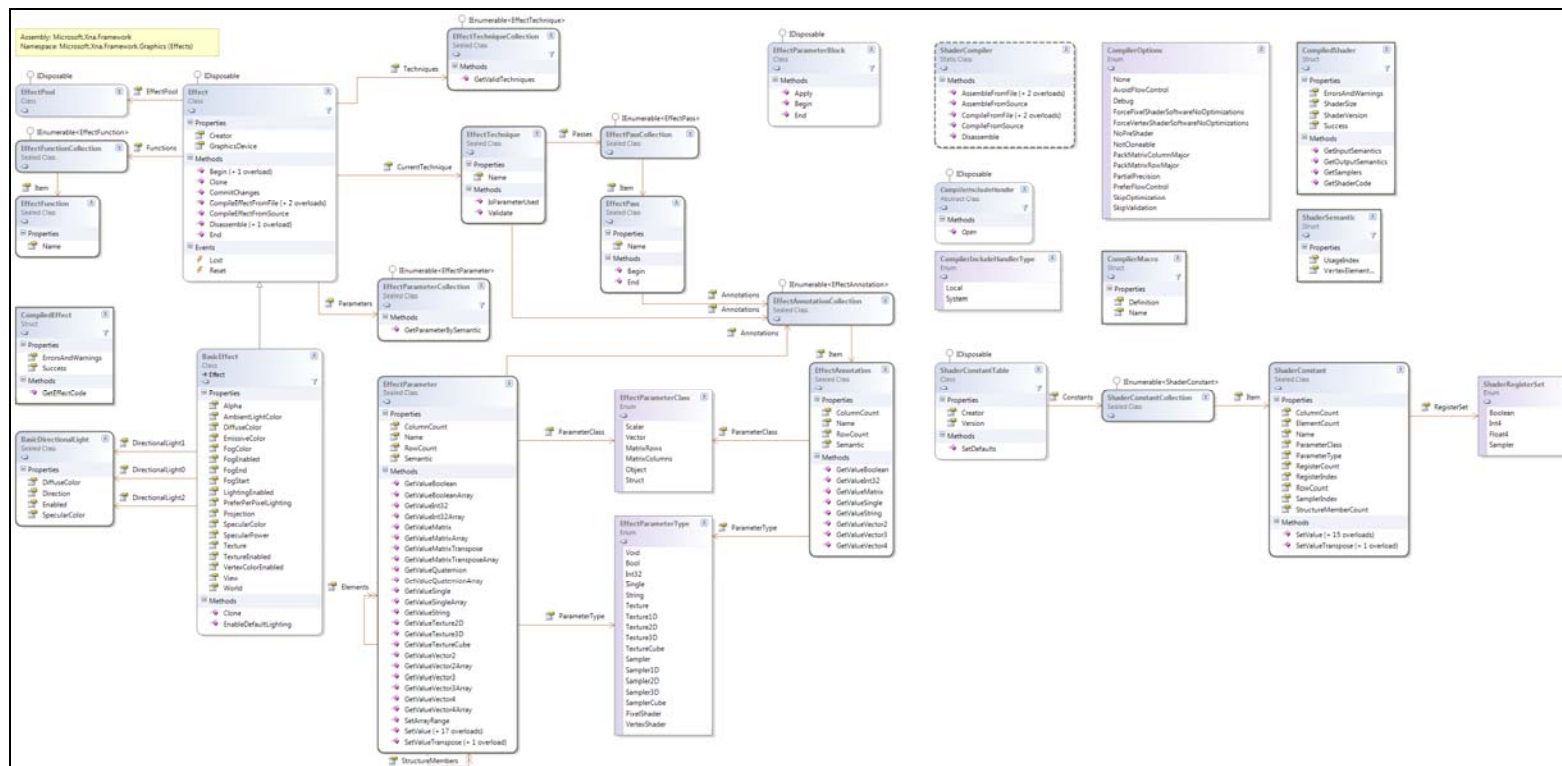


Figura Anexo 15 - Framework.Graphics (Effects)

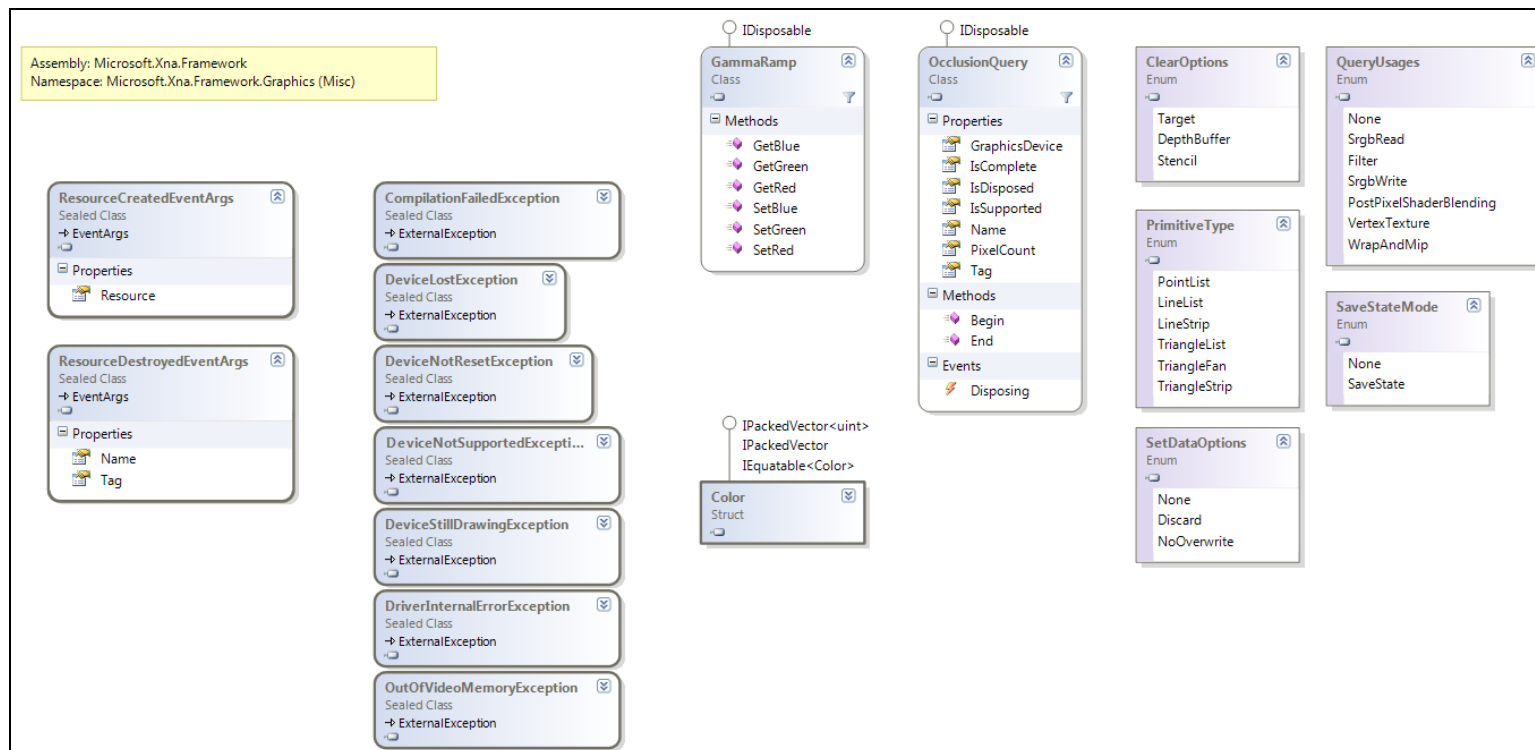


Figura Anexo 16 - Framework.Graphics (Misc)

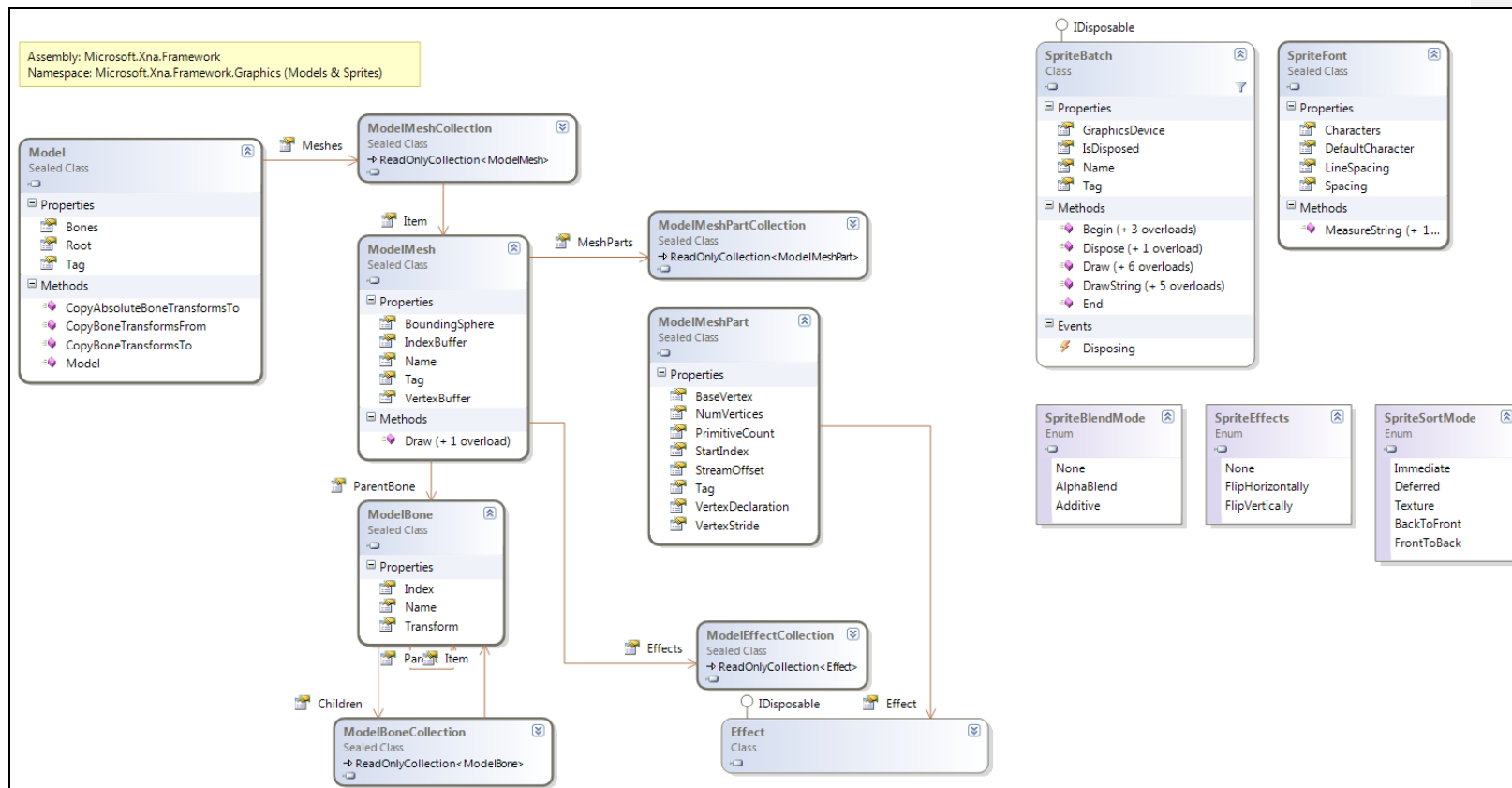


Figura Anexo 17 - Framework.Graphics (Models&Sprites)

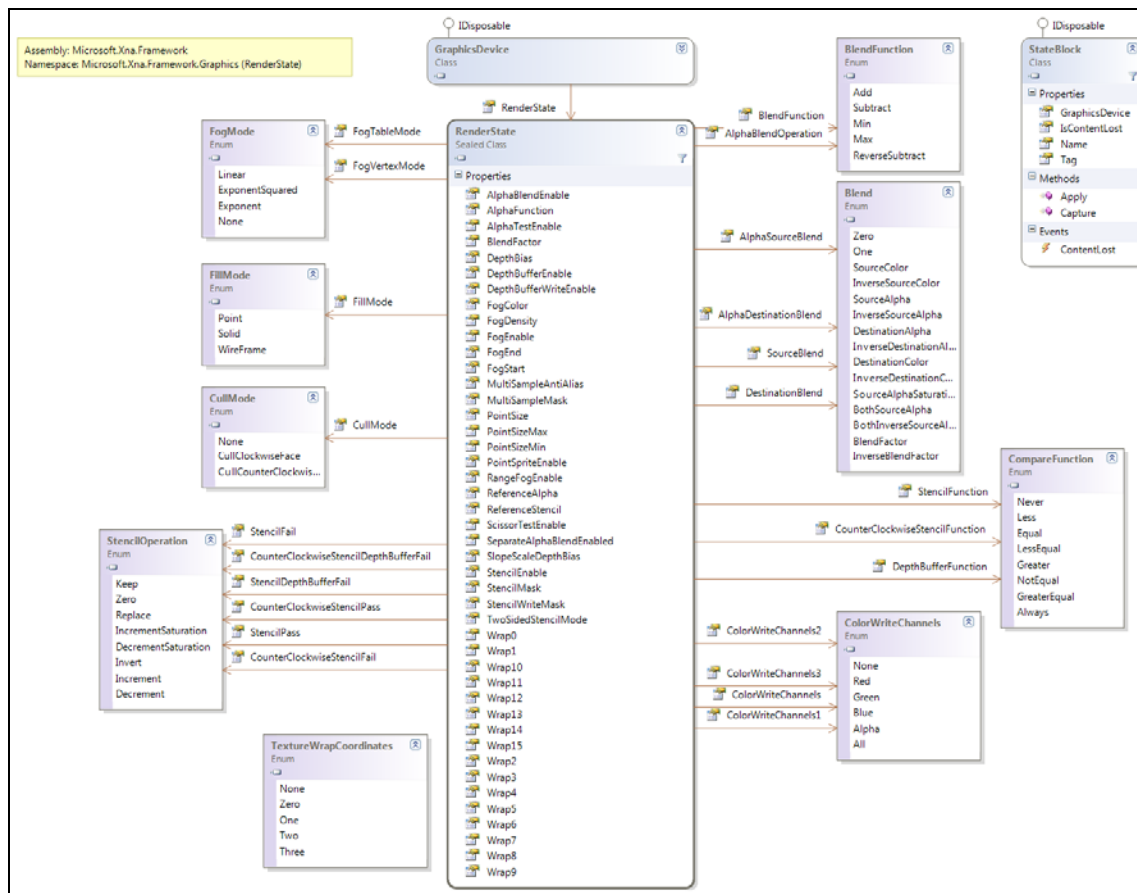


Figura Anexo 18 - Framework.Graphics (RenderState)

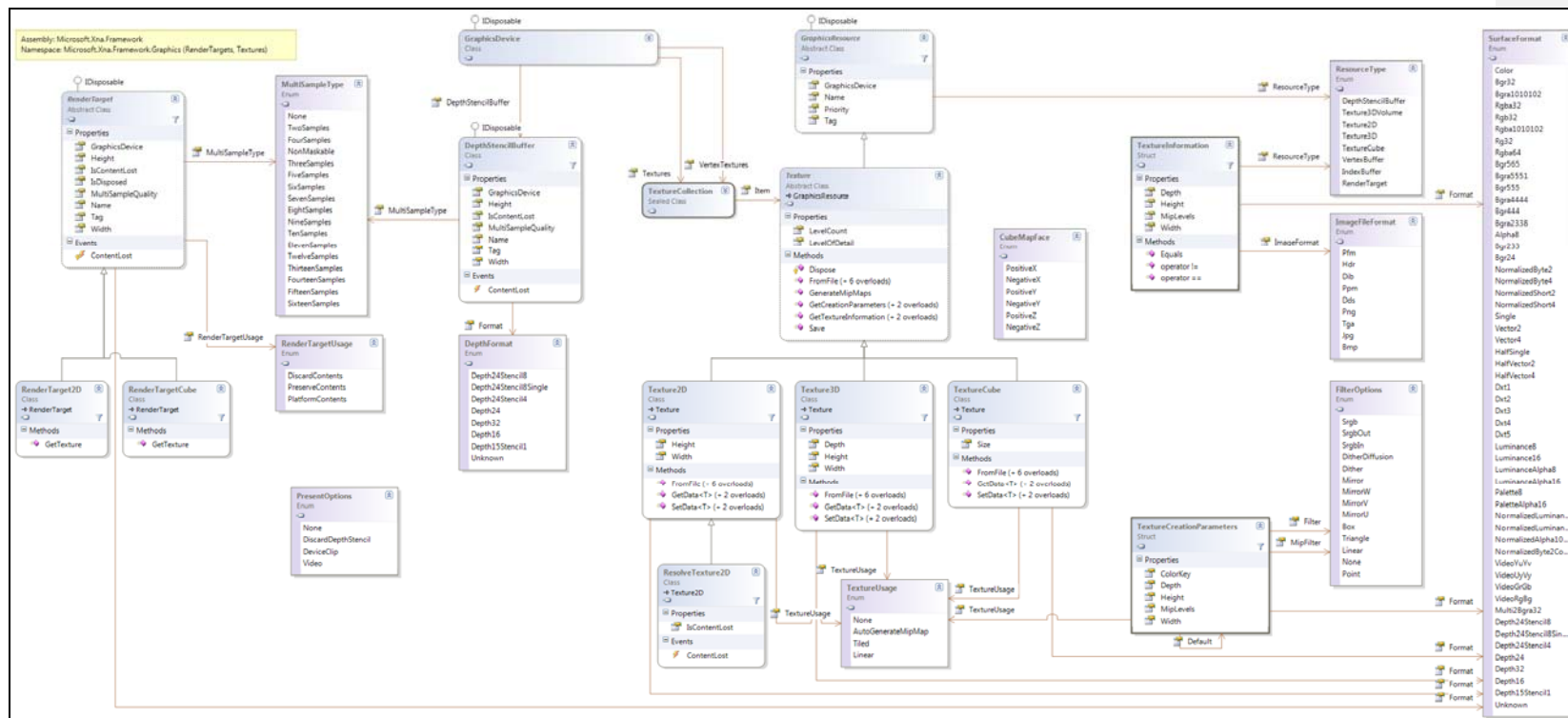


Figura Anexo 19 - Framework.Graphics (RenderTargets, Textures)

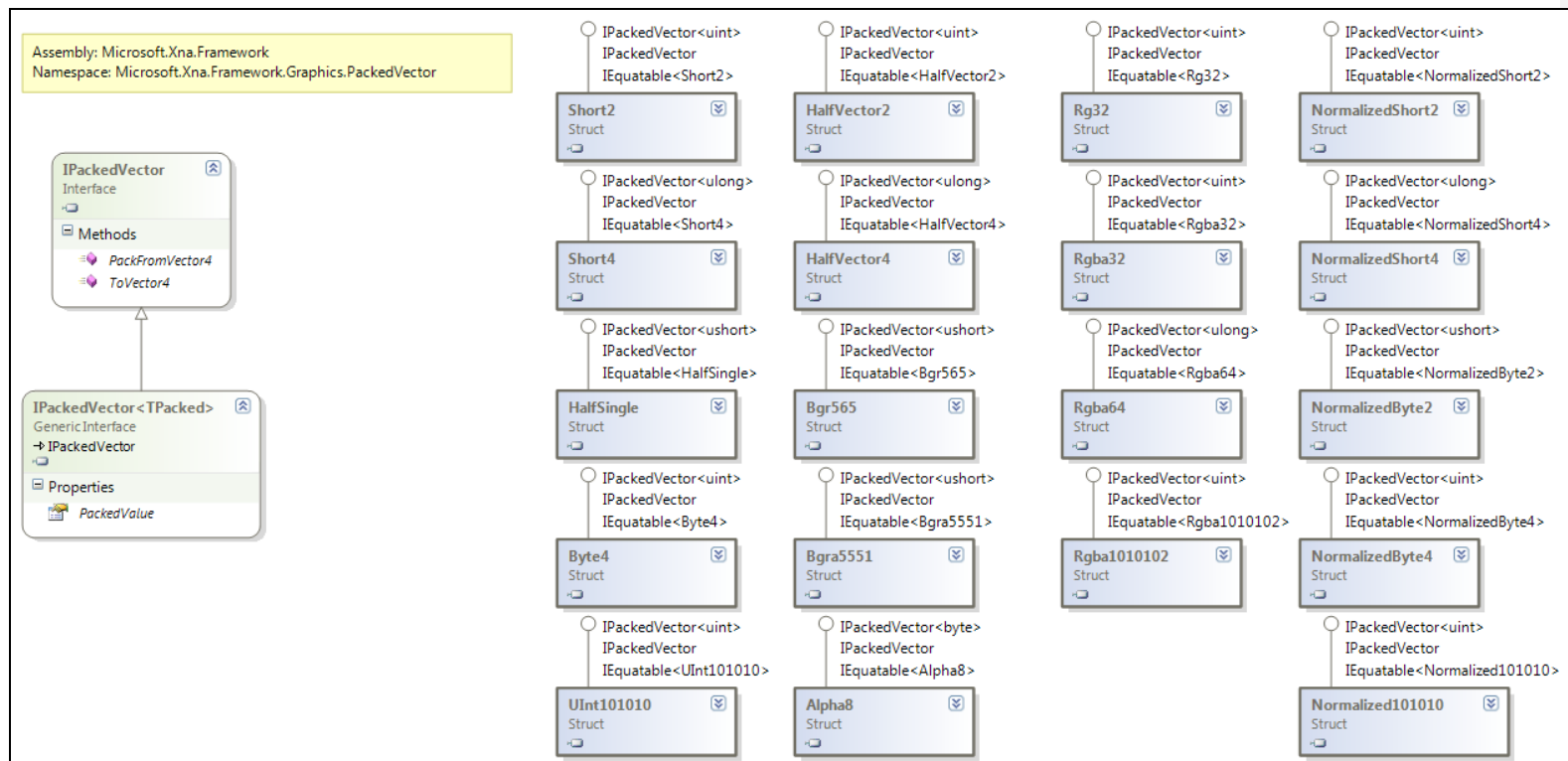


Figura Anexo 20 - Framework.Graphics.PackedVector

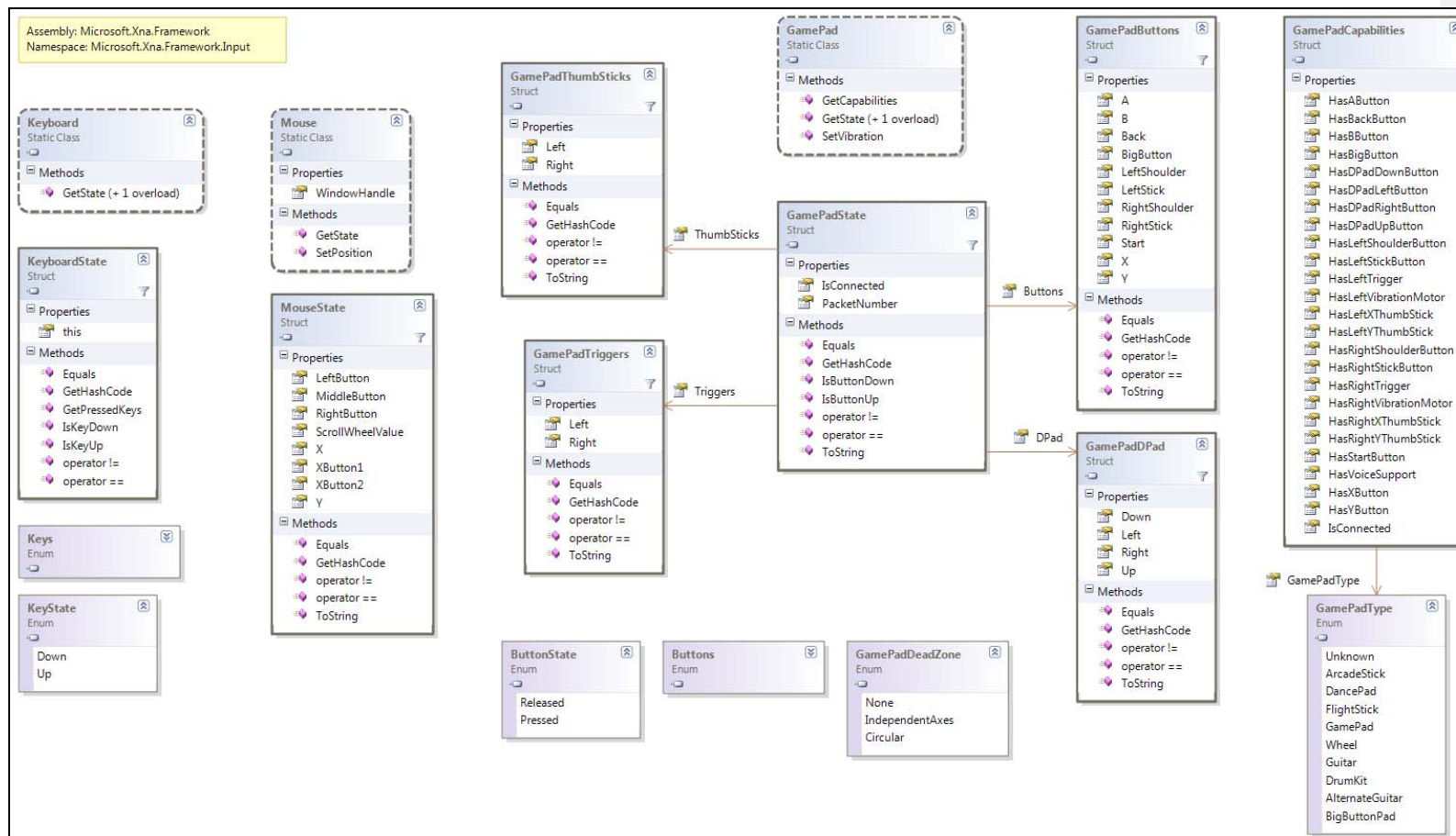


Figura Anexo 21 - Framework.Input

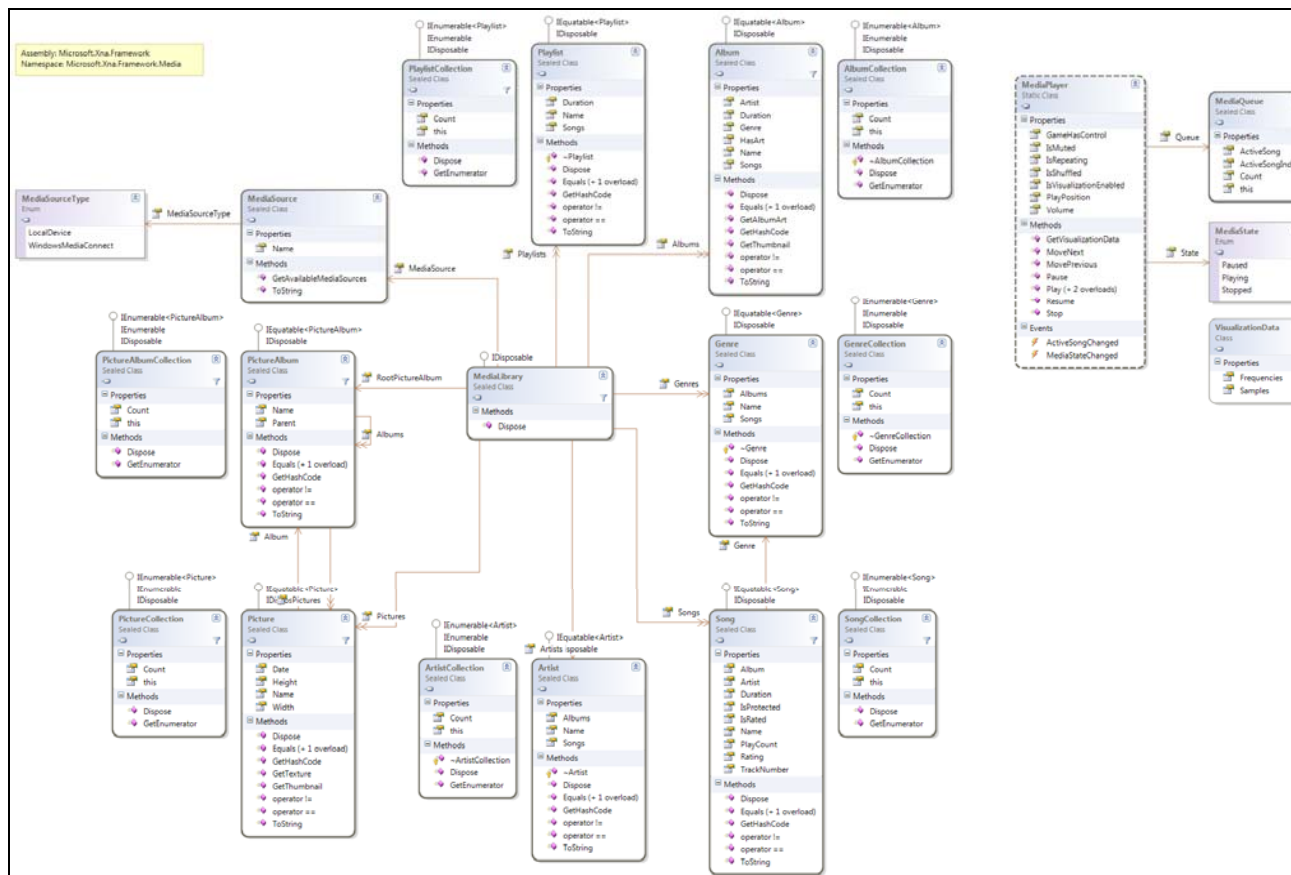


Figura Anexo 22 - Framework.Media

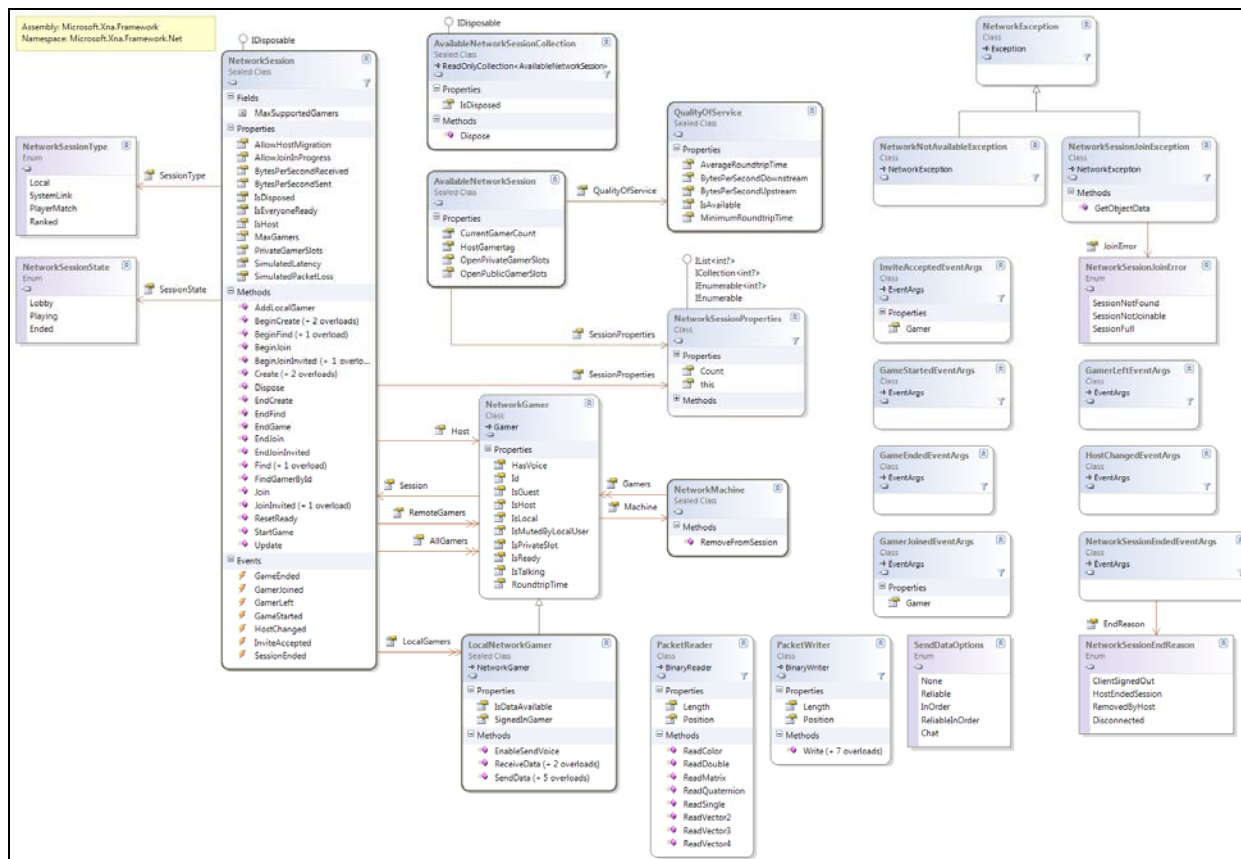


Figura Anexo 23 - Framework.Net

Navegación con Wiimote en entorno 3D mediante XNA

Alumno: Francisco Javier Labiano Jáuregui
Tutor: Eduardo Alfaro Larragueta

Pamplona, 15 de Noviembre de 2011

Introducción

■ Objetivos

- ❑ Integración Wiimote + Nunchuck con mundo 3D
- ❑ Entorno realista

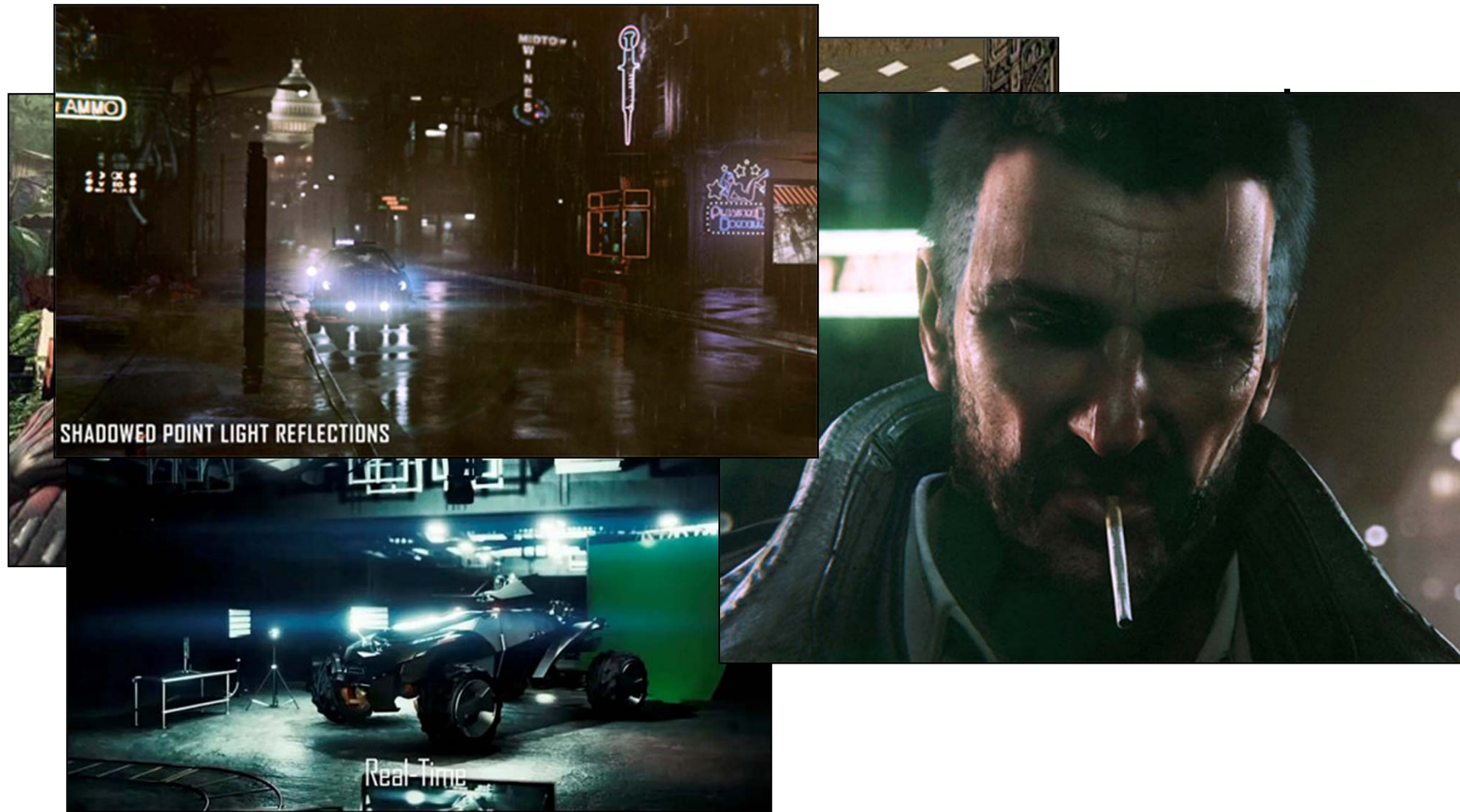
■ Requisitos

- ❑ Windows XP/Vista/7
- ❑ Tarjeta grafica → shaders 1.1 y DirectX 9.0c. Se recomienda shaders modelo 2.0.
- ❑ 256 MB de memoria RAM, aunque se recomienda tener entre 512 MB y 1 GB como mínimo.

■ Solución propuesta

- ❑ XNA + WiimoteLib
-

Introducción a los Frameworks y a los game engines

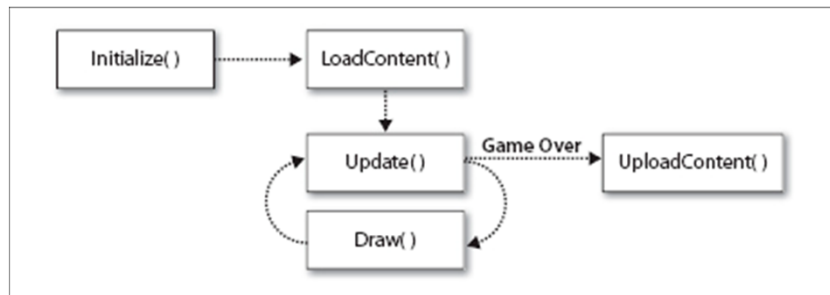


XNA - Introducción

- ¿Qué es?
 - XNA Framework
 - XNA Build
 - XNA Game Studio
- Game engines basados en XNA:
 - SlimDX
 - OpenTK
 - SunBurn GameEngine

XNA – Introducción II

■ Estructura básica



XNA – Introducción III

■ Ejemplos



Implementación de la solución I

- Objetivos:
 - ❑ Entorno 3D
 - ❑ Fácil manejo
 - ❑ Física creíble
 - ❑ Iluminación avanzada

Implementación de la solución II

■ Componentes necesarios:

□ Escena

■ 3D

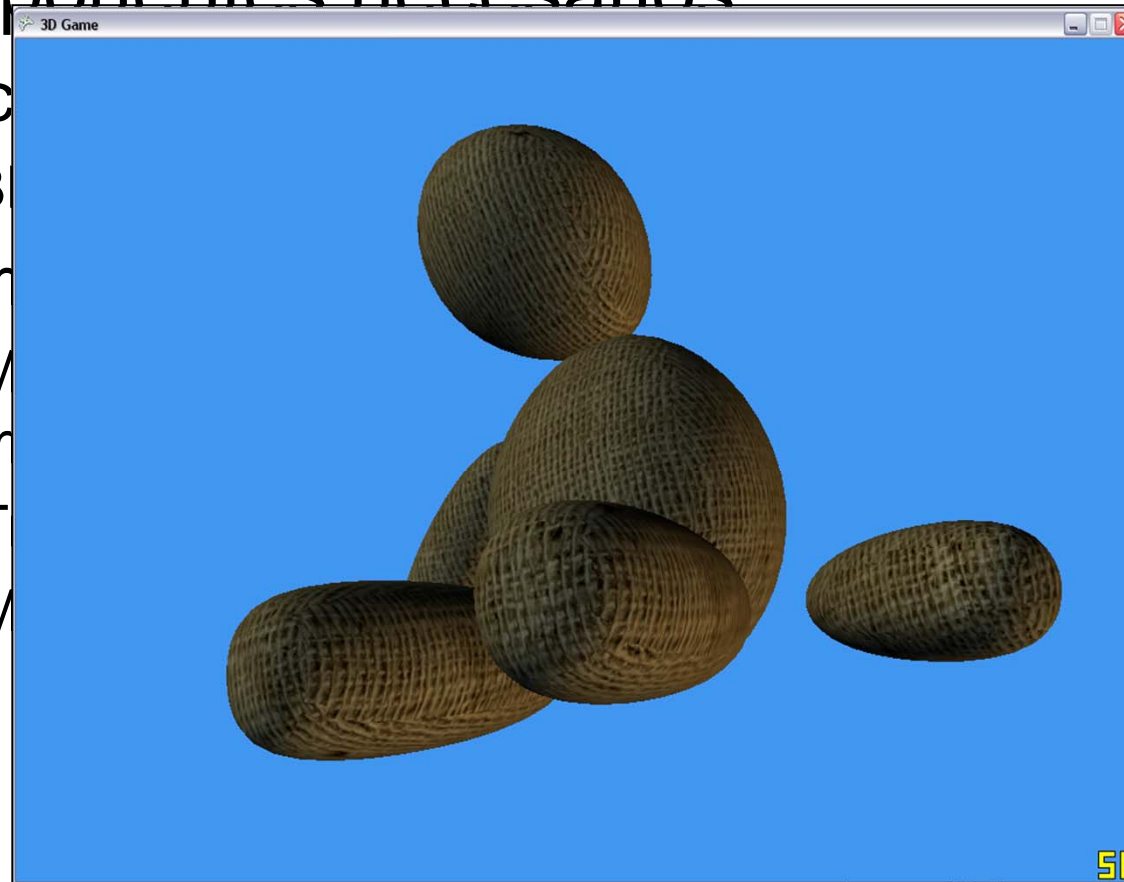
□ Cámara

■ M

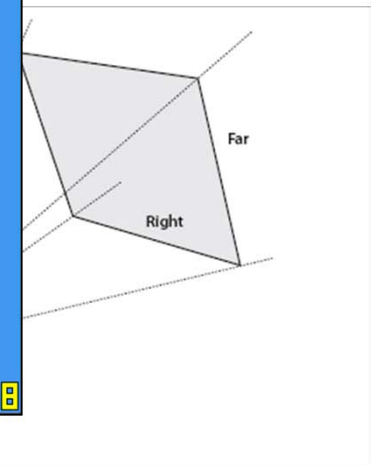
m

■ T

■ M



on) y

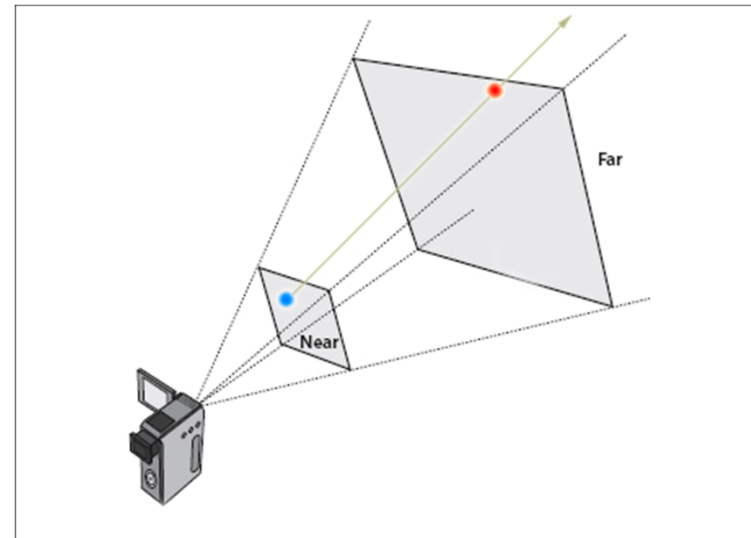


Implementación de la solución III

■ Cursor:

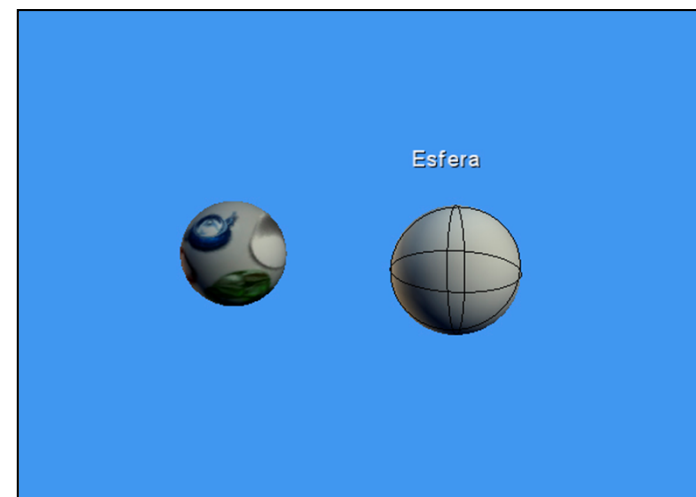
□ Reconocimiento:

- Método Unproject
- RayCasting



□ Selección:

- Se utiliza también el RayCasting



Implementación de la solución IV

- Física:

- Primera aproximación (Física propia)

- Fórmulas de física clásica

- Problemas:

- Se plantea la necesidad de una física de cuerpos rígidos

- Alcance del proyecto desmesurado

Implementación de la solución V

- ❑ Motor físico sencillo de cálculo de física de cuerpos rígidos
 - Carga de modelos con sus propiedades
 - ❑ Volumen
 - ❑ Masa
 - ❑ Densidad
 - ❑ ...
 - Cálculo del poliedro y más propiedades (caras, vértices, tensor de inercia, centro de gravedad, etc.)
 - Revisión de colisiones
- ❑ Problemas
 - Cálculo impreciso, límite de polígonos...

Implementación de la solución VI

- ❑ Motor físico BEPU Physics
 - Motor gratuito
 - Recientemente código liberado
 - Comunidad muy activa
 - Fluido y de fácil uso

Implementación de la solución VII

■ Iluminación:

□ Tipos de luz:

- Ambiental
- Difusa
- Especular



□ Motor de iluminación SunBurn

- Synapse Gaming
- Framework gratuito

Implementación de la solución VIII

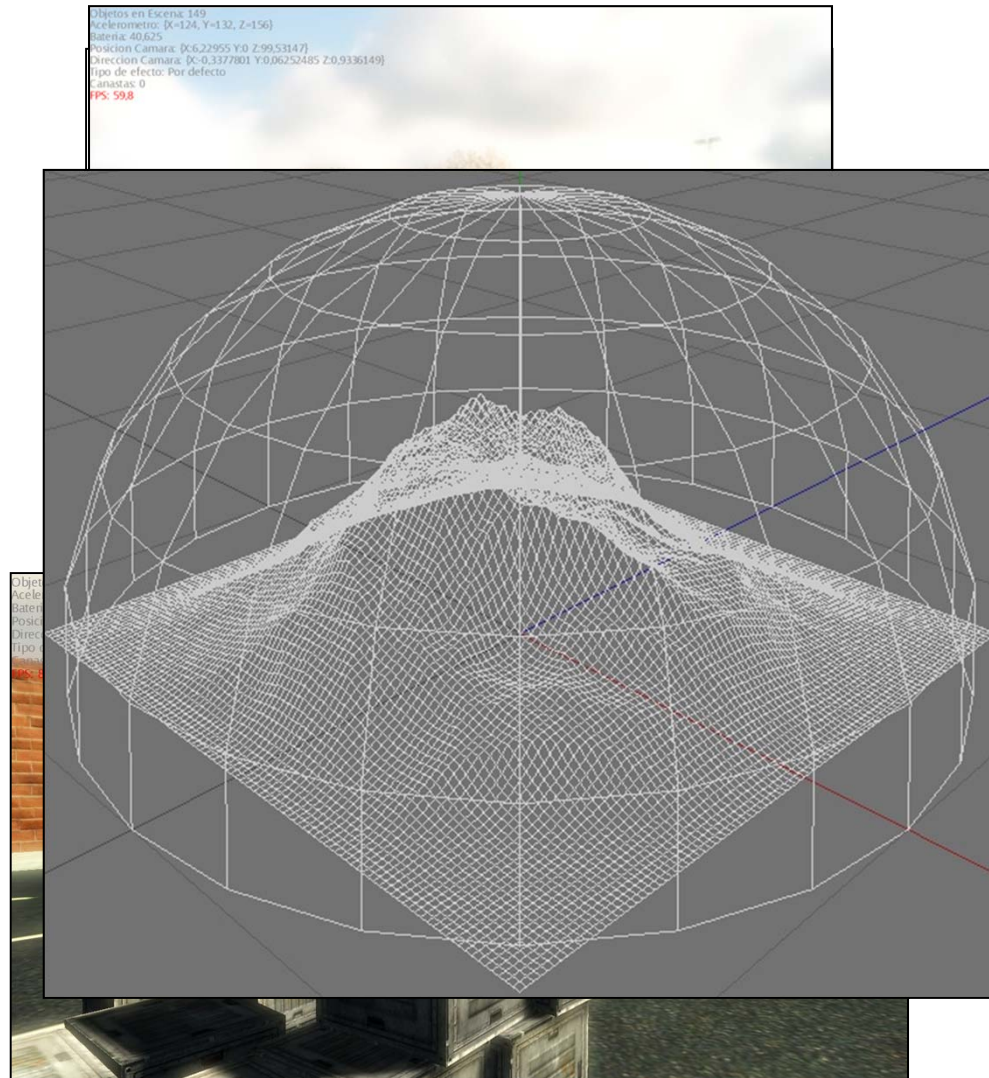
■ Ejemplos iluminación:



Implementación de la solución IX

■ SkyBox

- Definición
- Ejemplo
- Otras técnicas (SkyDome)



Implementación de la solución X

- Postprocesamiento:
 - Tipos:
 - Desenfoque de movimiento
 - Desenfoque gaussiano
 - Profundidad de campo
 - Lens Flare
 - Bloom
 - ...

Implementación de la solución XI

- Ejemplos:
 - **Desarrollo de un sistema de gestión de residuos**



Implementación de la solución XII

■ HLSL

□ Ejemp

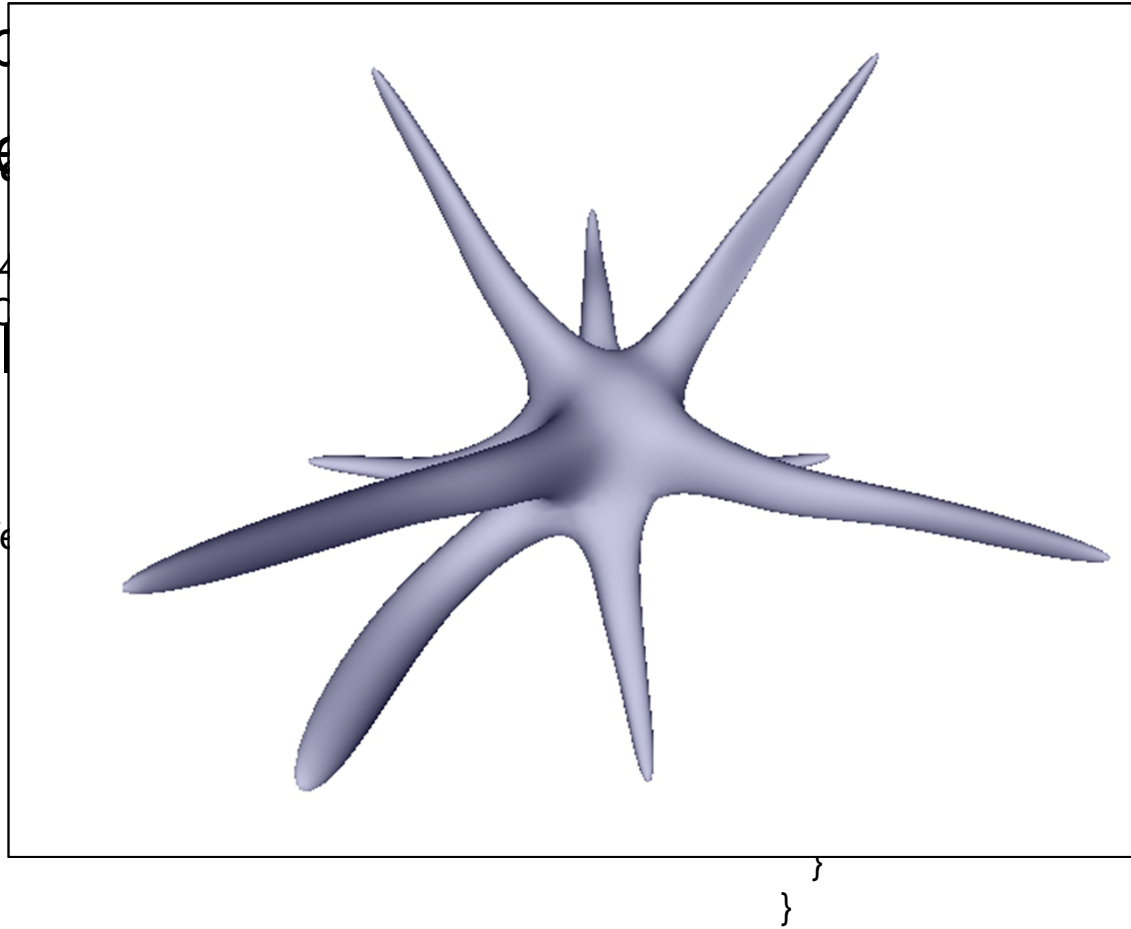
□ Vertex

float4x4
struct C
□ Pixel

};

OUT ve
{

}



COLOR

Vertex Shader

.8f;

float4(0.85, 0.75, 0.2,

AC:

Rasterizer

• Mueve todo a
coordenadas de pantalla

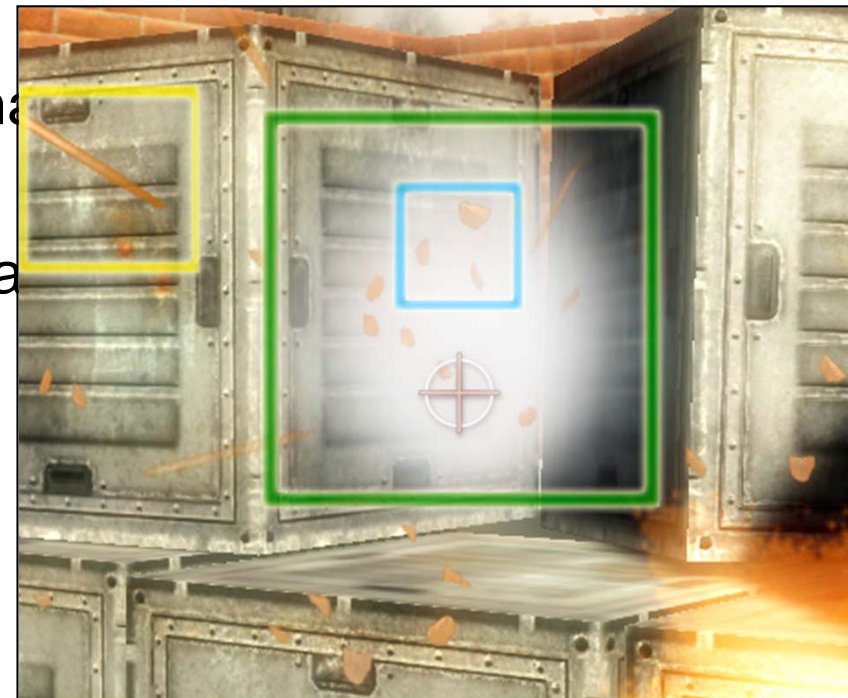
nt

mpile vs_1_1

mpile ps_1_1

Implementación de la solución XIII

- Partículas:
 - Librería DPSF:



Implementación de la solución XIV

■ Navegación:

□ Wiimote + Nunchuck

- Librería WiimoteLib de Brian Peek + Bluetooth
- Sencillez de uso

```
Wiimote wiimote = new Wiimote();
wiimote.Connect();

wiimote.SetRumble(true);
wiimote.SetLEDs(false, true, false, true);

WiimoteState wiimoteState = wiimote.WiimoteState;

float accelX = wiimoteState.AccelState.X;
float accelY = wiimoteState.AccelState.Y;
float accelZ = wiimoteState.AccelState.Z;
```

□ Acelerómetro

- Lanzamientos de balón y movimiento de la cámara basados en el acelerómetro.

Líneas futuras

- Creación de juego con objetivos claros
- Uso de otros dispositivos (Kinect)
- Implementación de física e iluminación propias

Conclusiones

- XNA provee un entorno para crear juegos de forma rápida, sencilla y portable a diferentes plataformas
- El mercado del futuro pasa por la multiplataforma. Los engines y frameworks soportan varios soportes (iPhone, iPad, PC, Android, Xbox,...)
- Personalmente ha captado mi atención por el mundo del desarrollo de videojuegos

Ruegos y preguntas
