UCLM
UNIVERSIDAD DE CASTILLA-LA MANCHA

2

Desarrollo de Videojuegos

Programación Gráfica

Carlos González Morcillo
Javier A. Albusac Jiménez
Sergio Pérez Camacho
Jorge López González
César Mora Castro

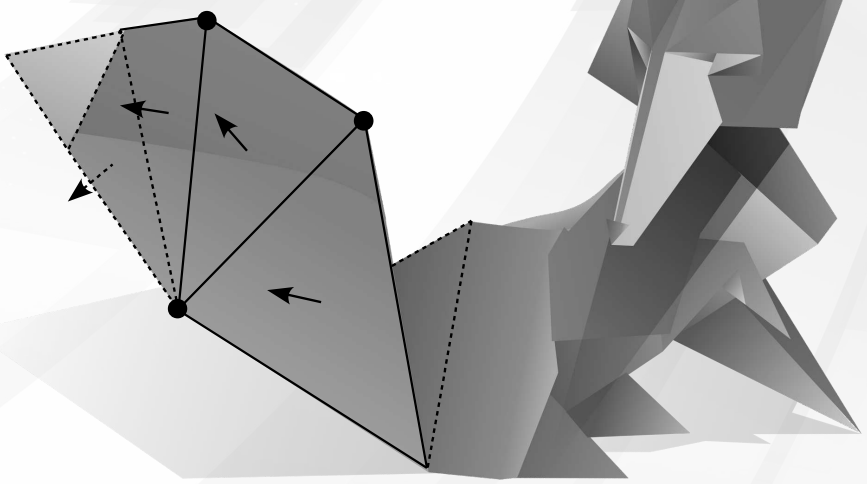
esi Escuela Superior de Informática

Paseo de la Universidad, 4
13071, Ciudad Real
Email: esi@uclm.es

2

Desarrollo de Videojuegos

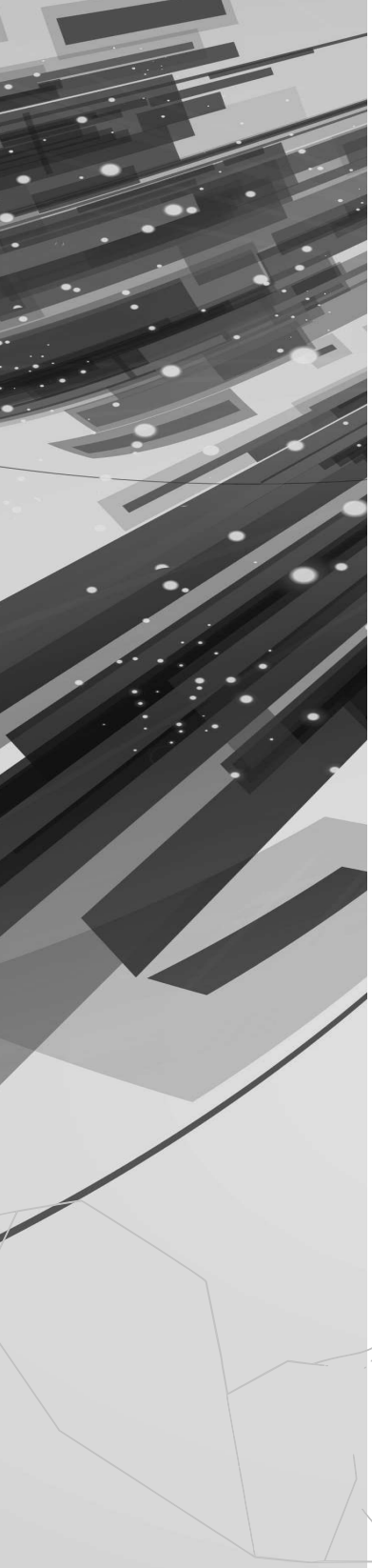
Programación Gráfica



Carlos González Morcillo
Javier A. Albusac Jiménez
Sergio Pérez Camacho
Jorge López González
César Mora Castro

Escuela Superior de Informática

Paseo de la Universidad, 4
13071, Ciudad Real
Email: esi@uclm.es



Título: Desarrollo de Videojuegos: Programación Gráfica
Autores: Carlos González Morcillo, Javier A. Albusac Jiménez, Sergio Pérez Camacho, Jorge López González, César Mora Castro
ISBN: 978-84-686-1058-0 (de la *Edición Física, a la venta en www.bubok.es*)
ISBN (C): 978-84-686-1056-6 *(ISBN de la Colección)*
Publica: Bubok (*Edición Física*) LibroVirtual.org (*Edición electrónica*)
Edita: David Vallejo Fernández y Carlos González Morcillo
Diseño: Carlos González Morcillo

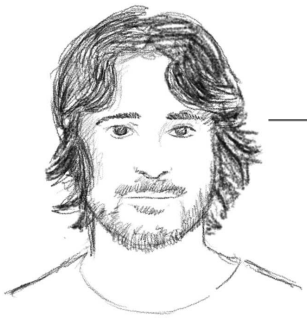
Este libro fue compuesto con LaTeX a partir de una plantilla de Carlos González Morcillo y Sergio García Mondaray. La portada y las entradillas fueron diseñadas con GIMP, Blender, InkScape y OpenOffice.

Autores

Carlos González (2007, Doctor Europeo en Informática, Universidad de Castilla-La Mancha) es Profesor Titular de Universidad e imparte docencia en la Escuela de Informática de Ciudad Real (UCLM) en asignaturas relacionadas con Informática Gráfica, Síntesis de Imagen Realista y Sistemas Operativos desde 2002. Actualmente, su actividad investigadora gira en torno a los Sistemas Multi-Agente, el Rendering Distribuido y la Realidad Aumentada.

Javier Albusac (2009, Doctor Europeo en Informática, Universidad de Castilla-La Mancha) es Profesor Ayudante Doctor e imparte docencia en la Escuela de Ingeniería Minera e Industrial de Almadén (EIMIA) en las asignaturas de Informática, Ofimática Aplicada a la Ingeniería y Sistemas de Comunicación en Edificios desde 2007. Actualmente, su actividad investigadora gira en torno a la Vigilancia Inteligente, Robótica Móvil y Aprendizaje Automático.

Sergio Pérez (2011, Ingeniero en Informática, Universidad de Castilla-La Mancha) trabaja como ingeniero consultor diseñando software de redes para Ericsson R&D. Sus intereses principales son GNU/Linux, las redes, los videojuegos y la realidad aumentada.



Jorge López (2011, Ingeniero en Informática por la UCLM y Máster en Diseño y Desarrollo de videojuegos por la UCM). Especializado en desarrollo 3D con C++ y OpenGL, y en el engine Unity 3D. Actualmente trabaja como programador en Minimal Drama Game Studio.

César Mora (2011, Ingeniero en Informática, Universidad de Casilla-La Mancha - UCLM) Tecnólogo en el grupo de investigación Oreto, desarrollando proyectos relacionados con Informática Gráfica, Visión Artificial y Realidad Aumentada.

Prefacio

Este libro forma parte de una colección de cuatro volúmenes dedicados al Desarrollo de Videojuegos. Con un perfil principalmente técnico, estos cuatro libros cubren los aspectos esenciales en programación de videojuegos:

1. **Arquitectura del Motor**. En este primer libro se estudian los aspectos esenciales del diseño de un motor de videojuegos, así como las técnicas básicas de programación y patrones de diseño.

2. **Programación Gráfica**. El segundo libro de la colección se centra en los algoritmos y técnicas de representación gráfica y optimizaciones en sistemas de despliegue interactivo.

3. **Técnicas Avanzadas**. En este tercer volumen se recogen ciertos aspectos avanzados, como estructuras de datos específicas, técnicas de validación y pruebas o simulación física.

4. **Desarrollo de Componentes**. El último libro está dedicado a ciertos componentes específicos del motor, como la Inteligencia Artificial, Networking, Sonido y Multimedia o técnicas avanzadas de Interacción.

Sobre este libro...

Este libro que tienes en tus manos es una ampliación y revisión de los apuntes del *Curso de Experto en Desarrollo de Videojuegos*, impartido en la Escuela Superior de Informática de Ciudad Real de la Universidad de Castilla-La Mancha. Puedes obtener más información sobre el curso, así como los resultados de los trabajos creados por los alumnos en la web del mismo: http://www.esi.uclm.es/videojuegos.

La versión electrónica de este manual (y del resto de libros de la colección) puede descargarse desde la web anterior. El libro «físico» puede adquirirse desde la página web de la editorial online *Bubok* en http://www.bubok.es.

Requisitos previos

Este libro tiene un público objetivo con un perfil principalmente técnico. Al igual que el curso del que surgió, está orientado a la capacitación de profesionales de la programación de videojuegos. De esta forma, este libro no está orientado para un público de perfil artístico (modeladores, animadores, músicos, etc) en el ámbito de los videojuegos.

Se asume que el lector es capaz de desarrollar programas de nivel medio en C y C++. Aunque se describen algunos aspectos clave de C++ a modo de resumen, es recomendable refrescar los conceptos básicos con alguno de los libros recogidos en la bibliografía del curso. De igual modo, se asume que el lector tiene conocimientos de estructuras de datos y algoritmia. El libro está orientado principalmente para titulados o estudiantes de últimos cursos de Ingeniería en Informática.

Programas y código fuente

El código de los ejemplos del libro pueden descargarse en la siguiente página web: http://www.esi.uclm.es/videojuegos. Salvo que se especifique explícitamente otra licencia, todos los ejemplos del libro se distribuyen bajo GPLv3.

Agradecimientos

Los autores del libro quieren agradecer en primer lugar a los alumnos de la primera edición del *Curso de Experto en Desarrollo de Videojuegos* por su participación en el mismo y el excelente ambiente en las clases, las cuestiones planteadas y la pasión demostrada en el desarrollo de todos los trabajos.

De igual modo, se quiere reflejar el agradecimiento especial al personal de administración y servicios de la Escuela Superior de Informática, por su soporte, predisposición y ayuda en todos los caprichosos requisitos que planteábamos a lo largo del curso.

Por otra parte, este agradecimiento también se hace extensivo a la Escuela de Informatica de Ciudad Real y al Departamento de Tecnologías y Sistema de Información de la Universidad de Castilla-La Mancha.

Finalmente, los autores desean agradecer su participación a los colaboradores de esta primera edición: *Indra Software Labs*, la asociación de desarrolladores de videojuegos *Stratos* y a *Libro Virtual*.

Resumen

El objetivo de este módulo titulado “Programación Gráfica” del *Curso de Experto en Desarrollo de Videojuegos* es cubrir los aspectos esenciales relativos al desarrollo de un motor gráfico interactivo.

En este contexto, el presente módulo cubre aspectos esenciales y básicos relativos a los fundamentos del desarrollo de la parte gráfica, como por ejemplo el *pipeline* gráfico, como elemento fundamental de la arquitectura de un motor de juegos, las bases matemáticas, las APIs de programación gráfica, el uso de materiales y texturas, la iluminación o los sistemas de partículas.

Así mismo, el presente módulo también discute aspectos relativos a la exportación e importación de datos, haciendo especial hincapié en los formatos existentes para tratar con información multimedia.

Finalmente, se pone de manifiesto la importancia de elementos como los *shaders*, con el objetivo de dotar de más realismo gráfico a los videojuegos, y la necesidad de llevar a cabo optimizaciones tanto en escenarios interiores como exteriores.

Índice general

1. Fundamentos de Gráficos Tridimensionales **1**

1.1. Introducción 1

1.2. El Pipeline Gráfico 4

1.2.1. Etapa de Aplicación 5

1.2.2. Etapa de Geometría 5

1.2.3. Etapa Rasterización 8

1.2.4. Proyección en Perspectiva 9

1.3. Implementación del Pipeline en GPU 10

1.3.1. Vertex Shader 12

1.3.2. Geometry Shader 12

1.3.3. Pixel Shader 12

1.4. Arquitectura del motor gráfico 13

1.5. Casos de Estudio 14

1.6. Introducción a OGRE 15

1.6.1. Arquitectura General 18

1.6.2. Instalación 23

1.7. Hola Mundo en OGRE 24

2. Matemáticas para Videojuegos **29**

2.1. Puntos, Vectores y Coordenadas 29

2.1.1. Puntos 30

2.1.2. Vectores 31

2.2. Transformaciones Geométricas 34

2.2.1. Representación Matricial 35
2.2.2. Transformaciones Inversas 37
2.2.3. Composición 38
2.3. Perspectiva: Representación Matricial 39
2.4. Cuaternios 41
2.4.1. Suma y Multiplicación 43
2.4.2. Inversa 43
2.4.3. Rotación empleando Cuaternios 44
2.5. Interpolación Lineal y Esférica 44
2.6. El Módulo Math en OGRE 45

3. Grafos de Escena **49**
3.1. Justificación 49
3.1.1. Operaciones a Nivel de Nodo 50
3.2. El Gestor de Escenas de OGRE 52
3.2.1. Creación de Objetos 52
3.2.2. Transformaciones 3D 54
3.2.3. Espacios de transformación 56

4. Recursos Gráficos y Sistema de Archivos **59**
4.1. Formatos de Especificación 59
4.1.1. Introducción 59
4.1.2. Recursos de gráficos 3D: formatos y requerimientos de almacenamiento 60
4.1.3. Casos de Estudio 66
4.2. Exportación y Adaptación de Contenidos 79
4.2.1. Instalación del exportador de Ogre en Blender . . . 81
4.2.2. Creación de un modelo en Blender 81
4.2.3. Aplicación de texturas mediante UV Mapping . . . 81
4.2.4. Exportación del objeto en formato Ogre XML . . . 85
4.2.5. Carga del objeto en una aplicación Ogre 87
4.3. Procesamiento de Recursos Gráficos 89
4.3.1. Ejemplo de uso 92
4.4. Gestión de Recursos y Escena 94
4.4.1. Recursos empaquetados 95
4.4.2. Gestión del ratón 96

4.4.3. Geometría Estática 97
4.4.4. Queries 98
4.4.5. Máscaras 101

5. APIS de Gráficos 3D **103**
5.1. Introducción 103
5.2. Modelo Conceptual 105
5.2.1. Cambio de Estado 105
5.2.2. Dibujar Primitivas 106
5.3. Pipeline de OpenGL 107
5.3.1. Transformación de Visualización 107
5.3.2. Transformación de Modelado 108
5.3.3. Transformación de Proyección 109
5.3.4. Matrices 109
5.3.5. Dos ejemplos de transformaciones jerárquicas . . 111
5.4. Ejercicios Propuestos 114

6. Gestión Manual OGRE 3D **115**
6.1. Inicialización Manual 115
6.1.1. Inicialización 116
6.1.2. Carga de Recursos 119
6.1.3. FrameListener 119
6.2. Uso de OIS 121
6.2.1. Uso de Teclado y Ratón 122
6.3. Creación manual de Entidades 125
6.4. Uso de Overlays 126

7. Materiales y Texturas **131**
7.1. Introducción 131
7.2. Modelos de Sombreado 133
7.3. Mapeado de Texturas 134
7.4. Materiales en Ogre 136
7.4.1. Composición 137
7.4.2. Ejemplo de Materiales 137
7.5. Mapeado UV en Blender 140
7.5.1. Costuras 144

7.6. Ejemplos de Materiales en Ogre 145
7.7. Render a Textura 148
7.7.1. Texture Listener 151
7.7.2. Espejo (Mirror) 152

8. Partículas y Billboards **155**
8.1. Fundamentos 155
8.1.1. Billboards 155
8.1.2. Sistemas de partículas 159
8.2. Uso de Billboards 160
8.2.1. Tipos de Billboard 162
8.2.2. Aplicando texturas 163
8.3. Uso de Sistemas de Partículas 164
8.3.1. Emisores 165
8.3.2. Efectores 165
8.3.3. Ejemplos de Sistemas de Partículas 166

9. Iluminación **169**
9.1. Introducción 169
9.2. Tipos de Fuentes de Luz 170
9.3. Sombras Estáticas Vs Dinámicas 171
9.3.1. Sombras basadas en Stencil Buffer 172
9.3.2. Sombras basadas en Texturas 174
9.4. Ejemplo de uso 176
9.5. Mapas de Iluminación 178
9.6. Ambient Occlusion 180
9.7. Radiosidad 181

10 Animación **187**
10.1 Introducción 187
10.1.1 Animación Básica 188
10.1.2 Animación de Alto Nivel 189
10.2 Animación en Ogre 191
10.2.1 Animación Keyframe 191
10.2.2 Controladores 192
10.3 Exportación desde Blender 192

10.4 Mezclado de animaciones 197

11. Exportación y Uso de Datos de Intercambio 203

11.1 Introducción 203

12. Shaders 211

12.1 Un poco de historia 211

12.1.1 ¿Y qué es un Shader? 213

12.2 Pipelines Gráficos 214

12.2.1 ¿Por qué un pipeline gráfico? 214

12.2.2 Fixed-Function Pipeline 215

12.2.3 Programmable-Function Pipeline 218

12.3 Tipos de Shader 218

12.4 Aplicaciones de los Shader 221

12.4.1 Vertex Skinning 221

12.4.2 Vertex Displacement Mapping 221

12.4.3 Screen Effects 221

12.4.4 Light and Surface Models 222

12.4.5 Non-Photorealistic Rendering 222

12.5 Lenguajes de Shader 223

12.6 Desarrollo de shaders en Ogre 224

12.6.1 Poniendo a punto el entorno 224

12.6.2 Primer Shader 225

12.6.3 Comprobando la interpolación del color 229

12.6.4 Usando una textura 231

12.6.5 Jugando con la textura 233

12.6.6 Jugando con los vértices 236

12.6.7 Iluminación mediante shaders 239

13. Optimización de interiores 241

13.1 Introducción 241

13.2 Técnicas y Algoritmos 242

13.2.1 Algoritmos basados en Oclusores 242

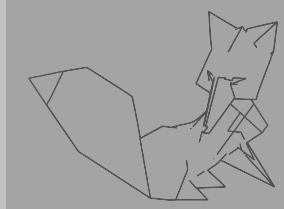
13.2.2 Algoritmo BSP 243

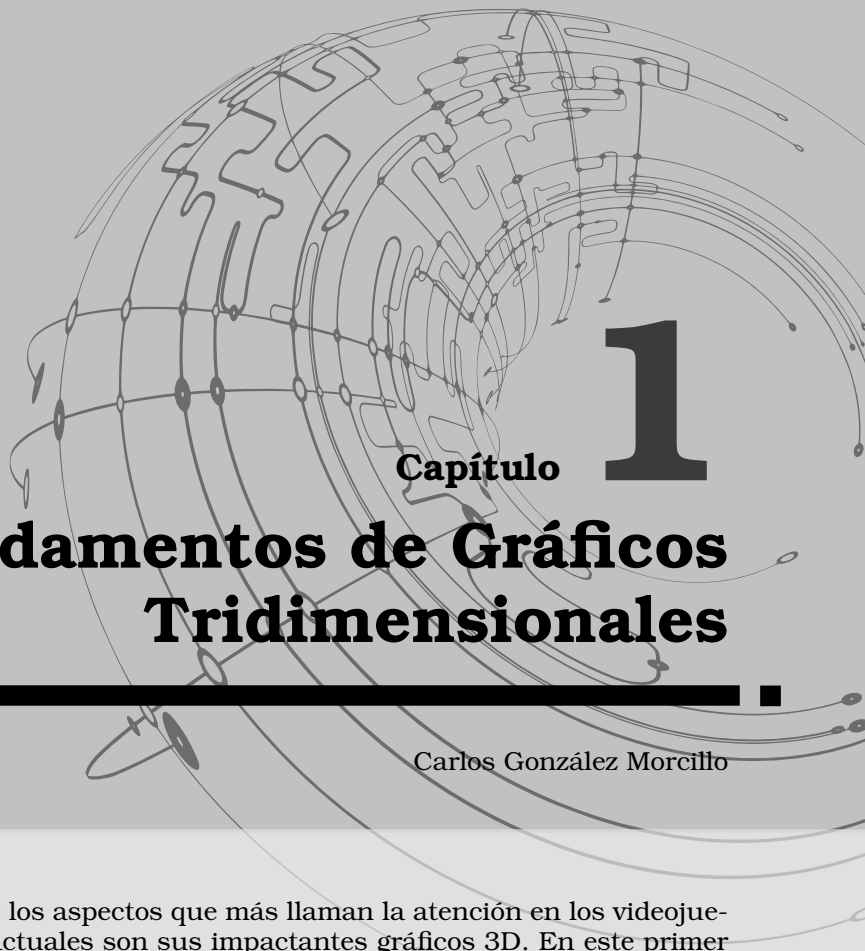
13.2.3 Portal *Rendering* 250

13.2.4 Mapas de Oclusión Jerárquicos (HOM) 252

13.2.5 Enfoques híbridos 254
13.2.6 Tests asistidos por *hardware* 255
13.3 Manejo de escenas en OGRE 256
13.3.1 Interiores en OGRE 257

14. Optimización de exteriores **259**
14.1 Introducción 259
14.2 Estructuras de datos 260
14.3 Determinación de la resolución 262
14.3.1 Políticas Discretas de LOD 262
14.3.2 Políticas Continuas de LOD 263
14.4 Técnicas y Algoritmos 263
14.4.1 *GeoMipmapping* 264
14.4.2 ROAM 266
14.4.3 *Chunked* LODs 268
14.4.4 Terrenos y GPU 269
14.4.5 *Scenegraphs* de Exteriores 269
14.5 Exteriores y LOD en OGRE 270
14.5.1 Terrenos 270
14.5.2 *Skyboxes*, *skydomes* y *skyplanes* 274
14.5.3 LOD : Materiales y Modelos 277
14.6 Conclusiones 280





Capítulo 1

Fundamentos de Gráficos Tridimensionales

Carlos González Morcillo

Uno de los aspectos que más llaman la atención en los videojuegos actuales son sus impactantes gráficos 3D. En este primer capítulo introduciremos los conceptos básicos asociados al pipeline en gráficos 3D. Se estudiarán las diferentes etapas de transformación de la geometría hasta su despliegue final en coordenadas de pantalla. En la segunda parte del capítulo estudiaremos algunas de las capacidades básicas del motor gráfico de videojuegos y veremos los primeros ejemplos básicos de uso de Ogre.

1.1. Introducción

¡¡Aún más rápido!!

Si cada *frame* tarda en desplegarse más de 40ms, no conseguiremos el mínimo de 25 fps (*frames per second*) establecido como estándar en cine. En videojuegos, la frecuencia recomendable *mínima* es de unos 50 fps.

Desde el punto de vista del usuario, un videojuego puede definirse como una aplicación software que responde a una serie de eventos, *redibujando* la escena y generando una serie de respuestas adicionales (sonido en los altavoces, vibraciones en dispositivos de control, etc...).

El *redibujado* de esta escena debe realizarse lo más rápidamente posible. La capa de aplicación en el despliegue gráfico es una de las actividades que más ciclos de CPU consumen (incluso, como veremos en la sección 1.3, con el apoyo de las modernas GPUs *Graphics Processing Unit* existentes en el mercado).

Habitualmente el motor gráfico trabaja con geometría descrita mediante mallas triangulares. Las técnicas empleadas para optimizar el despliegue de esta geometría, junto con las propiedades de materia-

les, texturas e iluminación, varían dependiendo del tipo de videojuego que se está desarrollando. Por ejemplo, en un simulador de vuelo, el tratamiento que debe darse de la geometría distante (respecto de la posición de la cámara virtual) debe ser diferente que la empleada para optimizar la visualización de interiores en un videojuego de primera persona.

Uno de los errores habituales en videojuegos *amateur* es la falta de comunicación clara entre programadores y diseñadores. El desarrollador debe especificar claramente las capacidades soportadas por el motor gráfico al equipo de modeladores, animadores y equipo artístico en general. Esta información debe comprender tanto las técnicas de despliegue soportadas, como el número de polígonos disponibles para el personaje principal, enemigos, fondos, etc...

A un alto nivel de abstracción podemos ver el proceso de *render* como el encargado de convertir la descripción de una escena tridimensional en una imagen bidimensional. En los primeros años de estudio de esta disciplina, la investigación se centró en cómo resolver problemas relacionados con la detección de superficies visibles, sombreado básico, etc. Según se encontraban soluciones a estos problemas, se continuó el estudio de algoritmos más precisos que simularan el comportamiento de la luz de una forma más precisa.

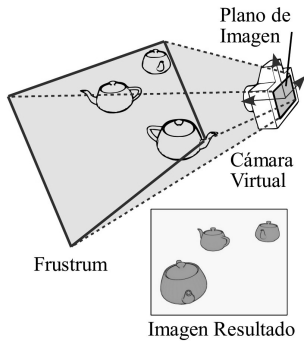


Figura 1.1: Descripción general del proceso de *rendering*.

En esencia, el proceso de *rendering* de una escena 3D requiere los siguientes elementos:

- **Superficies**. La geometría de los objetos que forman la escena debe ser definida empleando alguna representación matemática, para su posterior procesamiento por parte del ordenador.
- **Cámara**. La situación del visor debe ser definida mediante un par (posición, rotación) en el espacio 3D. El plano de imagen de esta cámara virtual definirá el resultado del proceso de *rendering*. Como se muestra en la Figura 1.1, para imágenes generadas en perspectiva, el volumen de visualización define una pirámide truncada que selecciona los objetos que serán representados en la escena. Esta pirámide se denomina *Frustum*.
- **Fuentes de luz**. Las fuentes de luz emiten rayos que interactúan con las superficies e impactarán en el plano de imagen. Dependiendo del modo de simulación de estos impactos de luz (de la resolución de la denominada *ecuación de render*), tendremos diferentes *métodos* de rendering.
- **Propiedades de las superficies**. En este apartado se incluyen las propiedades de materiales y texturas que describen el modelo de *rebote* de los fotones sobre las superficies.

Tiempo de cómputo

En síntesis de imagen realista (como por ejemplo, las técnicas de rendering empleadas en películas de animación) el cálculo de un solo fotograma de la animación puede requerir desde varias horas hasta días, empleando computadores de altas prestaciones.

Uno de los principales objetivos en síntesis de imagen es el realismo. En general, según el método empleado para la resolución de la

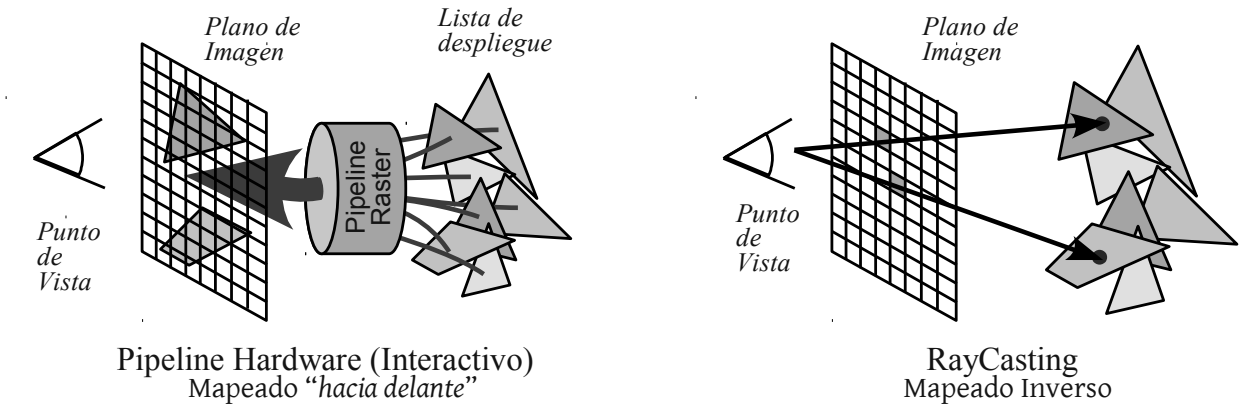


Figura 1.2: Diferencias entre las técnicas de despliegue interactivas (métodos basados en *ScanLine*) y realistas (métodos basados en RayCasting). En el Pipeline Hardware, la selección del píxel más cercano relativo a cada triángulo se realiza directamente en Hardware, empleando información de los *fragmentos* (ver Sección 1.2).

ecuación de render tendremos diferentes niveles de realismo (y diferentes tiempos de cómputo asociados). El principal problema en gráficos en tiempo real es que las imágenes deben ser generadas muy rápidamente. Eso significa, como hemos visto anteriormente, que el motor gráfico dispone de menos de 40 ms para generar cada imagen. Habitualmente este tiempo es incluso menor, ya que es necesario reservar tiempo de CPU para otras tareas como el cálculo de la Inteligencia Artificial, simulación física, sonido...

Los primeros métodos de sombreado de superficies propuestos por *Gouraud* y *Phong* no realizaban ninguna simulación física de la reflexión de la luz, calculando únicamente las contribuciones locales de iluminación. Estos modelos tienen en cuenta la posición de la luz, el observador y el vector normal de la superficie. Pese a su falta de realismo, la facilidad de su cómputo hace que estas aproximaciones sigan siendo ampliamente utilizadas en el desarrollo de videojuegos.

RayTracing

El algoritmo original del *RayCasting* de Appel, fue el precursor del método de *RayTracing* (*Trazado de Rayos*) de Whitted de 1980. El método de RayTracing sirvió de base para los principales métodos de síntesis de imagen hiperrealistas que se emplean en la actualidad (Metrópolis, Path Tracing, etc...).

En 1968 Arthur Appel describió el primer método para generar imágenes por computador lanzando rayos desde el punto de vista del observador. En este trabajo, generaba la imagen resultado en un plotter donde dibujaba punto a punto el resultado del proceso de render. La idea general del método de RayCasting es lanzar rayos desde el plano de imagen, uno por cada píxel, y encontrar el punto de intersección más cercano con los objetos de la escena. La principal ventaja de este método frente a los métodos de tipo *scanline* que emplean *zbuffer* es que es posible generar de forma consistente la imagen que represente el mundo 3D, ya que cualquier objeto que pueda ser descrito mediante una ecuación puede ser representado de forma correcta mediante RayCasting.

Como puede verse en la Figura 1.2, existen diferencias importantes entre el método de despliegue que implementan las tarjetas aceleradoras 3D (y en general los motores de visualización para aplicaciones interactivas) y el método de RayCasting. El *pipeline* gráfico de aplicacio-

nes interactivas (como veremos en la sección 1.2) puede describirse de forma general como el que, a partir de una lista de objetos geométricos a representar y, tras aplicar la serie de transformaciones geométricas sobre los objetos, la vista y la perspectiva, obtienen una imagen raster dependiente del dispositivo de visualización. En este enfoque, las primitivas se ordenan según la posición de la cámara y sólo las visibles serán dibujadas. Por el contrario, en métodos de síntesis de imagen realista (como la aproximación inicial de RayCasting) calcula los rayos que pasan por cada píxel de la imagen y recorre la lista de objetos, calculando la intersección (si hay alguna) con el objeto más cercano. Una vez obtenido el punto de intersección, se evalúa (empleando un modelo de iluminación) el valor de sombreado correspondiente a ese píxel.

1.2. El Pipeline Gráfico

Para obtener una imagen de una escena 3D definida en el *Sistema de Referencia Universal*, necesitamos definir un sistema de referencia de coordenadas para los **parámetros de visualización** (también denominados *parámetros de cámara*). Este sistema de referencia nos definirá el plano de proyección, que sería el equivalente de la zona de la cámara sobre la que se registrará la imagen[1]. De este modo se transfieren los objetos al sistema de coordenadas de visualización y finalmente se proyectan sobre el plano de visualización (ver Figura 1.4).

El proceso de visualizar una escena en 3D mediante gráficos por computador es similar al que se realiza cuando se toma una fotografía real. En primer lugar hay que situar el *trípode* con la cámara en un lugar del espacio, eligiendo así una posición de visualización. A continuación, rotamos la cámara eligiendo si la fotografía la tomaremos en vertical o en apaisado, y apuntando al motivo que queremos fotografiar. Finalmente, cuando disparamos la fotografía, sólo una pequeña parte del mundo queda representado en la imagen 2D final (el resto de elementos son *recortados* y no aparecen en la imagen).

La Figura 1.3 muestra los pasos generales del *Pipeline* asociado a la transformación de una escena 3D hasta su representación final en el dispositivo de visualización (típicamente una pantalla con una determinada resolución).

El *Pipeline* está dividido en etapas funcionales. Al igual que ocurre en los *pipeline* de fabricación industrial, algunas de estas etapas se realizan en paralelo y otras secuencialmente. Idealmente, si dividimos un proceso en n etapas se incrementará la velocidad del proceso en ese factor n. Así, la velocidad de la cadena viene determinada por el tiempo requerido por la etapa más lenta.

Figura 1.3: Pipeline general en gráficos 3D.

[1] En el mundo físico, la *película* en antiguas cámaras analógicas, o el sensor de imagen de las cámaras digitales.

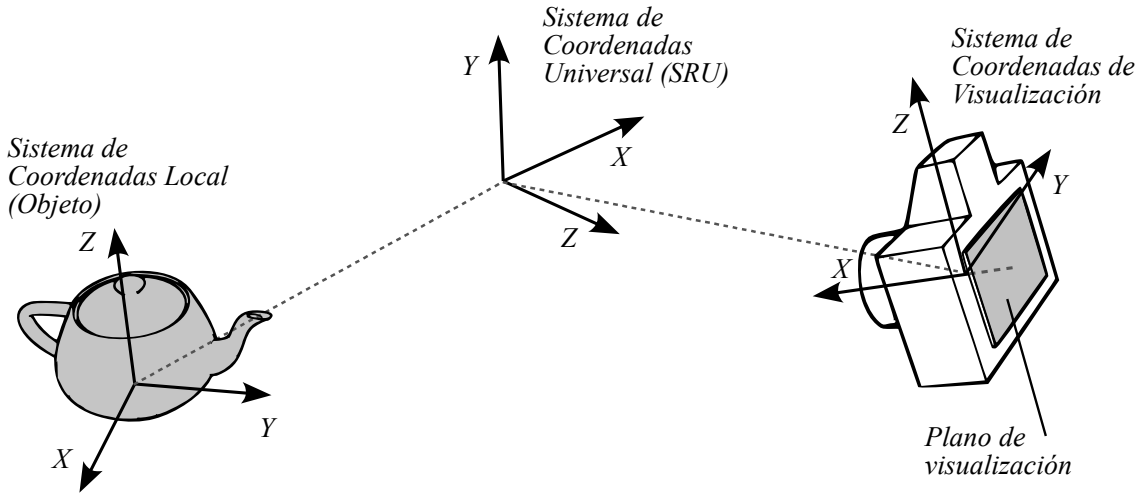


Figura 1.4: Sistema de coordenadas de visualización y su relación con otros sistemas de coordenadas de la escena.

Como señala Akenine-Möler [AMHH08], el pipeline interactivo se divide en tres etapas conceptuales de *Aplicación*, *Geometría* y *Rasterización* (ver Figura 1.3). A continuación estudiaremos estas etapas.

1.2.1. Etapa de Aplicación

La etapa de aplicación se ejecuta en la CPU. Actualmente la mayoría de las CPUs son multinúcleo, por lo que el diseño de esta aplicación se realiza mediante diferentes hilos de ejecución en paralelo. Habitualmente en esta etapa se ejecutan tareas asociadas al cálculo de la posición de los modelos 3D mediante simulaciones físicas, detección de colisiones, gestión de la entrada del usuario (teclado, ratón, joystick...). De igual modo, el uso de estructuras de datos de alto nivel para la aceleración del despliegue (reduciendo el número de polígonos que se envían a la GPU) se implementan en la etapa de aplicación.

1.2.2. Etapa de Geometría

En su tortuoso viaje hasta la pantalla, cada objeto 3D se transforma en diferentes sistemas de coordenadas. Originalmente, como se muestra en la Figura 1.4, un objeto tiene su propio *Sistema de Coordenadas Local* que nos definen las **Coordenadas de Modelo**, por lo que *desde su punto de vista* no está transformado.

A los vértices de cada modelo se le aplican la denominada **Transformación de Modelado** para posicionarlo y orientarlo respecto del *Sistema de Coordenadas Universal*, obteniendo así las denominadas **Coordenadas Universales** o *Coordenadas del Mundo*.

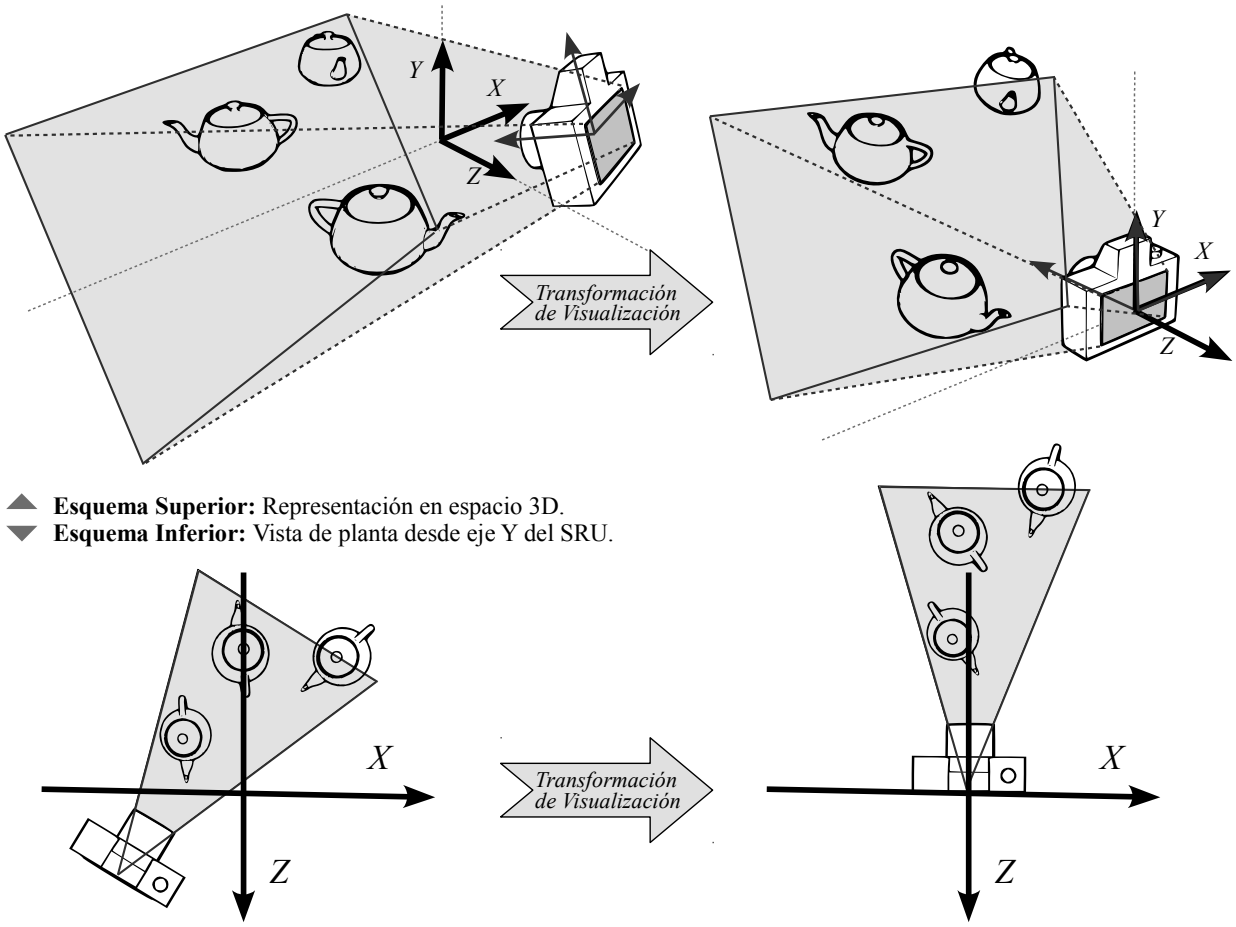


Figura 1.5: El usuario especifica la posición de la cámara (izquierda) que se transforma, junto con los objetos de la escena, para posicionarlos a partir del origen del SRU y mirando en la dirección negativa del eje Z. El área sombreada de la cámara se corresponde con el volumen de visualización de la misma (sólo los objetos que estén contenidos en esa pirámide serán finalmente representados).

Instancias

Gracias a la separación entre *Coordenadas de Modelo* y *Transformación de Modelado* podemos tener diferentes instancias de un mismo modelo para construir una escena a las que aplicamos diferentes transformaciones. Por ejemplo, para construir un templo romano tendríamos un único objeto de columna y varias *instancias* de la columna a las que hemos aplicado diferentes traslaciones.

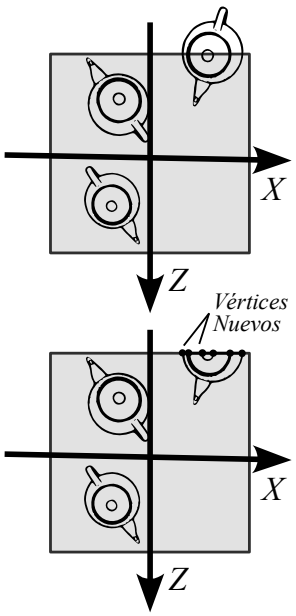


Figura 1.6: Los objetos que intersecan con los límites del *cubo unitario* (arriba) son recortados, añadiendo nuevos vértices. Los objetos que están totalmente dentro del cubo unitario se pasan directamente a la siguiente etapa. Los objetos que están totalmente fuera del *cubo unitario* son descartados.

Como este sistema de coordenadas es único, tras aplicar la transformación de modelado a cada objeto, ahora todas las coordenadas estarán expresadas en el mismo espacio.

La posición y orientación de la cámara nos determinará qué objetos aparecerán en la imagen final. Esta cámara tendrá igualmente unas coordenadas universales. El propósito de la **Transformación de Visualización** es posicionar la cámara en el origen del *SRU*, apuntando en la dirección negativa del eje Z y el eje Y hacia arriba. Obtenemos de este modo las **Coordenadas de Visualización** o *Coordenadas en Espacio Cámara* (ver Figura 1.5).

Habitualmente el pipeline contiene una etapa adicional intermedia que se denomina **Vertex Shader** *Sombreado de Vértice* que consiste en obtener la representación del material del objeto modelando las transformaciones en las fuentes de luz, utilizando los vectores normales a los puntos de la superficie, información de color, etc. Es conveniente en muchas ocasiones transformar las posiciones de estos elementos (fuentes de luz, cámara, ...) a otro espacio (como *Coordenadas de Modelo*) para realizar los cálculos.

La **Transformación de Proyección** convierte el **volumen de visualización** en un cubo unitario (ver Sección 1.2.4). Este *volumen de visualización* se define mediante planos de recorte 3D y define todos los elementos que serán visualizados. En la figura 1.5 se representa mediante el volumen sombreado. Existen multitud de métodos de proyección, aunque como veremos más adelante, los más empleados son la ortográfica (o paralela) y la perspectiva.

En la sección 1.2.4 estudiaremos cómo se realiza la proyección en perspectiva de un modo simplificado. Cuando veamos en el capítulo 2, determinaremos la expresión mediante la que los objetos de la escena se proyectan en un volumen simple (el *cubo unitario*) antes de proceder al recorte y su posterior *rasterización*.

Tras la proyección, el *volumen de visualización* se transforma en **Coordenadas Normalizadas** (obteniendo el *cubo unitario*), donde los modelos son proyectados de 3D a 2D. La coordenada Z se guarda habitualmente en un buffer de profundidad llamado *Z-Buffer*.

Únicamente los objetos que están dentro del *volumen de visualización* deben ser generados en la imagen final. Los objetos que están *totalmente* dentro del volumen de visualización serán copiados íntegramente a la siguiente etapa del *pipeline*. Sin embargo, aquellos que estén parcialmente incluidas necesitan ser recortadas, generando nuevos vértices en el límite del recorte. Esta operación de **Transformación de Recorte** se realiza automáticamente por el hardware de la tarjeta gráfica. En la Figura 1.6 se muestra un ejemplo simplificado de recorte.

Finalmente la **Transformación de Pantalla** toma como entrada las coordenadas de la etapa anterior y produce las denominadas **Coordenadas de Pantalla**, que ajustan las coordenadas x e y del cubo unitario a las dimensiones de ventana finales.

1.2.3. Etapa Rasterización

A partir de los vértices proyectados (en *Coordenadas de Pantalla*) y la información asociada a su sombreado obtenidas de la etapa anterior, la etapa de *rasterización* se encarga de calcular los colores finales que se asignarán a los píxeles de los objetos. Esta etapa de rasterización se divide normalmente en las siguientes etapas funciones para lograr mayor paralelismo.

En la primera etapa del pipeline llamada **Configuración de Triángulos** (*Triangle Setup*), se calculan las coordenadas 2D que definen el contorno de cada triángulo (el primer y último punto de cada vértice). Esta información es utilizada en la siguiente etapa (y en la interpolación), y normalmente se implementa directamente en hardware dedicado.

A continuación, en la etapa del **Recorrido de Triángulo** (*Triangle Traversal*) se generan *fragmentos* para la parte de cada píxel que pertenece al triángulo. El recorrido del triángulo se basa por tanto en encontrar los píxeles que forman parte del triángulo, y se denomina *Triangle Traversal* (o *Scan Conversion*). El *fragmento* se calcula interpolando la información de los tres vértices denifidos en la etapa de *Configuración de Triángulos* y contiene información calculada sobre la profundidad desde la cámara y el sombreado (obtenida en la etapa de geometría a nivel de todo el triángulo).

La información interpolada de la etapa anterior se utiliza en el **Pixel Shader** (*Sombreado de Píxel*) para aplicar el sombreado a nivel de píxel. Esta etapa habitualmente se ejecuta en núcleos de la GPU programables, y permite implementaciones propias por parte del usuario. En esta etapa se aplican las texturas empleando diversos métodos de proyección (ver Figura 1.7).

Finalmente en la etapa de **Fusión** (*Merging*) se almacena la información del color de cada píxel en un array de colores denominado *Color Buffer*. Para ello, se combina el resultado de los *fragmentos* que son visibles de la etapa de *Sombreado de Píxel*. La visibilidad se suele resolver en la mayoría de los casos mediante un buffer de profundidad *Z-Buffer*, empleando la información que almacenan los *fragmentos*.

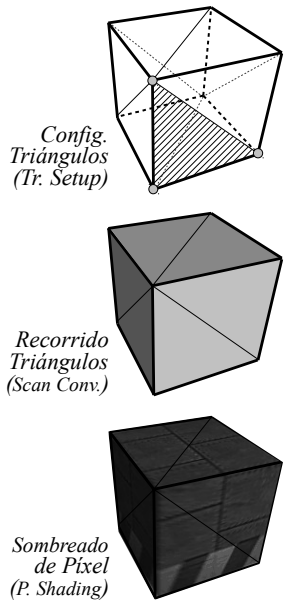


Figura 1.7: Representación del resultado de las principales etapas del Pipeline de Rasterización.

El **Z-Buffer** es un buffer ampliamente empleado en gráficos por computador. Tiene el mismo tamaño en píxeles que el buffer de color, pero almacena la *menor* distancia para cada píxel a todos los fragmentos de la escena. Habitualmente se representa como una imagen en escala de grises, y asocia valores más cercanos a blanco a distancias menores.

1.2.4. Proyección en Perspectiva

En gráficos por computador es posible elegir entre diferentes modelos de proyección de los objetos sobre el plano de visualización. Un modo muy utilizado en aplicaciones de CAD es la proyección de los objetos empleando líneas paralelas sobre el plano de proyección, mediante la denominada **proyección paralela**. En este modo de proyección se conservan las proporciones relativas entre objetos, independientemente de su distancia.

Mediante la **proyección en perspectiva** se proyectan los puntos hasta el plano de visualización empleando trayectorias convergentes en un punto. Esto hace que los objetos situados más distantes del plano de visualización aparezcan más pequeños en la imagen. Las escenas generadas utilizando este modelo de proyección son más realistas, ya que ésta es la manera en que el ojo humano y las cámaras físicas forman imágenes.

Perspectiva

La mayoría de los juegos 3D se basan en el uso de cámaras de proyección en perspectiva. Estudiaremos con más detalle este tipo de modelos de proyección en el capítulo 2.

En la proyección en perspectiva, las líneas paralelas convergen en un punto, de forma que los objetos más cercanos se muestran de un tamaño mayor que los lejanos. Desde el 500aC, los griegos estudiaron el fenómeno que ocurría cuando la luz pasaba a través de pequeñas aberturas. La primera descripción de una cámara estenopeica se atribuye al atrónomo y matemático holandés *Gemma Frisius* que en 1545 publicó la primera descripción de una *cámara oscura* en la observación de un eclipse solar (ver Figura 1.8). En las cámaras esteneopeicas la luz pasa a través de un pequeño agujero para formar la imagen en la película fotosensible, que aparece invertida. Para que la imagen sea nítida, la abertura debe ser muy pequeña.

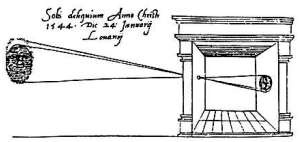


Figura 1.8: Descripción de la primera cámara estenopeica (*pinhole camera* o *camera obscura*) por Gemma Frisius.

Siguiendo la misma idea y desplazando el plano de proyección delante del origen, tenemos el modelo general proyección en perspectiva.

Consideraremos en el resto de la sección que ya se ha realizado la *transformación de visualización* alineando la cámara y los objetos de la escena mirando en dirección al eje negativo Z, que el eje Y está apuntando hacia arriba y el eje X positivo a la derecha (como se muestra en la Figura 1.5).

En la Figura 1.9 se muestra un ejemplo de proyección simple, en la que los vértices de los objetos del mundo se proyectan sobre un plano infinito situado en $z = -d$ (con $d > 0$). Suponiendo que la *transformación de visualización* se ha realizado, proyectamos un punto p sobre el plano de proyección, obteniendo un punto $p' = (p'_x, p'_y, -d)$.

Empleando triángulos semejantes (ver Figura 1.9 derecha), obtenemos las siguientes coordenadas:

$$\frac{p'_x}{p_x} = \frac{-d}{p_z} \quad \Leftrightarrow \quad p'_x = \frac{-d\,p_x}{p_z} \tag{1.1}$$

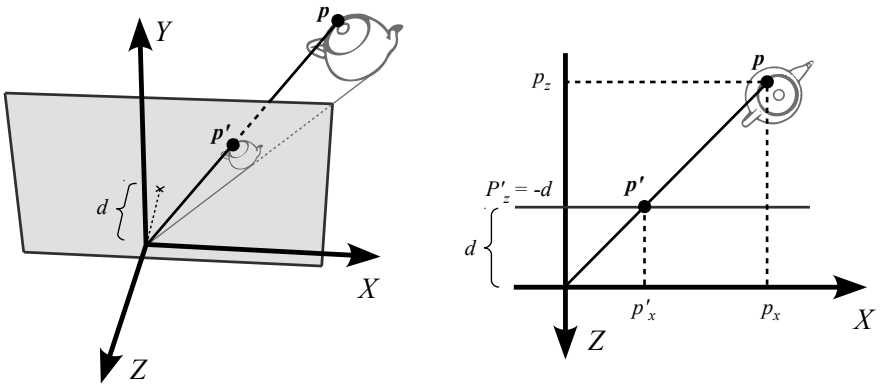


Figura 1.9: Modelo de proyección en perspectiva simple. El plano de proyección infinito está definido en $z = -d$, de forma que el punto p se proyecta sobre p'.

De igual forma obtenemos la coordenada $p'_y = -d\, p_y/p_z$, y $p'_z = -d$. Como veremos en el Capítulo 2, estas ecuaciones se pueden expresar fácilmente de forma matricial (que es la forma habitual de trabajar internamente en el pipeline). Estudiaremos más detalles sobre el modelo de proyección en perspectiva en el Capítulo 2, así como la transformación que se realiza de la pirámide de visualización (*Frustum*) al cubo unitario.

1.3. Implementación del Pipeline en GPU

El término GPU

Desde el 1999, se utiliza el término GPU (*Grpahics Processing Unit*) acuñado por NVIDIA para diferenciar la primera tarjeta gráfica que permitía al programador implementar sus propios algoritmos (*GeForce 256*).

El hardware de aceleración gráfica ha sufrido una importante transformación en la última década. Como se muestra en la Figura 1.10, en los últimos años el potencial de las GPUs ha superado con creces al de la CPU.

Resulta de especial interés conocer aquellas partes del Pipeline de la GPU que puede ser programable por el usuario mediante el desarrollo de *shaders*. Este código se ejecuta directamente en la GPU, y permite realizar operaciones a diferentes niveles con alta eficiencia.

En GPU, las etapas del *Pipeline gráfico* (estudiadas en la sección 1.2) se implementan en un conjunto de etapas diferente, que además pueden o no ser programables por parte del usuario (ver Figura 1.11). Por cuestiones de eficiencia, algunas partes del Pipeline en GPU no son programables, aunque se prevee que la tendencia en los próximos años sea permitir su modificación.

La etapa asociada al *Vertex Shader* es totalmente programable, y encapsula las cuatro primeras etapas del pipeline gráfico que estudiamos en la sección 1.2: *Transformación de Modelado*, *Transformación de Visualización*, *Sombreado de Vértices (Vertex Shader)* y *Transformación de Proyección*.

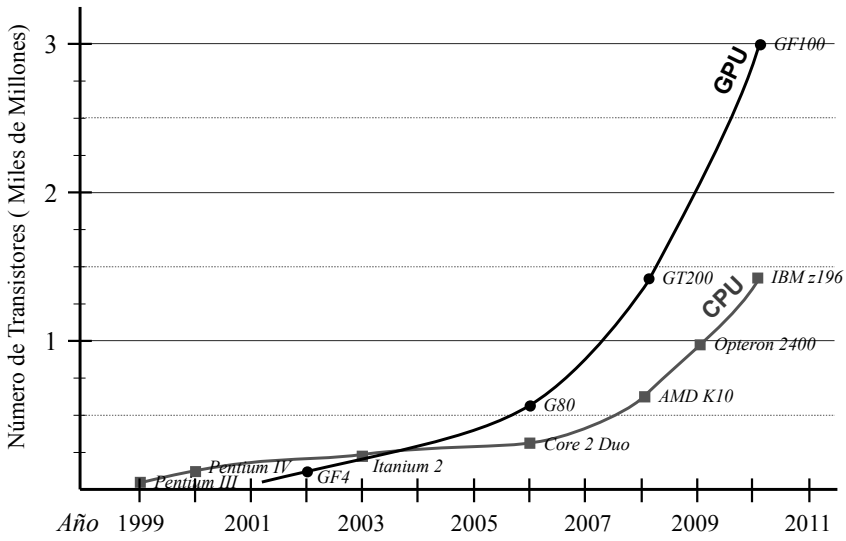


Figura 1.10: Evolución del número de transistores en GPU y CPU (1999-2010)

Figura 1.11: Implementación típica del pipeline en GPU. La letra asociada a cada etapa indica el nivel de modificación permitida al usuario; **P** indica totalmente programable, **F** indica fijo (no programable), y **C** indica configurable pero no programable.

La etapa referente al *Geometry Shader* es otra etapa programable que permite realizar operaciones sobre las primitivas geométricas.

Las etapas de *Transformación de Recorte*, *Transformación de Pantalla*, *Configuración de Triángulo*, *Recorrido de Triángulo* y *Fusión* tienen un comportamiento funcional similar al estudiado en la sección 1.2, por lo que no serán descritas de nuevo.

Finalmente, el *Pixel Shader* es la última etapa totalmente programable del pipeline en GPU y permite al programador desarrollar operaciones específicas a nivel de píxel.

El desarrollo de shaders actualmente se realiza en lenguajes de alto nivel con una sintaxis similar a C, como HLSL, GLSL y Cg. Estos lenguajes de alto nivel son compilados a un lenguaje ensamblador intermedio independiente de la tarjeta gráfica. Son los drivers de cada tarjeta gráfica los que transforman este lenguaje intermedio en instrucciones específicas para cada tarjeta.

Los tres lenguajes de desarrollo de shading más empleados actualmente son HLSL (*High Level Shader Language*), desarrollado por Microsoft en colaboración con NVIDIA para Direct3D, GLSL (*OpenGL Shading Language*), que forma parte del estándar multiplataforma OpenGL, y Cg (*C for Graphics*) lenguaje abierto propuesto por NVidia.

Veremos a continuación brevemente algunas características de estas etapas programables de la GPU.

1.3.1. Vertex Shader

Los *vertex shaders* permiten aplicar transformaciones y deformaciones a nivel de vértice. Este *shader* se aplica en la primera etapa del pipeline de la GPU. En esta etapa, los flujos de datos que la CPU envía a la tarjeta son procesados y se aplican las matrices de transformación especificadas por el usuario.

En esta etapa se aplican las instancias sobre los datos enviados a la GPU (evitando enviar varias veces las mismas primitivas geométricas). A este nivel, el *vertex shader* únicamente trabaja con la información relativa a los vértices (posición, vector normal, color y coordenadas de textura). El *vertex shader* no conoce nada sobre la conexión de estos vértices entre sí para formar triángulos.

Algunas operaciones clásicas que se implementan empleando *vertex shader* son efectos de lente (como por ejemplo, de ojo de pez o distorsiones como las causadas en escenas submarinas), deformaciones de objetos, animaciones de textura, etc.

1.3.2. Geometry Shader

Los *geometry shaders* facilitan la creación y destrucción de primitivas geométricas en la GPU en tiempo de ejecución (vértices, líneas y triángulos).

La entrada de este módulo lo forman la especificación de los objetos con sus vértices asociados. Para cada primitiva de entrada, el *geometry shader* devolverá cero o más primitivas de salida. Las primitivas de entrada y salida no tienen por qué ser del mismo tipo. Por ejemplo, es posible indicar un triángulo como entrada (tres vértices 3d) y devolver el centroide (un punto 3D) como salida. Las primitivas del flujo de salida del *geometry shader* se obtienen en el mismo orden que se especificaron las primitivas de entrada.

Este tipo de shaders se emplean para la simulación de pelo, para encontrar los bordes de los objetos, o para implementar algunas técnicas de visualización avanzadas como metabolas o simulación de telas.

1.3.3. Pixel Shader

A nivel de *pixel shader* se pueden aplicar operaciones a nivel de píxel, permitiendo definir complejas ecuaciones de sombreado que serán evaluadas para cada píxel de la imagen.

Ensamblador?

En realidad este código ensamblador intermedio puede verse como una especie de código de máquina virtual que garantiza la compatibilidad entre diferentes disposi-

Figura 1.12: Resultado de aplicar un *Vertex Shader* para deformar un modelo.

Notación...

En OpenGL al *Pixel Shader* se le denomina *Fragment Shader*. En realidad es un mejor nombre, porque se trabaja a nivel de *fragmento*.

El *Pixel Shader* tiene influencia únicamente sobre el fragmento que está manejando. Esto implica que no puede aplicar ninguna transformación sobre fragmentos vecinos.

El uso principal que se da a este tipo de *shaders* es el establecimiento mediante código del color y la profundidad asociada al fragmento. Actualmente se emplea para aplicar multitud de efectos de representación no realista, reflexiones, etc.

1.4. Arquitectura del motor gráfico

El objetivo de esta sección es proporcionar una primera visión general sobre los conceptos generales subyacentes en cualquier motor gráfico 3D interactivo. Estos conceptos serán estudiados en profundidad a lo largo de este documento, mostrando su uso práctico mediante ejemplos desarrollados en C++ empleando el motor gráfico OGRE.

Como se ha visto en la introducción del capítulo, los videojuegos requieren hacer un uso eficiente de los recursos gráficos. Hace dos décadas, los videojuegos se diseñaban específicamente para una plataforma hardware específica, y las optimizaciones podían realizarse a muy bajo nivel. Actualmente, el desarrollo de un videojuego tiende a realizarse para varias plataformas, por lo que el uso de un motor gráfico que nos abstraiga de las particularidades de cada plataforma no es una opción, sino una necesidad.

Compatiblidad

En algunos casos, como en el desarrollo de videojuegos para PC, el programador no puede hacer casi ninguna suposición sobre el hardware subyacente en la máquina donde se ejecutará finalmente el programa. En el caso de desarrollo para consolas, los entornos de ejecución concretos están mucho más controlados.

En estos desarrollos multiplataforma es necesario abordar aproximaciones de diseño que permitan emplear diferentes *perfiles* de ejecución. Por ejemplo, en máquinas con grandes prestaciones se emplearán efectos y técnicas de despliegue más realistas, mientras que en máquinas con recursos limitados se utilizarán algoritmos con menores requisitos computacionales y versiones de los recursos gráficos adaptadas (con diferente nivel de detalle asociado).

Las limitaciones asociadas a los recursos computacionales son una constante en el área del desarrollo de videojuegos. Cada plataforma conlleva sus propias limitaciones y restricciones, que pueden asociarse en las categorías de:

- **Tiempo de Procesamiento**. El desarrollo de videojuegos en prácticamente cualquier plataforma actual require el manejo de múltiples núcleos de procesamiento (tanto de CPU como GPU). El manejo explícito de la concurrencia (habitualmente a nivel de hilos) es necesario para mantener una alta tasa de *Frames por Segundo*.

- **Almacenamiento**. En el caso de ciertos dispositivos como consolas, la variedad de unidades de almacenamiento de los recursos del juego (con velocidades de acceso y transferencia heterogéneas), dificultan el desarrollo del videojuego. En ciertas plataformas, no se dispone de aproximaciones de memoria virtual, por lo que el programador debe utilizar explícitamente superposiciones

Figura 1.13: Capturas de algunos videojuegos desarrollados con motores libres. La imagen de la izquierda se corresponde con *Planeshift*, realizado con Crystal Space. La captura de la derecha es del videojuego *H-Craft Championship*, desarrollado con Irrlicht.

(*overlays*) para cargar las zonas del juego que van a emplearse en cada momento.

Dada la gran cantidad de restricciones que deben manejarse, así como el manejo de la heterogeneidad en términos software y hardware, es necesario el uso de un motor de despliegue gráfico para desarrollar videojuegos. A continuación enunciaremos algunos de los más empleados.

1.5. Casos de Estudio

En esta sección se estudiarán algunos de los motores gráficos 3D libres, comentando brevemente algunas de sus características más destacables.

- **Crystal Space**. Crystal Space (`http://www.crystalspace3d.org/`) es un framework completo para el desarrollo de videojuegos escrito en C++, desarrollado inicialmente en 1997. Se distribuye bajo licencia libre LGPL, y es multiplataforma (GNU/Linux, Windows y Mac).
- **Panda 3D**. Panda 3D (`http://www.panda3d.org/`) es un motor para el desarrollo de videojuegos multiplataforma. Inicialmente fue desarrollado por Disney para la construcción del software asociado a las atracciones en parques temáticos, y posteriormente liberado en 2002 bajo licencia BSD. Este motor multiplataforma (para GNU/Linux, Windows y Mac) incluye interfaces para C++ y Python. Su corta curva de aprendizaje hace que haya sido utilizado en varios cursos universitarios, pero no ofrece características

de representación avanzadas y el interfaz de alto nivel de Python conlleva una pérdida de rendimiento.

- **Irrlicht**. Este motor gráfico de renderizado 3D, con primera versión publicada en el 2003 (`http://irrlicht.sourceforge.net/`), ofrece interfaces para C++ y .NET. Existen gran cantidad de *wrappers* a diferentes lenguajes como Java, Perl, Python o Lua. Irrlicht tiene una licencia Open Source basada en la licencia de ZLib. Irrlicht es igualmente multiplataforma (GNU/Linux, Windows y Mac).
- **OGRE**. OGRE (`http://www.ogre3d.org/`) es un motor para gráficos 3D libre multiplataforma. Sus características serán estudiadas en detalle en la sección 1.6.

Figura 1.14: El logotipo de OGRE 3D es un... OGRO!

De entre los motores estudiados, OGRE 3D ofrece una calidad de diseño superior, con características técnicas avanzadas que han permitido el desarrollo de varios videojuegos comerciales. Además, el hecho de que se centre exclusivamente en el capa gráfica permite utilizar gran variedad de bibliotecas externas (habitualmente accedidas mediante *plugins*) para proporcionar funcionalidad adicional. En la siguiente sección daremos una primera introducción y toma de contacto al motor libre OGRE.

1.6. Introducción a OGRE

OGRE es un motor orientado a objetos libre para aplicaciones gráficas 3D interactivas. El nombre del motor OGRE es un acrónimo de *Object-oriented Graphics Rendering Engine*. Como su propio nombre indica, OGRE no es un motor para el desarrollo de videojuegos; se centra exclusivamente en la definición de un *middleware* para el renderizado de gráficos 3D en tiempo real.

¡Sólo Rendering!

El desarrollo de OGRE se centra exclusivamente en la parte de despliegue gráfico. El motor no proporciona mecanismos para capturar la interacción del usuario, ni para reproducción audio o gestión del estado interno del videojuego.

El proyecto de OGRE comenzó en el 2000 con el propósito de crear un motor gráfico bien diseñado. El líder del proyecto Steve Streeting define el desarrollo de OGRE como un proyecto basado en la *calidad* más que en la *cantidad* de características que soporta, porque la cantidad viene con el tiempo, y la calidad nunca puede añadirse a posteriori. La popularidad de OGRE se basa en los principios de *meritocracia* de los proyectos de software libre. Así, el sitio web de OGRE [2] recibe más de 500.000 visitas diarias, con más de 40.000 descargas mensuales.

El núcleo principal de desarrolladores en OGRE se mantiene deliberadamente pequeño y está formado por profesionales con dilatada experiencia en proyectos de ingeniería reales.

OGRE tiene una licencia LGPL *Lesser GNU Public License*. Esta licencia se utiliza con frecuencia en bibliotecas que ofrecen funcionalidad que es similar a la de otras bibliotecas privativas. Por cuestión de estrategia, se publican bajo licencia LGPL (o GPL *Reducida*) para

[2] http://www.ogre3d.org

permitir que se enlacen tanto por programas libres como no libres. La única restricción que se aplica es que si el enlazado de las bibliotecas es estático, la aplicación resultado debe ser igualmente LGPL (porque el *enlazado estático* también *enlaza la licencia*).

La versión oficial de OGRE está desarrollada en C++ (el lenguaje estándar en el ámbito del desarrollo de videojuegos). La rama oficial de OGRE únicamente se centra en este lenguaje sobre los sistemas operativos GNU/Linux, Mac OS X y Microsoft Windows. No obstante, existen *wrappers* de la API a otros lenguajes (como Java, Python o C#) que son mantenidos por la comunidad de usuarios (presentando diferentes niveles de estabilidad y completitud), que no forman parte del núcleo oficial de la biblioteca.

Algunas características destacables de OGRE son:

- **Motor Multiplataforma**. Aunque el desarrollo original de OGRE se realizó bajo plataformas Microsoft Windows, la distribución oficial ofrece versiones binarias para GNU/Linux y Mac OS X. Además, gracias al soporte nativo de OpenGL, es posible compilar la biblioteca en multitud de plataformas (como diversas versiones de Unix, además de algunos *ports* no oficiales para Xbox y dispositivos portátiles). OGRE soporta la utilización de las APIs de despliegue gráfico de bajo nivel OpenGL y Direct3D.

Portabilidad

Uno de los objetivos de diseño de OGRE es utilizar otras bibliotecas multiplataforma estables en su desarrollo, como *FreeType* para el despliegue de fuentes TrueType, *OpenIL* para la carga y manipulación de imágenes y *ZLib* para la gestión de archivos comprimidos ZIP.

- **Diseño de Alto Nivel**. OGRE encapsula la complejidad de acceder directamente a las APIs de bajo nivel (como OpenGL y Direct3D) proporcionando métodos intuitivos para la manipulación de objetos y sus propiedades relacionadas. De este modo no es necesario gestionar manualmente la geometría o las matrices de transformación. Todos los objetos representables de la escena se abstraen en un interfaz que encapsula las operaciones necesarias para su despliegue (técnicas empleadas y pasadas de composición).

  OGRE hace un uso de varios **patrones de diseño** para mejorar la usabilidad y la flexibilidad de la bibloteca. Por ejemplo, para informar a la aplicación sobre eventos y cambios de estado utiliza el patrón *Observador*. El patrón *Singleton* se emplea en gran número de *Gestores* para forzar que únicamente exista una instancia de una clase. El patrón *Visitor* se emplea para permitir operaciones sobre un objeto sin necesidad de modificarlo (como en el caso de los nodos del grafo de escena), el patrón *Facade* para unificar el acceso a operaciones, *Factoría* para la creación de instancias concretas de interfaces abstractos, etc. La definición de estos patrones serán estudiadas en detalle en el Módulo 1 del curso.

- **Grafos de Escena**. Prácticamente cualquier biblioteca de despliegue de gráficos 3D utiliza un *Grafo de Escena* para organizar los elementos que serán representados en la misma. Un objetivo fundamental en el diseño de esta estructura de datos es permitir búsquedas eficientes. Una de las características más potentes de OGRE es el desacople del grafo de escena del contenido de la escena, definiendo una arquitectura de plugins. En pocas palabras,

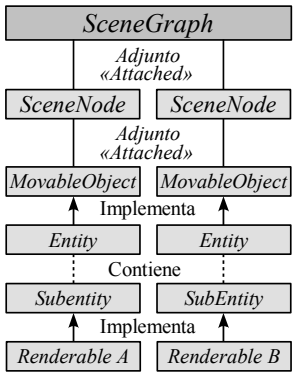


Figura 1.15: Esquema general de la gestión del grafo de escena en OGRE.

a diferencia de otros motores gráficos como Irrlicht3D, Blitz3D o TrueVision3D (o motores de videojuegos como Torque, CryEngine o Unreal), OGRE no se basa en la *Herencia* como principio de diseño del Grafo de Escena, sino en la *Composición*. Esto permite expandir el diseño cómodamente para soportar otros tipos de datos (como audio o elementos de simulación física).

La Figura 1.15 muestra el esquema general de gestión del grafo de escena en OGRE. Los *Renderables* manejan la geometría de la escena. Todas las propiedades para el despliegue de estos *Renderables* (como por ejemplo los materiales) se gestionan en objetos de tipo Entidad (*Entity*) que pueden estar formados por varios objetos SubEntidad (*SubEntity*). Como se ha comentado anteriormente, la escena se *Compone* de nodos de escena. Estos *SceneNodes* se adjuntan a la escena. Las propiedades de esos nodos de escena (geometría, materiales, etc...) se ofrecen al *SceneGraph* mediante un *MovableObject*. De forma similar, los *MovableObjects* no son subclases de *SceneNode*, sino que se adjuntan. Esto permite realizar modificaciones en la implementación del grafo de escena sin necesidad de tocar ninguna línea de código en la implementación de los objetos que contiene. Veremos más detalles sobre el trabajo con el grafo de escena a lo largo de este módulo.

- **Aceleración Hardware**. OGRE necesita una tarjeta aceleradora gráfica para poder ejecutarse (con soporte de *direct rendering mode*). OGRE permite definir el comportamiento de la parte programable de la GPU mediante la definición de *Shaders*, estando al mismo nivel de otros motores como Unreal o CryEngine.

- **Materiales**. Otro aspecto realmente potente en OGRE es la gestión de materiales. Es posible crear materiales sin modificar ni una línea de código a compilar. El sistema de *scripts* para la definición de materiales de OGRE es uno de los más potentes existentes en motores de rendering interactivo. Los materiales de OGRE se definen mediante una o más **Técnicas**, que se componen a su vez de una o más **Pasadas** de rendering (el ejemplo de la Figura 1.16 utiliza una única pasada para simular el sombreado a lápiz mediante *hatching*). OGRE busca automáticamente la mejor técnica disponible en un material que esté soportada por el hardware de la máquina de forma transparente al programador. Además, es posible definir diferentes **Esquemas** asociados a modos de calidad en el despliegue.

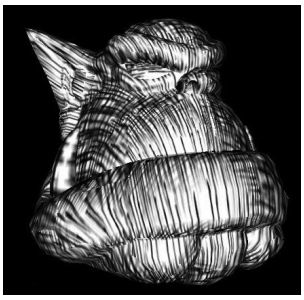


Figura 1.16: Ejemplo de Shader desarrollado por Assaf Raman para simular trazos a Lápiz empleando el sistema de materiales de OGRE.

- **Animación**. OGRE soporta tres tipos de animación ampliamente utilizados en la construcción de videojuegos: basada en esqueletos (*skeletal)*, basada en vértices (*morph* y *pose*). En la animación mediante esqueletos, OGRE permite el uso de esqueletos con animación basada en cinemática directa. Existen multitud de exportadores para los principales paquetes de edición 3D. En este módulo utilizaremos el exportador de Blender.

  El sistema de animación de OGRE se basa en el uso de **controladores**, que modifican el valor de una propiedad en función de otro valor. En el caso de animaciones se utiliza el *tiempo* como

valor para modificar otras propiedades (como por ejemplo la posición del objeto). El motor de OGRE soporta dos modelos básicos de interpolación: lineal y basada en splines cúbicas.

La animación y la geometría asociada a los modelos se almacena en un único formato binario optimizado. El proceso más empleado se basa en la exportación desde la aplicación de modelado y animación 3D a un formato XML (*Ogre XML*) para convertirlo posteriormente al formato binario optimizado mediante la herramienta de línea de órdenes `OgreXMLConverter`.

Optimización offline

El proceso de optimización de al formato binario de OGRE calcula el orden adecuado de los vértices de la malla, calcula las normales de las caras poligonales, así como versiones de diferentes niveles de detalle de la malla. Este proceso evita el cálculo de estos parámetros en tiempo de ejecución.

- **Composición y Postprocesado**. El framework de composición facilita al programador incluir efectos de postprocesado en tiempo de ejecución (siendo una extensión del *pixel shader* del pipeline estudiado en la sección 1.3.3). La aproximación basada en *pasadas* y diferentes *técnicas* es similar a la explicada para el gestor de materiales.
- **Plugins**. El diseño de OGRE facilita el diseño de *Plugins* como componentes que cooperan y se comunican mediante un interfaz conocido. La gestión de archivos, sistemas de rendering y el sistema de partículas están implementados basados en el diseño de Plugins.
- **Gestión de Recursos**. Para OGRE los recursos son los elementos necesarios para representar un objetivo. Estos elementos son la geometría, materiales, esqueletos, fuentes, scripts, texturas, etc. Cada uno de estos elementos tienen asociado un gestor de recurso específico. Este gestor se encarga de controlar la cantidad de memoria empleada por el recurso, y facilitar la carga, descarga, creación e inicialización del recurso. OGRE organiza los recursos en niveles de gestión superiores denominados *grupos*. Veremos en la sección 1.6.1 los recursos gestionados por OGRE.
- **Características específicas avanzadas**. El motor soporta gran cantidad de características de visualización avanzadas, que estudiaremos a lo largo del módulo, tales como sombras dinámicas (basadas en diversas técnicas de cálculo), sistemas de partículas, animación basada en esqueletos y de vértices, y un largo etcétera. OGRE soporta además el uso de otras bibliotecas auxiliares mediante plugins y conectores. Entre los más utilizados cabe destacar las bibliotecas de simulación física ODE, el soporte del metaformato Collada, o la reproducción de streaming de vídeo con Theora. Algunos de estos módulos los utilizaremos en el módulo 3 del presente curso.

1.6.1. Arquitectura General

El diagrama de la Figura 1.17 resume algunos de los objetos principales del motor OGRE. No es un diagrama exhaustivo, pero facilita la comprensión de los principales módulos que utilizaremos a lo largo del curso.

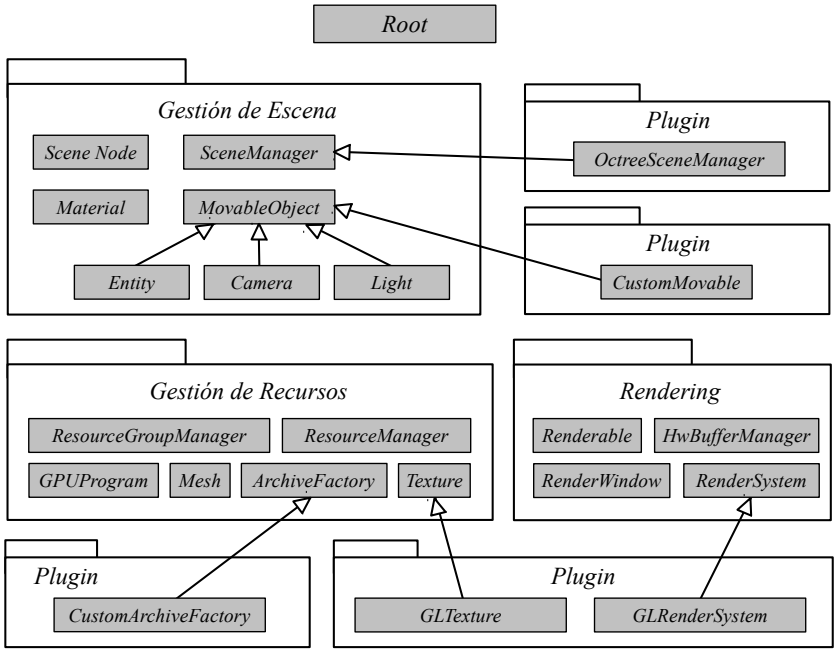


Figura 1.17: Diagrama general de algunos de los objetos principales de OGRE

Objeto Root

El objeto Root es el eje pricipal sobre el que se define una aplicación que utiliza OGRE. La creación de una instancia de esta clase hará que se inicie Ogre, y su destrucción hará que se libere la memoria asociada a todos los objetos que dependen de él.

Uno de los objetos principales del sistema es el denominado *Root*. Root proporciona mecanismos para la creación de los objetos de alto nivel que nos permitirán gestionar las escenas, ventanas, carga de plugins, etc. Gran parte de la funcionalidad de Ogre se proporciona a través del objeto *Root*. Como se muestra en el diagrama 1.17, existen otros tres grupos de clases principales en OGRE:

- **Gestión de Escena**: Los objetos de este grupo de clases se encargan de definir el contenido de la escena virtual, su estructura, así como otras propiedades de alto nivel de abstracción (posición de la cámara, posición de los objetos, materiales, etc...). Como se estudió en la Figura 1.15, el grafo de escena es un elemento principal en la arquitectura de cualquier motor 3D. En el caso de OGRE, el Gestor de Escena (clase *SceneManager*) es el que se encarga de implementar el grafo de escena. Los nodos de escena *SceneNode* son elementos relacionados jerárquicamente, que pueden adjuntarse o desligarse de una escena en tiempo de ejecución. El contenido de estos *SceneNode* se adjunta en la forma de instancias de Entidades (*Entity*), que son implementaciones de la clase *MovableObject*.
- **Gestión de Recursos**: Este grupo de objetos se encarga de gestionar los recursos necesarios para la representación de la escena (geometría, texturas, tipografías, etc...). Esta gestión permite su

carga, descarga y reutilización (mediante cachés de recursos). En la siguiente subsección veremos en detalle los principales gestores de recursos definidos en OGRE.

- **Rendering**: Este grupo de objetos sirve de *intermediario* entre los objetos de *Gestión de Escena* y el pipeline gráfico de bajo nivel (con llamadas específicas a APIs gráficas, trabajo con buffers, estados internos de despliegue, etc.). La clase *RenderSystem* es un interfaz general entre OGRE y las APIs de bajo nivel (OpenGL o Direct3D). La forma más habitual de crear la *RenderWindow* es a través del objeto *Root* o mediante el *RenderSystem* (ver Figura 1.18). La creación manual permite el establecimiento de mayor cantidad de parámetros, aunque para la mayoría de las aplicaciones la creación automática es más que suficiente.

Entidades Procedurales

La mayor parte de las entidades que incluyas en el *SceneNode* serán cargadas de disco (como por ejemplo, una malla binaria en formato `.mesh`). Sin embargo, OGRE permite la definción en código de otras entidades, como una textura procedural, o un plano.

Por ser la gestión de recursos uno de los ejes principales en la creación de una aplicación gráfica interactiva, estudiaremos a continuación los grupos de gestión de recursos y las principales clases relacionadas en OGRE.

Gestión de Recursos

Como hemos comentado anteriormente, cualquier elemento necesario para represesntar una escena se denomina *recurso*. Todos los recursos son gestionados por un único objeto llamado *ResourceGroupManager*, que se encarga de buscar los recursos definidos en la aplicación e inicializarlos. Cada tipo de recurso tiene asociado un gestor de recurso particular. Veamos a continuación los tipos de recursos soportados en OGRE:

- **Mallas**. La geometría de los elementos de la escena se especifica en un formato de malla binario (`.mesh`). Este formato almacena la geometría y la animación asociada a los objetos.
- **Materiales**. Como se ha descrito anteriormente, el material se especifica habitualmente mediante scripts (en ficheros de extensión `.material`). Estos scripts pueden estar referenciados en el propio archivo `.mesh` o pueden ser enlazados a la malla mediante código compilado.
- **Texturas**. OGRE soporta todos los formatos de texturas 2D admitidos por la biblioteca OpenIL. El formato se reconoce por la extensión asociada al mismo.
- **Esqueletos**. Habitualmente el esqueleto está referenciado en el fichero `.mesh`. El fichero de definición de esqueleto contiene la jerarquía de huesos asociada al mismo y tiene una extensión `.skeleton`.
- **Fuentes**. Las fuentes empleadas en la etapa de despliegue de *Overlays* se definen en un archivo `.fontdef`.
- **Composición**. El framework de composición de OGRE carga sus scripts con extensión `.compositor`.

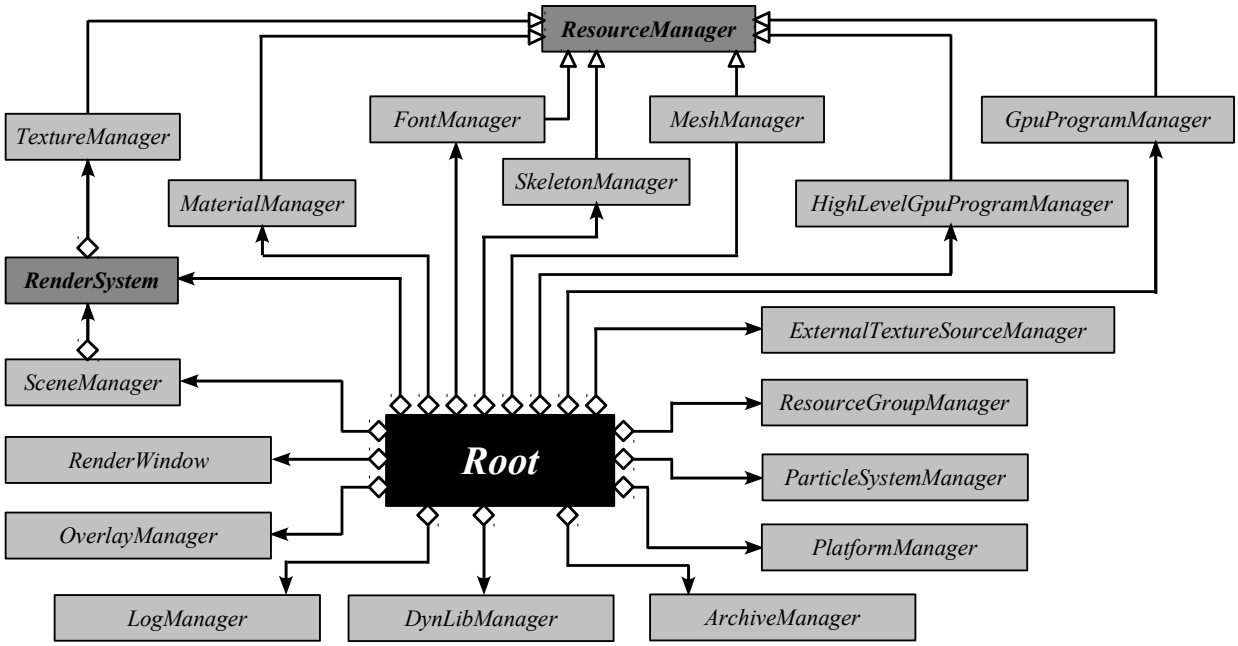


Figura 1.18: Diagrama de clases simplificado de los *Gestores* de alto nivel que dan acceso a los diferentes subsistemas definidos en OGRE

- **GPU**. El código de shaders definidos para la GPU (de HLSL, GLSL y Cg) se describe en archivos con extensión `.program`. De igual modo se pueden definir código en ensamblador mediante archivos `.asm`. Los programas de GPU son cargados antes que se procese cualquier archivo de material `.material`, de forma que estos shaders puedan ser referenciados en los archivos de definición de material.

Un gestor (*Manager*) en OGRE es una clase que gestiona el acceso a otros tipos de objetos (ver Figura 1.18). Los *Managers* de OGRE se implementan mediante el patrón *Singleton* que garantiza que únicamente hay una istancia de esa clase en toda la aplicación. Este patrón permite el acceso a la instancia *única* de la clase *Manager* desde cualquier lugar del código de la aplicación.

Como puede verse en la Figura 1.18, el *ResourceManager* es una clase abstracta de la que heredan un gran número de *Managers*, tales como el encargado de las Fuentes, las Texturas o las Mallas. A continuación se ofrece una descripción general de los *Managers* (por orden alfabético) definidos en OGRE. A lo largo del documento se describirán (y se utilizarán ampliamente) muchos de estos gestores.

- **Archive Manager**. Se encarga de abstraer del uso de ficheros con diferentes extensiones, directorios y archivos empaquetados `.zip`.

- **CompositorManager**. Esta clase proporciona acceso al framework de composición y postprocesado de OGRE.
- **ControllerManager**. Es el gestor de los *controladores* que, como hemos indicado anteriormente, se encargan de producir cambios en el estado de otras clases dependiendo del valor de ciertas entradas.
- **DynLibManager**. Esta clase es una de las principales en el diseño del sistema de Plugins de OGRE. Se encarga de gestionar las bibliotecas de enlace dinámico (DLLs en Windows y objetos compartidos en GNU/Linux).
- **ExternalTextureSourceManager**. Gestiona las fuentes de textura externas (como por ejemplo, en el uso de *video streaming*).
- **FontManager**. Gestiona las fuentes disponibles para su representación en superposiciones 2D (ver *OverlayManager*).
- **GpuProgramManager**. Carga los programas de alto nivel de la GPU, definidos en ensamblador. Se encarga igualmente de cargar los programas compilados por el *HighLevelGpuProgramManager*.
- **HighLevelGpuProgramManager**. Gestiona la carga y compilación de los programas de alto nivel en la GPU (*shaders*) utilizados en la aplicación en HLSL, GLSL o Cg.
- **LogManager**. Se encarga de enviar mensajes de *Log* a la salida definida por OGRE. Puede ser utilizado igualmente por cualquier código de usuario que quiera enviar eventos de *Log*.
- **MaterialManager**. Esta clase mantiene las instancias de *Material* cargadas en la aplicación, permitiendo reutilizar los objetos de este tipo en diferentes objetos.
- **MeshManager**. De forma análoga al *MaterialManager*, esta clase mantiene las instancias de *Mesh* permitiendo su reutilización entre diferentes objetos.
- **OverlayManager**. Esta clase gestiona la carga y creación de instancias de superposiciones 2D, que permiten dibujar el interfaz de usuario (botones, iconos, números, radar...). En general, los elementos definidos como HUDs (*Head Up Display*).
- **ParticleSystemManager**. Gestiona los sistemas de partículas, permitiendo añadir gran cantidad de efectos especiales en aplicaciones 3D. Esta clase gestiona los emisores, los límites de simulación, etc.
- **PlatformManager**. Esta clase abstrae de las particularidades del sistema operativo y del hardware subyacente de ejecución, proporcionando rutinas independientes del sistema de ventanas, temporizadores, etc.
- **ResourceGroupManager**. Esta clase gestiona la lista de grupos de recursos y se encarga de notificar a los *Managers* de la necesidad de cargar o liberar recursos en cada grupo.

- **SceneManager**. Como se ha explicado anteriormente, esta clase se encarga de la gestión, organización y *rendering* de la escena. Esta clase permite definir subclases que organicen la escena de una forma más eficiente, dependiendo del tipo de aplicación. Por defecto, el *SceneManager* utiliza una jerarquía de cajas límite (*bounding boxes*) para optimizar el despliegue de los objetos.
- **SkeletonManager**. Al igual que el *MaterialManager* y el *MeshManager*, esta clase mantiene las instancias de *Skeleton* cargadas en la aplicación, permitiendo su reutilización entre diferentes objetos.
- **TextureManager**. Gestiona la carga y uso de las texturas de la aplicación.

SceneManager

Los *SceneManagers* de OGRE son implementados como Plugins, de forma que el usuario puede cargar varios gestores de escena en su aplicación. Si el videojuego requiere escenas de interiores con mucha geometría, así como escenas de exteriores, puede ser interesante cargar dos gestores de escena diferentes optimizados para cada parte del juego.

1.6.2. Instalación

En esta sección se detallará el proceso de instalación de la biblioteca OGRE 1.7 en sistemas *GNU/Linux* y *Microsoft Windows*. Nos centraremos en la instalación en distribuciones Debian, que servirán como base para el desarrollo del presente curso. No obstante, se proporcionarán igualmente las herramientas necesarias y makefiles adaptados para Microsoft Windows empleando *MinGW*.

GNU/Linux (Debian)

Para comenzar, instalaremos las herramientas de compilación básicas necesarias para compilar: gcc, g++ y make se encuentran disponibles en el metapaquete build-essential:

```
apt-get install build-essential
```

A continuación instalaremos los paquetes específicos de OGRE:

```
apt-get install libogre-1.7.3 libogre-dev ogre-doc ogre-tools
```

El paquete `libogre-1.7.3` contiene las bibliotecas necesarias para la ejecución de las aplicaciones desarrolladas con OGRE. El paquete `libogre-dev` contiene los ficheros de cabecera instalados en `/usr/include/OGRE` necesarios para compilar nuestros propios ejemplos. El paquete de documentación `ogre-doc` instala en `/usr/share/doc/ogre-doc` la documentación (manual de usuario y API). Finalmente el paquete `ogre-tools` contiene las herramientas para convertir al formato de malla binario optimizado de OGRE.

Como se comentó en la introducción, OGRE se centra en proporcionar exclusivamente un motor de despliegue gráfico 3D interactivo. OGRE no proporciona mecanismos para gestionr la entrada del usuario. En este módulo utilizaremos OIS (*Object Oriented Input System*), una biblioteca desarrollada en C++ multiplataforma que permite trabajar con teclado, ratón, joysticks y otros dispositivos de juego. Instalaremos igualmente el paquete binario y las cabeceras.

```
apt-get install libois-1.3.0 libois-dev
```

Tanto OGRE como OIS puede ser igualmente instalado en cualquier distribución compilando directamente los fuentes. En el caso de OGRE, es necesario *CMake* para generar el *makefile* específico para el sistema donde va a ser instalado. En el caso de OIS, es necesario *autotools*.

Microsoft Windows

Aunque no utilizaremos ningún entorno de desarrollo en esta plataforma, dada su amplia comunidad de usuarios, puede ser conveniente generar los ejecutables para esta plataforma. A continuación se detallarán los pasos necesarios para instalar y compilar los desarrollos realizados con OGRE en plataformas Windows.

Como entorno de compilación utilizaremos *MinGW* (*Minimalist GNU for Windows*), que contiene las herramientas básicas de compilación de GNU para Windows.

El instalador de *MinGW* `mingw-get-inst` puede obtenerse de la página web `http://www.mingw.org/`. Puedes instalar las herramientas en `C:\MinGW\`. Una vez instaladas, deberás añadir al *path* el directorio `C:\MinGW\bin`. Para ello, podrás usar la siguiente orden en un terminal del sistema.

```
path = %PATH%;C:\MinGW\bin
```

De igual modo, hay que descargar el SDK de DirectX[3]. Este paso es opcional siempre que no queramos ejecutar ningún ejemplo de OGRE que utilice Direct3D.

A continuación instalaremos la biblioteca OGRE3D para MinGW[4]. Cuando acabe, al igual que hicimos con el directorio bin de MinGW, hay que añadir los directorios `boost_1_44\lib\` y `bin\Release` al path.

```
path = %PATH%;C:\Ogre3D\boost_1_44\lib\; C:\Ogre3D\bin\
```

Sobre Boost...

Boost es un conjunto de bibliotecas libres que añaden multitud de funcionalidad a la biblioteca de C++ (están en proceso de aceptación por el comité de estandarización del lenguaje).

1.7. Hola Mundo en OGRE

A continuación examinaremos un ejemplo básico de funcionamiento de OGRE. Utilizaremos la estructura de directorios mostrada en la Figura 1.19 en los ejemplos desarrollados a lo largo de este módulo.

- En el directorio `include` se incluirán los archivos de cabecera. En este primer ejemplo, no es necesario ningún archivo de cabecera adicional, por lo que este directorio estará vacío.

[3]Puede descargarse de: http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=6812

[4]Puede descargarse de: http://www.ogre3d.org/download/sdk

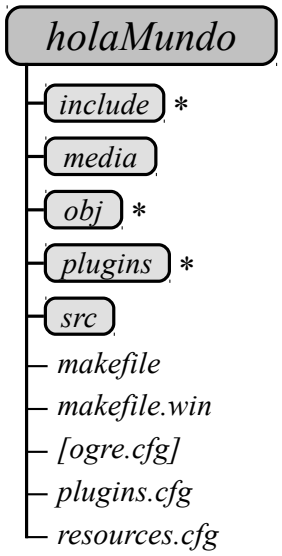


Figura 1.19: Descripción de directorios del ejemplo "*Hola Mundo*".

- El directorio `media` contendrá los archivos de geometría, animaciones y texturas necesarios para ejecutar el ejemplo. Todos estos archivos serán cargados por el *ResourceManager*.
- En `obj` se generarán automáticamente los ficheros objeto compilados a partir de los fuentes existentes en `src`.
- El diretorio `plugins` contendrá los plugins de Ogre. En GNU/Linux, podemos crear un enlace simbólico de este directorio al lugar donde se encuentran los plugins (archivos `.so`) en disco. En Windows deberemos copiar los `.dll` a este directorio. La ruta de este directorio, como veremos más adelante, se debe indicar en el archivo `plugins.cfg`.
- El directorio `src` contiene los ficheros fuente de la aplicación. Gracias al `makefile` que veremos a continuación, la compilación de todos los ficheros fuente en objetos binarios se realiza automáticamente.

Para compilar el ejemplo, definiremos un `makefile` para ambos sistemas operativos. En el siguiente listado se muestra el listado para GNU/Linux.

En la línea [0] se define el nombre del ejecutable que queremos obtener. A continuación en las líneas [1-3] se definen los directorios para los archivos fuente, objetos y cabeceras. Las líneas [7] y [9] definen los flags para la compilación y enlazado respectivamente.

El makefile construido permite indicar el modo de compilación, utilizando unos flags de compilación en modo *Debug* y otros en modo *Release*. Si llamamos a make con `mode=release`, se utilizarán los flags de compilación optimizada. En otro caso, utilizaremos la compilación con símbolos para el depurado posterior con *GDB*.

En las líneas [20-21] se utilizan las funciones de *make* para generar la lista de objetos a partir de los fuentes `.cpp` existentes en el directorio apuntado por `DIRSRC`. De este modo, se obtienen los objetos con el mismo nombre que los fuentes, pero situados en el directorio indicado por `DIROBJ`.

Finalmente, en las líneas [32-37] se emplean las reglas de compilación implícitas de *make* para generar el ejecutable. Se incluye al final la típica regla de `clean` para limpiar los temporales obtenidos.

De este modo, para compilar el ejemplo en modo *release* ejecutaremos en un terminal

```
make mode=release
```

Si no indicamos modo, o indicamos explícitamente *mode=debug*, se utilizarán los flags de compilación en modo debug.

Listado 1.1: Makefile genérico para GNU/Linux

```
0  EXEC := helloWorld
1  DIRSRC := src/
2  DIROBJ := obj/
3  DIRHEA := include/
4  CXX := g++
5
6  # Flags de compilacion ------------------------------------------
7  CXXFLAGS := -I $(DIRHEA) -Wall `pkg-config --cflags OGRE`
8  # Flags del linker ----------------------------------------------
9  LDFLAGS := `pkg-config --libs OGRE` -lGL -lOIS -lstdc++
10
11 # Modo de compilacion (-mode=release -mode=debug) ----------------
12 ifeq ($(mode), release)
13     CXXFLAGS += -O2 -D_RELEASE
14 else
15     CXXFLAGS += -g -D_DEBUG
16     mode := debug
17 endif
18
19 # Obtencion automatica de la lista de objetos a compilar ----------
20 OBJS := $(subst $(DIRSRC), $(DIROBJ), \
21     $(patsubst %.cpp, %.o, $(wildcard $(DIRSRC)*.cpp)))
22
23 .PHONY: all clean
24
25 all: info $(EXEC)
26
27 info:
28     @echo '-----------------------------------------------------'
29     @echo '>>> Using mode $(mode)'
30     @echo '     (Please, call "make" with [mode=debug|release])'
31     @echo '-----------------------------------------------------'
32 # Enlazado ------------------------------------------------------
33 $(EXEC): $(OBJS)
34     $(CXX) $(LDFLAGS) -o $@ $^
35 # Compilacion ---------------------------------------------------
36 $(DIROBJ)%.o: $(DIRSRC)%.cpp
37     $(CXX) $(CXXFLAGS) -c $< -o $@
38 # Limpieza de temporales ----------------------------------------
39 clean:
40     rm -f *.log $(EXEC) *~ $(DIROBJ)* $(DIRSRC)*~ $(DIRHEA)*~
```

El *makefile* en su versión para plataformas windows se ha nombrado como *makefile.win*. Básicamente es similar al estudiado para GNU/Linux, pero cambian los flags de compilación. Para compilar, tendremos que ejecutar el make de *MinGW* que se denomina `mingw32-make`. De este modo, para compilar ejecutaremos desde una consola:

```
mingw32-make -f makefile-windows mode=release
```

Como se muestra en la Figura 1.19, además de los archivos para make, en el directorio raiz se encuentran tres archivos de configuración. Estudiaremos a continuación su contenido:

- **ogre.cfg**. Este archivo, si existe, intentará ser cargado por el objeto *Root*. Si el archivo no existe, o está creado de forma incorrecta, se le mostrará al usuario una ventana para configurar los parámetros (resolución, método de anti-aliasing, profundidad de color, etc...). En el ejemplo que vamos a realizar, se fuerza a

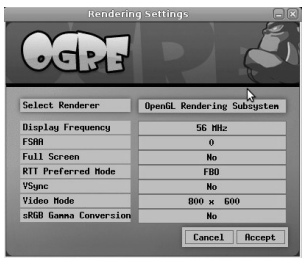


Figura 1.20: Ventana de configuración de las propiedades de *Rendering*.

que siempre se abra el diálogo (ver Figura 1.20), recuperando los parámetros que el usuario especificó en la última ejecución.

- **plugins.cfg**. Como hemos comentado anteriormente, un *plugin* es cualquier módulo que implementa alguno de los interfaces de *plugins* de Ogre (como *SceneManager* o *RenderSystem*). En este archivo se indican los plugins que debe cargar OGRE, así como su localización. En el ejemplo del holaMundo, el contenido del archivo indica la ruta al lugar donde se encuentran los plugins (`PluginFolder`), así como el Plugin que emplearemos (pueden especificarse varios en una lista) para el rendering con OpenGL.

Plugins.cfg en Windows

En sistemas Windows, como no es posible crear enlaces simbólicos, se deberían copiar los archivos `.dll` al directorio plugins del proyecto e indicar la ruta relativa a ese directorio en el archivo `plugins.cfg`. Esta aproximación también puede realizarse en GNU/Linux copiando los archivos `.so` a dicho directorio.

- **resources.cfg**. En esta primera versión del archivo de recursos, se especifica el directorio general desde donde OGRE deberá cargar los recursos asociados a los nodos de la escena. A lo largo del curso veremos cómo especificar manualmente más propiedades a este archivo.

Una vez definida la estructura de directorios y los archivos que forman el ejemplo, estudiaremos la docena de líneas de código que tiene nuestro *Hola Mundo*.

Listado 1.2: El main.cpp del Hola Mundo en OGRE

```
0  #include <ExampleApplication.h>
1
2  class SimpleExample : public ExampleApplication {
3    public : void createScene() {
4      Ogre::Entity *ent = mSceneMgr->createEntity("Sinbad", "Sinbad.
           mesh");
5      mSceneMgr->getRootSceneNode()->attachObject(ent);
6    }
7  };
8
9  int main(void) {
10   SimpleExample example;
11   example.go();
12
13   return 0;
14 }
```

Como vemos, en el main se crea una instancia de la clase *SimpleExample* (definida en las líneas 2–7). Esta clase es derivada de la clase base *ExampleApplication*, que se proporciona en el SDK de OGRE para facilitar el aprendizaje de la biblioteca.

Nombres en entidades

Si no indicamos ningún nombre, OGRE elegirá automáticamente un nombre para la misma. El nombre debe ser único. Si el nombre existe, OGRE devolverá una excepción.

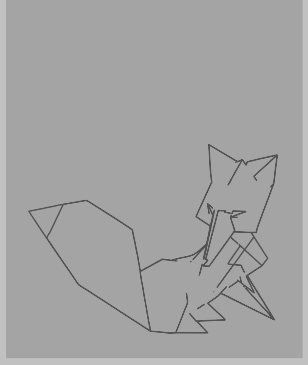
En esta clase definimos el método público *CreateScene*. En la línea 4 llamamos al *SceneManager* (creado automáticamente por la clase base *ExampleApplication*) a través de su puntero, solicitando la creación de una entidad con nombre único `Sinbad`, asociado a `Sinbad.mesh` (que se encontrará en el directorio especificado en `resources.cfg`). Esta llamada nos creará la entidad, y nos devolverá un puntero a un objeto de ese tipo.

A continuación, adjuntamos la entidad creada a la escena. Para ello, accedemos al método *attachObject()* del nodo raíz devuelto por *getRootSceneNode()*.

Si compilamos y ejecutamos el *holaMundo*, primero nos aparecerá una ventana para configurar las opciones de rendering (como se muestra en la Figura 1.20). Cuando pulsemos [Accept], se abrirá una ventana con el modelo cargado (muy pequeñito). Si pulsamos la flecha del cursor [↑], el modelo se acercará. Podemos variar la posición de la cámara y la rotación del modelo (así como su modo de rendering) con los otros tres cursores [*leftarrow*] [*downarrow*] [*rightarrow*] , y las teclas [A], [S], [D], [W] y [R]. La salida del modelo se muestra en la Figura 1.21.

Figura 1.21: Resultado, tras ajustar el modelo, del Hello World.

La clase *ExampleApplication* nos abstrae de la complejidad de crear una aplicación desde cero con OGRE. Utilizaremos esta clase base en algunos ejemplos más del siguiente capítulo, pero la abandonaremos rápidamente para controlar todos los aspectos relativos a la inicialización (carga de Plugins, definición del *RenderSystem*, gestión de la cámara virtual, control de eventos, etc).



Capítulo 2

Matemáticas para Videojuegos

Carlos González Morcillo

En este capítulo comenzaremos a estudiar las transformaciones afines básicas que serán necesarias para en el desarrollo de Videojuegos. Las transformaciones son herramientas imprescindibles para cualquier aplicación que manipule geometría o, en general, objetos descritos en el espacio 3D. La mayoría de APIs y motores gráficos que trabajan con gráficos 3D (como OGRE) implementan clases auxiliares para facilitar el trabajo con estos tipos de datos.

Figura 2.1: Aunque en desarrollo de videojuegos se hace uso de prácticamente todas las áreas de las matemáticas (desde trigonometría, álgrebra, estadística o cálculo), en este capítulo nos centraremos en los fundamentos más básicos del álgebra lineal.

2.1. Puntos, Vectores y Coordenadas

Los videojuegos necesitan posicionar objetos en el espacio 3D. El motor gráfico debe gestionar por tanto la posición, orientación y escala de estos objetos (y sus cambios a lo largo del tiempo). Como vimos en el capítulo 1, en gráficos interactivos suelen emplearse representaciones poligionales basadas en triángulos para mejorar la eficiencia. Los vértices de estos triángulos se representan mediante puntos, y las normales mediante vectores. Veamos a continuación algunos conceptos básicos relacionados con los puntos y los vectores.

2.1.1. Puntos

Un *punto* puede definirse como una localización en un espacio n-dimensional. En el caso de los videojuegos, este espacio suele ser bidimensional o tridimensional. El tratamiento discreto de los valores asociados a la posición de los puntos en el espacio exige elegir el tamaño mínimo y máximo que tendrán los objetos en nuestro videojuego. Es crítico definir el convenio empleado relativo al sistema de coordenadas y las unidades entre programadores y artistas. Existen diferentes sistemas de coordenadas en los que pueden especificarse estas posiciones, como se puede ver en la Figura 2.2:

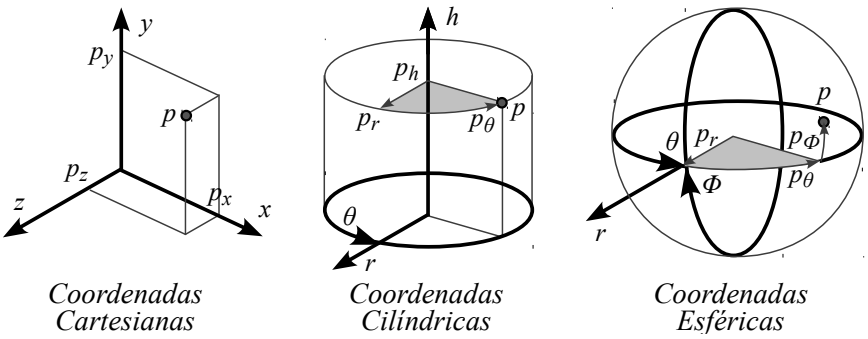


Figura 2.2: Representación de un punto empleando diferentes sistemas de coordenadas.

- **Coordenadas Cartesianas**. Es sin duda el sistema de coordenadas más habitual. Este sistema de coordenadas define ejes perpediculares para especificar la posición del punto en el espacio. Como se muestra en la figura 2.3, existen dos convenios para la definición de los ejes de coordenadas cartesianas; según la regla de la mano derecha o de la mano izquierda. Es muy sencillo convertir entre ambos convenios; basta con invertir el valor del eje que cambia.
- **Coordenadas Cilíndricas**. En este sistema se utiliza un eje vertical para definir la altura h, un eje radial r que mide la distancia con ese eje h, y un ángulo de rotación θ definido en la circunferencia sobre ese radio.
- **Coordenadas Esféricas**. Este sistema emplea dos ángulos ϕ y θ, y una distancia radial r.

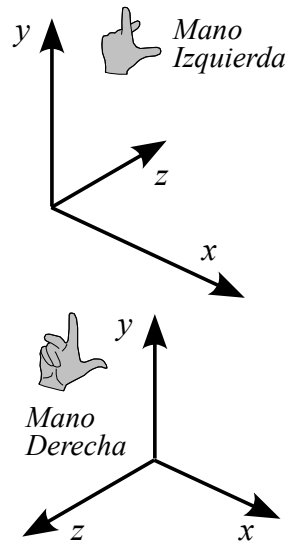


Figura 2.3: Convenios para establecer los sistemas de coordenadas cartesianas.

Coordenadas en OGRE. OGRE utiliza el convenio de la **mano derecha**. El pulgar (eje positivo X) y el índice (eje positivo Y) definen el plano de la pantalla. El pulgar apunta hacia la derecha de la pantalla, y el índice hacia el "techo". El anular define el eje positivo de las Z, saliendo de la pantalla hacia el observador.

2.1.2. Vectores

Un vector es una tupla n-dimensional que tiene una longitud (denominada *módulo*), una dirección y un sentido. Puede ser representado mediante una *flecha*. Dos vectores son iguales si tienen la misma longitud, dirección y sentido. Los vectores son entidades *libres* que no están ancladas a ninguna posición en el espacio.

Convenio en OGRE

En OGRE se utilizan las clases `Vector2` y `Vector3` para trabajar indistintamente con puntos y vectores en 2 y 3 dimensiones.

Como en algún momento es necesario representar los vectores como tuplas de números en nuestros programas, en muchas bibliotecas se emplea la misma clase `Vector` para representar puntos y vectores en el espacio. Es imprescindible que el programador distinga en cada momento con qué tipo de entidad está trabajando.

A continuación describiremos brevemente algunas de las operaciones habituales realizadas con vectores.

Suma y resta de vectores

La suma de vectores se puede realizar gráficamente empleando la *regla del paralelogramo*, de modo de la suma puede calcularse como el resultado de unir el inicio del primero con el final del segundo (situando el segundo a continuación del primero). La suma, al igual que con números reales es conmutativa. El vector suma gráficamente "completa el triángulo" (ver Figura 2.4).

Dado un vector a, el vector $-a$ tiene el mismo módulo y dirección que a, pero sentido contrario.

La resta de un vector con otro puede igualmente representarse mediante un paralelogramo. En el diagrama de la Figura 2.4, puede verse cómo efectivamente la resta del vector $b - a$ puede verse como la suma de $(b - a) + a$, como hemos indicado anteriormente.

El resultado de multiplicar un vector a por un valor escalar obtiene como resultado un vector con la misma dirección y sentido, pero con el módulo *escalado* al factor por el que se ha multiplicado. Gráficamente, la operación de multiplicación puede verse como el efecto de *estirar* del vector manteniendo su dirección y sentido.

Las operaciones de suma y resta, multiplicación por un escalar e inversión, se realizan componente a componente del vector:

$$\begin{aligned} a + b &= [(a_x + b_x), (a_y + b_y), (a_z + b_z)] \\ a - b &= [(a_x - b_x), (a_y - b_y), (a_z - b_z)] \\ n \cdot a &= (n \cdot a_x, n \cdot a_y, n \cdot a_z) \\ -a &= (-a_x, -a_y, -a_z) \end{aligned}$$

Recordemos que, la suma de dos vectores da como resultado un vector, mientras que la suma de un punto y un vector se interpreta como la obtención del *punto destino* de aplicar la transformación del vector sobre el punto. La resta de dos puntos ($p_b - p_a$) da como resultado un vector, con módulo igual a la distancia existente entre ambos puntos, la dirección de la recta que pasa por ambos puntos y el sentido de ir del punto p_a a p_b.

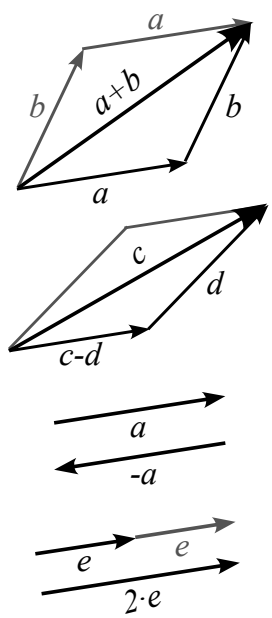


Figura 2.4: Representación de algunas operaciones con vectores. Comenzando por arriba: suma y resta de vectores, inversión y multiplicación por escalar.

Módulo y normalización

El módulo de un vector es un valor escalar que representa la longitud del mismo. Puede calcularse como:

$$|a| = \sqrt{a_x^2 + a_y^2 + a_z^2}$$

Si dividimos un vector por su longitud (es decir, escalamos el vector a por el factor $1/|a|$), obtenemos como resultado un vector con la misma dirección y sentido, pero de módulo la unidad (vector *unitario*). Esta operación se denomina *normalización*, y da como resultado un vector *normalizado* que se emplea en gran cantidad de algoritmos en programación de videojuegos.

$$a_N = \frac{a}{|a|}$$

Los vectores pueden multiplicarse entre sí, pero a diferencia de los escalares, admiten dos operaciones de multiplicación; el *producto escalar* (que da como resultado un escalar), y el *producto vectorial* (que lógicamente da como resultado otro vector). Veamos a continuación cómo se definen estas dos operaciones y algunas de sus aplicaciones prácticas.

Producto Escalar

El *producto escalar* (*dot product*) es conmutativo ($a \cdot b = b \cdot a$), y se calcula como:

$$a \cdot b = a_x b_x + a_y b_y + a_z b_z \tag{2.1}$$

También puede ser calculado mediante la siguiente expresión que relaciona el ángulo θ que forman ambos vectores.

$$a \cdot b = |a|\,|b|\,cos(\theta) \tag{2.2}$$

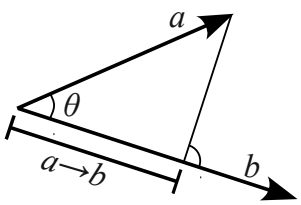


Figura 2.5: La proyección de a sobre b obtiene como resultado un escalar.

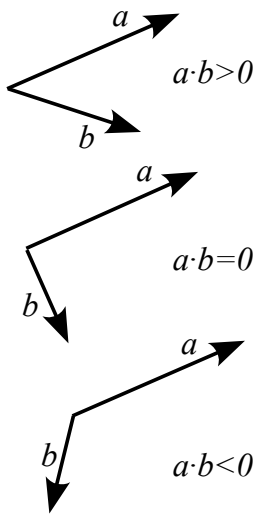


Figura 2.6: Algunos ejemplos de uso del producto escalar.

Combinando las expresiones 2.1 y 2.2, puede calcularse el ángulo que forman dos vectores (operación muy utilizada en informática gráfica).

Otra operación muy útil es el cálculo de la *proyección* de un vector sobre otro, que se define $a \rightarrow b$ como la longitud del vector a que se proyecta mediante un ángulo recto sobre el vector b (ver Figura 2.5).

$$a \rightarrow b = |a|\, cos(\theta) = \frac{a \cdot b}{|b|}$$

El producto escalar se emplea además para detectar si dos vectores tienen la misma dirección, o si son perpendiculares, o si tienen la misma dirección o direcciones opuestas (para, por ejemplo, eliminar geometría que no debe ser representada).

- **Colineal**. Dos vectores son colineales si se encuentran definidos en la misma línea recta. Si *normalizamos* ambos vectores, y aplicamos el producto escalar podemos saber si son *colineales* con el mismo sentido ($a \cdot b = 1$) o sentido opuesto ($a \cdot b = -1$).
- **Perpendicular**. Dos vectores son perpendiculares si el ángulo que forman entre ellos es 90 (o 270), por lo que $a \cdot b = 0$.
- ***Misma* dirección**. Si el ángulo que forman entre ambos vectores está entre 270 y 90 grados, por lo que $a \cdot b > 0$ (ver Figura 2.6).
- **Dirección *Opuesta***. Si el ángulo que forman entre ambos vectores es mayor que 90 grados y menor que 270, por lo que $a \cdot b < 0$.

Esta interpretación de *misma* dirección y dirección *opuesta* no es literal. Nos servirá para comprobar si el vector normal de una cara poligonal está *de frente* a la cámara virtual (si tiene dirección opuesta al vector *look*) de la cámara, o por el contrario está siendo visto *de espaldas*.

Producto Vectorial

Mediante el *producto vectorial* (*cross product*) de dos vectores se obtiene otro vector que es perpendicular a los dos vectores originales.

$$a \times b = [(a_y b_z - a_z b_y), (a_z b_x - a_x b_z), (a_x b_y - a_y b_x)]$$

El sentido del vector resultado del producto escalar sigue la regla de la *mano derecha* (ver Figura 2.7): si *agarramos* $a \times b$ con la palma de la mano, el pulgar apunta en el sentido positivo del vector producto si el giro del resto de los dedos va de a a b. En caso contrario, el sentido del vector es el inverso. Por tanto, el producto vectorial no es conmutativo.

El módulo del producto vectorial está directamente relacionado con el ángulo que forman ambos vectores θ. Además, el módulo del producto vectorial de dos vectores es igual al área del paralelogramo formado por ambos vectores (ver Figura 2.7).

$$|a \times b| = |a|\,|b|\,sin\theta$$

El producto vectorial se utiliza en gran cantidad de situaciones. El cálculo del vector perpendicular a dos vectores es útil para calcular el vector normal asociado a un triángulo (habitualmente se devuelve normalizado para su posterior utilización en la etapa de *shading*).

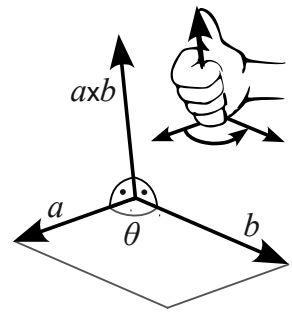
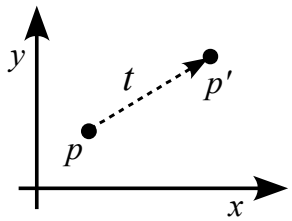


Figura 2.7: Representación del produto vectorial. El módulo del producto vectorial $a \times b$ es igual al área del paralelogramo representado en la figura.



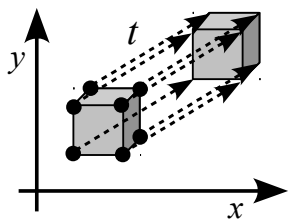


Figura 2.8: Arriba. Traslación de un punto p a p' empleando el vector t. **Abajo**. Es posible trasladar un objeto poligonal completo aplicando la traslación a todos sus vértices.

Más Mates?, Síií!! Existen multitud de objetos matemáticos empleados en gráficos interactivos, como rayos, planos, segmentos, así como estructuras de datos para la aceleración de los cálculos. Estos elementos serán estudiados en cada sección que haga uso de ellos.

A continuación describiremos las transformaciones geométricas más comunes empleadas en gráficos por computador. Para introducir los conceptos generales asociados a las transformaciones, comenzaremos con una discusión sobre las operaciones en 2D para pasar a la notación matricial 2D y, posteriormente, a la generalización tridimensional empleando coordenadas homogeneas.

2.2. Transformaciones Geométricas

En la representación de gráficos 3D es necesario contar con herramientas para la transformación de los objetos básicos que compondrán la escena. En gráficos interactivos, estas primitivas son habitualmente conjuntos de triángulos que definen mallas poligonales. Las operaciones que se aplican a estos triángulos para cambiar su posición, orientación y tamaño se denominan **transformaciones geométricas**. En general podemos decir que una transformación toma como entrada elementos como vértices y vectores y los convierte de *alguna manera*.

La transformación básica bidimensional más sencilla es la **traslación**. Se realiza la traslación de un punto mediante la suma de un vector de desplazamiento a las coordenadas iniciales del punto, para obtener una nueva posición de coordenadas. Si aplicamos esta traslación a *todos* los puntos del objeto, estaríamos desplazando ese objeto de una posición a otra. De este modo, podemos definir la traslación como la suma de un vector libre de traslación t a un punto original p para obtener el punto trasladado p' (ver Figura 2.8). Podemos expresar la operación anterior como:

$$p'_x = p_x + t_x \qquad p'_y = p_y + t_y \tag{2.3}$$

De igual modo podemos expresar una **rotación** de un punto $p = (x, y)$ a una nueva posición rotando un ángulo θ respecto del origen de coordenadas, especificando el eje de rotación y un ángulo θ. Las coordenadas iniciales del punto se pueden expresar como (ver Figura 2.9):

$$p_x = d\,cos\alpha \qquad p_y = d\,sen\alpha \tag{2.4}$$

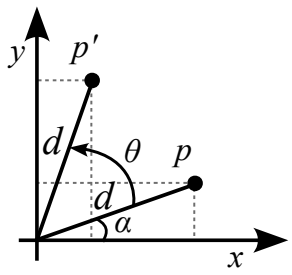


Figura 2.9: Rotación del punto p un ángulo θ respecto del origen de coordenadas.

Siendo d la distancia entre el punto y el origen del sistema de coordenadas. Así, usando identidades trigonométricas se pueden expresar las coordenadas transformadas como la suma de los ángulos del punto original α y el que queremos rotar θ como:

$$p'_x = d\,cos(\alpha + \theta) = d\,cos\alpha\,cos\theta - d\,sen\alpha\,sen\theta$$
$$p'_y = d\,sen(\alpha + \theta) = d\,cos\alpha\,sen\theta - d\,sen\alpha\,cos\theta$$

Que sustituyendo en la ecuación 2.4, obtenemos:

$$p'_x = p_x\,cos\theta - p_y\,sen\theta \qquad p'_y = p_x\,sin\theta - p_y\,cos\theta \tag{2.5}$$

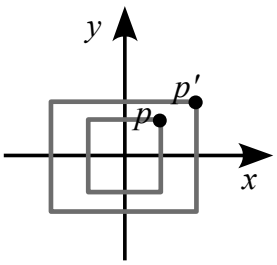


Figura 2.10: Conversión de un cuadrado a un rectángulo empleando los factores de escala $S_x = 2, S_y = 1.5$.

De forma similar, un **cambio de escala** de un objeto bidimensional puede llevarse a cabo multiplicando las componentes x, y del objeto por el factor de escala S_x, S_y en cada eje. Así, como se muestra en la Figura 2.10 un cambio de escala se puede expresar como:

$$p'_x = p_x S_x \qquad p'_y = p_y S_y \tag{2.6}$$

Cuando queremos cambiar la localización de un objeto, habitualmente necesitamos especificar una **combinación de traslaciones y rotaciones** en el mismo (por ejemplo, cuando cogemos el teléfono móvil de encima de la mesa y nos lo guardamos en el bolsillo, sobre el objeto se aplican varias traslaciones y rotaciones). Es interesante por tanto disponer de alguna representación que nos permita combinar transformaciones de una forma eficiente.

2.2.1. Representación Matricial

Homogeneízate!!

Gracias al uso de coordenadas homogéneas es posible representar las ecuaciones de transformación geométricas como multiplicación de matrices, que es el método estándar en gráficos por computador (soportado en hardware por las tarjetas aceleradoras gráficas).

En muchas aplicaciones gráficas los objetos deben transformarse geométricamente de forma constante (por ejemplo, en el caso de una animación, en la que en cada *frame* el objeto debe cambiar de posición. En el ámbito de los videojuegos, es habitual que aunque un objeto permanezca inmóvil, es necesario cambiar la posición de la cámara virtual para que se ajuste a la interacción con el jugador.

De este modo resulta crítica la eficiencia en la realización de estas transformaciones. Como hemos visto en la sección anterior, las ecuaciones 2.3, 2.5 y 2.6 nos describían las operaciones de traslación, rotación y escalado. Para la primera es necesario realizar una *suma*, mientras que las dos últimas requieren *multiplicaciones*. Sería conveniente poder **combinar las transformaciones** de forma que la posición final de las coordenadas de cada punto se obtenga de forma directa a partir de las coordenadas iniciales.

Si reformulamos la escritura de estas ecuaciones para que todas las operaciones se realicen multiplicando, podríamos conseguir homogeneizar estas transformaciones.

Si añadimos un término extra (parámetro homogéneo h) a la representación del punto en el espacio (x, y), obtendremos la **representación homogénea** de la posición descrita como (x_h, y_h, h). Este *parámetro homogéneo* h es un valor distinto de cero tal que $x = x_h/h$, $y = y_h/h$. Existen, por tanto infinitas representaciones homogéneas equivalentes de cada par de coordenadas, aunque se utiliza normalmente $h = 1$. Como veremos en la sección 2.3, no siempre el parámetro h es igual a uno.

Puntos y Vectores

En el caso de puntos, la componente homogénea $h = 1$. Los vectores emplean como parámetro $h = 0$.

De este modo, la operación de traslación, que hemos visto anteriormente, puede expresarse de forma matricial como:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2.7}$$

Al resolver la multiplicación matricial se obtienen un conjunto de ecuaciones equivalentes a las enunciadas en 2.3. De forma análoga, las operaciones de rotación T_r y escalado T_s tienen su equivalente matricial homogéneo.

$$T_r = \begin{bmatrix} cos\theta & -sen\theta & 0 \\ sen\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad T_s = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.8}$$

Las transformaciones inversas pueden realizarse sencillamente cambiando el signo en el caso de la traslación y rotación (distancias y ángulos negativos), y en el caso de la escala, utilizando los valores $1/S_x$ y $1/S_y$.

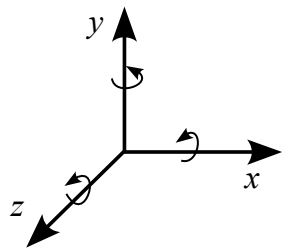


Figura 2.11: Sentido de las rotaciones positivas respecto de cada eje de coordenadas.

Las **transformaciones en el espacio 3D** requieren simplemente añadir el parámetro homogéneo y describir las matrices (en este caso 4x4). Así, las traslaciones T_t y escalados T_s en 3D pueden representarse de forma homogénea mediante las siguientes matrices:

$$T_t = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T_s = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.9}$$

Las rotaciones requieren distinguir el eje sobre el que se realizará la rotación. Las **rotaciones positivas** alrededor de un eje se realizan en sentido opuesto a las agujas del reloj, cuando se está mirando a lo largo de la mitad positiva del eje hacia el origen del sistema de coordenadas (ver Figura 2.11).

Las expresiones matriciales de las rotaciones son las siguientes:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sen\theta & 0 \\ 0 & sen\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_y = \begin{bmatrix} cos\theta & 0 & sen\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sen\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_z = \begin{bmatrix} cos\theta & -sen\theta & 0 & 0 \\ sen\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

Las tres transformaciones estudiadas (traslación, rotación y escalado) son ejemplos de **transformaciones afines**, en las que cada una de las coordenadas transformadas se pueden expresar como una función lineal de la posición origen, y una serie de constantes determinadas por el tipo de transformación.

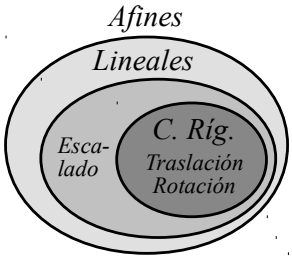


Figura 2.12: Esquema de subclases de transformaciones afines y operaciones asociadas.

Una subclase de las *transformaciones afines*, es la formada por las transformaciones lineales, que permiten aplicar todas las transformaciones mediante multiplicaciones. Como hemos visto anteriormente, las traslación afines no homogénea no es una operación lineal (porque no puede realizarse mediante una multiplicación). Las transformaciones afines mantienen la mayoría de las propiedades de los objetos, excepto los ángulos y las distancias. Aplicando transformaciones afines se mantiene la *colineridad* (ver Tabla 2.1), por lo que las líneas paralelas seguirán siendo paralelas.

Como caso particular de estudio, las *transformaciones de cuerpo rígido* preservan todas las propiedades geométricas de los objetos. Cualquier combinación de *rotaciones y traslaciones homogéneas* son transformaciones de cuerpo rígido.

Tabla 2.1: Propiedades geométricas preservadas según la clase de transformación: Transformaciones afines, Lineales y de Cuerpo Rígido.

Propiedad	**T. Afines**	**T. Lineales**	**T. Cuerpo Rígido**
Ángulos	No	No	Sí
Distancias	No	No	Sí
Ratios de distancias	Sí	Sí	Sí
Líneas paralelas	Sí	Sí	Sí
Líneas rectas	Sí	Sí	Sí

2.2.2. Transformaciones Inversas

En muchas situaciones resulta interesante calcular la inversa de una matriz. Un ejemplo típico es en la resolución de ecuaciones, como en el caso de la expresión $A = Bx$. Si queremos obtener el valor de B, tendríamos $B = A/x$. Por desgracia, las matrices no tienen asociado un operador de división, por lo que debemos usar el concepto de *matriz inversa*.

Para una matriz A, se define su inversa A^{-1} como la matriz que, multiplicada por A da como resultado la matriz identidad I:

$$A \cdot A^{-1} = A^{-1} \cdot A = I \tag{2.11}$$

Matrices Inversas

No todas las matrices tienen inversa (incluso siendo cuadradas). Un caso muy simple es una matriz cuadrada cuyos elementos son cero.

Tanto la matriz A como su inversa deben ser *cuadradas* y del mismo tamaño. Otra propiedad interesante de la inversa es que la inversa de la inversa de una matriz es igual a la matriz original $(A^{-1})^{-1} = A$.

En la ecuación inicial, podemos resolver el sistema utilizando la matriz inversa. Si partimos de $A = B \cdot x$, podemos multiplicar ambos términos a la izquierda por la inversa de B, teniendo $B^{-1} \cdot A = B^{-1} \cdot B \cdot x$, de forma que obtenemos la matriz identidad $B^{-1} \cdot A = I \cdot x$, con el resultado final de $B^{-1} \cdot A = x$.

En algunos casos el cálculo de la matriz inversa es directo, y puede obtenerse de forma intuitiva. Por ejemplo, en el caso de una traslación pura (ver ecuación 2.9), basta con emplear como factor de traslación el mismo valor en negativo. En el caso de escalado, como hemos visto bastará con utilizar $1/S$ como factor de escala.

Cuando se trabaja con matrices compuestas, el cálculo de la inversa tiene que realizarse con métodos generales, como por ejemplo el método de eliminación de Gauss o la traspuesta de la matriz adjunta.

2.2.3. Composición

Como hemos visto en la sección anterior, una de las principales ventajas derivadas del trabajo con sistemas homogéneos es la **composición de matrices**. Matemáticamente esta *composición* se realiza multiplicando las matrices en un orden determinado, de forma que es posible obtener la denominada **matriz de transformación neta** M_N resultante de realizar sucesivas transformaciones a los puntos. De este modo, bastará con multiplicar la M_N a cada punto del modelo para obtener directamente su posición final. Por ejemplo, si P es el punto original y P' es el punto transformado, y $T_1 \cdots T_n$ son transformaciones (rotaciones, escalados, traslaciones) que se aplican al punto P, podemos expresar la transformación neta como:

$$P' = T_n \times \cdots \times T_2 \times T_1 \times P$$

$$\begin{bmatrix} RS_{11} & RS_{12} & RS_{13} & P_x \\ RS_{21} & RS_{22} & RS_{23} & P_y \\ RS_{31} & RS_{32} & RS_{33} & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figura 2.13: El formato de la matriz de transformación neta permite identificar la posición final del objeto (traslación) en la cuarta columna $P_x P_y P_z$. La matriz 3×3 interior combina las rotaciones y escalados que se aplicarán al objeto.

Este orden de multiplicación de matrices es el habitual empleado en gráficos por computador, donde las transformaciones se premultiplican (la primera está más cerca del punto original P, más a la derecha.

La matriz de transformación neta M_N se definiría como $M_N = T_n \times \cdots T_2 \times T_1$, de tal forma que sólo habría que calcularla una vez para todos los puntos del modelo y aplicarla a todos vértices en su posición original para obtener su posición final. De este modo, si un objeto poligonal está formado por V vértices, habrá que calcular la matriz de transformación neta M_N y aplicarla una vez a cada vértice del modelo.

$$P' = M_N \times P$$

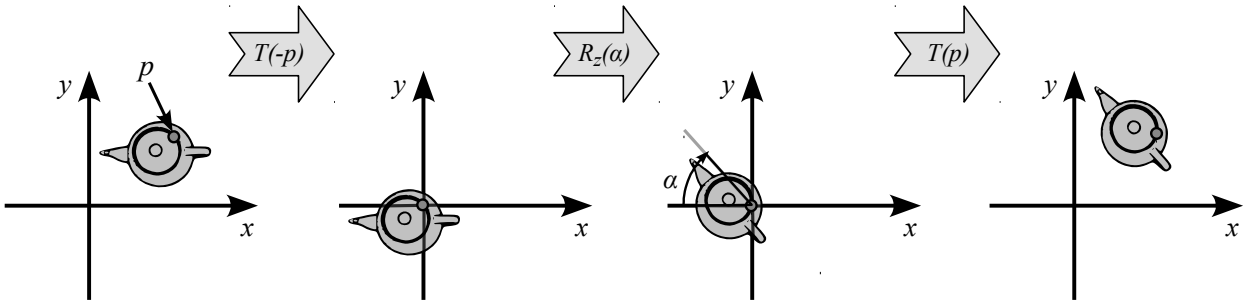


Figura 2.14: Secuencia de operaciones necesarias para rotar una figura respecto de un origen arbitrario.

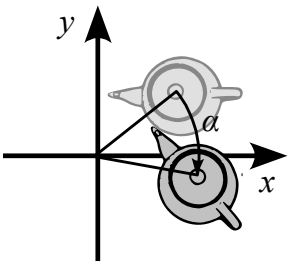


Figura 2.15: Resultado de aplicar directamente la rotación $R_z(\alpha)$ respecto del SRU. Puede comprobarse el diferente resultado obtenido en comparación con el de la figura 2.14.

Conmutatividad. Recordemos que la multiplicación de matrices es asociativa, pero *no* es conmutativa, por lo que el orden de aplicación de las transformaciones es importante (ver Figura 2.16). La propiedad asociativa se utiliza para *resumir* en la *Matriz de Transformación Neta* la secuencia de transformaciones que se aplicarán sobre los modelos. De este modo, bastará con multiplicar una única matriz todos los vértices del modelo.

Otro aspecto a tener en cuenta es que la expresión de las transformaciones para trabajar con coordenadas homogeneas, que se han comentado en las ecuaciones 2.9 y 2.10 se refieren al **Sistema de Referencia Universal** (*SRU*) o Sistema de Referencia Global.

Esto implica que si se quiere realizar una transformación respecto de un punto distinto a ese origen del sistema de referencia universal, habrá que hacer coincidir primero el punto con el origen del sistema de referencia, aplicar la transformación y devolver el objeto a su posición original. Así, en el ejemplo de la Figura 2.14 si queremos rotar el objeto respecto del punto p es necesario, primero trasladar el objeto para que su origen quede situado en el origen del *SRU*, luego aplicar la rotación, y finalmente aplicar la traslación inversa. De este modo, la Matriz Neta quedaría definida como $M_N = T_{-p} \times R_z \times T_p$.

2.3. Perspectiva: Representación Matricial

Como vimos en la sección 1.2.4, en la proyección en perspectiva se tomaban como entrada las coordenadas de visualización, generando la proyección de los objetos sobre el plano de imagen.

Vimos en la Figura 1.9 que, empleando triángulos semejantes, obteníamos las siguientes coordenadas:

$$\frac{p'_x}{p_x} = \frac{-d}{p_z} \quad \Leftrightarrow \quad p'_x = \frac{-d\, p_x}{p_z} \tag{2.12}$$

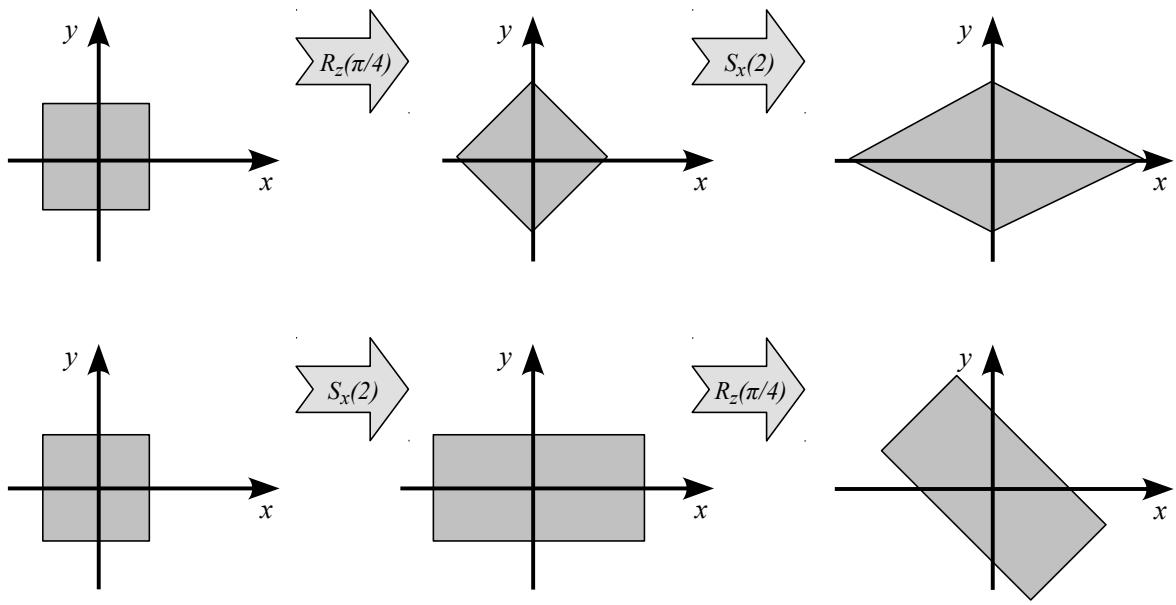


Figura 2.16: La multiplicación de matrices no es conmutativa, por lo que el orden de aplicación de las transformaciones es relevante para el resultado final. Por ejemplo, la figura de arriba primero aplica una rotación y luego el escalado, mientras que la secuencia inferior aplica las transformaciones en orden inverso.

De igual forma obtenemos la coordenada $p'_y = -d\,p_y/p_z$, y $p'_z = -d$. Estas ecuaciones se pueden expresar fácilmente de forma matricial como se muestra en la siguiente expresión (siendo M_p la matriz de proyección en perspectiva).

$$p' = M_p\,p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ -p_z/d \end{bmatrix} = \begin{bmatrix} -d\,p_x/p_z \\ -d\,p_y/p_z \\ -d \\ 1 \end{bmatrix} \qquad (2.13)$$

El último paso de la ecuación 2.13 corresponde con la normalización de los componentes dividiendo por el parámetro homogéneo $h = -p_z/d$. De esta forma tenemos la matriz de proyección que nos *aplasta* los vértices de la geometría sobre el plano de proyección. Desafortunadamente esta operación no puede *deshacerse* (no tiene inversa). La geometría una vez *aplastada* ha perdido la información sobre su componente de profundidad. Es interesante obtener una trasnformación en perspectiva que proyecte los vértices sobre el *cubo unitario* descrito previamente (y que sí puede deshacerse).

De esta forma, definimos la *pirámide de visualización* o *frustum*, como la pirámide truncada por un plano paralelo a la base que define los objetos de la escena que serán representados. Esta *pirámide de visualización* queda definida por cuatro vértices que definen el plano

de proyección (left l, right r, top t y bottom b), y dos distancias a los planos de recorte (near n y far f), como se representa en la Figura 2.17. El *ángulo de visión* de la cámara viene determinado por el ángulo que forman l y r (en horizontal) y entre t y b (en vertical).

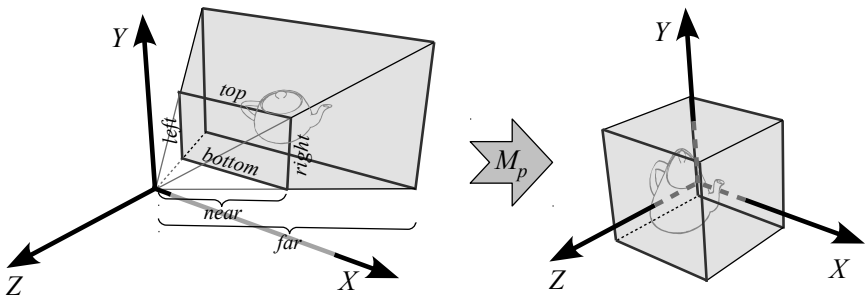


Figura 2.17: La matriz M_p se encarga de transformar la *pirámide de visualización* en el cubo unitario.

Definición *near* y *far*

Un error común suele ser que la definición correcta de los planos *near* y *far* únicamente sirve para limitar la geometría que será representada en el *Frustum*. La definición correcta de estos parámetros es imprescindible para evitar errores de precisión en el Z-Buffer.

La matriz que transforma el *frustum* en el cubo unitario viene dada por la expresión de la ecuación 2.14.

$$M_p = \begin{bmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.14}$$

Un efecto de utilizar este tipo de proyección es que el valor de profundidad normalizado no cambia linealmente con la entrada, sino que se va perdiendo precisión. Es recomendable situar los planos de recorte cercano y lejano (distancias n y f en la matriz) lo más juntos posibles para evitar errores de precisión en el *Z-Buffer* en distancias grandes.

2.4. Cuaternios

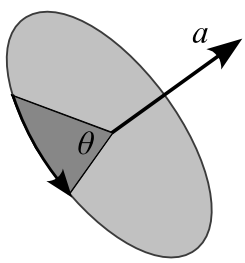


Figura 2.18: Representación de un cuaternio unitario.

Los cuaternios (*quaternion*) fueron propuestos en 1843 por William R. Hamilton como extensión de los números complejos. Los cuaternios se representan mediante una 4-tupla $q = [q_x, q_y, q_z, q_w]$. El término q_w puede verse como un término **escalar** que se añade a los tres términos que definen un **vector** (q_x, q_y, q_z). Así, es común representar el cuaternio como $q = [q_v, q_s]$ siendo $q_v = (q_x, q_y, q_z)$ y $q_s = q_w$ en la tupla de cuatro elementos inicial.

Los cuaternios unitarios, que cumplen la restricción de que $(q_x^2 + q_y^2 + q_z^2 + q_w^2) = 1$, se emplean ampliamente para representar rotaciones en el espacio 3D. El conjunto de todos los cuaternios unitarios definen la hiperesfera unitaria en $\mathbb{R}^4$.

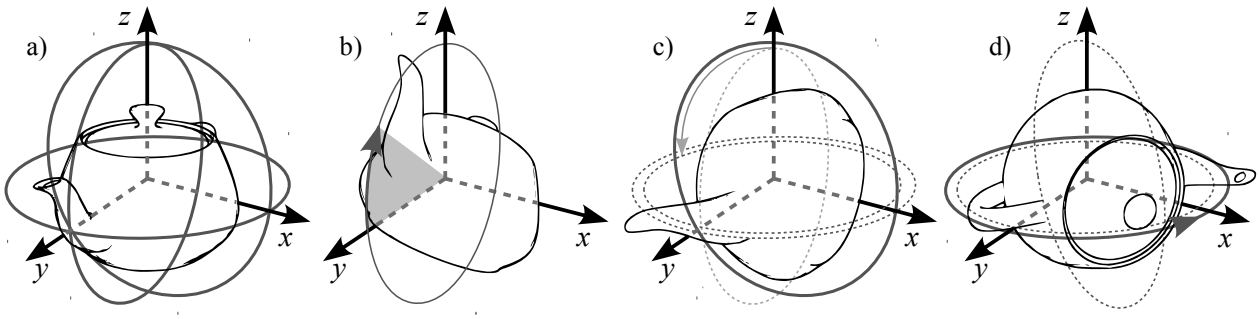


Figura 2.19: El problema del bloqueo de ejes (*Gimbal Lock)*. En **a)** se muestra el convenio de sistema de coordenadas que utilizaremos, junto con las elipses de rotación asociadas a cada eje. En **b)** aplicamos una rotación respecto de x, y se muestra el resultado en el objeto. El resultado de aplicar una rotación de 90^o en y se muestra en **c)**. En esta nueva posición cabe destacar que, por el orden elegido de aplicación de las rotaciones, ahora el eje x y el eje z están alineados, perdiendo un grado de libertad en la rotación. En **d)** se muestra cómo una rotación en z tiene el mismo efecto que una rotación en x, debido al efecto de *Gimbal Lock*.

Si definimos como a al vector unitario que define el eje de rotación, θ como el ángulo de rotación sobre ese eje empleando la regla de *la mano derecha* (si el pulgar apunta en la dirección de a, las rotaciones positivas se definen siguiendo la dirección de los dedos curvados de la mano), podemos describir el cuaternio como (ver Figura 2.18):

$$q = [q_v, q_s] = [a\, sin(\theta/2),\, cos(\theta/2)]$$

De forma que *la parte vectorial* se calcula escalando el vector de dirección unitario por el seno de $\theta/2$, y la parte escalar como el coseno de $\theta/2$. Obviamente, esta multiplicación en la parte vectorial se realiza componente a componente.

Como hemos visto en la sección 2.2.1, es posible utilizar una matriz para *resumir* cualquier secuencia de rotaciones en el espacio 3D. Sin embargo, las matrices no son la mejor forma de representar rotaciones, por las siguientes razones:

1. **Almacenamiento**. La representación mediante matrices require nueve valores de punto flotante para almacenar una rotación. Según el *Teorema de Rotación de Euler* es suficiente con establecer la rotación frente a un único eje (el Polo de Euler).

2. **Tiempo de Cómputo**. En la composición de la matriz con un vector, se necesitan calcular nueve multiplicaciones y seis sumas en punto flotante.

3. **Interpolación**. En muchas ocasiones es necesario calcular valores intermedios entre dos puntos conocidos. En el caso de rotaciones, es necesario calcular los valores de rotación intermedios entre dos rotaciones clave (en la animación de la rotación de una cámara virtual, o en las articulaciones de un personaje animado). La interpolación empleando matrices es muy complicada.

Teorema de Euler

Según el Teorema de Rotación de Leonhard Euler (1776), cualquier composición de rotaciones sobre un sólido rígido es equivalente a una sola rotación sobre un eje, llamado *Polo de Euler*.

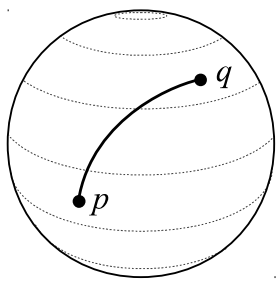


Figura 2.20: Los cuaternios unitarios pueden ser representados como puntos sobre la esfera unitaria. La interpolación esférica lineal entre dos cuaternios p y q se puede igualmente representar como un arco sobre la esfera..

4. **Bloque de Ejes**. El problema del bloqueo de ejes (denominado *Gimbal Lock*) ocurre cuando se trabaja con ángulos de Euler. Como hemos visto, cualquier rotación se puede descomponer en una secuencia de tres rotaciones básicas sobre cada uno de los ejes. Como hemos visto, el orden de aplicación de las rotaciones importa (no es conmutativo), por lo que tendremos que decidir uno. Por ejemplo, podemos aplicar primero la rotación en x, a continuación en y y finalmente en z. El orden de las rotaciones es relevante porque rotaciones posteriores tienen influencia jerárquica sobre las siguientes. El problema de *Gimbal Lock* ocurre cuando, por accidente uno de los ejes queda alineado con otro, reduciendo los grados de libertad del objeto. En el ejemplo de la Figura 2.19, tras aplicar la rotación de 90º sobre el eje y en *c)*, el eje x sufre la misma transformación (por arrastrar la rotación en y por jerarquía), de modo que queda alineado con Z. Ahora, la rotación en z equivale a la rotación en x, por lo que hemos perdido un grado de libertad.

De este modo, es interesante trabajar con cuaternios para la especificación de rotaciones. Veamos a continuación algunas de las operaciones más comunmente utilizadas con esta potente herramienta matemática.

2.4.1. Suma y Multiplicación

La suma de dos cuaternios se realiza sumando la parte escalar y vectorial por separado:

$$p + q = [(p_v + q_v), (p_s + q_s)]$$

La multiplicación de dos cuaternios representa su composición (es decir, el resultado de aplicar la rotación de uno a continuación del otro). Existen diferentes formas de multiplicar cuaternios. A continuación definiremos la más ampliamente utilizada en gráficos por computador, que es la multiplicación de *Grassman*:

$$pq = [(p_s q_v + q_s p_v + p_v \times q_v), (p_s q_s - p_v \cdot q_v)]$$

Como vemos, este tipo de multiplicación utiliza el producto vectorial y la suma de vectores en la parte vectorial de cuaternio, y el producto escalar en la parte escalar del cuaternio.

Inversión de cuaternios

La inversa de un cuaternio es mucho más rápida que el cálculo de la inversa de una matriz cuadrada. Ésta es otra razón más por la que elegir cuaternios para especificar rotaciones.

2.4.2. Inversa

En el caso de cuaternios unitarios, la inversa del cuaternio q^{-1} es igual al conjugado q^*. El conjugado se calcula simplemente negando la parte vectorial, por lo que, trabajando con cuaternios unitarios, podemos calcularlos simplemente como:

$$q^{-1} = q^* = [-q_v, q_s]$$

La multiplicación de un cuaternio por su inversa obtiene como resultado el escalar 1 (es decir, rotación 0). De este modo, $q\,q^{-1} = [0\,0\,0\,1]$.

2.4.3. Rotación empleando Cuaternios

Para aplicar una rotación empleando cuaternios, primero convertimos el punto o el vector que rotaremos a cuaternio. En ambos casos, se le aplica 0 como término de la parte escalar. Así, dado el vector v, el cuaternio correspondiente a v se calcula como $v = [v\,0] = [v_x\,v_y\,v_z\,0]$.

Una vez que tenemos expresado en formato de cuaternio el vector, para aplicar la rotación empleando el cuaternio, primero lo premultiplicamos por q y posteriormente lo post-multiplicamos por el inverso (o el conjugado, en el caso de trabajar con cuaternios unitarios). Así, el resultado de la rotación v' puede expresarse como:

$$v' = qvq^{-1}$$

Asociatividad

La concatenación de cuaternios cumple la propiedad asociativa, por lo que es posible obtener la expresión del *cuaternio neto* y aplicarlo a varios objetos de la escena.

La concatenación de cuaternios funciona exactamente igual que la concatenación de matrices. Basta con multiplicar los cuaternios entre sí. Si quisiéramos aplicar las rotacioens de los cuaternios q_1 y q_2 sobre el vector v (especificado en formato de cuaternio), bastará con realizar la operación:

$$v' = q_2 q_1\, v\, q_1^{-1} q_2^{-1}$$

Es habitual realizar conversiones entre matrices y cuaternios. OGRE incorpora llamadas a su biblioteca (ver sección 2.6) para realizar esta operación. Se recomienda el artículo de Gamasutra de Nick Bobic *Rotating Objects Using Quaternions* para profundizar en su uso.

2.5. Interpolación Lineal y Esférica

Una de las operaciones más empleadas en gráficos por computador es la interpolación. Un claro ejemplo de su uso es para calcular la posición intermedia en una animación de un objeto que se traslada desde un punto A, hasta un punto B.

La *interpolación lineal* es un método para calcular valores intermedios mediante polinomios lineales. Es una de las formas más simples de interpolación. Habitualmente se emplea el acrónimo *LERP* para referirse a este tipo de interpolación. Para obtener valores intermedios entre dos puntos A y B, habitualmente se calcula el vector $\vec{v} = B - A$ con origen en A y destino en B.

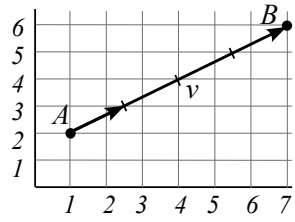


Figura 2.21: Ejemplo de interpolación lineal entre dos puntos. Se han marcado sobre el vector $\vec{v}$ los puntos correspondientes a $t = 0,25$, $t = 0.5$ y $t = 0.75$.

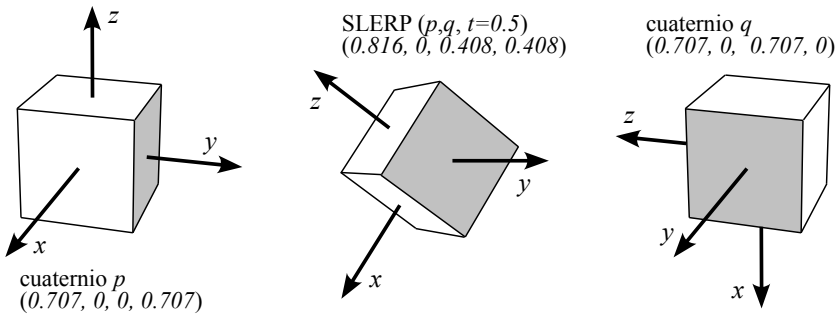


Figura 2.22: Ejemplo de aplicación de interpolación esférica *SLERP* en la rotación de dos cuaternios p y q sobre un objeto.

Para calcular valores intermedios, se utiliza una representación paramétrica, de forma que, modificando el valor de $t \in (0, 1)$, obtenemos puntos intermedios:

$$LERP(A, B, u) = A + \vec{v}t$$

Como podemos ver en el ejemplo de la Figura 2.21, para $t = 0.25$ tenemos que $\vec{v} = (6, 4)$, por lo que $\vec{v} \cdot 0.25 = (1.5, 1)$. Así, $LERP(A, B, 0.25) = (1, 2) + (1.5, 1) = (2.5, 3)$.

Como se enunció en la sección 2.4, una de las ventajas del uso de cuaternios es la relativa a la facilidad de realizar interpolación entre ellos. La *Interpolación Lineal Esférica* (también llamada *slerp* de *Spherical Linear Interpolation*) es una operación que, dados dos cuaternios y un parámetro $t \in (0, 1)$, calcula un cuaternio interpolado entre ambos, mediante la siguiente expresión:

$$SLERP(p, q, t) = \frac{sin((1-t)\theta)}{sin(\theta)}p + \frac{sin(t\theta)}{sin(\theta)}q$$

Siendo θ el ángulo que forman los dos cuaternios unitarios.

Documentación...

Recordemos que en distribuciones de GNU/Linux basadas en Debian, puedes encontrar la documentación de OGRE en local en `/usr/share/doc/ogre-doc`.

2.6. El Módulo Math en OGRE

La biblioteca de clases de OGRE proporciona una amplia gama de funciones matemáticas para trabajar con matrices, vectores, cuaternios y otras entidades matemáticas como planos, rayos, esferas, polígonos, etc...

La documentación relativa a todas estas clases se encuentra en el *Módulo Math*, que forma parte del núcleo (*Core*) de OGRE. A continuación se muestra un sencillo ejemplo que utiliza algunos operadores de la clase `Vector3` y `Quaternion`.

Listado 2.1: Ejemplo de algunas clases de Math

```
 1 cout << " Ejemplo de algunas clases de Math en OGRE " << endl;
 2 cout << "-----------------------------------------" << endl;
 3
 4 Vector3 v1(1.0, 0.0, 0.0);
 5 Vector3 v2(0.0, 2.0, 0.0);
 6 Quaternion p(0.707107, 0.0, 0.0, 0.707107);
 7 Quaternion q(Degree(90), Vector3(0.0, 1.0, 0.0));
 8
 9 cout << " Vector V1 = " << v1 << endl;
10 cout << " Vector V2 = " << v2 << endl;
11 cout << " Cuaternio P = " << p << endl;
12 cout << " Cuaternio Q = " << q << endl;
13
14 cout << "--- Algunos operadores de Vectores ------" << endl;
15 cout << " Suma: V1 + V2 = " << v1 + v2 << endl;
16 cout << " Producto por escalar: V1 * 7.0 = " << v1*7.0 << endl;
17 cout << " P. escalar: V1 * V2 = " << v1.dotProduct(v2) << endl;
18 cout << " P. vectorial: V1 x V2 =" << v1.crossProduct(v2) << endl;
19 cout << " Modulo: |V1| = " << v1.length() << endl;
20 cout << " Normalizar: V2n = " << v2.normalisedCopy() << endl;
21 cout << " Angulo (V1,V2)= " << v1.angleBetween(v2).valueDegrees()
       << endl;
22 cout << "--- Algunos operadores de Cuaternios ----" << endl;
23 cout << " Suma: P + Q = " << p + q << endl;
24 cout << " Producto: P * Q = " << p * q << endl;
25 cout << " Producto escalar: P * Q = " << p.Dot(q) << endl;
26 cout << " SLERP(p,q,0.5)= "<< Quaternion::Slerp(0.5, p, q) << endl;
```

Como puede verse en el listado anterior, OGRE incorpora multitud de operadores de alto nivel para sumar, restar, multiplicar y volcar por pantalla los objetos de la clase `Vector` y `Quaternion`.

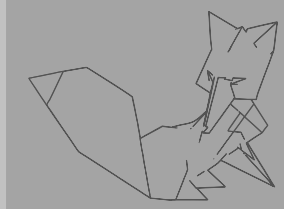
Además, OGRE facilita la creación de objetos de estos tipos proporcionando diversos constructores. Por ejemplo, en la línea [7] se emplea un constructor de cuaternio que permite especificar por separado el ángulo de rotación y el vector.

Algunas clases, como la clase *Quaternion* incorpora funciones auxiliares como el método de interpolación lineal esférico SLERP. En la línea [26] se emplea para obtener el cuaternio interpolado para $t = 0.5$. La Figura 2.22 muestra el resultado de este caso de aplicación concreto. El cuaternio definido en p se corresponde con una rotación de 90º en Z (del *Sistema de Referencia Universal - SRU*), mientras que el cuaternio definido en q equivale a una rotación de 90º en Y (del *SRU*).

El resultado de ejecutar el código del listado anterior es:

```
 Ejemplo de algunas clases de Math en OGRE
-----------------------------------------
 Vector V1 = Vector3(1, 0, 0)
 Vector V2 = Vector3(0, 2, 0)
 Cuaternio P = Quaternion(0.707107, 0, 0, 0.707107)
 Cuaternio Q = Quaternion(0.707107, 0, 0.707107, 0)
--- Algunos operadores de Vectores ------
 Suma: V1 + V2 = Vector3(1, 2, 0)
 Producto por escalar: V1 * 7.0 = Vector3(7, 0, 0)
 Producto escalar: V1 * V2 = 0
 Producto vectorial: V1 x V2 = Vector3(0, 0, 2)
 Modulo: |V1| = 1
 Normalizar: V2n = Vector3(0, 1, 0)
```

```
 Angulo entre V1 y V2 = 90
--- Algunos operadores de Cuaternios ----
 Suma: P + Q = Quaternion(1.41421, 0, 0.707107, 0.707107)
 Producto: P * Q = Quaternion(0.5, -0.5, 0.5, 0.5)
 Producto escalar: P * Q = 0.5
 SLERP(p,q,0.5) = Quaternion(0.816497, 0, 0.408248, 0.408248)
```



Capítulo 3

Grafos de Escena

Carlos González Morcillo

Como vimos en el capítulo 1, uno de los pilares clave de cualquier motor gráfico es la organización de los elementos que se representarán en la escena. Esta gestión se realiza mediante el denominado *Grafo de Escena* que debe permitir inserciones, búsquedas y métodos de ordenación eficientes. OGRE proporciona una potente aproximación a la gestión de los *Grafos de Escena*. En este capítulo trabajaremos con los aspectos fundamentales de esta estructura de datos jerárquica.

3.1. Justificación

Como señala D. Eberly [Ebe05], el motor de despliegue gráfico debe dar soporte a cuatro características fundamentales, que se apoyan en el *Grafo de Escena*:

Test de Visibilidad

La gestión de la visibilidad de los elementos de la escena es una de las tareas típicas que se encarga de realizar el motor gráfico empleando el *Grafo de Escena*.

1. **Gestión Datos Eficiente**. Es necesario eliminar toda la geometría posible antes de enviarla a la GPU. Aunque la definición del *Frustum* y la etapa de *recorte* eliminan la geometría que no es visible, ambas etapas requieren tiempo de procesamiento. Es posible utilizar el conocimiento sobre el tipo de escena a representar para optimizar el envío de estas *entidades* a la GPU. Las relaciones entre los objetos y sus atributos se modelan empleando el *Grafo de Escena*.

2. **Interfaz de Alto Nivel**. El *Grafo de Escena* puede igualmente verse como un interfaz de alto nivel que se encarga de alimentar

al motor gráfico de bajo nivel empleando alguna API específica. El diseño de este interfaz de alto nivel permite abstraernos de futuros cambios necesarios en las capas dependientes del dispositivo de visualización.

3. **Facilidad de Uso**. Esta característica está directamente relacionada con la anterior. El *Grafo de Escena* forma parte de la especificación del middleware de despliegue gráfico que facilita el uso al programador final del videojuego. De este modo, el programador se centra en la semántica asociada a la utilización del *Grafo*, dejando de lado los detalles de representación de bajo nivel.

4. **Extensibilidad**. En gran cantidad de videojuegos comerciales, los equipos de trabajo de programación frecuentemente se quejan del cambio en la especificación de requisitos iniciales por parte del equipo artístico. Esto implica añadir soporte a nuevos tipos de geometría, efectos visuales, etc. El motor debe estar preparado para añadir los nuevos tipos sin necesidad de cambios profundos en la arquitectura. El diseño correcto del *Grafo de Escena* mitiga los problemas asociados con la extensibilidad y la adaptación al cambio en los requisitos iniciales.

Las estructuras de datos de *Grafos de Escena* suelen ser grafos que representen las relaciones jerárquicas en el despliegue de objetos compuestos. Esta dependencia espacial se modela empleando *nodos* que representan agregaciones de *elementos* (2D o 3D), así como transformaciones, procesos y otras entidades representables (sonidos, vídeos, etc...).

El **Grafo de Escena** puede definirse como un grafo dirigido sin ciclos. Los arcos del grafo definen dependencias del nodo hijo respecto del padre, de modo que la aplicación de una transformación en un nodo padre hace que se aplique a todos los nodos hijo del grafo.

El *nodo raíz* habitualmente es un nodo abstracto que proporciona un punto de inicio conocido para acceder a la escena. Gracias al grafo, es posible especificar fácilmente el movimiento de escenas complejas de forma relativa a los elementos padre de la jerarquía. En la Figura 3.1, el movimento que se aplique a la *Tierra* (rotación, traslación, escalado) se aplicará a todos los objetos que dependan de ella (como por ejemplo a las *Tiritas*, o las *Llantas* de la exacavadora). A su vez, las modificaciones aplicadas sobre la *Cabina* se aplicarán a todos los nodos que herenden de la jerarquía, pero no al nodo *Llantas* o al nodo *Tierra*.

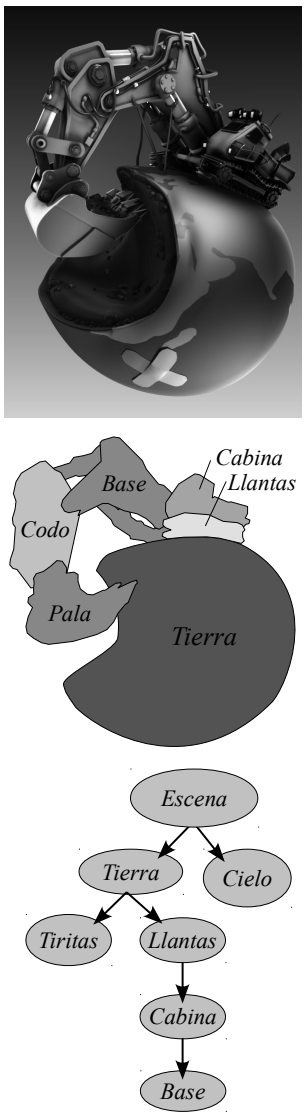


Figura 3.1: Ejemplo de grafo asociado a la escena de la excavadora. Escena diseñada por Manu Järvinen.

3.1.1. Operaciones a Nivel de Nodo

Como señala Theoharis et al. [TPP08], en términos funcionales, la principal ventaja asociada a la construcción de un *Grafo de Escena* es que cada operación se propaga de forma jerárquica al resto de entidades mediante el recorrido del grafo. Las principales operaciones que se realizan en cada nodo son la *inicialización*, *simulación*, *culling* y *dibujado*. A continuación estudiaremos qué realiza cada operación.

- **Inicialización**. Este operador establece los valores iniciales asociados a cada entidad de la jerarquía. Es habitual que existan diferentes *instancias* referenciadas de las entidades asociadas a un nodo. De este modo se separan los datos estáticos asociados a las entidades (definición de la geometría, por ejemplo) y las transformaciones aplicados sobre ellos que se incluyen a nivel de nodo.
- **Simulación**. En esta operación se determinan los parámetros y las variables asociadas a los tipos de datos existentes en el nodo. En el caso de animaciones, se actualizan los controladores asociados para que reflejen el instante de tiempo actual.

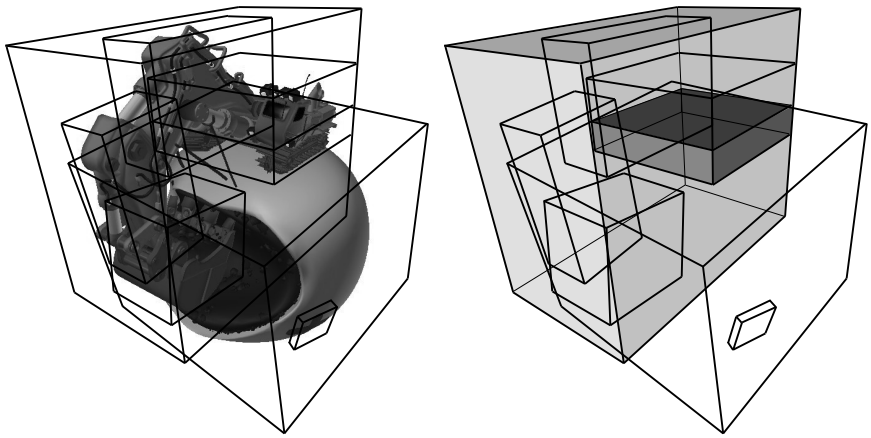


Figura 3.2: Ejemplo de estructura de datos para la gestión del *culling* jerárquico. En el ejemplo, el nodo referente a las *Llantas* (ver Figura 3.1) debe mantener la información de la caja límite *(Bounding Box)* de todos los hijos de la jerarquía, remarcada en la imagen de la derecha.

- **Culling**. En esta operación se estudia la visibilidad de los elementos contenidos en los nodos. Gracias a la especificación jerárquica del grafo, es posible *podar* ramas completas de la jerarquía. Para realizar esta operación, se utilizan estructuras de datos adicionales que mantienen información sobre las coordenadas límite (máximo y mínimo) asociadas a cada objeto. Así, consultando estos límites, si un nodo queda totalmente fuera de la pirámide de visualización (*Frustum*), implicará que todos sus nodos hijo están igualmente fuera y no será necesario su posterior dibujado (ver Figura 3.2).
- **Dibujado**. En esta operación se aplican los algoritmos de *rendering* a cada nodo de la jerarquía (comenzando por la raíz, *bajando* hasta las hojas). Si un nodo contiene órdenes que cambien el modo de dibujado, se aplicarán igualmente a todos los nodos hijo de la jerarquía.

A continuación estudiaremos el interfaz de alto nivel que proporciona OGRE para la gestión de *Grafos de Escena*. Veremos las facilidades

de gestión *orientada a objetos*, así como la abstracción relativa a los tipos de datos encapsulados en cada nodo.

Tareas del Gestor

En este capítulo nos centraremos en la parte específica de aplicación de transformaciones a los objetos. A lo largo del módulo estudiaremos otros aspectos relacionados con el gestor, como el tratamiento del *culling* o la gestión de sombras dinámicas.

3.2. El Gestor de Escenas de OGRE

Como hemos visto en la sección anterior, el *Gestor de Escenas*, apoyado en el *Grafo de Escena*, permite optimizar los datos que se representarán finalmente, podando parte de la geometría que forma la escena. El *Gestor de Escenas* en OGRE se encarga de las siguientes tareas:

- Gestión, creación y acceso eficiente a objetos móviles, luces y cámaras.
- Carga y ensamblado de la geometría (estática) del mundo 3D.
- Implementación de la operación de *Culling* para eliminación de superficies no visibles.
- Dibujado de todos los elementos que forman la escena.
- Gestión y representación de sombras dinámicas.
- Interfaz para realizar consultas a la escena. Estas consultas son del tipo: *¿Qué objetos están contenidos en una determinada región del espacio 3D?*

3.2.1. Creación de Objetos

La tarea más ampliamente utilizada del *Gestor de Escenas* es la creación de los objetos que formarán la escena: luces, cámaras, sistemas de partículas, etc. Cualquier elemento que forme parte de la escena será gestionado por el *Gestor de Escenas*, y formará parte del *Grafo de Escena*. Esta gestión está directamente relacionada con el ciclo de vida completa de los objetos, desde su creación hasta su destrucción.

Visibilidad

Si quieres que un nodo no se dibuje, simplemente ejecutas la operación de *detach* y no será *renderizado*.

Los **Nodos** del *Grafo de Escena* se crean empleando el *Gestor de Escena*. Los nodos del grafo en OGRE tienen asignado un único nodo padre. Cada nodo padre puede tener cero o más hijos. Los nodos pueden *adjuntarse (attach)* o *separarse (detach)* del grafo en tiempo de ejecución. El nodo no se destruirá hasta que se le indique explícitamente al *Gestor de Escenas*.

Destrucción de nodos

La destrucción de un *nodo* de la escena no implica la liberación de la memoria asignada a los objetos adjuntos al nodo. Es responsabilidad del programador liberar la memoria de esos objetos.

En la inicialización, el *Gestor de Escena* se encarga de crear al menos un nodo: el *Root Node*. Este nodo no tiene padre, y es el padre de toda la jerarquía. Aunque es posible aplicar transformaciones a cualquier nodo de la escena (como veremos en la sección 3.2.2), al nodo *Root* no se le suele aplicar ninguna transformación y es un buen punto en el que adjuntar toda la geometría estática de la escena. Dado el objeto `SceneManager`, una llamada al método `getRootSceneNode()` nos devuelve un puntero al **Root SceneNode**. Este nodo es, en realidad, una variable miembro de la clase `SceneManager`.

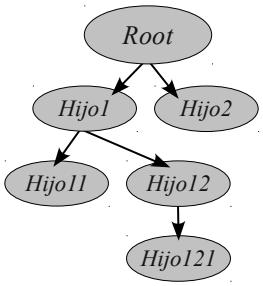


Figura 3.3: Ejemplo de grafo de escena válido en OGRE. Todos los nodos (salvo el *Root*) tienen un nodo padre. Cada nodo padre tiene cero o más hijos.

Los objetos de la escena se pueden adjuntar a cualquier nodo de la misma. En un nodo pueden existir varios objetos. Sin embargo, no es posible adjuntar la misma instancia de un objeto a varios nodos de escena al mismo tiempo. Para realizar esta operación primero hay que *separar (detach)* el objeto del nodo, y posteriormente *adjuntarlo (attach)* a otro nodo distinto.

Para **crear un nodo** en OGRE, se utiliza el método `createSceneNode`, que puede recibir como parámetro el nombre del nodo. Si no se especifica, OGRE internamente le asigna un nombre único que podrá ser accedido posteriormente.

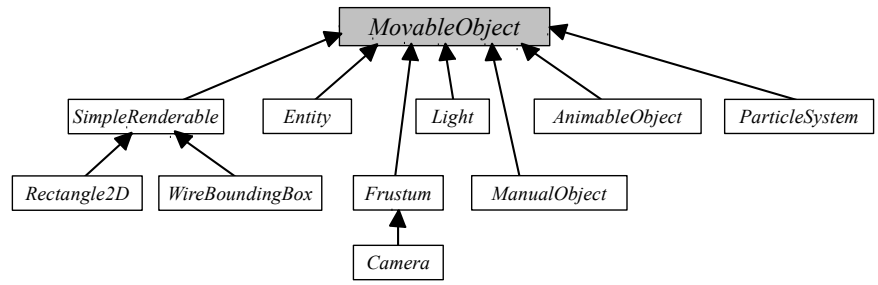


Figura 3.4: Algunos de los tipos de objetos que pueden ser añadidos a un nodo en OGRE. La clase abstracta *MovableObject* define *pequeños* objetos que serán añadidos *(attach)* al nodo de la escena.

La operación de **añadir un nodo hijo** se realiza mediante la llamada al método `addChild(Node *child)` que añade el nodo hijo previamente creado al nodo existente. La clase `SceneNode` es una subclase de la clase abstracta `Node`, definida para contener información sobre las transformaciones que se aplicarán a los nodos hijo. De este modo, la transformación neta que se aplica a cada hijo es el resultado de componer las de sus padres con las suyas propias.

Como hemos visto antes, los nodos de la escena contienen objetos. OGRE define una clase abstracta llamada `MovableObject` para definir todos los tipos de objetos que pueden ser añadidos a los nodos de la escena. En la Figura 3.4 se definen algunos de las principales subclases de esta clase abstracta. Para **añadir un objeto** a un nodo de la escena, se emplea la llamada al método `attachObject(MovableObject *obj)`.

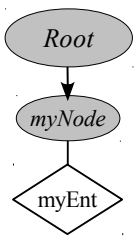


Figura 3.5: Jerarquía obtenida en el grafo de escena asociado al listado de ejemplo.

La clase `Entity` se emplea para la gestión de pequeños objetos móviles basados en mallas poligonales. Para la definición del escenario (habitualmente con gran complejidad poligonal e inmóvil) se emplea la clase `StaticGeometry`. La **creación de entidades** se realiza empleando el método `createEntity` del Gestor de Escena, indicando el nombre del modelo y el nombre que quiere asignarse a la entidad.

El código del siguiente listado (modificación del *Hola Mundo* del capítulo 1) crea un nodo hijo `myNode` de *RootSceneNode* en las líneas `3-4`, y añade la entidad `myEnt` al nodo `myChild` en la línea `6` (ver Figura 3.5). A partir de ese punto del código, cualquier transformación que se aplique sobre el nodo *myNode* se aplicarán a la entidad *myEnt*.

Listado 3.1: Creación de nodos

```
1 class SimpleExample : public ExampleApplication {
2   public : void createScene() {
3     SceneNode* node = mSceneMgr->createSceneNode("myNode");
4     mSceneMgr->getRootSceneNode()->addChild(node);
5     Entity *myEnt = mSceneMgr->createEntity("cuboejes", "cuboejes.
          mesh");
6     node->attachObject(myEnt);
7   }
8 };
```

3.2.2. Transformaciones 3D

Las transformaciones 3D se realizan a nivel de **nodo** de escena, no a nivel de objetos. En realidad en OGRE, los elementos que se mueven son los *nodos*, no los objetos individuales que han sido añadidos a los nodos.

Como vimos en la sección 2.1, OGRE utiliza el convenio de la mano derecha para especificar su sistema de coordenadas. Las rotaciones positivas se realizan en contra del sentido de giro de las agujas del reloj (ver Figura 2.11).

Las transformaciones en el *Grafo de Escena* de OGRE siguen el convenio general explicado anteriormente en el capítulo, de forma que se especifican de forma relativa al nodo padre.

Traslación

La traslación de un nodo que ya ha sido posicionado anteriormente se realiza mediante la llamada a `translate`.

La **traslación** absoluta se realiza mediante la llamada al método `setPosition`, que admite un *Vector3* o tres argumentos de tipo Real. Esta traslación se realiza de forma relativa al nodo padre de la jerarquía. Para recuperar la traslación relativa de un nodo con respecto a su nodo padre se puede emplear la llamada `getPosition`.

La **rotación** puede realizarse mediante diversas llamadas a métodos. Uno de los más sencillos es empleando las llamadas a `pitch`, `yaw` y `roll` que permiten especificar la rotación (en radianes) respecto del eje X, Y y Z respectivamente. La forma más general de aplicar una rotación es mediante la llamada a `rotate` que require un eje de rotación y un ángulo o un cuaternio como parámetro. De igual modo, la llamada a `getOrientation` nos devuelve la rotación actual del nodo.

El **escalado** análogamente se realiza mediante la llamada al método `setScale`. Mediante el método `getScale` obtenemos el factor de escala aplicado al nodo.

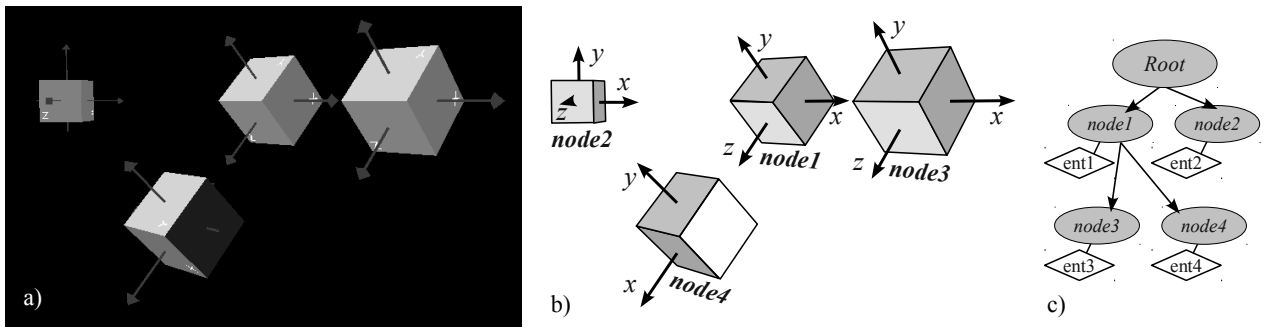


Figura 3.6: Ejemplo de transformaciones empleando el *Grafo de Escena* de OGRE. **a)** Resultado de ejecución del ejemplo. **b)** Representación de los nodos asociados a la escena. **c)** Jerarquía del grafo de escena.

A continuación veremos un código que utiliza algunas de las llamadas a métodos estudiadas anteriormente para aplicar transformaciones en 3D. El resultado de la ejecución de dicho código se muestra en la Figura 3.6.

Listado 3.2: Ejemplo de uso del Grafo de Escena

```
1  class SimpleExample : public ExampleApplication {
2   public : void createScene() {
3    SceneNode* node1 = mSceneMgr->createSceneNode("Node1");
4    Entity *ent1 = mSceneMgr->createEntity("ent1", "cuboejes.mesh");
5    node1->attachObject(ent1);
6    mSceneMgr->getRootSceneNode()->addChild(node1);
7    node1->setPosition(0,0,480);
8    node1->yaw(Degree(-45));
9    node1->pitch(Radian(Math::PI/4.0));
10
11   SceneNode* node2 = mSceneMgr->createSceneNode("Node2");
12   Entity *ent2 = mSceneMgr->createEntity("ent2", "cuboejes.mesh");
13   node2->attachObject(ent2);
14   mSceneMgr->getRootSceneNode()->addChild(node2);
15   node2->setPosition(-10,0,470);
16
17   SceneNode* node3 = mSceneMgr->createSceneNode("Node3");
18   Entity *ent3 = mSceneMgr->createEntity("ent3", "cuboejes.mesh");
19   node3->attachObject(ent3);
20   node1->addChild(node3);
21   node3->setPosition(5,0,0);
22
23   SceneNode* node4 = mSceneMgr->createSceneNode("Node4");
24   Entity *ent4 = mSceneMgr->createEntity("ent4", "cuboejes.mesh");
25   node4->attachObject(ent4);
26   node1->addChild(node4);
27   node4->setPosition(0,0,5);
28   node4->yaw(Degree(-90));
29  }
30 };
```

En el listado anterior se utilizan las funciones de utilidad para convertir a radianes (línea [7]), así como algunas constantes matemáticas (como π en la línea [8]).

Como se observa en la Figura 3.6, se han creado 4 nodos. La transformación incial aplicada al *node1* es *heredada* por los nodos 3 y 4. De este modo, basta con aplicar una traslación de 5 unidades en el eje x al nodo 3 para obtener el resultado mostrado en la Figura 3.6.b). Esta traslación es relativa al sistema de referencia *transformado* del nodo padre. No ocurre lo mismo con el nodo 2, cuya posición se especifica de nuevo desde el origen del sistema de referencia universal (del nodo *Root*).

Posición de Root

Aunque es posible aplicar transformaciones al nodo *Root*, se desaconseja su uso. El nodo *Root* debe mantenerse estático, como punto de referencia del SRU.

Los métodos de trasnformación de 3D, así como los relativos a la gestión de nodos cuentan con múltiples versiones sobrecargadas. Es conveniente estudiar la documentación de la API de OGRE para emplear la versión más interesante en cada momento.

3.2.3. Espacios de transformación

Las transformaciones estudiadas anteriormente pueden definirse relativas a diferentes espacios de transformación. Muchos de los métodos explicados en la sección anterior admiten un parámetro opcional de tipo `TransformSpace` que indica el espacio de transformación *relativo* de la operación. OGRE define tres espacios de trasnformación como un tipo enumerado `Node::TransformSpace`:

- **TS_LOCAL**. Sistema de coordenadas local del nodo.
- **TS_PARENT**. Sistema de coordenadas del nodo padre.
- **TS_WORLD**. Sistema de coordenadas universal del mundo.

El valor por defecto de este espacio de transformación depende de la operación a realizar. Por ejemplo, la rotación (ya sea mediante la llamada a *rotate* o a *pitch, yaw* o *roll*) se realiza por defecto con respecto al sistema de local, mientras que la traslación se realiza por defecto relativa al padre.

Veamos en el siguiente listado un ejemplo que ilustre estos conceptos. En el listado se definen tres nodos, en el que *node1* es el padre de *node2* y *node3*. A node 2 se le aplica una rotación respecto del eje y en la línea 15. Como hemos comentado anteriormente, por defecto (si no se le indica ningún parámetro adicional) se realiza sobre `TS_LOCAL`[1] Posteriormente se aplica una traslación en la línea 16, que por defecto se realiza relativa al sistema de coordenadas del nodo padre. Al *node3*

[1] Especificar el valor por defecto del espacio de transformación tiene el mismo efecto que no especificarlo. Por ejemplo, cambiar la línea 15 del ejemplo por `node2->yaw(Degree(-90), Node::TS_LOCAL);` no tendría ningún efecto diferente sobre el resultado final.

se el aplican exactamente la misma rotación y traslación, pero ésta última se aplica respecto del sistema de referencia local (línea 23). Como puede verse en la Figura 3.7, la entidad adjunta al *node3* se representa trasladada 5 unidades respecto de su sistema de coordenadas local.

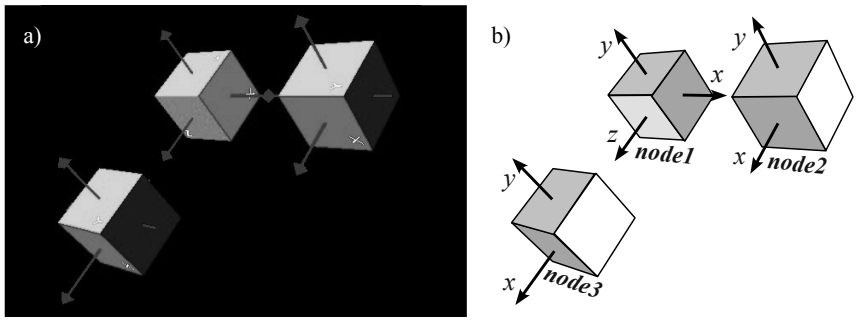


Figura 3.7: Ejemplo de trabajo con diferentes espacios de transformación. **a)** Resultado de ejecución del ejemplo. **b)** Distribución de los nodos del *Grafo de Escena*.

Listado 3.3: Uso de diferentes espacios de transformación

```
 1 class SimpleExample : public ExampleApplication {
 2  public : void createScene() {
 3   SceneNode* node1 = mSceneMgr->createSceneNode("Node1");
 4   Entity *ent1 = mSceneMgr->createEntity("ent1", "cuboejes.mesh");
 5   node1->attachObject(ent1);
 6   mSceneMgr->getRootSceneNode()->addChild(node1);
 7   node1->setPosition(0,0,480);
 8   node1->yaw(Degree(-45));    // Por defecto es Node::TS_LOCAL
 9   node1->pitch(Degree(45));   // Por defecto es Node::TS_LOCAL
10
11   SceneNode* node2 = mSceneMgr->createSceneNode("Node2");
12   Entity *ent2 = mSceneMgr->createEntity("ent2", "cuboejes.mesh");
13   node2->attachObject(ent2);
14   node1->addChild(node2);
15   node2->yaw(Degree(-90));   // Por defecto es Node::TS_LOCAL
16   node2->translate(5,0,0);   // Por defecto es Node::TS_PARENT
17
18   SceneNode* node3 = mSceneMgr->createSceneNode("Node3");
19   Entity *ent3 = mSceneMgr->createEntity("ent3", "cuboejes.mesh");
20   node3->attachObject(ent3);
21   node1->addChild(node3);
22   node3->yaw(Degree(-90));   // Por defecto es Node::TS_LOCAL
23   node3->translate(5,0,0, Node::TS_LOCAL);  // Cambiamos a LOCAL!
24  }
25 };
```

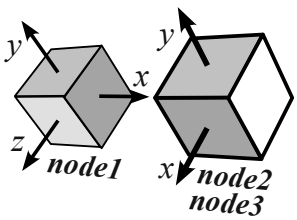
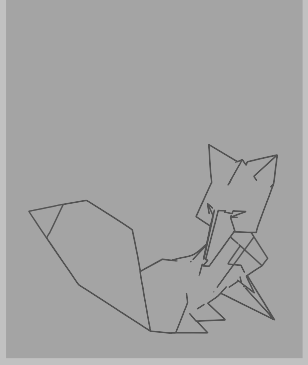


Figura 3.8: Tras aplicar la traslación de la línea 23, se aplica la rotación local al *node3*. Ahora el objeto del *node2* y *node3* aparecen alineados exactamente en la misma posición del espacio.

El orden de las operaciones resulta especialmente relevante cuando se trabaja con diferentes espacios de transformación. ¿Qué ocurriría por ejemplo si invertimos el orde de las líneas 21 y 22 del código anterior?. En ese caso, las entidades relativas al *node2* y al *node3* aparecerán desplegadas exactamente en el mismo lugar, como se muestra en la Figura 3.8.



Capítulo 4

Recursos Gráficos y Sistema de Archivos

Javier Alonso Albusac Jiménez
Carlos González Morcillo

En este capítulo se realizará un análisis de los recursos que son necesarios en los gráficos 3D, centrándose sobre todo en los formatos de especificación y requisitos de almacenamiento. Además, se analizarán diversos casos de estudio de formatos populares como MD2, MD3, MD5, Collada y se hará especial hincapié en el formato de Ogre 3D.

4.1. Formatos de Especificación

4.1.1. Introducción

En la actualidad, el desarrollo de videojuegos de última generación implica la manipulación simultánea de múltiples ficheros de diversa naturaleza, como son imágenes estáticas (BMP, JPG, TGA, PNG ...), ficheros de sonido (WAV, OGG, MP3), mallas que representan la geometría de los objetos virtuales, o secuencias de vídeo (AVI, BIK, etc). Una cuestión relevante es cómo se cargan estos ficheros y qué recursos son necesarios para su reproducción, sobre todo teniendo en cuenta que la reproducción debe realizarse en tiempo real, sin producir ningún tipo de demora que acabaría con la paciencia del usuario.

A la hora de diseñar un videojuego es muy importante tener en mente que los recursos hardware no son ilimitados. Las plataformas empleadas para su ejecución poseen diferentes prestaciones de memoria, potencia de procesamiento, tarjeta gráfica, etc. No todos los

usuarios que adquieren un juego están dispuestos a mejorar las prestaciones de su equipo para poder ejecutarlo con ciertas garantías de calidad [McS03]. Por todo ello, es fundamental elegir los formatos adecuados para cada tipo de archivo, de tal forma que se optimice en la mayor medida de lo posible los recursos disponibles.

El formato empleado para cada tipo de contenido determinará el tamaño y este factor afecta directamente a uno de los recursos más importantes en la ejecución de un videojuego: la memoria [McS03]. Independientemente del tamaño que pueda tener la memoria de la plataforma donde se ejecuta el videojuego, ésta suele estar completa durante la ejecución. Durante la ejecución se producen diversos cambios de contexto, en los que se elimina el contenido actual de la memoria para incluir nuevos datos. Por ejemplo, cuando se maneja un personaje en una determinada escena y éste entra en una nueva habitación o nivel, los datos de las escena anterior son eliminados temporalmente de la memoria para cargar los nuevos datos correspondientes al nuevo lugar en el que se encuentra el personaje protagonista. Es entonces cuando entran en juego los mecanismos que realizan el intercambio de datos y gestionan la memoria. Lógicamente, cuanto menor sea el espacio que ocupan estos datos más sencillo y óptimo será el intercambio en memoria. A la hora de elegir formato para cada tipo de fichero siempre hay que tratar de encontrar un equilibrio entre calidad y tamaño.

En la mayoría de videojuegos, los bits seleccionados se empaquetan y comprimen para ser ejecutados en el momento actual; a estos archivos se les conoce como ficheros de recursos, los cuales contienen una amplia variedad de datos multimedia (imágenes, sonidos, mallas, mapas de nivel, vídeos, etc).

Normalmente estos archivos están asociados a un nivel del juego. En cada uno de estos niveles se definen una serie de entornos, objetos, personajes, objetivos, eventos, etc. Al cambiar de nivel, suele aparecer en la pantalla un mensaje de carga y el usuario debe esperar; el sistema lo que está haciendo en realidad es cargar el contenido de estos ficheros que empaquetan diversos recursos.

Cada uno de los archivos empaquetados se debe convertir en un formato adecuado que ocupe el menor número de recursos posible. Estas conversiones dependen en la mayoría de los casos de la plataforma *hardware* en la que se ejecuta el juego. Por ejemplo, las plataformas PS3 y Xbox360 presentan formatos distintos para el sonido y las texturas de los objetos 3D.

En la siguiente sección se verá con mayor grado de detalle los recursos gráficos 3D que son necesarios, centrándonos en los formatos y los requisitos de almacenamiento.

4.1.2. Recursos de gráficos 3D: formatos y requerimientos de almacenamiento

Los juegos actuales con al menos una complejidad media-alta suelen ocupar varios GigaBytes de memoria. La mayoría de estos juegos constan de un conjunto de ficheros cerrados con formato privado que

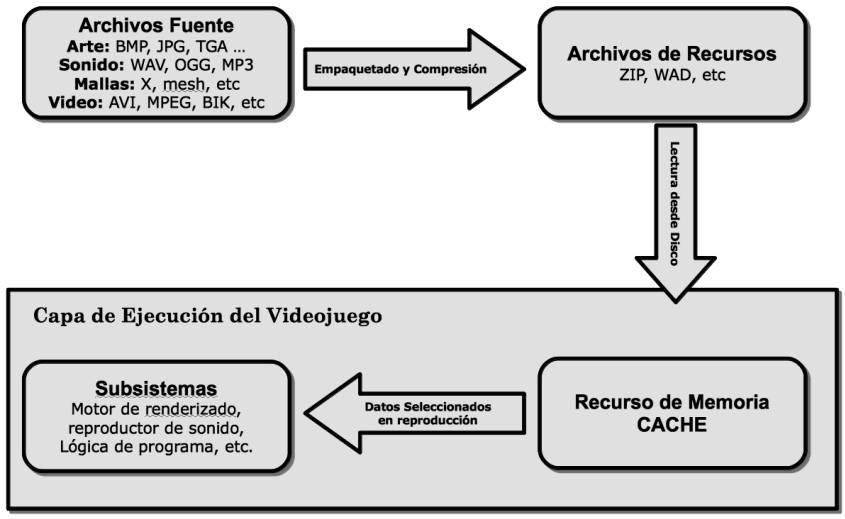


Figura 4.1: Flujo de datos desde los archivos de recursos hasta los subsistemas que se encargan de la reproducción de los contenidos [McS03].

encapsulan múltiples contenidos, los cuales están distribuidos en varios DVDs (4.7 GB por DVD) o en un simple Blue-Ray (25GB) como es en el caso de los juegos vendidos para la plataforma de Sony - PS3. Si tenemos algún juego instalado en un PC, tenemos acceso a los directorios de instalación y podemos hacernos una idea del gran tamaño que ocupan una vez que los juegos han sido instalados.

Lo que vamos a intentar en las siguientes secciones es hacernos una idea de cómo estos datos se almacenan, qué formatos utilizan y cómo se pueden comprimir los datos para obtener el producto final. Por tanto, lo primero que debemos distinguir son los tipos de ficheros de datos que normalmente se emplean. La siguiente clasificación, ofrece una visión general [McS03]:

- **Objetos y mallas 3D para el modelado de entornos virtuales:** para el almacenamiento de este tipo de datos, normalmente son necesarias unas pocas decenas de megabytes para llevar a cabo dicha tarea. En este tipo de ficheros se almacena toda la geometría asociada al videojuego.
- **Mallas 3D y datos de animación:** estos datos en realidad no ocupan demasiado espacio pero suele suponer un número elevado de ficheros, la suma de todos ellos puede ocupar varias decenas de MB también.
- **Mapa/ Datos de nivel:** en este tipo de archivos se almacenan

disparadores de eventos (eventos que se pueden producir en un determinado escenario y asociación de acciones a realizar una vez que se producen), Tipos de objetos del entorno, *scripts*, etc. No ocupan demasiado espacio al igual que el caso anterior, y suele ser bastante sencillo compactarlos o comprimirlos.

- **Sprites (personajes) y texturas asociadas a los materiales:** suele haber bastante información asociada a estos tipos de datos. En un juego medianamente complejo, los ficheros de este tipo comienzan enseguida a ocupar bastante espacio, hablamos de cientos de megas.
- **Sonido, música y diálogos:** suelen ser los datos que ocupan mas espacio de todo el juego, sobre todo cuando los juegos relatan una profunda y larga historia.
- **Vídeo y escenas pre-grabadas:** Cuando se usa este tipo de recursos suele ocupar la mayoría de espacio, por eso se usan con moderación. Suelen ser la combinación de personajes animados con archivos de sonido.

En las siguientes secciones se describirá con mayor detalle cada uno de los puntos anteriores.

Objetos y Mallas 3D

Al contrario de lo que normalmente se puede pensar, la geometría de los elementos tridimensionales que se emplean en un videojuego no ocupan demasiado espacio en comparación con otro tipo de contenidos [McS03]. Como se comentó en las secciones anteriores, los principales consumidores de espacio son los ficheros de audio y vídeo.

Una malla 3D, independientemente de que corresponda con un personaje, objeto o entorno, es una colección de puntos situados en el espacio, con una serie de datos asociados que describen cómo estos puntos están organizados y forman un conjunto de polígonos y cómo éstos deben ser renderizados.

Por otro lado, a los puntos situados en el espacio se les llama vértices y se representan mediante tres puntos pertenecientes a las tres coordenadas espaciales (X,Y,Z), tomando como referencia el punto de origen situado en (0,0,0). Cualquier elemento virtual se modela mediante triángulos que forman la malla, y cada triángulo se define por medio de tres o mas índices en una lista de puntos. Aquí se puede mostrar un ejemplo de una malla que representa un cubo.

Para representar los triángulos del cubo siguiente, necesitaríamos tres vértices por triángulo y tres coordenadas por cada uno de esos vértices. Existen diferentes formas de ahorrar espacio y optimizar la representación de triángulos. Si se tiene en cuenta que varios triángulos tienen vértices en común, no es necesario representar esos vértices más de una vez. El método consiste en representar únicamente los tres índices del primer triángulo y, para el resto de triángulos, únicamente se añade el vértice adicional. Esta técnica sería similar a dibujar el

cubo con un lápiz sin separar en ningún momento la punta del lápiz del papel.

Listado 4.1: Representación de los vértices de un cubo con respecto al origen de coordenadas

```
Vec3 TestObject::g_SquashedCubeVerts[] =
{
    Vec3( 0.5,0.5,-0.25),      // Vertex 0.
    Vec3(-0.5,0.5,-0.25),      // Vertex 1.
    Vec3(-0.5,0.5,0.5),        // And so on.
    Vec3(0.75,0.5,0.5),
    Vec3(0.75,-0.5,-0.5),
    Vec3(-0.5,-0.5,-0.5),
    Vec3(-0.5,-0.3,0.5),
    Vec3(0.5,-0.3,0.5)
};
```

Listado 4.2: Representación de los triángulos de un cubo mediante asociación de índices

```
WORD TestObject::g_TestObjectIndices[][3] =
{
    { 0,1,2 },   { 0,2,3 },   { 0,4,5 },
    { 0,5,1 },   { 1,5,6 },   { 1,6,2 },
    { 2,6,7 },   { 2,7,3 },   { 3,7,4 },
    { 3,4,0 },   { 4,7,6 },   { 4,6,5 }
};
```

Con un simple cubo es complicado apreciar los beneficios que se pueden obtener con esta técnica, pero si nos paramos a pensar que un modelo 3D puede tener miles y miles de triángulos, la optimización es clara. De esta forma es posible almacenar *n* triángulos con *n+2* índices, en lugar de *n*3* vértices como sucede en el primer caso.

Además de la información relativa a la geometría del objeto, la mayoría de formatos soporta la asociación de información adicional sobre el material y textura de cada uno de los polígonos que forman el objeto. El motor de *rendering* asumirá, a menos que se indique lo contrario, que cada grupo de triángulos tiene asociado el mismo material y las mismas texturas. El material define el color de un objeto y como se refleja la luz en él. El tamaño destinado al almacenamiento de la información relativa al material puede variar dependiendo del motor de *rendering* empleado. Si al objeto no le afecta la luz directamente y tiene un color sólido, tan sólo serán necesarios unos pocos de bytes extra. Pero, por el contrario, si al objeto le afecta directamente la luz y éste tiene una textura asociada, podría suponer casi 100 bytes más por cada vértice.

De todo esto debemos aprender que la geometría de un objeto puede ocupar mucho menos espacio si se elige un formato adecuado para representar los triángulos que forman el objeto. De cualquier forma, los requisitos de memoria se incrementan notablemente cuando se emplean materiales complejos y se asocian texturas al objeto.

Datos de Animación

Una animación es en realidad la variación de la posición y la orientación de los vértices que forman un objeto a lo largo del tiempo. Como se comento anteriormente, una forma de representar una posición o vértice en el espacio es mediante tres valores reales asociados a las tres coordenadas espaciales X, Y y Z. Estos números se representan siguiendo el estándar IEEE-754 para la representación de números reales en coma flotante mediante 32 bits (4 bytes). Por tanto, para representar una posición 3D será necesario emplear 12 bytes.

Además de la posición es necesario guardar información relativa a la orientación, y el tamaño de las estructuras de datos empleadas para ello suele variar entre 12 y 16 bytes, dependiendo del motor de *rendering*. Existen diferentes formas de representar la orientación, el método elegido influirá directamente en la cantidad de bytes necesarios. Dos de los métodos más comunes en el desarrollo de videojuegos son los ángulos de Euler y el uso de cuaterniones o también llamados cuaternios.

Para hacernos una idea del tamaño que sería necesario en una sencilla animación, supongamos que la frecuencia de reproducción de frames por segundo es de 25. Por otro lado, si tenemos en cuenta que son necesarios 12 bytes por vértice más otros 12 (como mínimo) para almacenar la orientación de cada vértice, necesitaríamos por cada vértice 12 + 12 = 24 bytes por cada frame. Si en un segundo se reproducen 25 frames, 25 x 24 = 600 bytes por cada vértice y cada segundo. Ahora supongamos que un objeto consta de 40 partes movibles (normalmente las partes movibles de un personaje las determina el esqueleto y los huesos que lo forman). Si cada parte movible necesita 600 bytes y existen 40 de ellas, se necesitaría por cada segundo un total de 24.000 bytes.

Naturalmente 24.000 bytes puede ser un tamaño excesivo para un solo segundo y existen diferentes formas de optimizar el almacenamiento de los datos de animación, sobre todo teniendo en cuenta que no todas las partes del objeto se mueven en cada momento y, por tanto, no sería necesario almacenarlas de nuevo. Tampoco es necesario almacenar la posición de cada parte movible en cada uno de los 25 frames que se reproducen en un segundo. Una solución elegante es establecer frames claves y calcular las posiciones intermedias de un vértice desde un frame clave al siguiente mediante interpolación lineal. Es decir, no es necesario almacenar todas las posiciones (únicamente la de los frames claves) ya que el resto pueden ser calculadas en tiempo de ejecución.

Otra posible mejora se podría obtener empleando un menor número de bytes para representar la posición de un vértice. En muchas ocasiones, el desplazamiento o el cambio de orientación no son exageradamente elevados y no es necesario emplear 12 bytes. Si los valores no son excesivamente elevados se pueden emplear, por ejemplo, números enteros de 2 bytes. Estas técnicas de compresión pueden reducir considerablemente el tamaño en memoria necesario para almacenar los datos de animación.

Mapas/Datos de Nivel

Cada uno de los niveles que forman parte de un juego tiene asociado diferentes elementos como objetos estáticos 3D, sonido de ambientación, diálogos, entornos, etc. Normalmente, todos estos contenidos son empaquetados en un fichero binario que suele ser de formato propietario. Una vez que el juego es instalado en nuestro equipo es complicado acceder a estos contenidos de manera individual. En cambio, durante el proceso de desarrollo estos datos se suelen almacenar en otros formatos que sí son accesibles por los miembros del equipo como, por ejemplo, en formato XML. El formato XML es un formato accesible, interpretable y permite a personas que trabajan con diferentes herramientas y en diferentes ámbitos, disponer de un medio para intercambiar información de forma sencilla.

Texturas

Hasta el momento, los datos descritos para el almacenamiento de la geometría de los objetos, animación y datos de nivel no suponen un porcentaje alto de la capacidad de almacenamiento requerida. Las texturas (en tercer lugar), los ficheros de audio y vídeo son los que implican un mayor coste en términos de memoria.

La textura es en realidad una imagen estática que se utiliza como "piel"para cubrir la superficie de un objeto virtual. Existen multitud de formatos que permiten obtener imágenes de mayor o menor calidad y, en función de esta calidad, de un mayor o menor tamaño. Para obtener la máxima calidad, los diseñadores gráficos prefieren formatos no comprimidos de 32 bits como TIF o TGA. Por contra, el tamaño de estas imágenes podría ser excesivo e influir en el tiempo de ejecución de un videojuego. Por ejemplo, una imagen RAW (imagen sin modificaciones, sin comprimir) de 32 bits con una resolución de 1024 x 768 píxeles podría alcanzar el tamaño de 3MB. Por tanto, una vez más es necesario encontrar un equilibrio entre calidad y tamaño. Cuando se diseña un videojuego, una de las principales dificultades que se plantean es la elección de un formato adecuado para cada uno de los recursos, tal que se satisfagan las expectativas de calidad y eficiencia en tiempo de ejecución.

Uno de los parámetros característicos de cualquier formato de imagen es la profundidad del color [McS03]. La profundidad del color determina la cantidad de bits empleados para representar el color de un píxel en una imagen digital. Si con n bits se pueden representar 2^n valores, el uso de n bits por píxel ofrecerá la posibilidad de representar 2^n colores distintos, es decir, cuando mayor sea el número n de bits empleados, mayor será la paleta de colores disponibles.

- **32-bits (8888 RGBA).** Las imágenes se representan mediante cuatro canales, R(red), G(green), B(blue), A(alpha), y por cada uno de estos canales se emplean 8 bits. Es la forma menos compacta de representar un mapa de bits y la que proporciona un mayor abanico de colores. Las imágenes representadas de esta forma

poseen una calidad alta, pero en muchas ocasiones es innecesaria y otros formatos podrían ser más apropiados considerando que hay que ahorrar el mayor número de recursos posibles.

- **24-bits (888 RGB)**. Este formato es similar al anterior pero sin canal alpha, de esta forma se ahorran 8 bits y los 24 restantes se dividen en partes iguales para los canales R(red), G(green) y B(blue). Este formato se suele emplear en imágenes de fondo que contienen una gran cantidad de colores que no podrían ser representados con 16 u 8 bits.
- **24-bits (565 RGB, 8A).** Este formato busca un equilibrio entre los dos anteriores. Permite almacenar imágenes con una profundidad de color aceptable y un rango amplio de colores y, además, proporciona un canal alfa que permite incluir porciones de imágenes traslúcidas. El canal para el color verde tiene un bit extra debido a que el ojo humano es más sensible a los cambios con este color.
- **16-bits (565 RGB).** Se trata de un formato compacto que permite almacenar imágenes con diferentes variedades de colores sin canal alfa. El canal para el color verde también emplea un bit extra al igual que en el formato anterior.
- **16-bits (555 RGB, 1 A)**. Similar al formato anterior, excepto el bit extra del canal verde que, ahora, es destinado al canal alfa.
- **8-bits indexado**. Este formato se suele emplear para representar iconos o imágenes que no necesitan alta calidad, debido a que la paleta o el rango de colores empleado no es demasiado amplio. Sin embargo, son imágenes muy compactas que ahorran mucho espacio en memoria y se transmiten o procesan rápidamente. En este caso, al emplearse 8 bits, la paleta sería de 256 colores y se representa de forma matricial, donde cada color tiene asociado un índice. Precisamente los índices son los elementos que permiten asociar el color de cada píxel en la imagen a los colores representados en la paleta de colores.

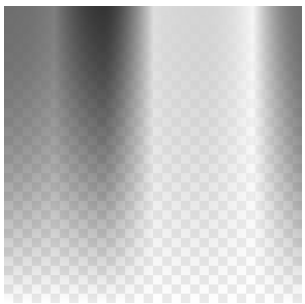


Figura 4.2: Ejemplo de una imagen RGBA con porciones transparentes (canal alpha).

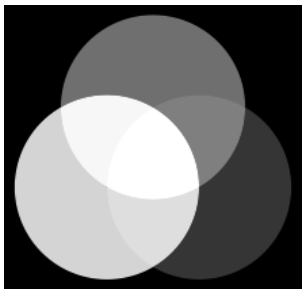


Figura 4.3: Modelo aditivo de colores rojo, verde y azul.

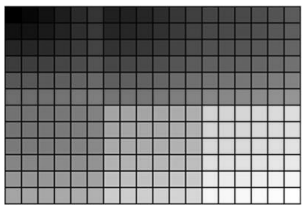


Figura 4.4: Paleta de 256 colores con 8 bits

4.1.3. Casos de Estudio

Formato MD2/MD3

El formato MD2 es uno de los más populares por su simplicidad, sencillez de uso y versatilidad para la creación de modelos (personajes, entornos, armas, efectos, etc). Fue creado por la compañía id Software y empleado por primera vez en el videojuego Quake II (ver Figura 4.5).

El formato MD2 representa la geometría de los objetos, texturas e información sobre la animación de los personajes en un orden muy concreto, tal como muestra la Figura 4.6 [Pip02]. El primer elemento en la mayoría de formatos 3D es la cabecera. La cabecera se sitúa al comienzo del fichero y es realmente útil ya que contiene información relevante sobre el propio fichero sin la necesidad de tener que buscarla a través de la gran cantidad de datos que vienen a continuación.

Figura 4.5: Captura del vídeojuego Quake II, desarrollado por la empresa id Software, donde se empleó por primera vez el formato MD2

Cabecera

La cabecera del formato MD2 contiene los siguientes campos:

Listado 4.3: Cabecera del formato MD2

```
1  struct SMD2Header {
2      int m_iMagicNum;
3      int m_iVersion;
4      int m_iSkinWidthPx;
5      int m_iSkinHeightPx;
6      int m_iFrameSize;
7      int m_iNumSkins;
8      int m_iNumVertices;
9      int m_iNumTexCoords;
10     int m_iNumTriangles;
11     int m_iNumGLCommands;
12     int m_iOffsetSkins;
13     int m_iOffsetTexCoords;
14     int m_iOffsetTriangles;
15     int m_iOffsetFrames;
16     int m_iOffsetGlCommands;
17     int m_iFileSize;
18     int m_iNumFrames;
19
20 };
```

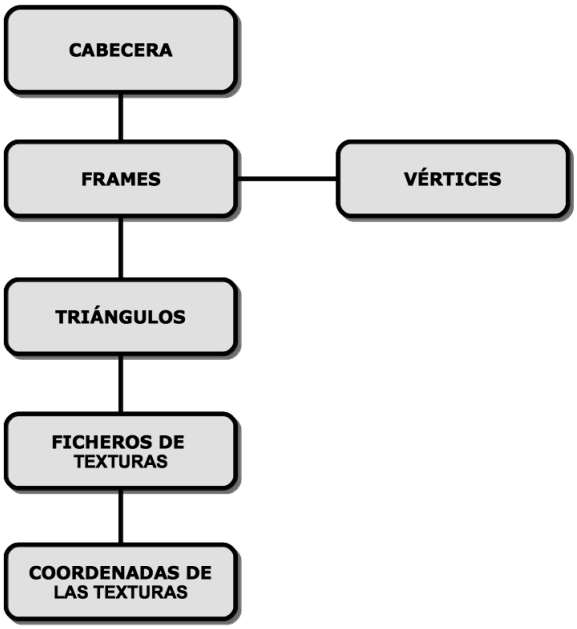


Figura 4.6: Jerarquía de capas en un fichero con formato MD2

El tamaño total es de 68 bytes, debido a que cada uno de los enteros se representa haciendo uso de 4 bytes. A continuación, en el siguiente listado se describirá brevemente el significado de cada uno de estos campos [Pip02].

- En primer lugar podemos apreciar un campo que se refiere al "número mágico". Este número sirve para identificar el tipo de archivo y comprobar que en efecto se trata de un fichero con formato MD2.
- Versión del fichero. De esta forma se lleva a cabo un control de versiones y se evita el uso de versiones obsoletas del fichero.
- La siguiente variable hace referencia a la textura que se utiliza para cubrir la superficie del modelo. En este formato, cada modelo MD2 puede utilizar un piel o textura en un instante concreto de tiempo, aunque se podrían cargar varias texturas e ir alternándolas a lo largo del tiempo.
- *m_iSkinWidthPx* y *m_iSkinHeightPx* representan la anchura y altura de la textura en píxeles.
- *m_iFrameSize* es un entero que representa el tamaño en bytes de cada frame clave.

Las seis variables siguientes se refieren a cantidades [Pip02]:

- *m_iNumSkins* es el número de texturas definidas en el fichero. Como se comentó anteriormente, el formato no soporta la aplicación simultánea de múltiples texturas, pero sí cargar varias texturas y ser aplicadas en diferentes instantes.
- *m_iNumVertices* es un entero que representa el número de vértices por frame.
- *m_iNumTexCoords* representa el número de coordenadas de la textura. El número no tiene que coincidir con el número de vértices y se emplea el mismo número de coordenadas para todos los frames.
- *m_iNumTriangles* es el número de triángulos que compone el modelo. En el formato MD2, cualquier objeto está formado mediante la composición de triángulos, no existe otro tipo de primitiva como, por ejemplo, cuadriláteros.
- *m_iNumGLCommands* especifica el número de comandos especiales para optimizar el renderizado de la malla del objeto. Los comandos GL no tienen por qué cargar el modelo, pero sí proporcionan una forma alternativa de renderizarlo.
- *m_iNumFrames* representa el número de frames en el fichero MD2 file. Cada uno de los frames posee información completa sobre las posiciones de los vértices para el proceso de animación.

Las cinco variables siguientes se emplean para definir el *offset* o dirección relativa (el número de posiciones de memoria que se suman a una dirección base para obtener la dirección absoluta) en bytes de cada una de las partes del fichero. Esta información es fundamental para facilitar los saltos en las distintas partes o secciones del fichero durante el proceso de carga. Por último, la variable *m_iFileSize* representa el tamaño en bytes desde el inicio de la cabecera hasta el final del fichero.

Frames y Vértices

A continuación se muestra la estructura que representa la información que se maneja por frame en el formato MD2 [Pip02]:

Listado 4.4: Información empleada en cada frame en el formato MD2

```
1  struct SMD2Frame {
2      float m_fScale[3];
3      float m_fTrans[3];
4      char m_caName[16];
5      SMD2Vert * m_pVertss
6
7      SMD2Frame()
8      {
9          m_pVerts = 0;
10     }
11     ~SMD2Frame()
12     {
13         if(m_pVerts) delete [] m_pVerts;
14     }
15 };
```

Como se puede apreciar en la estructura anterior, cada frame comienza con seis números representados en punto flotante que representan la escala y traslación de los vértices en los ejes X, Y y Z. Posteriormente, se define una cadena de 16 caracteres que determinan el nombre del frame, que puede resultar de gran utilidad si se quiere hacer referencia a éste en algún momento. Por último se indica la posición de todos los vértices en el frame (necesario para animar el modelo 3D). En cada uno de los frames habrá tantos vértices como se indique en la variable de la cabecera *m_iNumVerts*. En la última parte de la estructura se puede diferenciar un constructor y destructor, que son necesarios para reservar y liberar memoria una vez que se ha reproducido el frame, ya que el número máximo de vértices que permite almacenar el formato MD2 por frame es 2048.

Si se renderizaran únicamente los vértices de un objeto, en pantalla tan sólo se verían un conjunto de puntos distribuidos en el espacio. El siguiente paso consistirá en unir estos puntos para definir los triángulos y dar forma al modelo.

Triangularización

En el formato MD2 los triángulos se representan siguiendo la técnica descrita en la Sección 4.1.2. Cada triángulo tiene tres vértices y varios triángulos tienen índices en común [Pip02]. No es necesario representar un vértice más de una vez si se representa la relación que existe entre ellos mediante el uso de índices en una matriz.

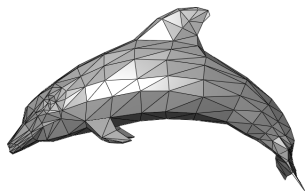


Figura 4.7: Modelado de un objeto tridimensional mediante el uso de triángulos.

Además, cada triángulo debe tener asociado una textura, o sería más correcto decir, una parte o fragmento de una imagen que representa una textura. Por tanto, es necesario indicar de algún modo las coordenadas de la textura que corresponden con cada triángulo. Para asociar las coordenadas de una textura a cada triángulo se emplean el mismo número de índices, es decir, cada índice de un vértice en el triángulo tiene asociado un índice en el array de coordenadas de una textura. De esta forma se puede texturizar cualquier objeto fácilmente.

A continuación se muestra la estructura de datos que se podía emplear para representar los triángulos mediante asociación de índices en un array y la asociación de coordenadas de la textura para cada vértice:

Listado 4.5: Estructura de datos para representar los triángulos y las coordenadas de la textura

```
1 struct SMD2Tri {
2     unsigned short m_sVertIndices[3];
3     unsigned short m_sTexIndices[3];
4 };
```

Inclusión de Texturas

En el formato MD2 existen dos formas de asociar texturas a los objetos. La primera de ellas consiste en incluir el nombre de las texturas embebidos en el fichero MD2. El número de ficheros de texturas y la localización, se encuentran en las variables de la cabecera *m_iNumSkins* y *m_iOffsetSkins*. Cada nombre de textura ocupa 64 ca-

Figura 4.8: Ejemplo de textura en la que se establece una correspondencia entre los vértices del objeto y coordenadas de la imagen.

racteres alfanuméricos como máximo. A continuación se muestra la estructura de datos empleada para definir una textura [Pip02]:

Listado 4.6: Estructura de datos para definir una textura embebida en un fichero MD2

```
1 struct SMD2Skin
2 {
3     char m_caSkin[64];
4     CImage m_Image;
5 };
```

Tal como se puede apreciar en la estructura anterior, existe una instancia de la clase *CImage*. Esta clase posee varias funciones para cargar y asociar varias clases de texturas. Existe una textura diferente por cada nombre de fichero que aparece en la sección de texturas dentro del fichero MD2.

Una segunda alternativa es utilizar la clase *CImage* y el método *SetSkin* para cargar una textura y asociarla a un objeto [Pip02]. Si existe una correspondencia por coordenadas entre la textura y el objeto, será necesario cargar las coordenadas antes de asociar la textura al objeto. El número de coordenadas de la textura se puede encontrar en la variable de la cabecera *m_iNumTexCoords*. La estructura de datos que se utiliza para definir las coordenadas de una textura es la siguiente:

Listado 4.7: Estructura de datos para representar las coordenadas de una textura

```
1 struct SMD2TexCoord {
2     float m_fTex[2];
3 };
```

Cada coordenada de una textura consiste en un par de números de tipo *float* de 2 bytes. La primera coordenada de la textura comienza en el cero y finaliza en el valor equivalente a la anchura de la imagen; la segunda desde el cero hasta el valor de la altura. En el resto de coordenadas se establecen las correspondencias entre zonas de la imagen de la textura y las superficies del objeto. Finalmente, una vez que se han definido las coordenadas ya se puede asociar la textura al objeto, mediante la función *Bind* de la clase *CImage*.

Figura 4.9: Captura del videojuego Quake III.

Principales diferencias entre el formato MD2 y MD3

El formato MD3 fue el formato que se empleó en el desarrollo del videojuego Quake III y sus derivados (Q3 mods, Return to Castle Wolfenstein, Jedi Knights 2, etc.). MD3 se basa en su predecesor pero aporta mejoras notables en dos aspectos claves:

- **Animación de personajes.** La frecuencia de reproducción de frames en el formato MD2 está limitada a 10 fps. En el caso del formato MD3 la frecuencia es variable, permitiendo animaciones de vértices más complejas.

- **Modelado de objetos 3D**. Otra diferencia significativa entre los formatos MD2 y MD3 es que en este último los objetos se dividen en tres bloques diferenciados, normalmente, cabeza, torso y piernas. Cada uno de estos bloques se tratan de manera independiente y esto implica que cada parte tenga su propio conjunto de texturas y sean renderizados y animados por separado.

Formato MD5

El formato MD5 se empleó en el desarrollo de los videojuegos Doom III y Quake IV. El formato presenta mejoras significativas en la animación de personajes. En el caso de los formatos MD2 y MD3 los movimientos de cada personaje se realizaban mediante animación de vértices, es decir, se almacenaba la posición de los vértices de un personaje animado en cada frame clave. Tratar de forma individualizada cada vértice es realmente complejo y animar un personaje siguiendo esta metodología no es una tarea sencilla.

En el formato MD5 es posible definir un esqueleto formado por un conjunto de huesos y asociarlo a un personaje. Cada uno de los huesos tiene a su vez asociado un conjunto de vértices. La animación en el formato MD5 se basa en la animación de los huesos que forman el esqueleto; el grupo de vértices asociado a un hueso variará su posición en base al movimiento de éste. Una de las grandes ventajas de la animación mediante movimiento de huesos de un esqueleto es que ésta se puede almacenar y reutilizar para personajes que tengan un esqueleto similar.

COLLADA

El formato COLLADA (COLLAborative Design Activity) [Gro12] surge ante la necesidad de proponer un formato estándar de código abierto que sirva como medio de intercambio en la distribución de contenidos. La mayoría de empresas utilizan su propio formato de código cerrado y en forma binaria, lo que dificulta el acceso a los datos y la reutilización de contenidos por medio de otras herramientas que permiten su edición. Con la elaboración de COLLADA se pretenden alcanzar una serie de objetivos básicos [Noc12]:

- COLLADA no es un formato para motores de videojuegos. En realidad, COLLADA beneficia directamente a los usuarios de herramientas de creación y distribución de contenidos. Es decir, COLLADA es un formato que se utiliza en el proceso de producción como mecanismo de intercambio de información entre los miembros del equipo de desarrollo y no como mecanismo final de producción.

- El formato COLLADA debe ser independiente de cualquier plataforma o tecnología de desarrollo (sistemas operativos, lenguajes de programación, etc).

- Ser un formato de código libre, representado en XML para liberar los recursos digitales de formatos binarios propietarios.
- Proporcionar un formato estándar que pueda ser utilizado por una amplia mayoría de herramientas de edición de modelos tridimensionales.
- Intentar que sea adoptado por una amplia comunidad de usuarios de contenidos digitales.
- Proveer un mecanismo sencillo de integración, tal que toda la información posible se encuentre disponible en este formato.
- Ser la base común de todas las transferencias de datos entre aplicaciones 3D.

Cualquier fichero XML representado en el formato COLLADA está dividido en tres partes principales [Noc12]:

- **COLLADA Core Elements**. Donde se define la geometría de los objetos, información sobre la animación, cámaras, luces, etc.
- **COLLADA Physics**. En este apartado es posible asociar propiedades físicas a los objetos, con el objetivo de reproducir comportamientos lo más realistas posibles.
- **COLLADA FX.** Se establecen las propiedades de los materiales asociados a los objetos e información valiosa para el renderizado de la escena.

No es nuestro objetivo ver de forma detallada cada uno de estos tres bloques, pero sí se realizará una breve descripción de los principales elementos del núcleo ya que éstos componen los elementos básicos de una escena, los cuales son comunes en la mayoría de formatos.

COLLADA Core Elements

En este bloque se define la escena (<*scene*>) donde transcurre una historia y los objetos que participan en ella, mediante la definición de la geometría y la animación de los mismos. Además, se incluye información sobre las cámaras empleadas en la escena y la iluminación. Un documento con formato COLLADA sólo puede contener un nodo <*scene*>, por tanto, será necesario elaborar varios documentos COLLADA para definir escenas diferentes. La información de la escena se representa mediante un grafo acíclico y debe estar estructurado de la mejor manera posible para que el procesamiento sea óptimo. A continuación se muestran los nodos/etiquetas relacionados con la definición de escenas [Noc12].

Listado 4.8: Nodos empleados en COLLADA para la definición de escenas

```
<instance_node>
<instance_visual_scene>
<library_nodes>
<library_visual_scenes>
<node>
<scene>
<visual_scene>
```

Para facilitar el manejo de una escena, COLLADA ofrece la posibilidad de agrupar los elementos en librerías. Un ejemplo sencillo de definición de una escena podría ser el siguiente:

Al igual que en el caso anterior, la definición de la geometría de un objeto también puede ser estructurada y dividida en librerías, muy útil sobre todo cuando la geometría de un objeto posee una complejidad considerable. De esta forma los nodos *geometry* se pueden encapsular dentro de una librería de objetos denominada *library_geometrics*.

Listado 4.9: Ejemplo de definición de una escena en el formato COLLADA

```
<COLLADA>
<library_nodes id="comp">
  <node name="earth">
  </node>
  <node name="sky">
  </node>
</library_nodes>

<library_visual_scenes>
  <visual_scene id="world">
   <instance_library_nodes url="#comp">
  </visual_scene>
</library_visual_scenes>

<scene>
  <instance_visual_scene url="#world"/>
</scene>
</COLLADA>
```

El elemento *<geometry>* permite definir la geometría de un objeto. En el caso de COLLADA (y en la mayoría de formatos) la geometría se define mediante la definición de una malla que incluye información del tipo: cantidad de puntos, posición, información de color, coordenadas de la textura aplicada al objeto, líneas que los conectan, ángulos, superficies, etc. Los nodos que se emplean para definir la geometría de un objeto en COLLADA son los siguientes [Noc12]:

Listado 4.10: Nodos utilizados para definir la geometría de un objeto en el formato COLLADA

```
<control_vertices>
<geometry>
<instance_geometry>
<library_geometries>
<lines>
```

```
<linestrips>
<mesh>
<polygons>
<polylist>
<spline>
<triangles>
<trifans>
```

Un ejemplo de definición de una malla en el formato COLLADA podría ser el que se muestra en el siguiente listado [Noc12].

Para completar la información de la geometría de un objeto es necesario incluir datos sobre la transformación de los vértices. Algunas de las operaciones más comunes son la rotación (<*rotate*>), escalado (<*scale*>) o traslaciones (<*translate*>).

Listado 4.11: Ejemplo de definición de una malla en el formato COLLADA

```
<mesh>
  <source id="position" />
  <source id="normal" />
  <vertices id="verts">
    <input semantic="POSITION" source="#position"/>
  </vertices>
  <polygons count="1" material="Bricks">
    <input semantic="VERTEX" source="#verts" offset="0"/>
    <input semantic="NORMAL" source="#normal" offset="1"/>
    <p>0 0  2 1  3 2  1 3</p>
  </polygons>
</mesh>
```

En cuanto a la iluminación, COLLADA soporta la definición de las siguientes fuentes de luces [Noc12]:

- Luces ambientales
- Luces puntuales
- Luces direccionales
- Puntos de luz

y se utilizan los siguientes nodos o etiquetas:

Listado 4.12: Nodos empleados en COLLADA para la definición de fuentes de luz

```
<ambient>
<color>
<directional>
<instance_light>
<library_lights>
<Light>
<point>
<spot>
```

Por otro lado, COLLADA también permite la definición de cámaras. Una cámara declara una vista de la jerarquía del grafo de la escena y contiene información sobre la óptica (perspectiva u ortográfica). Los nodos que se utilizan para definir una cámara son los siguientes [Noc12]:

Listado 4.13: Nodos empleados en COLLADA para definir cámaras

```
<camera>
<imager>
<instance_camera>
<library_cameras>
<optics>
<orthographic>
<Perspective>
```

Un ejemplo de definición de una cámara podría ser el siguiente [Noc12]:

Listado 4.14: Ejemplo de definición de una cámara en el formato COLLADA

```
<camera name="eyepoint">
  <optics>
    <technique_common>
      <perspective>
        <yfov>45</yfov>
           <aspect_ratio>1.33333
            </aspect_ratio>
        <znear>1.0</znear>
        <zfar>1000.0</zfar>
      </perspective>
    </technique_common>
  </optics>
</camera>
```

En cuanto a la parte de animación no entraremos en detalle en esta sección ya que se verá en capítulos posteriores. Simplemente recalcar que COLLADA está preparado para soportar los dos tipos de animación principales: animación de vértices y animación de esqueletos asociado a objetos.

Formato para OGRE 3D

Los tres elementos esenciales en el formato de Ogre son los siguientes [Ibe12]:

- **Entity o entidad.** Una entidad es cualquier elemento que se puede dibujar en pantalla; por tanto, quedan excluidos de esta categoría las fuentes de luz y las cámaras. La posición y la orientación de una malla no se controla mediante este tipo de objeto pero sí el material y la textura asociada.
- **SceneNode o nodo de la escena**. Un nodo se utiliza para manipular las propiedades o principales características de una entidad (también sirven para controlar las propiedades de las fuentes

de luz y cámaras). Los nodos de una escena están organizados de forma jerárquica y la posición de cada uno de ellos siempre es relativa a la de los nodos padre.

- **SceneManager o controlador de la escena**. Es el nodo raíz y de él derivan el resto de nodos que se dibujan en la escena. A través de este nodo es posible acceder a cualquier otro nodo en la jerarquía.

Si analizamos cualquier fichero OgreXML podemos apreciar que existe información relativa a las mallas de los objetos. Cada malla puede estar formada a su vez por una o varias submallas, que contienen la siguiente información:

- Definición de caras (<face>).
- Definición de vértices (<vertex>) con su posición, normal, color y coordenadas UV.
- Asignación de vértices a huesos (<vertexboneassignment>).
- Enlace a un esqueleto (<skeletonlink>).

A su vez cada submalla contiene el nombre del material y el número de caras asociada a la misma. Para cada una de las caras se almacenan los vértices que la componen:

Listado 4.15: Creación de mallas y submallas con materiales asociados

```
<mesh>
    <submeshes>
     <submesh material="blanco_ojo" usesharedvertices="false">
         <faces count="700">
             <face v1="0" v2="1" v3="2"/>
            ..........................
        </faces>
```

Además es necesario saber la información geométrica de cada uno de los vértices, es decir, posición, normal, coordenada de textura y el número de vértices de la submalla

Listado 4.16: Geometría de un Objeto en OgreXML

```
<geometry vertexcount="3361">
 <vertexbuffer positions="true" normals="true" texture_coords="1">
  <vertex>
   <position x="-0.000000" y="0.170180" z="-0.000000"/>
   <normal x="0.000000" y="1.000000" z="0.000000"/>
   <texcoord u="0.000000" v="1.000000"/>
  </vertex>
```

Para la asignación de los huesos se indican los huesos y sus pesos. Como se puede observar, el índice utilizado para los huesos empieza por el 0.

Listado 4.17: Asignación de huesos a una malla en OgreXML

```
<boneassignments>
 <vertexboneassignment vertexindex="0" boneindex="26" weight="
     1.000000"/>
 ..........................
</boneassignments>
</submesh>
```

Si la malla o mallas que hemos exportado tienen asociado un esqueleto se mostrará con la etiqueta "skeletonlink". El campo *name* corresponde con el archivo xml que contiene información sobre el esqueleto: <*skeletonlink name=çuerpo.skeleton/*>. En este caso el archivo "name.skeleton.xml"define el esqueleto exportado, es decir, el conjunto de huesos que componen el esqueleto. El exportador asigna a cada hueso un índice empezando por el 0 junto con el nombre, la posición y rotación del mismo:

Listado 4.18: Definición de un esqueleto en OgreXML

```
<skeleton>
<bones>
 <bone id="0" name="cerrada">
  <position x="5.395440" y="6.817142" z="-0.132860"/>
  <rotation angle="0.000000">
   <axis x="1.000000" y="0.000000" z="0.000000"/>
  </rotation>
 </bone>
 ..........................
</bones>
```

La definición del esqueleto no estaría completa si no se conoce la jerarquía de los huesos, esta información se describe indicando cuál es el padre de cada uno e los huesos:

Listado 4.19: Definición de la jerarquía de huesos en un esqueleto en el formato OgreXML

```
<bonehierarchy>
     <boneparent bone="torso" parent="caderas" />
     <boneparent bone="rota_ceja_izquierda" parent="ceja_izquierda"
          />
     <boneparent bone="rota_ceja_derecha" parent="ceja_derecha" />
     <boneparent bone="pecho" parent="torso" />
     <boneparent bone="hombro.r" parent="pecho" />
    ..........................
</bonehierarchy>
```

Por último, si se han creado animaciones asociadas al esqueleto y han sido exportadas, se mostrará cada una de ellas identificándolas con el nombre definido previamente en Blender y la longitud de la acción medida en segundos. Cada animación contendrá información sobre los huesos que intervienen en el movimiento (traslación, rotación y escalado) y el instante de tiempo en el que se ha definido el *frame* clave:

Listado 4.20: Animación de los huesos de un esqueleto en OgreXML

```
<animations>
 <animation name="ascensor" length="2.160000">
  <track bone="d1root.l">
   <keyframes>
   <keyframe time="0.000000">
    <translate x="-0.000000" y="0.000000" z="0.000000"/>
    <rotate angle="0.000000">
     <axis x="-0.412549" y="0.655310" z="0.632749"/>
    </rotate>
    <scale x="1.000000" y="1.000000" z="1.000000"/>
   </keyframe>

    ..........................
   <keyframe time="2.160000">
    <translate x="0.000000" y="-0.000000" z="0.000000"/>
      <rotate angle="0.000000">
       <axis x="-0.891108" y="0.199133" z="0.407765"/>
      </rotate>
      <scale x="1.000000" y="1.000000" z="1.000000"/>
   </keyframe>
  </keyframes>
  ..........................
</track>
```

Otros Formatos

A continuación se incluye una tabla con alguno de los formatos más comunes para la representación de objetos 3D, su extensión, editores que lo utilizan y un enlace donde se puede obtener más información sobre el formato (ver Tabla 3.1) [Pip02].

4.2. Exportación y Adaptación de Contenidos

Como se comentó en secciones anteriores la mayoría de herramientas de creación de contenidos digitales posee su propio formato privado. Sin embargo, la mayoría de estas herramientas permiten la exportación a otros formatos para facilitar la distribución de contenidos. En esta sección nos centraremos en la creación de un modelo en Blender, la aplicación de texturas a dicho modelo mediante la técnica de UV Mapping y su exportación a los formatos XML y binario de Ogre. Finalmente, cargaremos el objeto exportado en una aplicación de Ogre 3D.

EXT.	EDITOR	Link
3DMF	3D Meta File, QuickDraw3D	http://www.apple.com
3DO	Jedi Knight	http://www.lucasarts.com
3DS	3D Studio Max, formato binario	http://www.discreet.com
ACT	Motor Genesis 3D	http://www.genesis3D.com
ASE	3D Studio Max: versión de 3DS basada en texto	http://www.discreet.com
ASC	3D Studio Max: mínima representación de datos representada en ASCII	http://www.discreet.com
B3D	Bryce 3D	http://www.corel.com
BDF	Okino	http://www.okino.com
BLEND	Blender	http://www.blender.com
CAR	Carrara	http://www.eovia.com/carrara
COB	Calgari TrueSpace	http://www.calgari.com
DMO	Duke Nukem 3D	http://www.3drealms.com
DXF	Autodesk Autocad	http://www.autodesk.com
HRC	Softimage 3D	http://www.softimage.com
INC	POV-RAY	http://www.povray.org
KF2	Animaciones y poses en Max Payne	http://www.maxpayne.com
KFS	Max Payne: información de la malla y los materiales	http://www.maxpayne.com
LWO	Lightwave	http://www.newtek.com
MB	Maya	http://www.aliaswavefront.com
MAX	3D Studio Max	http://www.discreet.com
MS3D	Milkshape 3D	http://www.swissquake.ch
OBJ	Alias|Wavefront	http://aliaswavefront.com
PZ3	Poser	http://www.curioslab.com
RAW	Triángulos RAW	http://
RDS	Ray Dream Studio	http://www.metacreations.com
RIB	Renderman File	http://www.renderman.com
VRLM	Lenguaje para el modelado de realidad virtual	http://www.web3d.org
X	Formato Microsoft Direct X	http://www.microsoft.com
XGL	Formato que utilizan varios programas CAD	http://www.xglspec.com

Tabla 4.1: Otros formatos para la representación de objetos en 3D

4.2.1. Instalación del exportador de Ogre en Blender

La instalación del exportador del formato Ogre a XML en distribuciones GNU/Linux basadas en Debian es realmente sencillo. Tan sólo es necesario instalar el paquete *blender-ogrexml* utilizando el gestor de paquetes *synaptic*, o bien, utilizando el comando *apt-get install* desde la línea de comandos.

En los sistemas operativos Microsoft Windows y Mac OS X se puede instalar de forma manual. En primer lugar es necesario descargar el exportador desde algún repositorio, por ejemplo [1]. En segundo lugar se debe descomprimir el archivo y almacenar el contenido en el directorio *.blender/scripts* que se encuentra en el directorio de instalación de Blender.

4.2.2. Creación de un modelo en Blender

El objetivo es modelar una caja o cubo y aplicar una textura a cada una de las seis caras. Cuando ejecutamos Blender aparece en la escena un cubo creado por defecto, por tanto, no será necesario crear ningún modelo adicional para completar este ejercicio. En el caso de que quisiéramos añadir algún cubo más sería sencillo, bastaría con pulsar la tecla **SPACE**, menú *Add >Mesh >Cube* y aparecería un nuevo cubo en la escena.

Además del cubo, Blender crea por defecto una fuente de luz y una cámara. Al igual que sucede con los objetos, es posible añadir fuentes de luz adicionales y nuevas cámaras. Para este ejercicio será suficiente con los elementos creados por defecto.

4.2.3. Aplicación de texturas mediante UV Mapping

La técnica de texturizado UV Mapping permite establecer una correspondencia entre los vértices de un objeto y las coordenadas de una textura. Mediante esta técnica no es necesario crear un fichero o imagen para representar cada una de las texturas que se aplicará a cada una de las superficies que forman un objeto. En otras palabras, en un mismo fichero se puede representar la "piel"que cubrirá diferentes regiones de una superficie tridimensional. En está sección explicaremos como aplicar una textura a cada una de las seis caras del cubo creado en la sección anterior. La textura que se aplica a cada una de las caras puede aparecer desglosada en una misma imagen tal como muestra la Figura 4.10.

Cada uno de los recuadros corresponde a la textura que se aplicará a una de las caras del cubo en el eje indicado. En la Figura 4.11 se muestra la textura real que se aplicará al modelo; tal como se puede apreciar está organizada de la misma forma que se presenta en la Figura 4.10 (cargar la imagen llamada textura512.jpg que se aporta con el material del curso). Todas las texturas se encuentran en una misma

[1] http://www.xullum.net/lefthand/downloads/temp/BlenderExport.zip

Figura 4.10: Orientación de las texturas aplicadas a un cubo

imagen cuya resolución es 512x512. Los módulos de memoria de las tarjetas gráficas están optimizados para trabajar con imágenes representadas en matrices cuadradas y con una altura o anchura potencia de dos. Por este motivo, la resolución de la imagen que contiene las texturas es de 512x512 a pesar de que haya espacios en blanco y se pueda compactar más reduciendo la altura.

Una vez conocida la textura a aplicar sobre el cubo y cómo está estructurada, se describirán los pasos necesarios para aplicar las texturas en las caras del cubo mediante la técnica UV Mapping. En primer lugar ejecutamos Blender 2.49 y dividimos la pantalla en dos partes. Para ello situamos el puntero del ratón sobre el marco superior hasta que el cursor cambie de forma a una flecha doblemente punteada, pulsamos el botón derecho y elegimos en el menú flotante la opción *Split Area.*

En el marco de la derecha pulsamos el menú desplegable situado en la esquina inferior izquierda y elegimos la opción *UV/Image Editor*. A continuación cargamos la imagen de la textura (textura512.jpg); para ello pulsamos en el menú *Image >Open* y elegimos la imagen que contiene las texturas. Después de realizar estos datos la apariencia del editor de Blender debe ser similar a la que aparece en la Figura 4.12

En el marco de la izquierda se pueden visualizar las líneas que delimitan el cubo, coloreadas de color rosa, donde el modo de vista por defecto es *Wireframe*. Para visualizar por pantalla las texturas en todo momento elegimos el modo de vista *Textured* en el menú situado en el marco inferior *Viewport Shading*. Otra posibilidad consiste en pulsar las teclas [SHIFT] + [Z].

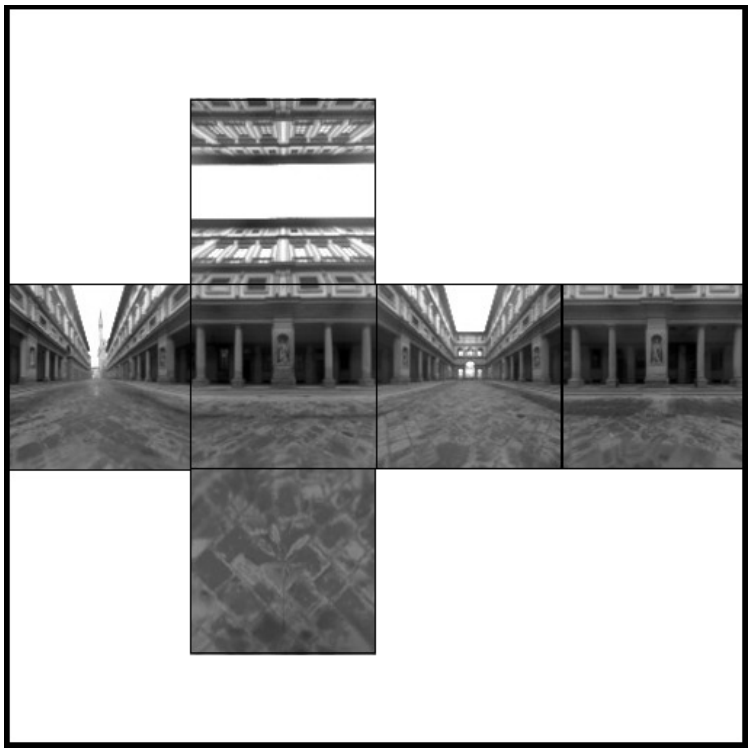


Figura 4.11: Imagen con resolución 512x512 que contiene las texturas que serán aplicadas a un cubo

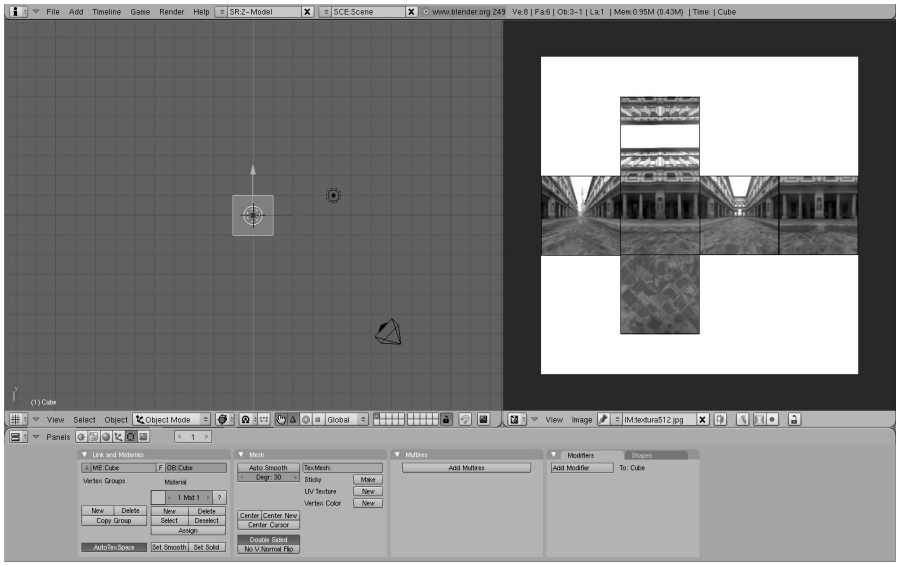


Figura 4.12: Editor de Blender con una división en dos marcos. En la parte izquierda aparece la figura geométrica a texturizar y en la parte derecha las texturas que serán aplicadas mediante la técnica de UV/Mapping

Por otro lado, el cubo situado en la escena tiene asociado un material por defecto, pero éste no tiene asignada ninguna textura. Para ello, abrimos el panel *Shading* pulsando F5, y en el panel pulsamos el botón *Texture Buttons* o bien F6. En el tipo de textura se elige *Image* y con el botón *load* de la parte derecha, situado en el cuarto panel, elegimos la imagen de la textura.

Volvemos al panel con la información relativa al material pulsando F5 y elegimos el cubo en el modo de previsualización. A continuación, nos dirigimos al quinto panel y pulsamos sobre la pestaña *Map Input*, finalmente marcamos la opción *UV*.

El siguiente paso consiste en elegir los vértices de cada una de las caras del cubo y establecer una correspondencia con coordenadas de la textura. Para ello, pulsamos la tecla TAB para visualizar los vértices y nos aseguramos que todos están seleccionados (todos deben estar coloreados de amarillo). Para seleccionar o eliminar la selección de todos los vértices basta con pulsar la tecla A. Con todos los vértices del cubo seleccionados, pulsamos la tecla U y en el menú *UV Calculation* elegimos la opción *Cube Projection*. En el editor UV de la derecha deben aparecer tres recuadros nuevos de color morado.

Pulsamos A para que ninguno de los vértices quede seleccionado y, de forma manual, seleccionamos los vértices de la cara superior (z+). Para seleccionar los vértices existen dos alternativas; la primera es seleccionar uno a uno manteniendo la tecla SHIFT pulsada y presionar el botón derecho del ratón en cada uno de los vértices. Una segunda opción es dibujar un área que encuadre los vértices, para ello hay que pulsar primero la tecla B (Pulsa el botón intermedio del ratón y muévelo para cambiar la perspectiva y poder asegurar así que se han elegido los vértices correctos).

En la parte derecha, en el editor UV se puede observar un cuadrado con los bordes de color amarillo y la superficie morada. Este recuadro corresponde con la cara del cubo seleccionada y con el que se pueden establecer correspondencias con coordenadas de la textura. El siguiente paso consistirá en adaptar la superficie del recuadro a la imagen que debe aparecer en la parte superior del cubo (z+), tal como indica la Figura 4.10. Para adaptar la superficie existen varias alternativas; una de ellas es escalar el cuadrado con la tecla S y para desplazarlo la tecla G (para hacer zoom sobre la textura se gira la rueda central del ratón). Un segunda opción es ajustar cada vértice de forma individual, para ello se selecciona un vértice con el botón derecho del ratón, pulsamos G y desplazamos el vértice a la posición deseada. Las dos alternativas descritas anteriormente no son excluyentes y se pueden combinar, es decir, se podría escalar el recuadro en primer lugar y luego ajustar los vértices, e incluso escalar de nuevo si fuera necesario.

Los pasos que se han realizo para establecer la textura de la parte superior del cubo deben ser repetidos para el resto de caras del cubo. Si renderizamos la escena pulsando F12 podemos apreciar el resultado final.

Por último vamos a empaquetar el archivo *.blend* para que sea autocontenido. La textura es un recurso externo; si cargáramos el fichero *.blend* en otro ordenador posiblemente no se visualizaría debido a que las rutas no coinciden. Si empaquetamos el fichero eliminamos este tipo de problemas. Para ello, seleccionamos *File >External Data >Pack into .blend file*.

4.2.4. Exportación del objeto en formato Ogre XML

Para exportar la escena de Blender al formato de Ogre seleccionamos *File >Export >OGRE Meshes*. En la parte inferior aparecerá un nuevo panel con las opciones del exportador tal como muestra la Figura 4.13. Tal como se puede apreciar, el exportador dispone de múltiples opciones encuadradas en dos categorías: exportación de materiales y exportación de mallas.

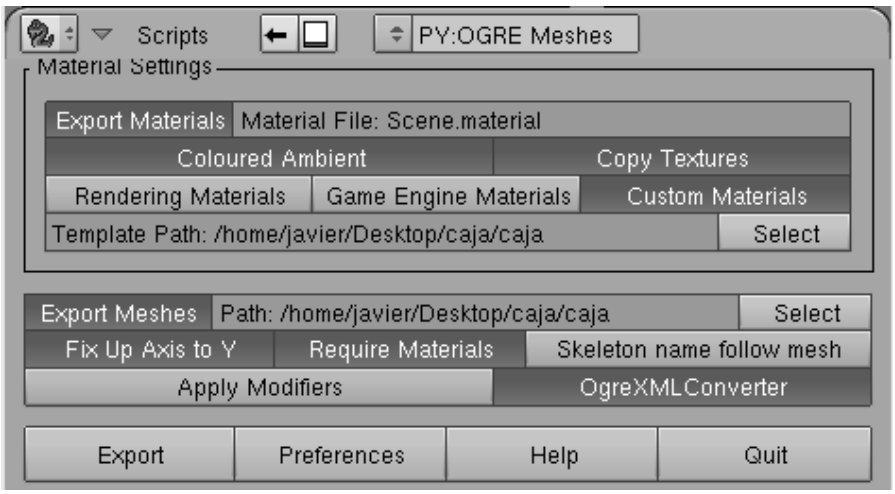


Figura 4.13: Exportador de Ogre integrado en Blender

Si se selecciona la opción de exportación de materiales, el exportador generará un archivo *.material* con el material asociado al modelo. A continuación se incluye una breve descripción de las opciones del exportador asociadas al material [Ibe12]:

- **Material File:** Se establece la ruta y el nombre del fichero con extensión *.material*.
- **Coloured Ambient:** Si se marca esta opción se utiliza como color ambiente el color que se indique en Blender en *diffuse colour* (opciones del material) . Está opción funciona únicamente si la opción *TextFace* no está activada (opción que también se encuentra en los materiales asignados a un objeto en Blender).
- **Copy Textures:** realiza una copia de las imágenes de las texturas al directorio de exportación.

- **Rendering Materials:** Exporta el fichero *.material* tal y como se visualizaría en el renderizado de Blender.
- **Game Engine Materials:** Exporta el fichero *.material* tal y como se visualizaría dentro del motor *Blender Engine*.
- **Custom Materials:** exporta los materiales utilizando plantillas especializadas.

El siguiente bloque de opciones corresponde a la exportación de mallas. Por cada objeto en la escena se creará un fichero *.mesh* [Ibe12].

- **Fix Up Axis to Y:** Blender utiliza un sistema de coordenadas distinto al de Ogre. Si no activamos esta opción es posible que los modelos creados en blender aparezcan invertidos al cargarlos en una aplicación de Ogre. En Ogre el eje de Y corresponde con el eje Z de Blender.
- **Require Materials:** Incluye una referencia al fichero *.material* desde el propio modelo. Si se activa no será necesario cargar manualmente el material una vez que creemos la entidad con el modelo.
- **Skeleton Name Follow Mesh:** Asocia el fichero *.skeleton* con la información sobre el esqueleto y los huesos del modelo al fichero *.mesh*.
- **Apply Modifiers:** Aplica los modificadores que hayamos establecido en Blender (por ejemplo, un espejo en un determinado eje para conseguir la simétria del objeto).
- **OgreXMLConverter:** Por defecto, el exportador convierte la escena de Blender en formato XML que aún no es válido para utilizarlo directamente desde OGRE. Cuando instalamos OGRE en nuestro ordenador, se instala también una herramienta llamada OgreXMLConverter que se puede utilizar directamente desde Blender, si está correctamente enlazado. Si activamos esta opción se transforma el fichero *.xml* en el binario reconocido por OGRE *.mesh*. Si el exportador no encontrara la ruta del ejecutable OgreXMLConverter se le puede indicar manualmente desde las opciones.

Dada la descripción de las opciones del exportador, marcamos las siguientes opciones para exportar el cubo con las texturas:

- Exportación de materiales:
  - Nombre del fichero *.material*.
  - Copy Textures
  - Custom Materials
- Exportación de la malla
  - Ruta donde se almacenará los ficheros XML y *.mesh*.

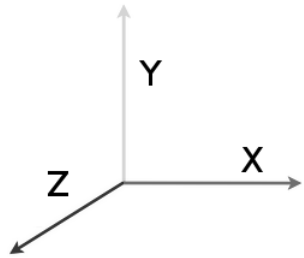


Figura 4.14: Sistema de coordenadas utilizado en Ogre.

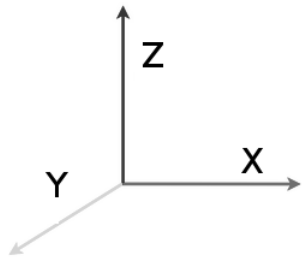


Figura 4.15: Sistema de coordenadas utilizado en Blender.

- Fix Up Axis to Y
- Require Materials
- OgreXMLConverter

4.2.5. Carga del objeto en una aplicación Ogre

Para cargar el cubo exportado en Ogre vamos a partir de ejemplo visto al comienzo del curso *Hello World*. Recordemos que la estructura de directorios para la aplicación en Ogre era la siguiente:

- Directorios:
  - Include
  - Media
  - Obj
  - Plugins
  - src
- Ficheros en el directorio raíz del proyecto Ogre:
  - ogre.cfg
  - plugins.cfg
  - resources.cfg

Una vez que se ha exportado el cubo con las texturas asociadas y se han generado los ficheros *Cube.mesh*, *Cube.mesh.xml* y *Scene.material*, los incluimos en el directorio *media* (también incluimos la textura "textura512.jpg"). Los ficheros deben incluirse en este directorio porque así está configurado en el fichero *resources.cfg*, si deseáramos incluir los recursos en otro directorio deberíamos variar la configuración en dicho fichero.

En segundo lugar, cambiamos el nombre del ejecutable; para ello, abrimos el archivo *makefile* y en lugar de *EXEC := helloWorld*, ponemos *EXEC:= cubo*. Después es necesario cambiar el código de ejemplo para que cargue nuestro modelo. Abrimos el archivo */src/main.cpp* que se encuentra dentro en el directorio */src*. El código es el siguiente:

Listado 4.21: Código incluido en el archivo main.cpp

```
1  #include <ExampleApplication.h>
2  class SimpleExample : public ExampleApplication {
3      public : void createScene() {
4          Ogre::Entity *ent = mSceneMgr->createEntity("Sinbad", "
               Sinbad.mesh");
5          mSceneMgr->getRootSceneNode()->attachObject(ent);
6      }
7  };
8
9  int main(void) {
10     SimpleExample example;
11     example.go();
12     return 0;
13 }
```

En nuestro caso la entidad a crear es aquella cuya malla se llama *Cube.Mesh*, ésta ya contiene información sobre la geometría de la malla y las coordenadas UV en el caso de utilización de texturas. Por lo tanto el código tras la modificación de la función *createScene*, sería el siguiente:

Listado 4.22: Modificación del fichero main.cpp para cargar el cubo modelado en Blender

```
1 public : void createScene() {
2     Ogre::Entity *ent = mSceneMgr->createEntity("Caja", "cubo.mesh"
          );
3     mSceneMgr->getRootSceneNode()->attachObject(ent);
4 }
```

Al compilar (make) y ejecutar (./Cubo) aceptamos las opciones que vienen por defecto y se puede observar el cubo pero muy lejano. A partir de ahora todas las operaciones que queramos hacer (translación, escalado, rotación,...) tendrá que ser a través de código. A continuación se realiza una operación de traslación para acercar el cubo a la cámara:

Listado 4.23: Traslación del cubo para acercarlo hacia la cámara

```
1 Entity *ent1 = mSceneMgr->createEntity( "Cubo", "Cube.mesh" );
2 SceneNode *node1 = mSceneMgr->getRootSceneNode()->
      createChildSceneNode( );
3 node1->attachObject( ent1 );
4 node1->translate(Vector3(0, 0, 490 ));
```

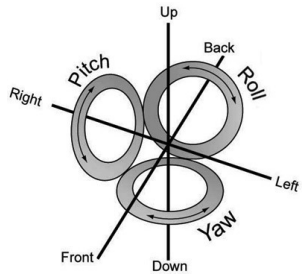


Figura 4.16: Operaciones de rotación.

Ejercicio: intentar escalar el cubo y rotarlo en función de los ejes usando *Pitch*, *Yaw* y *Roll*.

Listado 4.24: Ejemplo de escalado de un objeto

```
1 node1->scale( 3, 3, 3 );
```

Listado 4.25: Ejemplo de rotación

```
1 node1->yaw(Ogre::Degree( -90 ) );
2 node1->pitch(Ogre::Degree( -90 ) );
3 node1->roll(Ogre::Degree( -90 ) );
```

4.3. Procesamiento de Recursos Gráficos

En esta sección estudiaremos cómo adaptar recursos 3D realizados con Blender en Ogre. Tendremos en cuenta aspectos relativos a la escala, posicionamiento de los objetos (respecto de su sistema de referencia local y global) y estudiaremos algunas herramientas disponibles en Blender para el posicionamiento preciso de modelos 3D.

Antes de continuar, el lector podría (opcionalmente) completar el estudio del capítulo 6, ya que utilizaremos como base el código fuente obtenido en el último ejemplo de dicho capítulo, y estudiaremos algunas cuestiones referentes al gestor de recursos.

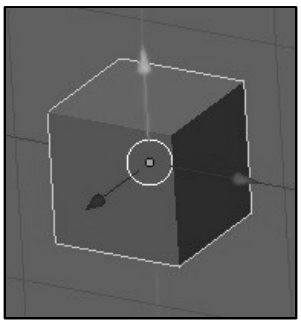


Figura 4.17: El centro del objeto define el origen del sistema de referencia local. Así, la posición de los vértices del cubo se definen según este sistema local.

Blender, a diferencia de los sistemas de CAD (que emplean modelos de Geometría Sólida Constructiva *CSG*), es una herramienta de modelado de contorno B-Rep (*Boundary Representation*). Esto implica que, a diferencia de los sistemas basados en CSG, trabaja con modelos huecos definidos por vértices, aristas y caras. Como hemos estudiado en el capítulo de introducción matemática, las coordenadas de estos modelos se especifican de forma relativa a su centro, que define el origen de su sistema de referencia local.

En Blender el centro del objeto se representa mediante un punto de color rosa (ver Figura 4.17). Si tenemos activos los manejadores en la cabecera de la ventana 3D, será el punto de donde comienzan estos elementos de la interfaz.

Este centro define el sistema de referencia local, por lo que resulta crítico poder cambiar el centro del objeto ya que, tras su exportación, se dibujará a partir de esta posición.

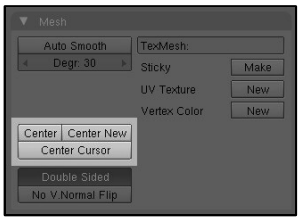


Figura 4.18: Botones para el cambio del centro del objeto, en la pestaña *Mesh*.

Con el objeto seleccionado en Modo de Objeto se puede indicar a Blender que recalcule el centro geométrico del objeto en los botones de edición, en la pestaña Mesh (ver Figura 4.18), pulsando en el botón *Center New*. Se puede cambiar el centro del objeto a cualquier posición del espacio situando el puntero 3D (pinchando con) y pulsando posteriormente en *Center Cursor*.

Si queremos situar un objeto virtual correctamente alineado con un objeto real, será necesario situar con absoluta precisión el centro y los vértices que lo forman. Para situar con precisión numérica los vértices del modelo, basta con pulsar la tecla N en modo edición. En la vista 3D aparecerá una ventana como la que se muestra en la Figura 4.19, en la que podremos especificar las coordenadas específicas de cada vértice. Es interesante que esas coordenadas se especifiquen *localmente* (botón Local activado), porque el exportador de Ogre trabajará con coordenadas locales relativas a ese centro.

El centro del objeto puede igualmente situarse de forma precisa, empleando la posición del puntero 3D. El puntero 3D puedes situarse en cualquier posición del espacio con precisión, accediendo al menú *View/ View Properties* de la cabecera de una ventana 3D. En los campos de *3D Cursor* podemos indicar numéricamente la posición del cursor 3D (ver Figura 4.21) y posteriormente utilizar el botón *Center Cursor* (ver Figura 4.18).

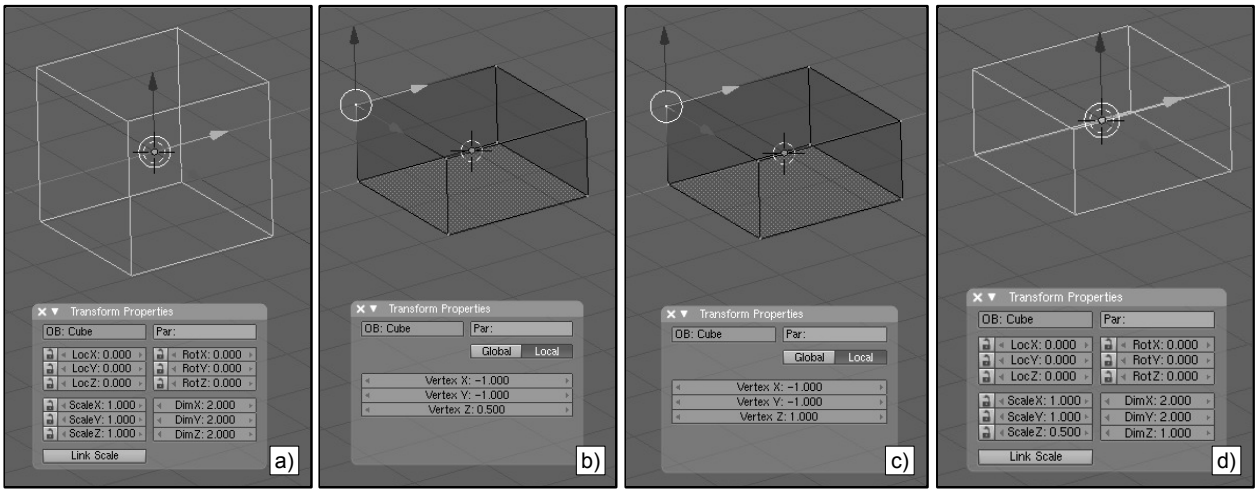


Figura 4.20: Aplicación de transformaciones geométricas en modo Edición. **a)** Situación inicial del modelo al que le aplicaremos un escalado de 0.5 respecto del eje Z. **b)** El escalado se aplica en modo Edición. El campo Vertex Z muestra el valor correcto. **c)** Si el escalado se aplicó en modo objeto, el campo Vertex Z no refleja la geometría *real* del objeto. Esto puede comprobarse fácilmente si alguno de los campos *Scale* no son igual a uno, como en **d)**.

Es muy importante que las transformaciones de modelado se apliquen finalmente a las coordenadas locales del objeto, para que se apliquen de forma efectiva a las coordenadas de los vértices. Es decir, si después de trabajar con el modelo, pulsando la tecla N en modo Objeto, el valor *ScaleX*, *ScaleY* y *ScaleZ* es distinto de 1 en alguno de los casos, implica que esa transformación se ha realizado en modo objeto, por lo que los vértices del mismo no tendrán esas coordenadas asociadas.

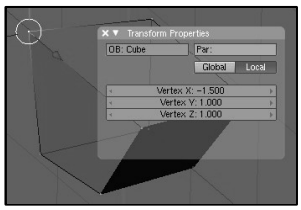


Figura 4.19: Ventana emergente al pulsar la tecla *N* para indicar los valores numéricos de coordenadas de los vértices del modelo.

De igual modo, también hay que prestar atención a la rotación del objeto. La Figura 4.20 muestra un ejemplo de este problema; el cubo de la izquierda se ha escalado en modo edición, mientras que el de la derecha se ha escalado en modo objeto. El modelo de la izquierda se exportará correctamente, porque los vértices tienen asociadas sus coordenadas locales. El modelo de la derecha sin embargo aplica una transformación a nivel de objeto, y se exportará como un cubo perfecto.

Blender dispone de una orden para aplicar las transformaciones realizadas en modo Objeto al sistema de coordenadas local del objeto. Para ello, bastará con acceder al menú *Barra Espaciadora / Transform / Clear/Apply/ Apply Scale/Rotation to ObData* (o accediendo mediante el atajo de teclado Control A).

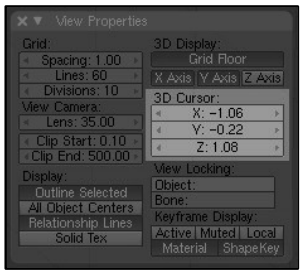


Figura 4.21: Posicionamiento numérico del cursor 3D.

Otra operación muy interesante consiste en posicionar un objeto con total precisión. Para realizar esta operación puede ser suficiente con situarlo numéricamente, como hemos visto anteriormente, accediendo a las propiedades de transformación mediante la tecla N. Sin embargo, otras veces necesitamos situarlo en una posición relativa a otro objeto; necesitaremos situarlo empleando el cursor 3D.

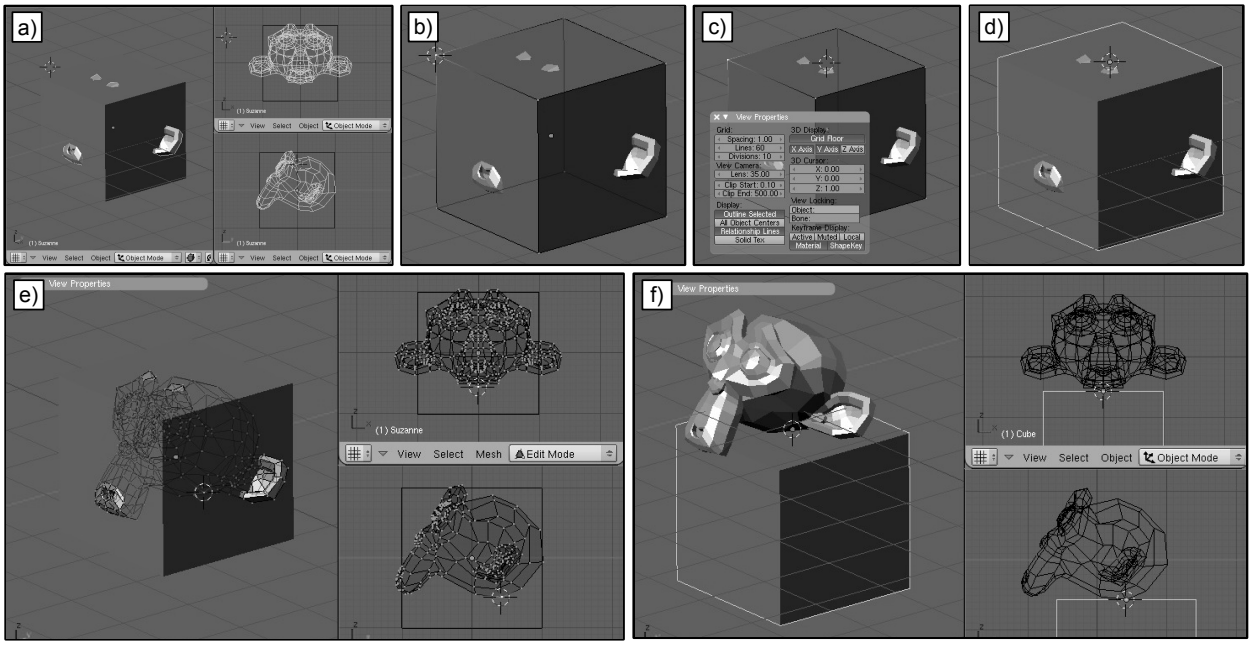


Figura 4.22: Ejemplo de uso del operador *Cursor to Selection* y *Selection to Cursor*. **a)** Posición inicial de los dos modelos. Queremos llegar a la posición *f)*. En modo edición, elegimos un vértice superior del cubo y posicionamos el puntero 3D ahí (empleando *Shift S Cursor ->Selection*). **c)** A continuación modificamos el valor de X e Y de la posición del cursor numéricamente, para que se sitúe en el centro de la cara. **d)** Elegimos *Center Cursor* de la pestaña *Mesh* para cambiar el centro del objeto. Ahora el cubo tiene su centro en el punto medio de la cara superior. **e)** Elegimos el vértice señalado en la cabeza del mono y posicionamos ahí el puntero 3D. Finalmente en **f)** cambiamos la posición del cubo de forma que interseca exactamente en ese vértice (*Shift S Selection ->Cursor*.

El centro de un objeto puede situarse en la posición del puntero 3D, y el puntero 3D puede situarse en cualquier posición del espacio. Podemos, por ejemplo, en modo edición situar el puntero 3D en la posición de un vértice de un modelo, y situar ahí el centro del objeto. Desde ese momento, los desplazamientos del modelo se harán tomando como referencia ese punto.

Mediante el atajo Shift S *Cursor ->Selection* podemos situar el puntero 3D en la posición de un elemento seleccionado (por ejemplo, un vértice, o en el centro de otro objeto que haya sido seleccionado en modo objeto) y mediante Shift S *Selection ->Cursor* moveremos el objeto seleccionado a la posición del puntero 3D (ver Figura 4.22). Mediante este sencillo mecanismo de 2 pasos podemos situar los objetos con precisión, y modificar el centro del objeto a cualquier punto de interés.

Una vez que se ha posicionado el cursor 3D, podemos cambiar su posición empleando valores numéricos (por ejemplo, cambiar su valor Z para que sea exactamente 0), mediante las propiedades de la vista (*View Properties*) como se muestra en la Figura 4.21.

4.3.1. Ejemplo de uso

A continuación veremos un ejemplo de utilización de los operadores comentados anteriormente. Partiremos de un sencillo modelo (de 308 vértices y 596 caras triangulares) creado en Blender, que se muestra en la Figura 4.23. Al importar el modelo en Ogre, se ha creado un plano manualmente que sirve como "*base*", y que está posicionado en $Y = 0$ (con vector normal el Y unitario). El siguiente fragmento de código muestra las líneas más relevantes relativas a la creación de la escena.

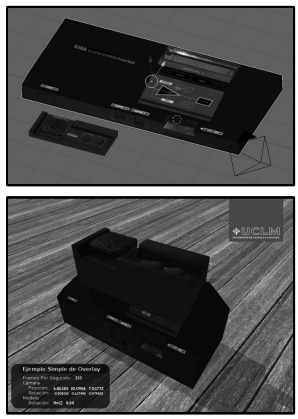


Figura 4.23: Resultado de exportar directamente el modelo de la *Master System* desde Blenderx. La imagen superior muestra el modelo en Blender, y las inferior el resultado de desplegar el modelo en Ogre, con proporciones claramente erróneas.

Listado 4.26: Fragmento de MyApp.c

```
1  int MyApp::start() {
2    // ... Carga de configuracion, creacion de window ...
3    Ogre::Camera* cam = _sceneManager->createCamera("MainCamera");
4    cam->setPosition(Ogre::Vector3(5,20,20));
5    cam->lookAt(Ogre::Vector3(0,0,0));
6    cam->setNearClipDistance(5);
7    cam->setFarClipDistance(100);
8    // ... Viewport, y createScene...
9  }
10
11 void MyApp::createScene() {
12   Ogre::Entity* ent1 = _sceneManager->createEntity("MS.mesh");
13   Ogre::SceneNode* node1 = _sceneManager->createSceneNode("MS");
14   node1->attachObject(ent1);
15   node1->translate(0,2,0);
16   _sceneManager->getRootSceneNode()->addChild(node1);
17
18   Ogre::Entity* ent2 = _sceneManager->createEntity("Mando.mesh");
19   Ogre::SceneNode* node2 = _sceneManager->createSceneNode("Mando");
20   node2->attachObject(ent2);
21   node2->translate(0,2,0);
22   node1->addChild(node2);
23   // ... creacion del plano, luces, etc...
24 }
```

El problema viene asociado a que el modelo ha sido construido empleando transformaciones a nivel de objeto. Si accedemos a las propiedades de transformación [N] para cada objeto, obtenemos la información mostrada en la Figura 4.24. Como vemos, la escala a nivel de objeto de ambos modelos es distinta de 1.0 para alguno de los ejes, lo que indica que la transformación se realizó a nivel de objeto.

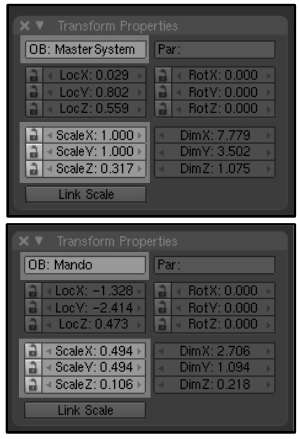


Figura 4.24: Propiedades de transformación de los dos objetos del ejemplo.

Aplicaremos la escala (y la rotación, aunque en este caso el objeto no fue rotado) a los vértices del modelo, eliminando cualquier operación realizada en modo objeto. Para ello, con cada objeto seleccionado, pulsaremos [Control] [A] *Apply Object / Scale and Rotation to ObData*. Ahora las propiedades de transformación (de la Figura 4.24) deben mostrar una escala de 1.0 en cada eje.

Como hemos visto, la elección correcta del centro del objeto facilita el código en etapas posteriores. Si el modelo está normalizado (con escala 1.0 en todos los ejes), las coordenadas del espacio 3D de Blender pueden ser fácilmente transformadas a coordenadas En este caso, vamos a situar el centro de cada objeto de la escena de forma que esté situado exactamente en el $Z = 0$. Para ello, con cada objeto seleccio-

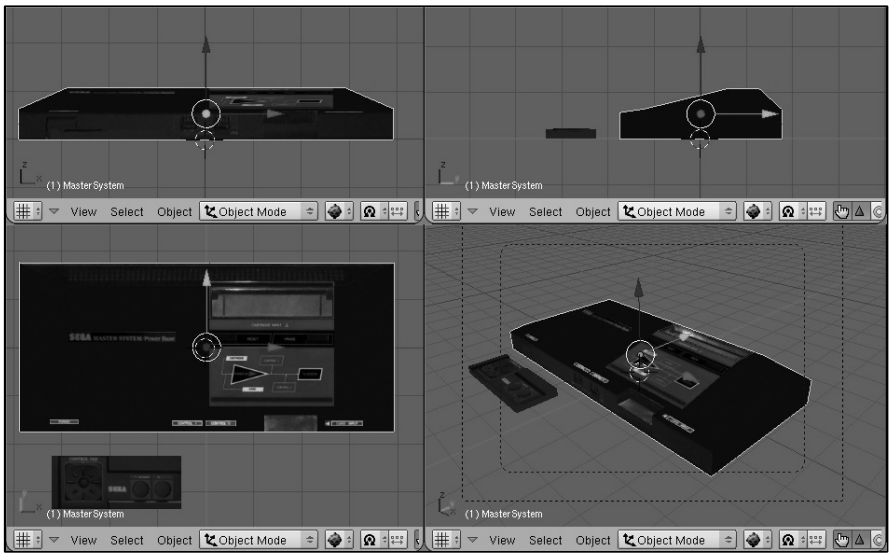


Figura 4.25: Posicionamiento del puntero 3D en relación al centro del objeto. En muchas situaciones puede ser conveniente especificar que varios objetos tengan el centro en una misma posición del espacio (como veremos en la siguiente sección). Esto puede facilitar la construcción de la aplicación.

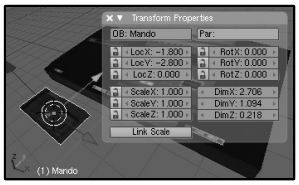


Figura 4.26: Posiconamiento del objeto mando.

nado, pulsaremos en **Center New** (de la pestaña *Mesh* de los botones de edición).

Ahora posicionaremos el cursor 3D en esa posición **Shift** **S** *Cursor ->Selection* y modificaremos numéricamente la coordenada del cursor 3D, para que se sitúe en $Z = 0$ (accediendo a la pestaña *View Properties* de la cabecera de la ventana 3D). El resultado de posicionar el cursor 3D se muestra en la Figura 4.25. Finalmente pulsaremos en **Center Cursor** de la pestaña *Mesh* para situar el centro en la posición del puntero 3D.

Sistemas de Coordenadas: Recordemos que los sistemas de coordenadas de Ogre y Blender siguen convenios diferentes. Si utilizamos la opición del exportador de Ogre *Flip Up Axis to Y* (por defecto), para obtener las coordenadas equivalentes de Ogre desde Blender bastará con aplicar la misma coordenada X, la coordenada Z de Blender utilizarla en Y en Ogre, y la coordenada Y de Blender aplicarla invertida en Z en Ogre.

Ahora podemos exportar el modelo a Ogre, y las proporciones se mantendrán correctamente. Sería conveniente posicionar exactamente el mando en relación a la consola, tal y como está en el modelo .blend. Para ello, podemos modificar y anotar manualmente la posición de objeto 3D (en relación con el sistema de referencia universal *SRU*). Como el centro de la consola se ha posicionado en el origen del SRU,

la posición del centro del mando será relativa al centro de la consola. Bastará con consultar estos valores (LocX, LocY, LocZ) en el *Transform Properties* (ver Figura 4.26) y utilizarlos en Ogre.

En el ejemplo de la Figura 4.26 que tiene asociadas en Blender las coordenadas (-1.8, -2.8, 0), obtendríamos el equivalente en Ogre de (-1.8, 0, 2.8). De este modo, el siguiente fragmento de código muestra la creación de la escena correcta en Ogre. En este caso ya no es necesario aplicar ninguna traslación al objeto *MS* (se posicionará en el origen del SRU). Por su parte, al objeto *Mando* se aplicará la traslación indicada en la línea 9, que se corresponde con la obtenida de Blender.

Listado 4.27: Fragmento de MyApp.c

```
1  Ogre::Entity* ent1 = _sceneManager->createEntity("MS.mesh");
2  Ogre::SceneNode* node1 = _sceneManager->createSceneNode("MS");
3  node1->attachObject(ent1);
4  _sceneManager->getRootSceneNode()->addChild(node1);
5
6  Ogre::Entity* ent2 = _sceneManager->createEntity("Mando.mesh");
7  Ogre::SceneNode* node2 = _sceneManager->createSceneNode("Mando");
8  node2->attachObject(ent2);
9  node2->translate(-1.8,0,2.8);
10 node1->addChild(node2);
```

Automatízate!!

Obviamente, el proceso de exportación de los datos relativos al posicionamiento de objetos en la escena (junto con otros datos de interés) deberán ser automatizados definiendo un formato de escena para Ogre. Este formato tendrá la información necesaria para cada juego particular.

Por último, sería deseable poder exportar la posición de las cámara para percibir la escena con la misma configuración que en Blender. Existen varias formas de configurar el *camera pose*. Una de las más cómodas para el diseñador es trabajar con la posición de la cámara y el punto al que ésta mira. Esta especificación es equivalente a indicar la posición y el punto *look at* (en el primer fragmento de código de la sección, en las líneas 4 y 5).

Para que la cámara apunte hacia un objeto de la escena, puede crearse una restricción de tipo *Track To*. Para ello, añadimos un objeto vacío a la escena (que no tiene representación), mediante Shift A *Add / Empty*. Ahora seleccionamos primero la cámara, y luego con Shift pulsado seleccionamos el Empty y pulsamos Control T *Track To Constraint*. De este modo, si desplazamos la cámara, obtendremos que siempre está *mirando* hacia el objeto Empty creado (ver Figura 4.27).

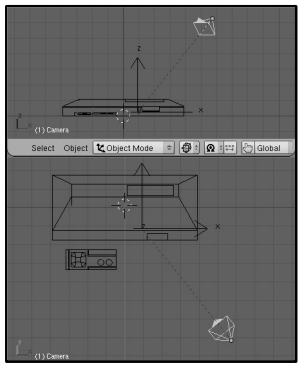
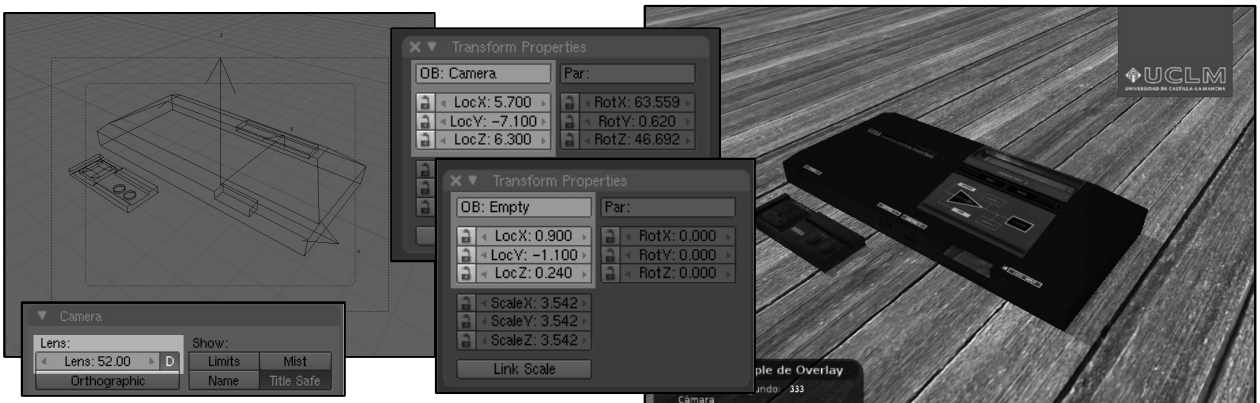


Figura 4.27: Track de la cámara al objeto Empty. Blender representa la relación con una línea punteada que va de la cámara al objeto Empty.

Cuando tengamos la "fotografía" de la escena montada, bastará con consultar el valor de posición de la cámara y el Empty, y asignar esos valores al código de posicionamiento y *look at* de la misma (teniendo en cuenta las diferencias entre sistemas de coordenadas de Blender y Ogre). El resultado se encuentra resumido en la Figura 4.28.

4.4. Gestión de Recursos y Escena

En esta sección estudiaremos un ejemplo que cubre varios aspectos que no han sido tratados en el documento, relativos al uso del gestor de recursos y de escena. Desarrollaremos un demostrador de *3D Picking* que gestionará manualmente el puntero del ratón (mediante *overlays*), gestionará recursos empaquetados en archivos `.zip` (empleando




```
cam->setPosition(Vector3(5.7,6.3,7.1));
cam->lookAt(Vector3(0.9,0.24,1.1));
cam->setFOVy(Ogre::Degree(52));
```

Figura 4.28: Resultado de aplicar la misma cámara en Blender y Ogre.

las facilidades que proporciona Ogre), cargará geometría estática y utilizará el potente sistema de *queries* del *SceneManager* con máscaras.

La Figura 4.30 muestra el interfaz del ejemplo desarrollado. Mediante la rueda del ratón es posible desplazar la cámara. Si se pulsa la rueda, se aplicará un rotación sobre la misma. Si se pulsa el botón izquierdo del ratón sobre un objeto de colisión del escenario (como el mostrado en la figura), se seleccionará mostrando su caja límite *bounding box*. Si se pulsa con el botón izquierdo sobre el suelo de la sala, se añadirá aleatoriamente una caja de tipo 1 o 2. Si pinchamos con el botón izquierdo sobre alguna de las cajas creadas, se seleccionará. Mediante el botón derecho únicamente podremos seleccionar cajas creadas (no se añadirán nunca cajas, aunque pulsemos sobre el suelo de la sala).

Sobre una caja creada, podemos aplicar tres operaciones: con la tecla Supr eliminamos el objeto. Con la tecla R rotaremos la caja respecto de su eje Y. Finalmente, con la tecla S modificamos su escala. El operador de escala y de rotación permiten invertir su comportamiento si pulsamos simultáneamente la tecla Shift. Veamos a continuación algunos aspectos relevantes en la construcción de este ejemplo.

4.4.1. Recursos empaquetados

```
[General]
FileSystem=media
Zip=media/cube.zip
Zip=media/stage.zip
Zip=media/colision.zip
Zip=media/overlay.zip
```

Figura 4.29: Contenido del archivo de configuración de recursos *resource.cfg*.

El gestor de recursos de Ogre permite utilizar ficheros `.zip` definiendo en el archivo de configuración de recursos que el tipo es *Zip*.

De este modo, el archivo de configuración de recursos asociado a este ejemplo se muestra en la Figura 4.29. La implementación del cargador de recursos que estudiamos en el capítulo 6 permite su utilización directamente.

Figura 4.30: Ejemplo de aplicación que desarrollaremos en esta sección. El interfaz permite seleccionar objetos en el espacio 3D y modificar algunas de sus propiedades (escala y rotación).

4.4.2. Gestión del ratón

En el siguiente listado se resume el código necesario para posicionar una imagen usada como puntero del ratón en Ogre. La Figura 4.31 define el material creado (textura con transparencia) para cargar la imagen que emplearemos como puntero.

```
material pointer{
 technique {
  pass {
   scene_blend src_alpha
     one_minus_src_alpha
   texture_unit {
     texture cursor.png
}}}}
```

Figura 4.31: Material asociado al puntero del ratón.

En la línea [11] se posiciona el elemento del overlay llamado *cursor* en la posición absoluta obtenida del ratón por OIS (líneas [1] y [2]). En el constructor del *FrameListener* hay que indicar a OIS las dimensiones de la ventana, para que pueda posicionar adecuadamente el ratón (de otra forma, trabajará en un cuadrado de 50 píxeles de ancho y alto). Esta operación se realiza como se indica en las líneas [14-15].

El desplazamiento de la rueda del ratón se obtiene en la coordenada Z con *getMouseState* (línea [5]). Utilizamos esta información para desplazar la cámara relativamente en su eje local Z.

Listado 4.28: Fragmentos de MyFrameListener.cpp

```
1  int posx = _mouse->getMouseState().X.abs;   // Posicion del puntero
2  int posy = _mouse->getMouseState().Y.abs;   // en pixeles.
3
4  // Si usamos la rueda, desplazamos en Z la camara -----------
5  vt+= Vector3(0,0,-10)*deltaT * _mouse->getMouseState().Z.rel;
6  _camera->moveRelative(vt * deltaT * tSpeed);
7
8  // Gestion del overlay ----------------------------
9  OverlayElement *oe;
10 oe = _overlayManager->getOverlayElement("cursor");
11 oe->setLeft(posx);  oe->setTop(posy);
12
13 // En el constructor de MyFrameListener...
14 _mouse->getMouseState().width = _win->getWidth();
15 _mouse->getMouseState().height = _win->getHeight();
```

4.4.3. Geometría Estática

Como hemos estudiado anteriormente, el modelo abstracto de los *MovableObject* abarca multitud de tipos de objetos en la escena (desde luces, cámaras, entidades, etc...). Uno de los tipos más empleados son las mallas poligionales, que tienen asociada información geométrica y datos específicos para el posterior uso de materiales.

La **geometría estática**, como su nombre indica, está pensada para elementos gráficos que no modifican su posición durante la ejecución de la aplicación. Está pensada para enviar *pocos* paquetes (lotes) *grandes* de geometría a la GPU en lugar de muchos pequeños. Esto permite optimizar el rendimiento en su posterior despliegue. De este modo, a menor número de paquetes enviados a la GPU tendremos mayor rendimiento en la aplicación.

Sobre eficiencia...

Realizar 100 llamadas de 10.000 polígonos a la GPU puede ser un orden de magnitud menos eficiente que realizar 10 llamadas de 100.000 polígonos.

El uso de la memoria empleando *geometría estática* sin embargo es mayor. Mientras que los *MovableObject* comparten mallas poligonales (siempre que sean instancias del mismo objeto), empleando geometría estática se copian los datos para cada instancia de la geometría.

Existen algunas características principales que deben tenerse en cuenta a la hora de trabajar con geometría estática:

- **Construcción**. La geometría estática debe ser construída previamente a su uso. Esta etapa debe realizarse una única vez, por lo que puede generarse en la carga del sistema y no supone una carga real en la tasa de refresco interactiva del juego.
- **Gestión de materiales**. El número de materiales diferentes asociados a la geometría estática delimita el número de lotes (paquetes) que se enviarán a la GPU. De este modo, aunque un bloque de geometría estática contenga gran cantidad de polígonos, se crearán tantos paquetes como diferentes materiales tenga asociado el bloque. Así, existe un compromiso de eficiencia relacionado con el número de materiales diferentes asociados a cada paquete de geometría estática.
- **Rendering en grupo**. Aunque únicamente una pequeña parte de la geometría estática sea visible en el *Frustum* todo el paquete será enviado a la GPU para su representación. De este modo, es conveniente separar la geometría estática en diferentes bloques para evitar el despliegue global de todos los elementos.

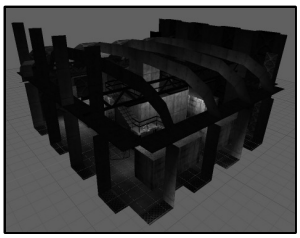


Figura 4.32: El escenario del ejemplo está realizado en baja poligonalización (1616 vértices y 1003 caras). Aun así, el uso de geometría estática *acelera* su despliegue en 3x!.

En el siguiente fragmento de código relativo a MyApp.c se muestra las instrucciones necesarias para definir un paquete de geometría estática en Ogre. Basta con llamar al *SceneManager*, e indicarle el nombre del bloque a definir (en este caso "SG" en la línea 1). A continuación se añade una entidad cargada desde un archivo `.mesh`. Finalmente en la línea 4 se ejecuta la operación de construcción del bloque de geometría.

Listado 4.29: Fragmento de MyApp.c

```
1 StaticGeometry* stage = _sceneManager->createStaticGeometry("SG");
2 Entity* ent1 = _sceneManager->createEntity("Escenario.mesh");
3 stage->addEntity(ent1, Vector3(0,0,0));
4 stage->build();  // Operacion para construir la geometria
```

Otro aspecto importante a tener en cuenta a la hora de trabajar con geometría estática es que no es tenida en cuenta a la hora de realizar preguntas al gestor de escena. En la siguiente sección veremos cómo *resolver* este inconveniente.

4.4.4. Queries

El gestor de escena permite resolver preguntas relativas a la relación espacial entre los objetos de la escena. Estas preguntas pueden planterase relativas a los objetos móviles *MovableObject* y a la geometría del mundo *WorldFragment* (que estudiaremos en el capítulo de *Optimización de Exteriores* en este documento).

Como se ha comentado anteriormente, Ogre no gestiona las preguntas relativas a la geometría estática. Una posible solución pasa por crear objetos móviles invisibles de baja poligonalización que servirán para realizar estas preguntas. Estos objetos están exportados conservando las mismas dimensiones y rotaciones que el bloque de geometría estática (ver Figura 4.33). Además, para evitar tener que cargar manualmente los centros de cada objeto, todos los bloques han redefinido su centro en el origen del SRU (que coincide con el centro de la geometría estática). El siguiente listado muestra la carga de estos elementos en el grafo de escena. Veamos algunos detalles del código.

Listado 4.30: Fragmento de MyApp.cpp (Create Scene)

```
1  // Objeto movable "suelo"para consultar al SceneManager
2  SceneNode *nodecol = _sceneManager->createSceneNode("Col_Suelo");
3  Entity *entcol = _sceneManager->createEntity
4                                  ("Col_Suelo", "Col_Suelo.mesh");
5  entcol->setQueryFlags(STAGE);   // Usamos flags propios!
6  nodecol->attachObject(entcol);
7  nodecol->setVisible(false);     // Objeto oculto
8  _sceneManager->getRootSceneNode()->addChild(nodecol);
9
10 // Cajas del escenario (baja poligonalizacion)
11 stringstream sauxnode, sauxmesh;
12 string s = "Col_Box";
13 for (int i=1; i<6; i++) {
14   sauxnode << s << i; sauxmesh << s << i << ".mesh";
15   SceneNode *nodebox = _sceneManager->createSceneNode
16                                                (sauxnode.str());
17   Entity *entboxcol = _sceneManager->createEntity
18                                (sauxnode.str(), sauxmesh.str());
19   entboxcol->setQueryFlags(STAGE);    // Escenario
20   nodebox->attachObject(entboxcol);
21   nodebox->setVisible(false);
22   nodecol->addChild(nodebox);
23   sauxnode.str(""); sauxmesh.str(""); // Limpiamos el stream
24 }
```

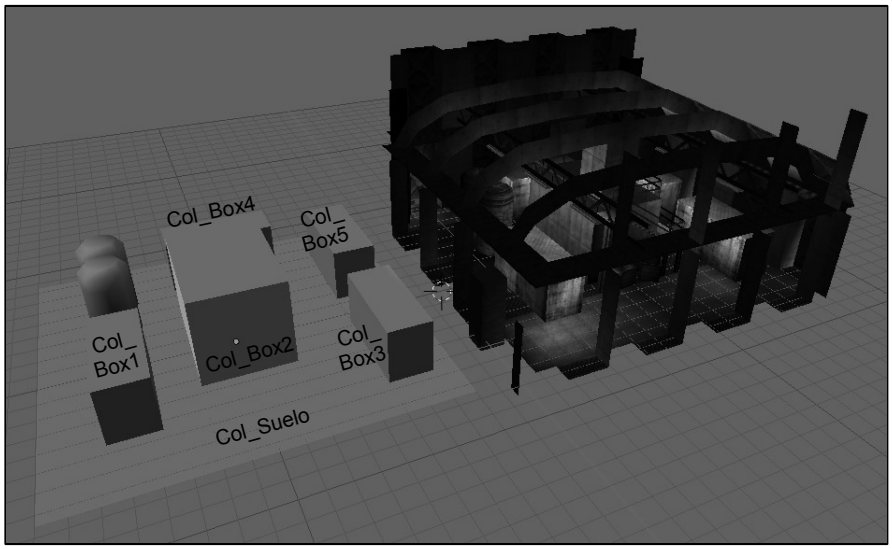


Figura 4.33: A la izquierda la geometría utilizada para las preguntas al gestor de escena. A la derecha la geometría estática. El centro de todas las entidades de la izquierda coinciden con el centro del modelo de la derecha. Aunque en la imagen los modelos están *desplazados* con respecto del eje X, en realidad ambos están exactamente en la misma posición del mundo.

Cabe destacar el establecimiento de flags propios (en las líneas 5 y 19) que estudiaremos en la sección 4.4.5, así como la propiedad de visibilidad del nodo (en 7 y 21). El bucle definido en 13–23 sirve para cargar las cajas límite mostradas en la Figura 4.33.

Las preguntas que pueden realizarse al gestor de escena se dividen en cuatro categorías principales: **1. Caja límite**, definida mediante dos vértices opuestos, **2. Esfera**, definida por una posición y un radio, **3. Volumen**, definido por un conjunto de tres o más planos y **4. Rayo**, denifido por un punto (origen) y una dirección.

Otras intersecciones

Ogre soporta igualmente realizar consultas sobre cualquier tipo de intersecciones arbitrarias entre objetos de la escena.

En este ejemplo utilizaremos las intersecciones Rayo-Plano. Una vez creado el objeto de tipo *RaySceneQuery* en el constructor (mediante una llamada a *createRayQuery* del *SceneManager*), que posteriormente será eliminado en el destructor (mediante una llamada a *destroyQuery* del *SceneManager*), podemos utilizar el objeto para realizar consultas a la escena.

En la línea 20 se utiliza un método auxiliar para crear la Query, indicando las coordenadas del ratón. Empleando la función de utilidad de la línea 2, Ogre crea un rayo con origen en la cámara y la dirección indicada por las dos coordendas X, Y normalizadas (entre 0.0 y 1.0) que recibe como argumento (ver Figura 4.34). En la línea 4 se establece ese rayo para la consulta y se indica en la línea 5 que ordene los resultados por distancia de intersección. La consulta de tipo rayo devuelve una lista con todos los objetos que han intersecado con el

rayo[2]. Ordenando el resultado por distancia, tendremos como primer elemento el primer objeto con el que ha *chocado* el rayo. La ejecución de la consulta se realiza en la línea [21].

Para recuperar los datos de la consulta, se emplea un iterador. En el caso de esta aplicación, sólo nos interesa el primer elemento devuelto por la consulta (el primer punto de intersección), por lo que en el *if* de la línea [25] preguntamos si hay algún elemento devuelto por la consulta.

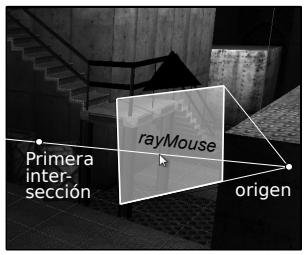


Figura 4.34: Empleando la llamada a *getCameraToViewportRay*, el usuario puede fácilmente obtener un rayo con origen en la posición de la cámara, y con la dirección definida por la posición del ratón.

La creación de las *Queries* son costosas computacionalmente (por lo que interesa realizarlas al inicio; en los constructores). Sin embargo su ejecución puede realizarse sin problema en el bucle principal de dibujado. El rendimiento se verá notablemente afectado si creas y destruyes la *query* en cada frame.

El iterador obtiene punteros a objetos de tipo *RaySceneQueryResultEntry*, que cuentan con 3 atributos públicos. El primero llamado distance es de tipo *Real* (usado en la línea [35]) nos indica la distancia de intersección desde el origen del rayo. El segundo atributo llamado *movable* contiene un puntero al MovableObject con el que intersecó (si existe). El tercer atributo *worldFragment* contiene un puntero al objeto de ese tipo (en este ejemplo no utilizamos geometría de esta clase).

El test de intersección rayo-objeto se realiza empleando cajas límite. Esto resulta muy eficiente (ya que únicamente hay que comparar con una caja y no con todos los polígonos que forman la entidad), pero tiene el inconveniente de la pérdida de precisión (Figura 4.35).

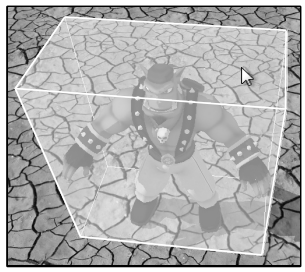


Figura 4.35: En el caso de objetos móviles, Ogre emplea una caja límite (definida por las coordenadas mayor y menor de los vértices del modelo) para calcular la intersección *rayo-objeto* y optimizar así los cálculos. Esto implica que, en el caso del test de intersección de la figura daría que hay colisión entre el rayo y el objeto. No obstante, es posible realizar *manualmente* un test de colisión con precisión (a nivel de polígono) si el juego lo requiere.

Finalmente, para añadir una caja en el punto de intersección del rayo con el suelo, tenemos que obtener la posición en el espacio 3D. Ogre proporciona una función de utilidad asociada a los rayos, que permite obtener el punto 3D, indicando una distancia desde el origen (línea [35]). De esta forma, indicando la distancia de intersección (obtenida en el iterador) en el método *getPoint* del rayo, obtenemos el *Vector3* que usamos directamente para posicionar el nuevo nodo en la escena.

[2]El rayo se describe por un punto de origen y una dirección. Desde ese origen, y siguiendo esa dirección, el rayo describe una línea infinita que podrá intersecar con *infinitos* objetos

Listado 4.31: Fragmento de MyFrameListener.cpp

```
1  Ray MyFrameListener::setRayQuery(int posx, int posy, uint32 mask) {
2    Ray rayMouse = _camera->getCameraToViewportRay
3      (posx/float(_win->getWidth()), posy/float(_win->getHeight()));
4    _raySceneQuery->setRay(rayMouse);
5    _raySceneQuery->setSortByDistance(true);
6    _raySceneQuery->setQueryMask(mask);
7    return (rayMouse);
8  }
9
10 bool MyFrameListener::frameStarted(const FrameEvent& evt) {
11   // ... Codigo anterior eliminado...
12   if (mbleft || mbright) {  // Boton izquierdo o derecho ------
13     if (mbleft) mask = STAGE | CUBE1 | CUBE2;  // Todos
14     if (mbright) mask = ~STAGE;   // Todo menos el escenario
15
16     if (_selectedNode != NULL) {  // Si hay alguno seleccionado...
17       _selectedNode->showBoundingBox(false);  _selectedNode = NULL;
18     }
19
20     Ray r = setRayQuery(posx, posy, mask);
21     RaySceneQueryResult &result = _raySceneQuery->execute();
22     RaySceneQueryResult::iterator it;
23     it = result.begin();
24
25     if (it != result.end()) {
26       if (mbleft) {
27           if (it->movable->getParentSceneNode()->getName() ==
28                                                    "Col_Suelo") {
29             SceneNode *nodeaux = _sceneManager->createSceneNode();
30             int i = rand()%2;   stringstream saux;
31             saux << "Cube" << i+1 << ".mesh";
32             Entity *entaux=_sceneManager->createEntity(saux.str());
33             entaux->setQueryFlags(i?CUBE1:CUBE2);
34             nodeaux->attachObject(entaux);
35             nodeaux->translate(r.getPoint(it->distance));
36             _sceneManager->getRootSceneNode()->addChild(nodeaux);
37           }
38       }
39       _selectedNode = it->movable->getParentSceneNode();
40       _selectedNode->showBoundingBox(true);
41     }
42   }
```

4.4.5. Máscaras

```
#define STAGE 1 << 0
#define CUBE1 1 << 1
#define CUBE2 1 << 2
```

Es equivalente a:

STAGE
0000000000000...00001
CUBE1
0000000000000...00010
CUBE2
0000000000000...00100

Figura 4.36: Máscaras binarias definidas en *MyFrameListener.h* para el ejemplo.

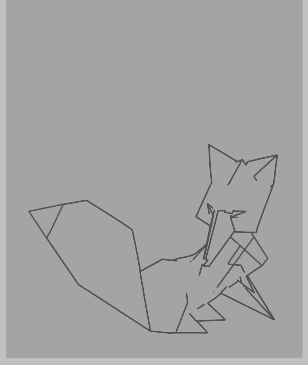
Las máscaras permiten restringir el ámbito de las consultas realizadas al gestor de escena. En el listado anterior, en las líneas 13 y 14 especificamos la máscara binaria que utilizaremos en la consulta (línea 6). Para que una máscara sea válida (y permita realizar operaciones lógicas con ella), debe contener un único uno.

Así, la forma más sencilla de definirlas es desplazando el 1 tantas posiciones como se indica tras el operador << como se indica en la Figura 4.36. Como el campo de máscara es de 32 bits, podemos definir 32 máscaras personalizadas para nuestra aplicación.

En el listado de la sección 4.4.4, vimos que mediante la llamada a `setQueryFlags` se podían asociar máscaras a entidades (líneas 5 y 19). Si no se especifica ninguna máscara en la creación de la entidad, ésta

responderá a *todas* las *Queries*, planteando así un esquema bastante flexible.

Los operadores lógicos pueden emplearse para combinar varias máscaras, como el uso del AND `&&`, el OR `||` o la negación ~ (ver líneas 13 y 14 del pasado código fuente). En este código fuente, la consulta a `~STAGE` sería equivalente a consultar por `CUBE1 | CUBE2`. La consulta de la línea 13 sería equivalente a preguntar por todos los objetos de la escena (no especificar ninguna máscara).



Capítulo 5

APIS de Gráficos 3D

Carlos González Morcillo

En este capítulo se introducirán los aspectos más relevantes de OpenGL. Como hemos señalado en el capítulo de introudcción, muchas bibliotecas de visualización (como OGRE) nos abstraen de los detalles de *bajo nivel*. Sin embargo, resulta interesante conocer el modelo de estados de este ipo de APIs, por compartir multitud de convenios con las bibliotecas de alto nivel. Entre otros aspectos, en este capítulo se estudiará la gestión de pilas de matrices y los diferentes modos de transformación de la API.

5.1. Introducción

Actualmente existen en el mercado dos alternativas principales como bibliotecas de representación 3D a bajo nivel: Direct3D y OpenGL. Estas dos bibliotecas son soportadas por la mayoría de los dispositivos hardware de aceleración gráfica. Direct3D forma parte del framework *DirectX* de *Microsoft*. Es una biblioteca ampliamente extendida, y multitud de motores 3D comerciales están basados en ella. La principal desventaja del uso de *DirectX* es la asociación exclusiva con el sistema operativo *Microsoft Windows*. Aunque existen formas de ejecutar un programa compilado para esta plataforma en otros sistemas, no es posible crear aplicaciones nativas utilizando *DirectX* en otros entornos. De este modo, en este primer estudio de las APIs de bajo nivel nos centraremos en la API multiplataforma OpenGL.

OpenGL es, probablemente, la biblioteca de programación gráfica más utilizada del mundo; desde videojuegos, simulación, CAD, visualización científica, y un largo etecétera de ámbitos de aplicación la configuran como la mejor alternativa en multitud de ocasiones. En esta sección se resumirán los aspectos básicos más relevantes para comenzar a utilizar OpenGL.

En este breve resumen de OpenGL se dejarán gran cantidad de aspectos sin mencionar. Se recomienda el estudio de la guía oficial de OpenGL [Shr09] (también conocido como *El Libro Rojo de OpenGL* para profundizar en esta potente biblioteca gráfica. Otra fantástica fuente de información, con ediciones anteriores del *Libro Rojo* es la página oficial de la biblioteca[1].

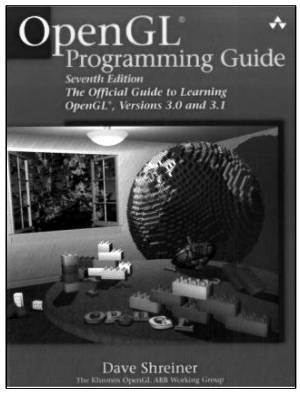


Figura 5.1: La última edición del libro oficial de OpenGL, la referencia imprescindible de más de 900 páginas.

El propio nombre *OpenGL* indica que es una *Biblioteca para Gráficos Abierta*[2]. Una de las características que ha hecho de OpenGL una biblioteca tan famosa es que es independiente de la plataforma sobre la que se está ejecutando (en términos software y hardware). Esto implica que *alguien* tendrá que encargarse de abrir una ventana gráfica sobre la que OpenGL pueda dibujar los preciosos gráficos 3D.

Para facilitar el desarrollo de aplicaciones con OpenGL sin preocuparse de los detalles específicos de cada sistema operativo (tales como la creación de ventanas, gestión de eventos, etc...) *M. Kilgard* creó GLUT, una biblioteca independiente de OpenGL (no forma parte de la distribución oficial de la API) que facilita el desarrollo de pequeños prototipos. Esta biblioteca auxiliar es igualmente multiplataforma. En este capítulo trabajaremos con FreeGLUT, la alternativa con licencia GPL totalmente compatible con GLUT. Si se desea mayor control sobre los eventos, y la posibilidad de extender las capacidades de la aplicación, se recomienda el estudio de otras APIs multiplataforma compatibles con OpenGL, como por ejemplo SDL[3] o la que emplearemos con OGRE (OIS). Existen alternativas específicas para sistemas de ventanas concretos (ver Figura 5.2, como *GLX* para plataformas Unix, *AGL* para Macintosh o *WGL* para sistemas Windows.

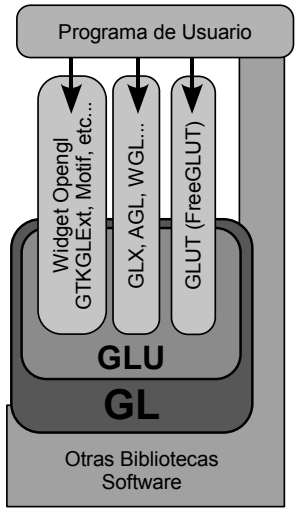


Figura 5.2: Relación entre los módulos principales de OpenGL.

De este modo, el núcleo principal de las biblioteca se encuentra en el módulo *GL* (ver Figura 5.2). En el módulo *GLU (OpenGL Utility Library* se encuentran funciones de uso común para el dibujo de diversos tipos de superficies (esferas, conos, cilindros, curvas...). Este módulo es parte oficial de la biblioteca.

Uno de los objetivos principales de OpenGL es la representación de imágenes de alta calidad a alta velocidad. OpenGL está diseñado para la realización de **aplicaciones interactivas**, como videojuegos. Gran cantidad de plataformas actuales se basan en esta especificación para definir sus interfaces de dibujado en gráficos 3D.

[1] http://www.opengl.org/
[2] Las siglas de *GL* corresponden a *Graphics Library*
[3] http://www.libsdl.org/

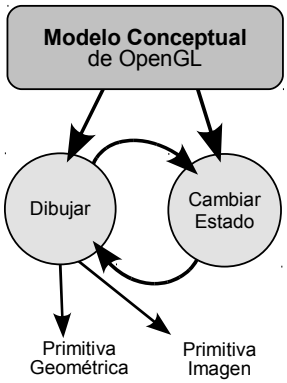


Figura 5.3: Modelo conceptual general de OpenGL.

5.2. Modelo Conceptual

Las llamadas a funciones de OpenGL están diseñadas para aceptar diversos tipos de datos como entrada. El nombre de la función identifica además los argumentos que recibirá. Por ejemplo, en la Figura 5.4 se llama a una función para especificar un nuevo vértice en coordenadas homogéneas (4 parámetros), con tipo de datos *double* y en formato vector. Se definen tipos enumerados como redefinición de tipos básicos (como *GLfloat*, *GLint*, etc) para facilitar la compatibilidad con otras plataformas. Es buena práctica utilizar estos tipos redefinidos si se planea compilar la aplicación en otros sistemas.

El **Modelo Conceptual General** de OpenGL define dos operaciones básicas que el programador puede realizar en cada instante; 1) dibujar algún elemento o 2) cambiar el estado de cómo se dibujan los elementos. La primera operación de *dibujar algún elemento* tiene que ser a) una primitiva geométrica (puntos, líneas o polígonos) o b) una primitiva de imagen.

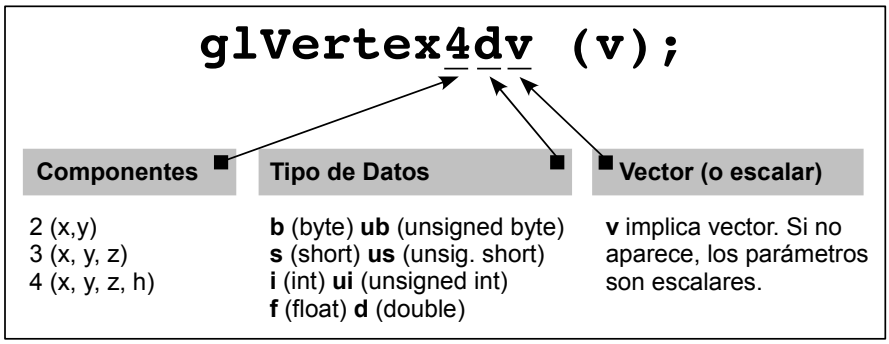


Figura 5.4: Prototipo general de llamada a función en OpenGL.

La Figura 5.3 resume el modelo conceptual de OpenGL. Así, la aplicación que utilice OpenGL será simplemente una colección de ciclos de cambio de estado y dibujado de elementos.

5.2.1. Cambio de Estado

Cambio de Estado

A modo de curiosidad: OpenGL cuenta con más de 400 llamadas a función que tienen que ver con el cambio del estado interno de la biblioteca.

La operación de **Cambiar el Estado** se encarga de inicializar las variables internas de OpenGL que definen cómo se dibujarán las primitivas. Este cambio de estado puede ser de mútiples tipos; desde cambiar el color de los vértices de la primitiva, establecer la posición de las luces, etc. Por ejemplo, cuando queremos dibujar el vértice de un polígono de color rojo, primero cambiamos el color del vértice con `glColor()` y después dibujamos la primitiva en ese nuevo estado con `glVertex()`.

Algunas de las formas más utilizadas para cambiar el estado de OpenGL son:

1. **Gestión de Vértices**: Algunas llamadas muy utilizadas cuando se trabaja con modelos poligonales son `glColor()` que utilizaremos en el primer ejemplo del capítulo para establecer el color con el que se dibujarán los vértices[4], `glNormal()` para especificar las normales que se utilizarán en la iluminación, o `glTexCoord()` para indicar coordendas de textura.

2. **Activación de Modos**: Mediante las llamadas a `glEnable` y `glDisable` se pueden activar o desactivar características internas de OpenGL. Por ejemplo, en la línea [3] del primer ejemplo del capítulo se activa el Test de Profundidad que utiliza y actualiza el Z-Buffer. Este test se utilizará hasta que de nuevo se cambia el *interruptor* desactivando esta funcionalidad en la línea [17].

3. **Características Especiales**: Existen multitud de características particulares de los elementos con los que se está trabajando. OpenGL define valores por defecto para los elementos con los que se está trabajando, que pueden ser cambiadas empleando llamadas a la API.

5.2.2. Dibujar Primitivas

La operación de **Dibujar Primitivas** requiere habitualmente que éstas se definan en coordenadas homogéneas. Todas las primitivas geométricas se especifican con vértices. El tipo de primitiva determina cómo se combinarán los vértices para formar la superfice poligional final. La creación de primitivas se realiza entre llamadas a `glBegin(PRIMITIVA)` y `glEnd()`, siendo `PRIMITIVA` alguna de las 10 primitivas básicas soportadas por OpenGL[5]. No obstante, el módulo GLU permite dibujar otras superficies más complejas (como cilindros, esferas, discos...), y con GLUT es posible dibujar algunos objetos simples. Como se puede ver, OpenGL no dispone de funciones para la carga de modelos poligonales creados con otras aplicaciones (como por ejemplo, en formato OBJ o MD3). Es responsabilidad del programador realizar esta carga y dibujarlos empleando las primitivas básicas anteriores.

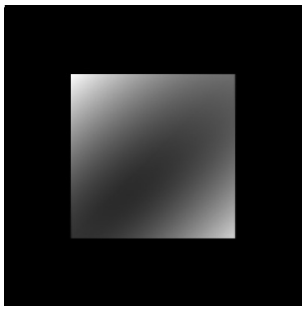


Figura 5.5: Salida por pantalla del ejemplo.

Una vez estudiados los cambios de estado y el dibujado de primitivas básicas, podemos especificar en la función `display` del siguiente listado para que dibuje un cuadrado (primitiva `GL_QUADS`). Las líneas relativas al despliegue del cuadrado se corresponden con el intervalo [7-17]. Especificamos la posición de cada vértice del cuadrado modificando el color (el estado interno). OpenGL se encargará de calcular la transición de color entre cada punto intermedio del cuadrado.

[4]Como veremos a continuación, los colores en OpenGL se especifican en punto flotante con valores entre 0 y 1. Las primeras tres componentes se corresponden con los canales *RGB*, y la cuarta es el valor de transparencia *Alpha*.

[5]Las 10 primitivas básicas de OpenGL son `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINES`, `GL_LINE_LOOP`, `GL_POLYGON`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLES`, `GL_TRIANGLE_FAN`, `GL_QUADS` y `GL_QUAD_STRIP`

Listado 5.1: Ejemplo de cambio de estado y dibujo de primitivas.

```
1  void display() {
2      glClear( GL_COLOR_BUFFER_BIT );
3      glEnable(GL_DEPTH_TEST);
4      glLoadIdentity();        /* Cargamos la matriz identidad */
5      glTranslatef( 0.f, 0.f, -4.f );
6
7      glBegin(GL_QUADS);           /* Dibujamos un cuadrado    */
8      glColor3f(1.0, 0.0, 0.0);    /* de dos unidades de lado. */
9      glVertex3f(1.0, 1.0, 0.0);   /* Especificamos la coorde- */
10     glColor3f(0.0, 1.0, 0.0);    /* nada 3D de cada vertice  */
11     glVertex3f(1.0, -1.0, 0.0);  /* y su color asociado.     */
12     glColor3f(0.0, 0.0, 1.0);
13     glVertex3f(-1.0, -1.0, 0.0);
14     glColor3f(1.0, 1.0, 1.0);
15     glVertex3f(-1.0, 1.0, 0.0);
16     glEnd();
17     glDisable(GL_DEPTH_TEST);
18
19     glutSwapBuffers();
20 }
```

5.3. Pipeline de OpenGL

Como vimos en el capítulo 1.2, los elementos de la escena sufren diferentes transformaciones en el *pipeline* de gráficos 3D. Algunas de las principales etapas se corresponden con las siguiente operaciones:

- **Transformación de Modelado**: En la que los modelos se posicionan en la escena y se obtienen las Coordenadas Universales.
- **Transformación de Visualización**: Donde se especifica la posición de la cámara y se mueven los objetos desde las coordenadas del mundo a las Coordenadas de Visualización (o coordenadas de cámara).
- **Transformación de Proyección**: Obteniendo Coordenadas Normalizadas en el cubo unitario.
- **Transformación de Recorte y de Pantalla**: Donde se obtienen, tras el recorte (o *clipping* de la geometría), las coordenadas 2D de la ventana en pantalla.

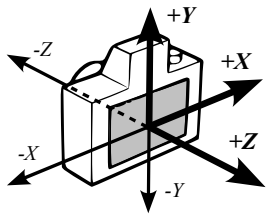


Figura 5.6: Descripción del sistema de coordenadas de la cámara definida en OpenGL.

OpenGL combina la Transformación de Modelado y la de Visualización en una Transformación llamada **"Modelview"**. De este modo OpenGL transforma directamente las Coordenadas Universales a Coordenadas de Visualización empleando la matriz *Modelview*. La posición inicial de la cámara en OpenGL sigue el convenio estudiado en el capítulo 2 y que se resume en la Figura 5.6.

5.3.1. Transformación de Visualización

La transformación de Visualización debe especificarse **antes** que ninguna otra transformación de Modelado. Esto es debido a que **Open-**

GL aplica las transformaciones en orden inverso. De este modo, aplicando en el código las transformaciones de Visualización antes que las de Modelado nos aseguramos que ocurrirán después que las de Modelado.

Para comenzar con la definición de la Transformación de Visualización, es necesario *limpiar* la matriz de trabajo actual. OpenGL cuenta con una función que carga la matriz identidad como matriz actual `glLoadIdentity()`.

Matriz Identidad

La *Matriz Identidad* es una matriz 4x4 con valor 1 en la diagonal principal, y 0 en el resto de elementos. La multiplicación de esta matriz I por una matriz M cualquiera siempre obtiene como resultado M. Esto es necesario ya que OpenGL siempre multiplica las matrices que se le indican para modificar su estado interno.

Una vez hecho esto, podemos posicionar la cámara virtual de varias formas:

1. **glulookat**. La primera opción, y la más comunmente utilizada es mediante la función `gluLookAt`. Esta función recibe como parámetros un punto (*eye*) y dos vectores (*center* y *up*). El punto *eye* es el punto donde se encuentra la cámara en el espacio y mediante los vectores libres *center* y *up* orientamos hacia dónde mira la cámara (ver Figura 5.7).
2. **Traslación y Rotación**. Otra opción es especificar *manualmente* una secuencia de traslaciones y rotaciones para posicionar la cámara (mediante las funciones `glTraslate()` y `glRotate()`, que serán estudiadas más adelante.
3. **Carga de Matriz**. La última opción, que es la utilizada en aplicaciones de Realidad Aumentada, es la carga de la matriz de Visualización calculada *externamente*. Esta opción se emplea habitualmente cuando el programador quiere calcular la posición de la cámara virtual empleando métodos externos (como por ejemplo, mediante una biblioteca de *tracking* para aplicaciones de Realidad Aumentada).

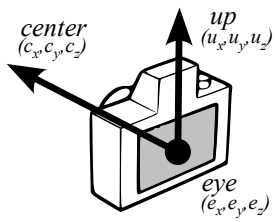


Figura 5.7: Parámetros de la función *glulookat*.

5.3.2. Transformación de Modelado

Las transformaciones de modelado nos permiten modificar los objetos de la escena. Existen tres operaciones básicas de modelado que implementa OpenGL con llamadas a funciones. No obstante, puede especificarse cualquier operación aplicando una matriz definida por el usuario.

- **Traslación**. El objeto se mueve a lo largo de un vector. Esta operación se realiza mediante la llamada a `glTranslate(x,y,z)`.
- **Rotación**. El objeto se rota en el eje definido por un vector. Esta operación se realiza mediante la llamada a `glRotate(`α`,x,y,z)`, siendo α el ángulo de rotación en grados sexagesimales (en sentido contrario a las agujas del reloj).
- **Escalado**. El objeto se escala un determinado valor en cada eje. Se realiza mediante `glScale(x,y,z)`.

5.3.3. Transformación de Proyección

Como hemos visto, las transformaciones de proyección definen el volumen de visualización y los planos de recorte. OpenGL soporta dos modelos básicos de proyección; la proyección ortográfica y la proyección en perspectiva.

El núcleo de OpenGL define una función para definir la pirámide de visualización (o *frustum*) mediante la llamada a `glFrustum()`. Esta función require los seis parámetros (t, b, r, l, f y n) estudiados en la sección 2.3.

Otro modo de especificar la transformación es mediante la función de GLU `gluPerspective(fov, aspect, near, far)`. En esta función, *far* y *near* son las distancias de los planos de recorte (igual que los parámetros f y n del *frustum*. *fov* especifica en grados sexagesimales el ángulo en el eje Y de la escena que es visible para el usuario, y *aspect* indica la relación de aspecto de la pantalla (ancho/alto).

Sobre el uso de Matrices

Como el convenio en C y C++ de definición de matrices bidimensionales es ordenados por filas, suele ser una fuente de errores habitual definir la matriz como un array bidimensional. Así, para acceder al elemento superior derecho de la matriz, tendríamos que acceder al `matriz[3][0]` según la notación OpenGL. Para evitar errores, se recomienda definir el array como unidimensional de 16 elementos `GLfloat matriz[16]`.

5.3.4. Matrices

OpenGL utiliza matrices 4x4 para representar todas sus transformaciones geométricas. Las matrices emplean coordenadas homogéneas, como se estudió en la sección 2.2.1. A diferencia de la notación matemática estándar, OpenGL especifica por defecto las matrices por columnas, por lo que si queremos cargar nuestras propias matrices de transformación debemos tener en cuenta que el orden de elementos en OpenGL es el siguiente:

$$\begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix} \tag{5.1}$$

OpenGL internamente maneja *pilas de matrices*, de forma que únicamente la matriz de la cima de cada pila es la que se está utilizando en un momento determinado. Hay cuatro pilas de matrices en OpenGL:

1. Pila *Modelview* (`GL_MODELVIEW`). Esta pila contiene las matrices de Transformación de modelado y visualización. Tiene un tamaño mínimo de 32 matrices (aunque, dependiendo del sistema puede haber más disponibles).

2. Pila *Projection* (`GL_PROJECTION`). Esta pila contiene las matrices de proyección. Tiene un tamaño mínimo de 2 elementos.

3. Pila *Color* (`GL_PROJECTION`). Utilizada para modificar los colores.

4. Pila *Texture* (`GL_TEXTURE`). Estas matrices se emplean para transformar las coordenadas de textura.

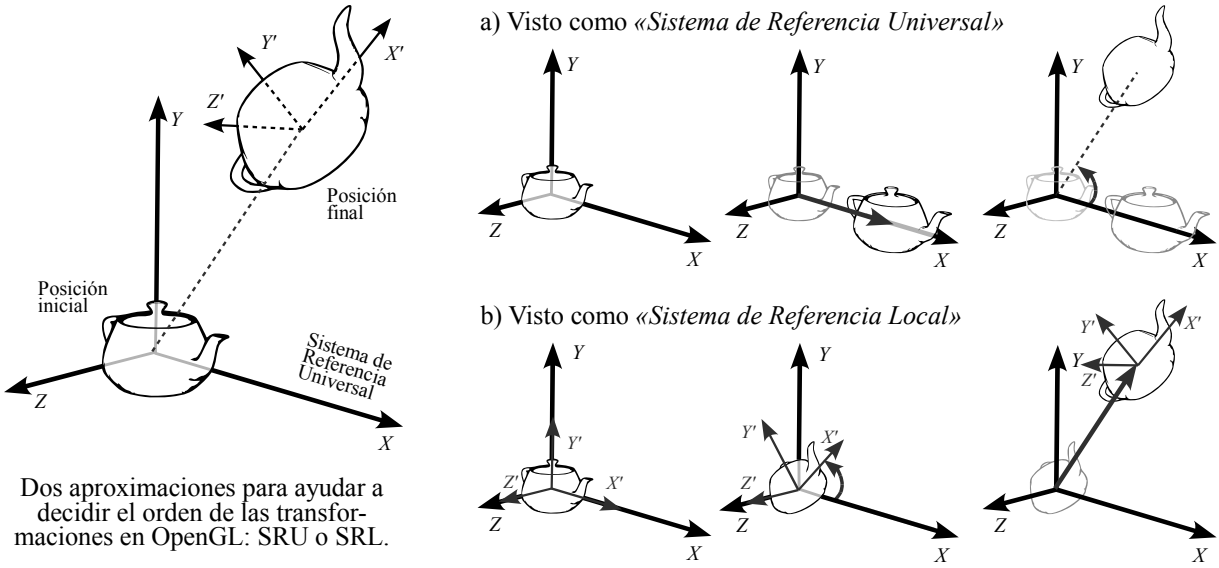


Figura 5.8: Cómo decidir el orden de las operaciones en OpenGL.

Es posible cambiar la pila sobre la que especificaremos las transformaciones empleando la llamada a `glMatrixMode()`. Por ejemplo, para utilizar la pila de *Modelview* utilizaríamos `glMatrixMode(GL_MODELVIEW)`.

El uso de *pilas* de matrices nos facilita la construcción de modelos jerárquicos, donde es posible utilizar modelos simples y ensamblarlos para construir modelos más complejos. Por ejemplo, si queremos dibujar un coche, tendríamos que dibujar las 4 ruedas con su posición relativa al chasis del vehículo. Así, dibujaríamos el chasis, y luego desplazándonos *cada vez desde el centro del coche* dibujaríamos cada una de las ruedas. Esta operación de *"volver al centro del coche"* se realiza fácilmente empleando pilas de matrices. Este concepto está directamente relacionado con el *Grafo de Escena* que estudiamos en el capítulo 3.

Recordemos que únicamente trabajamos con la matriz que está en la cima de la pila. La función `glPushMatrix()` añade una copia de la matriz de trabajo actual a la parte superior de la pila. Por su parte, la función `glPopMatrix()` elimina la matriz superior de la pila, descartando su información. La siguiente matriz de la pila pasará a ser la matriz *activa*. El efecto de esta función es el de "volver" al último punto que guardamos en la pila.

Supongamos ahora que queremos rotar un objeto 45º respecto del eje *Z* y trasladarlo 5 unidades respecto del eje X, obteniendo una determinada posición final (ver Figura 5.8 izquierda). El objeto inicialmente está en el origen del *SRU*. ¿En qué orden debemos aplicar las transformaciones de OpenGL? Como la transformación es de modelo, tendremos que aplicarla sobre la pila de matrices *Modelview*, con el siguiente código resultante:

Listado 5.2: Ejemplo de transformaciones.

```
1 glMatrixMode(GL_MODELVIEW);
2 glLoadIdentity();
3 glRotatef(45,0,0,1);
4 glTraslatef(5,0,0);
5 dibujar_objeto();
```

El código anterior dibuja el objeto en la posición deseada, pero ¿cómo hemos llegado a ese código y no hemos intercambiado las instrucciones de las líneas 3 y 4?. Existen dos formas de imaginarnos cómo se realiza el dibujado que nos puede ayudar a plantear el código fuente. Ambas formas son únicamente aproximaciones conceptuales, ya que el resultado en código debe ser exactamente el mismo.

- **Idea de Sistema de Referencia Universal Fijo**. La composición de movimientos aparece en orden inverso al que aparece en el código fuente. Esta idea es como ocurre realmente en OpenGL. Las transformaciones se aplican siempre respecto del *SRU*, y en orden inverso a como se indica en el código fuente. De este modo, la primera transformación que ocurrirá será la traslación (línea 4) y después la rotación *respecto del origen del sistema de referencia universal* (línea 3). El resultado puede verse en la secuencia de la Figura 5.8 a).
- **Idea del Sistema de Referencia Local**. También podemos imaginar que cada objeto tiene un sistema de referencia local *interno* al objeto que va cambiando. La composición se realiza en el mismo orden que aparece en el código fuente, y siempre respecto de ese sistema de referencia local. De esta forma, como se muestra en la secuencia de la Figura 5.8 b), el objeto primero rota (línea 3) por lo que su sistema de referencia local queda rotado respecto del *SRU*, y respecto de ese sistema de referencia local, posteriormente lo trasladamos 5 unidades respecto del eje X' (local).

Como hemos visto, es posible además cargar matrices de transformación definidas por nuestros propios métodos (podría ser interesante si empleáramos, por ejemplo, algún método para calcular la proyección de sombras). Esto se realiza con la llamada a función `glLoadMatrix()`. Cuando se llama a esta función se reemplaza la cima de la pila de matrices activa con el contenido de la matriz que se pasa como argumento.

Si nos interesa es multiplicar una matriz definida en nuestros métodos por el contenido de la cima de la pila de matrices, podemos utilizar la función `glMultMatrix` que postmultiplica la matriz que pasamos como argumento por la matriz de la cima de la pila.

Ejemplo Planetario

Ejemplo sencillo para afianzar el orden de las transformaciones.

5.3.5. Dos ejemplos de transformaciones jerárquicas

Veamos a continuación un ejemplo sencillo que utiliza transformaciones jerárquicas. Definiremos un sistema planetario que inicialmente estará formado por el Sol y la Tierra.

Listado 5.3: Sistema Planetario

```
1  // ==== Definicion de constantes y variables globales =============
2  long hours = 0;   // Horas transcurridas (para calculo rotaciones)
3  // ======== display ============================================
4  void display()
5  {
6    float RotEarthDay=0.0;  // Movimiento de rotacion de la tierra
7    float RotEarth=0.0;     // Movimiento de traslacion de la tierra
8    glClear( GL_COLOR_BUFFER_BIT );
9    glPushMatrix();
10
11   RotEarthDay = (hours % 24) * (360/24.0);
12   RotEarth = (hours / 24.0) * (360 / 365.0) * 10; // x10 rapido!
13
14   glColor3ub (255, 186, 0);
15   glutWireSphere (1, 16, 16);             // Sol (radio 1 y 16 div)
16   glRotatef (RotEarth, 0.0, 0.0, 1.0);
17   glTranslatef(3, 0.0, 0.0);              // Distancia Sol, Tierra
18   glRotatef (RotEarthDay, 0.0, 0.0, 1.0);
19   glColor3ub (0, 0, 255);
20   glutWireSphere (0.5, 8, 8);             // Tierra (radio 0.5)
21   glutSwapBuffers();
22   glPopMatrix();
23 }
```

Como puede verse en el listado anterior, se ha incluido una variable global `hours` que se incrementa cada vez que se llama a la función `display`. En este ejemplo, esta función se llama cada vez que se pulsa cualquier tecla. Esa variable modela el paso de las horas, de forma que la traslación y rotación de la *Tierra* se calculará a partir del número de horas que han pasado (líneas 11 y 12). En la simulación se ha acelerado 10 veces el movimiento de traslación para que se vea más claramente.

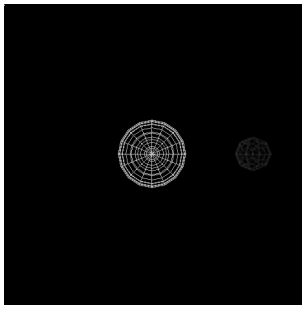


Figura 5.9: Salida por pantalla del ejemplo del planetario.

Empleando la *Idea de Sistema de Referencia Local* podemos pensar que desplazamos el sistema de referencia para dibujar el sol. En la línea 15, para dibujar una esfera alámbrica especificamos como primer argumento el radio, y a continuación el número de *rebanadas* (horizontales y verticales) en que se dibujará la esfera.

En ese punto dibujamos la primera esfera correspondiente al Sol. Hecho esto, rotamos los grados correspondientes al movimiento de traslación de la tierra (línea 16) y nos desplazamos 3 unidades respecto del eje X local del objeto (línea 17). Antes de dibujar la *Tierra* tendremos que realizar el movimiento de rotación de la tierra (línea 18). Finalmente dibujamos la esfera en la línea 20.

Ejemplo Brazo Robot

Un segundo ejemplo que utiliza las funciones de añadir y quitar elementos de la pila de matrices.

Veamos ahora un segundo ejemplo que utiliza `glPushMatrix()` y `glPopMatrix()` para dibujar un brazo robótico sencillo. El ejemplo simplemente emplea dos cubos (convenientemente escalados) para dibujar la estructura jerárquica de la figura 5.10.

En este ejemplo, se asocian manejadores de *callback* para los eventos de teclado, que se corresponden con la función `keyboard` (en 8–16). Esta función simplemente incrementa o decrementa los ángulos de rotación de las dos articulaciones del robot, que están definidas como variables globales en 2 y 3.

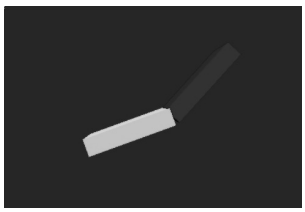


Figura 5.10: Salida por pantalla del ejemplo del robot.

La parte más *interesante* del ejemplo se encuentra definido en la función `dibujar` en las líneas 19–42.

Listado 5.4: Brazo Robótico

```
1  // ==== Definicion de constantes y variables globales =============
2  static int hombro = 15;
3  static int codo = 30;
4  GLfloat matVerde[] = {0.0, 1.0, 0.0, 1.0};
5  GLfloat matAzul[] = {0.0, 0.0, 1.0, 1.0};
6
7  // ==== Funcion de callback del teclado ============================
8  void teclado(unsigned char key, int x, int y) {
9    switch (key) {
10   case 'q': hombro = (hombro++) % 360; break;
11   case 'w': hombro = (hombro--) % 360; break;
12   case 'a': codo = (codo++) % 360; break;
13   case 's': codo = (codo--) % 360; break;
14   }
15   glutPostRedisplay();
16 }
17
18 // ==== Funcion de dibujado ========================================
19 void dibujar() {
20   glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
21   glLoadIdentity();
22   gluLookAt(0.0, 1.0, 6.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0);
23
24   glRotatef(hombro, 0.0, 0.0, 1.0);
25   glTranslatef(1.0, 0.0, 0.0);  // Nos posicionamos en la mitad
26   glPushMatrix();               // Guardamos la posicion
27   glScalef (2.0, 0.7, 0.1);     // Establecemos la escala
28   glMaterialfv(GL_FRONT, GL_DIFFUSE, matVerde);
29   glutSolidCube(1.0);           // Dibujamos el çubo"
30   glPopMatrix();                // Recuperamos la posicion
31
32   glTranslatef(1.0,0.0,0.0);    // Continuamos hasta el extremo
33   glRotatef(codo, 0.0, 0.0, 1.0);
34   glTranslatef(1.0,0.0,0.0);    // Nos posicionamos en la mitad
35   glPushMatrix();               // Guardamos la posicion
36   glScalef (2.0, 0.7, 0.1);     // Establecemos la .escala"
37   glMaterialfv(GL_FRONT, GL_DIFFUSE, matAzul);
38   glutSolidCube(1.0);           // Dibujamos el çubo"
39   glPopMatrix();
40
41   glutSwapBuffers();
42 }
```

Aplicamos de nuevo la idea de trabajar en un sistema de referencia local que se desplaza con los objetos según los vamos dibujando. De este modo nos posicionaremos en la mitad del trayecto para dibujar el cubo *escalado*. El cubo siempre se dibuja en el centro del sistema de referencia local (dejando mitad y mitad del cubo en el lado positivo y negativo de cada eje). Por tanto, para que el cubo *rote* respecto del extremo tenemos que rotar primero (como en la línea 24) y luego desplazarnos hasta la mitad del trayecto (línea 25), dibujar y recorrer la otra mitad 32 antes de dibujar la segunda parte del robot.

5.4. Ejercicios Propuestos

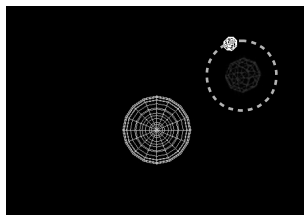


Figura 5.11: Ejemplo de salida del ejercicio propuesto.

Se recomienda la realización de los ejercicios de esta sección en orden, ya que están relacionados y su complejidad es ascendente.

1. Modifique el ejemplo del listado del planetario para que dibuje una esfera de color blanco que representará a la *Luna*, de radio 0.2 y separada 1 unidad del centro de la *Tierra* (ver Figura 5.11). Este objeto tendrá una rotación completa alrededor de la *Tierra* cada 2.7 días (10 veces más rápido que la rotación real de la *Luna* sobre la *Tierra*. Supondremos que la luna no cuenta con rotación interna[6].

2. Modifique el ejemplo del listado del brazo robótico para que una base (añadida con un toroide `glutSolidTorus`) permita rotar el brazo robótico respecto de la base (eje *Y*) mediante las teclas *1* y *2*. Además, añada el código para que el extremo cuente con unas pinzas (creadas con `glutSolidCone`) que se abran y cierren empleando las teclas *z* y *x*, tal y como se muestra en la Figura 5.12.

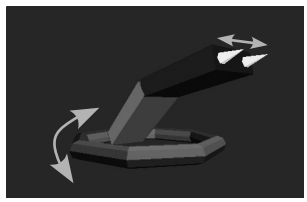
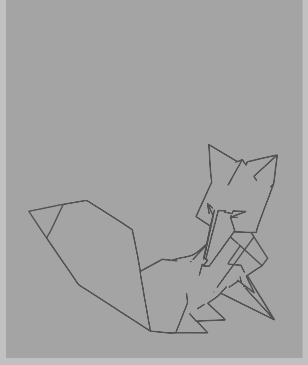


Figura 5.12: Ejemplo de salida del ejercicio del robot.

[6]Nota: Para la resolución de este ejercicio es recomendable utilizar las funciones de `glPushMatrix()` y `glPopMatrix()` para volver al punto donde se dibujó la *Tierra* antes de aplicar su rotación interna.



Capítulo 6

Gestión Manual OGRE 3D

Carlos González Morcillo

En las sesiones anteriores hemos delegado la gestión del arranque, incialización y parada de Ogre en una clase *SimpleExample* proporcionada junto con el motor gráfico para facilitar el desarrollo de los primeros ejemplos. En este capítulo estudiaremos la gestión semi-automática de la funcionalidad empleada en los ejemplos anteriores, así como introduciremos nuevas características, como la creación manual de entidades, y el uso de *Overlays*, luces y sombras dinámicas.

6.1. Inicialización Manual

Inicio Casi-Manual

En este capítulo no describiremos el inicio totalmente manual de Ogre; emplearemos las llamadas de alto nivel para cargar plugins, inicializar ventana, etc...

En esta sección introduciremos un esqueleto de código fuente que emplearemos en el resto del capítulo, modificándolo de manera incremental. Trabajaremos con dos clases; `MyApp` que proporciona el núcleo principal de la aplicación, y la clase `MyFrameListener` que es una isntancia de clase que hereda de `Ogre::FrameListener`, basada en el patrón *Observador*.

Como puede verse en el primer listado, en el fichero de cabecera de `MyApp.h` se declara la clase que contiene tres miembros privados; la instancia de *root* (que hemos estudiado en capítulos anteriores), el gestor de escenas y un puntero a un objeto de la clase propia *MySceneManager* que definiremos a continuación.

Listado 6.1: MyApp.h

```
1  #include <Ogre.h>
2  #include "MyFrameListener.h"
3
4  class MyApp {
5  private:
6    Ogre::SceneManager* _sceneManager;
7    Ogre::Root* _root;
8    MyFrameListener* _framelistener;
9
10 public:
11   MyApp();
12   ~MyApp();
13   int start();
14   void loadResources();
15   void createScene();
16 };
```

El programa principal únicamente tendrá que crear una instancia de la clase `MyApp` y ejecutar su método `start`. La definición de este método puede verse en el siguiente listado, en las líneas 13-44.

6.1.1. Inicialización

En la línea 14 se crea el objeto *Root*. Esta es la primera operación que se debe realizar antes de ejecutar cualquier otra operación con Ogre. El constructor de *Root* está sobrecargado, y permite que se le pase como parámetros el nombre del archivo de configuración de plugins (por defecto, buscará *plugins.cfg* en el mismo directorio donde se encuentra el ejecutable), el de configuración de vídeo (por defecto *ogre.cfg*), y de log (por defecto *ogre.log*). Si no le indicamos ningún parámetro (como en el caso de la línea 14), tratará de cargar los ficheros con el nombre por defecto.

El comportamiento del constructor de *Root* es diferente entre no especificar parámetro, o pasar la cadena vacía "". En el primer caso se buscarán los archivos con el nombre por defecto. En el segundo caso, indicamos a Ogre que cargaremos *manualmente* los plugins y la configuración de vídeo.

En la línea 16 se intenta cargar la configuración de vídeo existente en el archivo *ogre.cfg*. Si el archivo no existe, o no es correcto, podemos abrir el diálogo que empleamos en los ejemplos de las sesiones anteriores (línea 17), y guardar a continuacion los valores elegidos por el usuario (línea 18). Con el objeto *Root* creado, podemos pasar a crear la ventana. Para ello, en la línea 21 solicitamos al objeto *Root* que finalice su inicialización creando una ventana que utilice la configuración que eligió el usuario, con el título especificado como segundo parámetro (*MyApp* en este caso).

El primer parámetro booleano del método indica a Ogre que se encarge de crear automáticamente la ventana. La llamada a este método nos devuelve un puntero a un objeto *RenderWindow*, que guardaremos en la variable `window`.

Lo que será representado en la ventana vendrá definido por el volumen de visualización de la cámara virtual. Asociado a esta cámara, crearemos una *superficie* (algo que conceptualmente puede verse como el lienzo, o el *plano de imagen*) sobre la que se dibujarán los objetos 3D. Esta superficie se denomina *viewport*. El *Gestor de Escena* admite diversos modos de gestión, empleando el patrón de *Factoría*, que son cargados como plugins (y especificados en el archivo *plugins.cfg*). El parámetro indicado en la llamada de (22) especifica el tipo de gestor de escena que utilizaremos `ST_GENERIC`[1]. Esta gestión de escena mínima no está optimizada para ningún tipo de aplicación en particular, y resulta especialmente adecuada para pequeñas aplicaciones, menús de introducción, etc.

Listado 6.2: MyApp.cpp

```
1  #include "MyApp.h"
2
3  MyApp::MyApp() {
4    _sceneManager = NULL;
5    _framelistener = NULL;
6  }
7
8  MyApp::~MyApp() {
9    delete _root;
10   delete _framelistener;
11 }
12
13 int MyApp::start() {
14   _root = new Ogre::Root();       // Creamos el objeto root
15
16   if(!_root->restoreConfig()) {   // Si no se puede restaurar
17     _root->showConfigDialog();    // Abrimos ventana de config
18     _root->saveConfig();          // Guardamos la configuracion
19   }
20
21   Ogre::RenderWindow* window = _root->initialise(true,"MyApp");
22   _sceneManager = _root->createSceneManager(Ogre::ST_GENERIC);
23
24   Ogre::Camera* cam = _sceneManager->createCamera("MainCamera");
25   cam->setPosition(Ogre::Vector3(5,20,20));
26   cam->lookAt(Ogre::Vector3(0,0,0));
27   cam->setNearClipDistance(5);    // Establecemos distancia de
28   cam->setFarClipDistance(10000); // planos de recorte
29
30   Ogre::Viewport* viewport = window->addViewport(cam);
31   viewport->setBackgroundColour(Ogre::ColourValue(0.0,0.0,0.0));
32   double width = viewport->getActualWidth();
33   double height = viewport->getActualHeight();
34   cam->setAspectRatio(width / height);
35
36   loadResources();   // Metodo propio de carga de recursos
37   createScene();     // Metodo propio de creacion de la escena
38
```

[1] Existen otros métodos de gestión, como `ST_INTERIOR`, `ST_EXTERIOR`... Estudiaremos en detalle las opciones de estos gestores de escena a lo largo de este módulo.

```
39   _framelistener = new MyFrameListener();   // Clase propia
40   _root->addFrameListener(_framelistener);  // Lo anadimos!
41
42   _root->startRendering();    // Gestion del bucle principal
43   return 0;                   // delegada en OGRE
44 }
45
46 void MyApp::loadResources() {
47   Ogre::ConfigFile cf;
48   cf.load("resources.cfg");
49
50   Ogre::ConfigFile::SectionIterator sI = cf.getSectionIterator();
51   Ogre::String sectionstr, typestr, datastr;
52   while (sI.hasMoreElements()) {    // Mientras tenga elementos...
53     sectionstr = sI.peekNextKey();
54     Ogre::ConfigFile::SettingsMultiMap *settings = sI.getNext();
55     Ogre::ConfigFile::SettingsMultiMap::iterator i;
56     for (i = settings->begin(); i != settings->end(); ++i) {
57       typestr = i->first;    datastr = i->second;
58       Ogre::ResourceGroupManager::getSingleton().
             addResourceLocation(datastr, typestr, sectionstr);
59     }
60   }
61   Ogre::ResourceGroupManager::getSingleton().
         initialiseAllResourceGroups();
62 }
63
64 void MyApp::createScene() {
65   Ogre::Entity* ent = _sceneManager->createEntity("Sinbad.mesh");
66   _sceneManager->getRootSceneNode()->attachObject(ent);
67 }
```

El gestor de escena es el encargado de crear las cámaras virtuales. En realidad el gestor de escena trabaja como una factoría de diferentes tipos de objetos que crearemos en la escena. En la línea [24] obtenemos un puntero a una cámara que llamaremos *MainCamera*. A continuación establecemos la posición de la cámara virtual (línea [25]), y el punto hacia el que mira del SRU (línea [26]). En las líneas [27-28] establecemos la distancia con los planos de recorte cercano y lejano (ver Sección 2.3). En la línea [34] se establece la relación de aspecto de la cámara. Esta relación de aspecto se calcula como el número de píxeles en horizontal con respecto del número de píxeles en vertical. Resoluciones de 4/3 (como 800x600, o 1024x768) tendrán asociado un valor de ratio de 1.3333, mientras que resoluciones panorámicas de 16/9 tendrán un valor de 1.77, etc.

Para calcular el aspect ratio de la cámara, creamos un *viewport* asociado a esa cámara y establecemos el color de fondo como negro (líneas [30-31]).

Viewports...

Una cámara puede tener cero o más *Viewports*. Un uso común de los viewports es la generación de imágenes dentro de otros viewports. Veremos un uso de esta característica en próximos capítulos del módulo.

A continuación en las líneas [36-37] se ejecutan métodos propios de la clase para cargar los recursos y crear la escena. La carga de recursos manual la estudiaremos en la sección 6.1.2. La creación de la escena (líneas [64-67]), se realiza añadiendo la entidad directamente al nodo *Root*.

Las líneas [39-40] crean un objeto de tipo *FrameListener*. Una clase *FrameListener* es cualquier clase que implemente el interfaz de *FrameListener*, permitiendo que Ogre llame a ciertos métodos al inicio y al final del dibujado de cada frame empleando el patrón *Observer*. Ve-

remos en detalle cómo se puede implementar esta clase en la sección 6.1.3.

Al finalizar la inicialización, llamamos al método *startRendering* de *Root* (línea 42), que inicia el bucle principal de rendering. Esto hace que Ogre entre en un bucle infinito, ejecutando los métodos asociados a los *FrameListener* que hayan sido añadidos anteriormente.

En muchas ocasiones, no es conveniente delegar en Ogre la gestión del bucle principal de dibujado. Por ejemplo, si queremos atender peticiones de networking, o actualizar estados internos del juego. Además, si queremos integrar la ventana de visualización de Ogre como un widget dentro de un entorno de ventanas, es posible que éste no nos deje utilizar nuestro propio bucle pricipal de dibujado. En este caso, como veremos en sucesivos ejemplos, puede ser conveniente emplear la llamada a `renderOneFrame()` y gestionar manualmente el bucle de rendering.

6.1.2. Carga de Recursos

El fichero de definición de recursos se especifica en la línea 48. Este archivo está formado de diferentes secciones, que contienen pares de clave y valor. En el sencillo ejemplo que vamos a definir en este capítulo, tendremos el contenido definido en la Figura 6.1.

```
[General]
FileSystem=media
```

Figura 6.1: Contenido del archivo `resources.cfg` del ejemplo.

Como puede verse, la única sección del archivo se denomina *General*. Crearemos un iterador que permita cargar cada una de las secciones y, dentro de cada una, obtendremos los elementos que la definen.

La línea 50 crea el *SectionIterator* asociado al fichero de configuración. Mientras existan elementos que procesar por el iterador (bucle *while* de las líneas 52–60), obtiene la **clave** del primer elemento de la colección (línea 53) sin avanzar al siguiente (en el caso del ejemplo, obtiene la sección *General*). A continuación, obtiene en *settings* un puntero al **valor** del elemento actual de la colección, *avanzando al siguiente*. Este valor en realidad otro mapa. Emplearemos un segundo iterador (líneas 56–69) para recuperar los nombres y valores de cada entrada de la sección. En el fichero de ejemplo, únicamente iteraremos una vez dentro del segundo iterador, para obtener en `typestr` la entrada a `FileSystem`, y en `datastr` el valor del directorio `media`.

Finalmente la llamada al *ResourceGroupManager* (línea 61) solicita que se inicialicen todos los grupos de recursos que han sido añadidos en el iterador anterior, en la línea 58.

6.1.3. FrameListener

Como hemos comentado anteriormente, el uso de *FrameListener* se basa en el patrón *Observador*. Añadimos instancias de esta clase

al método Root de forma que será notificada cuando ocurran ciertos eventos. Antes de representar un frame, Ogre itera sobre todos los *FrameListener* añadidos, ejecutando el método *frameStarted* de cada uno de ellos. Cuando el frame ha sido dibujado, Ogre ejecutará igualmente el método *frameEnded* asociado a cada *FrameListener*. En el siguiente listado se declara la clase `MyFrameListener`.

FrameStarted

Si delegamos la gestión de bucle de dibujado a Ogre, la gestión de los eventos de teclado, ratón y joystick se realiza en *FrameStarted*, de modo que se atienden las peticiones *antes* del dibujado del frame.

Listado 6.3: MyFrameListener.h

```
1 #include <OgreFrameListener.h>
2
3 class MyFrameListener : public Ogre::FrameListener {
4 public:
5   bool frameStarted(const Ogre::FrameEvent& evt);
6   bool frameEnded(const Ogre::FrameEvent& evt);
7 };
```

Vemos que en esta sencilla implementación de la clase *MyFrameListener* no es necesario tener ninguna variable miembro específica, ni definir ningún constructor o destructor particular asociado a la clase. Veremos en las siguientes secciones cómo se complica la implementación de la misma según necesitamos añadir nueva funcionalidad.

La definición de la clase es igualmente sencilla. Los métodos *frameStarted* o *frameEnded* devolverán *false* cuando queramos finalizar la aplicación. En este ejemplo, tras llamar la primera vez a *frameStarted*, se enviará al terminal la cadena `Frame Started` y Ogre finalizará. De este modo, no se llegará a imprimir la cadena asociada a *frameEnded* (ni se representará el primer frame de la escena!). Si modificamos la implementación del método *frameStarted*, devolviendo *true*, se dibujará el primer frame y al ejecutar el método *frameEnded* (devolviendo *false*), Ogre finalizará el bucle principal de dibujado.

Habitualmente el método *frameEnded* se define pocas veces. Únicamente si necesitamos liberar memoria, o actualizar algún valor tras dibujar el frame, se realizará una implementación específica de este método. En los próximos ejemplos, únicamente definiremos el método *frameStarted*.

Listado 6.4: MyFrameListener.cpp

```
1  #include "MyFrameListener.h"
2
3  bool MyFrameListener::frameStarted(const Ogre::FrameEvent& evt) {
4    std::cout << "Frame started" << std::endl;
5    return false;
6  }
7
8  bool MyFrameListener::frameEnded(const Ogre::FrameEvent& evt)  {
9    std::cout << "Frame ended" << std::endl;
10   return false;
11 }
```

6.2. Uso de OIS

Sobre OIS

Recordemos que OIS *Object Oriented Input System* no forma parte de la distribución de Ogre. Es posible utilizar cualquier biblioteca para la gestión de eventos.

A continuación utilizaremos OIS para añadir un soporte de teclado mínimo, de forma que cuando se pulse la tecla [ESC], se cierre la aplicación. Bastará con devolver *false* en el método *frameStarted* del *FrameListener* cuando esto ocurra. Como vemos en el siguiente listado, es necesario añadir al constructor de la clase (línea [11]) un puntero a la *RenderWindow*.

Listado 6.5: MyFrameListener.h

```
1  #include <OgreFrameListener.h>
2  #include <OgreRenderWindow.h>
3  #include <OIS/OIS.h>
4
5  class MyFrameListener : public Ogre::FrameListener {
6  private:
7    OIS::InputManager* _inputManager;
8    OIS::Keyboard* _keyboard;
9
10 public:
11   MyFrameListener(Ogre::RenderWindow* win);
12   ~MyFrameListener();
13   bool frameStarted(const Ogre::FrameEvent& evt);
14 };
```

Esta ventana se necesita para obtener el *manejador* de la ventana. Este manejador es un identificador único que mantiene el sistema operativo para cada ventana. Cada sistema operativo mantiene una lista de atributos diferente. Ogre abstrae del sistema operativo subyacente empleando una función genérica, que admite como primer parámetro el tipo de elemento que queremos obtener, y como segundo un puntero al tipo de datos que vamos a recibir. En el caso de la llamada de la línea [7], obtenemos el manejador de la ventana (el tipo asociado al manejador se especifica mediante la cadena `WINDOW`).

En las líneas [8–9] convertimos el manejador a cadena, y añadimos el par de objetos (empleando la plantilla *pair* de std) como parámetros de OIS. Como puede verse el convenio de Ogre y de OIS es similar a la hora de nombrar el manejador de la ventana (salvo por el hecho de que OIS requiere que se especifique como cadena, y Ogre lo devuelve como un entero).

Con este parámetro, creamos un *InputManager* de OIS en [11], que nos permitirá crear diferentes interfaces para trabajar con eventos de teclado, ratón, etc. Así, en las líneas [12–13] creamos un objeto de tipo *OIS::Keyboard*, que nos permitirá obtener las pulsaciones de tecla del usuario. El segundo parámetro de la línea [13] permite indicarle a OIS si queremos que almacene en un buffer los eventos de ese objeto. En este caso, la entrada por teclado se consume directamente sin almacenarla en un buffer intermedio.

Tanto el *InputManager* como el ojbeto de teclado deben ser eliminados explícitamente en el destructor de nuestro *FrameListener*. Ogre se encarga de liberar los recursos asociados a todos sus *Gestores*. Como Ogre es independiente de OIS, no tiene constancia de su uso y

es responsabilidad del programador liberar los objetos de entrada de eventos, así como el propio gestor *InputManager*.

Listado 6.6: MyFrameListener.cpp

```
1  #include "MyFrameListener.h"
2
3  MyFrameListener::MyFrameListener(Ogre::RenderWindow* win) {
4    OIS::ParamList param;
5    unsigned int windowHandle;  std::ostringstream wHandleStr;
6
7    win->getCustomAttribute("WINDOW", &windowHandle);
8    wHandleStr << windowHandle;
9    param.insert(std::make_pair("WINDOW", wHandleStr.str()));
10
11   _inputManager = OIS::InputManager::createInputSystem(param);
12   _keyboard = static_cast<OIS::Keyboard*>
13     (_inputManager->createInputObject(OIS::OISKeyboard, false));
14 }
15
16 MyFrameListener::~MyFrameListener() {
17   _inputManager->destroyInputObject(_keyboard);
18   OIS::InputManager::destroyInputSystem(_inputManager);
19 }
20
21 bool MyFrameListener::frameStarted(const Ogre::FrameEvent& evt) {
22   _keyboard->capture();
23   if(_keyboard->isKeyDown(OIS::KC_ESCAPE)) return false;
24   return true;
25 }
```

La implementación del método *frameStarted* es muy sencilla. En la línea [22] se obtiene si hubo alguna pulsación de tecla. Si se presionó la tecla Escape (línea [23]), se devuelve *false* de modo que Ogre finalizará el bucle principal de dibujado y liberará todos los recursos que se están empleando.

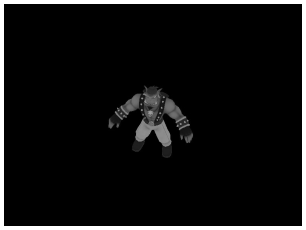


Figura 6.2: Resultado de ejecución del ejemplo básico con OIS.

Hasta ahora, la escena es totalmente estática. La Figura 6.2 muestra el resultado de ejecutar el ejemplo (hasta que el usuario presiona la tecla [ESC]). A continuación veremos cómo modificar la posición de los elementos de la escena, definiendo un interfaz para el manejo del ratón.

6.2.1. Uso de Teclado y Ratón

En el ejemplo de esta sección desplazaremos la cámara y rotaremos el modelo empleando el teclado y el ratón. Como en el diseño actual de nuestra apliación el despliegue se gestiona íntegramente en la clase *MyFrameListener*, será necesario que esta clase *conozca* los nuevos objetos sobre los que va a trabajar: el ratón (línea [8]), la cámara (declarada igualmente como variable miembro privada en [9]), y el nodo (que contendrá las entidades que queremos mover en [10]).

En estos primeros ejemplos nos centraremos en el aspecto funcional de los mismos. En sucesivos capítulos estudiaremos otras aproximaciones de diseño para la gestión de eventos.

Listado 6.7: MyFrameListener.h

```
#include <Ogre.h>
#include <OIS/OIS.h>

class MyFrameListener : public Ogre::FrameListener {
private:
  OIS::InputManager* _inputManager;
  OIS::Keyboard* _keyboard;
  OIS::Mouse* _mouse;
  Ogre::Camera* _camera;
  Ogre::SceneNode *_node;

public:
  MyFrameListener(Ogre::RenderWindow* win, Ogre::Camera* cam,
          Ogre::SceneNode* node);
  ~MyFrameListener();
  bool frameStarted(const Ogre::FrameEvent& evt);
};
```

Listado 6.8: MyFrameListener.cpp

```
#include "MyFrameListener.h"

MyFrameListener::MyFrameListener(Ogre::RenderWindow* win,
     Ogre::Camera* cam,  Ogre::SceneNode *node) {
  OIS::ParamList param;
  unsigned int windowHandle;  std::ostringstream wHandleStr;

  _camera = cam;  _node = node;

  win->getCustomAttribute("WINDOW", &windowHandle);
  wHandleStr << windowHandle;
  param.insert(std::make_pair("WINDOW", wHandleStr.str()));

  _inputManager = OIS::InputManager::createInputSystem(param);
  _keyboard = static_cast<OIS::Keyboard*>
    (_inputManager->createInputObject(OIS::OISKeyboard, false));
  _mouse = static_cast<OIS::Mouse*>
    (_inputManager->createInputObject(OIS::OISMouse, false));
}

MyFrameListener::~MyFrameListener() {
  _inputManager->destroyInputObject(_keyboard);
  _inputManager->destroyInputObject(_mouse);
  OIS::InputManager::destroyInputSystem(_inputManager);
}

bool MyFrameListener::frameStarted(const Ogre::FrameEvent& evt) {
  Ogre::Vector3 vt(0,0,0);     Ogre::Real tSpeed = 20.0;
  Ogre::Real r = 0;
  Ogre::Real deltaT = evt.timeSinceLastFrame;

  _keyboard->capture();
  if(_keyboard->isKeyDown(OIS::KC_ESCAPE)) return false;
  if(_keyboard->isKeyDown(OIS::KC_UP))   vt+=Ogre::Vector3(0,0,-1);
  if(_keyboard->isKeyDown(OIS::KC_DOWN)) vt+=Ogre::Vector3(0,0,1);
  if(_keyboard->isKeyDown(OIS::KC_LEFT)) vt+=Ogre::Vector3(-1,0,0);
  if(_keyboard->isKeyDown(OIS::KC_RIGHT))vt+=Ogre::Vector3(1,0,0);
  _camera->moveRelative(vt * deltaT * tSpeed);

  if(_keyboard->isKeyDown(OIS::KC_R)) r+=180;
  _node->yaw(Ogre::Degree(r * deltaT));

  _mouse->capture();
```

```
44    float rotx = _mouse->getMouseState().X.rel * deltaT * -1;
45    float roty = _mouse->getMouseState().Y.rel * deltaT * -1;
46    _camera->yaw(Ogre::Radian(rotx));
47    _camera->pitch(Ogre::Radian(roty));
48
49    return true;
50  }
```

La implementación de la clase *MyFrameListener* requiere asignar en el constructor los punteros de la cámara y del nodo de escena a las variables miembro privadas (línea 8). De forma análoga, crearemos un *InputObject* para el ratón en las líneas 17-18 (que eliminaremos en el destructor en 23).

El método *frameStarted* es algo más complejo que el estudiado en el ejemplo anterior. En esta ocasión, se definen en las líneas 28-30 una serie de variables que comentaremos a continuación. En `vt` almacenaremos el vector de traslación relativo que aplicaremos a la cámara, dependiendo de la pulsación de teclas del usuario. La variable `r` almacenará la rotación que aplicaremos al nodo de la escena (pasado como tercer argumento al constructor). En `deltaT` guardaremos el número de segundos transcurridos desde el despliegue del último frame. Finalmente la variable `tSpeed` servirá para indicar la traslación (distancia en unidades del mundo) que queremos recorrer con la cámara en un segundo.

La necesidad de medir el tiempo transcurrido desde el despliegue del último frame es imprescindible si queremos que las unidades del espacio recorridas por el modelo (o la cámara en este caso) sean dependientes del tiempo e independientes de las capacidades de representación gráficas de la máquina sobre la que se están ejecutando. Si aplicamos directamente un incremento del espacio sin tener en cuenta el tiempo, como se muestra en la Figura 6.3, el resultado del espacio recorrido dependerá de la velocidad de despliegue (ordenadores más potentes avanzarán más espacio). La solución es aplicar incrementos en la distancia dependientes del tiempo.

De este modo, aplicamos un movimiento relativo a la cámara (en función de sus ejes locales), multiplicando el vector de traslación (que ha sido definido dependiendo de la pulsación de teclas del usuario en las líneas 34-37), por `deltaT` y por la velocidad de traslación (de modo que avanzará 20 unidades por segundo).

De forma similar, cuando el usuario pulse la tecla R, se rotará el modelo respecto de su eje Y local a una velocidad de 180º por segundo (líneas 40-41).

Finalmente, se aplicará una rotación a la cámara respecto de sus ejes Y y Z locales empelando el movimiento relativo del ratón. Para ello, se obtiene el estado del ratón (líneas 44-45), obteniendo el incremento relativo en píxeles desde la última captura (mediante el atributo `abs` se puede obtener el valor absoluto del incremento).

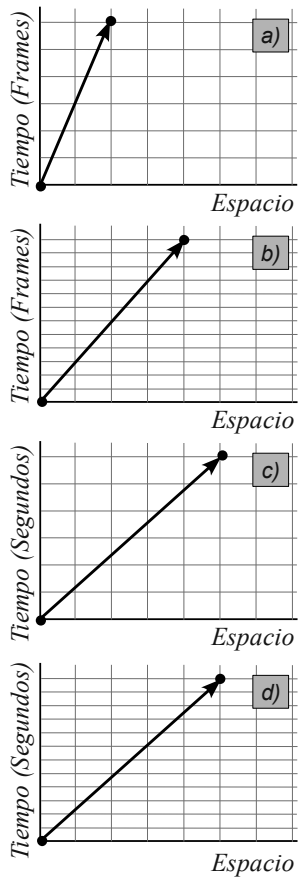


Figura 6.3: Comparación entre animación basada en frames y animación basada en tiempo. El eje de abcisas representa el espacio recorrido en una trayectoria, mientras que el eje de ordenadas representa el tiempo. En las gráficas a) y b) el tiempo se especifica en frames. En las gráficas c) y d) el tiempo se especifica en segundos. Las gráficas a) y c) se corresponden con los rsultados obtenidos en un computador con bajo rendimiento gráfico (en el mismo tiempo presenta una baja tasa de frames por segundo). Las gráficas b) y d) se corresponden a un computador con el doble de frames por segundo.

6.3. Creación manual de Entidades

Ogre soporta la creación manual de multitud de entidades y objetos. Veremos a continuación cómo se pueden añadir planos y fuentes de luz a la escena.

La creación manual del plano se realiza en las líneas [8-11] del siguiente listado. En la línea [8] se especifica que el plano tendrá como vector normal, el vector unitario en Y, y que estará situado a -5 unidades respecto del vector normal. Esta definición se corresponde con un plano infinito (descripción matemática abstracta). Si queremos representar el plano, tendremos que indicar a Ogre el tamaño (finito) del mismo, así como la resolución que queremos aplicarle (número de divisiones horizontales y verticales). Estas operaciones se indican en las líneas [9-11].

Listado 6.9: Definición de createScene (MyApp.cpp)

```
1  void MyApp::createScene() {
2    Ogre::Entity* ent1 = _sceneManager->createEntity("Sinbad.mesh");
3    Ogre::SceneNode* node1 =
4          _sceneManager->createSceneNode("SinbadNode");
5    node1->attachObject(ent1);
6    _sceneManager->getRootSceneNode()->addChild(node1);
7
8    Ogre::Plane pl1(Ogre::Vector3::UNIT_Y, -5);
9    Ogre::MeshManager::getSingleton().createPlane("pl1",
10     Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
11     pl1,200,200,1,1,true,1,20,20,Ogre::Vector3::UNIT_Z);
12
13   Ogre::SceneNode* node2 = _sceneManager->createSceneNode("node2");
14   Ogre::Entity* grEnt = _sceneManager->createEntity("pEnt", "pl1");
15   grEnt->setMaterialName("Ground");
16   node2->attachObject(grEnt);
17
18   _sceneManager->setShadowTechnique(Ogre::
         SHADOWTYPE_STENCIL_ADDITIVE);
19   Ogre::Light* light = _sceneManager->createLight("Light1");
20   light->setType(Ogre::Light::LT_DIRECTIONAL);
21   light->setDirection(Ogre::Vector3(1,-1,0));
22   node2->attachObject(light);
23
24   _sceneManager->getRootSceneNode()->addChild(node2);
25 }
```

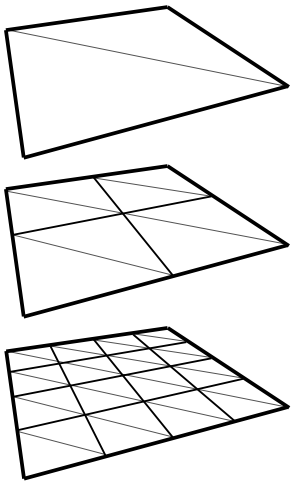


Figura 6.4: Número de segmentos de definición del plano. La imagen superior implica un segmento en X y un segmento en Y (valores por defecto). La imagen central se corresponde con una definición de 2 segmentos en ambos planos, y la inferior de 4 segmentos.

El uso del *MeshManager* nos permite la creación de planos, skyboxes, superficies de Bezier, y un largo etcétera. En el caso de este ejemplo, se utiliza la llamada a `createPlane` que recibe los siguientes parámetros indicados en orden: el primer parámetro es el nombre de la malla resultante, a continuación el nombre del grupo de mallas (emplearemos el grupo por defecto). El tercer parámetro es el objeto definición del plano infinito (creado en la línea [8]). Los siguientes dos parámetros indican el ancho y alto del plano en coordenadas del mundo (200x200 en este caso). Los siguientes dos parámetros se corresponden con el número de segmentos empleados para definir el plano (1x1 en este caso), como se indica en la Figura 6.4. Estos parámetros sirven para especificar la resolución geométrica del plano creado (por

si posteriormente queremos realizar operaciones de distorsión a nivel de vértice, por ejemplo).

El siguiente parámetro booleano indica (si es cierto) que los vectores normales se calcularán perpendiculares al plano. El siguiente parámetro (por defecto, 1) indica el conjunto de coordenadas de textura que serán creadas. A continuación se indica el número de repetición (*uTile* y *vTile*) de la textura (ver Figura 6.6). El último parámetro indica la dirección del vector *Up* del plano.

A continuación en la línea [15] asignamos al plano el material "*Ground*". Este material está definido en un archivo en el directorio `media` que ha sido cargado empleando el cargador de recursos explicado en la sección 6.1.2. Este material permite la recepción de sombras, tiene componente de brillo difuso y especular, y una textura de imagen basada en el archivo *ground.jpg*. Estudiaremos en detalle la asignación de materiales en el próximo capítulo del documento.

Finalmente las líneas [18-22] se encargan de definir una fuente de luz dinámica direccional [2] y habilitar el cálculo de sombras. Ogre soporta 3 tipos básicos de fuentes de luz y 11 modos de cálculo de sombras dinámicas, que serán estudiados en detalle en próximos capítulos.

```
material Ground
{
 receive_shadows on
 technique
 {
   pass
   {
     ambient 0.7 0.7 0.7
     diffuse 1 1 1 1
     texture_unit
     {
       texture
         ground.jpg
     }
   }
 }
}
```

Figura 6.5: Fichero de definición del material `Ground.material`.

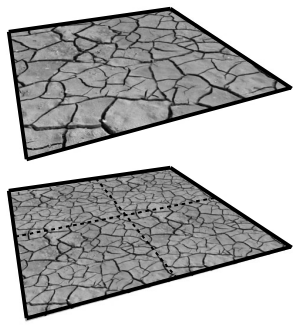


Figura 6.6: Valores de repetición en la textura. La imagen superior se corresponde con valores de $uTile = 1, vTile = 1$. La imagen inferior implica un valor de 2 en ambos parámetros (repitiendo la textura 2x2 veces en el plano).

6.4. Uso de Overlays

En el último ejemplo del capítulo definiremos superposiciones (*Overlays*) para desplegar en 2D elementos de información (tales como marcadores, botones, logotipos, etc...).

La gestión de los *Overlays* en Ogre se realiza mediante el gestor de recursos llamado *OverlayManager*, que se encarga de cargar los scripts de definición de los *Overlays*. En un overlay pueden existir objetos de dos tipos: los **contenedores** y los **elementos**. Los elementos pueden ser de tres tipos básicos: *TextArea* empleado para incluir texto, *Panel* empleado como elemento que puede agrupar otros elementos con un fondo fijo y *BorderPanel* que es igual que *Panel* pero permite que la textura de fondo se repita y definir un borde independiente del fondo. La definición del *Overlay* del ejemplo de esta sección se muestra en la Figura 6.8. En el caso de utilizar un *TextArea*, será necesario especificar la carga de una fuente de texto. Ogre soporta texturas basadas en imagen o truetype. La definición de las fuentes se realiza empleando un fichero de extensión `.fontdef` (ver Figura 6.7).

En el listado de la Figura 6.8 se muestra la definición de los elementos y contendores del Overlay, así como los ficheros de material (extensión `.material`) asociados al mismo. En el siguiente capítulo estudiaremos en detalle la definición de materiales en Ogre, aunque un simple vistazo al fichero de materiales nos adelanta que estamos creando dos materiales, uno llamado *panelInfoM*, y otro *matUCLM*. Ambos utilizan texturas de imagen. El primero de ellos utiliza una técnica de mezclado que tiene en cuenta la componente de opacidad de la pasada.

```
Blue
{
  type truetype
  source Blue.ttf
  size 25
  resolution 96
}
```

Figura 6.7: Fichero de definición de la fuente `Blue.fontdef`.

[2] Este tipo de fuentes de luz únicamente tienen dirección. Definen rayos de luz paralelos de una fuente distante. Es el equivalente al tipo de fuente *Sun* de Blender.

info.overlay

```
template element TextArea(MyTemplates/Text)
{
  font_name Blue
  metrics_mode pixels
  char_height 15
  colour 1.0 1.0 1.0
}

template element TextArea(MyTemplates/SmallText)
{
  font_name Blue
  metrics_mode pixels
  char_height 12
  colour 1.0 1.0 1.0
}
```

panel.material

```
material panelInfoM
{
 technique
 {
   pass
   {
     scene_blend src_alpha one_minus_src_alpha
     texture_unit
     {
       texture panel.tga
     }
   }
 }
}

material matUCLM
{
 technique
 {
   pass
   {
     texture_unit
     {
       texture logouclm.jpg
     }
   }
 }
}
```

```
Info
{
  zorder 500
  container Panel(panelInfo)
  {
    metrics_mode pixels
    left 10
    top -140
    width 280
    height 130
    vert_align bottom
    material panelInfoM

    element TextArea(fpsInfo) : MyTemplates/Text
    {
      top 35
      left 180
    }

    element TextArea(camPosInfo) : MyTemplates/Text
    {
      top 65
      left 120
    }

    element TextArea(camRotInfo) : MyTemplates/SmallText
    {
      top 80
      left 120
    }

    element TextArea(modRotInfo) : MyTemplates/Text
    {
      top 105
      left 120
    }
  }

  container Panel(logoUCLM)
  {
    metrics_mode pixels
    left -180
    top 0
    width 150
    height 120
    vert_align top
    horz_align right
    material matUCLM
  }
}
```

Figura 6.8: Definición de los Overlays empleando ficheros externos. En este fichero se describen dos paneles contenedores; uno para la zona inferior izquierda que contiene cuatro áreas de texto, y uno para la zona superior derecha que representará el logotipo de la UCLM.

Atributo	**Descripción**
metrics_mode	Puede ser *relative* (valor entre 0.0 y 1.0), o *pixels* (valor en píxeles). Por defecto es *relative*. En coordenadas relativas, la coordenada superior izquierda es la (0,0) y la inferior derecha la (1,1).
horz_align	Puede ser *left* (por defecto), *center* o *right*.
vert_align	Puede ser *top* (por defecto), *center* o *bottom*.
left	Posición con respecto al extremo izquierdo. Por defecto 0. Si el valor es negativo, se interpreta como espacio desde el extremo derecho (con alineación *right*).
top	Posición con respecto al extremo superior. Por defecto 0. Si el valor es negativo, se interpreta como espacio desde el extremo inferior (con alineación vertical *bottom*).
width	Ancho. Por defecto 1 (en *relative*).
height	Alto. Por defecto 1 (en *relative*).
material	Nombre del material asociado (por defecto Ninguno).
caption	Etiqueta de texto asociada al elemento (por defecto Ninguna).
rotation	Ángulo de rotación del elemento (por defecto sin rotación).

Tabla 6.1: Atributos generales de *Elementos* y *Contenedores* de Overlays.

El primer contenedor de tipo Panel (llamado *PanelInfo*), contiene cuatro TextArea en su interior. El posicionamiento de estos elementos se realiza de forma relativa al posicionamiento del contenedor. De este modo, se establece una relación de jerarquía implícita entre el contenedor y los objetos contenidos. El segundo elemento contenedor (llamado *logoUCLM*) no tiene ningún elemento asociado en su interior.

La Tabla 6.1 describe los principales atributos de los contenedores y elementos de los overlays. Estas propiedades pueden ser igualmente modificadas en tiempo de ejecución, por lo que son *animables*. Por ejemplo, es posible modificar la rotación de un elemento del *Overlay* consiguiendo bonitos efectos en los menús del juego.

El sistema de scripting de Overlays de Ogre permite definir plantillas de estilo que pueden utilizar los elementos y contenedores. En este ejemplo se han definido dos plantillas, una para texto genérico (de nombre *MyTemplates/Text*), y otra para texto pequeño (*MyTemplates/SmallText)*. Para aplicar estas plantillas a un elemento del *Overlay*, basta con indicar seguido de dos puntos el nombre de la plantilla a continuación del elemento (ver Figura 6.8).

En el listado del Overlay definido, se ha trabajado con el modo de especificación del tamaño en píxeles. La alineación de los elementos se realiza a nivel de píxel (en lugar de ser relativo a la resolución de la pantalla), obteniendo un resultado como se muestra en la Figura 6.8. El uso de valores negativos en los campos de alineación permite posicionar con exactitud los paneles alineados a la derecha y en el borde inferior. Por ejemplo, en el caso del panel *logoUCLM*, de tamaño (150x120), la alineación horizontal se realiza a la derecha. El valor de -180 en el campo *left* indica que queremos que quede un margen de 30 píxeles entre el lado derecho de la ventana y el extremo del logo.

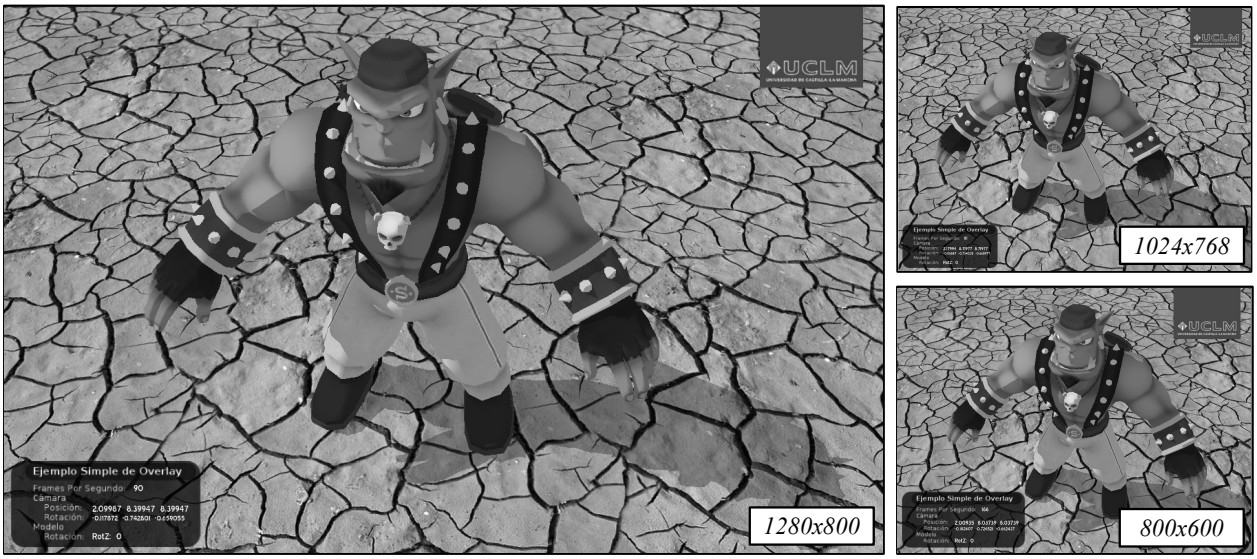


Figura 6.9: Resultado de ejecución del ejemplo de uso de Overlays, combinado con los resultados parciales de las secciones anteriores. La imagen ha sido generada con 3 configuraciones de resolución diferentes (incluso con diferente relación de aspecto): 1280x800, 1024x768 y 800x600.

Los *Overlays* tienen asociada una profundidad, definida en el campo *zorder* (ver Figura 6.8), en el rango de 0 a 650. *Overlays* con menor valor de este campo serán dibujados encima del resto. Esto nos permite definir diferentes niveles de despliegue de elementos 2D.

Para finalizar estudiaremos la modificación en el código de *MyApp* y *MyFrameListener*. En el siguiente listado se muestra que el constructor del *FrameListener* (línea 7) necesita conocer el *OverlayManager*, cuya referencia se obtiene en la línea 14. En realidad no sería necesario pasar el puntero al constructor, pero evitamos de esta forma que el *FrameListener* tenga que solicitar la referencia.

En las líneas 15–16, obtenemos el *Overlay* llamado *Info* (definido en el listado de la Figura 6.8), y lo mostramos.

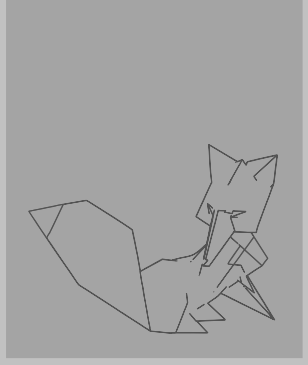
Para finalizar, el código del *FrameListener* definido en las líneas 6–16 del siguiente listado parcial, modifica el valor del texto asociado a los elementos de tipo TextArea. Hacemos uso de la clase auxiliar *Ogre::StringConverter* que facilita la conversión a String de ciertos tipos de datos (vectores, cuaternios...).

Listado 6.10: Modificación en MyApp.cpp

```
int MyApp::start() {
   ...
  loadResources();
  createScene();
  createOverlay();   // Metodo propio para crear el overlay
   ...
  _framelistener = new MyFrameListener(window, cam, node,
                       _overlayManager);
  _root->addFrameListener(_framelistener);
   ...
}

void MyApp::createOverlay() {
  _overlayManager = Ogre::OverlayManager::getSingletonPtr();
  Ogre::Overlay *overlay = _overlayManager->getByName("Info");
  overlay->show();
}
```

Listado 6.11: Modificación en MyFrameListener.cpp

```
bool MyFrameListener::frameStarted(const Ogre::FrameEvent& evt) {
  ...
  _camera->yaw(Ogre::Radian(rotx));
  _camera->pitch(Ogre::Radian(roty));

  Ogre::OverlayElement *oe;
  oe = _overlayManager->getOverlayElement("fpsInfo");
  oe->setCaption(Ogre::StringConverter::toString(fps));
  oe = _overlayManager->getOverlayElement("camPosInfo");
  oe->setCaption(Ogre::StringConverter::toString(_camera->
      getPosition()));
  oe = _overlayManager->getOverlayElement("camRotInfo");
  oe->setCaption(Ogre::StringConverter::toString(_camera->
      getDirection()));
  oe = _overlayManager->getOverlayElement("modRotInfo");
  Ogre::Quaternion q = _node->getOrientation();
  oe->setCaption(Ogre::String("RotZ: ") +
     Ogre::StringConverter::toString(q.getYaw()));

  return true;
}
```



Capítulo 7

Materiales y Texturas

Carlos González Morcillo

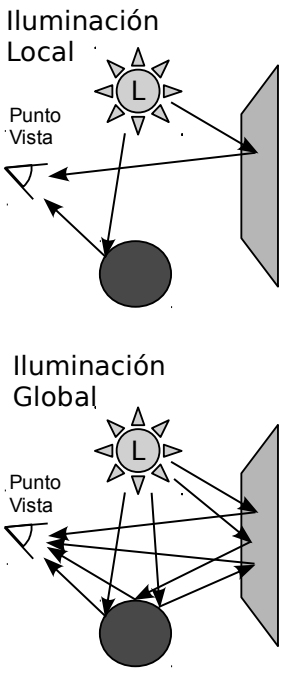


Figura 7.1: Diferencias entre los modelos de iluminación local y global.

En este capítulo estudiaremos los conceptos fundamentales con la definición de materiales y texturas. Introduciremos la relación entre los modos de sombreado y la interacción con las fuentes de luz, describiendo los modelos básicos soportados en aplicaciones interactivas. Para finalizar, estudiaremos la potente aproximación de Ogre para la definición de materiales, basándose en los conceptos de técnicas y pasadas.

7.1. Introducción

Los materiales describen las propiedades físicas de los objetos relativas a cómo reflejan la luz incidente. Obviamente, el aspecto final obtenido será dependiente tanto de las propiedades del material, como de la propia definición de las fuentes de luz. De este modo, materiales e iluminación están íntimamente relacionados.

Desde los inicios del estudio de la óptica, investigadores del campo de la física han desarrollado modelos matemáticos para estudiar la interacción de la luz en las superficies. Con la aparición del microprocesador, los ordenadores tuvieron suficiente potencia como para poder simular estas complejas interacciones.

Así, usando un ordenador y partiendo de las propiedades geométricas y de materiales especificadas numéricamente es posible simular la reflexión y propagación de la luz en una escena. A mayor precisión, mayor nivel de realismo en la imagen resultado.

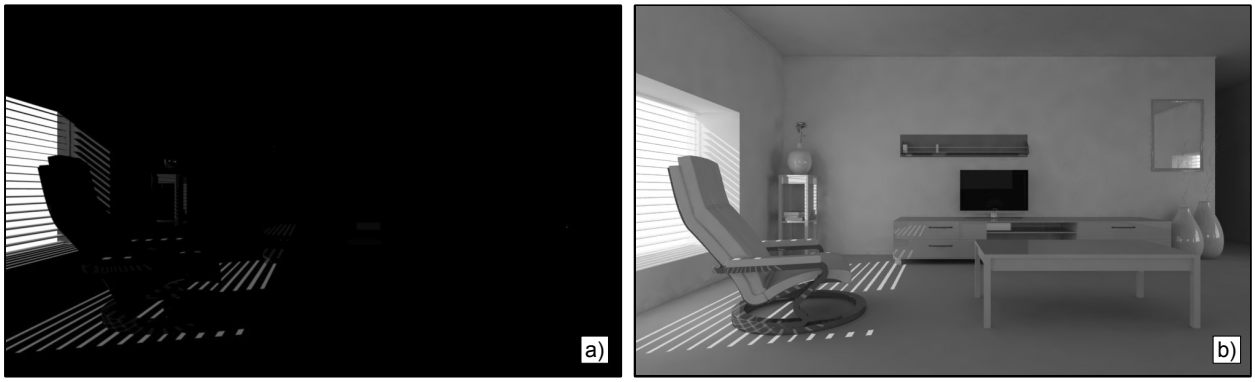


Figura 7.2: Ejemplo de resultado utilizando un modelo de iluminación local y un modelo de iluminación global con la misma configuración de fuentes de luz y propiedades de materiales. **a)** En el modelo de iluminación local, si no existen fuentes de luz en el interior de la habitación, los objetos aparecerán totalmente "*a oscuras*", ya que no se calculan los rebotes de luz indirectos. **b)** Los modelos de iluminación global tienen en cuenta esas contribuciones relativas a los rebotes de luz indirectos.

Esta conexión entre la simulación del comportamiento de la luz y el nivel de realismo queda patente en las aproximaciones existentes de diferentes métodos de render. Una ecuación que modela el comportamiento físico de la luz, ampliamente aceptada por la comunidad, es la propuesta por *Kajiya* en 1986. De forma general podemos decir que a mayor simplificación en la resolución de los términos de esta ecuación tendremos métodos menos realistas (y computacionalmente menos costosos).

A un alto nivel de abstracción, podemos realizar una primera taxonomía de métodos de render entre aquellos que realizan una simulación de **iluminación local**, teniendo en cuenta únicamente una interacción de la luz con las superficies, o los métodos de **iluminación global** que tratan de calcular *todas* [1] las interacciones de la luz con las superficies de la escena. La Figura 7.2 muestra el resultado de renderizar la misma escena con un método de iluminación local y uno global. Los modelos de iluminación global incorporan la iluminación directa que proviene de la primera interacción de las superficies con las fuentes de luz, así como la iluminación indirecta reflejada por otras superficies existentes en la escena.

La potencia de cálculo actual hace inviable el uso de métodos de iluminación global. Se emplean aproximaciones de precálculo de la iluminación, que serán estudiadas en el capítulo de iluminación. Así, en los motores gráficos actuales como Ogre, los materiales definen cómo se refleja la luz en los objetos (pero no su contribución con otros objetos), empleando un esquema de **iluminación local**.

[1] Debido a que es imposible calcular las infinitas interacciones de los rayos de luz con todos los objetos de la escena, las aproximaciones de iluminación global se ocuparán de calcular algunas de estas interacciones, tratando de minimizar el error de muestreo.

7.2. Modelos de Sombreado

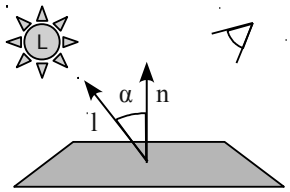


Figura 7.3: Modelo de sombreado difuso básico de Lambert en el que el color c se define como $c \propto n \cdot l$. Los vectores n y l deben estar normalizados.

Como hemos comentado anteriormente, es habitual en gráficos por computador interactivos (y en el caso de videojuegos especialmente) emplear modelos de iluminación local. En cierto modo, el sombrado es equivalente a *pintar con luz*. En este apartado estudiaremos los modelos de sombreado más ampliamente utilizados en gráficos interactivos, que fueron inicialmente desarrollados en los años 70.

- **Sombreado difuso**. Muchos objetos del mundo real tienen una acabado eminentemente mate (sin brillo). Por ejemplo, el papel o una tiza pueden ser superficies con sombreado principalmente difuso. Este tipo de materiales reflejan la luz en todas las direcciones, debido principalmente a las rugosidades microscópicas del material. Como efecto visual, el resultado de la iluminación es mayor cuando la luz incide perpendicularmente en la superficie. La intensidad final viene determinada por el ángulo que forma la luz y la superficie (es independiente del punto de vista del observador). En su expresión más simple, el modelo de reflexión difuso de *Lambert* dice que el color de una superficie es proporcional al coseno del ángulo formado entre la normal de la superficie y el vector de dirección de la fuente de luz (ver Figura 7.3).

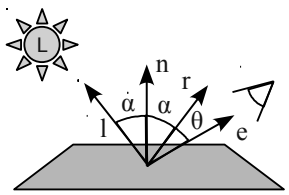


Figura 7.4: En el modelo de sombreado especular, el color c se obtiene como $c \propto (r \cdot e)^h$. Los vectores r y e deben estar normalizados.

- **Sombreado especular**. Es el empleado para simular los *brillos* de algunas superficies, como materiales pulidos, pintura plástica, etc. Una característica principal del sombreado especular es que el brillo se *mueve con el observador*. Esto implica que es necesario tener en cuenta el vector del observador. La *dureza* del brillo (la cantidad de reflejo del mismo) viene determinada por un parámetro *h*. A mayor valor del parámetro *h*, más concentrado será el brillo. El comportamiento de este modelo de sombreado está representado en la Figura 7.4, donde r es el vector reflejado del l (forma el mismo ángulo α con n) y e es el vector que se dirige del punto de sombreado al observador. De esta forma, el color final de la superficie es proporcional al ángulo θ.

- **Sombreado ambiental**. Esta componente básica permite añadir una *aproximación* a la *iluminación global* de la escena. Simplemente añade un color base independiente de la posición del observador y de la fuente de luz. De esta forma se evitan los tonos *absolutamente negros* debidos a la falta de iluminación global, añadiendo este término constante. En la Figura 7.5 se puede ver la componente ambiental de un objeto sencillo.

- **Sombreado de emisión**. Finalmente este término permite añadir una simulación de la iluminación propia del objeto. No obstante, debido a la falta de interacción con otras superficies (incluso con las caras poligonales del propio objeto), suele emplearse como una alternativa al sombreado ambiental a nivel local. El efecto es como tener un objeto que emite luz pero cuyos rayos no interactúan con ninguna superficie de la escena (ver Figura 7.5).

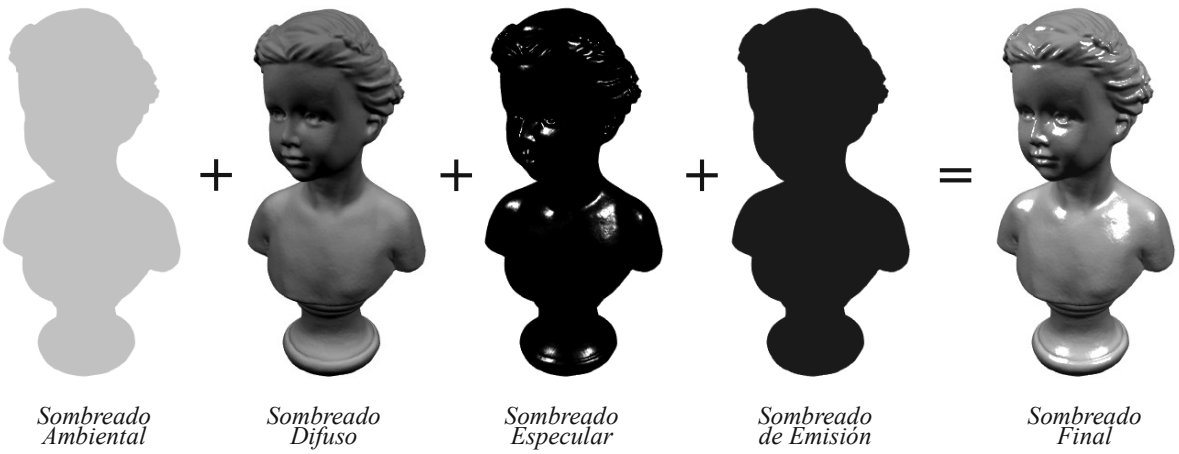


Figura 7.5: Modos básicos de sombreado de iluminación local.

El sombreado final de la superficie se obtiene como combinación de los cuatro modos de sombreado anteriores (ver Figura 7.5). Esta aproximación es una simplificación del modelo de reflexión físicamente correcto definido por la Función de Distribución de Reflactancia Bidireccional *(BRDF Bidirectional Reflactance Distribution Function)*. Esta función define cómo se refleja la luz en cualquier superficie opaca, y es empleada en motores de *rendering* fotorrealistas.

7.3. Mapeado de Texturas

Los materiales definen propiedades que son constantes a lo largo de la superficie. Hasta ahora, hemos hablado de materiales básicos en las superficies, con propiedades (como el color) constantes.

Las texturas permiten variar estas propiedades, determinando en cada punto cómo cambian concretamente estas propiedades. Básicamente podemos distinguir dos tipos de texturas:

- **Texturas Procedurales.** Su valor se determina mediante una ecuación. Estas texturas se calculan rápidamente y no tienen requisitos de espacio en disco, por lo que son ampliamente utilizadas en síntesis de imagen realista para simular ciertos patrones existentes en la naturaleza (madera, mármol, nubes, etc).

  La Figura 7.6 muestra un ejemplo de este tipo de texturas. En videojuegos sin embargo, se emplean en menor medida, ya que resulta habitualmente más interesante emplear texturas de imagen con una resolución controlada.

- **Texturas de Imagen.** Almacenan los valores en una imagen, típicamente bidimensional.

```
color func(p3d p){
 if (sin(p_z)>0)
   return C_1
 else
   return C_2
}
```

Figura 7.6: Definición de una sencilla textura procedural que define bandas de color dependiendo del valor de coordenada Z del punto 3D.

Las texturas procedurales obtienen valores habitualmente en el espacio 3D, por lo que no es necesaria ninguna función de proyección de estas texturas sobre el objeto. Sin embargo, para utilizar las texturas de imagen es necesario indicar cómo queremos aplicar esa textura (2D) a la geometría del modelo 3D. Es decir, debemos espeicifcar cómo se recubrirá el objeto con ese mapa de textura. Existen varias alternativas para realizar este mapeado. Empleando proyecciones ortogonales es posible describir esta correspondencia. Se emplean cuatro modos básicos de proyección (ver Figura 7.8). Estos modos de proyección utilizas las coordenadas del objeto 3D normalizadas en el interior de una caja unitaria.

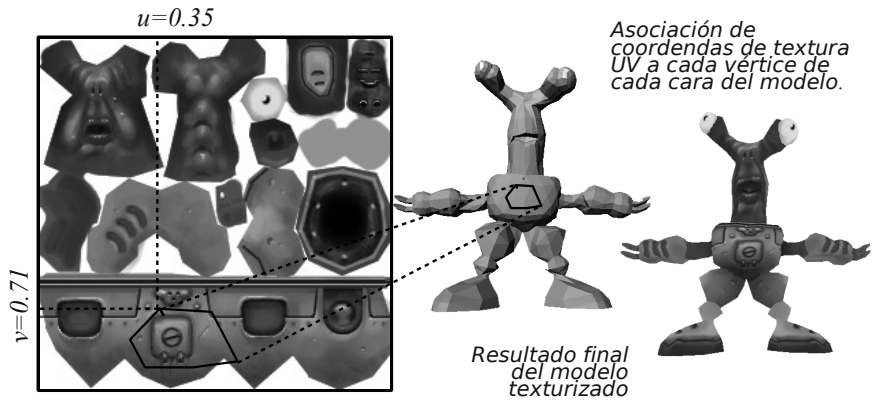


Figura 7.7: Asignación de coordenadas UV a un modelo poligonal. Esta operación suele realizarse con el soporte de alguna herramienta de edición 3D.

Una de las características interesantes de los modelos de mapeado de texturas (tanto ortogonales como mediante mapas paramétricos UV) es la **independencia de la resolución** de la imagen. De este modo es posible tener texturas de diferentes tamaños y emplearlas aplicando técnicas de nivel de detalle (*LOD*). Así, si un objeto se muestra a una gran distancia de la cámara es posible cargar texturas de menor resolución que requieran menor cantidad de memoria de la GPU.

Como hemos visto en capítulos anteriores, un método de proyección de texturas de imagen muy empleado en videojuegos es el mapeado paramétrico, también denominado mapeado UV. En este método se definen dos coordenadas paramétricas (entre 0 y 1) para cada vértice de cada cara del modelo (ver Figura 7.7). Estas coordenadas son independientes de la resolución de la imagen, por lo que permite cambiar en tiempo de ejecución la textura teniendo en cuenta ciertos factores de distancia, importancia del objeto, etc. El mapeado UV permite pegar la textura al modelo de una forma muy precisa. Incluso si se aplica sobre el modelo deformación de vértices (*vertex blending*), el mapa se-

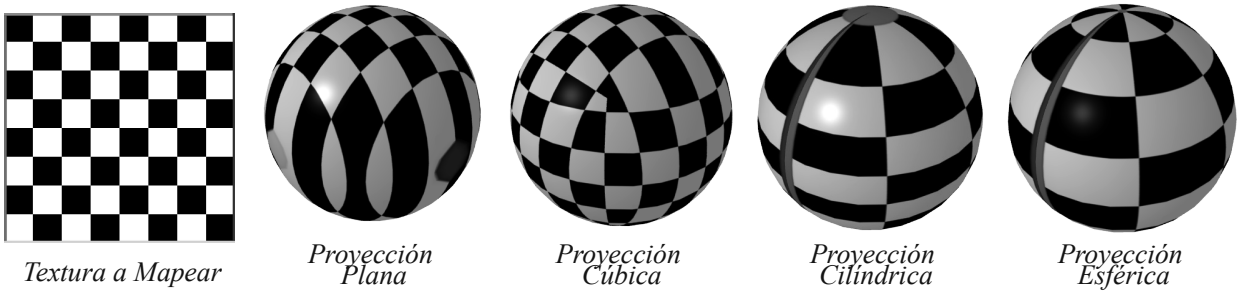


Figura 7.8: Métodos básicos de mapeado ortogonal sobre una esfera. Los bordes marcados con líneas de colores en la textura a mapear en la izquierda se proyectan de forma distinta empleado diversos métodos de proyección.

guirá aplicándose correctamente. Por esta razón y su alta eficiencia es una técnica ampliamente utilizada en gráficos por computador.

7.4. Materiales en Ogre

El despliegue de entidades en Ogre se realiza en paquetes, de modo que existe una relación directa entre el número de materiales y el número de *paquetes* que Ogre enviará a la tarjeta gráfica. Por ejemplo, si 10 elementos comparten el mismo material, podrán ser enviados en un único paquete a la GPU (en lugar de en 10 paquetes por separado), de modo que podrán compartir el mismo *estado interno*.

Con la idea de realizar estas optimizaciones, Ogre define que por defecto los materiales son compartidos entre objetos. Esto implica que el mismo puntero que referencia a un material es compartido por todos los objetos que utilizan ese material. De este modo, si queremos cambiar la propiedad de un material de modo que únicamente afecte a un objeto es necesario **clonar** este material para que los cambios no se propaguen al resto de objetos.

Los materiales de Ogre se definen empleando técnicas y esquemas. Una **Técnica** puede definirse como cada uno de los modos *alternativos* en los que puede renderizarse un material. De este modo, es posible tener, por ejemplo, diferentes niveles de detalle asociados a un material. Los **esquemas** agrupan técnicas, permitiendo definir nombres y grupos que identifiquen esas técnicas como *alto nivel de detalle*, *medio rendimiento*, etc.

Shading programable

En esta sección estudiaremos únicamente lo que se conoce como *shading* fijo (*fixed shading*). En capítulos posteriores del documento estudiaremos cómo aplicar shaders programando la GPU (en *pixel shading* o *fragment shading*).

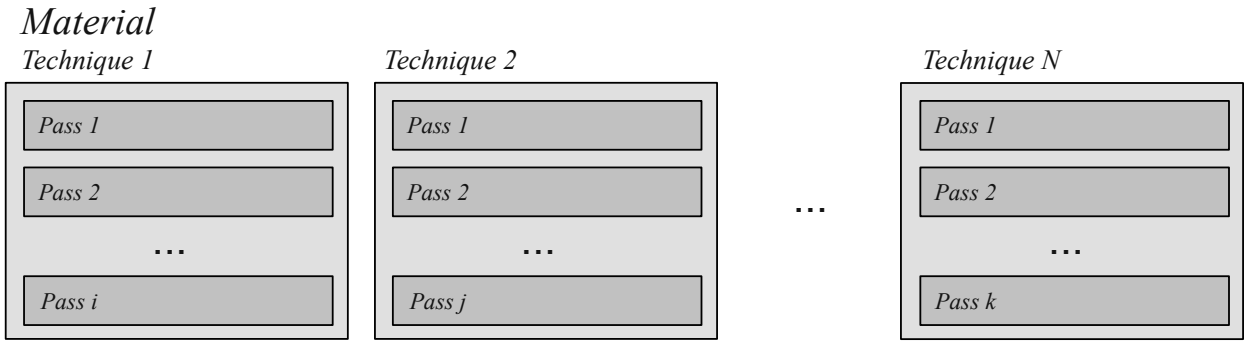


Figura 7.10: Descripción de un material en base a técnicas y pasadas. El número de técnicas y pasadas asociadas a cada técnica puede ser diferente.

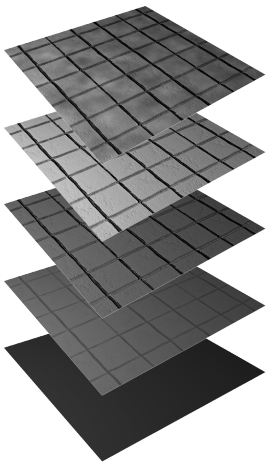


Figura 7.9: Un material en Ogre se describe mediante sucesivas pasadas que van configurando la apariencia final.

7.4.1. Composición

Un material en Ogre está formado por una o varias *técnicas*, que contienen a su vez una o varias **Pasadas** (ver Figura 7.10). En cada momento sólo puede existir una *técnica* activa, de modo que el resto de técnicas no se emplearán en esa etapa de render.

Una vez que Ogre ha decidido qué *técnica* empleará, generará tantas pasadas (en el mismo orden de definición) como indique el material. Cada *pasada* define una operación de despliegue en la GPU, por lo que si una técnica tiene cuatro pasadas asociadas a un material tendrá que desplegar el objeto tres veces en cada *frame*.

Las **Unidades de Textura** (*Texture Unit*) contienen referencias a una única textura. Esta textura puede ser generada en código (*procedural*), puede ser obtenida mediante un archivo o mediante un flujo de vídeo. Cada *pasada* puede utilizar tantas *unidades de textura* como sean necesarias. Ogre optimiza el envío de texturas a la GPU, de modo que únicamente se descargará de la memoria de la tarjeta cuando no se vaya a utilizar más.

7.4.2. Ejemplo de Materiales

A continuación veremos un ejemplo sencillo de definición de materiales en Ogre. Como hemos visto, el material definido en el listado de la Figura 7.11 contiene una técnica y una única pasada que define un método de sombreado difuso (especificando el color base en RGB). El nombre asignado al material especificado a la derecha de la etiqueta *Material* (en este caso "*Material1*") debe ser único a lo largo de la aplicación. El nombre de los *elementos* que definen el material (*técnicas*, *pasadas* y *unidades de textura*) es opcional. Si no se especifica ninguno, Ogre comenzará a nombrarlas comenzando en 0 según el orden de especificación del script. El nombre de estos *elementos* puede repetirse en diferentes materiales.

Atributo	Descripción
lod_values	Lista de distancias para aplicar diferentes niveles de detalle. Está relacionado con el campo *lod_strategy* (ver API de Ogre).
receive_shadows	Admite valores *on* (por defecto) y *off*. Indica si el objeto sólido puede recibir sombras. Los objetos transparentes *nunca* pueden recibir sombras (aunque sí arrojarlas, ver el siguiente campo).
transparency_casts_shadows	Indica si el material transparente puede arrojar sombras. Admite valores *on* y *off* (por defecto).
set_texture_alias	Permite crear alias de un nombre de textura. Primero se indica el nombre del alias y después del nombre de la textura original.

Tabla 7.1: Atributos generales de *Materiales*.

Atributo	Formato	Descripción
ambient	r g b [a]	Valor de sombreado ambiente (por defecto 1.0 1.0 1.0 1.0).
diffuse	r g b [a]	Valor de sombreado difuso (por defecto 1.0 1.0 1.0 1.0).
specular	r g b [a] h	Valor de sombreado especular (por defecto 0.0 0.0 0.0 0.0). El valor de dureza (*shininess*) puede ser cualquier valor >0.
emissive	r g b [a]	Valor de emisión (por defecto 0.0 0.0 0.0 0.0).
scene_blend	(Ver valores)	Tipo de mezclado de esta pasada con el resto de la escena. Por defecto no se realiza mezclado. Si se especifica, puede tomar valores entre *add*, *modulate*, *colour_blend* y *alpha_blend*.
depth_check	on | off	Por defecto *on*. Indica si se utilizará el *depth-buffer* para comprobar la profundidad.
lighting	on | off	Por defecto *on*. Indica si se empleará iluminación dinámica en la pasada.
shading	(Ver valores)	Indica el tipo de método de interpolación de iluminación a nivel de vértice. Se especifican los valores de interpolación de sombreado *flat* o *gouraud* o *phong*. Por defecto se emplea el método de *gouraud*.
polygon_mode	(Ver valores)	Indica cómo se representarán los polígonos. Admite tres valores: *solid* (por defecto), *wireframe* o *points*.

Tabla 7.2: Atributos generales de las *Pasadas*. Ver API de Ogre para una descripción completa de todos los atributos soportados.

Veamos a continuación un ejemplo más complejo de definición de material. En la pasada definida se utiliza una *texture_unit* que referencia a un archivo de mapa de entorno esférico.

Los mapas de entorno se utilizan para simular la reflexión del mundo (representado en el mapa) dependiendo de la relación entre las normales de las caras poligionales y la posición del observador. En este caso, se indica a Ogre que el tipo de mapa de entorno es esférico, por lo que la imagen empleada será del tipo de *ojo de pez* (ver Figura 7.13). El operador de *colour_op* empleado indica cómo se combinará el color de la textura con el color base del objeto. Existen diversas alternativas; mediante *modulate* se indica a Ogre que *multiplique* el color base


```
material Material1
{
  technique
  {
    pass
    {
      diffuse 0.5 0.5 0.5
    }
  }
}
```

Figura 7.11: Definición y resultado de un sencillo material con sombreado difuso.


```
material Material2
{
  technique
  {
    pass
    {
      diffuse 0.99 0.9 0.5
      ambient 0.2 0.2 0.2
      specular 1.0 1.0 0.9 30
      texture_unit
      {
        texture envmap.png
        env_map spherical
        colour_op modulate
      }
    }
  }
}
```

Figura 7.12: Un material más complejo que incluye la definición de una *texture_unit* con diversas componentes de sombreado.

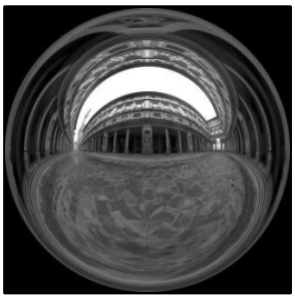


Figura 7.13: Mapa de entorno esférico empleado para simular la reflexión de un supuesto mundo.

(en este caso, un color amarillo indicado en la componente difusa del color) y el color del mapa.

A continuación se describen algunas de las propiedades generales de los materiales. En la tabla 7.1 se resumen los atributos generales más relevantes empleados en la definición de materiales, mientras que las tablas 7.2 y 7.3 resumen los atributos globales más utilizados en la definición de las pasadas y unidades de textura resepectivamente. Cada pasada tiene asociada cero o más unidades de textura. En los siguientes ejemplos veremos cómo combinar varias pasadas y unidades de textura para definir materiales complejos en Ogre.

Atributo	Descripción
texture	Especifica el nombre de la textura (estática) para esta *texture_unit*. Permite especificar el tipo, y el uso o no de canal alpha separado.
anim_texture	Permite utilizar un conjunto de imágenes como textura animada. Se especifica el número de frames y el tiempo (en segundos).
filtering	Filtro empleado para ampliar o reducir la textura. Admite valores entre *none*, *bilinear* (por defecto), *trilinear* o *anisotropic*.
colour_op	Permite determinar cómo se mezcla la *unidad de textura*. Admite valores entre *replace*, *add*, *modulate* (por defecto) o *alpha_blend*.
colour_op_ex	Versión extendida del atributo anterior, que permite especificar con mucho mayor detalle el tipo de mezclado.
env_map	Uso de mapa de entorno. Si se especifica (por defecto está en *off*, puede tomar valores entre *spherical*, *planar*, *cubic_reflection* y *cubic_normal*.
rotate	Permite ajustar la rotación de la textura. Requiere como parámetro el ángulo de rotación (en contra de las agujas del reloj).
scale	Ajusta la escala de la textura, especificando dos factores (en X e Y).

Tabla 7.3: Atributos generales de las *Unidades de Textura*. Ver API de Ogre para una descripción completa de todos los atributos soportados

7.5. Mapeado UV en Blender

Mapas UV

El mapeado UV es el estándar en desarrollo de videojuegos, por lo que prestaremos especial atención en este documento.

La forma más flexible para la proyección de texturas en aplicaciones interactivas es el mapeado paramétrico UV. Como hemos visto, a cada vértice del modelo (con coordenadas *X,Y,Z*) se le asocian dos coordenadas paramétricas 2D (*U,V*).

Los modelos de mapeado ortogonales no se ajustan bien en objetos complejos. Por ejemplo, la textura asociada a una cabeza de un personaje no se podría mapear adecuadamente empleando modelos de proyección ortogonal. Para tener el realismo necesario en modelos pintados manualmente (o proyectando texturas basadas en fotografías) es necesario tener control sobre la posición final de cada píxel sobre la textura. El mapa UV describe qué parte de la textura se asociará a cada polígono del modelo, de forma que, mediante una operación de *despliegue* (*unwrap*) del modelo obtenemos la equivalencia de la superficie en el plano 2D.

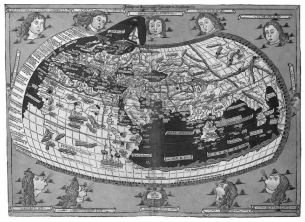


Figura 7.14: El problema del *despliegue* se ha afrontado desde los orígenes de la cartografía. En la imagen, un mapa de 1482 muestra una proyección del mundo conocido hasta el momento.

En esta sección estudiaremos con más detalle las opciones de Blender para trabajar con este tipo de coordenadas.

Como vimos en el capítulo 4, para comenzar es necesario aplicar una operación de *despliegue* del modelo para obtener una o varias regiones en la ventana de UV/Image Editor . Esta operación de despliegue se realiza siempre en modo edición. Con el objeto en modo edición, en los botones de edición aparece un nuevo panel *UV Calculation* (ver Figura 7.15) que controla el modo en el que se realiza el *despliegue* del modelo. Este panel aparece mientras el objeto está en modo de edición, y tiene las siguientes opciones:

Isla UV

Se define una isla en un mapa UV como cada región que contiene vértices no conectados con el resto.

- **Image Aspect**. Permite el escalado de los vértices desplegados en UV para ajustarse a la relación de aspecto de la textura a utilizar.
- **Transform Correction**. Corrige la distorsión del mapa UV mientras se está editando.
- **Pack Margin**. Define la separación mínima entre islas.
- **Angle Based | Conformal**. Define el método de cálculo de la proyección. El basado en ángulo crea una nueva isla cuando el ángulo entre caras vecinas sea relevante. En el modo *conformal* se realiza dependiendo del método de proyección seleccionado. El método basado en ángulo ofrece, en general, mejores resultados.
- **Fill Holes**. Intenta ordenar todas las islas para evitar que queden huecos en la textura sin ninguna cara UV.
- **Cube Size**. Cuando se emplea una proyección cúbica, este parámetro indica el porcentaje de la imagen que se emlpeará por el mapa UV (por defecto el 100 %).
- **Cyl Radius**. Cuando se emplea el método de proyección *Cylinder from View*, este parámetro define el radio del cilindro de proyección. A menores valores, las coordenadas del objeto aparecerán estiradas en el eje Y.
- **View Aligns Face | VA Top | Al Obj** Permite elegir el eje de alineación del objeto cuando se elige el método de proyección *From View*.
- **Polar ZX | Polar ZY**. Determina el plano frontal de proyección.

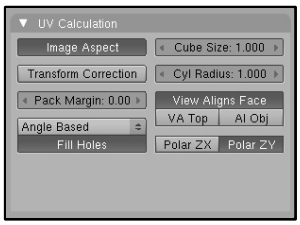


Figura 7.15: Opciones del panel *UVCalculation*.

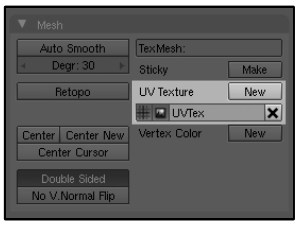


Figura 7.16: Opciones del panel *Mesh*.

El modo general de trabajo asociado al despliegue de una malla está compuesto por la siguiente serie de pasos secuenciales:

1. Seleccionar el objeto que queremos desplegar.
2. Suele ser interesante ver los cambios realizados sobre la textura UV cambiando el modo de dibujado del objeto en la ventana 3D a Textured .
3. Establecer los valores globales de despliegue en la pestaña *UV Calculation* (ver Figura 7.15).
4. Cambiar al modo edición (mediante TAB) y seleccionar las caras que queremos desplegar. En algunos casos serán todas (tecla A), o puede ser conveniente elegir individualmente (en este caso es conveniente elegir el modo de selección de caras en la cabecera de la ventana 3D).
5. Añadimos una nueva capa de textura UV en el panel Mesh del grupo de botones de edición (ver Figura 7.16). Cada cara de la malla puede tener varias texturas UV, pero cada textura únicamente puede tener una única imagen asignada.

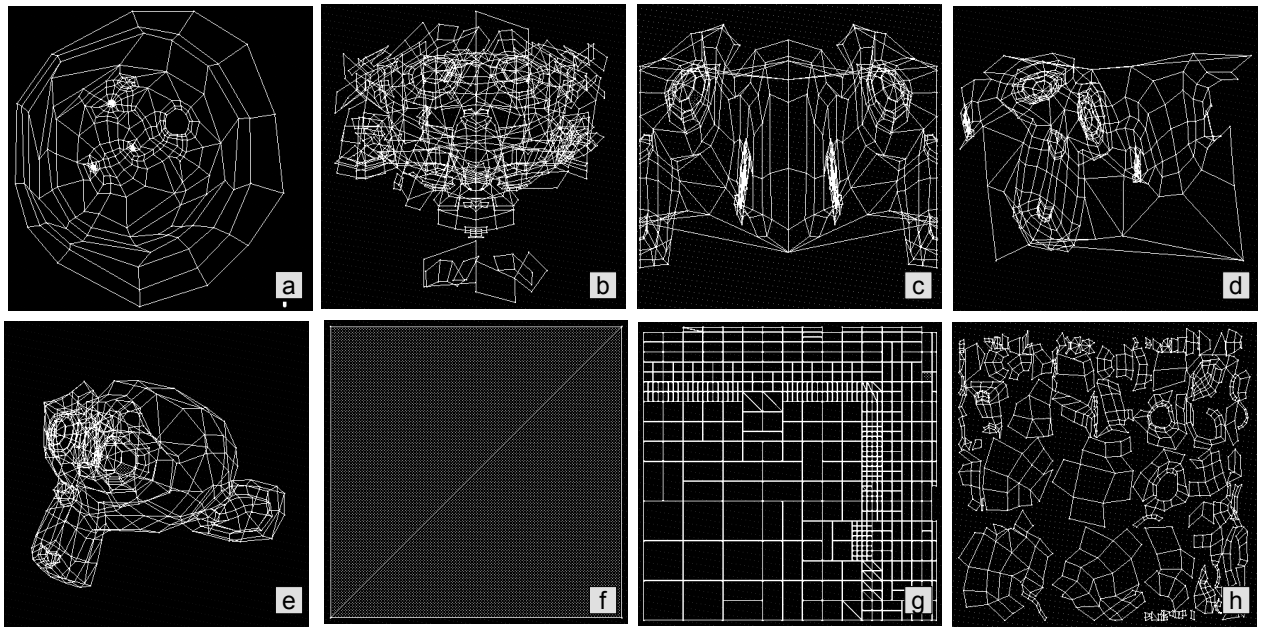


Figura 7.17: Resultado obtenido tras aplicar diferentes métodos de despliegue (*unwrap*) al modelo de Suzanne con todas las caras seleccionadas: **a)** Unwrap, **b)** Cube, **c)** Cylinder, **d)** Sphere, **e)** Project from view, **f)** Reset, **g)** Lightmap y **h)** Smart Projections.

6. Pulsando la tecla [u] accedemos al menú de despliegue (*unwrap*) que nos permite elegir el método de despiegue que aplicaremos as la malla. Entre estos métodos podemos destacar los siguientes (ver Figura 7.17):

   - **Unwrap**. Esta opción despliega las caras del objeto tratando de obtener una configuración que rellene de la mejor forma posible la textura, empleando información topológica de la malla (cómo están conectadas las caras de la malla). Si es posible, cada cara desplegada tendrá su propia región de la imagen sin solapar con otras caras.
   - **Cube, Cylinder, Sphere**. Estos métodos tratan de despegar el objeto empleando estos modelos de proyección ortogonales. Como hemos visto anteriormente, en estos métodos resulta especialmente relevante la definición de las opciones del panel de la Figura 7.15.
   - **Project from View**. Emplea el punto de vista 3D para desplegar el objeto.
   - **Reset**. Mediante esta opción todas las caras se despliegan ocupando el 100 % de la imagen, con vértices en las esquinas de la misma.
   - **Lightmap UVPack**. Despliega las caras de forma regular para ser utilizados en el cálculo de mapas de luz. Este tipo de despiegue será utilizado en el Capítulo 9 para el precálculo de iluminación.

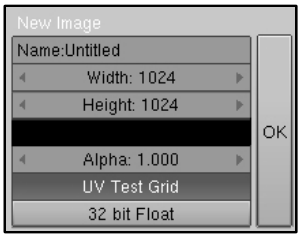


Figura 7.18: Ventana para la creación de una nueva textura para el mapeado UV.

- **Unwrap (Smart Projections)**. Este método estudia la forma del objeto, estudiando la relación entre las caras seleccionadas y crea un mapa UV basada en las opciones de configuración que se muestran en la Figura 7.19. A menor límite de ángulo, mayor número de islas se crearán. El margen existente entre islas se puede elegir en el parámetro *Island Margin*. Mediante *Fill Holes* se puede forzar a que el método trate de utilizar todo el espacio posible de la textura. Este método de proyección automático se ajusta perfectamente en multitud de situaciones, siendo el que probablemente ofrezca una mejor configuración de partida para ajustar posteriormente los vértices de forma manual.

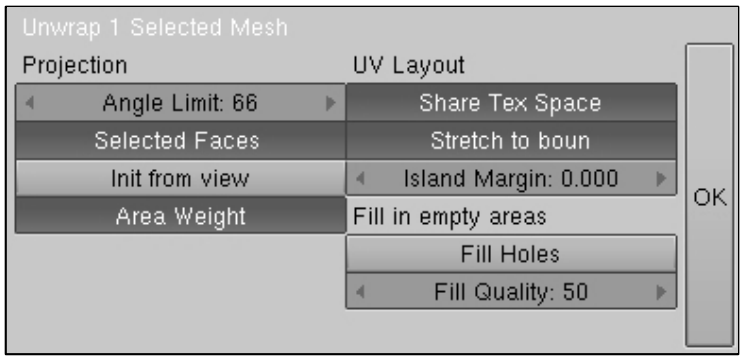


Figura 7.19: Configuración de las opciones de despliegue en *Smart Projections*.

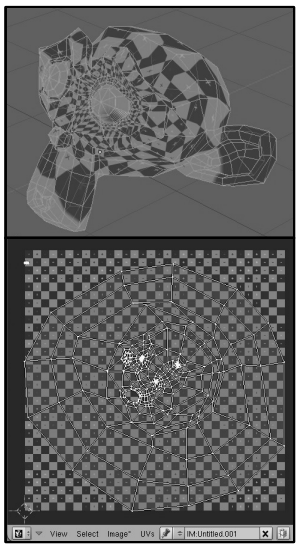


Figura 7.20: Resultado de aplicar la textura de prueba a la malla de *Suzanne* con el modo de despliegue básico.

Una vez que la malla ha sido desplegada, podemos ayudarnos de la herramienta de generación de rejillas de prueba que trae integrado Blender para hacernos una idea del resultado del despliegue. En la cabecera de la ventana del editor UV, accediendo al menú *Image/ New...* obtenemos la ventana de la Figura 7.18, donde podemos elegir la resolución en píxeles de la nueva imagen (por defecto 1024x1024 píxeles), el color de fondo y el nivel de Alpha. Si activamos el botón *UV Test Grid*, se creará la imagen con la rejilla de fondo que se muestra en la Figura 7.20. Esta imagen permite hacerse una idea del resultado del despliegue tras aplicar la textura al objeto 3D.

Las coordenadas UV se almacenan en el modelo de Blender y son pasadas a Ogre empleando el script de exportación. Sin embargo, es común utilizar un editor de imágenes externo para asignar colores a la textura. Para guardar el estado del despliegue (y utilizarlo como plantilla para el programa de edición, como GIMP), es posible emplear el script de la cabecera de la ventana del *UV Image Editor* en *UVs / Scripts / Save UV Face Layout* para generar una imagen TGA o SVG.

En la cabecera de la ventana 3D puede elegirse el modo *Texture Paint* (ver Figura 7.21. Cuando este modo está activo, nos permite dibujar directamente sobre el modelo 3D, accediendo a los controles que se encuentran en la pestaña *Paint* del grupo de botones de edición (ver Figura 7.22)

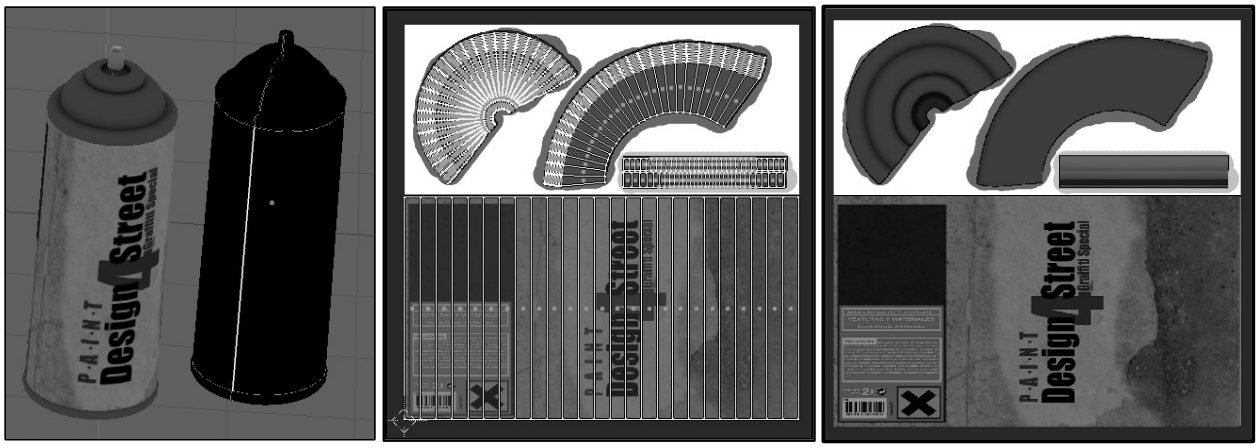


Figura 7.23: Utilización de *costuras* y despliegue del modelo utilizando diferentes técnicas de despliegue. En la imagen de la izquerda se marcan perfectamente los *seams* definidos para recortar el modelo adecuadamente.

7.5.1. Costuras

En el despliegue de mallas complejas, es necesario ayudar a los métodos de despliegue definiendo costuras (*seams*). Estas costuras servirán para definir las zonas de recorte, guiando así al método de desplegado.

Para definir una costura basta con elegir las aristas que la definen en modo edición, tal y como se muestra en la Figura 7.23 . La selección de bucles de aristas (como en el caso del bote de spray del modelo) puede realizarse cómodamente seleccionando una de las aristas del bucle y empleando la combinación de teclas Control E *Edge Loop Select*. Cuando tenemos la zona de recorte seleccionada, definiremos la costura mediante la combinación Control E *Mark Seam*. En este menú aparece igualmente la opción para eliminar una costura *Clear Seam*.

Una vez definidas las costuras, podemos emplear la selección de zonas enlazadas (mediante la tecla L) para elegir regiones conectadas. Podemos aplicar a cada grupo de caras un modo de despliegue distinto. Por ejemplo, en la Figura 7.23 se ha utilizado una proyección cilíndrica para el cuerpo central del Spray y para el difusor, mientras que la base y la zona superior se han desplegado mediante el modo general Unwrap.

Cada zona de despliegue en la ventana de *UV Image Editor* puede igualmente desplazarse empleando los operadores habituales de traslación G, rotación R y escalado S. En esta ventana puede igualmente emplearse el atajo de teclado para seleccionar los vértices conectados (link) L y seleccionar rápidamente regiones desplegadas (incrementalmente si pulsamos L mientras mantenemos pulsada la tecla Shift).

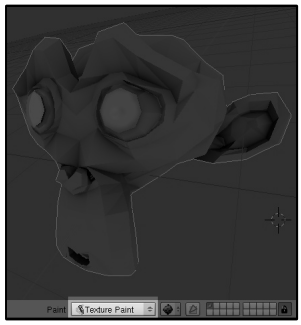


Figura 7.21: Editando la textura aplicada a Suzanne en el espacio 3D mediante el modo *Texture Paint*.

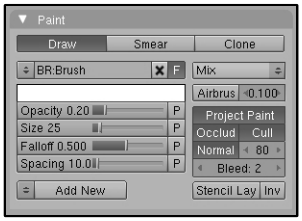
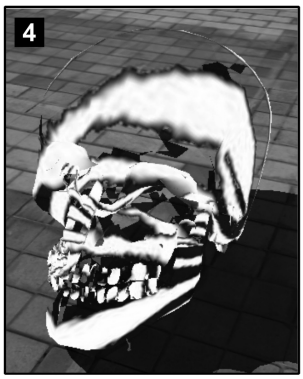
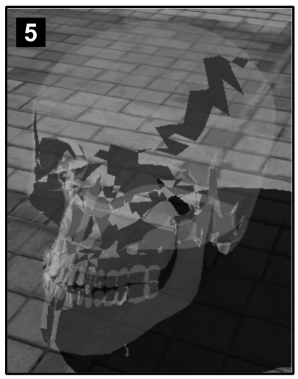


Figura 7.22: Panel *Paint* en el que puede elegirse el color de pintado, tamaño del pincel, opacidad, etc...

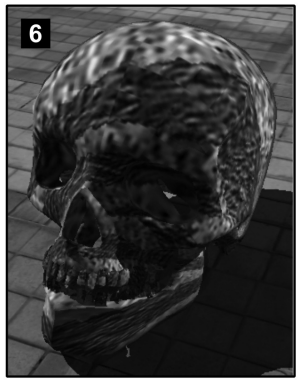

```
material Material3
{
 technique
 {
  pass
  {
   diffuse 0.5 0.5 0.5
   texture_unit
   {
    texture grid.jpg
   }
  }
 }
}
```




```
material Material4
{
 technique
 {
  pass
  {
   scene_blend
     alpha_blend
   depth_write off
   lighting off

   texture_unit
   {
     texture
       zebratrp.png
   }
  }
 }
}
```




```
material Material5
{
 technique
 {
  pass
  {
   lighting off
   scene_blend
      alpha_blend
   depth_write off

   texture_unit
   {
    colour_op_ex
      source1
      src_manual
      src_current
      0.0 1.0 0.0
    alpha_op_ex
      source1
      src_manual
      src_current 0.25
}}}}}
```




```
material Material6
{
 technique
 {
 pass
 {
  texture_unit
  {
   texture grass.jpg
  }
  texture_unit
  {
   texture zebratrp.png
   colour_op alpha_blend
  }
  texture_unit
  {
    texture rock.jpg
    colour_op_ex
    blend_current_alpha
    src_texture
       src_current
}}}}
```

Figura 7.24: Ejemplos de definición de algunos materiales empleando diversos modos de composición en Ogre.

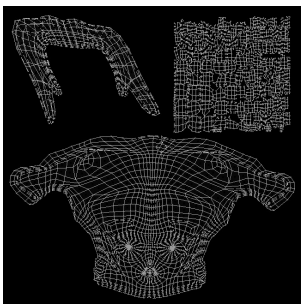


Figura 7.25: Despliegue del modelo de los ejemplos. Se ha empleado una operación de despliegue cinlíndrico para la mandíbula inferior, *Smart Projections* para los dientes y el *Unwrap* básico para el resto del objeto.

7.6. Ejemplos de Materiales en Ogre

En esta sección estudiaremos algunos ejemplos de definición de materiales empleando diversas unidades de tetura en Ogre. Nos centraremos en el uso de algunos operadores para la composición de diferentes unidades de textura. Las Figuras 7.24 y 7.27 muestran el resultado de definición de estos materiales y texturas. Veamos algunas de las propiedades empleadas en su definición.

En la definición del **Material3** se ha empleado el despliegue que se muestra en la Figura 7.25. Este material simplemente utiliza las coordenadas UV del modelo para proyectar la misma textura que en el suelo de la escena.

En la definición del **Material4** se utiliza una textura de tipo *cebra* donde las bandas negras son totalmente transparentes. En la pasada de este material se utiliza el atributo *scene_blend* que permite especificar el tipo de mezclado de esta pasada con el resto de contenido de la escena. Como veremos en el ejemplo del *Material6*, la pricipal diferen-

cia entre las operaciones de mezclado a nivel de pasada o a nivel de textura es que las primeras definen la mezcla con el contenido *global* de la escena, mientras que las segundas están limitadas entre capas de textura. Mediante el modificador *alpha_blend* se indica que utilizaremos el valor *alpha* de la textura. A continuación estudiaremos las opciones permitidas por el atributo *scene_blend* en sus dos versiones.

El formato de **scene_blend** permite elegir cuatro parámetros en su forma básica:

- **add**. El color del resultado de la pasada se *suma* al resultado de la escena.
- **modulate**. El resultado de la pasada se *multiplica* con el resultado de color de la escena.
- **colour_blend**. El color resultado de la pasada se mezcla empleando la componente de brillo de los colores.
- **alpha_blend**. Este modo utiliza la información de transparencia del resultado.

Figura 7.26: En los ejemplos se han utilizado dos versiones de la textura de cebra; una con transparencia total definida en las bandas negras (en formato PNG), y otra en JPG sin transparencia.

El atributo **scene_blend** permite otro formato mucho más general y flexible, que requiere dos parámetros: *scene_blend src dest*. En esta segunda versión, el color final se obtiene como: *c = (texture * src) + (scene_pixel * dest)*. En esta segunda versión, *src* y *dest* son factores de mezclado, entre 10 posibles valores:

- *one*. Valor constante 1.0.
- *zero*. Valor constante 0.0.
- *dest_colour*. Color del píxel en la escena.
- *src_colour*. Color del téxel (de la pasada).
- *one_minus_dest_colour*. 1 - (dest_colour).
- *one_minus_src_colour*. 1 - (src_colour).
- *dest_alpha*. Valor alfa del píxel en la escena.
- *src_alpha*. Valor alfa del téxel (de la pasada).
- *one_minus_dest_alpha*. 1 - (dest_alpha).
- *one_minus_src_alpha*. 1 - (src_alpha).

De este modo, el *scene_blend* por defecto que se aplica es el totalmente opaco, utilizando totalmente el valor del téxel de la pasada y sin tener en cuenta el valor del píxel existente en la escena (*scene_blend one zero*). Análogamente, el equivalente al modo básico de *alpha_blend* estudiado anteriormente sería *scene_blend src_alpha one_minus_src_alpha*, el modo básico *add* se escribiría como *scene_blend one one*, etc...

Consulta el manual!

En el manual de Ogre pueden consultarse todos los parámetros que admiten todos los operadores disponibles a nivel de textura y pasadas.

Para finalizar la descripción de los atributos utilizados en el *Material4*, mediante *depth_write off* se fuerza a que la pasada no utilice el

ZBuffer. Es habitual desactivar el *ZBuffer* cuando se despliegan objetos transparentes sobre la escena, para que se superpongan adecuadamente. El atributo *lighting off* indica que no se tenga en cuenta la iluminación dinámica en esta pasada. Cuando se desactiva la iluminación dinámica, las propiedades de sombreado difusas, ambientales, especulares y de emisión no se tienen en cuenta, por lo que cualquier definición en el material sería redundante.

El **Material5** define un color transparente definiendo una unidad de textura manualmente. Este material hace uso de las versiones que más precisión ofrecen a la hora de definir cómo se combina el color (y la opacidad) de una capa de textura con las anteriores. Así, la definición del color manualmente de la fuente como verde, y el nivel de transparencia como 0.25. En términos generales, el atributo *colour_op_ex* requiere como primer parámetro el identificador de una operación *operation*, y luego dos fuentes *src1 src2*. Las fuentes pueden ser una de las siguientes cinco opciones:

- *src_current*. El color obtenido en capas anteriores.
- *src_texture*. El color de la capa actual.
- *src_diffuse*. El color difuso de los vértices.
- *src_specular*. El color especular de los vértices.
- *src_manual*. Definición manual del color.

Como operación admite 15 valores diferentes. Si se indica *source1* o *source2* se utilizará su valor directamente sin modificación. La operación *blend_current_alpha* (como veremos en el siguiente ejemplo) utiliza la información de *alpha* de las capas anteriores. La operación *modulate* multiplica los valores de *src1* y *src2*.

El **Material6** define tres capas de textura. La primera carga una textura de césped. La segunda capa carga la textura de cebra con transparencia, e indica que la combinará con la anterior empleando la información de alpha de esta segunda textura. Finalmente la tercera capa utiliza el operador extendido de *colour_op_ex*, donde define que combinará el color de la capa actual (indicado como primer parámetro en *src_texture* con el obtenido en las capas anteriores *src_current*. El parámetro *blend_current_alpha* multiplica el alpha de *src2* por *(1-alpha(src1))*.

Los ejemplos de la Figura 7.27 utilizan algunas de las opciones de animación existentes en las unidades de textura. El **Material7** por ejemplo define dos capas de textura. La primera aplica la textura del suelo al objeto, utilizando el atributo *rotate_anim*. Este atributo requiere un parámetro que indica el número de revoluciones por segundo (empleando velocidad constante) de la textura.

La composición de la segunda capa de textura (que emplea un mapeado de entorno esférico) se realiza utilizando el atributo *colour_op_ex* estudiado anteriormente. La versión de la operación *modulate_x2* multiplica el resultado de *modulate* (multiplicación) por dos, para obtener un resultado más brillante.

```
material Material7
{
 technique
 {
  pass
  {
    ambient 1 1 1
    diffuse 1 1 1
    Specular 1 1 1 9800
    emissive 0 0 0 1
    texture_unit
    {
      rotate_anim 0.01
      texture grid.jpg
    }
    texture_unit
    {
      texture envmap.png
      colour_op_ex
        modulate_x2
        src_texture
        src_current
      env_map spherical
}}}}
```


```
material Material8
{
 technique
 {
  pass
  {
   ambient 0.5 0.5 0.5
   diffuse 1.0 1.0 1.0

   texture_unit
   {
     texture grass.jpg
     scroll_anim 0.1 0.0
     wave_xform
       scale sine
       0.0 0.2 0.0 0.2
   }

   texture_unit
   {
     texture zebra.jpg
     rotate_anim 0.15
     colour_op add
]}}}
```

Figura 7.27: Ejemplos de uso de animación de texturas (rotación y onda senoidal).

En el ejemplo del **Material8** se definen dos capas de textura, ambas con animación. La segunda capa utiliza una operación de suma sobre la textura de *cebra* sin opacidad, de modo que las bandas negras se *ignoran*. A esta textura se aplica una rotación de 0.15 revoluciones por segundo. La primera capa utiliza el atributo *scroll_anim* que permite definir un desplazamiento constante de la textura en X (primer parámetro) y en Y (segundo parámetro). En este ejemplo la textura únicamente se desplaza en el eje X. De igual modo, la primera capa emplea *wave_xform* para definir una animación basado en una función de onda. En este caso se utiliza una función senoidal indicando los parámetros requeridos de base, frecuencia, fase y amplitud (ver el manual de Ogre para más detalles sobre el uso de la función).

7.7. Render a Textura

En esta sección estudiaremos un ejemplo en el que se definirá una textura que contendrá el resultado de renderizar la escena empleando otra cámara auxiliar. Para ello, estudiaremos la creación manual de texturas, y el uso de un *Listener* particular que será ejecutado cada vez que se produzca una actualización en la textura.

El objeto sobre el que se proyectará la textura está definido en la Figura 7.28.a). En este modelo se han definido tres materiales, cada uno con sus propias coordenadas de despiegue. Las coordenadas más importantes son las relativas a la pantalla de la televisión, sobre la que desplegaremos la textura. Estas coordenadas se han definido de modo que ocupan todo el área de mapeado (como se muestra en la Figura 7.28.b). El material asociado a la pantalla se ha nombrado como *"pantallaTV"*, y será editado en código a nivel de *SubEntity*. Cuando un modelo cuenta con diversos materiales (como es el caso), Ogre crea

Figura 7.28: Construcción del ejemplo de *Render a Textura (RTT)*. **a)** Modelo empleado para desplegar la textura de *RTT*. El modelo se ha definido con tres materiales; uno para la carcasa, otro para las letras del logotipo (ambos desplegados en **c)**, y uno para la pantalla llamado "*pantallaTV*", desplegado en **b)**. En **d)** se muestra el resultado de la ejecución del programa de ejemplo.

un objeto de la clase *SubEntity* para cada material, de modo que, como veremos a continuación, tendremos que acceder a todos los *SubEntity* del objeto para elegir el que tiene el material de la pantalla y cambiarlo por el que calcularemos en tiempo de ejecución.

Render a Textura

Existen multitud de aplicaciones en videojuegos para utilizar Render a Textura. Por ejemplo, los espejos de cualquier simulador de conducción o cámaras de vigilancia dentro del juego, así como multitud de efectos gráficos (como espejos, motion blur...) y de postproducción que se realizan empleando esta técnica.

Hasta ahora hemos desplegado el resultado de la escena sobre la ventana principal de la aplicación. Sin embargo, en multitud de aplicaciones es habitual renderizar la escena total o parcialmente sobre una textura. Ogre facilita enormemente la construcción de este tipo de texturas. Una vez que tenemos la textura, podemos aplicar cualquier operador de los estudiados en la sección anterior para mezclarlas con otras capas de textura.

El siguiente listado muestra el código relevante para el ejemplo de Render a Textura.

El primer paso para aplicar el Render a Textura es obtener un objeto de tipo *TexturePtr* (líneas 1–3), que permite crear una textura manualmente, indicando el nombre de la textura ("*RttT*"), y las propiedades de tamaño (*512x512* píxeles), así como el formato de color (32Bits en RGB, sin canal alfa *PF_R8G8B8*). El último parámetro de *TU_RENDERTARGET* indica a Ogre el tipo de uso que haremos de la textura.

Listado 7.1: Fragmento de CreateScene (MyApp.cpp).

```
1  TexturePtr rtt = TextureManager::getSingleton().createManual(
2   "RttT", ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
3   TEX_TYPE_2D, 512, 512, 0, PF_R8G8B8, TU_RENDERTARGET);
4
5  RenderTexture *rtex = rtt->getBuffer()->getRenderTarget();
6
7  Camera *cam = _sceneManager->createCamera("SecondCamera");
8  cam->setPosition(Vector3(17,16,-4));
9  cam->lookAt(Vector3(-3,2.7,0));
10 cam->setNearClipDistance(5);
11 cam->setFOVy(Degree(38));
12
13 rtex->addViewport(cam);
14 rtex->getViewport(0)->setClearEveryFrame(true);
15 rtex->getViewport(0)->setBackgroundColour(ColourValue::Black);
16 rtex->getViewport(0)->setOverlaysEnabled(false);
17 rtex->setAutoUpdated(true);
18
19 MaterialPtr mPtr = MaterialManager::getSingleton().create(
20  "RttMat",Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
21 Technique* matTechnique = mPtr->createTechnique();
22 matTechnique->createPass();
23 mPtr->getTechnique(0)->getPass(0)->setLightingEnabled(true);
24 mPtr->getTechnique(0)->getPass(0)->setDiffuse(.9,.9,.9,1);
25 mPtr->getTechnique(0)->getPass(0)->setSelfIllumination(.4,.4,.4);
26
27 mPtr->getTechnique(0)->getPass(0)->createTextureUnitState("RttT");
28
29 for (unsigned int i=0; i<entTV->getNumSubEntities(); i++) {
30   SubEntity *aux = entTV->getSubEntity(i);
31   if (aux->getMaterialName() == "pantallaTV")
32     aux->setMaterialName("RttMat");
33 }
```

En la línea [5] se obtiene un puntero a un *RenderTexture* que es una especialización de la clase *RenderTarget* específica para renderizar sobre una textura.

Puede haber varios *RenderTargets* que generen resultados sobre la misma textura. A este objeto de tipo *RenderTexture* le asociamos una cámara en la línea [13] (que ha sido creada en las líneas [7-11]), y configuramos las propiedades específidas del viewport asociado (como que se limpie en cada frame en la línea [14], y que no represente los overlays en la línea [16]).

En la línea [17] indicamos que la textura se actualice automáticamente (gestionada por el bucle principal de Ogre). En otro caso, será necesario ejecutar manualmente el método *update* del *RenderTarget*.

Las líneas [19-25] declaran manualmente el material sobre el que emplearemos la *Texture Unit* con la textura anteriormente definida. En [19-20] creamos un material llamado "*RttMat*", con una única técnica (línea [21]) y una pasada (línea [22]). Esa pasada define sus propiedades en las líneas [23-25], y añade como *TextureUnit* a la textura "*RttT*".

El último bucle recorre todas las *SubEntities* que forman a la entidad de la televisión. En el caso de que el material asociado a la subentidad sea el llamado "*pantallaTV*" (línea [31]), le asignamos el material que hemos creado anteriormente (línea [32]).

7.7.1. Texture Listener

Figura 7.29: Tras aplicar el texture listener que oculta la entidad de la televisión, se consigue eliminar el efecto de despliegue recursivo.

En el ejemplo anterior, la textura proyectada sobre la televisión tiene un efecto recursivo debido a que aparece en el render de la cámara auxiliar. En muchos casos, interesa ocultar uno o varios objetos de la escena para realizar el render a textura (por ejemplo si queremos simular el punto de vista desde el interior de un objeto).

Al igual que ocurre a nivel de *Frame*, es posible añadir uno o varios objetos *Listener* para controlar la actualización de las texturas. De este modo, cada vez que se renderiza un *RenderTarget* (en nuestro caso concreto una textura), Ogre invocará previamente el método asociado al *preRenderTargetUpdate*. Cuando la textura se haya actualizado, se ejecutará el método llamado *postRenderTargetUpdate*.

El siguiente listado muestra el fichero de cabecera de la declaración de una clase propia llamada *MyTextureListener* que implementa el interfaz definido en *RenderTargetListener*. Esta clase recibe en el constructor un parámetro con el puntero a un objeto de tipo *Entity*. La clase se encargará de ocultar esa entidad antes de renderizar la escena sobre la textura y de volver a mostrarlo tras el despliegue.

Listado 7.2: MyTextureListener.h

```
1  #include <Ogre.h>
2  using namespace std;
3  using namespace Ogre;
4
5  class MyTextureListener : public RenderTargetListener {
6  private:
7   Entity* _ent;
8  public:
9   MyTextureListener(Entity *ent);
10  ~MyTextureListener();
11  virtual void preRenderTargetUpdate(const RenderTargetEvent& evt);
12  virtual void postRenderTargetUpdate(const RenderTargetEvent& evt);
13 };
```

A continuación se muestra el listado que define la clase MyTextureListener. La funcionalidad ha sido comentada anteriormente.

Listado 7.3: MyTextureListener.cpp

```
1  #include "MyTextureListener.h"
2
3  MyTextureListener::MyTextureListener(Entity* ent){
4    _ent = ent;
5  }
6
7  MyTextureListener::~MyTextureListener(){ }
8
9  void MyTextureListener::preRenderTargetUpdate(const
       RenderTargetEvent& evt) {
10   cout << "preRenderTargetupdate" << endl;
11   _ent->setVisible(false);
12 }
13
14 void MyTextureListener::postRenderTargetUpdate(const
       RenderTargetEvent& evt) {
```

```
15    cout << "postRenderTargetupdate" << endl;
16    _ent->setVisible(true);
17  }
```

El uso de la clase es muy sencillo. Basta con añadir el listener al objeto de tipo *RenderTexture* (ver línea [2] del siguiente listado).

Listado 7.4: Utilización en MyApp.cpp

```
1 _textureListener = new MyTextureListener(entTV);
2 rtex->addListener(_textureListener);
```

7.7.2. Espejo (Mirror)

La reflexión en *Espejo* es otro de los efectos clásicos que se obtienen mediante render a textura. El siguiente listado muestra el código necesario para realizar este ejemplo. Estudiaremos los aspectos que lo distinguen sobre el código de la sección anterior.

En las líneas [8-12] se define la cámara que utilizaremos para crear el efecto de *mirror*. Esta cámara *debe* tener la misma posición y orientación que la cámara desde donde percibimos la escena, pero debe ser independiente (ya que activaremos la reflexión sobre un plano, y modificaremos su plano de recorte cercano en las líneas *42-43*). Esta cámara para el efecto de espejo (llamada *MirrorCamera* en este ejemplo) debe estar *siempre* correctamente alineada con la cámara principal. En este ejemplo, la cámara principal es estática, por lo que tampoco modificaremos la posición y orientación de la *MirrorCamera*. Queda como ejercicio propuesto para el lector añadir movimiento a la cámara principal, actualizando análogamente en cada frame la posición de la cámara *Mirror*.

Figura 7.30: Resultado de aplicar el material con Render a Textura para simular espejo sobre el plano del suelo.

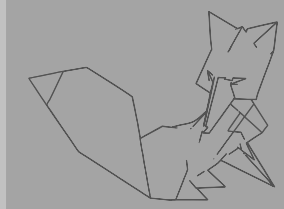
En las líneas [24-30] se definen las propiedades generales del material. La línea [31] nos permite generar las coordenadas de textura según la cámara que se le pasa como argumento, de modo que da la impresión de que la textura ha sido proyectada sobre la superficie. De esta forma, la textura que generamos será posteriormente renderizada utilizando el modelo de proyección definido por la cámara de *Mirror*.

Para concluir, en las líneas [42-43] se configuran los aspectos relativos a la cámara *Mirror* en relación al plano de reflexión. La llamada a *enableReflection* hace que se modifique el *Frustum* de la cámara de modo que renderice empleando la reflexión con respecto del plano que se le pasa como argumento.

Finalmente, la llamada a *enableCustomNearClipPlane* permite recortar la geometría situada *debajo* del plano pasado como argumento, de modo que únicamente la geometría que está situada *sobre* el plano será finalmente desplegada en el reflejo, evitando así errores de visualización.

Listado 7.5: Definición del Material tipo "Espejo"

```
TexturePtr rttM_texture = TextureManager::getSingleton()
 .createManual("RttMTex", ResourceGroupManager::
 DEFAULT_RESOURCE_GROUP_NAME, TEX_TYPE_2D, 512, 512, 0, PF_R8G8B8,
 TU_RENDERTARGET);

RenderTexture *rMtex= rttM_texture->getBuffer()->getRenderTarget();

Camera *camM = _sceneManager->createCamera("MirrorCamera");
Camera *mainCam = _sceneManager->getCamera("MainCamera");
camM->setPosition(mainCam->getPosition());
camM->setOrientation(mainCam->getOrientation());
camM->setAspectRatio(mainCam->getAspectRatio());

rMtex->addViewport(camM);
rMtex->getViewport(0)->setClearEveryFrame(true);
rMtex->getViewport(0)->setBackgroundColour(ColourValue::Black);
rMtex->getViewport(0)->setOverlaysEnabled(false);
rMtex->setAutoUpdated(true);

MaterialPtr mMPtr=MaterialManager::getSingleton().create("RttMMat",
  Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
Technique* matMTechnique = mMPtr->createTechnique();
matMTechnique->createPass();
TextureUnitState *t = mMPtr->getTechnique(0)->getPass(0)->
    createTextureUnitState("grid.jpg");
t = mMPtr->getTechnique(0)->
    getPass(0)->createTextureUnitState("RttMTex");
t->setColourOperationEx(LBX_BLEND_MANUAL, LBS_TEXTURE,
    LBS_CURRENT, ColourValue::White, ColourValue::White, 0.5);

t->setTextureAddressingMode(TextureUnitState::TAM_CLAMP);
t->setProjectiveTexturing(true, camM);

// Creacion del plano del suelo...
Plane plane1(Vector3::UNIT_Y, 0);
MeshManager::getSingleton().createPlane("plane1",
 ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane1,
 200,200,1,1,true,1,10,10,Vector3::UNIT_Z);

SceneNode* node3 = _sceneManager->createSceneNode("ground");
Entity* grEnt = _sceneManager->createEntity("planeEnt", "plane1");

camM->enableReflection(plane1);
camM->enableCustomNearClipPlane(plane1);

grEnt->setMaterialName("RttMMat");
```



Capítulo 8

Partículas y Billboards

César Mora Castro

Los sistemas de partículas suponen una parte muy importante del impacto visual que produce una aplicación 3D. Sin ellos, las escenas virtuales se compondrían básicamente de geometría sólida y texturas. Los elementos que se modelan utilizando estos tipos de sistemas no suelen tener una forma definida, como fuego, humo, nubes o incluso aureolas simulando ser escudos de fuerza. En las siguientes secciones se van a introducir los fundamentos de los sistemas de partículas y billboards, y cómo se pueden generar utilizando las múltiples características que proporciona Ogre.

8.1. Fundamentos

Para poder comprender cómo funcionan los sistemas de partículas, es necesario conocer primero el concepto de Billboard, ya que dependen directamente de estos. A continuación se define qué son los Billboard, sus tipos y los conceptos matemáticos básicos asociados a estos, para despues continuar con los sistemas de partículas.

8.1.1. Billboards

Billboard significa, literalmente, *valla publicitaria*, haciendo alusión a los grandes carteles que se colocan cerca de las carreteras para anunciar un producto o servicio. Aquellos juegos en los que aparece el nombre de un jugador encima de su personaje, siempre visible a la cámara, utilizan billboards. Los árboles de juegos de carrera que

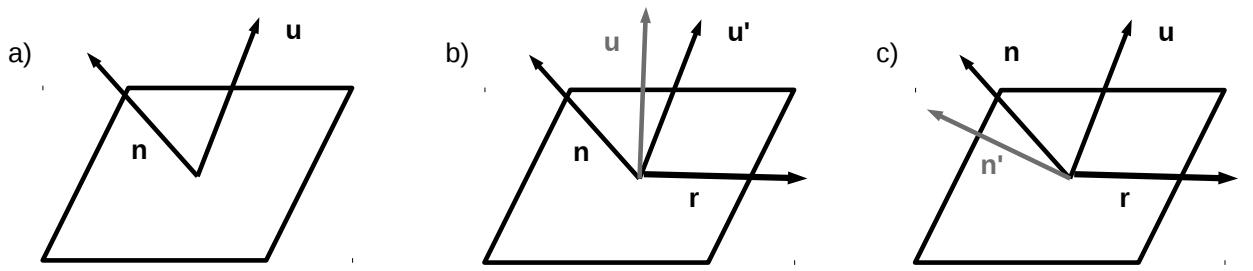


Figura 8.2: En a) se puede observar un Billboard con el vector up y el vector normal. En b) y c) se muestran ejemplos de cómo calcular uno de los vectores dependiendo del otro en el caso de no ser perpendiculares.

daban la sensación de ser un dibujo plano, utilizan Billboards.

Figura 8.1: Ejemplo típico de videojuego que utiliza billboards para modelar la vegetación. Screenshot tomado del videojuego libre Stunt Rally.

Un Billboard es un polígono con una textura, y con un vector de orientación. A medida que la posición de la cámara cambia, la orientación del Billboard cambiará.

Esta técnica se combina con transparencia mediante un canal alfa y con texturas animadas para representar vegetación (especialmente hierba), humo, niebla, explosiones...

Cada billboard está formado por un vector normal $\vec{n}$ y un vector up $\vec{u}$, como se aprecia en la Figura 8.2. Esto es suficiente para definir la orientación del polígono, y poder extraer así la matriz de rotación. El último dato necesario es la *posición* del polígono en el espacio.

Para que el billboard sea correcto, el vector normal y el up deben ser perpendiculares. A menudo no lo son, por lo que hay que utilizar una serie de transformaciones para conseguirlo. El método consiste en tomar uno de los dos vectores como *fijo*, mientras que será el otro el que se recalcule.

Una vez tomado un vector fijo, se calcula el vector $\vec{r}$, resultado de realizar el producto vectorial entre $\vec{u}$ y $\vec{n}$, por lo que será perpendicular a ellos:

$$\vec{r} = \vec{u} \cdot \vec{v}$$

El siguiente paso es normalizarlo, pues se tomará como vector canónico para la matriz de rotación del billboard.

En caso de haber tomado como vector fijo el vector normal $\vec{n}$ (como representa el caso b) de la Figura 8.2), se calcula el nuevo vector $\vec{u'}$ mediante:

$$\vec{u'} = \vec{n} \cdot \vec{r}$$

En el caso de haber sido el vector up $\vec{u}$ el escogido como fijo (caso c) de la Figura 8.2), la ecuación es la que sigue:

$$\vec{n'} = \vec{r} \cdot \vec{u}$$

El nuevo vector es normalizado, y ya se podría obtener la matriz de rotación del billboard. El criterio para escoger un vector como fijo

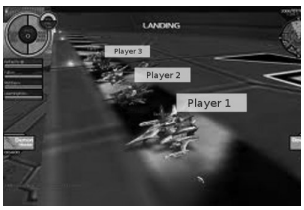


Figura 8.3: *Screenshot* de billboard alineado con la pantalla. Obtenido del videojuego libre Air Rivals.

para calcular la orientación del billboard depende del tipo de este, y del efecto que se quiera obtener. A continuación se explican los tres tipos básicos de billboard existentes.

Billboard alineado a la pantalla

Estos son los tipos más simples de billboard. Su vector up $\vec{u}$ siempre coincide con el de la cámara, mientras que el vector normal $\vec{n}$ se toma como el inverso del vector normal de la cámara (hacia donde la cámara está mirando). Estos dos vectores son siempre perpendiculares, por lo que no es necesario recalcular ninguno con el método anterior. La matriz de rotación de estos billboard es la misma para todos.

Por lo tanto estos billboard siempre estarán alienados con la pantalla, aun cuando la cámara realice giros sobre el eje Z (*roll*), como se puede apreciar en la Figura 8.3. Esta técnica también puede utilizarse para *sprites* circulares como partículas.

Billboard orientado en el espacio

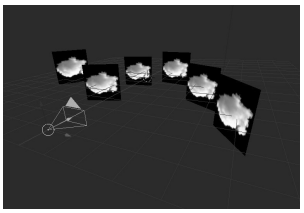


Figura 8.4: Ejemplo de utilización de billboards orientados en el espacio.

En el caso de tratarse de un objeto físico, en el que el vector up debe corresponder con el vector up del mundo, el tipo anterior no es el apropiado. En este caso, el vector up del billboard no es de la cámara, si no el del mundo virtual, mientras que el vector normal sigue siento la inversa del vector hacia donde mira la cámara. En este caso el vector fijo es el normal, mientras que el que se recalcula es el vector up.

Sin embargo, utilizar esta misma matriz de rotación para todos los billboard puede dar lugar a imprecisiones. Según se ha explicado, estos tipos de billboard se mostrarían como en el caso a) de la Figura 8.5. Al estar lo suficientemente cerca de la cámara puede sufrir una distorsión debido a la perspectiva del punto de vista.

La solución a esta distorsión son los billboard orientados al punto de vista. El vector up seguiría siendo el mismo (el vector up del mundo), mientras que el vector normal es el que une el centro del billboard con la posición de la cámara. De esta forma, cada billboard tendría su propia matriz de rotación, y dicha distorsión no existiría. En el caso b) de la Figura 8.5 se aprecia este último tipo.

Evidentemente, este tipo de billboard es menos eficiente, ya que cada uno debe calcular su propia matriz de rotación. Para paliar este aumento de consumo de tiempo de cómputo, podría implementarse dos niveles de billboard dependiendo de la distancia de estos a la cámara. Si están lo suficientemente lejos, la distorsión es mínima y pueden utilizarse los primeros, mientras que a partir de cierta distancia se aplicarían los billboard orientados al punto de vista.

Los billboard orientados en el espacio son muy útiles para la representación de llamas, humo, explosiones o nubes. Una técnica que se suele utilizar es añadir una textura animada a un billboard, y luego crear de forma caótica y aleatoria instancias de este billboard, cambiando parámetros como el tamaño o la transparencia. De esta forma

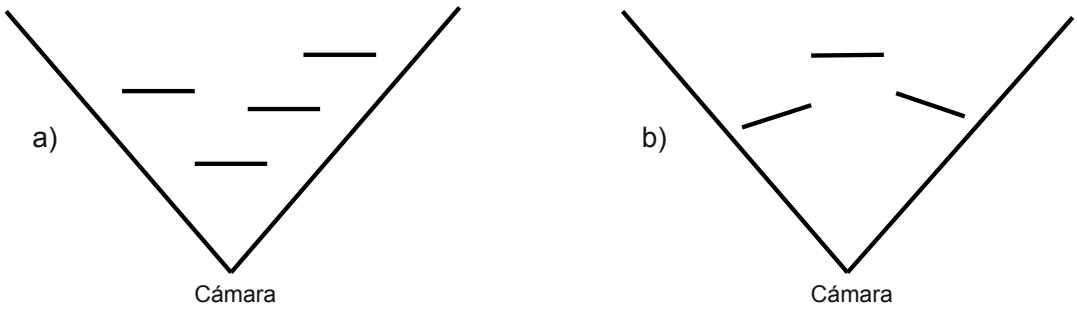


Figura 8.5: Billboards con el mismo vector normal que la cámara en a), mientras que en b) son orientados al punto de vista.

se elimina la sensación de bucle repetitivo en la animación. Este es un método muy común para representar algunos sistemas de partículas. En [WN04] se describe la técnica utilizada para la implementación de las nubes en Microsoft Flight Simulator.

Un inconveniente de esta forma de implementar sistemas de partículas es cuando intersectan con objetos con geometría real. Al realizarse esta intersección se pierde la ilusión. Para ello se implementa un *fade-out*, que consiste en hacer que el billboard sea más transparente a medida que se acerca a cualquier objeto. Este tipo de billboards se denominan *soft particles*.

En [AMH08] se pueden encontrar más técnicas utilizadas para dar más sensación de realismo a estos billboard.

Billboard axial

El último tipo de billboard son los axiales. Estos no miran directamente a la cámara, simplemente giran alrededor de un eje fijo definido, normalmente su vector up. Este técnica suele utilizarse para representar árboles lejanos, como el de la Figura 8.1. En este caso el vector fijo es el vector up, mientras que el recalculado es el normal.

El mayor problema con este tipo de billboard es que si la cámara se sitúa justo encima del billboard (en algún punto de su eje de rotación), la ilusión desaparecía al no mostrarse el billboard. Una posible solución es añadir otro billboard horizontal al eje de rotación. Otra sería utilizar el billboard cuando el modelo esté lo suficientemente lejos, y cambiar a geometría tridimensional cuando se acerque.

8.1.2. Sistemas de partículas

Figura 8.6: Sistema de partículas simulando una llamarada.

Como se ha visto en la sección anterior, con los billboard se pueden representar de forma eficiente y visualmente efectiva muchos tipos de elementos como nubes, humo o llamas. A continuación se va a explicar de forma más concreta en qué consiste un sistema de partículas y qué técnicas se utilizan para implementarlos.

Según [Ree83], un sistema de partículas es un conjunto de pequeños objetos separados en movimiento de acuerdo a algún algoritmo. Su objetivo principal es la simulación de fuego, humo, explosiones, flujos de agua, árboles, etc.

Las tareas típicas de un sistema de partículas incluyen la creación, puesta en movimiento, transformación y eliminado de dichas partículas durante sus diferentes periodos de vida. Sin embargo, la que más nos interesa es la representación de dichas partículas.

Dependiendo del elemento que se quiera representar, las partículas pueden ser mostradas como simples puntos, líneas o incluso billboards. Algunas bibliotecas gráficas como DirectX da soporte para la representación de puntos, y eliminar así la necesidad de crear un billboard con un polígono.

Algunos sistemas de partículas pueden implementarse mediante el *vertex shader*, para calcular la posición de las distintas partículas y así delegar esa parte de cómputo a la GPU. Además puede realizar otras tareas como detección de colisiones.

Elementos de vegetación como hierba o árboles pueden realizarse mediante estas partículas, y determinar la cantidad de estas dependiendo de la distancia de la cámara, todo mediante el *geometry shader*.

A continuación se explican con más detalle dos conceptos básicos utilizados en los sistemas de partículas: los *impostors* y las *nubes de billboards*.

Impostors

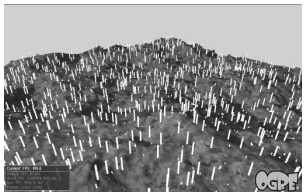


Figura 8.7: Ejemplo de utilización de impostors utilizando Ogre.

Un *impostor* es un billboard cuya textura es renderizada en tiempo de ejecución a partir de la geometría de un objeto más complejo desde la posición actual de la cámara. El proceso de rendering es proporcional al número de píxeles que el impostor ocupa en la pantalla, por lo que es mucho más eficiente. Un buen uso para estos impostors es para representar un elemento que esté compuesto por muchos objetos pequeños iguales, o para objetos muy lejanos. Además, dependiendo de la distancia, la frecuencia con la que se renderizan esos objetos es menor.

Una ventaja importante de los impostors es el poder añadir un desenfoque a la textura de forma rápida para poder dar la sensación de profundidad de campo.

Figura 8.8: Modelado de un helicóptero mediante nube de billboards.

Nubes de Billboards

El problema de los impostors es que estos deben continuar orientadas a la cámara, por lo que si esta hace un movimiento el impostor debe ser renderizado de nuevo. Las nubes de billboards consiste en representar un modelo complejo como un conjunto pequeño de billboards superpuestos. A cada uno de ellos se les puede aplicar un material y propiedades diferentes, para conseguir un buen resultado con un alto rendimiento. En la Figura 8.8 se puede apreciar el modelo geométrico de un helicóptero (izquierda), su correspondiente representación con una nube de billboards (centro) y la descomposición de cada billboard (derecha). Este ejemplo ha sido tomado de [DDSD03].

Algunas aproximaciones propuestas sugieren utilizar parte del modelo como geometría convencional, y para el resto utilizar una nube de billboards. Por ejemplo, hacer el tronco de un arbol geométricamente y añadir billboards para las hojas.

8.2. Uso de Billboards

En Ogre, no existe un elemento billboard por sí sólo que se pueda representar. Estos deben pertenecer a un objeto de la clase *BillboardSet*. Esta clase se encarga de *gestionar* los diferentes billboards que están contenidos en ella.

Todos los *Billboard* que pertenezcan a un *BillboardSet* deben ser idénticos en cuanto a tamaño y material. Este es un requerimiento semi-obligatorio por cuestiones de eficiencia. Una vez añadidos, es posible cambiar los *Billboard* individualmente, aunque esto se desaconseja debido a la penalización en el rendimiento, a no ser que haya una buena razón (por ejemplo, volúmenes de humo que se expanden).

Ogre tratará el *BillboardSet* como un elemento único: o se representan todos los *Billboard* contenidos, o no se representa ninguno. El posicionamiento de los *Billboards* se realiza relativo a la posición del *SceneNode* al que pertenece el *BillboardSet*.

Billboards individuales

También es posible controlar la representación individual de cada *Billboard* dentro de un *BillboardSet*, pero la penalización en rendimiento es mucho mayor. En la mayoría de los casos es más eficiente crear distintos *BillboardSets*.

La forma más común de crear un *BillboardSet* es indicando el número de *Billboard* que son necesarios en el constructor del *BillboardSet*. Cada vez que queramos crear un *Billboard* mediante el método *createBillboard* de *BillboardSet*, se nos devolverá uno. Una vez que se agote la capacidad el método devolverá *NULL*.

En el siguiente código vemos un ejemplo sencillo de una escena con tres billboards:

Listado 8.1: Primer ejemplo con Billboards

```
1  void MyApp::createScene() {
2    Ogre::BillboardSet* billboardSet = _sceneManager->
         createBillboardSet("BillboardSet",3);
3    billboardSet->setMaterialName("Cloud");
4    billboardSet->setDefaultDimensions(10.,10.);
5    billboardSet->setSortingEnabled(true);
6
7    billboardSet->createBillboard(Ogre::Vector3(0,0,0));
8    billboardSet->createBillboard(Ogre::Vector3(50,0,50));
9    billboardSet->createBillboard(Ogre::Vector3(-50,0,-50));
10
11   Ogre::SceneNode* node1 = _sceneManager->createSceneNode("Node1");
12   node1->attachObject(billboardSet);
13   _sceneManager->getRootSceneNode()->addChild(node1);
14 }
```

Capacidad dinámica

Se puede indicar al *BillboardSet* que aumente de forma dinámica su capacidad. Así, cada vez que se pida un Billboard y no quede ninguno, se aumentará la capacidad al doble automáticamente. Este método es potencialmente peligroso, sobretodo si se utiliza en algún tipo de bucle.

Como se ha explicado anteriormente, para poder crear *Billboards* es necesario que estos pertenezcan a un *BillboardSet*. En la línea [2] creamos uno, con nombre *BillboardSet* y con capacidad para tres *Billboard*. Por defecto, el tipo de *Billboard* es *point*. Más adelante se explicarán los tres tipos básicos que ofrece Ogre. En la línea [3] se asigna un material a esos Billboard. En este caso, el material se encuentra descrito en un fichero llamado *Cloud.material*. En la línea [4] se especifica el tamaño del rectángulo que definirán cada uno de los *Billboard*. En la línea [5] se activa la opción de que Ogre ordene automáticamente los *Billboard* según su distancia a la cámara, para que, en caso de que el material tenga transparencia, no se superpongan unos encima de otros y causen un efecto indeseado.

De las líneas [7–9] se crean tantos *Billboard* como capacidad se dió al *BillboardSet* en el momento de su creación. A cada uno de ellos se le indica la posición relativa al *SceneNode* al que pertenece el *BillboardSet*. Por último, en las líneas [11–13] se crea un nodo y se adjunta el *BillboardSet* a él.

8.2.1. Tipos de Billboard

Ogre, por defecto, al crear un *Billboard* lo crea utilizando el tipo *point billboard*. Este tipo se puede cambiar a través del método *BillboardSet::setBillboardType*, y recibe un argumento del tipo *BillboardType*. Los tres tipos básicos existentes son:

- **Point Billboard:** se indican con el valor *BBT_POINT*, y se corresponden con los Billboards alineados en la pantalla. Se trata del tipo más simple, y no es necesario indicar ningún parámetro adicional.
- **Oriented Billboard:** se indican con los valores:
  - *BBT_ORIENTED_COMMON*
  - *BBT_ORIENTED_SELF*

  Se corresponden con los Billboard axiales. Es necesario indicar un eje sobre el cuál girar (en el ejemplo de los árboles, se corresponde con el tronco). En caso de utilizarse la opción *Oriented Common*, este vector se indica mediante el método *BillboardSet::setCommonDirection*, y todos los *Billboard* del conjunto lo utilizarán. En el caso de utilizar *Oriented Self*, cada *Billboard* podrá tener su propio "tronco", y se especifica en la variable pública *Billboard::mDirection* de cada *Billboard*.
- **Perpendicular Billboard:** se indican con los valores:
  - *BBT_PERPENDICULAR_COMMON*
  - *BBT_PERPENDICULAR_SELF*

  Se corresponde con los Billboard orientados en el espacio. Siempre apuntan a la cámara, pero al contrario que con los Billboard alineados en la pantalla, el vector up no tiene por qué ser el mismo que el de ella. En cualquier caso es necesario indicar un vector up mediante la llamada *Billboard::setCommonUpVector*. También se debe indicar un vector de dirección. En caso de haber escogido *Perpendicular Common* se indica mediante la llamada *BillboardSet::setCommonDirection*. En caso de haber escogido *Perpendicular Self*, se almacena en la variable pública *Billboard::mDirection*. Este vector se escoge como fijo, y se recalcula el vector up, según el método explicado en la primera sección. Este vector suele ser el inverso al vector de dirección de la cámara, o el vector con origen en la posición del *Billboard* y destino la posición de la cámara. Es importante no olvidar normalizarlo.

En el caso de los tipos *BBT_PERPENDICULAR_COMMON*, *BBT_PERPENDICULAR_SELF* y *BBT_ORIENTED_SELF* es necesario actualizar los valores de los vectores según la posición actual de la cámara en cada frame. Para ello se debe recuperar los *Billboard* desde el método *frameStarted* del *FrameListener*, y actualizarlos según el valor escogido.

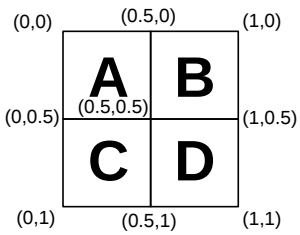


Figura 8.9: Ejemplo de subdivisión de una textura.

8.2.2. Aplicando texturas

Hasta ahora se ha explicado que todos los *Billboard* de un mismo *BillboardSet* deben tener el mismo tamaño y el mismo material. Sin embargo, exite la opción de indicar a cada *Billboard* qué porción de la textura del material puede representar. De esta forma, se puede indicar una textura que esté dividida en filas y en columnas con varias *subtexturas*, y asignar a cada *Billboard* una de ellas.

En la Figura 8.9 se observa un ejemplo de textura subdividida. Las coordenadas están contenidas en el rango de 0 a 1.

En el siguiente código se va a utilizar esta técnica para que, cada uno de los *Billboard*, muestren un trozo de la textura asociada:

Listado 8.2: Ejemplo de coordenadas de texturas.

```
1  void MyApp::createScene() {
2    /* Preparing simbad and ground */
3    /* Creating Sinbad and Ground ...*/
4
5    /* ABC BillboardSet*/
6    Ogre::BillboardSet* abcBillboardSet = _sceneManager->
         createBillboardSet("AbcBillboardSet",4);
7    abcBillboardSet->setBillboardType(Ogre::BBT_POINT);
8    abcBillboardSet->setMaterialName("ABC");
9    abcBillboardSet->setDefaultDimensions(7.,7);
10
11   Ogre::Billboard* b;
12   b = abcBillboardSet->createBillboard(Ogre::Vector3(-15,0,0));
13   b->setTexcoordRect(Ogre::FloatRect(0.,0.,0.5,0.5));
14   b = abcBillboardSet->createBillboard(Ogre::Vector3(-5,0,0));
15   b->setTexcoordRect(Ogre::FloatRect(0.5,0.,1.,0.5));
16   b = abcBillboardSet->createBillboard(Ogre::Vector3(5,0,0));
17   b->setTexcoordRect(Ogre::FloatRect(0.,0.5,0.5,1.));
18   b = abcBillboardSet->createBillboard(Ogre::Vector3(15,0,0));
19   b->setTexcoordRect(Ogre::FloatRect(0.5,0.5,1.,1.));
20
21   Ogre::SceneNode* abcNameNode =_sceneManager->getRootSceneNode()->
         createChildSceneNode("AbcNameNode");
22   abcNameNode->setPosition(0,10,0);
23   abcNameNode->attachObject(abcBillboardSet);
24 }
```

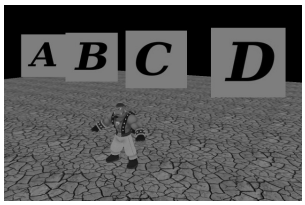


Figura 8.10: Resultado del ejemplo de Billboard con coordenadas de texturas.

En 2-9 se crea el escenario y un *BillboardSet* de tipo *point* y un material asociado llamado “ABC”, como la de la Figura 8.9. De las líneas 11-19 se instancian los 4 *Billboard*. Se ha declarado el puntero a *Billboard b* para poder referenciar cada uno de los *Billboard* según se crean y poder indicar las coordenadas de las texturas asociadas. Esto se hace mediante el método *Billboard::setTexcoordRect*, al cual se le pasa un objeto del tipo *FloatRec*, indicando las coordenadas de la esquina superior izquierda y de la esquina inferior derecha. Los valores de esa esquina están en el rango de 0 a 1. Para terminar, de las líneas 21-23 se crea el nodo y se adjunta el *BillboardSet*.

8.3. Uso de Sistemas de Partículas

Los sistemas de partículas en Ogre se implementan típicamente mediante scripts, aunque cualquier funcionalidad se puede realizar también por código. La extensión de los script que definen las plantillas de estos sistemas es *.particle*. Son plantillas porque en ellas se definen sus características, y luego se pueden instanciar tanto sistemas como se desee. Es decir, se puede crear un fichero *.particle* para definir un tipo de explosión, y luego instanciar cualquier número de ellas.

Los sistemas de partículas son entidades que se enlazan a *SceneNodes*, por lo que están sujetos a la orientación y posicionamiento de estos. Una vez que se han emitido las partículas, estas pasan a formar parte de la escena, por lo que si se mueve el punto de emisión del sistema, las partículas no se verán afectadas, quedando en el mismo sitio. Esto es interesante si se quiere dejar una estela, por ejemplo, de humo. Si se desea que las partículas ya creadas se trasladen con el nodo al que pertenece el sistema, se puede indicar que el posicionamiento se haga referente al sistema de coordenadas local.

Los sistemas de partículas deben tener siempre una cantidad límite de estas, o *quota*. Una vez alcanzado esta cantidad, el sistema dejará de emitir hasta que se eliminen algunas de las partículas *antiguas*. Las partículas tienen un límite de vida configurable para ser eliminadas. Por defecto, este valor de *quota* es 10, por lo que puede interesar al usuario indicar un valor mayor en la plantilla.

Ogre necesita calcular el espacio físico que ocupa un sistema de partículas (su *BoundingBox*) de forma regular. Esto es computacionalmente costoso, por lo que por defecto, deja de recalcularlo pasados 10 segundos. Este comportamiento se puede configurar mediante el método *ParticleSystem::setBoundsAutoUpdated()*, el cual recibe como parámetro los segundos que debe recalcular la *BoundingBox*. Si se conoce de antemano el tamaño aproximado del espacio que ocupa un sistema de partículas, se puede indicar a Ogre que no realice este cálculo, y se le indica el tamaño fijo mediante el método *ParticleSystem::setBounds()*. De esta forma se ahorra mucho tiempo de procesamiento. Se puede alcanzar un compromiso indicando un tamaño inicial aproximado, y luego dejando a Ogre que lo recalcule durante poco tiempo pasados algunos segundos desde la creación del sistema.

Eficiencia ante todo

Los sistemas de partículas pueden rápidamente convertirse en una parte muy agresiva que requiere mucho tiempo de cómputo. Es importante dedicar el tiempo suficiente a optimizarlos, por el bien del rendimiento de la aplicación.

A continuación se describen los dos elementos básicos que definen los sistemas de partículas en Ogre: los *emisores* y los *efectores*.

8.3.1. Emisores

Los emisores definen los *objetos* que literalmente *emiten* las partículas a la escena. Los distintos emisores que proporciona Ogre son:

- **Puntos:** *point*. Todas las partículas son emitidas desde un mismo punto.
- **Caja:** *box*. Las partículas son emitidas desde cualquier punto dentro de un volumen rectangular.
- **Cilindro:** *cylinder*. Las partículas son emitidas desde un volumen cilíndrico definido.
- **Elipsoide:** *ellipsoid*. Las partículas se emiten desde un volumen elipsoidal.
- **Superficie de elipsoide:** *hollow elipsoid*. Las partículas se emiten desde la superficie de un volumen elipsoidal.
- **Anillo:** *ring*. Las partículas son emitidas desde los bordes de un anillo.

La velocidad, frecuencia y dirección de emisión de las partículas es completamente configurable. Estos emisores se posicionan de forma relativa al *SceneNode* al que pertenecen.

Las partículas no son emitidas en una línea recta. Se debe especificar un ángulo mediante el parámetro *angle* para definir el *cono* de emisión. Un valor 0 indica que se emiten en línea recta, mientras que un valor de 180 significa que se emite en cualquier dirección. Un valor de 90 implica que se emiten de forma aleatoria en el hemisferio centrado en el vector de dirección.

Otros parámetros que se pueden configurar son la frecuencia de emisión (partículas/segundo), la velocidad (puede ser una velocidad establecida o aleatoria para cada partícula), el tiempo de vida o *TTL* (definido o aleatorio), y el tiempo de emisión del sistema. Más adelante se mostrarán ejemplos de uso de estos parámetros.

8.3.2. Efectores

Los efectores o *affectors* realizan cambios sobre los sistemas de partículas. Estos cambios pueden ser en su dirección, tiempo de vida, color, etc. A continuación se explican cada uno de los efectores que ofrece Ogre.

- **LinearForce:** aplica una *fuerza* a las partículas del sistema. Esta fuerza se indica mediante un vector, cuya dirección equivale a la dirección de la fuerza, y su módulo equivale a la magnitud de la fuerza. La aplicación de una fuerza puede resultar en un incremento enorme de la velocidad, por lo que se dispone de

un parámetro, *force_application* para controlar esto. El valor *force_application average* ajusta el valor de la fuerza para estabilizar la velocidad de las partículas a la media entre la magnitud de la fuerza y la velocidad actual de estas. Por el contrario, *force_application add* deja que la velocidad aumente o se reduzca sin control.

- **ColourFader:** modifica el color de una partícula mientras ésta exista. El valor suministrado a este modificador significa ‘ ‘ cantidad de cambio de una componente de color en función del tiempo”. Por lo tanto, un valor de *red -0.5* decrementará la componente del color rojo en 0.5 cada segundo.
- **ColourFader2:** es similar a *ColourFader*, excepto que el modificador puede cambiar de comportamiento pasada una determinada cantidad de tiempo. Por ejemplo, el color de una partícula puede decrementarse suavemente hasta el 50% de su valor, y luego caer en picado hasta 0.
- **ColourInterpolator:** es similar a *ColourFader2*, sólo que se pueden especificar hasta 6 cambios de comportamiento distintos. Se puede ver como una generalización de los otros dos modificadores.
- **Scaler:** este modificador cambia de forma proporcional el tamaño de la partícula en función del tiempo.
- **Rotator:** rota la textura de la partícula por bien un ángulo aleatorio, o a una velocidad aleatoria. Estos dos parámetros son definidos dentro de un rango (por defecto 0).
- **ColourImage:** este modificador cambia el color de una partícula, pero estos valores se toman de un fichero imagen (con extensión .png, .jpg, etc.). Los valores de los píxeles se leen de arriba a abajo y de izquierda a derecha. Por lo tanto, el valor de la esquina de arriba a la izquierda será el color inicial, y el de abajo a la derecha el final.

ColourFader

Un valor de -0.5 no quiere decir que se reduzca a la mitad cada segundo, y por lo tanto, nunca alcanzará el valor de 0. Significa de al valor de la componente (perteneciente al intervalo de 0 a 1) se le restará 0.5. Por lo tanto a un valor de blanco (1) se reducirá a negro (0) en dos segundos.

8.3.3. Ejemplos de Sistemas de Partículas

Para poder utilizar los sistemas de partículas en Ogre, es necesario editar el fichero *plugins.cfg* y añadir la línea *Plugin=Plugin_ParticleFX* para que pueda encontrar el plugin. En caso de utilizarse en Windows, hay que asegurarse de tener la biblioteca *Plugin_ParticleFX.dll* en el directorio de plugins del proyecto, o la biblioteca *Plugin_ParticleFX.so* en caso de sistemas UNIX

Este primer ejemplo ilustra un anillo de fuego. A continuación se explica paso a paso cómo se instancia el sistema de partículas y qué significan cada uno de los campos del script que lo define.

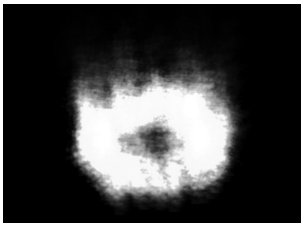


Figura 8.11: Captura de pantalla del sistema de partículas *RingOfFire*

Listado 8.3: Instancia de un sistema de partículas.

```
1 void MyApp::createScene() {
2   Ogre::ParticleSystem* ps = _sceneManager->createParticleSystem("
        Ps","ringOfFire");
3
4   Ogre::SceneNode* psNode = _sceneManager->getRootSceneNode()->
        createChildSceneNode("PsNode");
5   psNode->attachObject(ps);
6 }
```

Como se puede observar, crear un sistema de partículas en código no es nada complicado. En la línea [2] se crea un objeto de tipo *ParticleSystem*, indicando su nombre y el nombre del script que lo define, en este caso *ringOfFire*. En las líneas [4-5] se crea un *SceneNode* y se añade a él.

El siguiente es el script que define realmente las propiedades del sistema de partículas.

Listado 8.4: Script para el sistema *ringOfFire*

```
1  particle_system ringOfFire
2  {
3      quota   1000
4      material    explosion
5      particle_width  10
6      particle_height 10
7
8      emitter Ring
9      {
10         angle   10
11         direction   0 1 0
12         emission_rate   250
13         velocity_min    3
14         velocity_max    11
15         time_to_live    3
16         width   30
17         height  30
18         depth   2
19     }
20
21     affector ColourFader
22     {
23         red -0.5
24         green   -0.5
25         blue    -0.25
26     }
27 }
```

Los scripts de sistemas de partículas comienzan con la palabra reservada *particle_system* seguido del nombre del sistema, como se indica en la línea [1]. De las líneas [3-6] se indican varios parámetros generales del sistema:

- *quota:* Indica el número máximo de partículas que pueden haber vivas en un momento. Si el número de partículas alcanza esta cantidad, no se crearán más partículas hasta que mueran otras.

- *material:* indica el material de cada partícula. Cada una de las partículas es, por defecto, un *Billboard* como los estudiados anteriormente. Este material indica qué textura se representará en cada uno de ellos.
- *particle_width:* indica el ancho de cada partícula.
- *particle_height:* indica el alto de cara partícula.

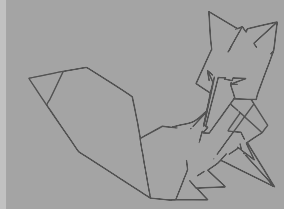
A continuación se declaran tantos emisores y modificadores como se deseen. En las líneas 8–19 se declara un emisor del tipo anillo. Los parámetros especificados para este emisor son:

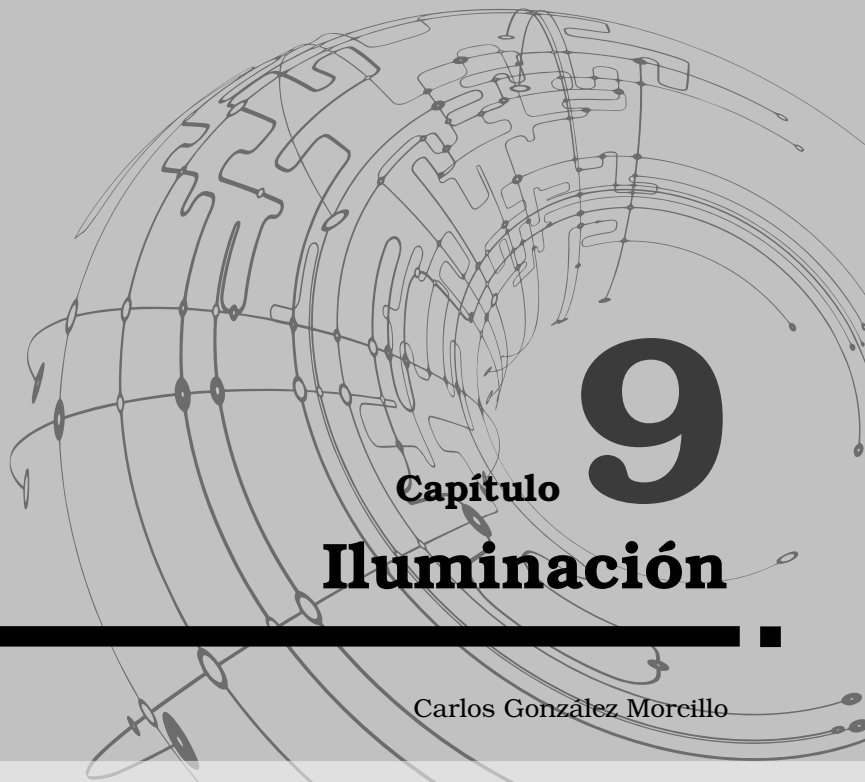
¡Y aún hay más!

Existen multitud de parámetros para configurar los sistemas de partículas, emisores y modificadores en Ogre. En la API oficial se detallan todos y cada uno de estos parámetros.

- *angle:* define el ángulo de apertura con el que las partículas salen disparadas.
- *direction:* indica la dirección de salida d e las partículas, teniendo en cuenta el ángulo. En realidad, el par dirección-ángulo define un cono por el cual las partículas se crean.
- *emission_rate:* indica el ratio de partículas por segundo emitidas.
- *velocity_min:* velocidad mínima inicial de las partíclas.
- *velocity_max:* velocidad máxima inicial de las partículas.
- *time_to_live* tiempo de vida de una partícula.
- *width:* determina el ancho del anillo.
- *height:* determina el alto del anillo.
- *depth:* determina la profundidad del anillo.

Además se ha declarado un modificado del tipo *ColourFader* que cambia el color de las partículas en el tiempo. Concretamente, por cada segundo que pasa decrementa la componente del color rojo en 0.5, la del color verde en 0.5 y la del color azul en 0.25.





Capítulo 9

Iluminación

Carlos González Morcillo

En el capítulo anterior hemos estudiado algunas de las características relativas a la definición de materiales básicos. Estas propiedades están directamente relacionadas con el modelo de iluminación y la simulación de las fuentes de luz que realicemos. Este capítulo introduce algunos conceptos generales sobre iluminación en videojuegos, así como técnicas ampliamente utilizadas para la simulación de la iluminación global.

9.1. Introducción

Una pequeña parte de los rayos de luz son visibles al ojo humano. Aquellos rayos que están definidos por una onda con longitud de onda λ entre 700 y 400nm. Variando las longitudes de onda obtenemos diversos colores.

En gráficos por computador es habitual emplear los denominados *colores-luz*, donde el Rojo, Verde y Azul son los colores primarios y el resto se obtienen de su combinación. En los *colores-luz* el color blanco se obtiene como la suma de los tres colores básicos.

El RGB es un modelo clásico de este tipo. En el mundo físico real se trabaja con *colores-pigmento*, donde el Cyan, el Magenta y el Amarillo forman los colores primerarios. La combinación de igual cantidad de los tres colores primarios obtiene el color negro[1]. Así, el CMYK empleado por las impresoras es un clásico modelo de este tipo. De un modo

[1] En realidad se obtiene un tono *parduzco*, por lo que habitualmente es necesario incorporar el negro como color primario.

simplificado, podemos definir los pasos más relevantes para realizar la representación de una escena sintética:

1. La luz es emitida por las fuentes de luz de la escena (como el sol, una lámpara de luz situada encima de la mesa, o un panel luminoso en el techo de la habitación).
2. Los rayos de luz interactúan con los objetos de la escena. Dependiendo de las propiedades del material de estos objetos, parte de la luz será absorbida y otra parte reflejada y propagada en diversas direcciones. Todos los rayos que no son totalmente absorbidos continuarán rebotando en el entorno.
3. Finalmente algunos rayos de luz serán capturados por un sensor (como un ojo humano, el sensor CCD de una cámara digital o una película fotográfica).

La luz puede ser modelada empleando diversas aproximaciones, centrándose en las propiedades direccionales (rayos puramente geométricos), como ondas electromagnéticas o como partículas cuánticas (fotones). Dependiendo del método de representación, suele emplearse un modelo u otro. Independientemente del tratamiento que demos a la luz, ésta debe ser simulada como energía que *viaja* en el espacio. Las fuentes de luz serán *emisores* de esta energía.

Directamente asociado al concepto de iluminación encontramos las sombras. Gracias a la proyección de sombras, podemos establecer relaciones espaciales entre los objetos de la escena. Por ejemplo, en la Figura 9.1, gracias al uso de sombras podemos saber la posición exacta de la esfera relativa a la escalera.

A continuación estudiaremos los principales tipos de luz que suelen emplearse en videojuegos, así como los modelos de sombreado estáticos y dinámicos más utilizados.

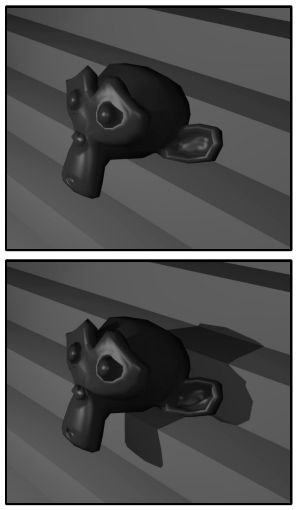


Figura 9.1: Gracias a la proyección de sombras es posible conocer la posición relativa entre objetos. En la imagen superior no es posible determinar si el modelo de *Suzanne* reposa sobre algún escalón o está flotando en el aire. La imagen inferior, gracias al uso de sombras elimina esa ambigüedad visual.

Simplifica!!

Como ya comentamos en el Capítulo 7, este modelo de iluminación es una simplificación del modelo físicamente correcto que se resuelve con mejores aproximaciones de la ecuación de Rendering de James Kajiya.

9.2. Tipos de Fuentes de Luz

Las fuentes de luz pueden representarse de diversas formas, dependiendo de las características que queramos simular en la etapa de rendering.

Para especificar la *cantidad* de energía emitida por una fuente de luz, la *radiometría* (ciencia que se encarga de medir la luz) define la *irradiancia* como la cantidad de fotones que pasan por una superficie por segundo.

En videojuegos suelen permitirse tres tipos de fuentes de luz directamente soportadas por el hardware de aceleración gráfico:

- Las **fuentes puntuales** (*point lights*) irradian energía en todas las direcciones a partir de un punto que define su posición en el espacio. Este tipo de fuentes permite variar su posición pero no su dirección. En realidad, las fuentes de luz puntuales no

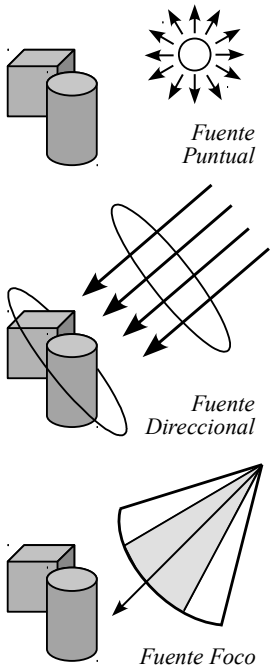


Figura 9.2: Tipos de fuentes de luz utilizadas en aplicaciones interactivas. Ogre soporta únicamente estos tres tipos de fuentes básicos.

existen como tal en el mundo físico (cualquier fuente de luz tiene asociada un área, por lo que para realizar simulaciones *realistas* de la iluminación tendremos que trabajar con fuentes de *área*. Ogre define este tipo de fuente como `LT_POINT`.

- Uno de los tipos de fuentes más sencillo de simular son las denominadas **fuentes direccionales** (*directional lights*), que pueden considerarse fuentes situadas a una distancia muy grande, por lo que los rayos viajan en una única dirección en la escena (son paralelos entre sí). El *sol* podría ser modelado mediante una fuente de luz direccional. De este modo, la dirección viene determinada por un vector l (especificado en coordenadas universales). Este tipo de fuentes de luz no tienen por tanto una posición asociada (únicamente dirección). Este tipo de fuente está descrito como `LT_DIRECTIONAL` en Ogre.

- Finalmente los **focos** (*spot lights*) son en cierto modo similares a las fuentes de luz puntuales, pero añadiendo una dirección de emisión. Los focos arrojan luz en forma cónica o piramidal en una dirección específica. De este modo, requieren un parámetro de dirección, además de dos ángulos para definir los conos de emisión interno y externo. Ogre las define como `LT_SPOTLIGHT`.

Las fuentes de luz permiten especificar multitud de parámetros y propiedades, como el color difuso y especular. Una fuente de luz puede definir un color de iluminación difuso (como si el cristal de la *bombilla* estuviera tintado), y un color diferente para el brillo especular. Ambas propiedades están directamente relacionadas con el modo en el que reflejarán la luz los materiales.

En aplicaciones de síntesis de imagen realista suelen definirse además fuentes de luz de área. Este tipo de fuentes simulan el comportamiento de la luz de un modo más realista, donde potencialmente cada punto de la superficie se comporta como un emisor de luz. En la sección 9.7 estudiaremos el modelo de *Radiosidad* que permite trabajar con fuentes de luz de área.

9.3. Sombras Estáticas Vs Dinámicas

La gestión de sombras es un aspecto crítico para dotar de realismo a las escenas sintéticas. Sin embargo, el cálculo de sombras trae asociado un coste computacional importante. De esta forma, en multitud de ocasiones se emplean técnicas de *pre-cálculo* de la iluminación y las sombras. Estos mapas de iluminación permiten generar sombras suaves sin coste computacional adicional.

Ogre soporta dos técnicas básicas de sombreado dinámico: sombreado empleando el *Stencil Buffer* y mediante *Mapas de Texturas* (ver Figura 9.4). Ambas aproximaciones admiten la especificación como *Additive* o *Modulative*.

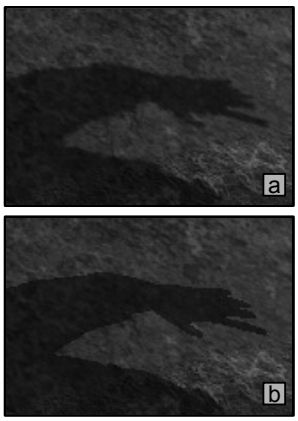


Figura 9.3: Comparativa entre sombras calculadas por **a)** *Mapas de Texturas* y **b)** *Stencil Buffer*. Las basadas en mapas de texturas son claramente dependientes de la resolución del mapa, por lo que deben evitarse con áreas de proyección muy extensas.

En muchas ocasiones se implementan, empleando el Pipeline en GPU programable, algoritmos de sombreado avanzados como aproximaciones al Ambient Occlusion (que veremos en la sección 9.6 o Radiosidad Instantánea (ver sección 9.7).

Las técnicas de tipo *Modulative* únicamente oscurecen las zonas que quedan en sombra, sin importar el número de fuentes de luz de la escena. Por su parte, si la técnica se especifica de tipo *Additive* es necesario realizar el cálculo por cada fuente de luz de modo acumulativo, obteniendo un resultado más preciso (pero más costoso computacionalmente).

Cada técnica tiene sus ventajas e inconvenientes (como por ejemplo, el relativo a la resolución en el caso de los Mapas de Textura, como se muestra en la Figura 9.3). No es posible utilizar varias técnicas a la vez, por lo que deberemos elegir la técnica que mejor se ajuste a las características de la escena que queremos representar. En términos generales, los métodos basados en mapas de textura suelen ser más precisos, pero requieren el uso de tarjetas aceleradoras más potentes. Veamos las características generales de las técnicas de sombreado para estudiar a continuación las características particulares de cada método.

- Únicamente podemos utilizar una técnica de sombreado en la escena. Esta técnica debe ser especificada preferiblemente *antes* de especificar los objetos que formen parte de la escena.
- Las propiedades del material determinan si el objeto arrojará o recibirá sombra. Por defecto los objetos arrojan sombra (salvo los objetos con materiales transparentes, que no arrojarán sombra).
- Es posible definir fuentes de luz que no arrojen sombras.
- En ambos casos es conveniente evitar que las sombras se proyecten de forma extrema (por ejemplo, simulando un amanacer). Este tipo de situaciones hace que la calidad de la sombra se degrade enormemente.
- Dado su coste computacional, por defecto las sombras están *desactivadas* en Ogre.
- Para que un material reciba o arroje sombras, el parámetro *lighting* del material debe estar en *on* (por defecto).

A continuación estudiaremos los detalles de ambos tipos de sombras soportados en Ogre.

9.3.1. Sombras basadas en Stencil Buffer

Empleando esta técnica de sombreado, la forma de la sombra que será proyectada se obtiene proyectando la silueta del objeto calculada desde la perspectiva de la fuente de luz.

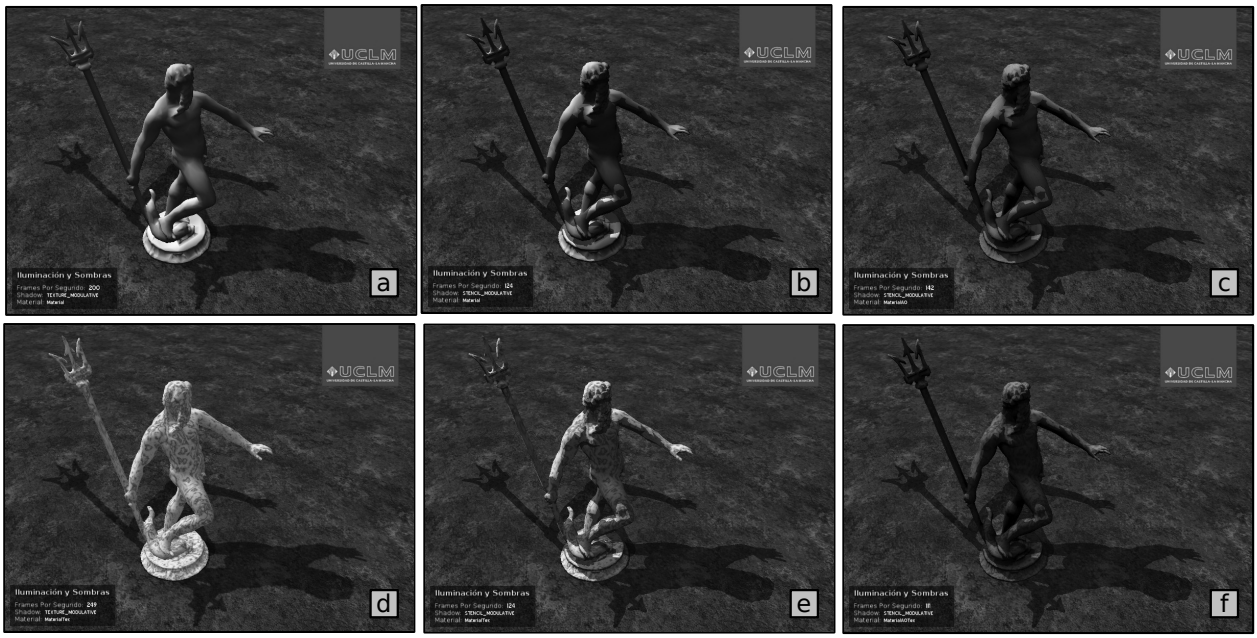


Figura 9.4: Utilización de diversos modos de cálculo de sombras y materiales asociados. **a)** Sombras mediante Mapas de Texturas y Material simple. **b)** Sombras por Stencil Buffer y Material simple. **c)** Sombras por Stencil Buffer y Material con Ambient Occlusion precalculado. **d)** Sombras mediante Mapas de Texturas y Material basado en textura de mármol *procedural* precalculada. **e)** Sombras por Stencil Buffer y Material basado en textura de mármol *procedural* precalculada. **f)** Sombras por Stencil Buffer y Material combinado de c) + e) (Ambient Occlusion y Textura de Mármol precalculados).

El *Stencil Buffer* es un buffer extra disponible en las GPUs modernas que permite almacenar un byte por píxel. Habitualmente se emplea para *recortar* el área de renderizado de una escena. En el cálculo de sombras, se utiliza en combinación con el *ZBuffer* para recortar la zona de sombra relativa al punto de vista. El proceso de cálculo puede resumirse en los siguientes pasos (ver Figura 9.5):

1. Para cada fuente de luz, obtener la lista de aristas de cada objeto que comparten polígonos cuyo vector normal apunta "*hacia*" la fuente de luz y las que están "*opuestas*" a la misma. Estas aristas definen la *silueta* del objeto desde el punto de vista de la fuente de luz.

2. Proyectar estas *aristas de silueta* desde la fuente de luz hacia la escena. Obtener el volumen de sombra proyectado sobre los objetos de la escena.

3. Finalmente, utilizar la información de profundidad de la escena (desde el punto de vista de la cámara virtual) para definir en el *Stencil Buffer* la zona que debe recortarse de la sombra (aquella cuya profundidad desde la cámara sea mayor que la definida en el *ZBuffer*). La Figura 9.5 muestra cómo la proyección del volumen de sombra es recortado para aquellos puntos cuya profundidad es menor que la relativa a la proyección del volumen de sombra.

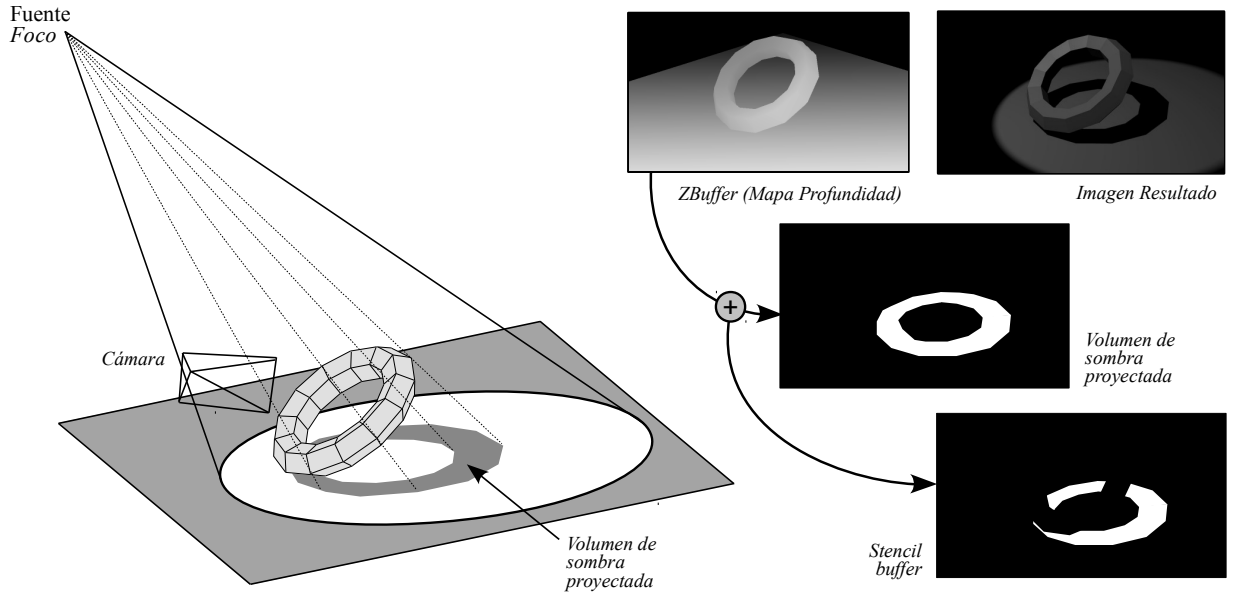


Figura 9.5: Proceso de utilización del *Stencil Buffer* para el recorte de la proyección del volumen de sombra.

Esta técnica de generación de sombras, a diferencia de las basadas en Mapas de Textura, utiliza ciclos de la CPU para el renderizado (las operaciones relativas al cálculo de aristas y proyección sobre la escena 3D). A continuación se describen algunas de las características fundamentales de las sombras basadas en Stencil Buffer.

- Empleando el Stencil Buffer, las sombras de objetos transparentes emitirán sombras totalmente sólidas. Es posible desactivar totalmente las sombras de este tipo de objetos, pero empleando Stencil Buffer no es posible obtener sombras semitransparentes.
- Empleando esta técnica no es posible obtener sombras con aristas suaves. En el caso de necesitar este tipo de sombras será necesario emplear la técnica basada de Mapas de Textura (ver Sección 9.3.2).
- Esta técnica permite que el objeto reciba su propia sombra, como se muestra en la Figura 9.6.b.
- Es necesario que Ogre conozca el conjunto de aristas que definen el modelo. Exportando desde Blender, la utilidad *OgreXMLConverter* realiza el cálculo automáticamente. Sin embargo, si implementamos nuestro propio cargador, deberemos llamar a *buildEdgeList* de la clase *Mesh* para que construya la lista.

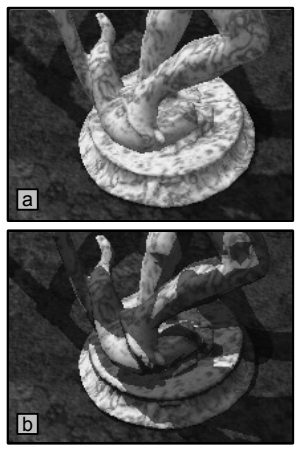


Figura 9.6: En el cálculo de sombras mediante Mapas de Textura (en **a)**), el objeto emisor de sombras no recibe su propia sombra proyectada. Empleando el Stencil Buffer (**b)**) es posible recibir la sombra proyectada.

9.3.2. Sombras basadas en Texturas

El cálculo de las sombras basadas en texturas se basa en un simple principio: si observamos una escena desde el punto de vista de

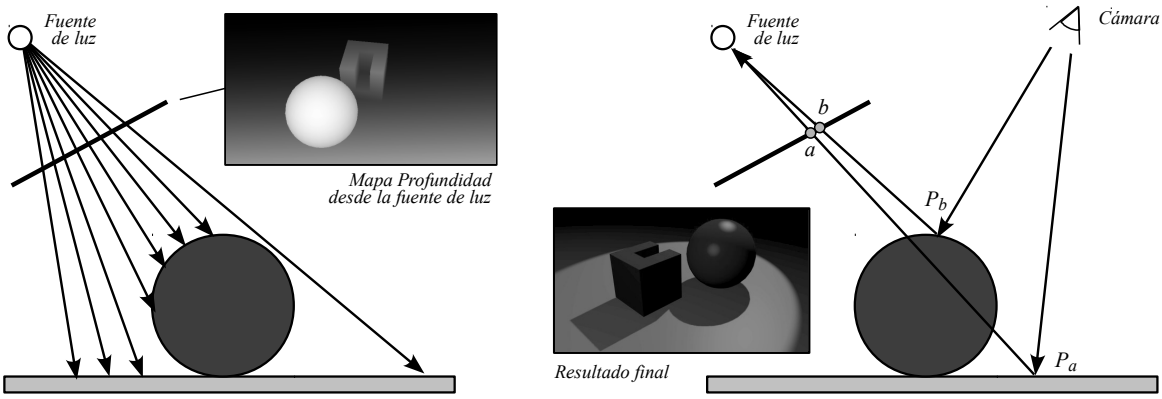


Figura 9.7: Pasos en el cálculo de sombras mediante mapas de textura.

la fuente de luz, cualquier punto situado *detrás* de lo que ve la fuente de luz estará en sombra. Esta idea puede realizarse en dos pasos principales empleando texturas calculadas directamente en la GPU.

En el primer paso se construye el mapa de profundidad desde el punto de vista. Este mapa simplemente codifica la distancia menor de los objetos de la escena a la fuente de luz (como se muestra en la parte izquierda de la Figura 9.7).

El segundo paso utiliza esta información para construir la sombra. El algoritmo calcula la distancia del objeto a la cámara y compara esta distancia con la codificada en el mapa de profundidad de la fuente de luz. Si la distancia entre cada punto y la fuente de luz es mayor que la almacenada en el mapa de profundidad, el punto está en sombra. (ver Figura 9.7 derecha).

Empleando sombras basadas en texturas el objeto debe ser definido como receptor o emisor de sombras. Un objeto no puede ser emisor y receptor de sombras a la vez (por lo que un emisor no puede recibir sus propias sombras).

A continuación enumeraremos algunas de las características fundamentales de este tipo de técnica.

Z-Fighting

La distinción en grupos de emisores y receptores de sombras se crea para evitar problemas de *Z-fighting* (cuando dos o más elementos tienen asociada la misma profundidad en el ZBuffer y su representación es incorrecta).

- Este tipo de sombras permiten el manejo correcto de la transparencia. Además, las sombras pueden tener un color propio.
- Se basan principalmente en el uso de la GPU, por lo que descargan en gran medida la CPU. Directamente relacionado con esta característica, las sombras basadas en mapas de textura permiten componer otros efectos en la GPU (como deformaciones empleando un *Vertex Shader*).
- Esta técnica no permite que el objeto reciba sus propias sombras.

9.4. Ejemplo de uso

El siguiente ejemplo de uso construye una aplicación que permite elegir entre el uso de texturas basadas en *Stencil Buffer* o en *Mapas de Textura*.

Listado 9.1: Fragmento de MyApp.cpp

```
1  void MyApp::createScene() {
2    _sceneManager->setShadowTechnique(SHADOWTYPE_STENCIL_MODULATIVE);
3    _sceneManager->setShadowColour(ColourValue(0.5, 0.5, 0.5));
4    _sceneManager->setAmbientLight(ColourValue(0.9, 0.9, 0.9));
5
6    _sceneManager->setShadowTextureCount(2);
7    _sceneManager->setShadowTextureSize(512);
8
9    Light* light = _sceneManager->createLight("Light1");
10   light->setPosition(-5,12,2);
11   light->setType(Light::LT_SPOTLIGHT);
12   light->setDirection(Vector3(1,-1,0));
13   light->setSpotlightInnerAngle(Degree(25.0f));
14   light->setSpotlightOuterAngle(Degree(60.0f));
15   light->setSpotlightFalloff(0.0f);
16   light->setCastShadows(true);
17
18   Light* light2 = _sceneManager->createLight("Light2");
19   light2->setPosition(3,12,3);
20   light2->setDiffuseColour(0.2,0.2,0.2);
21   light2->setType(Light::LT_SPOTLIGHT);
22   light2->setDirection(Vector3(-0.3,-1,0));
23   light2->setSpotlightInnerAngle(Degree(25.0f));
24   light2->setSpotlightOuterAngle(Degree(60.0f));
25   light2->setSpotlightFalloff(5.0f);
26   light2->setCastShadows(true);
27
28   Entity* ent1 = _sceneManager->createEntity("Neptuno.mesh");
29   SceneNode* node1 = _sceneManager->createSceneNode("Neptuno");
30   ent1->setCastShadows(true);
31   node1->attachObject(ent1);
32   _sceneManager->getRootSceneNode()->addChild(node1);
33
34   // ... Creamos plano1 manualmente (codigo eliminado)
35
36   SceneNode* node2 = _sceneManager->createSceneNode("ground");
37   Entity* groundEnt = _sceneManager->createEntity("p", "plane1");
38   groundEnt->setMaterialName("Ground");
39   groundEnt->setCastShadows(false);
40   node2->attachObject(groundEnt);
41   _sceneManager->getRootSceneNode()->addChild(node2);
42 }
```

En la línea [2] se define la técnica de cálculo de sombras que se utilizará por defecto, aunque en el *FrameListener* se cambiará en tiempo de ejecución empleando las teclas [1] y [2]. Las líneas [2-3] definen propiedades generales de la escena, como el color con el que se representarán las sombras y el color de la luz ambiental (que en este ejemplo, debido a la definición de los materiales, tendrá especial importancia).

Las líneas [6-9] únicamente tienen relevancia en el caso del uso del método de cálculo basado en mapas de textura. Si el método utiliza-

do es el de *Stencil Buffer* simplemente serán ignoradas. La línea 6 configura el número de texturas que se emplearán para calcular las sombras. Si se dispone de más de una fuente de luz (como en este ejemplo), habrá que especificar el número de texturas a utilizar.

El tamaño de la textura (cuadrada) se indica en la línea 7. A mayor tamaño, mejor resolución en la sombra (pero mayor cantidad de memoria gastada). El tamaño por defecto es 512, y *debe ser potencia de dos*.

A continuación en las líneas 9–26 se definen dos fuente de luz de tipo *SPOTLIGHT*. Cada luz define su posición y rotación. En el caso de la segunda fuente de luz se define además un color difuso (línea 20). El ángulo interno y externo 13–14 define el cono de iluminación de la fuente. En ambas fuentes se ha activado la propiedad de que la fuente de luz permita el cálculo de sombras.

```
material MaterialAOTex
{
 receive_shadows on
 technique
 {
   pass
   {
     texture_unit
     {
     texture
       neptuno_tex.jpg
     }
     texture_unit
     {
     texture
       neptuno_lm.jpg
       colour_op_ex
         modulate
         src_texture
         src_current
     }
    }
  }
}
```

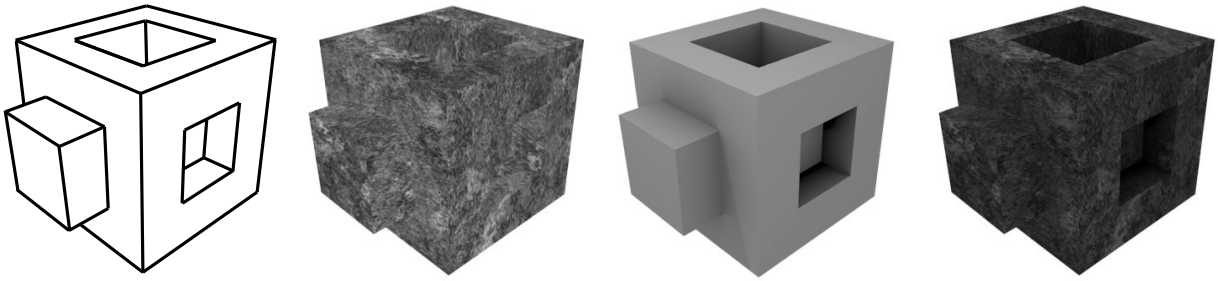
Figura 9.8: Definición del material multicapa para componer una textura con iluminación de tipo AbmientOcclusion con la textura de color. Ambas texturas emplean un despliegue automático (tipo *LightMap* en Blender).

El método de *setCastShadows* pertenece a la clase *MovableObject*. En el caso de entidades, si se especifica como cierto, el objeto arrojará sombras sobre otros objetos. En caso contrario, funcionará como un receptor de sombras. El mismo método se aplica sobre luces (que, es un tipo específico de *MovableObject*). En el caso de fuentes de luz, el mismo método sirve para especificar si esa fuente de luz se utilizará en el cálculo de sombras.

Como hemos comentado en la sección 9.3.2, si utilizamos mapas de textura para calcular sombras, un objeto puede funcionar únicamente como receptor o como emisor de sombras. En este ejemplo, el plano se configura como receptor de sombras (en la línea 39), mientras que el objeto *Neptuno* funciona como emisor (línea 30).

Mediante las teclas 7 ... 0 es posible cambiar el material asociado al objeto principal de la escena. En el caso del último material definido para el objeto (tecla 0), se compone en dos capas de textura el color base (calculado mediante una textura procedural) y una capa de iluminación basada en Ambient Occlusion. La definición del material compuesto se muestra en la Figura 9.8. Gracias a la separación de la iluminación y del color en diferentes mapas, es posible cambiar la resolución individualmente de cada uno de ellos, o reutilizar el mapa de iluminación en diferentes modelos que compartan el mismo mapa de iluminación pero tengan diferente mapa de color.

Multitud de juegos utilizan el precálculo de la iluminación. Quake II y III utilizaron modelos de Radiosidad para crear mapas de iluminación para simular de una forma mucho más realista el comportamiento físico de la luz en la escena. Aunque actualmente poco a poco se implantan las técnicas de iluminación dinámicas a nivel de píxel, los mapas de iluminación siguen siendo una opción ampliamente utilizada.



Geometría base | *Mapa de color* | *Mapa de Iluminación (Ambient Occlusion)* | *Resultado Final*

Figura 9.9: Ejemplo de uso de un mapa de iluminación para definir sombras suaves en un objeto. El mapa de color y el mapa de iluminación son texturas independiente que se combinan (multiplicando su valor) en la etapa final.

9.5. Mapas de Iluminación

Como hemos visto anteriormente, los modelos de iluminación local calculan en cada vértice del modelo la interacción con la luz. El color final de cada punto del modelo se calcula mediante técnicas de interpolación. Las sombras obtenidas empleando estos métodos no son precisas (por ejemplo, no se pueden calcular sombras difusas).

En los últimos años las tarjetas gráficas permiten el cálculo de métodos de iluminación por píxel (*per pixel lighting*), de modo que por cada píxel de la escena se calcula la contribución *real* de la iluminación. El cálculo preciso de las sombras es posible, aunque a día de hoy todavía es muy costoso para videojuegos.

Los mapas de luz permiten precalcular la iluminación por píxel de forma estática. Esta iluminación precalculada puede ser combinada sin ningún problema con los métodos de iluminación dinámicos estudiados hasta el momento. La calidad final de la iluminación es únicamente dependiente del tiempo invertido en el precálculo de la misma y la resolución de las texturas.

En términos generales, el píxel de una textura se denomina *texel* (de *Texture Element*). De forma análoga, un píxel de un mapa de iluminación se denomina *lumel* (*Lumination Element*).

De este modo, para cada cara poligonal del modelo se definen una o varias capas de textura. La capa del mapa de iluminación es finalmente *multiplicada* con el resultado de las capas anteriores (como se puede ver en la Figura 9.9).

A diferencia de las texturas de color donde cada vértice del modelo puede compartir las mismas coordenadas UV con otros vértices, en mapas de iluminación cada vértice de cada cara *debe* tener una coor-

denada única. Los mapas de iluminación se cargan de la misma forma que el resto de texturas de la escena. En la Figura 9.8 hemos estudiado un modo sencillo de composición de estos mapas de iluminación, aunque pueden definirse otros métodos que utilicen varias pasadas y realicen otros modos de composición. A continuación estudiaremos dos técnicas ampliamente utilizadas en el precálculo de la iluminación empleando mapas de iluminación: Ambient Occlusion y Radiosidad.

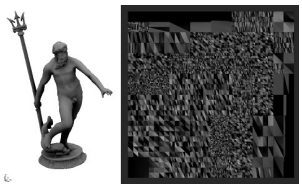


Figura 9.10: Despliegue del modelo de Neptuno empleando *Lightmap UVPack*. Cada cara poligonal tiene asociado un espacio propio en la textura.

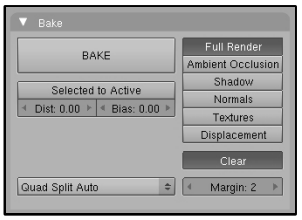


Figura 9.11: Opciones de la pestaña Bake del grupo de botones de Render.

> Aunque nos centremos en los métodos de Ambient Occlusion y Radiosidad, Blender permite precalcular (*Render Baking*) cualquier configuración de escena. De este modo, se puede precalcular cualquier configuración de luces y materiales de la escena.

Como hemos comentado antes, cualquiera de los métodos de cálculo de mapas de iluminación requiere que cada cara poligional del modelo tenga unas coordenadas UV únicas en el modelo. Como estudiamos en la sección 7.5, empleando el despliegue de tipo *Lightmap UVPack* conseguimos que Blender despliegue el modelo de esta forma, asignando el área de la imagen de forma proporcional al tamaño de cada cara del modelo. La Figura 9.10 muestra el despliegue realizado para el modelo de Neptuno tras aplicar Ambient Occlusion.

Una vez realizado el despliegue y con una imagen asociada al despiegue UV (puede crearse una imagen *nueva* vacía de color sólido desde la cabecera de la ventana UV Image Editor en el menú *Image/New*), el precálculo de la iluminación se realiza en la pestaña *Bake* del grupo de botones de Render (ver Figura 9.11).

A continuación se estudiarán las opciones más relevantes en el ámbito del precálculo de la iluminación.

Mediante el grupo de botones de la derecha, se elige el tipo de precálculo que va a realizarse. Con *Full Render* se realizará el cálculo de todas las propiedades del material, texturas e iluminación sobre la textura (sin tener en cuenta el brillo especular que es dependiente del punto de vista del observador). Si se activa el botón *Ambient Occlusion* únicamente se tendrá en cuenta la información de AO ignorando el resto de fuentes de luz de la escena (ver sección 9.6). Activando el botón *Textures* se asignan los colores base de los materiales y texturas (sin tener en cuenta el sombreado).

El botón *Clear* sirve para borrar la textura antes de realizar el *baking*. Puede ser interesante desactivarlo si queremos aplicar una pasada de AO a una textura previamente asignada manualmente.

El botón *Selected to Active* permite asignar la información de otro objeto. Un uso típico de esta herramienta es disponer de un modelo en alta resolución que define una geometría muy detallada, y mapear su información en otro modelo de baja resolución. En esta sección utilizaremos esta funcionalidad para asociar la información de una malla de radiosidad a la malla en baja resolución de la escena.

Finalmente, cuando se han elegido los parámetros en la pestaña, pulsando el botón [Bake] se inicia el proceso de cálculo. En la ventana de UV Mapping deberá verse la actualización en tiempo real de la textura mapeada al modelo.

9.6. Ambient Occlusion

El empleo del término de luz ambiente viene aplicándose desde el inicio de los gráficos por computador como un método muy rápido de simular la contribución de luz ambiental que proviene de todas las direcciones. La técnica de *Ambient Occlusion* es un caso particular del uso de pruebas de oclusión en entornos con iluminación local para determinar los efectos difusos de iluminación. Estas técnicas fueron introducidas inicialmente por Zhurov como alternativa a las técnicas de radiosidad para aplicaciones interactivas (videojuegos), por su bajo coste computacional.

En el esquema de la figura 9.12 podemos ver en qué se basan estas técnicas. Desde cada punto P de intersección con cada superficie (obtenido mediante trazado de rayos), calculamos el valor de ocultación de ese punto que será proporcional al número de rayos que alcanzan el "*cielo*" (los que no intersectan con ningún objeto dada una distancia máxima de intersección). En el caso de la figura serán $4/7$ de los rayos lanzados.

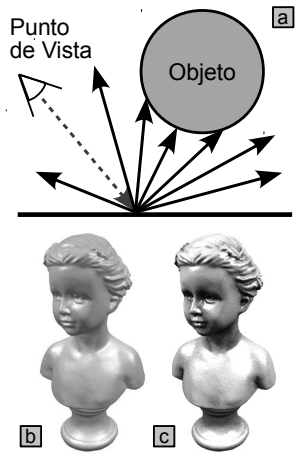


Figura 9.12: Descripción esquemática del cálculo de Ambient Occlusion. **a)** Esquema de cálculo del valor de ocultación $W(P)$. **b)** Render obtenido con iluminación lobal (dos fuentes puntuales). **c)** La misma configuración que en b) pero con Ambient Occlusion de 10 muetras por píxel.

Podemos definir la ocultación de un punto de una superficie como:

$$W(P) = \frac{1}{\pi} \int_{\omega \in \Omega} \rho(d(P,\omega)) cos\theta d\omega \tag{9.1}$$

Obteniendo un valor de ocultación $W(P)$ entre 0 y 1, siendo $d(P,\omega)$ la distancia entre P y la primera intersección con algún objeto en la dirección de ω. $\rho(d(P,\omega))$ es una función con valores entre 0 y 1 que nos indica la magnitud de iluminación ambiental que viene en la dirección de ω, y θ es el ángulo formado entre la normal en el punto P y la dirección de ω.

Estas técnicas de ocultación (*obscurances*) se desacoplan totalmente de la fase de iluminación local, teniendo lugar en una segunda fase de iluminación secundaria difusa. Se emplea un valor de distancia para limitar la ocultación únicamente a polígonos cercanos a la zona a sombrear mediante una función. Si la función toma valor de ocultación igual a cero en aquellos puntos que no superan un umbral y un valor de uno si están por encima del umbral, la técnica de ocultación se denomina *Ambient Occlusion*. Existen multitud de funciones exponenciales que se utilizan para lograr estos efectos.

La principal ventaja de esta técnica es que es bastante más rápida que las técnicas que realizan un cálculo correcto de la iluminación indirecta. Además, debido a la sencillez de los cálculos, pueden realizarse aproximaciones muy rápidas empleando la GPU, pudiendo utilizarse en aplicaciones interactivas. El principal inconveniente es que no es un método de iluminación global y no puede simular efec-

tos complejos como caústicas o contribuciones de luz entre superficies con reflexión difusa.

Para activar el uso de Ambient Occlusion (AO) en Blender, en el grupo de botones del mundo , en la pestaña *Amb Occ* activamos el botón Ambient Occlusion. La primera lista desplegable permite elegir entre el tipo de cálculo de AO: Mediante trazado de rayos o Aproximada (ver Figura 9.13).

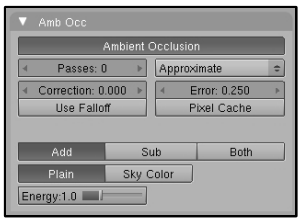


Figura 9.13: Pestaña Amb Occ en Blender.

El cálculo de AO mediante trazado de rayos ofrece resultados más precisos a costa de un tiempo de render mucho mayor. El efecto del ruido blanco debido al número de rayos por píxel puede disminuirse a costa de aumentar el tiempo de cómputo aumentando el número de muestras (Samples).

El método AO Aproximado (ver Figura 9.13) realiza una rápida aproximación que, en muchos casos, puede ser suficiente. No sufre de ruido de muestreo, por lo que es buena opción para ser utilizada en multitud de ocasiones. El parámetro *Error* define la calidad de las sombras calculadas (valores menores implican mejores resultados con mayor tiempo de cómputo). El botón *Pixel Cache* si está activo hace que el valor de sombreado se interpole entre píxeles vecinos, haciendo que el cálculo sea aún más rápido (aunque menos exacto). A continuación estudiaremos algunas opciones relevantes del método:

- **Use Falloff**: Esta opción controla el tamaño de las sombras calculadas por AO. Si está activa, aparece un nuevo control *Strength* que permite variar el factor de atenuación. Con valores mayores de *Strength*, la sombra aparece más enfocada (es más pequeña).
- **Add**: El punto recibe luz según los rayos que no se han chocado con ningún objeto. La escena esta más luminosa que la original sin AO.
- **Sub**: El punto recibe sombra según los rayos que han chocado con algún objeto. La escena es más oscura que la original sin AO.
- **Both**: Emplea Add y Sub a la vez. Si se activa este botón, normalmente se crean zonas de mayor contraste entre luces y sombras.

Mediante el último grupo de botones podemos controlar el color de la luz empleada en iluminación AO; *Plain* emplea luz de color blanca, *Sky Color* utiliza el color definido en el horizonte, o *Sky Texture* (sólo con AO mediante Trazado de Rayos) si queremos utilizar un mapa de textura (en este caso, el color de la luz se corresponderá con el color de pixel con el que choque cada rayo). Finalmente, *Energy* indica la intensidad que tendrán asignados los rayos de AO.

9.7. Radiosidad

En esta técnica se calcula el intercambio de luz entre superficies. Esto se consigue subdividiendo el modelo en pequeñas unidades denominadas *parches*, que serán la base de la distribución de luz final.

Inicialmente los modelos de radiosidad calculaban las interacciones de luz entre superficies difusas (aquellas que reflejan la luz igual en todas las direcciones), aunque existen modelos más avanzados que tienen en cuenta modelos de reflexión más complejos.

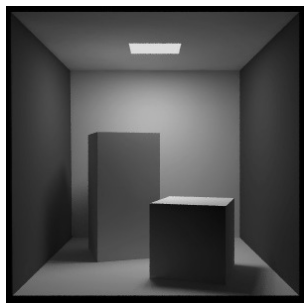


Figura 9.14: El modelo de Radiosidad permite calcular el intercambio de luz entre superficies difusas, mostrando un resultado de render correcto en el problema planteado en la histórica *Cornell Box*.

El modelo básico de radiosidad calcula una solución independiente del punto de vista. Sin embargo, el cálculo de la solución es muy costoso en tiempo y en espacio de almacenamiento. No obstante, cuando la iluminación ha sido calculada, puede utilizarse para renderizar la escena desde diferentes ángulos, lo que hace que este tipo de soluciones se utilicen en visitas interactivas y videojuegos en primera persona actuales. En el ejemplo de la figura 9.15, en el techo de la habitación se encuentran las caras poligonales con propiedades de emisión de luz. Tras aplicar el proceso del cálculo de radiosidad obtenemos la malla de radiosidad que contiene información sobre la distribución de iluminación entre superficies difusas.

En el modelo de radiosidad, cada superficie tiene asociados dos valores: la intensidad luminosa que recibe, y la cantidad de energía que emite (energía radiante). En este algoritmo se calcula la interacción de energía desde cada superficie hacia el resto. Si tenemos n superficies, la complejidad del algoritmo será $O(n^2)$. El valor matemático que calcula la relación geométrica entre superficies se denomina **Factor de Forma**, y se define como:

$$F_{ij} = \frac{cos\theta_i cos\theta_j}{\pi r^2} H_{ij} dA_j \tag{9.2}$$

Siendo F_{ij} el factor de forma de la superficie i a la superficie j, en el numerador de la fracción definimos el ángulo que forman las normales de las superficies, πr^2 mide la distancia entre las superficies, H_{ij} es el parámetro de visibilidad, que valdrá uno si la superfice j es totalmente visible desde i, cero si no es visible y un valor entre uno y cero según el nivel de oclusión. Finalmente, dA_j indica el área de la superfice j (no tendremos el mismo resultado con una pequeña superficie emisora de luz sobre una superfice grande que al contrario). Este factor de forma se suele calcular empleando un hemi-cubo.

Será necesario calcular n^2 factores de forma (no cumple la propiedad conmutativa) debido a que se tiene en cuenta la relación de área entre superficies. La matriz que contiene los factores de forma relacionando todas las superficies se denomina *Matriz de Radiosidad*. Cada elemento de esta matriz contiene un factor de forma para la interacción desde la superficie indexada por la columna hacia la superficie indexada por la fila (ver figura 9.16).

Algoritmo de refinamiento progresivo

```
1    Para cada iteración
2      seleccionar un parche i
3      calcular Fij para todas las superficies j
4      para cada superficie j hacer:
5        actualizar la radiosidad de la superficie j
6        actualizar la emisión de la superficie j
7      emision(i) = 0
```

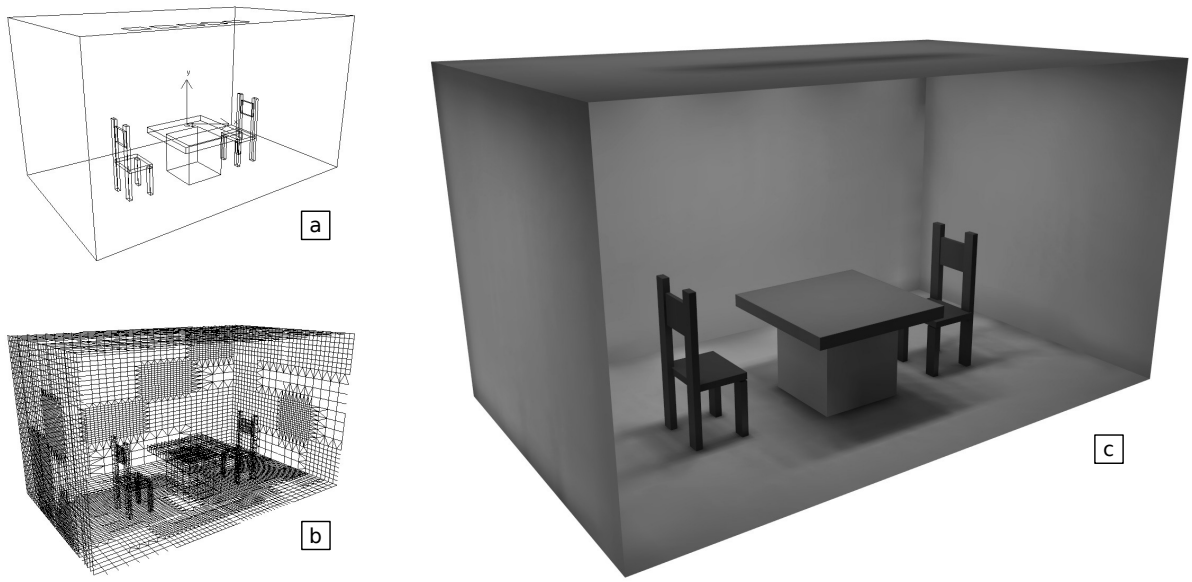


Figura 9.15: Ejemplo de aplicación del método de radiosidad sobre una escena simple. En el techo de la habitación se ha definido una fuente de luz. **a)** Malla original. **b)** Malla de radiosidad. **c)** Resultado del proceso de renderizado.

En 1988, Cohen introdujo una variante de cálculo basada en el *refinamiento progresivo* que permite que la solución de radiosidad encontrada en cada iteración del algoritmo sea mostrada al usuario. Este método progresivo es un método incremental que requiere menos tiempo de cómputo y de almacenamiento; en cada iteración se calcula los factores de forma entre una superficie y el resto (en el artículo original se requería el cálculo de n^2 factores de forma).

Este método de refinamiento progresivo finalmente obtiene la misma solución que el original, proporcionando resultados intermedios que van siendo refinados.

Refinamiento Progresivo

Blender implementa el método de Radiosidad basado en refinamiento progresivo

En general, el cálculo de la radiosidad es eficiente para el cálculo de distribuciones de luz en modelos simples con materiales difusos, pero resulta muy costoso para modelos complejos (debido a que se calculan los valores de energía para cada parche del modelo) o con materiales no difusos. Además, la solución del algoritmo se muestra como una nueva malla poligonal que tiende a desenfocar los límites de las sombras. Como en videojuegos suelen emplearse modelos en baja poligonalización, y es posible *mapear* esta iluminación precalculada en texturas, el modelo de Radiosidad se ha empleado en diveros títulos de gran éxito comercial. Actualmente incluso existen motores gráficos que calculan soluciones de radiosidad en tiempo real.

Veamos a continuación un ejemplo de utilización de Radiosidad en Blender. Como hemos indicado anteriormente, el modelo de Radiosidad calcula la interacción de iluminación entre superficies. Así, la

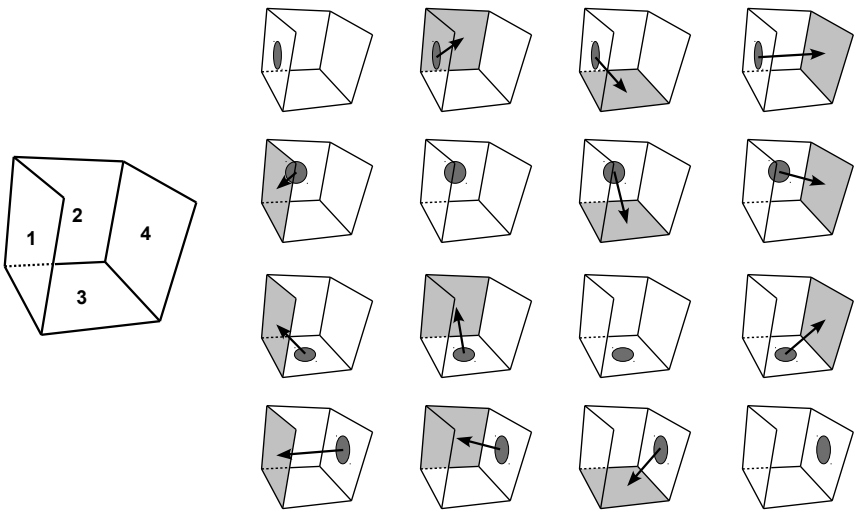


Figura 9.16: Esquema de la matriz de radiosidad. En cada posición de la matriz se calcula el *Factor de Forma*. La diagonal principal de la matriz no se calcula.

iluminación de la escena vendrá dada por las áreas de luz de las superficies superiores (planos) de la habitación. De esta forma, necesitaremos crear planos a los que asignaremos un material que emita luz. La escena de ejemplo (ver Figura 9.17) cuenta con planos con estas propiedades definidas.

Es muy importante la dirección del vector normal ya que Blender calculará la interacción de la luz en ese sentido. Por tanto, tendremos que asegurarnos que el vector normal de cada foco apunta "*hacia el suelo*", tal y como muestra la Figura 9.18. Para comprobar que es así, seleccionamos cada emisor de luz, en modo de edición de vértices activamos el botón *Draw Normals* de la pestaña *Mesh Tools More*. Podemos ajustar el tamaño de representación del vector normal en *Nsize*. En caso de que la normal esté invertida, podemos ajustarla pulsando [W] *Flip Normals*.

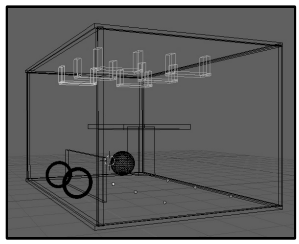


Figura 9.17: Escena de ejemplo para el cálculo de la Radiosidad.

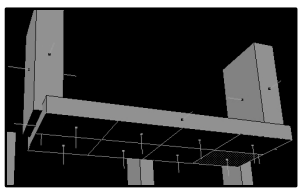


Figura 9.18: Focos del techo con el vector normal correctamente definido.

Los elementos que emiten luz tendrán el campo *Emit* del material con un valor mayor que 0. Los emisores de este ejemplo tienen un valor de emisión de 0.4.

Accedemos al menú de radiosidad dentro de los botones de sombreado . Seleccionamos todas las mallas que forman nuestra escena [A] y pinchamos en el botón *Collect Meshes* (ver Figura 9.19). La escena aparecerá con colores sólidos.

Hecho esto, pinchamos en [Go]. De esta forma comienza el proceso de cálculo de la solución de radiosidad. Podemos parar el proceso en cualquier momento pulsando [Escape], quedándonos con la aproximación que se ha conseguido hasta ese instante.

Figura 9.19: Opciones generales de Radiosidad (Paneles *Radio Render*, *Radio Tool* y *Calculation*.

Si no estamos satisfechos con la solución de Radiosidad calculada, podemos eliminarla pinchando en *Free Radio Data*.

En la Figura 9.19 se muestran algunas opciones relativas al cálculo de la solución de radiosidad. El tamaño de los *Parches* (*PaMax - PaMin*) determina el detalle final (cuanto más pequeño sea, más detallado será el resultado final), pero incrementamos el tiempo de cálculo. De forma similar ocurre con el tamaño de los *Elementos*[2] (*ElMax - ElMin*). En *Max Iterations* indicamos el número de pasadas que Blender hará en el bucle de Radiosidad. Un valor 0 indica que haga las que estime necesarias para minimizar el error (lo que es conveniente si queremos generar la malla de radiosidad final). *MaxEl* indica el número máximo de elementos para la escena. Hemires es el tamaño del hemicubo para el cálculo del factor de forma.

Una vez terminado el proceso de cálculo (podemos pararlo cuando la calidad sea aceptable con Esc), podemos añadir la nueva malla calculada reemplazando las creadas anteriormente (*Replace Meshes*) o añadirla como nueva a la escena (*Add new Meshes*). Elegiremos esta segunda opción, para aplicar posteriormente el resultado a la malla en baja poligonalización. Antes de deseleccionar la malla con la información de radiosidad calculada, la moveremos a otra capa (mediante la tecla M) para tener los objetos mejor organizados. Hecho esto, podemos liberar la memoria ocupada por estos datos pinchando en *Free Radio Data*.

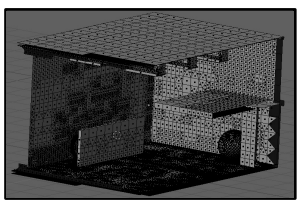


Figura 9.20: Resultado de la malla de radiosidad.

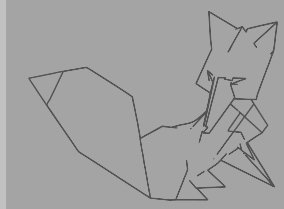
El resultado de la malla de radiosidad se muestra en la Figura 9.20. Como se puede comprobar, es una malla extremadamente densa, que no es directamente utilizable en aplicaciones de tiempo real. Vamos a utilizar el *Baking* de Blender para utilizar esta información de texturas sobre la escena original en baja poligonalización.

Por simplicidad, uniremos todos los objetos de la escena original en una única malla (seleccionándolos todos A y pulsando Control J *Join Selected Meshes*). A continuación, desplegaremos el objeto empleando el modo de despliegue de *Light Map*, y crearemos una nueva imagen que recibirá el resultado del render Baking.

[2] En Blender, cada Parche está formado por un conjunto de Elementos, que describen superficies de intercambio de mayor nivel de detalle.

Figura 9.21: Resultado de aplicación del mapa de radiosidad al modelo en baja resolución.

Ahora seleccionamos primero la malla de radiosidad, y después con Shift pulsado seleccionamos la malla en baja resolución. Pinchamos en el botón *Selected to Active* de la pestaña *Bake*, activamos *Full Render* y pinchamos en el botón Bake. Con esto debemos haber asignado el mapa de iluminación de la malla de radiosidad al objeto en baja poligonalización (ver Figura 9.21).



Capítulo 10

Animación

Carlos González Morcillo

En este capítulo estudiaremos los fundamentos de la animación por computador, analizando los diversos métodos y técnicas de construcción, realizando ejemplos de composición básicos con Ogre. Se estudiará el concepto de *Animation State*, exportando animaciones definidas previamente en Blender.

10.1. Introducción

Animación?

El término *animación* proviene del griego *Anemos*, que es la base de la palabra latina *Animus* que significa *Dar aliento, dar vida.*

En su forma más simple, la animación por computador consiste en generar un conjunto de imágenes que, mostradas consecutivamente, producen sensación de movimiento. Debido al fenómeno de la *Persistencia de la Visión*, descubierto en 1824 por Peter Mark Roget, el ojo humano retiene las imágenes una vez vistas unos 40 ms. Siempre que mostremos imágenes a una frecuencia mayor, tendremos sensación de movimiento continuo[1]. Cada una de las imágenes que forman la secuencia animada recibe el nombre de *frame*[2].

La animación por computador cuenta con una serie de ventajas que no se dan en animación tradicional; por ejemplo, la animación puede producirse directamente desde modelos o conjuntos de ecuaciones que especifican el comportamiento dinámico de los objetos a animar.

[1] A frecuencias menores de 20hz se percibirá la discretización del movimiento, en un efecto de tipo estroboscópico.

[2] también, aunque menos extendido en el ámbito de la animación por computador se utiliza el término cuadro o fotograma

Cuando hablamos de técnicas de animación por computador nos referimos a sistemas de *control del movimiento*. Existen multitud de elementos que hacen el problema del control del movimiento complejo, y de igual forma, existen varias aproximaciones a la resolución de estas dificultades. La disparidad de aproximaciones y la falta de unidad en los convenios de nombrado, hacen que la categorización de las técnicas de animación por computador sea difícil. Una propuesta de clasificación de se realiza según el nivel de abstracción.

Así, se distingue entre sistemas de animación de *alto nivel*, que permiten al animador especificar el movimiento en términos generales (definen el comportamiento en términos de eventos y relaciones), mientras que los sistemas de *bajo nivel* requieren que el animador indique los parámetros de movimiento individualmente.

Si se desean animar objetos complejos (como por ejemplo la figura humana), es necesario, al menos, un control de jerarquía para reducir el número de parámetros que el animador debe especificar. Incluso en personajes digitales sencillos, como el mostrado en la figura 10.1 el esqueleto interno necesario tiene un grado de complejidad considerable. En concreto, este esqueleto tiene asociados 40 huesos con diversos modificadores aplicados individualmente.

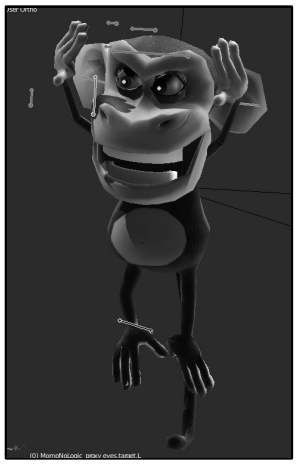


Figura 10.1: Uno de los personajes principales de Yo-Frankie, proyecto de videojuego libre de la Blender Foundation.

Al igual que en los lenguajes de programación de alto nivel, las construcciones deben finalmente *compilarse* a instrucciones de bajo nivel. Esto implica que cualquier descripción paramétrica de alto nivel debe transformarse en descripciones de salida de bajo nivel. Este proceso proceso finalizará cuando dispongamos de todos los datos necesarios para todos los frames de la animación.

De esta forma, lo que se busca en el ámbito de la animación es ir subiendo de nivel, obteniendo sistemas de mayor nivel de abstracción. Las primeras investigaciones comenzaron estudiando las técnicas de interpolación de movimiento entre frames clave, utilizando Splines. Mediante este tipo de curvas, los objetos podían moverse de forma suave a lo largo de un camino en el espacio. Desde este punto, han aparecido multitud de técnicas de animación de medio nivel (como la animación basada en scripts, animación procedural, animación jerárquica basada en cinemática directa e inversa y los estudios de síntesis animación automática).

A continuación estudiaremos los principales niveles de abstracción relativos a la definición de animación por computador.

10.1.1. Animación Básica

Técnicas básicas

Estas técnicas forman la base de los métodos de animación más avanzados, por lo que es imprescindible conocer su funcionamiento para emplear correctamente los métodos de animación basados en cinemática directa o inversa que estudiaremos más adelante.

Lo fundamental en este nivel es cómo parametrizar los caminos básicos de movimiento en el espacio. Hay diversas alternativas (no excluyentes) como los sistemas de Script, sistemas de Frame Clave y animación dirigida por Splines.

- **Sistemas de Script**. Históricamente, los primeros sistemas de control del movimiento fueron los sistemas basados en scripts. Este tipo de sistemas requieren que el usuario escriba un guión

en un lenguaje específico para animación y además, presuponen una habilidad por parte del animador de expresar la animación con el lenguaje de script. Este tipo de aproximación producen animaciones de baja calidad, dada la complejidad en la especificación de las acciones.

- **Sistema de Frame Clave**. Los sistemas de Frame Clave toman su nombre de la jerarquía de producción tradicional de Walt Disney. En estos sistemas, los animadores más experimentados diseñaban los fotogramas principales de cada secuencia. La producción se completaba con artistas jóvenes que añadían los frames intermedios[3]. En animación por computador, el animador puede modificar en tiempo real los frames clave de cada secuencia. Esto requiere que el ordenador calcule de forma automática los fotogramas intermedios de cada secuencia.

  Dependiendo del tipo de método que se utilice para el cálculo de los fotogramas intermedios estaremos ante una técnica de interpolación u otra. En general, suelen emplearse curvas de interpolación del tipo Spline, que dependiendo de sus propiedades de continuidad y el tipo de cálculo de los ajustes de tangente obtendremos diferentes resultados. En el ejemplo de la Figura 10.2, se han añadido claves en los frames 1 y 10 con respecto a la posición (localización) y rotación en el eje Z del objeto. El ordenador calcula automáticamente la posición y rotación de las posiciones intermedias. La figura muestra las posiciones intermedias calculadas para los frames 4 y 7 de la animación.

- **Animación Procedural**. En esta categoría, el control sobre el movimiento se realiza empleando procedimientos que definen explícitamente el movimiento como función del tiempo. La generación de la animación se realiza mediante descripciones físicas, como por ejemplo en visualizaciones científicas, simulaciones de fluidos, tejidos, etc... Estas técnicas de animación procedural serán igualmente estudiadas en el Módulo 3 del curso. La simulación dinámica, basada en descripciones matemáticas y físicas suele emplearse en animación de acciones secundarias. Es habitual contar con animaciones almacenadas a nivel de vértice simulando comportamiento físico en videojuegos. Sería el equivalente al *Baking* de animaciones complejas precalculadas.

Con base en estas técnicas fundamentales se definen los métodos de animación de alto nivel descritos a continuación.

10.1.2. Animación de Alto Nivel

Dentro de este grupo incluimos la animación mediante técnicas cinemáticas jerárquicas (como cinemática directa o inversa de figuras articuladas), las modernas aproximaciones de síntesis automática de animación o los sistemas de captura del movimiento. Estas técnicas específicas serán estudiadas en detalle en el Módulo 3 del curso, por

[3] Realizando la técnica llamada ïn between"

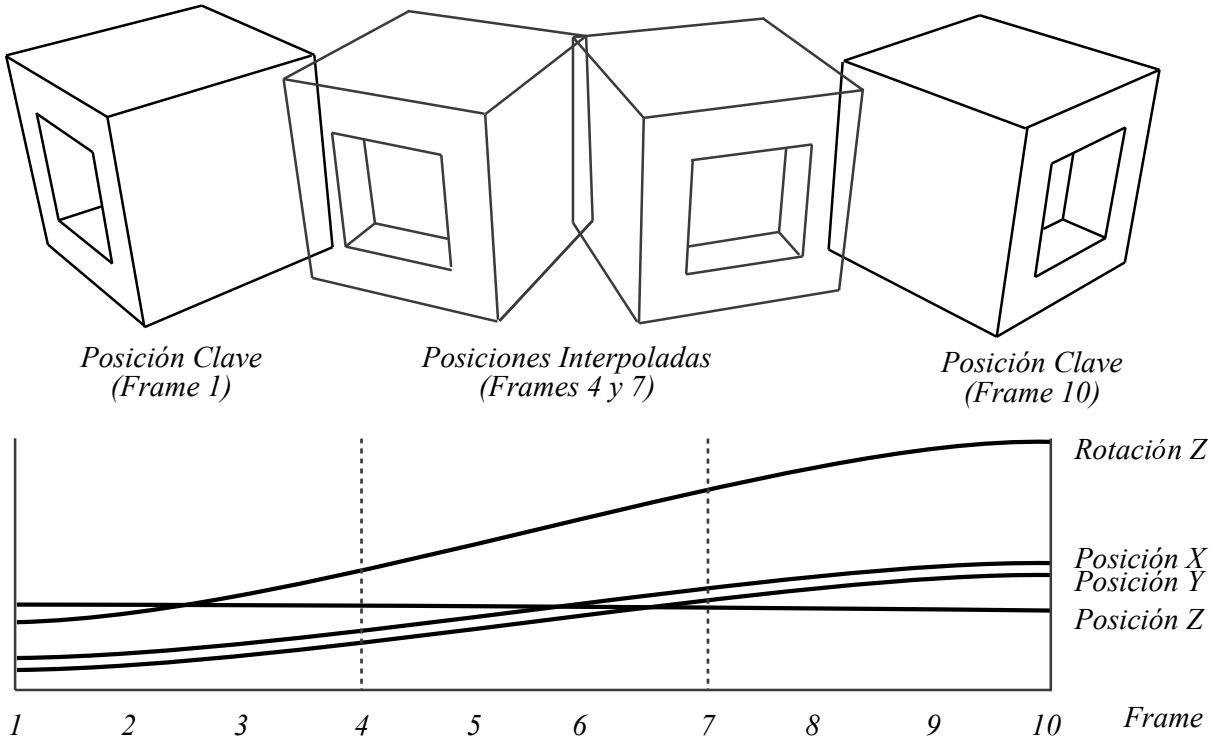


Figura 10.2: Ejemplo de uso de Frames Clave y Splines asociadas a la interpolación realizada.

lo que únicamente realizaremos aquí una pequeña descripción introductoria.

- **Cinemática Directa**. Empleando cinemática directa, el movimiento asociado a las articulaciones debe ser especificado explícitamente por el animador. En el ejemplo de la Figura 10.3, la animación del efector final vendría determinada indirectamente por la composición de transformaciones sobre la base, el hombro, el codo y la muñeca. Esto es, una estructura arbórea descendente. Esto es, dado el conjunto de rotaciones θ, obtenemos la posición del efector final X como $X = f(\theta)$.
- **Cinemática Inversa**. El animador define únicamente la posición del efector final. Mediante cinemática inversa se calcula la posición y orientación de todas las articulaciones de la jerarquía que consiguen esa posición particular del efector final mediante $\theta = f(X)$. En el Módulo 3 estudiaremos el método más utilizado para cadenas de cinemática complejas CCD (*Cyclic Coordinate Descent*).

Una vez realizada esta introducción general a las técnicas de animación por computador, estudiaremos en la siguiente sección los tipos

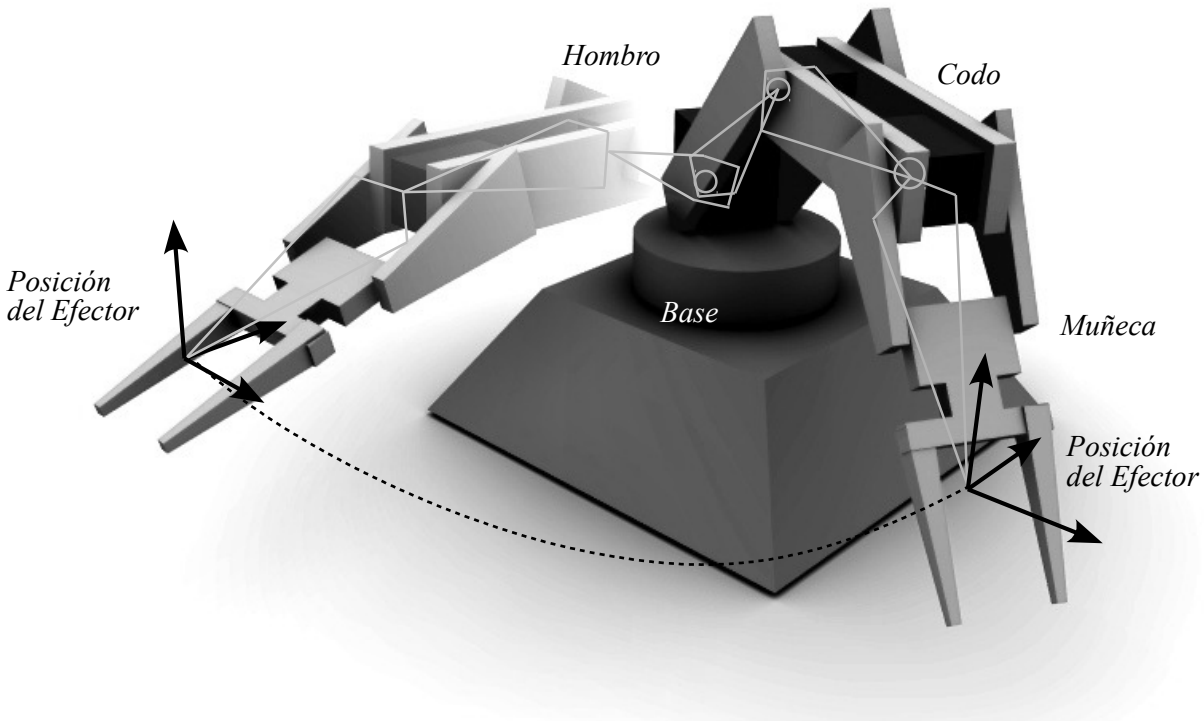


Figura 10.3: Ejemplo de uso de Frames Clave y Splines asociadas a la interpolación realizada.

de animación y las características generales de cada uno de ellos en el motor gráfico Ogre.

10.2. Animación en Ogre

Ogre gestiona la posición de los elementos de la escena en cada frame. Esto significa que la escena es *redibujada* en cada frame. Ogre no mantiene ningún tipo de estructuras de datos con el estado anterior de los objetos de la escena. La animación se gestiona con dos aproximaciones; una basada en frames clave y otra mediante una variable (habitualmente se empleará el tiempo) utilizada como controlador. El modo más sencillo de crear animaciones es mediante el uso de una herramienta externa y reproducirlas posteriormente.

Como hemos señalado anteriormente, de modo general Ogre soporta dos modos de animación: basada en Frames Clave (*Keyframe Animation*) y basada en Controladores.

10.2.1. Animación Keyframe

Ogre utiliza el término pista *Track* para referirse a un conjunto de datos almacenados en función del tiempo. Cada muestra realizada en

un determinado instante de tiempo es un *Keyframe*. Dependiendo del tipo de *Keyframes* y el tipo de pista, se definen diferentes tipos de animaciones.

Las animaciones asociadas a una Entidad se representan en un *AnimationState*. Este objeto tiene una serie de propiedades que pueden ser consultadas:

AnimationState

Cada animación de un *AnimationState* tiene asociado un nombre único mediante el que puede ser accedido.

- **Nombre**. Mediante la llamada a *getAnimationName se* puede obtener el nombre de la animación que está siendo utilizada por el *AnimationState*.
- **Activa**. Es posible activar y consultar el estado de una animación mediante las llamadas a *getEnabled* y *setEnabled*.
- **Longitud**. Obtiene en punto flotante el número de segundos de la animación, mediante la llamada a *getLength*.
- **Posición**. Es posible obtener y establecer el punto de reproducción actual de la animación mediante llamadas a *getTimePosition* y *setTimePosition*. El tiempo se establece en punto flotante en segundos.
- **Bucle**. La animación puede reproducirse en modo bucle (cuando llega al final continua reproduciendo el primer frame). Las llamadas a métodos son *getLoop* y *setLoop*.
- **Peso**. Este parámetro controla la mezcla de animaciones, que será descrita en la sección 10.4.

El estado de la animación requiere que se le especifique el tiempo transcurrido desde la última vez que se actualizó.

Tiempo

El tiempo transcurrido desde la última actualización puede especificarse con un valor o negativo (en el caso de querer retroceder en la animación). Este valor de tiempo simplemente se añade a la posición de reproducción de la animación.

10.2.2. Controladores

La animación basada en frames clave permite animar mallas poligionales. Mediante el uso de controladores se posicionan los nodos, definiendo una función. El controlador permite modificar las propiedades de los nodos empleando un valor que se calcula en tiempo de ejecución.

10.3. Exportación desde Blender

En esta sección estudiaremos la exportación de animaciones de cuerpos rígidos. Aunque para almacenar los *AnimationState* es necesario asociar un esqueleto al objeto que se desea animar, no entraremos en detalles para la animación de jerarquías compuestas (personajes). Esos aspectos avanzados serán estudiados en el tercer módulo del curso.

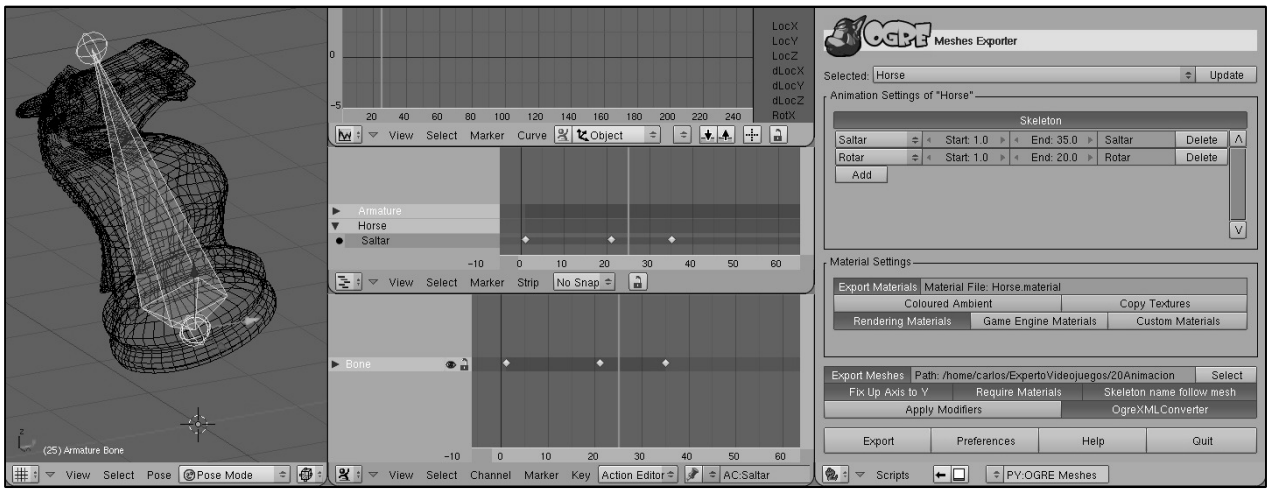


Figura 10.4: Principales ventanas de Blender empleadas en la animación de objetos. En el centro se encuentran las ventanas del *NLA Editor* y del *Action Editor*, que se utilizarán para definir acciones que serán exportadas empleando el script *Ogre Meshes Exporter* (ventana de la derecha).

Para exportar diferentes animaciones, será conveniente configurar el interfaz de Blender como se muestra en la Figura 10.4. A continuación describiremos brevemente las dos ventanas principales que utilizaremos en la exportación de animaciones:

El **NLA Editor** permite la edición de animación no lineal en Blender. Mediante esta técnica de composición, la animación final se crea mediante fragmentos de animación específicos (cada uno está especializado en un determinado movimiento) llamados *acciones*. Estas acciones pueden duplicarse, repetirse, acelerarse o decelerarse en el tiempo para formar la animación final.

Acciones

Las accciones permiten trabajar a un nivel de abstracción mayor que empleando curvas IPO. Es conveniente emplear esta aproximación siempre que definamos animaciones complejas.

El **Action Editor** permite trabajar con los datos relativos a las curvas definidas en el editor de Curvas IPO. Este editor está a un nivel de abstracción medio, entre las curvas IPO y el *NLA Editor* descrito anteriormente. Esta ventana permite trabajar muy rápidamente con los datos de la animación, desplazando rápidamente la posición de los frames clave (y modificando así el *Timing* de cada acción). En el área de trabajo principal del *Action Editor* se muestran los canales de animación asociados a cada objeto. Cada *diamante* de las barras asociadas a cada canal se corresponden con un *keyframe* y pueden ser desplazadas individualmente o en grupos para modificar la temporización de cada animación.

Las acciones aparecerán en el interfaz del exportador de Ogre, siempre que hayan sido correctamente creadas y estén asociadas a un hueso de un esqueleto. En la Figura 10.4 se han definido dos acciones llamadas “Saltar” y “Rotar”, descritas con 35 y 20 frames respectivamente. A continuación estudiaremos el proceso de exportación de animaciones.

Como se ha comentado anteriormente, todas las animaciones deben exportarse empleando animación basada en esqueletos. En el caso

más sencillo de animaciones de cuerpo rígido, bastará con crear un esqueleto de un único hueso y emparentarlo con el objeto que queremos animar.

Añadimos un esqueleto con un único hueso al objeto que queremos añadir mediante Shift A *Add/Armature*. Ajustamos el extremo superior del hueso para que tenga un tamaño similar al objeto a animar (como se muestra en la Figura 10.5). A continuación crearemos una relación de parentesco entre el objeto y el hueso del esqueleto, de forma que el objeto sea hijo de el esqueleto.

Para ello, utilizaremos el modo Pose. Con el hueso del esqueleto seleccionado, elegiremos el modo Pose en la cabecera de la ventana 3D (o bien empleando el atajo de teclado Control TAB). El hueso deberá representarse en color azul en el interfaz de Blender. Con el hueso en modo pose, ahora seleccionamos primero al caballo, y a continuación con Shift pulsado seleccionamos el hueso (se deberá elegir en color azul), y pulsamos Control P, eligiendo *Make Parent to ->Bone*.

Si ahora desplazamos el hueso del esqueleto en modo pose, el objeto deberá seguirle. A continuación creamos un par de animaciones asociadas a este hueso. En la ventana de tipo *Action Editor*, pincharemos en la lista desplegable de acciones y elegiremos *Add New* (ver Figura 10.6). A continuación pincharemos en la caja de texto de color azul que se acabará de crear situada a su derecha para definir el nombre. En este caso, definiremos que la acción se llamará "*Saltar*".

Añadiremos los frames clave de la animación (en este caso, hemos definido frames clave de *LocRot* en 1, 21 y 35).

Podemos comprobar que el proceso se ha realizado correctamente si, eligiendo el objeto (en este caso el caballo), si pinchamos en el botón *Update* del script de Ogre deberá aparecer en la pestaña *Animation Settings* un botón con el nombre del esqueleto (*Skeleton* en este caso, ver Figura 10.7). Si pulsamos sobre el botón *Add* del interfaz podemos elegir la acción que acabamos de crear, el rango de fotogramas que la definirán y el nombre que tendrá en Ogre.

Siguiendo el mismo procedimiento definimos otra acción llamada "*Rotar*". Pinchamos de nuevo en la lista desplegable de la ventana del *Action Editor* (ahora debemos tener la acción anteriormente creada), y elegimos crear otra nueva (ver Figura 10.8).

Es posible exportar diferentes animaciones asociadas a un objeto. El rango de definición de las animaciones puede ser diferente. Es posible incluso (aunque desaconsejable por cuestiones de mantenimiento de los conjuntos de animaciones) definir todas las animaciones de un objeto en una única pista y separar las animaciones empleando diferentes nombres utilizando el script de exportación de Ogre.

Mediante el atajo de teclado Alt A podemos reproducir la animación relativa a la acción que tenemos seleccionada. Resulta muy có-

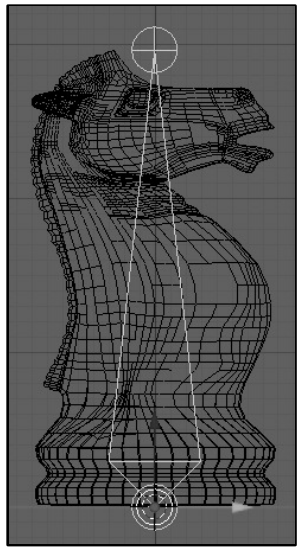


Figura 10.5: Ajuste del tamaño del hueso para que sea similar al objeto a animar.

Figura 10.6: Lista desplegable para añadir una nueva acción.

Figura 10.7: Interfaz del exportador de Ogre "*Animation Settings*".

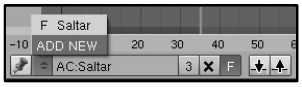


Figura 10.8: Creación de una nueva acción en Blender.

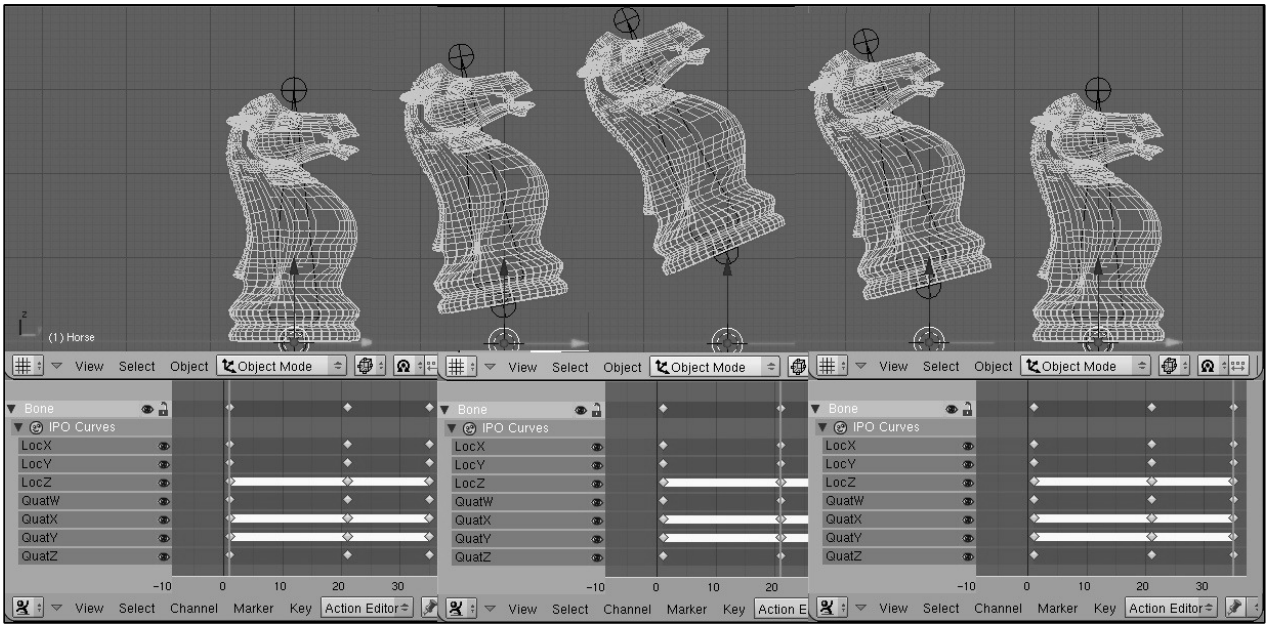


Figura 10.9: Resultado de la animación de una de las acciones del objeto. En la imagen se muestran los 3 frames clave (donde se han aplicado claves respecto de la localización en Z y sobre la rotación) y dos de los fotogramas intermedios. Como puede verse en la imagen, el origen del objeto sigue manteniéndose en la base a lo largo de toda la animación. Esta propiedad nos permitirá reproducir las animaciones en ogre incluso cuando el objeto se esté desplazando por la escena.

modo emplear una ventana de tipo *Timeline* para definir el intervalo sobre el que se crearán las animaciones.

Una vez que se han exportado las animaciones, se creará en el directorio de exportación un nuevo fichero con extensión `.skeleton`. Este archivo contiene la definición jerárquica del conjunto de huesos, junto con su posición y orientación. Esta jerarquía incluye la influencia que tiene cada hueso sobre los vértices del modelo poligonal.

Huesos y vértices

En este capítulo trabajaremos con huesos que tienen una influencia total sobre el objeto (es decir, influencia de 1.0 sobre todos los vértices del modelo).

El uso de las animaciones exportadas en Ogre es relativamente sencillo. En siguiente código muestra un ejemplo de aplicación de las animaciones previamente exportadas. Cuando se pulsa la tecla A o Z se reproduce la animación *Saltar* o *Rotar* hasta que finaliza. En la línea 18 se actualiza el tiempo transcurrido desde la última actualización. Cuando se pulsa alguna de las teclas anteriores, la animación seleccionada se reproduce desde el principio (línea 10).

En el ejemplo siguiente puede ocurrir que se interrumpa una animación por la selección de otra antes de finalizar el estado. Esto podía dar lugar a posiciones finales incorrectas. Por ejemplo, si durante la ejecución de la acción "*Saltar*" se pulsaba la tecla Z, el objeto se quedaba flotando en el aire ejecutando la segunda animación.

Listado 10.1: Fragmento de MyFrameListener.cpp.

```
1  if (_keyboard->isKeyDown(OIS::KC_A) ||
2      _keyboard->isKeyDown(OIS::KC_Z)) {
3    if (_keyboard->isKeyDown(OIS::KC_A))
4      _animState = _sceneManager->getEntity("Horse")->
5                   getAnimationState("Saltar");
6    else _animState = _sceneManager->getEntity("Horse")->
7                     getAnimationState("Rotar");
8    _animState->setEnabled(true);
9    _animState->setLoop(true);
10   _animState->setTimePosition(0.0);
11 }
12
13 if (_animState != NULL) {
14   if (_animState->hasEnded()) {
15     _animState->setTimePosition(0.0);
16     _animState->setEnabled(false);
17   }
18   else _animState->addTime(deltaT);
19 }
```

Es posible definir varias animaciones y gestionarlas mediante diferentes *AnimationState*. En el siguiente ejemplo se cargan dos animaciones fijas en el constructor del *FrameListener* (línea 3-8). Empleando las mismas teclas que en el ejemplo anterior, se resetean cada una de ellas, y se actualizan ambas de forma independiente (en las líneas 26 y 28). Ogre se encarga de mezclar sus resultados (incluso es posible aplicar transformaciones a nivel de nodo, pulsando la tecla R mientras se reproducen las animaciones del *AnimationState*).

Listado 10.2: Fragmento de MyFrameListener.cpp.

```
1  MyFrameListener::MyFrameListener() {
2    // ...
3     _animState = _sceneManager->getEntity("Horse")->
4                  getAnimationState("Saltar");
5    _animState->setEnabled(false);
6    _animState2 = _sceneManager->getEntity("Horse")->
7                  getAnimationState("Rotar");
8    _animState2->setEnabled(false);
9  }
10
11 bool MyFrameListener::frameStarted(const FrameEvent& evt) {
12   // ...
13   if (_keyboard->isKeyDown(OIS::KC_A)) {
14     _animState->setTimePosition(0.0);
15     _animState->setEnabled(true);
16     _animState->setLoop(false);
17   }
18
19   if (_keyboard->isKeyDown(OIS::KC_Z)) {
20     _animState2->setTimePosition(0.0);
21     _animState2->setEnabled(true);
22     _animState2->setLoop(false);
23   }
24
25   if (_animState->getEnabled() && !_animState->hasEnded())
26     _animState->addTime(deltaT);
27   if (_animState2->getEnabled() && !_animState2->hasEnded())
28     _animState2->addTime(deltaT);
29   // ...
```

Obviamente, será necesario contar con alguna clase de nivel superior que nos gestione las animaciones, y se encarge de gestionar los *AnimationState*, actualizándolos adecuadamente.

10.4. Mezclado de animaciones

En la sección anterior hemos utilizado dos canales de animación, con posibilidad de reproducción simultánea. En la mayoría de las ocasiones es necesario contar con mecanismos que permitan el mezclado controlado de animaciones (*Animation Blending*). Esta técnica fundamental se emplea desde hace años en desarrollo de videojuegos.

En la actualidad se emplean módulos específicos para el mezclado de animaciones. El uso de árboles de prioridad (*Priority Blend Tree*[4]) facilita al equipo artístico de una producción especificar con un alto nivel de detalle cómo se realizará la composición de las capas de animación.

El mezclado de animaciones requiere un considerable tiempo de CPU. La interpolación necesaria para mezclar los canales de animación en cada frame hace que el rendimiento del videojuego pueda verse afectado. Empleando interpoilación esférica SLERP, para cada elemento del esqueleto es necesario calcular varias operaciones costosas (cuatro *senos*, un *arccos*, y una raíz cuadrada).

Figura 10.10: La clase de *AnimationBlender* (Mezclado de Animaciones) no tiene nada que ver con el prestigioso paquete de animación del mismo nombre.

A un alto nivel de abstracción, cada animación en Ogre tiene asociado un peso. Cuando se establece la posición dentro de la animación, puede igualmente modificarse el peso del canal de animación. Dependiendo del peso asignado a cada animación, Ogre se encargará de realizar la interpolación de todas las animaciones activas para obtener la posición final del objeto. A continuación definiremos una clase llamada *AnimationBlender* que se encargará de realizar la composición básica de capas de animación.

Las variables miembro privadas de la clase *AnimationBlender* contienen punteros a las animaciones de inicio y fin del mezclado (líneas 10 y 11), el tipo de transición deseado (que es un tipo enumerado definido en las líneas 3-6), y un booleano que indica si la animación se reproducirá en bucle.

Las variables públicas (definidas en 16-18) contienen el tiempo restante de reproducción de la pista actual, la duración (en segundos) del mezclado entre pistas y un valor booleano que indica si la reproducción ha finalizado.

La clase incorpora, además de los tres métodos principales que estudiaremos a continuación, una serie de métodos auxiliares que nos permiten obtener valores relativos a las variables privadas (líneas 24-27).

[4]Más información sobre cómo se implementó el motor de mezclado de animaciones del *MechWarrior* en `http://www.gamasutra.com/view/feature/3456/`

Listado 10.3: AminationBlender.h

```
1  class AnimationBlender {
2   public:
3     enum BlendingTransition {
4       Switch,  // Parar fuente y reproduce destino
5       Blend    // Cross fade (Mezclado suave)
6     };
7
8   private:
9     Entity *mEntity;                 // Entidad a animar
10    AnimationState *mSource;         // Animacion inicio
11    AnimationState *mTarget;         // Animacion destino
12    BlendingTransition mTransition;  // Tipo de transicion
13    bool mLoop;                      // Animacion en bucle?
14
15  public:
16    Real mTimeleft;    // Tiempo restante de la animacion (segundos)
17    Real mDuration;    // Tiempo invertido en el mezclado (segundos)
18    bool mComplete;    // Ha finalizado la animacion?
19
20    AnimationBlender( Entity *);
21    void blend(const String &anim, BlendingTransition transition,
22            Real duration, bool l=true);
23    void addTime(Real);
24    Real getProgress() { return mTimeleft/mDuration; }
25    AnimationState *getSource() { return mSource; }
26    AnimationState *getTarget() { return mTarget; }
27    bool getLoop() { return mLoop; }
28  };
```

Veamos a continuación los principales métodos de la clase, declarados en las líneas [20-23] del listado anterior.

Listado 10.4: AminationBlender.cpp (Constructor)

```
1  AnimationBlender::AnimationBlender(Entity *ent) : mEntity(ent) {
2    AnimationStateSet *set = mEntity->getAllAnimationStates();
3    AnimationStateIterator it = set->getAnimationStateIterator();
4    // Inicializamos los AnimationState de la entidad
5    while(it.hasMoreElements()) {
6      AnimationState *anim = it.getNext();
7      anim->setEnabled(false);
8      anim->setWeight(0);
9      anim->setTimePosition(0);
10   }
11   mSource = NULL;  mTarget = NULL;  mTimeleft = 0;
12 }
```

En el constructor de la clase inicializamos todas las animaciones asociadas a la entidad (recibida como único parámetro). Mediante un iterador (línea [6]) desactivamos todos los *AnimationState* asociados (línea [7]), y reseteamos la posición de reproducción y su peso asociado para su posterior composición (líneas [8-9]).

La clase *AnimationBlender* dispone de un método principal para *cargar* animaciones, indicando (como segundo parámetro) el tipo de mezcla que quiere realiarse con la animación que esté reproduciéndose. La implementación actual de la clase admite dos modos de mezclado; un efecto de mezclado suave tipo *cross fade* y una transición básica de intercambio de canales.

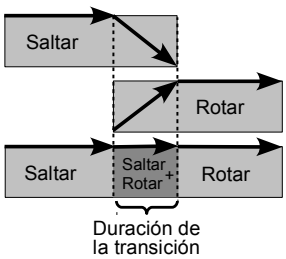


Figura 10.11: Efecto de transición tipo *Cross Fade*. En el tiempo especificado en la duración de la transición ambos canales de animación tienen influencia en el resultado final.

La transición de tipo *Blend* implementa una transición simple lineal de tipo *Cross Fade*. En la Figura 10.11 representa este tipo de transición, donde el primer canal de animación se mezclará de forma suave con el segundo, empleando una combinación lineal de pesos.

Listado 10.5: AminationBlender.cpp (Blend)

```
1  void AnimationBlender::blend (const String &anim,
2       BlendingTransition transition, Real duration, bool l) {
3
4    AnimationState *newTarget = mEntity->getAnimationState(anim);
5    newTarget->setLoop(l);
6    mTransition = transition;
7    mDuration = duration;
8    mLoop = l;
9
10   if ((mTimeleft<=0) || (transition == AnimationBlender::Switch)){
11     // No hay transicion (finalizo la anterior o Switch)
12     if (mSource != NULL) mSource->setEnabled(false);
13     mSource = newTarget;                  // Establecemos la nueva
14     mSource->setEnabled(true);
15     mSource->setWeight(1);                // Con maxima influencia
16     mSource->setTimePosition(0);          // Reseteamos la posicion
17     mTimeleft = mSource->getLength();     // Duracion del AnimState
18     mTarget = NULL;
19   }
20   else {                                  // Hay transicion suave
21     if (mSource != newTarget) {
22       mTarget = newTarget;                // Nuevo destino
23       mTarget->setEnabled(true);
24       mTarget->setWeight(0);              // Cambia peso en addTime
25       mTarget->setTimePosition(0);
26     }
27   }
28 }
```

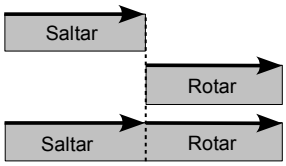


Figura 10.12: Transición de tipo *Intercambio*. El parámetro de duración, aunque se especifique, no tiene ningún efecto.

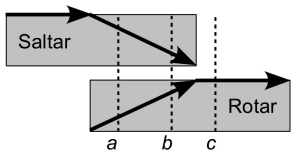
Aunque la implementación actual utiliza una simple combinación lineal de pesos, es posible definir cualquier función de mezcla. Bastará con modificar la implementación del método *addTime* que veremos a continuación.

Por su parte, el método de mezclado de tipo *Switch* realiza un intercambio directo de los dos canales de animación (ver Figura 10.12). Este método es el utilizado igualmente si el canal que se está reproduciendo actualmente ha finalizado (ver línea 10 del listado anterior).

La implementación del método anterior tiene en cuenta el canal de animación que se está reproduciendo actualmente. Si el canal a mezclar es igual que el actual, se descarta (línea 21). En otro caso, se añade como objetivo, activando el nuevo canal pero estableciendo su peso a 0 (línea 24). La mezcla efectiva de las animaciones (el cambio de peso asociado a cada canal de animación) se realiza en el método *addTime*, cuyo listado se muestra a continuación.

Listado 10.6: AminationBlender.cpp (addTime)

```
1  void AnimationBlender::addTime(Real time) {
2    if (mSource == NULL) return;   // No hay fuente
3    mSource->addTime(time);
4    mTimeleft -= time;
5    mComplete = false;
6    if ((mTimeleft <= 0) && (mTarget == NULL)) mComplete = true;
7
8    if (mTarget != NULL) {  // Si hay destino
9      if (mTimeleft <= 0) {
10       mSource->setEnabled(false);
11       mSource->setWeight(0);
12       mSource = mTarget;
13       mSource->setEnabled(true);
14       mSource->setWeight(1);
15       mTimeleft = mSource->getLength();
16       mTarget = NULL;
17     }
18     else {   // Queda tiempo en Source... cambiar pesos
19       Real weight = mTimeleft / mDuration;
20       if (weight > 1) weight = 1.0;
21       mSource->setWeight(weight);
22       mTarget->setWeight(1.0 - weight);
23       if (mTransition == AnimationBlender::Blend)
24         mTarget->addTime(time);
25     }
26   }
27   if ((mTimeleft <= 0) && mLoop) mTimeleft = mSource->getLength();
28 }
```



a) W(*Saltar*)=0.7 W(*Rotar*)=0.3
b) W(*Saltar*)=0.2 W(*Rotar*)=0.8
c) W(*Saltar*)=0.0 W(*Rotar*)=1.0

Figura 10.13: Cálculo del peso de cada animación basándose en la duración de la transición. En los instantes de tiempo *a*, *b* y *c* se muestra el valor del peso asociado a cada canal de animación.

Al igual que en el uso de animaciones básicas en Ogre, el método *addTime* debe ser llamado cada vez que se redibuje la escena, indicando el tiempo transcurrido desde la última actualización. La clase *AnimationBlender* se encargará a su vez de ejecutar el método *addTime* de los canales de animación activos (líneas 3 y 24). El método *addTime* lleva internamente la cuenta del tiempo que le queda al canal de reproducción al canal de animación. Cuando el tiempo es menor o igual que el tiempo empleado para la transición, se calculará el peso que se asignará a cada canal (líneas 19-22). El peso se calcula empleando una sencilla combinación lineal, de modo que el peso total para ambos canales en cualquier instante debe ser igual a uno (ver Figura 10.13).

No es necesario que el peso final combinado de todas las animaciones sea igual a uno. No obstante, es conveniente que la suma de todos los canales de animación estén normalizados y sean igual a 1.

En el caso de asignar pesos negativos, la animación se reproducirá empleando curvas de animación invertidas.

La composición de animaciones suele ser una tarea compleja. En el ejemplo desarrollado para ilustrar el uso de la clase *AnimationBlender* se han utilizado únicamente dos animaciones, y se han utilizado tiempos de transición fijos en todos los casos. En producción es importante trabajar las transiciones y las curvas de animación para obtener resultados atractivos.

Cuántas animaciones?

La cantidad de animaciones necesarias para aplicar correctamente las técnicas de *Animation Blending* pueden ser muy elevadas. Por ejemplo, en *MechWarrior* de Microsoft cada robot tenía asociadas más de 150 animaciones.

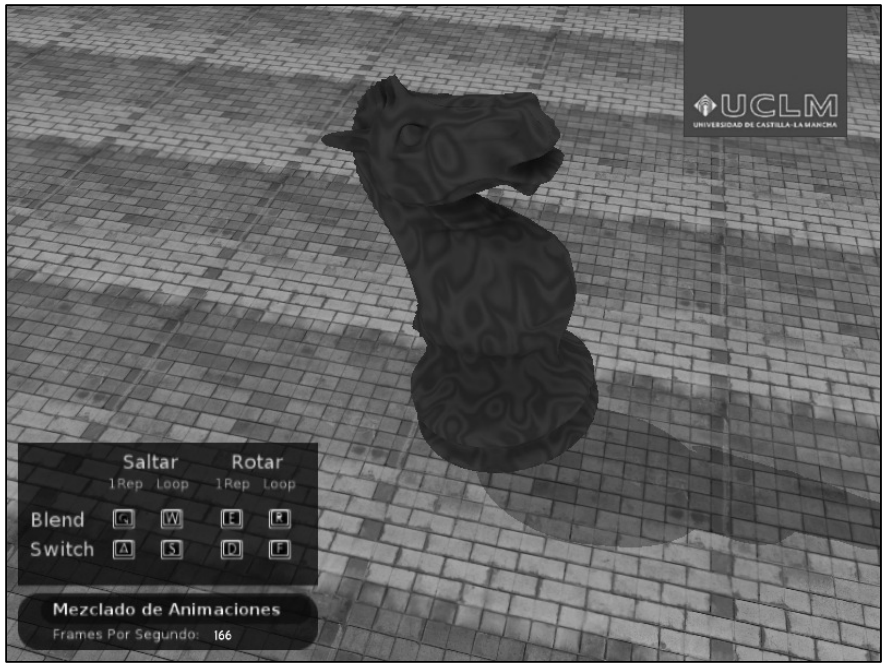


Figura 10.14: Ejemplo de aplicación que utiliza la clase *AnimationBlender*. Mediante las teclas indicadas en el interfaz es posible componer las animaciones exportadas empleando los dos modos de transición implementados.

El mezclado de animaciones que son muy diferentes provoca resultados extraños. Es mejor combinar canales de animación que son similares, de forma que la mezcla funciona adecuadamente. En el ejemplo de esta sección se mezclan animaciones muy diferentes, para mostrar un caso extremo de uso de la clase. En entornos de producción, suelen definirse puntos de conexión entre animaciones, de modo que la clase de *Blending* se espera hasta alcanzar uno de esos puntos para realizar la mezcla. De este modo, se generan un alto número de animaciones para garantizar que las mezclas funcionarán correctamente sin comportamientos extraños.

Es importante realizar pruebas de la correcta composición y mezclado de animaciones en las primeras etapas del desarrollo del juego. De otra forma, podemos sufrir desagradables sorpresas cuando se acerca la deadline del proyecto. Puede resultar complicado ajustar en código la mezcla de animaciones, por lo que suelen desarrollarse scripts de exportación adicionales para indicar los puntos adecuados en cada animación donde puede componerse con el resto.

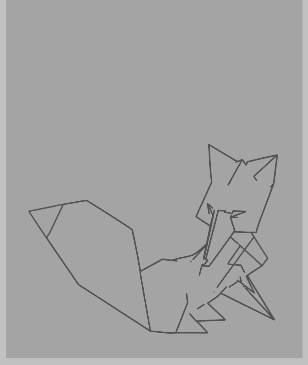
El siguiente listado muestra un ejemplo de uso de la clase. En el constructor del *FrameListener* se crea una variable miembro de la cla-

se que contendrá un puntero al objeto *AnimBlender*. Cada vez que se actualiza el frame, llamamos al método *addTime* (ver línea 11) que actualizará los canales de animación convenientes. En este ejemplo se han utilizado las 8 teclas descritas en la Figura 10.14 para mezclar las dos animaciones exportadas en Blender al inicio del capítulo.

Listado 10.7: Uso de AnimationBlender

```
1  if (_keyboard->isKeyDown(OIS::KC_Q))
2    _animBlender->blend("Saltar",AnimationBlender::Blend,0.5, false);
3  if (_keyboard->isKeyDown(OIS::KC_R))
4    _animBlender->blend("Rotar", AnimationBlender::Blend, 0.5, true);
5
6  if (_keyboard->isKeyDown(OIS::KC_S))
7    _animBlender->blend("Saltar", AnimationBlender::Switch, 0, true);
8  if (_keyboard->isKeyDown(OIS::KC_D))
9    _animBlender->blend("Rotar", AnimationBlender::Switch, 0, false);
10
11 _animBlender->addTime(deltaT);
```

En este capítulo hemos estudiado los usos fundamentales de la animación de cuerpo rígido. En el módulo 3 del curso estudiaremos algunos aspectos avanzados, como la animación de personajes empleando esqueletos y aproximaciones de cinemática inversa y el uso de motores de simulación física para obtener, de forma automática, animaciones realistas.



Capítulo 11

Exportación y Uso de Datos de Intercambio

Carlos González Morcillo

Hasta ahora hemos empleado exportadores genéricos de contenido de animación, mallas poligonales y definición de texturas. Estas herramientas facilitan enormemente la tarea de construcción de contenido genérico para nuestros videojuegos. En este capítulo veremos cómo crear nuestros propios scripts de exportación y su posterior utilización en una sencilla demo de ejemplo.

11.1. Introducción

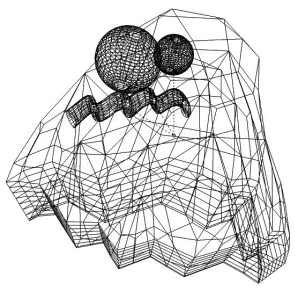


Figura 11.1: El *Wumpus* será el personaje principal del videojuego demostrador de este capítulo *NoEscape Demo*.

En capítulos anteriores hemos utilizado exportadores genéricos de contenido para su posterior importación. En multitud de ocasiones en necesario desarrollar herramientas de exportación de elementos desde las herramientas de diseño 3D a formatos de intercambio.

La comunidad de Ogre ha desarrollado algunos exportadores de geometría y animaciones para las principales suites de animación y modelado 3D. Sin embargo, estos elementos almacenan su posición relativa a su sistema de coordenadas, y no tienen en cuenta otros elementos (como cámaras, fuentes de luz, etc...).

En esta sección utilizaremos las clases definidas en el Capítulo 8 del Módulo 1 para desarrollar un demostrador llamado *NoEscapeDemo*. Estas clases importan contenido definido en formato XML, que fue descrito en dicho capítulo.

A continuación enumeraremos las características esenciales que

debe soportar el exportador a desarrollar:

- **Múltiples nodos**. El mapa del escenario (ver Figura 11.4) describe un grafo. Utilizaremos un objeto de tipo malla poligonal para modelarlo en Blender. Cada vértice de la malla representará un nodo del grafo. Los nodos del grafo podrán ser de tres tipos: *Nodo productor* de Wumpus (tipo **spawn**), *Nodo destructor* de Wumpus (tipo **drain**) o un *Nodo genérico* del grafo (que servirá para modelar las intersecciones donde el Wumpus puede cambiar de dirección).

- **Múltiples aristas**. Cada nodo del grafo estará conectado por uno o más nodos, definiendo múltiples aristas. Cada arista conectará una pareja de nodos. En este grafo, las aristas no tienen asociada ninguna dirección.

- **Múltiples cámaras animadas**. El exportador almacenará igualmente las animaciones definidas en las cámaras de la escena. En el caso de querer almacenar cámaras estáticas, se exportará únicamente un fotograma de la animación. Para cada frame de la animación de cada cámara, el exportador almacenará la posición y rotación (mediante un *cuaternio*) de la cámara en el XML.

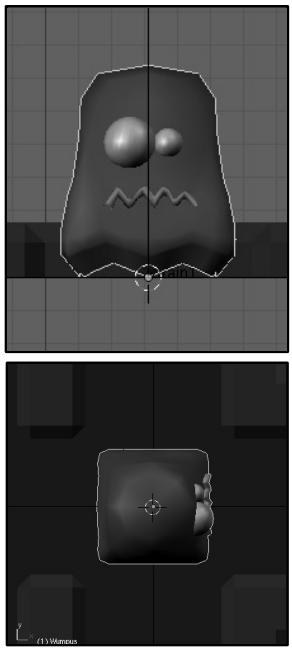


Figura 11.2: Como se estudió en el Capítulo 4, es necesario definir el centro de los objetos especificando adecuadamente su centro. En el caso del Wumpus, el centro se define en el centro de la base, escalándolo además al tamaño existente entre las paredes del escenario.

Existen algunas especificaciones en XML genéricas para la definición de escenas (como el formato "*.scene*") en Ogre, que está soportado por unas clases adicionales disponibles en el Wiki (`http://www.ogre3d.org/tikiwiki/DotScene`) del framework. Aunque este formato define gran cantidad de elementos de una escena, resulta crítico conocer los mecanismos de exportación para desarrollar los scripts específicos que necesitemos en cada proyecto.

El mapa del escenario y los personajes principales se exportarán atendiendo a las indicaciones estudiadas en el Capítulo 4 (ver Figura 11.2). La exportación del XML se realizará empleando el script descrito a continuación. Blender define una API de programación muy completa en Python.

Python es un lenguaje ampliamente utilizado por su facilidad, versatilidad y potencia [1]. Python es un lenguaje de muy alto nivel, interpretado (compilado a un código intermedio), orientado a objetos y libre bajo licencia GPL. Aunque en esta sección nos centraremos en su uso directo como lenguaje de acceso a la API de Blender, en el módulo 3 estudiaremos cómo utilizar Python como lenguaje de script dentro de nuestra apliación en C++.

[1] Resulta especialmente interesante la comparativa empírica de Python con otros lenguajes de programación disponible en `http://www.ipd.uka.de/~prechelt/-Biblio/jccpprtTR.pdf`

Figura 11.3: Especificación del nombre del objeto en la pestaña *Link and Materials*.

¿Quién usa Python? Existen multitud de títulos comerciales y videojuegos de éxito que han utilizado Python como lenguaje para su desarrollo. Por ejemplo, *Civilization 4* utiliza python para la gestión de múltiples tareas, *Vampire: The Masquerade* lo emplea para la descripción de Mods, o *Battlefield 2* (de *Digital Illusions*) para la definición de todos los complementos del juego.

La versión de Blender 2.49b utiliza Python 2.6, por lo que será necesaria su instalación para ejecutar los scripts desarrollados. En una ventana de tipo *Text Editor* podemos editar los scripts. Para su ejecución, bastará con situar el puntero del ratón dentro de la ventana de texto y pulsar Alt P.

La versión 2.49 de Blender no permite añadir nuevas propiedades a ningún objeto de la escena (en Blender 2.60 es posible incluir algunas propiedades definidas por el usuario). De este modo, una forma de añadir información adicional sobre los objetos de la escena es codificándolos mediante el nombre. En la pestaña de *Link and Materials* del grupo de botones de edición , es posible especificar el nombre del objeto (en el campo **OB:**), como se muestra en la Figura 11.3. En nuestro ejemplo, se ha utilizado el siguiente convenio de nombrado:

- **Cámaras**. Las cámaras codifican en el nombre (separadas por guión bajo), como primer parámetro el identificador de la misma (un índice único), y como segundo parámetro el número de frames asociado a su animación. De este modo, la cámara llamada *GameCamera_1_350* indica que es la primera cámara (que será la utilizada en el estado de *Juego*), y que tiene una animación definida de 350 frames.
- **Empty**. Como no es posible asignar tipos a los vértices de la malla poligonal con la que se definirá el grafo, se han añadido objetos de tipo Empty que servirán para asociar los generadores (*spawn*) y destructores (*drain*) de *Wumpus*. Estos objetos Empty tendrán como nombre el literal "spawn" o "drain" y a continuación un número del 1 al 9.

A continuación se muestra el código fuente del script de exportación. Las líneas 5–7 definen tres constantes que pueden ser modificadas en su ejecución: 5 el nombre del fichero XML que se exportará, 6 el nombre del objeto donde se define el grafo (como hemos indicado anteriormente será una malla poligonal), y 7 un valor ϵ que utilizaremos como distancia máxima de separación entre un vértice del grafo y cada *Empty* de la escena (para considerar que están asociados).

Listado 11.1: Script de exportación de *NoEscapeDemo*.

```
import Blender, os, sys
from math import *
from Blender import Mesh, Object, Scene, Mathutils

FILENAME = "output.xml"       # Archivo XML de salida
GRAPHNAME = "Graph"           # Nombre del objeto Mesh del grafo
EPSILON = 0.01                # Valor de distancia Epsilon

# isclose: Decide si un empty coincide con un vertice del grafo
def isclose(empty, coord):
    xo, yo, zo = coord
    xd, yd, zd = empty.getLocation()
    v = Mathutils.Vector(xo-xd, yo-yd, zo-zd)
    if (v.length < EPSILON):
        return True
    else:
        return False

# gettype: Devuelve una cadena con el tipo del nodo del grafo
def gettype (dv, key):
    scn = Scene.GetCurrent()
    ctx = scn.getRenderingContext()
    obs = [ob for ob in scn.objects if ob.type == 'Empty']
    for empty in obs:
        empName = empty.getName()
        if ((empName.find("spawn") != -1) or
            (empName.find("drain") != -1)):
            if (isclose(empty, dv[key])):
                return 'type ="'+ empName[:-1] +'"'
    return 'type=""'

ID1 = ' '*2    # Identadores para el xml
ID2 = ' '*4    # Solo con proposito de obtener un xml "bonito"
ID3 = ' '*6
ID4 = ' '*8

graph = Object.Get(GRAPHNAME)
mesh = graph.getData(False, True)

dv = {}        # Diccionario de vertices
for vertex in mesh.verts:
    dv[vertex.index+1] = vertex.co

de = {}        # Diccionario de aristas
for edge in mesh.edges:
    de[edge.index+1] = (edge.v1.index+1, edge.v2.index+1)

file = open(FILENAME, "w")
std=sys.stdout
sys.stdout=file

print "<?xml version='1.0' encoding='UTF-8'?>\n"
print "<data>\n"

# --------- Exportacion del grafo ----------------------
print "<graph>"
for key in dv.iterkeys():
    print ID1 + '<vertex index="' + str(key) + '" '+
          gettype(dv,key) +'>'
    x,y,z = dv[key]
    print ID2 + '<x> %f</x> <y> %f</y> <z> %f</z>' % (x,y,z)
    print ID1 + '</vertex>'
for key in de.iterkeys():
    print ID1 + '<edge>'
```

```
65      v1,v2 = de[key]
66      print ID2 + '<vertex>%i</vertex> <vertex>%i</vertex>' % (v1,v2)
67      print ID1 + '</edge>'
68  print "</graph>\n"
69
70  # --------- Exportacion de la camara -------------------
71  scn = Scene.GetCurrent()
72  ctx = scn.getRenderingContext()
73  obs = [ob for ob in scn.objects if ob.type == 'Camera']
74  for camera in obs:
75      camId = camera.getName()
76      camName = camId.split("_")[0]
77      camIndex = int(camId.split("_")[1])
78      camFrames = int (camId.split("_")[2])
79      print '<camera index="%i" fps="%i">' % (camIndex, ctx.fps)
80      print ID1 + '<path>'
81      for i in range (camFrames):
82          ctx.cFrame = i+1
83          x,y,z = camera.getMatrix().translationPart()
84          qx,qy,qz,qw = camera.getMatrix().toQuat()
85          print ID2 + '<frame index="%i">' % (i+1)
86          print ID3 + '<position>'
87          print ID4 + '<x>%f</x> <y>%f</y> <z>%f</z>' % (x,y,z)
88          print ID3 + '</position>'
89          print ID3 + '<rotation>'
90          print ID4 + '<x>%f</x> <y>%f</y> <z>%f</z> <w>%f</w>' %
91                    (qx,qy,qz,qw)
92          print ID3 + '</rotation>'
93          print ID2 + '</frame>'
94      print ID1 + '</path>'
95      print '</camera>'
96
97  print "</data>"
98
99  file.close()
100 sys.stdout = std
```

El proceso de exportación del XML se realiza directamente *al vuelo*, sin emplear las facilidades (como el DOM o SAX de Python). La sencillez del XML descrito, así como la independencia de una instalación *completa* de Python hace que ésta sea la alternativa implementada en otros exportadores (como el exportador de Ogre, o el de Collada).

En las líneas 40-46 se crean dos diccionarios donde se guardarán las listas relativas a las coordenadas de los vértices del grafo y las aristas. Blender numera los vértices y aristas asociados a una malla poligonal comenzando en 0. En la descripción del XML indicamos por convenio que todos los índices comenzarían en 1, por lo que es necesario sumar 1 a los índices que nos devuelve Blender.

La malla asociada al objeto de nombre indicado en *GRAPHNAME* se obtiene mediante la llamada a *getData* (línea 38). Como se especifica en la API de Blender, si se indica como segundo parámetro True, la llamada devuelve un objeto de tipo *Mesh*. La línea 52 simplemente imprime la cabecera del XML.

En el bloque descrito por 55-68 se generan las entradas relativas a la definición del grafo, empleando los diccionarios de vértices *dv* y de aristas *de* creados anteriormente.

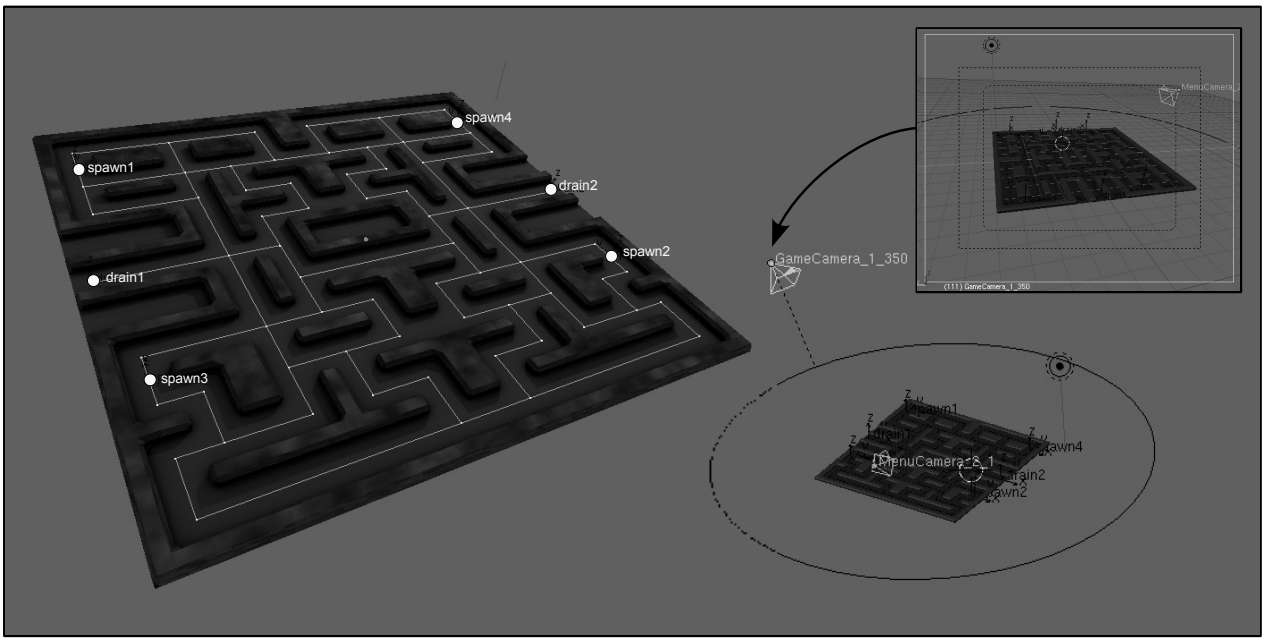


Figura 11.4: Definición del grafo asociado al juego *NoEscapeDemo* en Blender. En la imagen de la izquierda se han destacado la posición de los Emptys auxiliares para describir el tipo especial de algunos nodos (vértices) del grafo. La imagen de la derecha muestra las dos cámaras de la escena y la ruta descrita para la cámara principal del juego.

En la definición del XML (ver Capítulo 8 del Módulo 1), un nodo puede tener asociado un tipo. Para obtener el tipo de cada nodo utilizamos una función auxiliar llamada *gettype* (definida en las líneas 20–30).

En *gettype* se calcula la cadena correspondiente al tipo del nodo del grafo. Como hemos indicado anteriormente, el tipo del nodo se calcula según la distancia a los objetos Empty de la escena. Si el vértice está *muy cerca* de un Empty con subcadena "drain" o "spawn" en su nombre, le asignaremos ese tipo al vértice del grafo. En otro caso, el nodo del grafo será genérico. De este modo, en la línea 23 se obtiene en *obs* la lista de todos los objetos de tipo *Empty* de la escena. Para cada uno de estos objetos, si el nombre contiene alguna de las subcadenas indicadas y está suficientemente cerca (empleando la función auxiliar *isclose*), entonces devolvemos ese tipo (línea 29) para el nodo pasado como argumento de la función (línea 20). En otro caso, devolvemos la cadena vacía para tipo (que indica que no es un nodo especial del grafo) en la línea 30.

Muy Cerca!

Se emplea un valor Epsilon para calcular si un Empty ocupa la posición de un nodo del grafo porque las comparaciones de igualdad con valores de punto flotante pueden dar resultados incorrectos debido a la precisión.

La función auxiliar *isclose* (definida en las líneas 10–17 devolverá *True* si el empty está a una distancia menor que ϵ respecto del nodo del grafo. En otro caso, devolverá *False*. Para realizar esta comparación, simplemente creamos un vector restando las coordenadas del vértice con la posición del *Empty* y utilizamos directamente el atributo de longitud (línea 14) de la clase *Vector* de la biblioteca *Mathutils* de la API de *Blender*.

La biblioteca *Mathutils* de Blender contiene una gran cantidad de funciones para trabajar con Matrices, Vectores y Cuaternios, así como para realizar transformaciones y conversiones entre diversos tipos de representaciones.

Trayectorias genéricas

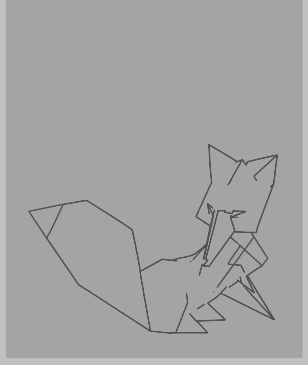
Aunque en este ejemplo exportaremos trayectorias sencillas, el exportar la posición de la cámara en cada frame nos permite tener un control absoluto sobre la pose en cada instante. En la implementación en OGRE tendremos que utilizar algún mecanismo de interpolación entre cada pose clave.

La segunda parte del código (definido en las líneas 71–95) se encarga de exportar las animaciones asociadas a las cámaras. Según el convenio explicado anteriormente, el propio nombre de la cámara codifica el índice de la cámara (ver línea 77) y el número de frames que contiene su animación (ver línea 78). En el caso de este ejemplo se han definido dos cámaras (ver Figura 11.4). La cámara del menú inicial es estática, por lo que define en la parte relativa a los frames un valor de 1 (no hay animación). La cámara del juego describirá rotaciones siguiendo una trayectoria circular.

El número de frame se modifica directamente en el atributo *cFrame* del contexto de render (ver línea 82). Para cada frame se exporta la posición de la cámara (obtenida en la última columna de la matriz de transformación del objeto, en la línea 83), y el cuaternio de rotación (mediante una conversión de la matriz en 84). La Figura 11.5 muestra un fragmento del XML exportado relativo a la escena base del ejemplo. Esta exportación puede realizarse ejecutando el fichero .py desde línea de órdenes, llamando a Blender con el parámetro *-P <script.py>* (siendo script.py el nombre del script de exportación).

```
<?xml version='1.0' encoding='UTF-8'?>
<data>
<graph>
  <vertex index="1" type="">
    <x>-2.500000</x> <y>-2.499999</y> <z>0.000000</z>
  </vertex>
  ...
  <vertex index="57" type ="drain">
    <x>-4.687499</x> <y>0.312501</y> <z>0.000000</z>
  </vertex>
  ...
  <edge> <vertex>5</vertex> <vertex>15</vertex> </edge>
  <edge> <vertex>6</vertex> <vertex>28</vertex> </edge>
  ...
</graph>
<camera index="1" fps="25">
  <path>
    <frame index="1">
      <position>
       <x>1.077411</x> <y>-15.124874</y> <z>6.665801</z>
      </position>
      <rotation>
        <x>0.836699</x> <y>0.544365</y> <z>-0.004856</z>  <w>0.059819</w>
      </rotation>
    </frame>
    <frame index="2">
      ...
```

Figura 11.5: Fragmento del resultado de la exportación del XML del ejemplo.



Capítulo 12

Shaders

Jorge López González

A lo largo de las siguiente páginas se introducirá y se enseñará a manejar, dentro del motor libre *Ogre 3D*, una de las herramientas más útiles que existen a la hora de sacar todo el partido al poder de nuestro hardware gráfico: los *shaders*. Veremos cómo, gracias a estos pequeños fragmentos de código, se nos permitirá tomar prácticamente todo el control sobre cómo nuestras escenas son renderizadas y también añadir una gran variedad de efectos que, hasta no hace mucho, eran imposibles en aplicaciones gráficas interactivas.

12.1. Un poco de historia

Para comenzar a entender lo que son los *shaders* es conveniente acercarse primero a la problemática que resuelven y, para ello, en esta sección, se hará un pequeño repaso a la historia de la generación de gráficos por computador, tanto en su vertiente interactiva como en la no interactiva, puesto que la historia de los *shaders* no es más que la historia de la lucha por conseguir mejorar, controlar y manipular a nuestro gusto el proceso de generación de imágenes por ordenador.

En los años 80, el desarrollo de aplicaciónes gráficas era realmente complicado, no sólo por la poca capacidad de las máquinas de la época, sino también por la falta de estandarización que existía en los APIs gráficos, por lo que era muy habitual que cada hardware necesitara de su propio software para poder ser usado. Esta dificultad hacía evidente, en aquella época, que los gráficos 3D no serían una cosa accesible para los ordenadores personales por lo menos durante un tiempo. Por lo tanto, los mayores avances en este campo estuvieron orientados

Año	**Tarjeta Gráfica**	**Hito**
1987	IBM VGA	Provee un *pixel framebuffer* que la CPU debe encargarse de llenar.
1996	3dfx Voodoo	Rasteriza y texturiza vértices pre-transformados de triángulos.
1999	NVIDIA GeForce 256	Aplica tanto transformaciones, como iluminación a los vértices. Usa una *fixed function pipeline*.
2001	NVIDIA GeForce 3	Incluye *pixel shader* configurable y *vertex shader* completamente programable.
2003	NVIDIA GeForce FX	Primera tarjeta con shaders completamente programables.

Tabla 12.1: Evolución del hardware de gráfico para PC.

hacia el renderizado no interactivo, ya que la principal utilidad que tenía la informática gráfica eran la investigación y el desarrollo de *CGI (Computer Generated Imagery)* para anuncios y películas.

Es en este punto donde la historia de los *shaders* comienza, en concreto, en la empresa *LucasFilms* a principios de los años 80. Por esas fechas el estudio decidió contratar programadores gráficos para que, comandados por *Edwin Catmull*, empezaran a informatizar la industria de los efectos especiales.

Se embarcaron en varios proyectos diferentes, uno de los cuales acabó siendo el germen de lo que más adelante se conocería como *Pixar Animation Studios*. Y fue en esta compañía donde, durante el desarrollo del API abierto *RISpec (RenderMan Interface Specification)*, se creó el concepto de *shader*. El propósito de *RISpec* era la descripción de escenas 3D para convertirlas en imágenes digitales fotorealistas. En este API se incluyó el *RenderMan Shading Language*, un lenguaje de programación al estilo C que permitía, por primera vez, que la descripción de materiales de las superficies no dependiera sólo de un pequeño conjunto de parámetros, sino que pudiera ser especificada con toda libertad.

Fixed-Function Pipeline

La programación gráfica antes de los *shaders* usaba un conjunto fijo de algoritmos que, colectivamente, son conocidos como la *fixed-function pipeline*. Esta, básicamente, permitía habilitar las distintas características y efectos, pudiendo manipular algunos parámetros, pero sin permitir un gran control sobre lo que ocurría en el proceso de renderizado.

RISpec fue publicado en 1988 y fue diseñado con la vista puesta en el futuro para, de esta manera, poder adaptarse a los avances en la tecnología durante un numero significativo de años. A las películas en que se ha usado nos remitimos para dar fe de que lo consiguieron (`http://www.pixar.com/featurefilms/index.html`).

El panorama para los gráficos en tiempo real, sin embargo, no era muy prometedor hasta que surgieron los primeros APIs estándar que abstraían el acceso al hardware gráfico. En 1992 apareció *OpenGL* y en 1995 *DirectX*. Además, en 1996 se ponía a la venta la primera tarjeta gráfica, la *3Dfx Voodoo*, que liberaba a la CPU de algunas de las tareas que implicaban la representación de gráficos por ordenador.

Estos dos elementos, combinados, permitieron dar el primer paso realmente serio para conseguir gráficos 3D espectáculares, en tiempo real, en ordenadores de escritorio.

Aun así, en los 90, tanto los APIs, como el hardware, ofrecían como único *pipeline* de procesamiento gráfico el *fixed-function pipeline* (FFP). El FFP permite varios tipos de configuración a la hora de establecer cómo se realizará el proceso de renderizado, sin embargo, estas posibilidades están predefinidas y, por tanto, limitadas. De este modo, aunque el salto de calidad era evidente, durante muchos años el renderizado interactivo estuvo muchísimo más limitado que su versión no interactiva.

En el lado no interactivo del espectro, las arquitecturas de alto rendimiento de renderizado por software usadas para el CG de las películas permitía llegar muchísimo más lejos en cuanto a la calidad de las imagenes generadas. *RenderMan* permitía a los artistas y programadores gráficos controlar totalmente el resultado del renderizado mediante el uso de este simple, pero potente, lenguaje de programación.

A partir del año 2000, con la evolución de la tecnología para construir el hardware gráfico y el incremento en la capacidad de procesamiento de los mismos, se acabó logrando trasladar la idea de *RenderMan* hacia el hardware gráfico de consumo. Pero esto no ocurrió hasta el año 2001, con la llegada de *DirectX 8.0* y la *NVIDIA GeForce3*, que introdujo por primera vez un pipeline gráfico programable, aunque limitado.

Y desde entonces, con el paso de los años, el hardware y los API gráficos no han hecho sino dar enormes saltos hacia delante, tanto en funcionalidad, como en rendimiento (un ejemplo de la evolución de las capacidades de las tarjetas gráficas se presenta en la tabla 12.1). Lo cual nos ha llevado al momento actual, en el que los FFP han sido casi sustituidos por los *programmable-function pipelines* y sus *shader* para controlar el procesamiento de los gráficos.

12.1.1. ¿Y qué es un Shader?

Una de las definiciones clásicas de *shader* los muestra como: *"piezas de código, que implementan un algoritmo para establecer como responde un punto de una superficie a la iluminación"*. Es decir, sirven para establecer el color definitivo de los pixeles que se mostrarán en pantalla.

Como veremos, esta definición ya no es del todo correcta. Los *shader* actuales cumplen muchas más funciones. Hoy por hoy, el propósito de los *shader* es: *"Especificar la forma en que se renderiza un conjunto de geometría"*, definición extraida del libro *"GPU Gems"* [Fer04], ideal para aprender sobre técnicas avanzadas de programación gráfica.

Desde un punto de vista de alto nivel, los shader nos permiten tratar el estado de renderizado como un recurso, lo cual es extremadamente poderoso porque permite que el dibujado de una aplicación, es decir, la configuración del dispositivo que se encarga de ello, sea casi completamente dirigido por recursos (de la misma forma en que la geometría y las texturas son recursos externos a la aplicación).

Sin embargo, la mejor manera de explicar qué son y cómo funcionan, es repasando el funcionamiento del *pipeline* gráfico: desde la transmisión de los vértices a la *GPU*, hasta la salida por pantalla de la imagen generada.

12.2. Pipelines Gráficos

En esta sección se realizará un seguimiento al funcionamiento interno de los dos tipos de *pipelines gráficos*, para que de esta forma quede clara la forma en que trabajan los *shader* y cual es exáctamente la función que desempeñan en el proceso de renderizado. Además, es muy importante conocer el funcionamiento del *pipeline* a la hora de desarrollar *shaders*.

En las secciones 1.2 y 1.3 del capitulo 1 se explica con mas detalle el *pipeline gráfico* usado en las aplicaciónes gráficas interactivas.

La tendencia dominante hoy en día en el diseño de hardware gráfico es enfocar un mayor esfuerzo en ofrecer un *pipeline* cuanto más programable mejor.

Antes de empezar, se explicará de forma breve el por qué de la organización de las GPUs como *pipelines* y por qué se usan procesadores específicos.

12.2.1. ¿Por qué un pipeline gráfico?

El motivo por el que se usan tarjetas gráficas busca, en definitiva, que la CPU delegue en la medida de lo posible la mayor cantidad de trabajo en la GPU. El motivo tiene su razón de ser en dos factores: el primero es que las CPU son procesadores de propósito general y la tarea de procesamiento de gráficos tiene características muy específicas, y el segundo, tenemos que muchas aplicaciones actuales (videojuegos, simulaciones, diseño gráfico, etc...) requieren de *rendering* interactivo con la mayor calidad posible.

La tarea de generar imagenes por ordenador suele implicar el proceso de un gran número de elementos, sin embargo, da la casualidad de que la mayor parte de las veces no hay dependencias entre ellos. El procesamiento de un vértice, por ejemplo, no depende del procesamiento de los demás o, tomando un caso en el que quisieramos aplicar iluminación local, se puede apreciar que la iluminación de un punto no depende de la de los demás.

Además, el procesamiento de los gráficos es áltamente paralelizable, puesto que los elementos involucrados suelen ser magnitudes vectoriales reales. Tanto la posición, como el color y otros elementos geométricos, se representan cómodamente mediante vectores a los que se aplican diferentes algoritmos. Estos, generalmente, suelen ser bastante simples y poco costosos computacionalmente (las operaciones más comunes en estos algorimtos son la suma, multiplicación o el producto escalar).

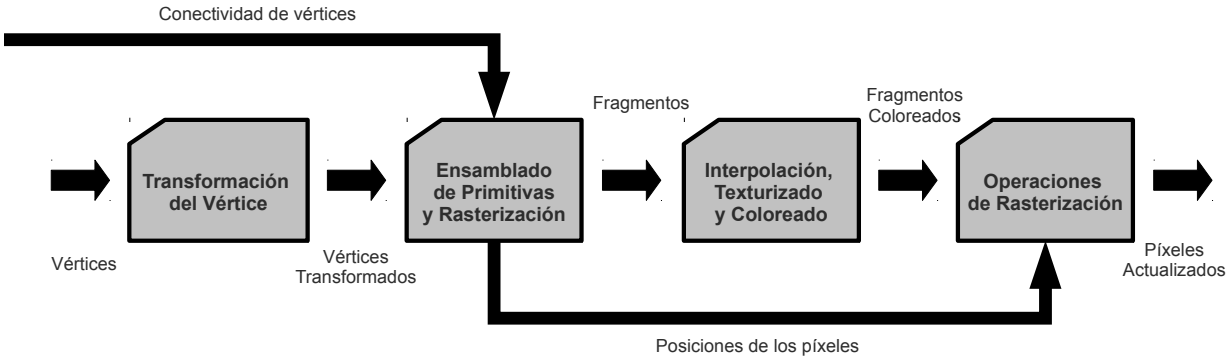


Figura 12.1: Pipeline del hardware gráfico simplificado.

Primitiva

Una primitiva es una región de una superficie definida por sus vértices. La primitiva más usada en informática gráfica es el triángulo, puesto que sus tres vértices siempre definen un plano.

En consecuencia, las GPUs se diseñan como *Stream Processors*. Estos procesadores están especializados en procesar gran cantidad de elementos en paralelo, distribuyéndolos en pequeñas unidades de procesamiento (etapas del *pipeline*), las cuales disponen de operaciones para tratar con vectores de forma eficiente y donde algunas son programables.

12.2.2. Fixed-Function Pipeline

Batch

Es conveniente recordar que la GPU no suele recibir toda la información de la escena de golpe, sino que se le envían para dibujar varios grupos de primitivas cada vez, agrupados en la unidad de trabajo conocida como *batch*.

Para seguir el proceso tomaremos como referencia el *pipeline gráfico* simplificado de la figura 12.1, que representa las etapas de un *fixed-function pipeline*. Este tipo de *pipeline* no ofrece mucha libertad en la manipulación de sus parámetros pero, sin embargo, su aparición supuso todo un avance en su momento.

Como entrada, el pipeline recibe el conjunto de vértices correspondientes a la geometría de la escena. Un vértice en el contexto en el que nos encontramos se corresponde con la posición de un punto en el espacio 3D, perteneciente a una superficie, para el que se conocen los valores exactos de ciertas propiedades (conocidas como componentes del vértice).

Textura

Es una matriz que almacena los valores de una propiedad de la superficie en los puntos interiores de las primitivas. Es una de las maneras para obtener un mayor nivel de detalle en la representación de los modelos 3D.

Estas propiedades pueden incluir atributos como el color de la superficie (primario y secundario, si tiene componente especular), uno o varios conjuntos de coordenadas de textura, o su vector normal que indicará la dirección en que la superficie está orientada con respecto al vértice y que se usa para los cálculos de iluminación.

Transformación de Vértices

El conjunto de vértices primero atraviesa esta etapa de procesamiento del pipeline gráfico, en la que se realizan una serie de operaciones matemáticas sobre los mismos. Estas operaciones incluyen las transformaciones necesarias para convertir la posición del vértice

a la posición que tendrá en pantalla y que será usada por el rasterizador, la generación de las coordenadas de textura y el cálculo de la iluminación sobre el vértice para conocer su color.

Ensamblado de Primitivas y Rasterización

Los vértices transformados fluyen en secuencia hacia la siguiente etapa, donde el ensamblado de primitiva toma los vértices y los une para formar las primitivas correspondientes gracias a la información recibida sobre la conectividad de los mismos que indica cómo se ensamblan.

Esta información se transmite a la siguiente etapa en unidades discretas conocidas como *batches*.

El resultado del ensamblado da lugar a una secuencia de triángulos, líneas o puntos, en la cual no todos los elementos tienen porque ser procesados. A este conjunto, por lo tanto, se le pueden aplicar dos procesos que aligerarán la carga de trabajo del hardware gráfico.

Por un lado las primitivas pueden ser descartadas mediante el proceso de *clipping*, en el cual se selecciona sólo a aquellas que caen dentro del volumen de visualización (la región visible para el usuario de la escena 3D, conocido también como *view frustum* o pirámide de visión).

Y por otro lado, el rasterizador puede descartar también aquellas primitivas cuya cara no esté apuntando hacia el observador, mediante el proceso conocido como *culling*.

Las primitivas que sobreviven a estos procesos son rasterizados. La rasterización es el proceso mediante el que se determina el conjunto de pixeles que cubren una determinada primitiva. Polígonos, líneas y puntos son rasterizados de acuerdo a unas reglas especificadas para cada tipo de primitiva. El resultado de la rasterización son un conjunto de localizaciones de pixeles, al igual que un conjunto de fragmentos. No hay ninguna relación entre el conjunto de fragmentos generados en la rasterización y el número de vértices que hay en una primitiva. Un triángulo que ocupe toda la pantalla provocará la creación de millones de fragmentos.

Framebuffer

Los *framebuffer* son dispositivos gráficos que ofrecen una zona de memoria de acceso aleatorio, que representa cada uno de los pixeles de la pantalla.

Habitualmente se usan indistintamente los términos pixel y fragmento para referirse al resultado de la fase de rasterización. Sin embargo, existe una diferencia importante entre estos dos términos. Pixel proviene de la abreviatura de *"picture element"* y representa el contenido del *framebuffer* en una localización específica, al igual que el color, profundidad y algunos otros valores asociados con esa localización. Un fragmento sin embargo, es el estado potencialmente requerido para actualizar un pixel en particular.

El término *"fragmento"* es usado porque la rasterización descompone cada primitiva geométrica, como puede ser un triángulo, en fragmentos del tamaño de un pixel por cada pixel que la primitiva cubre. Un fragmento tiene asociada una localización para el pixel, un valor de profundidad, y un conjunto de parámetros interpolados como son: el color primario, el color especular, y uno o varios conjuntos de coordenadas de textura. Estos parámetros interpolados son derivados de

los parámetros incluidos en los vértices transformados de la primitiva que generó los fragmentos. Se podría pensar en los fragmentos como píxeles potenciales. Si un fragmento pasa los distintos tests de rasterización, el fragmento actualiza el pixel correspondiente en el *framebuffer*.

Interpolación, texturizado y coloreado

Una vez las primitivas han sido rasterizadas en una colección de cero o más fragmentos, la fase de interpolación, texturizado y coloreado, se dedica precisamente a eso, a interpolar los parámetros de los fragmentos como sea necesario, realizando una secuencia de operaciones matemáticas y de texturizado, que determinan el color final de cada fragmento.

Como complemento a la determinación del color de los fragmentos, esta etapa puede encargarse también de calcular la profundidad de cada fragmento, pudiendo descartarlos y así evitar la actualización del correspondiente pixel en pantalla. Debido a que los fragmentos pueden ser descartados, esta etapa devuelve entre uno y cero fragmentos coloreados por cada uno recibido.

Scissor's Test

Este test permite restringir el área de dibujado de la pantalla, descartando así, todos aquellos fragmentos que no entren dentro.

Operaciones de Rasterización

La fase en que se ejecutan las operaciones de *rasterización* pone en marcha una secuencia de tests para cada cada fragmento, inmediatamente antes de actualizar el *framebuffer*. Estas operaciones son una parte estándar de *OpenGL* y *Direct3D*, e incluyen: el *scissor test*, *alpha test*, *stencil test* y el *depth test*. En ellos están involucrados el color final del fragmento, su profundidad, su localización, así como su valor de *stencil*.

Alpha Test

Permite descartar fragmentos comparando el valor de *alpha* de cada fragmento, con un valor constante especificado.

Si cualquier test falla, esta etapa descarta el fragmento sin actualizar el valor de color del pixel (sin embargo, podría ocurrir una operación de escritura para el valor *stencil*). Pasar el *depth test* puede reemplazar el valor de profundidad del pixel, por el valor de profundidad del fragmento. Despues de los tests, la operación de *blending* combina el color final del fragmento con el valor de color del pixel correspondiente. Finálmente, con una operación de escritura sobre el *framebuffer* se reemplaza el color del pixel, por el color mezclado.

Stencil Test

A partir del uso del *stencil buffer*, hace una comparación con el *framebuffer*, y descarta aquellos fragmentos que no superen la condición especificada. Como si usara una máscara, o una plantilla, para especificar qué se dibuja y qué no.

Conclusión

Tras concluir esta serie de pasos obtenemos, al fín, en el *framebuffer*, la imagen generada a partir de nuestra escena 3D. La cual podrá ser volcada a la pantalla, o usada para algún otro propósito.

Depth Test

A partir del valor de profundidad del fragmento establece qué fragmento está más cerca de la cámara y, en función de la condición especificada, lo descarta o no.

Como se puede apreciar, la libertad para influir en el proceso de renderizado en este tipo de *pipeline* está muy limitada. Cualquier transformación sobre los vértices debe hacerse mediante código de la aplicación (siendo la CPU la encargada de ejecutarlo) y se limita a ofrecer los diferentes tests comentados que son los que proveen algo de control sobre la forma en que se renderizan las imagenes.

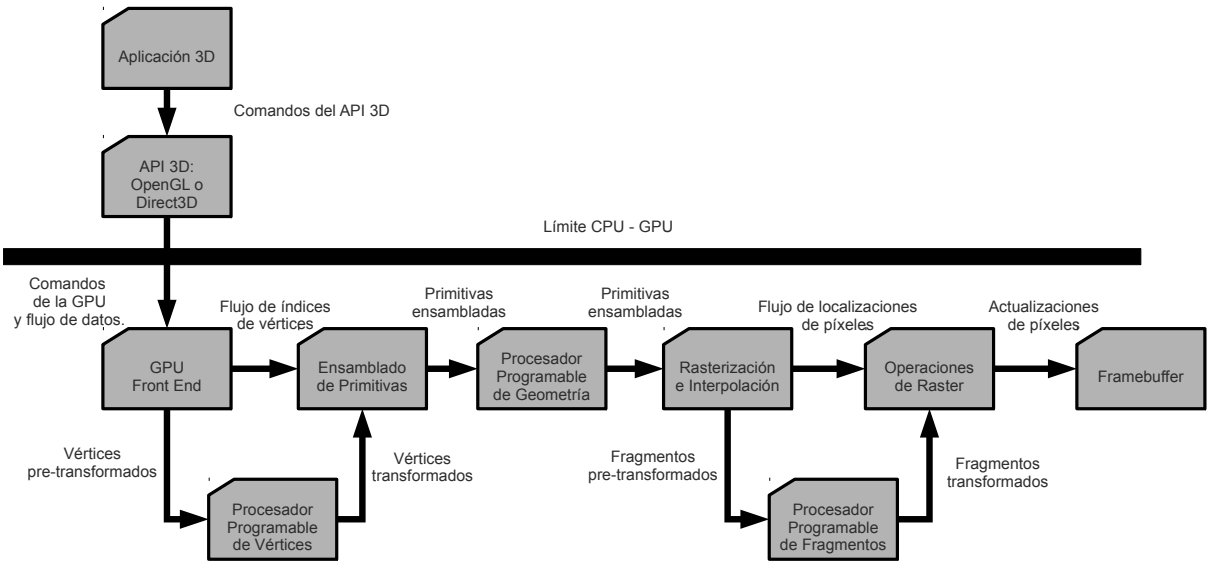


Figura 12.2: Pipeline del hardware gráfico programable simplificado.

12.2.3. Programmable-Function Pipeline

Gracias al *programmable-function pipeline* muchas de las operaciones que antes asumía la CPU, definitivamente pasan a manos de la GPU (caso de la interpolación de movimiento en los vértices de un modelo, por ejemplo).

La figura 12.2 muestra las etapas de procesamiento de vértices y fragmentos en una GPU con un pipeline programable simplificado. En el, se puede apreciar que se mantienen las mismas etapas que en el *fixed-function pipeline*, pero se añaden tres etapas nuevas en las que los *shader* se encargan de establecer cómo se procesan los vértices, primitivas y fragmentos.

A continuación, en la siguiente sección, se explica someramente cómo funcionan estas nuevas etapas, que se corresponden con cada uno de los tipos de *shader* que existen hoy por hoy.

12.3. Tipos de Shader

Originalmente los *shaders* sólo realizaban operaciones a nivel de pixel (o fragmento). Los que hoy se conocen como *fragment/pixel shaders*. A lo largo del tiempo se han introducido más tipos de *shader*], por lo que ahora el término *shader* se ha vuelto mucho más genérico, abarcando los tres tipos que se usan hoy en día en la generación de gráficos en tiempo real.

Vertex Shader

El flujo de datos del procesamiento de vértices comienza cargando los atributos de cada vértice (posición, color, coordenadas de textura, etc...) en el procesador de vértices. Este, va ejecutando las distintas operaciones secuencialmente para cada vértice hasta que termina. El resultado de esta etapa es un vértice transformado en función de las instrucciónes del *shader*. Después del ensamblaje de la primitiva geométrica y de la rasterización, los valores interpolados son pasados al procesador de fragmentos.

Como ya se ha comentado, los *vertex shader* tienen acceso y pueden modificar los atributos de los vértices. A su vez, se permite acceso para todos los vértices a lo que se conoce como variables *uniformes*. Estas son variables globales de sólo lectura que permiten acceder a información que no cambia a lo largo de una pasada como es, por ejemplo, la matriz mundo/vista o la delta de tiempo (tiempo pasada entre una vuelta del bucle principal y otra).

Un *vertex shader* recibe como entrada un vértice y devuelve siempre un sólo vértice, es decir, no puede crear nueva geometría.

Es importante recalcar que cada *vertex shader* afecta a un sólo vértice, es decir, se opera sobre vértices individuales no sobre colecciones de ellos. A su vez, tampoco se tiene acceso a la información sobre otros vértices, ni siquiera a los que forman parte de su propia primitiva.

En el listado de código 12.1 se muestra un ejemplo de este tipo de *shaders*. Cómo se puede ver, la sintaxis es muy parecida a la de un programa escrito en C.

Listado 12.1: Ejemplo de Vertex Shader

```
// Estructura de salida con la información del vertice procesado
struct tVOutput {
    float4  position:   POSITION;
    float4  color   :   COLOR;
};

// Usamos la posición y color del vértice
tVOutput v_color_passthrough(
    float4          position    :   POSITION,
    float4          color       :   COLOR
    uniform float4x4    worldViewMatrix)
{
    tVOutput    OUT;
    // Transformación del vértice
    OUT.position = mul(worldViewMatrix, position);
    OUT.color = color;

    return OUT;
}
```

El fragmento de código es bastante explicativo por si mismo porque, básicamente, no hace nada con el vértice. Recibe algunos parámetros (color y posición) y los devuelve. La única operación realizada tiene que ver con transformar la posición del vértice a coordenadas de cámara.

Fragment Shader

Los procesadores de fragmentos requieren del mismo tipo de operaciones que los procesadors de vértices con la salvedad de que, en estos, se tiene acceso a las operaciones de texturizado. Estas operaciones permiten al procesador acceder a la imagen, o imagenes, usadas de textura y permiten manipular sus valores.

Los *fragment shader* tienen como propósito modificar cada fragmento individual que les es suministrado desde la etapa de *rasterizacion*.

Estos tienen acceso a información como es: la posición del fragmento, los datos interpolados en la *rasterizacion* (color, profundidad, coordenadas de textura), así como a la textura que use la primitiva a la cual pertenece (en forma de variable uniforme, como los *vertex shader*), pudiendo realizar operaciones sobre todos estos atributos.

Al igual que los *vertex shader*, sólo puede procesar un fragmento cada vez y no puede influir sobre, ni tener acceso a, ningún otro fragmento.

Listado 12.2: Ejemplo de Fragment Shader

```
1  // Estructura de salida con la información del fragmento procesado
2  struct tFOutput {
3      float4 color : COLOR;
4  };
5
6  // Devuelve el color interpolado de cada fragmento
7  tFOutput f_color_passthrough(
8      float4 color : COLOR)
9  {
10     tFOutput OUT;
11     // Se aplica el color al fragmento
12     OUT.color = color;
13
14     return OUT;
15 }
```

En el listado de código 12.2 se muestra un ejemplo de este tipo de *shaders*.

Geometry Shader

Este es el más nuevo de los tres tipos de shader. Puede modificar la geometría e incluso generar nueva de forma procedural. Al ser este un tipo de *shader* muy reciente todavía no está completamente soportado por las tarjetas gráficas y no se ha extendido lo suficiente.

La etapa encargada de procesar la geometría estaría enclavada entre las etapas de ensamblado de primitivas y la de rasterización e interpolación (ver Figura 12.2).

Este tipo de *shader* recibe la primitiva ensamblada y, al contrario que los *vertex shader*, si tiene conocimiento completo de la misma. Para cada primitiva de entrada, tiene acceso a todos los vértices, así como a la información sobre cómo se conectan.

En esta sección no se tratarán, pero cualquiera que lo desee puede encontrar más información al respecto en el capítulo 3 del libro *"Real-Time Rendering"* [AMHH08].

12.4. Aplicaciones de los Shader

Existen múltiples y diferentes aplicaciones para los shaders. En esta sección se enumeran algunas de las funciones que pueden ayudar a cumplir.

12.4.1. Vertex Skinning

Los vértices de una superficie, al igual que el cuerpo humano, son movidos a causa de la influencia de una estructura esquelética. Como complemento, se puede aplicar una deformación extra para simular la dinámica de la forma de un músculo.

En este caso el *shader* ayuda a establecer cómo los vértices se ven afectados por el esqueleto y aplica las transformaciones en consecuencia.

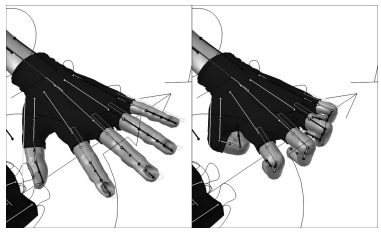


Figura 12.3: Ejemplos de *skinning* (Fuente: Wikipedia).

12.4.2. Vertex Displacement Mapping

Los vértices pueden ser desplazados, por ejemplo verticalmente, en función de los resultados ofrecidos por un algoritmo o mediante la aplicación de un mapa de alturas (una textura en escala de grises), consiguiendo con ello, por ejemplo, la generación de un terreno irregular.

12.4.3. Screen Effects

Los *shader* pueden ser muy útiles para lograr todo tipo de efectos sobre la imagen ya generada tras el renderizado. Gracias a las múltiples pasadas que se pueden aplicar, es posible generar todo tipo de efectos de post-procesado, del estilo de los que se usan en las películas actuales.

Figura 12.4: Ejemplos de *vertex displacement mapping* (Fuente: Wikipedia).

Los *fragment shader* realizan el renderizado en una textura temporal (un *framebuffer* alternativo) que luego es procesada con filtros antes de devolver los valores de color.

Figura 12.5: Ejemplo de *glow* o *bloom* (Fuente: Wikipedia).

12.4.4. Light and Surface Models

Mediante *shaders* es posible calcular los nuevos valores de color aplicando distintos modelos de iluminación, lo cual involucra parametros como son las normales de las superficies (N), angulo en el que incide la luz (L), angulo de la luz reflejada (R) y el ángulo de visión.

12.4.5. Non-Photorealistic Rendering

Los modelos de iluminación no tienen porque limitarse a imitar el "mundo real", pueden asignar valores de color correspondientes a mundos imaginarios, como puede ser el de los dibujos animados o el de las pinturas al oleo.

Figura 12.6: Diferencia entre *Per-Vertex Lighting* (izquierda) y *Per-Pixel Lighting* (derecha).

Figura 12.7: Ejemplo de *Toon Shading* (Fuente: Wikipedia).

12.5. Lenguajes de Shader

Existe una gran variedad de lenguajes para escribir *shaders* en tiempo real y no parece que vayan a dejar de aparecer más. Estos pueden agruparse en dos categorías: los basados en programas individuales o los que se basan en ficheros de efectos.

La primera aproximación necesita que se cree una colección de ficheros, cada uno de los cuales implementa un *Vertex Shader* o un *Fragment Shader* en una pasada de renderizado. Ejemplos de estos lenguajes son: *Cg*, *HLSL* o *GLSL*. Lo cual conlleva una cierta complejidad en su gestión pues para cada pasada de renderizado, en los efectos complejos, necesita de dos ficheros diferentes.

Para solucionar este problema surgen el siguiente tipo de lenguajes que son aquellos basados ficheros de efectos, de los cuales, el más conocido es el *CgFX* de *NVidia*, y que a su vez es un super conjunto del *Microsoft effect framework*.

Los lenguajes basados en ficheros de efectos permiten que los diferentes *shader* se incluyan en un mismo fichero y, además, introducen dos nuevos conceptos: técnica y pasada. En los cuales pueden ser agrupados los diferentes *shaders* y permiten que el estado del dispositivo gráfico pueda ser definido en cada pasada de renderizado.

Con esta aproximación se consigue que la gestión sea más manejable, mejor dirigida por recursos y mucho más poderosa.

12.6. Desarrollo de shaders en Ogre

Para crear y gestionar los diferentes *shaders* en Ogre se usa la aproximación basada en ficheros de efectos. Los cuales ya se han comentado en el capítulo 7.

Con el objetivo de hacer lo más sencillo el aprendizaje, primero se explicará cómo montar la escena a mostrar y más tarde se explicará qué es lo que ha pasado.

En estos ejemplos, se usará el lenguaje *Cg*, del que se puede encontrar una documentación muy completa en el libro *"The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics"* [FK03], que, afortunadamente, puede encontrarse online, y de forma gratuita, en la página de desarrolladores de NVIDIA:

`http://developer.nvidia.com/object/cg_tutorial_home.html`

También podemos recurrir al capítulo correspondiente a los materiales y *shaders* de *"Ogre 3D 1.7 Beginner's Guide"* [Ker10] que puede ser un buen punto de partida para empezar con este tema.

12.6.1. Poniendo a punto el entorno

Antes de empezar, debemos tener claro, que para usar los *shader* en Ogre, debemos contar con dos tipos de ficheros. Por un lado tendremos los *programas*, o archivos *.cg*, en los que se encontrarán los *shaders* escritos en *Cg*, y por otro lado tendremos los archivos de efectos, o materiales, que definirán cómo se usan nuestros *programas*

Por lo tanto, lo primero debería ser dejar claro donde se han colocar los ficheros correspondientes a los shader, para que así puedan ser usados por Ogre. Estos se deben colocar en la ruta:

```
/[directorio_ogre]/media/materials/programs
```

Si queremos usar otro directorio, deberemos indicarlo en el archivo *resources.cfg*, bajo la etiqueta *[popular]* por ejemplo (hay que tener cuidado sin embargo con lo que se coloca bajo esa etiqueta en un desarrollo serio).

```
[Popular]
...
FileSystem=../../media/materials/programs/mis_shaders
...
```

Por otro lado, es conveniente tener a mano el fichero de *log* que genera Ogre en cada ejecución, para saber por qué nuestro *shader* hace que todo aparezca blanco (o lo que es lo mismo ¿por qué ha fallado su compilación?).

Log de Ogre

En el fichero de log podremos encontrar información sobre los perfiles y funcionalidad soportada por nuestra tarjeta gráfica, así cómo información sobre posibles errores en la compilación de nuestros materiales o shaders.

12.6.2. Primer Shader

Este primer *shader* servirá de toma de contacto para familiarizarnos con algunos de los conceptos que introduce. Es por eso que no hace prácticamente nada, sólo deja que el vértice pase a través de el, modificando únicamente su color y enviándolo a la siguiente etapa.

La escena 3D puede ser cualquiera, siempre y cuando alguna de las entidades tenga asociado el material que se definirá ahora, aunque antes hay que indicarle a Ogre alguna información sobre los *shader* que queremos que use, la cual debe estar incluida en el propio archivo de material.

Al definir el *shader* que usará nuestro material hay que indicar al menos:

- El nombre del *shader*
- En qué lenguaje está escrito
- En qué fichero de código está almacenado
- Cómo se llama el punto de entrada al shader
- En qué perfil queremos que se compile

Por último, antes de definir el material en si, hay que indicar también a los shader cuales son aquellos parámetros que Ogre les pasará. En este caso sólo pasamos la matriz que usaremos para transformar las coordenadas de cada vértice a coordenadas de cámara. Es importante definir esta información al principio del fichero de material.

Listado 12.3: Declaración del *vertex shader* en el material

```
1  // Declaracion vertex shader y lenguaje usado
2  vertex_program VertexGreenColor cg
3  {
4      // Archivo con los programas
5      source firstshaders.cg
6
7      // Punto de entrada al shader
8      entry_point v_green_color
9
10     // Perfiles validos
11     profiles vs_1_1 arbvp1
12
13     // Parámetros usados
14     default_params
15     {
16         param_named_auto worldViewMatrix worldviewproj_matrix
17     }
18 }
```

Para este caso, el material quedaría tal y como se ve en el listado 12.4. Y queda claro cómo en la pasada se aplican los dos *shader*.

Listado 12.4: Primer Material con shaders

```
vertex_program VertexGreenColor cg
{
    source firstshaders.cg
    entry_point v_green_color
    profiles vs_1_1 arbvp1

    default_params
    {
        param_named_auto worldViewMatrix worldviewproj_matrix
    }
}

fragment_program FragmentColorPassthrough cg
{
    source firstshaders.cg
    entry_point f_color_passthrough
    profiles ps_1_1 arbfp1
}

material VertexColorMaterial
{
    technique
    {
        pass
        {
            vertex_program_ref VertexGreenColor
            {
            }

            fragment_program_ref FragmentColorPassthrough
            {
            }
        }
    }
}
```

Los dos programas que definen nuestros primeros *fragment* y *vertex shader* aparecen en los listados 12.5 y 12.6, y los dos deberían incluirse en el fichero *firstshaders.cg* (o en el archivo que queramos), tal y como indicamos en el material.

El *vertex shader* recibe cómo único parámetro del vértice su posición, esta es transformada a coordenadas del espacio de cámara gracias a la operación que realizamos con la variable que representa la matriz de transformación.

Por último se asigna el color primario al vértice, que se corresponde con su componente difusa, y que en este caso es verde.

Declaración shaders

Para más información sobre la declaración de los *shader*, sería buena idea dirigirse al manual en: http://www.ogre3d.org/-docs/manual/-manual_18.html

Listado 12.5: Primer vertex shader

```
// Estructura de salida con la información del vertice procesado
struct tVOutput {
    float4  position:   POSITION;
    float4  color   :   COLOR;
};

// Cambia el color primario (diffuse component) de cada vertice a
    verde
tVOutput v_green(
    float4  position : POSITION,
    uniform float4x4 worldViewMatrix)
{
    tVOutput OUT;
    // Transformamos la posición del vértice
    OUT.position = mul(worldViewMatrix, position);
    // Asignamos el valor RGBA correspondiente al verde
    OUT.color = float4(0, 1, 0, 1);

    return OUT;
}
```

Listado 12.6: Primer fragment shader

```
// Estructura de salida con la información del fragmento procesado
struct tFOutput {
    float4 color : COLOR;
};

// Devuelve el color interpolado de cada fragmento
tFOutput f_color_passthrough(
    float4 color : COLOR)
{
    tFOutput OUT;
    // Se aplica el color al fragmento
    OUT.color = color;

    return OUT;
}
```

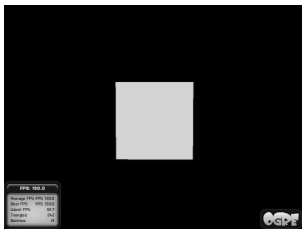


Figura 12.8: Resultado del uso de los primeros *shader*.

El *fragment shader* se limita símplemente a devolver el color interpolado (aunque en este caso no se note) y el resultado debería ser como el mostrado en la Figura 12.8

Cualquiera con conocimientos de C o C++, no habrá tenido mucha dificultad a la hora de entender los dos listado de código anteriores. Sin embargo si habrá notado la presencia de algunos elementos nuevos, propios de Cg. Y es que hay algunas diferencias importantes, al tratarse de un lenguaje tan especializado.

En primer lugar, en Cg no hay necesidad de especificar qué tipos de elementos o librerías queremos usar (como se hace en C o C++ con los `#include`). Automáticamente se incluye todo lo necesario para un programa escrito en este lenguaje.

Por otro lado, tenemos un conjunto completamente nuevo de tipos de datos, que se encargan de representar vectores y matrices.

En los lenguajes de programación clásicos, los tipos de datos representan magnitudes escalares (`int`, o `float` son ejemplos de ello). Pero,

cómo hemos visto en la sección 12.2.3, las GPU están preparadas para tratar con vectores (entiendolos como una colección de magnitudes escalares). Los vectores en C o C++ pueden representarse fácilmente mediante arrays de valores escalares, sin embargo, como su procesamiento es fundamental en el tratamiento de vértices o fragmentos, Cg ofrece tipos predefinidos para estas estructuras.

Para el tratamiento de vectores, podemos usar los siguientes tipos:

```
float2 uv_coord;

float3 position;

float4 rgba_color;
```

O sus equivalentes con la mitad de precisión:

```
half2 uv_coord;

half3 position;

half4 rgba_color;
```

Estos tipos de datos son mucho más eficientes que el uso de un array, por lo que no es recomendable sustituir un `float4 X`, por un `float X[4]`, ya que no son exactamente lo mismo. Cg se encarga de almacenar esta información en una forma, mediante la cual, es posible sacar mucho mejor partido de las características de una GPU a la hora de operar con estos elementos.

Por otro lado, Cg soporta nativamente tipos para representar matrices. Ejemplos de declaraciones serían:

```
float4x4 matrix1;

float2x4 matrix2;
```

Al igual que los vectores, el uso de estos tipos garantiza una forma muy eficiente de operar con ellos.

Otra de las cosas que seguramente pueden llamar la atención, es la peculiar forma en que se declaran los tipos de los atributos pertenecientes a la estructura usada *tVOutput*, o *tFOutput*.

```
float4 position : POSITION;
```

A los dos puntos, y una palabra reservada tras la declaración de una variable (como **POSITION** o **COLOR**), es lo que, en Cg, se conoce como *semántica*. Esto sirve para indicar al *pipeline* gráfico, qué tipo de datos deben llenar estas variables.

Es decir, como ya sabemos, los shader son programas que se ejecutan en ciertas etapas del *pipeline*. Por lo tanto, estas etapas reciben cierta información que nosotros podemos usar indicando con la *semántica* cuáles son estos datos. Los únicos sitios donde se pueden usar son en estructuras de entrada o salida (como las de los ejemplos) o en la definición de los parámetros que recibe el punto de entrada (método principal) de nuestro *shader*.

Por último, es necesario hablar brevemente sobre la palabra reservada *uniform*, su significado, y su conexión con lo definido en el fichero de material.

El *pipeline* debe proveer de algún mecanismo para comunicar a los *shader* los valores de aquellos elementos necesarios para conocer el estado de la simulación (la posición de las luces, la delta de tiempo, las matrices de transformación, etc...). Esto se consigue mediante el uso de las variable declaradas como *uniform*. Para Cg, estas son variables externas, cuyos valores deben ser especificados por otro elemento.

En la declaración del *vertex shader* en el material (listado 12.4) se puede ver cómo se declara el parámetro por defecto `param_named_auto worldViewMatrix worldviewproj_matrix`, que nos permite acceder a la matriz de transformación correspondiente mediante un parámetro *uniform*.

Para más información sobre esto, se pueden consultar los capítulos 2 y 3 del muy recomendable *"The Cg Tutorial"* [FK03] en:

`http://developer.nvidia.com/node/91`

y en:

`http://developer.nvidia.com/node/90`

12.6.3. Comprobando la interpolación del color

Con este segundo *shader*, se persigue el objetivo de comprobar cómo la etapa de rasterización e interpolación hace llegar al *fragment shader* los fragmentos que cubren cada primitiva con su componente de color interpolado a partir de los valores de color de los vértices.

Para ello es necesario que creemos una escena especialmente preparada para conseguir observar el efecto.

Listado 12.7: Escena definida en Ogre

```
1  void CPlaneExample::createScene(void)
2  {
3      // Creamos plano
4      Ogre::ManualObject* manual = createManualPlane();
5      manual->convertToMesh("Quad");
6
7      // Creamos la entidad
8      Ogre::Entity* quadEnt = mSceneMgr->createEntity("Quad");
9
10     // Lo agregamos a la escena
11     Ogre::SceneNode* quadNode = mSceneMgr->getRootSceneNode()->
           createChildSceneNode("QuadNode");
12
13     quadNode->attachObject(ent);
14 }
15
16 Ogre::ManualObject* CPlaneExample::createManualPlane() {
17     // Creamos un cuadrado de forma manual
18     Ogre::ManualObject* manual = mSceneMgr->createManualObject("
           Quad");
19
20     // Iniciamos la creacion con el material correspondiente al
           shader
21     manual->begin("VertexColorMaterial", Ogre::RenderOperation::
           OT_TRIANGLE_LIST);
22
```

```
23      // Situamos vértices y sus correspondientes colores
24      manual->position(5.0, 0.0, 0.0);
25      manual->colour(1, 1, 1);
26
27      manual->position(-5.0, 10.0, 0.0);
28      manual->colour(0, 0, 1);
29
30      manual->position(-5.0, 0.0, 0.0);
31      manual->colour(0, 1, 0);
32
33      manual->position(5.0, 10.0, 0.0);
34      manual->colour(1, 0, 0);
35
36      // Establecemos los indices
37      manual->index(0);
38      manual->index(1);
39      manual->index(2);
40
41      manual->index(0);
42      manual->index(3);
43      manual->index(1);
44
45      manual->end();
46
47      return manual;
48  }
```

El *vertex shader* lo único que tendrá que hacer es dejar pasar el vértice, pero esta vez, en vez de cambiarle el color dejamos que siga con el que se le ha asignado. Para ello, en el método que define el punto de entrada del *shader* hay que indicarle que recibe como parametro el color del vértice.

Listado 12.8: Segundo vertex shader

```
1  // Estructura de salida con la información del vertice procesado
2  struct tVOutput {
3      float4  position:   POSITION;
4      float4  color   :   COLOR;
5  };
6
7  // Usamos la posición y color del vértice
8  tVOutput v_color_passthrough(
9      float4          position    :   POSITION,
10     float4          color       :   COLOR
11     uniform float4x4    worldViewMatrix)
12 {
13     tVOutput    OUT;
14     // Transformación del vértice
15     OUT.position = mul(worldViewMatrix, position);
16     OUT.color = color;
17
18     return OUT;
19 }
```

El *fragment shader* podemos dejarlo igual, no es necesario que haga nada. Incluso es posible no incluirlo en el fichero de material (cosa que no se puede hacer con los *vertex shader*).

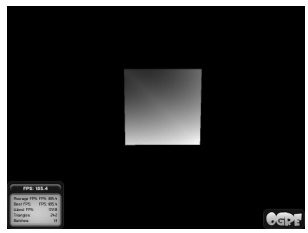


Figura 12.9: El cuadrado aparece con el color de sus cuatro vértices interpolado.

Figura 12.10: Ogro dibujado como si tuviera un mapa de normales encima.

El resultado obtenido, debería ser el mostrado en la figura 12.9.

Ahora que sabes cómo modificar el color de los vértices. ¿Serías capaz de pintar la entidad típica del Ogro como si de textura tuviera un mapa de normales? Figura 12.10 Tip: Has de usar la normal del vértice para conseguirlo.

12.6.4. Usando una textura

Pintar un modelo con el color de sus vértices puede ser interesante, pero desde luego no es muy impresionante. Por lo tanto con este tercer *shader* usaremos una textura para darle algo de realismo a nuestro plano anterior.

Listado 12.9: Creación de un plano al que se le asignan coordenadas de textura.

```
// Situamos vértices y coordenadas de textura
manual->position(5.0, 0.0, 0.0);
manual->textureCoord(0, 1);

manual->position(-5.0, 10.0, 0.0);
manual->textureCoord(1, 0);

manual->position(-5.0, 0.0, 0.0);
manual->textureCoord(1, 1);

manual->position(5.0, 10.0, 0.0);
manual->textureCoord(0, 0);
```

Listado 12.10: Declaración del material.

```
material TextureMaterial {
    technique {
        pass {
            vertex_program_ref VertexTexPassthrough
            {
            }

            fragment_program_ref FragmentTexPassthrough
            {
            }

            // Indicamos la textura
            texture_unit
            {
                texture sintel.png
            }
        }
    }
}
```

Para ello necesitamos acceder a las coordenadas de textura de nuestra entidad cuadrada, por lo que se las añadiremos en nuestro método *createManualPlane*.

Una vez podemos acceder a las coordenadas de textura del plano, podemos usarlas desde el *vertex shader*.

Listado 12.11: Tercer vertex shader. Hacemos uso de las coordenadas UV.

```
// Vertex shader
struct tVOutput {
    float4  position:   POSITION;
    float2  uv  :   TEXCOORD0;
};

// Usamos la posición y la coordenada de textura
tVOutput v_uv_passthrough(
    float4          position    :   POSITION,
    float2          uv      :   TEXCOORD0,
    uniform float4x4    worldViewMatrix)
{
    tVOutput    OUT;

    // Transformación del vértice
    OUT.position = mul(worldViewMatrix, position);
    OUT.uv = uv;

    return OUT;
}
```

Para este ejemplo, es necesario indicar la textura que se usará, y aunque en el capítulo 7 ya se habló del tema, será necesario apuntar un par de cosas sobre este nuevo material que aparece en el listado 12.10.

Listado 12.12: Segundo fragment shader. Mapeamos la textura en los fragmentos.

```
struct tFOutput {
    float4  color   :   COLOR;
};

// Devuelve el color correspondiente a la textura aplicada
tFOutput f_tex_passthrough(
    float2          uv  : TEXCOORD0,
    uniform sampler2D   texture)
{
    tFOutput OUT;
    // Asignamos el color de la textura correspondiente
    // en función de la coordenada UV interpolada
    OUT.color = tex2D(texture, uv);
    return OUT;
}
```

Ahora, el *fragment shader* recibe la coordenada interpolada de textura, e ignora el color del vértice. El resultado debería ser el que se ve en la Figura 12.11.

Figura 12.11: El cuadrado aparece con una textura.

En el anterior *shader* no se introdujo ningún concepto nuevo, más allá de cómo suministrar el color del vértice al *vertex shader*, sin embargo, ahora nos encontramos con una variable `uniform sampler2D`, un tipo nuevo que no habíamos visto, y el método `tex2D`.

Figura 12.12: La textura con una cierta tonalidad rojiza.

En Cg, un *sampler* se refiere a un objeto externo que se puede muestrear, como es una textura. El sufijo 2D indica que es una textura convencional en 2 dimensiones (existen texturas de 1D y 3D). Para ver los diferentes sampler que soporta Cg, puedes ir al capítulo 3 de *"The Cg Tutorial"* [FK03].

El método `tex2D` se encarga de devoler el color de la textura correspondiente a la coordenada de textura que se le pase como parámetro.

¿Serías capaz de crear un *fragment shader* que mostrara la textura modificada por los valores de color de los vértices como en la Figura 12.12?

12.6.5. Jugando con la textura

Esta subsección se limitará a mostrar algunos posibles efectos que se pueden conseguir mediante la modificación del *fragment shader*.

Primero, podríamos intentar devolver el color inverso de la textura que use nuestro modelo, dando un efecto como en el negativo de las fotos.

Esto es muy sencillo de conseguir, sólamente debemos restarle a 1 los valores correspondiente del vector RGBA. Veamos cómo:

Listado 12.13: Efecto negativo fotográfico.

```
1  // Estructura de salida con la información del fragmento procesado
2  struct tFOutput {
3      float4 color : COLOR;
4  };
5
6  // Devuelve el color inverso por cada fragmento
7  tFOutput f_tex_inverse(
8      float2          uv  : TEXCOORD0,
9      uniform sampler2D   texture)
10 {
11     tFOutput OUT;
12     OUT.color = 1 - tex2D(texture, uv);
13     return OUT;
14 }
```

Figura 12.13: La textura se muestra como el negativo de una foto.

Y ya que estamos modificando el color, quizás en lugar de usar el efecto negativo, tenemos intención de potenciar algún color. En el siguiente ejemplo, daremos un tono rojizo a la textura usada.

Listado 12.14: Modificando los colores de la textura con el color rojo.

```
1  // Devuelve el color correspondiente a la textura aplicada,
       modificandola para que predomine el rojo
2  tFOutput f_tex_red(
3      float2              uv  : TEXCOORD0,
4      uniform sampler2D   texture)
5  {
6      tFOutput OUT;
7      OUT.color = tex2D(texture, uv);
8      OUT.color.r *= 0.5f;
9      OUT.color.bg *= 0.15f;
10     return OUT;
11 }
```

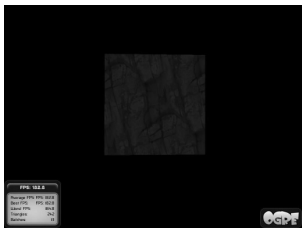


Figura 12.14: Los colores de la textura con su componente roja potenciada.

Como se puede ver, lo único que se ha hecho es multiplicar el componente de color por una cantidad (mayor en el caso del rojo, y menor en el caso del verde y el azúl). A pesar de la sencillez del ejemplo, si has intentado ejecutar este ejemplo con lo aprendido hasta ahora, es probable que no hayas podido.

El motivo es que se introducen dos conceptos nuevos. El primero de ellos se conoce como *swizzling* y consiste en una forma de reordenar los elementos de los vectores, o incluso de acortarlos. Por ejemplo:

```
float4 vec1 = float4(0, 2, 3, 5);

float2 vec2 = vec1.xz; // vec2 = (0, 3)

float scalar = vec1.w; // scalar = 5

float3 vec3 = scalar.xxx; // vec3 = (5, 5, 5)

float4 color = vec1.rgba; // color = (0, 2, 3, 5)
```

Por otro lado, entramos de lleno en el asunto de los perfiles. Los perfiles son una forma de gestionar la funcionalidad de la que disponen los *shader* que programamos, y de esta manera, saber en qué dispositivos podrán funcionar y qué funcionalidad tenemos disponible.

Todos los ejemplos usados hasta ahora usaban los perfiles *vs_1_1* y *ps_1_1* para DirectX 8, y *arbvp1* y *arbfp1* para OpenGL, que son los perfiles más simples a la vez que los más ampliamente soportados. Sin embargo para disponer del *swizzling* es necesario usar los más avanzados *vs_2_0* y *ps_2_0*, que son compatibles con DirectX y OpenGL.

La declaración del *fragment shader* en el material quedará, entonces, como se muetra en el Listado 12.15.

Listado 12.15: Declaración del *fragment shader*.

```
1 fragment_program FragmentRedTex cg
2 {
3     source texshaders.cg
4     entry_point f_tex_red
5     profiles ps_2_0 arbfp1
6 }
```

En el capítulo 2 de *"The Cg Tutorial"* [FK03], se puede encontrar más información al respecto.

A continuación se presentan algunos ejemplos más de *shaders* que servirán para familiarizarse con la forma en que se trabaja con ellos. Es muy recomendable intentar ponerlos en acción todos y modificar sus parámetros, jugar con ellos, a ver qué sucede.

Figura 12.15: Efecto interferencia.

Listado 12.16: Las funciones trigonómetricas ayudan a crear un patrón de interferencia.

```
// Devuelve el color de la textura modulado por una funciones
    trigonometricas
tFOutput f_tex_interference(
    float2              uv  : TEXCOORD0,
    uniform sampler2D   texture)
{
    tFOutput OUT;
    OUT.color = tex2D(texture, uv);
    OUT.color.r *= sin(uv.y*100);
    OUT.color.g *= cos(uv.y*200);
    OUT.color.b *= sin(uv.y*300);
    return OUT;
}
```

Figura 12.16: Efecto ondulado.

Listado 12.17: Las funciones trigonómetricas ayudan a crear la ondulación en la textura.

```
// Muestra la imagen modulada por una funcion trigonometrica
tFOutput f_tex_wavy(
    float2              uv  : TEXCOORD0,
    uniform sampler2D   texture)
{
    tFOutput OUT;
    uv.y = uv.y + (sin(uv.x*200)*0.01);
    OUT.color = tex2D(texture, uv);
    return OUT;
}
```

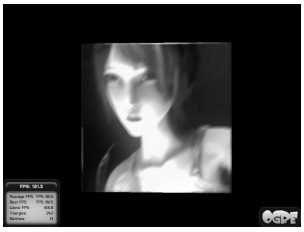


Figura 12.17: Efecto borroso (*blur*).

Listado 12.18: La composición del color de la textura en varias posiciones diferentes da lugar a un efecto borroso.

```
// Como si dibujaramos tres veces la textura
tFOutput f_tex_blurry(
    float2              uv  : TEXCOORD0,
    uniform sampler2D   texture)
{
    tFOutput OUT;
    OUT.color = tex2D(texture, uv);
    OUT.color.a = 1.0f;
    OUT.color += tex2D(texture, uv.xy + 0.01f);
    OUT.color += tex2D(texture, uv.xy - 0.01f);
    return OUT;
}
```

Listado 12.19: Este efecto se consigue con una combinación del efecto blur con la conversión del color a escala de grises.

```
// Dibuja la textura como si estuviera grabada en piedra
tFOutput f_tex_emboss(
    float2              uv  : TEXCOORD0,
    uniform sampler2D   texture)
{
    float sharpAmount = 30.0f;

    tFOutput OUT;
    // Color inicial
    OUT.color.rgb = 0.5f;
    OUT.color.a = 1.0f;

    // Añadimos el color de la textura
    OUT.color -= tex2D(texture, uv - 0.0001f) * sharpAmount;
    OUT.color += tex2D(texture, uv + 0.0001f) * sharpAmount;

    // Para finalizar hacemos la media de la cantidad de color de
        cada componente
    // para convertir el color a escala de grises
    OUT.color = (OUT.color.r + OUT.color.g + OUT.color.b) / 3.0f;

    return OUT;
}
```

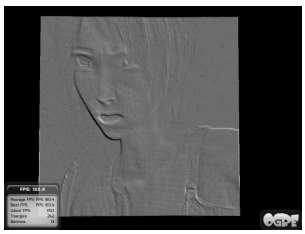


Figura 12.18: Efecto grabado en piedra.

12.6.6. Jugando con los vértices

Antes nos hemos dedicado a jugar un poco con los *fragment shader*, por lo que ahora sería conveniente hacer lo mismo con los *vertex shader*.

Primero usaremos el conocimiento hasta ahora recogido para, a partir de un sólo modelo, dibujarlo dos veces en posiciones distintas. Para conseguirlo se necesitará un material que defina dos pasadas. Una primera en la que el personaje se dibujará en su lugar, y otra en la que será desplazado.

Listado 12.20: Definición de las dos pasadas.

```
material CopyObjectMaterial
{
    technique
    {
        pass
        {
            vertex_program_ref VertexColorPassthrough
            {
            }

            fragment_program_ref FragmentTexPassthrough
            {
            }

            texture_unit
            {
                texture terr_rock6.jpg
            }
        }
```

```
20
21          pass
22          {
23              vertex_program_ref VertexDisplacement
24              {
25              }
26
27              fragment_program_ref FragmentTexPassthrough
28              {
29              }
30
31              texture_unit
32              {
33                  texture terr_rock6.jpg
34              }
35          }
36      }
37  }
```

Listado 12.21: Vértice desplazado cantidad constante en eje X.

```
1  // Desplazamiento del vértice 10 unidades en el eje X
2  tVOutput v_displacement(
3      float4              position    :   POSITION,
4      uniform float4x4    worldViewMatrix)
5  {
6      tVOutput    OUT;
7      // Modificamos el valor de la posición antes de transformarlo
8      OUT.position = position;
9      OUT.position.x += 10.0f;
10     OUT.position = mul(worldViewMatrix, OUT.position);
11     return OUT;
12 }
```

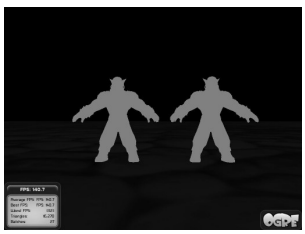


Figura 12.19: Resultado de las dos pasadas.

El anterior era un *shader* muy sencillo, por lo que ahora intentaremos crear nuestras propias animaciones mediante *shaders*.

Para ello necesitamos tener acceso a la delta de tiempo. En la página `http://www.ogre3d.org/docs/manual/manual_23.html#SEC128` aparecen listados los diferentes parámetros que Ogre expone para ser accedidos mediante variables `uniform`. En este caso, en la definición del *vertex shader* realizada en el material debemos indicar que queremos tener acceso a la delta de tiempo, como se ve en el Listado 12.22.

Listado 12.22: Acceso a la variable que expone la delta de tiempo.

```
1  vertex_program VertexPulse cg
2  {
3      source vertex_modification.cg
4      entry_point v_pulse
5      profiles vs_1_1 arbvp1
6
7      default_params
8      {
9          param_named_auto worldViewMatrix worldviewproj_matrix
10         param_named_auto pulseTime time
11     }
12 }
```

Una vez tenemos accesible el tiempo transcurrido entre dos vueltas del bucle principal, sólo necesitamos pasarlo como parámetro a nuestro *shader* y usarlo para nuestros propósitos. En este caso, modificaremos el valor del eje Y de todos los vértices del modelo, siguiendo una función cosenoidal, como se puede ver en el Listado 12.20.

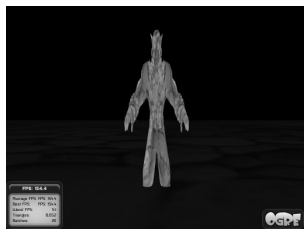


Figura 12.20: Personaje deformado en el eje Y.

Listado 12.23: Combinación de función trigonométrica y delta de tiempo para conseguir simular un movimiento continuo.

```
1  // A partir de la delta de tiempo simulamos una señal pulsante para
2  // escalar el modelo en Y
3  tVOutput v_pulse(
4      float4              position    :   POSITION,
5      float2              uv          :   TEXCOORD0,
6      uniform float       pulseTime,
7      uniform float4x4    worldViewMatrix)
8  {
9      tVOutput    OUT;
10
11     OUT.position = mul(worldViewMatrix, position);
12     OUT.position.y *= (2-cos(pulseTime));
13     OUT.uv = uv;
14
15     return OUT;
16 }
```

Sabiendo cómo hacer esto, se tiene la posibilidad de conseguir muchos otros tipos de efectos, por ejemplo en el siguiente se desplaza cada cara del modelo en la dirección de sus normales.

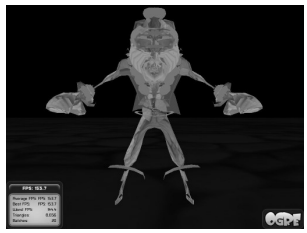


Figura 12.21: Figura con sus vértices desplazados en dirección de sus normales.

Listado 12.24: Uso de la normal y la delta de tiempo para crear un efecto cíclico sobre los vértices.

```
1  tVOutput v_extrussion(
2      float4              position    :   POSITION,
3      float4              normal      :   NORMAL,
4      float2              uv          :   TEXCOORD0,
5      uniform float       pulseTime,
6      uniform float4x4    worldViewMatrix)
7  {
8      tVOutput    OUT;
9      OUT.position = position + (normal * (cos(pulseTime)*0.5f));
10     OUT.position = mul(worldViewMatrix, OUT.position);
11     OUT.uv = uv;
12
13     return OUT;
14 }
```

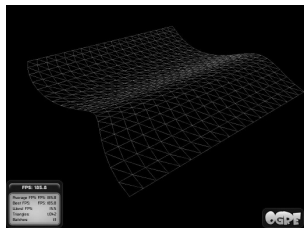


Figura 12.22: Plano con movimiento ondulatorio.

¿Serías capaz de montar una escena con un plano y crear un *shader* que simule un movimiento ondulatorio como el de la Figura 12.22?

12.6.7. Iluminación mediante shaders

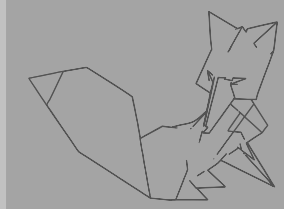
Como ejercicio final de esta sección se propone conseguir montar una escena con una luz y un plano. Y a partir de ella conseguir usar un modelo de iluminación por pixel sencillo (como se ve en la Figura 12.6).

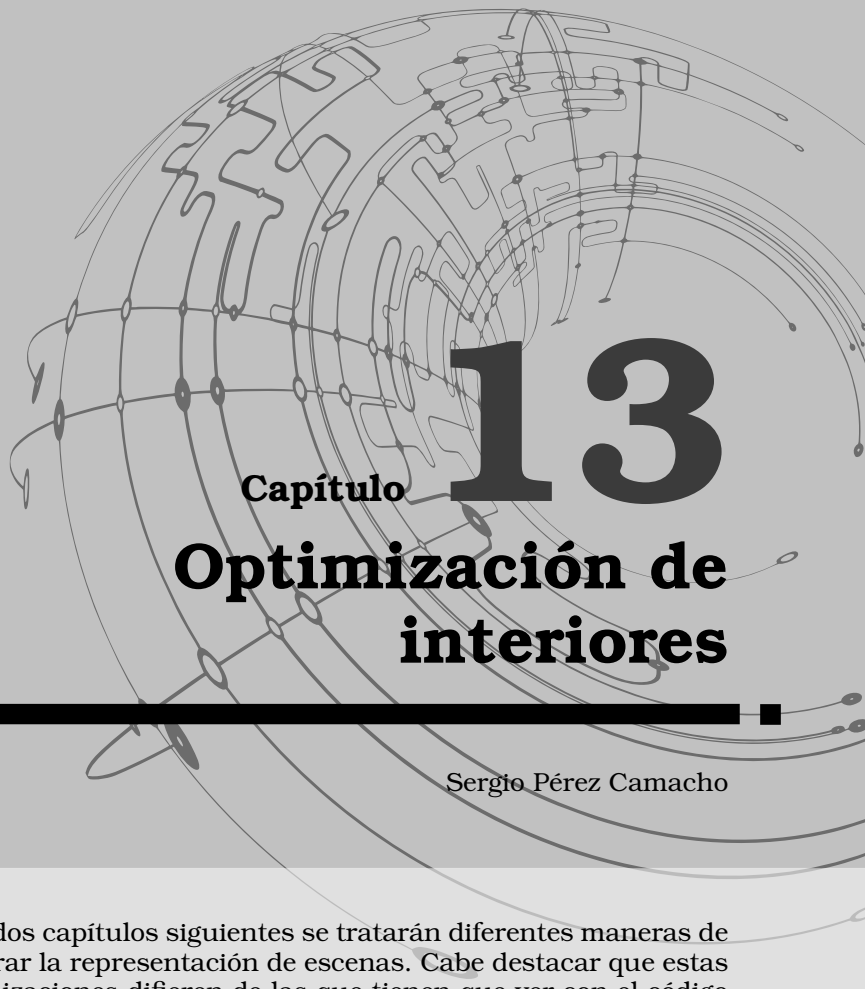
En el capítulo 9 se habla sobre el tema, a su vez, se puede encontrar una implementación, así como una explicación muy buena sobre iluminación, en el capítulo 5 de *"The Cg Tutorial"* [FK03].

Para conseguir usar todos los parámetros necesarios, se pueden cablear algunas variables en el material, como se puede ver en el Listado 12.25, gracias al uso de `param_named`.

Listado 12.25: Ejemplo de declaración de parámetros para ser usados como `uniform` por un *shader*.

```
1  default_params
2  {
3      param_named         globalAmbient float3 0.1 0.1 0.1
4
5      param_named         Ke  float3  0.0 0.0 0.0
6      param_named         Ka  float3  0.0215 0.1745 0.0215
7      param_named         Kd  float3  0.07568 0.61424 0.07568
8      param_named         Ks  float3  0.633 0.727811 0.633
9      param_named         shininess float 76.8
10
11     param_named_auto    eyePosition camera_position_object_space
12
13     param_named_auto    lightPosition
           light_position_object_space 0
14     param_named_auto    lightColorDiffuse   light_diffuse_colour 0
15 }
```





Capítulo 13

Optimización de interiores

Sergio Pérez Camacho

En los dos capítulos siguientes se tratarán diferentes maneras de acelerar la representación de escenas. Cabe destacar que estas optimizaciones difieren de las que tienen que ver con el código generado, o con los ciclos de reloj que se puedan usar para realizar una multiplicación de matrices o de vectores. Esta optimizaciones tienen que ver con una máxima: lo que no se ve no debería representarse.

13.1. Introducción

¿Más velocidad?

Las optimizaciones no sólo aumentan el número de *frames* por segundo, sino que puede hacer interactivo algo que no lo sería sin ellas.

Quizá cabe preguntarse si tiene sentido optimizar el dibujado de escenas ya que "todo eso lo hace la GPU". No es una mala pregunta, teniendo en cuenta que muchos monitores tienen una velocidad de refresco de $60Hz$, ¿para qué intentar pasar de una tasa de 70 *frames* por segundo a una de 100?. La respuesta es otra pregunta: ¿por qué no utilizar esos *frames* de más para añadir detalle a la escena? Quizá se podría incrementar el número de triángulos de los modelos, usar algunos *shaders* más complejos o añadir efectos de post-proceso, como *antialiasing* o profundidad de campo.

En este curso las optimizaciones se han dividido en dos tipos: optimización de interiores y optimización de exteriores. En este tema vamos a ver la optimización de interiores, que consiste en una serie de técnicas para discriminar qué partes se pintan de una escena y cuáles no.

Una escena de interior es una escena que se desarrolla dentro de un recinto cerrado, como por ejemplo en un edificio, o incluso entre las calles de edificios si no hay grandes espacios abiertos.

Ejemplos claros de escenas interiores se dan en la saga *Doom* y *Quake*, de *Id Software*, cuyo título *Wolfenstein 3D* fue precursor de este tipo de optimizaciones.

Sin el avance en la representación gráfica que supuso la división del espacio de manera eficiente, estos títulos jamás hubieran sido concebidos. Fue el uso de árboles BSP (2D y 3D, respectivamente) lo que permitió determinar qué parte de los mapas se renderizaba en cada momento.

En general, la optimización de interiores consiste en un algoritmo de renderizado que determina (de forma muy rápida) la oclusiones en el nivel de geometría. Esto diferencia claramente a este tipo de técnicas a las utilizadas para la representación de exteriores, que se basarán principalmente en el procesado del nivel de detalle (LOD).

13.2. Técnicas y Algoritmos

En su libro, Cormen et all [CLRS09] presentan a los algoritmos como una tecnología más a tener en cuenta a la hora de diseñar un sistema. Es decir, no sólo el hardware será determinante, sino que la elección de un algoritmo u otro resultará determinante en el buen rendimiento del mismo. Los videojuegos no son una excepción.

Elección **crucial**

Conocer diferentes algoritmos y saber elegir el más apropiado en cada momento es de suma importancia. Un buen algoritmo puede suponer la diferencia entre poder disfrutar de una aplicación interactiva o no poder hacerlo.

A continuación se presentarán algunas de las técnicas y algoritmos que recopila Dalmau [Dal04] en su libro, que se ha seguido para preparar esta y la siguiente lección, para la representación de escenas de interiores.

13.2.1. Algoritmos basados en Oclusores

Si tuviéramos que determinar qué triángulos dentro de *frustum* ocluyen a otros, la complejidad de llevar a cabo todas las comprobaciones sería de $O(n^2)$, siendo n el número de triángulos.

Dado que esta complejidad es demasiado alta para una aplicación interactiva, se hace necesario idear alguna forma para reducir el número de cálculos. Una forma sencilla es reducir la lista de posibles oclusores. Los triángulos más cercanos a la cámara tienen más posibilidades de ser oclusores que aquellos que están más alejados, ya que estadísticamente estos van a ocupar más área de la pantalla. De este modo, usando estos triángulos como oclusores y los lejanos como ocluidos, se podrá reducir la complejidad hasta casi $O(n)$. Si además se tiene en cuenta que el *frustum* de la cámara tiene un límite (plano *far*), se podrán descartar aun más triángulos, tomando como ocluidos los que estén más allá que este plano.

Listado 13.1: Algoritmo básico basado en oclusores

```
vector<Triangle> occluders = createOccludersSet(sceneTrianles);
vector<Triangle>    others = removeOccluders(sceneTriangles,
                                             occluders);
vector<Triangle>::iterator it;

for (it = others.begin(); it != others.end(); ++it)
{
  if (closerThanFarPlane(*it) &&
      !testOcclusion(*it, occluders))
  {
    (*it)->draw(); // (*it)->addToRendeQueue();
  }
}
```

Este algoritmo podría beneficiarse del *clipping* y del *culling* previo de los triángulos. No es necesario incluir en el algoritmo de oclusores ningún triángulo que quede fuera del *frustum* de la vista actual, tampoco los triángulos que formen parte de las caras traseras se pintarán, ni tendrán que formar parte de los oclusores. La pega es que hay que realizar *clipping* y *culling* por software. Tras esto, se tendrían que ordenar los triángulos en Z y tomar un conjunto de los n primeros, que harán de oclusores en el algoritmo. Computacionalmente, llevar a cabo todas estas operaciones es muy caro. Una solución sería hacerlas sólo cada algunos *frames*, por ejemplo cuando la cámara se moviese lo suficiente. Mientras esto no pasase, el conjunto de oclusores no cambiaría, o lo haría de manera mínima.

Listado 13.2: Actualización de los oclusores

```
const size_t maxOccluders = 300;

newPos = calculateNewCameraPosition();
if (absoluteDifference(oldPos, newPos) > delta)
{
  performClipping(sceneTriangles);
  performCulling(sceneTriangles);
  oldPos = newPos;
  vector<Triangles> occluders =
    getZOrderedTriangles(sceneTriangles,
                         maxOccluders);
}
```

13.2.2. Algoritmo BSP

Una de las formas para facilitar la representación de escenas interiores en tiempo real es el uso de estructuras de datos BSP (*Binary Space Partition*). Estas estructuras se han utilizado desde juegos como *Doom*, que usaba un arbol BSP de 2 dimensiones, y *Quake*, que fue el primero que usó uno de 3 dimensiones.

Un BSP es un árbol que se utiliza para clasificar datos espaciales, más concretamente triángulos en la mayoría de los casos. La ventaja principal de esta estructura es que se le puede preguntar por una serie de triángulos ordenados por el valor Z, desde cualquier punto de vista

de la escena. Estos árboles se usan también para detectar colisiones en tiempo real.

Construcción de estructuras BSP

Un BSP se construye a partir de un conjunto de triángulos, normalmente de la parte estática de una escena, esto es, el mapa del nivel. El algoritmo de construcción es recursivo:

1. Tomar el conjunto completo de triángulos como entrada.
2. Buscar un triángulo que divida a ese conjunto en dos partes más o menos equilibradas.
3. Calcular el plano que corresponde a ese triángulo.
4. Crear un nodo de árbol y almacenar el triángulo y su plano asociado en el mismo.
5. Crear nuevos triángulos a partir de los que queden cortados por este plano.
6. Dividir el conjunto total en dos nuevos conjuntos según queden delante o detrás de la división.
7. Para cada uno de los nuevos conjuntos que aun tengan triángulos (más que un umbral máximo de división dado), volver al paso 2.

Para explicar este algoritmo, se usará una representación de dos dimensiones para simplificar la complejidad. En la figura 13.2 se ve la planta de un nivel. Cada segmento representa a un plano.

Primero se tomará el plano 7 como divisor del nivel, y este dividirá al plano 5 y al 1 en dos (figura 13.3).

El nivel queda dividido en dos nuevos conjuntos, detrás de la división quedan cinco planos (1.1, 6, 5.1, 8 y 9), y delante también (1.2, 2, 3, 4 y 5.2).

En el árbol se crearía un nodo que contendría el plano 7, que es con el que se ha realizado la división. Como los dos subconjuntos nuevos no están vacíos, se crearían de nuevo otros dos nodos (ver figura 13.5).

El algoritmo de división se irá aplicando recursivamente hasta que se hayan realizado todas la divisiones posibles (o se hayan llegado al umbral deseado), tal y como se puede ver en la figura 13.4.

El árbol BSP que se obtiene después de rellenar todos los niveles se muestra en la figura 13.7. Notese cómo el árbol queda bastante equilibrado gracias a una buena elección del plano de división. Un buen equilibrio es fundamental para que pueda desempeñar su labor de manera eficiente, puesto que si el árbol estuviera desequilibrado el efecto sería parecido al no haber realizado una partición espacial.

Recursividad

Como casi todos los algoritmos de construcción de árboles, el BSP es un algoritmo recursivo. Se dice que una función es recursiva cuando se llama a sí misma hasta que se cumple una condición de parada. Una función que se llama a sí misma dos veces se podrá representar con un árbol binario; una que se llame n-veces, con un árbol n-ario.

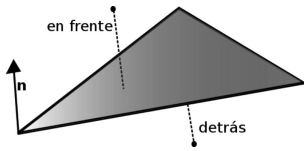


Figura 13.1: Posición con respecto a un plano de normal **n**.

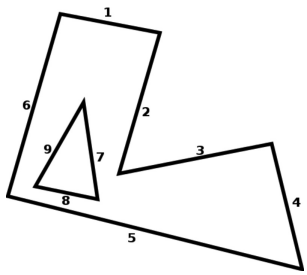


Figura 13.2: Mapa de una escena visto desde arriba.

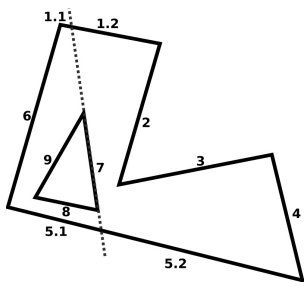


Figura 13.3: Primera división (BSP).

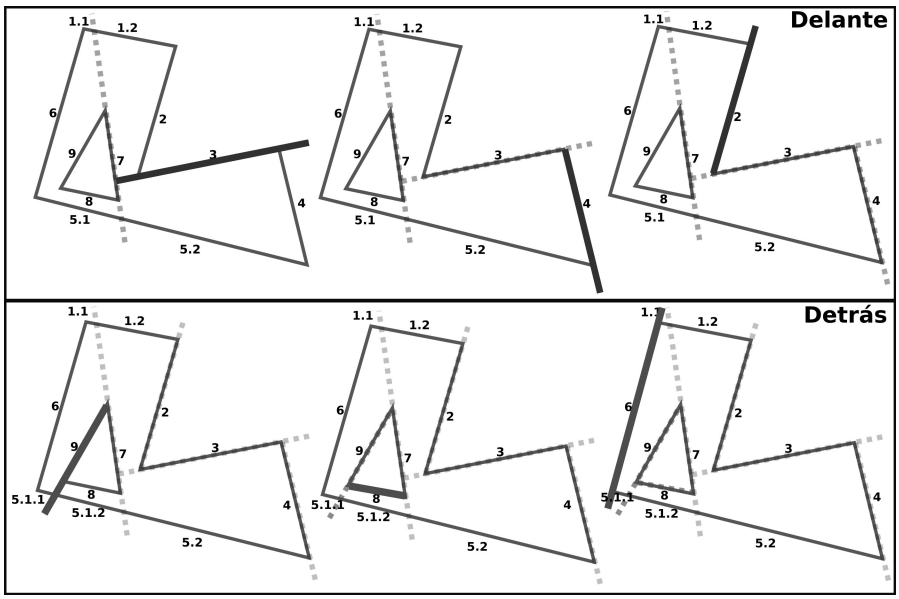


Figura 13.4: Sucesivas divisiones del nivel (BSP)

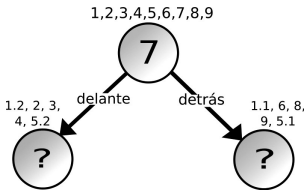


Figura 13.5: Árbol resultante de la primera división (BSP).

La elección de un buen triángulo para utilizar como plano de división no es trivial. Es necesario establecer algún criterio para encontrarlo. Un criterio podría ser tomar el triángulo más cercano al centro del conjunto que se tiene que dividir, que podría cortar a muchos otros triángulos, haciendo que creciera rápidamente el número de ellos en los siguientes subconjuntos. Otro criterio podría ser buscar el triángulo que corte a menos triángulos. Uno mejor sería mezclar esta dos ideas.

Ericson [Eri05] analiza algunos problemas relacionados con la elección del plano divisor y propone algunas soluciones simples, parecidas al último criterio propuesto anteriormente.

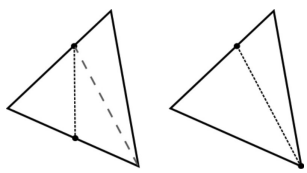


Figura 13.6: División de un triángulo por un plano.

Otro problema al que hay que enfrentarse es al de la división de los triángulos que resultan cortados por el plano divisor. Cuando se corta un triángulo, este puede dividirse en dos o tres triángulos, siendo mucho más probable que se de la segunda opción. Esto es debido a que la división de un triángulo normalmente genera otro triángulo y un cuadrilátero, que tendrá que ser dividido a su vez en otros dos triángulos (figura 13.6).

La solución pasa por tomar los vértices del triángulo atravesado como segmentos y hallar el punto de intersección de los mismos con el plano. Con esos puntos será trivial reconstruir los triángulos resultantes.

La estructura de árbol BSP podría estar representada en C++ como en el listado siguiente:

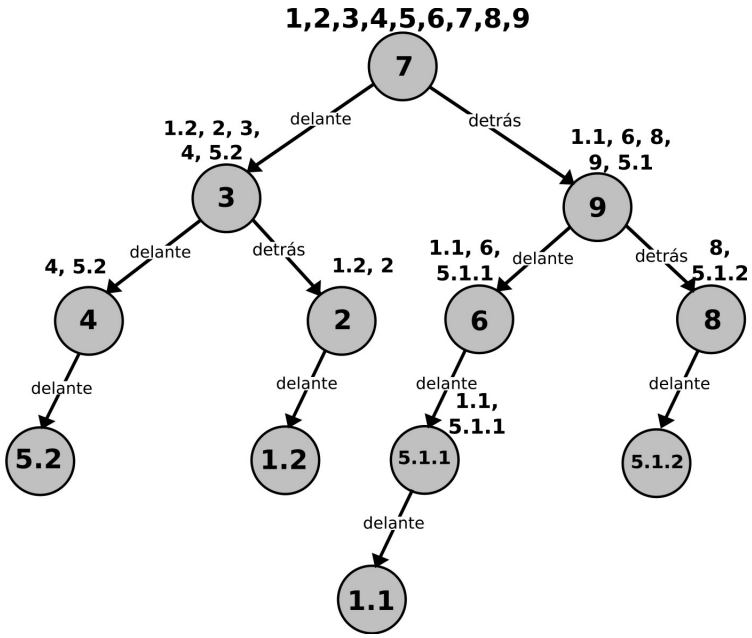


Figura 13.7: Sucesivas divisiones del nivel (BSP)

Listado 13.3: Class BSP

```
1  class BSP {
2  public:
3    BSP(vector<Triangle> vIn);
4
5  private:
6    BSP* front;
7    BSP* back;
8    Plane p;
9    Triangle t; // vector<Triangle>vt;
10 };
```

Su construcción, vendría dada por una función como esta:

Listado 13.4: createBSP

```
1  BSP* createBSP(const vector<Triangle>& vIn) {
2
3    BSP* bsp = new BSP;
4
5    bsp->t = getBestTriangle(vt);
6    vtNew = removeTriangleFromVector(vt, t);
7
8    bsp->p = planeFromTriangle(t);
9
10   vector<Triangle>::iterator it;
11
12   for (it = vt.begin(); vt != vt.end(); ++it) {
13     if (cuts(bsp->p, *it))
14       split(*it, bsp->p);
```

```
15    }
16    vector<Triangle> frontSet = getFrontSet(vtNew, t);
17    vector<Triangle> backSet  = getBackSet(vtNew, t);
18
19    bsp->front = createBSP(frontSet);
20    bsp->back = createBSP(backSet);
21
22    return bsp;
23  }
```

Orden dependiente de la vista

La principal ventaja de un BSP es que gracias a él es posible obtener una lista de triángulos ordenados, sea cual sea la vista en la que nos encontremos.

Obsérvese el siguiente listado de código:

Listado 13.5: paintBSP

```
1  void paintBSP(BSP* bsp, const Viewpoint& vp) {
2    currentPos = backOrFront(bsp, vp);
3    if (currentPos == front) {
4      paintBSP(back, vp);
5      bsp->t.addToRenderQueue();
6      paintBSP(front, vp);
7    } else {
8      paintBSP(front, vp);
9      bsp->t.addToRenderQueue();
10     paintBSP(back, vp);
11   }
12 }
```

La función anterior pinta los triángulos (incluidos los que quedan detrás de la vista) en orden, desde el más lejano al más cercano.

Esto era muy útil cuando el *hardware* no implementaba un *Z-buffer*, ya que está función obtenía los triángulos ordenados con un coste linear.

Si cambiamos el algoritmo anterior (le damos la vuelta) recorreremos las caras desde las más cercanas a las más lejanas. Esto sí puede suponer un cambio con el hardware actual, ya que si pintamos el triángulo cuyo valor va a ser mayor en el *Z-buffer*, el resto de los triángulos ya no se tendrán que pintar (serán descartados por el *hardware*).

Clipping Jerárquico

Un BSP se puede extender para usarlo como un sistema de aceleración de *clipping*, quitando los triángulos que queden fuera del *frustum* de la cámara. Lo único que hay que añadir en el árbol durante su construcción en una *bounding box* por cada nodo. Cuanto más se profundice en el árbol, más pequeñas serán, y si el algoritmo de equilibrado de la división es bueno, una *bounding box* contendrá otras dos

de un volumen más o menos parecido, equivalente a la mitad de la contenedora.

El algoritmo para recorrer el árbol es muy parecido al anterior, y bastaría con introducir una pequeña modificación.

Listado 13.6: BSPClipping

```
1 void paintBSP(BSP* bsp, const Viewpoint& vp, const Camera& cam) {
2   if ( isNodeInsideFrustum(bsp, cam.getCullingFrustum()) )
3   {
4     // Igual que en el ejemplo anterior
5   }
6 }
```

Detección de la oclusión

También es posible utilizar un árbol BSP para detectar oclusiones. Este uso se popularizó gracias al motor de *Quake*, que utilizaba un nuevo tipo de árbol llamado *leafy*-BSP, donde se utilizaron por primera vez para el desarrollo de un videojuego. Su propiedad principal es la de dividir de manera automática el conjuntos de triángulos entrante en un *array* de celdas convexas.

Este nuevo tipo de árboles son BSPs normales donde toda la geometría se ha propagado a las hojas, en vez de repartirla por todos los nodos a modo de triángulos divisores. De este modo, en un BSP normal, las hojas sólo almacenan el último triángulo divisor.

Para transformar un BSP en un *leafy*-BSP lo que hay que hacer es "agitar" el árbol y dejar caer los triángulos de los nodos intermedios en las hojas (ver figura 13.8)

Una vez que el árbol se haya generado, se podrá almacenar la lista de triángulos de cada nodo como una lista de celdas numeradas. Para el ejemplo anterior las celdas se muestran en la figura 13.9.

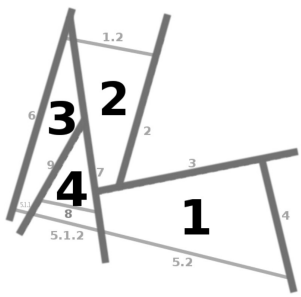


Figura 13.9: Nivel dividido en celdas (l-BSP).

Cada celda representa una zona contigua y convexa del conjunto de triángulos inicial. Las paredes de las celdas pueden ser o bien áreas ocupadas por geometría del nivel o espacios entre las celdas. Cada espacio abierto debe pertenecer exactamente a dos celdas.

Nótese que el algoritmo anterior convierte cualquier conjunto de triángulos en una lista de celdas y de pasillos que las conectan, que es la parte más complicada.

Es posible precalcular la visibilidad entre las celdas. Para ello se utilizan los pasillos (o portales, aunque diferentes a los que se verán un poco más adelante). Se mandan rayos desde algunos puntos en un portal hacia los demás, comprobándose si llegan a su destino. Si un rayo consigue viajar del portal 1 al portal 2, significa que las habitaciones conectadas a ese portal son visibles mutuamente. Este algoritmo fue presentado por Teller [TS91].

Esta información sobre la visibilidad se almacenará en una estructura de datos conocida como PVS (*Conjunto de Visibilidad Potencial*),

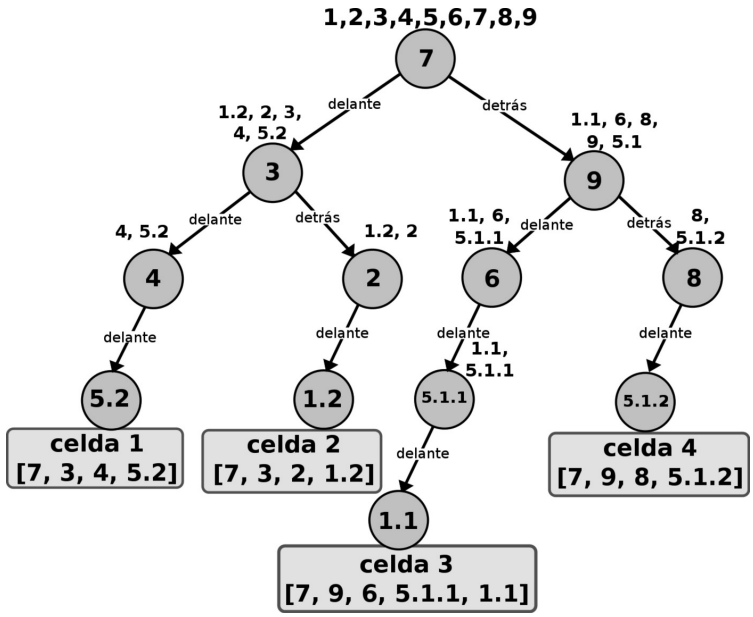


Figura 13.8: Transformación de BSP a *leafy*-BSP.

que es sólo una matriz de bits de NxN que relaciona la visibilidad de la fila i (celda i) con la de la columna j (celda j).

Rendering

Para representar los niveles de *Quake III: Arena* se utilizaba un algoritmo más o menos como el que se explica a continuación.

!**Precalcular** es la clave!

Los pasos más costosos han sido precalculados, haciendo factible la representación en tiempo real.

- Se comienza determinando en qué celda se encuentra la cámara (el jugador) utilizando el BSP. Se recorre el árbol desde la raíz, comparando la posición con la del plano divisor para bajar hasta una hoja y elegir una celda determinada.
- Se utiliza el PVS para determinar qué celdas son visibles desde la celda actual, utilizando la matriz de bits (o de booleanos).
- Se *renderizan* las celdas visibles. Se pintan desde el frente hasta el fondo, lo que ayudará al *Z-buffer* a descartar triángulos lo antes posible. Se ordenan las celdas por distancia, se usa su *bounding box* para determinar si quedan dentro del *frustum* para hacer el *clipping* de pantalla y se mandan a *renderizar*.

Como se ha podido ver, gracias al uso de un árbol *leafy*-BSP se han resuelto casi todos los problemas de determinación de la visibilidad utilizando una estructura precalculada. Esto hace que en el bucle principal del juego no se dedique ningún esfuerzo a computarla. Además, este tipo de estructuras son útiles para determinar colisiones en tiempo real y para ayudar a recorrer los niveles a la IA.

13.2.3. Portal *Rendering*

Otra de las técnicas utilizadas para optimizar la representación de interiores son los portales (Portals). Es un enfoque diferente a los árboles BSP, pero que ofrece una aceleración similar. El motor gráfico *Unreal* demostró su validez utilizando una versión del mismo, y ha ganado adeptos entre los desarrolladores desde entonces. Permite, al igual que los BSPs, representar sólo lo que se ve. En el caso de esta técnica, no se precalcula la visibilidad sino que es computada en tiempo real.

Esta técnica se basa en que los niveles de interiores de un juego están construidos a base de habitaciones interconectadas entres sí por puertas, por ventanas, o en general, por portales. Es de aquí de donde viene su nombre. Para representar estas conexiones entre las diferentes habitaciones será necesario una estructura de datos de grafo no dirigido. Cada nodo del grafo es una habitación y cada vértice del grafo es un portal. Ya que es posible que en una habitación se encuentren varios portales, es necesario que la estructura de datos permita conectar un nodo con varios otros, o que dos nodos estén conectados por dos vértices.

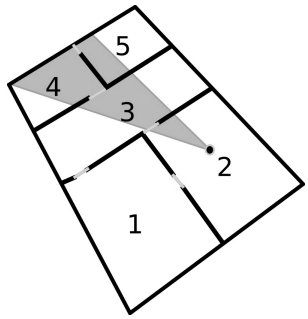


Figura 13.10: Mapa de habitaciones conectadas por portales

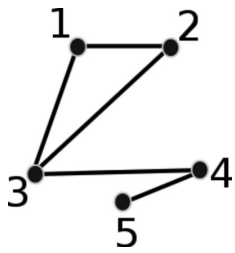


Figura 13.11: Grafo que representa las conexiones del mapa de habitaciones.

Al contrario que los BSP, la geometría de un nivel no determina de manera automática la estructura de portales. Así, será necesario que la herramienta que se use como editor de niveles soporte la división de niveles en habitaciones y la colocación de portales en los mismos. De esto modo, la creación de la estructura de datos es un proceso manual. Estas estructuras no almacenan datos precalculados sobre la visibilidad; esta se determinará en tiempo de ejecución.

El algoritmo de *renderizado* comienza por ver dónde se encuentra la cámara en un momento dado, utilizando los *bounding volumes* de cada habitación para determinar dentro de cuál está posicionada. Primero se pintará esta habitación y luego las que estén conectadas por los portales, de forma recursiva, sin pasar dos veces por un mismo nodo (con una excepción que se verá en el siguiente apartado). Lo complejo del algoritmo es utilizar los portales para hacer *culling* de la geometría.

Es necesario detectar qué triángulos se pueden ver a través de la forma del portal, ya que normalmente habrá un gran porcentaje no visible, tapados por las paredes que rodean al mismo. Desde un portal

se puede ver otro, y esto tendrá que tenerse en cuenta al calcular las oclusiones. Se utiliza una variante de la técnica de *view frustum* (que consiste en descartar los triángulos que queden fuera de un frustum, normalmente el de la cámara), que Dalmau llama *portal frustum*. El *frustum* que se utilizará para realizar el *culling* a nivel de portal tendrá un origen similar al de la cámara, y pasará por los vértices del mismo. Para calcular las oclusiones de un segundo nivel en el grafo, se podrá obtener la intersección de dos o más *frustums*.

Portal

¿Tendrá algo que ver esta técnica con algún juego de Valve™? Gracias a ella el concepto de una de sus franquicias ha sido posible.

Un portal puede tener un número de vértices arbitrario, y puede ser cóncavo o convexo. La intersección de dos portales no es más que una intersección 2D, en la que se comprueba vértice a vértice cuáles quedan dentro de la forma de la recursión anterior. Este algoritmo puede ralentizar mucho la representación, puesto que el número de operaciones depende del número de vértices, y la forma arbitraria de los portales no ayuda a aplicar ningún tipo de optimizaciones.

Luebke y Jobes [LG95] proponen que cada portal tenga asociada un *bounding volume*, que simplificará enormemente los cálculos. Este *bounding volume* rodea a todos los vértices por portal, lo que hará que el algoritmo de como visibles algunos triángulos que no lo son. La pérdida de rendimiento es mínima, y más en el *hardware* actual donde probablemente cada habitación esté representada como un *array* de triángulos.

Efectos ópticos utilizando portales

Espejos

Usando portales, poner espejos en la escena tiene un coste gratuito, excepto porque supone representar dos veces la misma habitación.

Una de las ventajas principales de utilizar la técnica de portales en comparación con la de BSP es que se pueden implementar efectos de reflexión y de transparencia, usando el algoritmo central de la misma. Para llevar a cabo este tipo de efectos, lo único que hay que añadir a cada portal es su tipo, por ejemplo los portales podrían ser del tipo normal, espejo, transparencia, o de cualquier otro efecto que se pueda llevar a cabo a través de este tipo de estructuras.

Listado 13.7: Ejemplo de representación de un portal

```
enum portalType {
  NORMAL,
  MIRROR,
  TRANPARENT,
  INTERDIMENSIONAL,
  BLACK_VOID
};

struct portal {
  vector<Vertex3D*> vertexes_;
  portalType type_;
  Room* room1;
  Room* room2;
};
```

A la hora de representar un portal, podría discriminarse por el tipo, utilizando la técnica adecuada según corresponda.

Listado 13.8: Ejemplo de elección de tipos de portales

```
switch(type_) {
case NORMAL:
  // Algoritmo Normal
  break;

case MIRROR:
  // Calcular la cámara virtual usando el plano de soporte del
      portal
  //
  // Invertir la view-matrix
  //
  // Pintar la habitación destino
  //
  // Pintar la geometría del portal de forma
  // translúcida con algo de opacidad si se desea
  break;

case TRANSPARENT:
  // Pintar de forma normal la habitación que corresponda
  //
  // Pintar la geometría del portal de forma
  // translúcida con algo de opacidad si se desea
  break;

case INTERDIMENSIONAL:
  // Modificar los vértices del array con una función sinuidal
  // Pintarlo
  // Añadir colores chillones a la opacidad del portal.
  break;

case BLACK_VOID:
  // Modificar la geometría para que la habitación
  // destino parezca estirada hacia un agujero negro
  //
  // Pintar un borde brillante en los vértices de la forma del
      portal.
  break;
}
```

¿Se le ocurre algún tipo de efecto más o alguna otra forma de aprovechar las características de los portales?

13.2.4. Mapas de Oclusión Jerárquicos (HOM)

Esta técnica, al igual que la de portales, computa la oclusiones en tiempo real durante la ejecución del bucle principal. La ventaja principal de esta técnica es que no es necesario preprocesar la geometría del nivel de ningún modo. Además, otra ventaja de HOM frente a BSP o portales es que permite utilizar geometría tanto estática como dinámica de manera indiferente.

HOM [ZMHHI97] está basado en una jerarquía de mapas de oclusión. Cada uno de ellos será de la mitad de tamaño que el anterior. Se

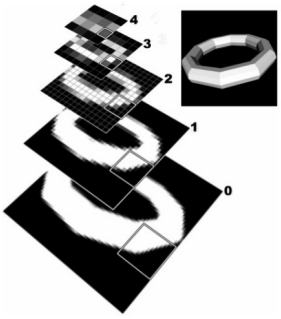


Figura 13.12: HOM. Jerarquía de imágenes (ZHANG).

comienza con una representación a pantalla completa de nuestra escena en blanco y negro. Tras esto se calculan una serie de *mipmaps*, donde cada $2x2$ píxeles se transforman en uno de la nueva imagen. Este algoritmo es muy sencillo si se comienza con una imagen cuadrada potencia de dos. Estas imágenes son las que forman la jerarquía de mapas de oclusión. El mapa de oclusión no contiene la escena completa, sólo un conjunto de posibles oclusores, elegidos con algún método parecido al explicado en el primer algoritmo.

En el bucle principal se pintará objeto por objeto, utilizando algún tipo de estructura no lineal, como un *octree* o un *quadtree*. Para cada objeto se calcula un *bounding rectangle* alineado con la pantalla. Después se toma la imagen (el nivel HOM) con un tamaño de píxel aproximadamente similar al mismo. Este rectángulo puede caer en una zona completamente blanca, y habrá que hacer más comprobaciones (existe un *full-overlap*, el objeto que comprobamos está completamente delante o detrás), puede caer en una zona negra, y se tendrá que pintar, o puede caer en una zona gris, caso en el que habrá que consultar con una imagen de mayor resolución.

Cuando el rectángulo cae en una zona con blanca, es necesario hacer una comprobación sobre los valores Z para comprobar si el objeto está delante o detrás. Esto se consigue con un DEB (*Buffer* de Estimación de Profundidad), que no es más que un Z*-buffer* construido por software, utilizando los posibles oclusores. El DEB almacena la información resultante de crear las *bounding boxes* de los oclusores y almacenar a modo de píxel el valor más lejano (al contrario que un Z*-buffer* normal) para cada posición de la pantalla.

El algoritmo completo podría describirse como sigue:

- Seleccionar un buen conjunto de oclusores. Se descartarán objetos pequeños, o con muchos polígonos, y los objetos redundantes. Es posible colocar objetos falsos que no se pintarán como oclusores a mano, para mejorar el conjunto de manera premeditada.
- En ejecución, se seleccionan los primeros N oclusores más cercanos.
- Calcular el HOM en función de estos objetos. Con la función de *render-to-texture* se crea la primera imagen. Las demás, por software o utilizando otras texturas y alguna función de *mipmapping* de la GPU.
- Mientras se recorre el *scene-graph* se comparan estos objetos con el mapa de oclusión. Si caen en una zona blanca, se comprueban contra el DEB; si caen en una zona negra, se pinta; y si caen en una zona gris, será necesario usar una imagen de mayor resolución.

Dalmau afirma que con esta técnica se evita pintar de media entre un 40 % y un 60 % de toda la geometría entrante.

13.2.5. Enfoques híbridos

En los videjuegos se suele utilizar la combinación de técnicas que más beneficio brinde al tipo de representación en tiempo real a la que se esté haciendo frente. De nuevo, Dalmau propone dos aproximaciones híbridas.

Equilibrio

Utilizar lo mejor de cada una de las técnicas hace que se puedan suplir sus debilidades.

Portal-Octree

En un juego donde el escenario principal está repleto de habitaciones y cada una de ellas está llena de objetos, una aproximación de través de un BSP quizá no sería la mejor idea. No sólo porque este tipo de estructuras está pensado principalmente para objetos estáticos, sino porque un árbol BSP suele extenderse muy rápido al empezar a dividir el espacio.

Si además el juego requiere que se pueda interactuar con los objetos que hay en cada habitación, el BSP queda descartado para almacenar los mismos. Quizá utilizar la técnica de portales pueda usarse para las habitaciones, descartando así algo de geometría del nivel. Aun así la gran cantidad de objetos haría que fueran inmanejables.

Una posible solución: utilizar portales para representar las habitaciones del nivel, y en cada habitación utilizar un *octree*.

Quadtree-BSP

Hay juegos que poseen escenarios gigantestos, con un área de exploración muy grande. Si se enfoca la partición de este tipo de escenas como la de un árbol BSP, el gran número de planos de división hará que crezca la geometría de manera exponencial, debido a los nuevos triángulos generados a partir de la partición.

Una forma de afrontar este problema es utilizar dos estructuras de datos. Una de ellas se usará para realizar una primera división espacial de la superficie (2D, un Quadtree, por ejemplo) y la otra para una división más exhaustiva de cada una de esas particiones. De esto modo, se podrá utilizar un Quadtree donde cada nodo contiene un BSP.

De este modo, se pueden utilizar las características especiales de cada uno de ellos para acelerar la representación. En un primer paso, el Quadtree facilitará la determinación de la posición global de una manera muy rápida. Una vez que se sepa en qué parte del escenario se encuentra la acción, se tomará el BSP asociado a la misma y se procederá a su representación como se mostró en el apartado anterior.

Este tipo de representaciones espaciales más complejas no son triviales, pero a veces son necesarias para llevar a cabo la implementación exitosa de un videojuego.

En el siguiente capítulo ese introducirán los *quadtrees*.

¿Recuerda alguno de estos tipo de escenas en los últimos videojuegos a los que ha jugado?

13.2.6. Tests asistidos por *hardware*

Las tarjetas gráficas actuales proveen de mecanismos para llevar a cabo los cálculos de detección de la oclusión por hardware. Estos mecanismos consisten en llamadas a funciones internas que reducirán la complejidad del código. El uso de estas llamadas no evitará la necesidad de tener que programar pruebas de oclusión, pero puede ser una ayuda bastante importante.

Hardware

El uso del hardware para realizar los tests de oclusión es el futuro, pero eso no quita que se deban conocer las técnicas en las que se basa para poder utilizarlo de manera efectiva.

La utilización del hardware para determinar la visibilidad se apoya en pruebas sobre objetos completos, pudiendo rechazar la inclusión de los triángulos que los forman antes de entrar en una etapa que realice cálculos sobre los mismos. Así, las GPUs actuales proveen al programador de llamadas para comprobar la geometría de objetos completos contra el *Z-buffer*. Nótese como estas llamadas evitarán mandar estos objetos a la GPU para se pintados, ahorrando las transformaciones que se producen antes de ser descartados. Además, como retorno a dichas llamadas se puede obtener el número de píxeles que modificría dicho objeto en el *Z-buffer*, lo que permitiría tomar decisiones basadas en la relevancia del objeto en la escena.

Cabe destacar que si se usa la geometría completa del objeto, mandando todos los triángulos del mismo al test de oclusión de la GPU, el rendimiento global podría incluso empeorar. Es algo normal, puesto que en una escena pueden existir objetos con un gran número de triángulos. Para evitar este deterioro del rendimiento, y utilizar esta capacidad del hardware en beneficio propio, lo más adecuado es utilizar *bounding-boxes* que contengan a los objetos. Una caja tiene tan solo 12 triángulos, permitiendo realizar tests de oclusión rápidos y bastante aproximados. Es fácil imaginarse la diferencia entre mandar 12 triángulos o mandar 20000.

Además, si las pruebas de oclusión para todos los objetos se llevan a cabo de forma ordenada, desde los más cercanos a los más lejanos, las probabilidades de descartar algunos de ellos aumentan.

Como ejemplo, uno tomado de las especificaciones de *occlusion query* de las extensiones de OpenGL [NVi01].

Listado 13.9: Oclusión por hardware

```
GLuint queries[N];
GLuint sampleCount;
GLint  available;
GLuint bitsSupported;

// Comprobar que se soporta la funcionalidad
glGetQueryiv(GL_QUERY_COUNTER_BITS_ARB, &bitsSupported);
if (bitsSupported == 0) {
  // Representar sin test de oclusion...
}

glGenQueriesARB(N, queries);
...
// Antes de este punto, renderizar los oclusores mayores
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glDepthMask(GL_FALSE);
// tambien deshabilitar el texturizado y los shaders inutiles
for (i = 0; i < N; i++) {
  glBeginQueryARB(GL_SAMPLES_PASSED_ARB, queries[i]);
  // renderizar la bounding box para el objecto i
  glEndQueryARB(GL_SAMPLES_PASSED_ARB);
}

glFlush();

// Hacer otro trabajo hasa que la mayoria de las consultas esten
    listas
// para evitar malgastar tiempo
i = N*3/4; // en vez de N-1, para evitar que la GPU se ponga en
    ïdle"
do {
  DoSomeStuff();
  glGetQueryObjectivARB(queries[i],
                        GL_QUERY_RESULT_AVAILABLE_ARB,
                        &available);
} while (!available);

glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_TRUE);
// habilitar otro estado, como el de texturizado
for (i = 0; i < N; i++) {
  glGetQueryObjectuivARB(queries[i], GL_QUERY_RESULT_ARB,
                         &sampleCount);
  if (sampleCount > 0) {
    // representar el objecto i
  }
}
```

13.3. Manejo de escenas en OGRE

La representación de escenas en OGRE tiene su base en un grafo de escena (*Scene Graph*) un tanto particular. En otros motores gráficos, el grafo de escena está completamente ligado al tipo nodo del mismo. Esto hace que exista un completo acoplamiento entre grafo y nodos, haciendo muy difícil cambiar el algoritmo de ordenación una vez realizada la implementación inicial.

Para solucionar este problema, OGRE interactúa con el grafo de escena (ver capítulo 3) sólo mediante su firma (sus métodos públicos), sin importar cómo se comporte internamente. Además, OGRE tan sólo utiliza este grafo como una estructura, ya que los nodos no contienen ni heredan ningún tipo de funcionalidad de control. En vez de eso, OGRE utiliza una clase *renderable* de donde se derivará cualquier tipo de geometría que pueda contener una escena. En la figura 1.15 del capítulo 1, se ve la relación entre un tipo *renderable* y un *Movable Object*, que estará ligado a un nodo de escena. Esto quiere decir que un nodo de escena puede existir sin tener ninguna capacidad representativa, puesto que no es obligatorio que tenga ligado ningún `MovableObject`. Con esto se consigue que los cambios en la implementación de los objetos *rendarables* y en la implementación del grafo de escena no tengan efectos entre ellos, desacoplando la implementación de los mismos.

Es posible incluso ligar contenido definido por el programador a un nodo de escena, implementando una interfaz muy sencilla. Gracias a esto, se podrían por ejemplo añadir sonidos ligados a ciertas partes de una escena, que se reproducirían en una parte determinada de la misma.

El grafo de escena que se utiliza en OGRE se conoce como *scene manager* (gestor de escena) y está representado por la clase `SceneManager`. El interfaz de la misma lo implementan algunos de los gestores de escena que incluye OGRE, y también algunos otros desarrollados por la comunidad o incluso comerciales.

En OGRE es posible utilizar varios gestores de escena a la vez. De esto modo, es posible utilizar uno de interiores y de exteriores a la vez. Esto es útil por ejemplo cuando desde un edificio se mira por una ventana y se ve un terreno.

Un gestor de escena de OGRE es responsable entre otras cosas de descartar los objetos no visibles (*culling*) y de colocar los objetos visibles en la cola de *renderizado*.

BSP

El soporte de OGRE para BSP es histórico. Se usa sólo para cargar mapas de Quake3 y este es el único casa para el que se recomienda usarlos.

El futuro de OGRE

Los gestores de escenas de OGRE están evolucionando. Utilizar el gestor por defecto es una buena garantía de que será compatible con las siguientes versiones.

13.3.1. Interiores en OGRE

El gestor de escena para interiores de OGRE está basado en BSP. De hecho, este gestor se utiliza con mapas compatibles con *Quake 3*. Hay dos formas de referirse a este gestor, la primera como una constante de la enumeración `Ogre::SceneType` (`ST_INTERIOR`) y otra como una cadena (`"BspSceneManager"`) que se refiere al nombre del *Plugin* que lo implementa. La forma más moderna y la preferida es la segunda.

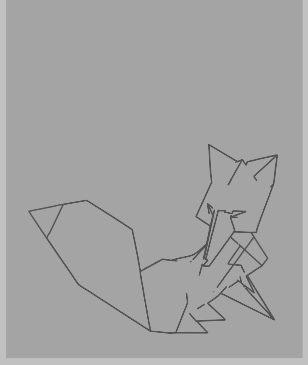
En la línea 9 se crea el `SceneManager` de BSP, y en la línea 30 se carga el mapa y se usa como geometría estática.

Cabe destacar que para poder navegar por los mapas BSP de manera correcta hay que rotar 90 grados en su vector **pitch** y cambiar el vector **up** de la cámara por el eje Z.

Listado 13.10: Ejemplo de carga de BSP en OGRE

```
_root = new Ogre::Root();

if(!_root->restoreConfig()) {
  _root->showConfigDialog();
  _root->saveConfig();
}

Ogre::RenderWindow* window = _root->initialise(true,"BSP");
_sceneManager = _root->createSceneManager("BspSceneManager");

Ogre::Camera* cam = _sceneManager->createCamera("MainCamera");
cam->setPosition(Ogre::Vector3(0,0,-20));
cam->lookAt(Ogre::Vector3(0,0,0));
cam->setNearClipDistance(5);
cam->setFarClipDistance(10000);
// Cambiar el eje UP de la cam, los mapas de Quake usan la Z
// Hacer los ajustes necesarios.
cam->pitch(Ogre::Degree(90));
cam->setFixedYawAxis(true, Ogre::Vector3::UNIT_Z);

Ogre::Viewport* viewport = window->addViewport(cam);
viewport->setBackgroundColour(Ogre::ColourValue(0.0,0.0,0.0));
double width = viewport->getActualWidth();
double height = viewport->getActualHeight();
cam->setAspectRatio(width / height);

loadResources();

[...]

_sceneManager->setWorldGeometry("maps/chiropteradm.bsp");
```

Se propone crear una escena utilizando un mapa BSP. Se sugiere utilizar el archivo *pk0* que se distribuye con OGRE. Pruebe a navegar por él con y sin el *fix* de la cámara. Añada alguna forma de mostrar los *FPS*. ¿En qué partes del mapa y en qué condiciones aumenta esta cifra? ¿Por qué?



Capítulo 14

Optimización de exteriores

Sergio Pérez Camacho

14.1. Introducción

Las diferencias entre una escena de interiores y una de exteriores son evidentes. Mientras una escena de interiores se dará en entornos cerrados, con muchas paredes o pasillos que dividen en espacio en habitaciones, una escena de exteriores normalmente no tiene ningún límite que no esté impuesto por la naturaleza. Si bien es cierto que es una escena de este tipo, por ejemplo, pudiesen existir colinas que se tapasen unas a las otras, si estamos situados frente a algunas de ellas en una posición muy lejana, el número de triángulos que se deberían representar sería tan elevado que quizá ningún hardware podría afrontar su renderizado.

Está claro que hay que afrontar la representación de exteriores desde un enfoque diferente: hacer variable el nivel de detalle (*level-of-detail* - **LOD**). De este modo, los detalles de una montaña que no se verían a cierta distancia no deberían renderizarse. En general, el detalle de los objetos que se muestran grandes en la pantalla (pueden ser pequeños pero cercanos), será mayor que el de los objetos menores.

Si bien el nivel de detalle es importante, tampoco se descarta el uso de oclusiones en algunos de los algoritmos que se presentarán a continuación, siguiendo de nuevo la propuesta de Dalmau. Se comenzará haciendo una pasada rápida por algunas de las estructuras de datos necesarias para la representación de exteriores eficiente.

14.2. Estructuras de datos

Uno de los principales problemas que se tienen que resolver para la representación de exteriores es la forma de almacenar escenas compuestas por grandes extensiones de tierra.

Las estructuras de datos utilizadas tendrán que permitir almacenar muchos datos, computar de manera eficiente el nivel de detalle necesario y permitir que la transición entre diferentes dichos niveles sea suave y no perceptible.

Mapas de altura

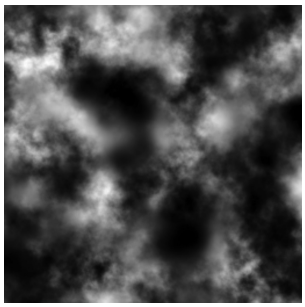


Figura 14.1: Mapa de altura (WIKIPEDIA - PUBLIC DOMAIN)

Los mapas de altura (*heigtfield*s o *heightmaps*) han sido utilizados desde hace mucho tiempo como forma para almacenar grandes superficies. No son más que imágenes en las que cada uno de sus píxeles almacenan una altura.

Cuando se empezó a usar esta técnica, las imágenes utilizaban tan solo la escala de grises de *8-bit*, lo que suponía poder almacenar un total de 256 alturas diferentes.

Los mapas de altura de hoy en día pueden ser imágenes de *32-bit*, lo que permite que se puedan representar un total de 4.294.967.296 alturas diferentes, si se usa el canal *alpha*.

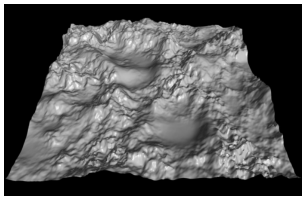


Figura 14.2: Mapa de altura renderizado (WIKIMEDIA COMMONS)

Para transformar uno de estos mapas en una representación 3D es necesario hacer uso de un vector de 3 componentes con la escala correspondiente. Por ejemplo, si se tiene un vector de escala $s = (3, 4, 0.1)$ quiere decir que entre cada uno los píxeles del eje X de la imagen habrá 3 unidades de nuestra escena, entre los de Y habrá 4, y que incrementar una unidad el valor del píxel significará subir 0.1 unidades. La posición varía según cómo estén situados los ejes.

Las ventajas de utilizar un mapa de altura es que se pueden crear con cualquier herramienta de manipulación de imágenes y que se pueden almacenar directamente en memoria como *arrays* de alturas que se transformarán en puntos 3D cuando sea necesario, liberando de este modo mucha memoria.

Las principal desventaja viene de que cada píxel representa una sola altura, haciendo imposible la representación de salientes o arcos. Normalmente todo este tipo de detalles tendrán que añadirse en otra capa.

Quadtrees

Un *quadtree* es un árbol donde cada nodo tendrá exactamente cuatro hijos, así, se dice que es un árbol 4-ario. Un *quadtree* divide un espacio en cuatro partes iguales por cada nivel de profundidad del árbol (figura 14.3).

Un *quadtree* permite que ciertas áreas del terreno se puedan representar con más detalle puesto que es posible crear árboles no equili-

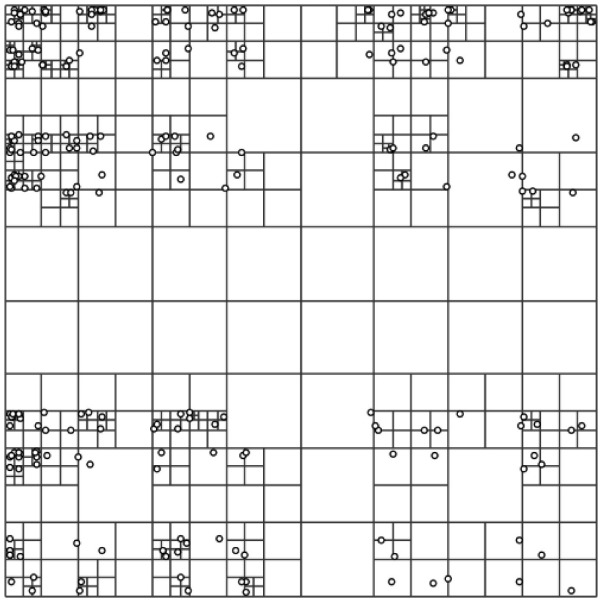


Figura 14.3: Representación de un *Quadtree* (WIKIMEDIA COMMONS - DAVID EPPSTEIN)

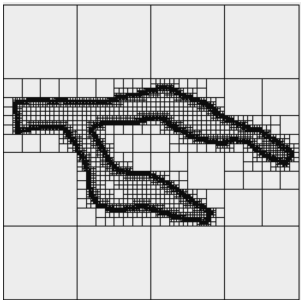


Figura 14.4: División de un terreno con un *quadtree* (QIUHUA LIANG)

brados, utilizando más niveles donde sea necesario. Una aproximación válida es comenzar con un mapa de altura y crear un *quadtree* a partir del mismo. Las partes del escenario donde exista más detalle (en el mapa de altura, habrá muchas alturas diferentes), se subdividirán más veces.

Además al representar la escena, es posible utilizar un heurístico basado en la distancia hasta la cámara y en el detalle del terreno para recorrer el árbol. Al hacerlo de este modo, será posible determinar qué partes hay que pintar con más detalle, seleccionando la representación más simple para los objetos más distantes y la más compleja para los más cercanos y grandes.

En el momento de escribir esta documentación, *InfiniteCode* permite descargar de su web [Inf02] un ejemplo de *Quadtrees* (apoyados en mapas de altura).

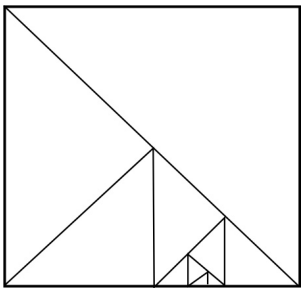


Figura 14.5: División de una superficie con un BTT.

Árboles binarios de triángulos (BTT)

Un BBT es un caso especial de un árbol binario. En cada subdivisión se divide al espacio en dos triángulos. El hecho de que cada nodo tenga menos descendientes que un *quadtree* e igualmente menos vecinos, hace que esta estructura sea mejor para algoritmo de nivel de detalle continuo.

Dallaire [Dal06] explica con detalle en *Gamasutra* cómo generar este tipo de estructuras para indexar fragmentos de terreno.

14.3. Determinación de la resolución

En una escena de exteriores, a parte de utilizar la estructura de datos correcta, es crucial disponer de un método para determinar la resolución de cada uno de los objetos de la escena.

Cada objeto podrá aparecer con diferentes resoluciones y para ello serán necesarias dos cosas: la forma de seleccionar la resolución correcta y la manera de representarla, esto es, un algoritmo de renderizado que permita hacerlo de manera correcta.

Determinar la resolución de manera exacta no es un problema abordable en un tiempo razonable, puesto que son muchas las variables que influyen en dicha acción. Lo mejor es afrontar este problema utilizando algún heurístico permita aproximar una buena solución.

Un primer enfoque podría ser utilizar la distancia desde la cámara hasta el objeto, y cambiar la resolución del mismo según objeto se acerca (más resolución) o se aleja (menos resolución). Puede que haya un objeto muy lejano pero que sea tan grande que requiera un poco más del detalle que le corresponda según la distancia. Un heurístico añadido que se podría utilizar es el número de píxeles que ocupa aproximadamente en el espacio de pantalla, utilizando una *boudingbox* y proyectándola sobre la misma. Incluso se podría utilizar el hardware, como se ha visto en el tema anterior, para realizar estas comprobaciones.

Una vez que se ha determinado cómo se selecciona la resolución el siguiente paso será aplicar la política de dibujado. Existe una gran división, dependiendo de la continuidad de los modelos a pintar. Si se almacenan diferentes modelos de un mismo objeto, con niveles diferentes de detalle, hablaremos de una política discreta de LOD. Si el detalle de los modelos se calcula en tiempo real según el criterio de resolución, hablaremos de una política de LOD continua.

14.3.1. Políticas Discretas de LOD

Si se utiliza una política discreta, se tendrán varias representaciones del mismo objeto, con diferentes nivel de detalle. Este nivel de detalle irá desde la versión original, con el mayor detalle posible, a una versión con muy pocos triángulos. Se creará la versión de alto detalle y se generarán versiones simplificadas reduciendo el número de triángulos gradualmente con alguna herramienta de diseño 3D.

Teniendo una tabla con diferentes modelos donde elegir, el algoritmo de pintado simplemente tiene que elegir el que corresponda y ponerlo en la cola de renderizado. El problema de estas políticas es que existen un momento en el que se produce un cambio notable en el objeto, y es completamente perceptible si no se disimula de alguna forma.

Una de las técnica que se usa para ocultar el salto que se produce al cambiar de modelo es utilizar *alpha blending* entre el modelo origen y destino. Este efecto se puede ver como un *cross-fade* entre los mismos,

cuya intensidad dependerá del heurístico utilizado. Así en la mitad de la transición, cada modelo se renderizará con un *alpha* de 0.5 (o del 50 %). Justo antes de empezar la transición, el modelo origen tendrá un valor *alpha* de 1 y el destino de 0, y al finalizar tendrán los valores intercambiados. Un inconveniente muy importante de esta técnica es que durante un tiempo durante el cual antes sólo se representaba un objeto, ahora se representarán dos, lo que supone una sobrecarga de la GPU.

14.3.2. Políticas Continuas de LOD

Si se quiere evitar del todo el salto producido por el intercambio de modelos, se podría implementar una forma de reducir el número de triángulos en tiempo real, y generar un modelo dependiendo de la resolución requerida.

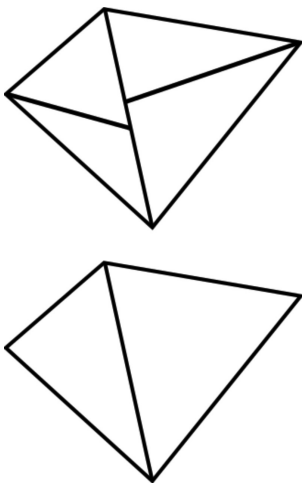


Figura 14.6: *Edge-Collapsing*. Arriba, el *strip* original. Abajo, el nuevo *strip* después de aplicar la técnica.

Este tipo de cálculos en tiempo real son muy costosos, porque hay que determinar qué aristas, vértices o triángulos se pueden eliminar y además aplicar esa modificación al modelo.

Hoppe [Hop98] propone una implementación eficiente de lo que llama mallas progresivas (*Progressive Meshes*). La técnica se basa en la eliminación de aristas de la malla, convirtiendo dos triángulos en sólo uno por cada arista eliminada (*edge-collapsing*). Hoppe determina que esta técnica es suficiente para simplificar mallas y propone algunos heurísticos para eliminar las aristas.

Hay dos posibles aproximaciones a la hora de quitar una arista, la primera, crear un nuevo vértice en el centro de la misma, y la otra eliminarla completamente (más eficiente). La primera aproximación es válida para todas las aristas, la segunda es sólo válida para aristas que corten a triángulos y no a cuadriláteros, puesto que al eliminarla se debería obtener un polígono con tres vértices.

Un posible heurístico para utilizar es el ángulo que forman los dos triángulos conectados por dicha arista. Si el ángulo es menor que un umbral, se quitará esa arista. Como optimización, esta técnica no debería utilizarse en cada *frame*, sino sólo cuando cambie la distancia o el área que ocupa el objeto en pantalla lo suficiente.

Este tipo de políticas permite obtener el mejor resultado, a costa de añadir un coste computacional. Además, otra desventaja muy importante es que la información de mapeado de las texturas del objeto se podrá ver afectada por la reducción del número de triángulos.

14.4. Técnicas y Algoritmos

Las estructuras de datos presentadas anteriormente se utilizan en las técnicas y algoritmos que se presentan a continuación.

14.4.1. *GeoMipmapping*

De Boer [DB00] presenta el *GeoMipmapping* como una técnica para representar de manera eficiente grandes terrenos. En su artículo, De Boer divide el algoritmo en tres fases diferentes: la representación en memoria de los datos del terreno y a qué corresponderá en la representación, el *frustum culling* con los pedazos de terreno disponibles (*chunks*) y por último, describe los *GeoMipMaps* haciendo una analogía con la técnica de *mipmapping* usada para la generación de texturas.

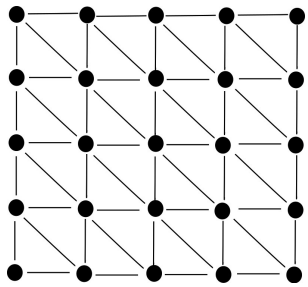


Figura 14.7: Malla con los datos del terreno. Cada círculo es un vértice.

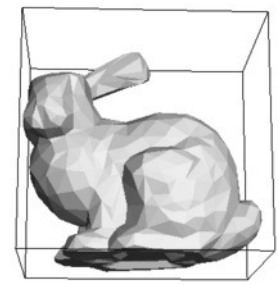
Representación del terreno

La representación del terreno elegida es la de una malla de triángulos cuyos vértices están separados por la misma distancia en el eje X y en el eje Z.

El número de vértices horizontales y verticales de la malla tendrá que ser de la forma $2^n + 1$, lo que significa tener mallas con 2^n cuadriláteros, que tendrán 4 vértices compartidos con sus vecinos. Cada cuadrilátero está compuesto de dos triángulos, que serán los que se mandarán a la cola de representación.

Cada vértice tendrá un valor fijo de X y de Z, que no cambiará durante el desarrollo del algoritmo. El valor de Y (altura) será leído de un mapa de altura de *8-bit*, que tendrá exactamente las misma dimensiones que la malla. Posteriormente esta se cortará en pedazos de tamaño 2^n+1. Estos pedazos se usarán en un *quadtree* para realizar el *frustum culling*, y como primitivas de nivel 0 para los *GeoMipMaps*. En la figura 14.7 se muestra una de estas mallas, donde n vale 2.

Una ventaja de utilizar este tipo de representación es que los pedazos de malla se pueden mandar como una sola primitiva (*strips*) el hardware. La desventaja es que los vértices de los 4 bordes de dichos trozos se comparten con los bloques que lo rodean, y se transformarán dos veces.



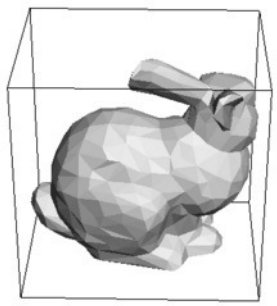


Figura 14.8: Ejemplo de una *bounding-box* (MATH IMAGES PROJECT).

View-Frustum Culling

Será necesario descartar los pedazos de terreno que no caigan dentro del *frustum* de la vista (cámara) puesto que no serán visibles. Para ello, lo ideal es utilizar un *quadtree*.

El *quadtree* será precalculado antes de comenzar la parte interactiva de la aplicación y consistirá tan solo en *bounding boxes* de tres dimensiones, que a su vez contendrán otras correspondientes a los subnodos del nodo padre. En cada hoja del árbol quedará un pedazo del nivel 0 de divisiones que se ha visto al principio. Es suficiente utilizar *quadtrees* y no *octrees* ya que la división se realiza de la superficie, y no del espacio.

Para descartar partes del terreno, se recorrerá el árbol desde la raíz, comprobando si la *bounding box* está dentro del *frustum* al menos parcialmente, y marcando dicho nodo en caso afirmativo. Si una

hoja está marcada, quiere decir que será visible y que se mandará a la cola de representación. A no ser que el terreno sea muy pequeño, se terminarán mandando muchos triángulos a la cola, y esta optimización no será suficiente. Es aquí donde De Boer introduce el concepto de *Geomipmapping*.

Geomipmaps y nivel de detalle

Esta técnica se basa en el hecho de que los bloques que están más lejos de la cámara no necesitan representarse con tan nivel de detalle como los más cercanos. De este modo, podrán ser representados con un número mucho menor de triángulos, lo que reducirá enormemente el número de triángulos del terreno que se mandarán a la cola de renderizado. Otro algorítmos utilizan una aproximación en la que hay que analizar cada triángulo para poder aplicar una política de nivel de detalle. Al contrario, esta técnica propone una política discreta que se aplicará en un nivel más alto.

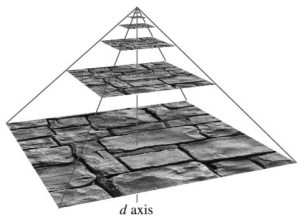


Figura 14.9: Construcción de los *mipmaps*. En cada nivel, la textura se reduce a un cuarto de su área. (AKENINE-MOLLER)

La técnica clásica de *mipmapping* se aplica a las texturas, y consiste en la generación de varios niveles de subtexturas a partir de la original. Este conjunto de texturas se utilizan en una política de nivel de detalle para texturas, usando unas u otras dependiendo de la distancia a la cámara. Esta es la idea que se va a aplicar a la mallas 3D de terrenos.

Cada bloque de terreno tendrá asociado varios *mipmaps*, donde el original corresponde al bloque del mapa de altura. Estos *GeoMipMaps* pueden ser precalculados y almacenados en memoria para poder ser utilizados en tiempo de ejecución directamente.

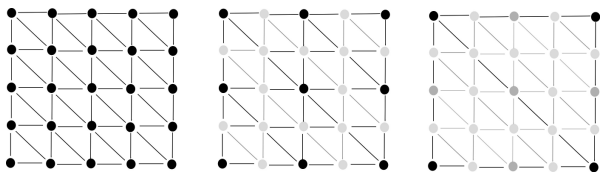


Figura 14.10: Diferentes niveles de detalle de la malla desde los *GeoMipMaps*: niveles 0, 1 y 2.

Para elegir qué *geomipmap* es adecuado para cada distancia y evitar saltos (producto de usar distancias fijas para cada uno) habrá que utilizar un método un poco más elaborado. En el momento en que se se pase del nivel 0 al 1 (ver figura 14.10) existirá un error en la representación del terreno, que vendrá dado por la diferencia en altura entre ambas representaciones. Debido a la perspectiva, la percepción del error tendrá que ver con los píxeles en pantalla a la que corresponda esa diferencia. Ya que cada nivel tiene muchos cambios en altura, se utilizará el máximo, que podrá almacenarse durante la generación para realizar decisiones más rápidas. Si el error en píxeles cometido es menor que un umbral, se utilizará un nivel más elevado.

Hay que tener en cuenta que la mayoría de las ocasiones el terreno estará formado por bloques con deferentes niveles, lo que puede hacer que existan vértices no conectados. Será necesario reorganizar las conexiones entre los mismos, creando nuevas aristas.

En la figura 14.11 se muestra una propuesta que consiste en conectar los vértices de la malla de nivel superior con los de la de nivel inferior pero saltando un vértice cada vez.

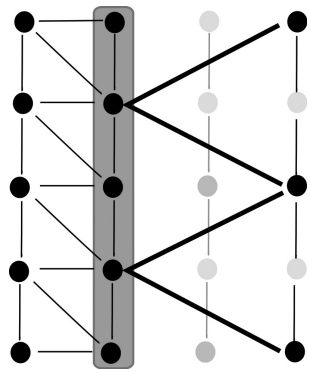


Figura 14.11: Unión de diferentes niveles de *GeoMipmaps*. En rojo la frontera común.

14.4.2. ROAM

Duchaineau [DWS+97] propone ROAM (*Real-time Optimally Adapting Meshes*) como un enfoque de nivel de detalle continuo al problema de la representación de exteriores. El algoritmo que propone combina una buena representación del terreno (lo que facilitará el *culling*) con un nivel de detalle dinámico que cambiará la resolución del terreno según la disposición de la cámara. ROAM es un algoritmo complejo, como el de los BSPs, que está dividido en dos pasadas y que permite una representación muy rápida de terrenos.

En este algoritmo, la malla se calcula en tiempo real, utilizando precálculos sobre la resolución necesaria en cada caso. En la primera pasada se rellena un BTT con la información geográfica, añadiendo información sobre el error producido al entrar en un subnodo (que será usada para detectar zonas que necesiten un mayor nivel de detalle). En la segunda pasada se construirá otro BTT, que será el encargado de crear la maya y de representar el terreno.

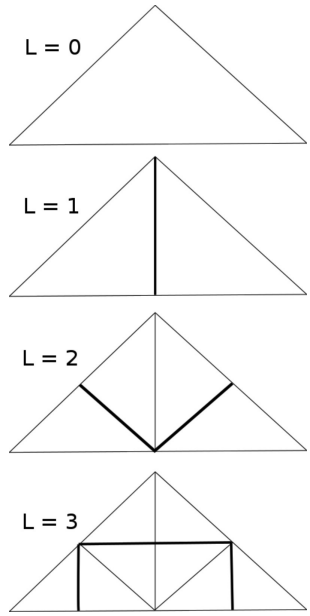


Figura 14.12: Distintos niveles de división en un BTT.

Primera pasada: Árbol de Varianza.

En la primera pasada se construirá un árbol que almacenará el nivel de detalle que existe en el terreno. Una buena métrica es la varianza. En cada hoja del árbol (un píxel de un mapa de altura), la varianza almacenada será por ejemplo la media de los píxeles que lo rodean. La varianza de los nodos superiores será la máxima varianza de los nodos hijos.

Segunda pasada: Malla.

Se utilizará un BTT, donde el nodo raíz representa un terreno triangular (si se quiere representar un terreno rectangular se necesitarán los nodos como este). Se almacenará la información de la altura para cada una de las esquinas del triángulo almacenado en cada nodo.

Si entre los vértices de este triángulo grande la información del terreno no es coplanar, se consultará el árbol de varianza para determinar si es conveniente explorar una nueva división para añadir un nivel más de detalle. Se rellenará el BTT hasta que no queden píxeles por añadir del mapa de altura, o hasta que el estadístico sea menor que un umbral elegido.

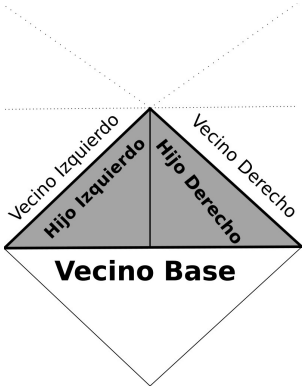


Figura 14.13: Etiquetado de un triángulo en ROAM. El triángulo junto a su vecino base forman un diamante.

De este modo se construye un árbol que representa la malla, donde cada nivel del mismo corresponde a un conjunto de triángulos que pueden ser encolados para su renderizado. El árbol podrá expandirse en tiempo real si fuera necesario. Bajar un nivel en el árbol equivale a una operación de partición y subir a una de unión.

El problema viene de la unión de regiones triangulares con diferente nivel de detalle, ya que si no se tienen en cuenta aparecerán agujeros en la malla representada. Mientras que en el *geomipmapping* se parchea la malla, en ROAM, cuando se detecta una discontinuidad se utiliza un *oversampling* de los bloques vecinos para asegurar que están conectados de manera correcta. Para ello, se añade información a cada lado de un triángulo y se utilizan alguna reglas que garantizarán la continuidad de la malla.

Se conoce como vecino base de un triángulo al que está conectado a este a través de la hipotenusa. A los otros dos triángulos vecinos se los conocerá como vecino izquierdo y derecho (figura 14.13). Analizando diferentes árboles se deduce que el vecino base será del mismo nivel o del anterior (menos fino) nivel de detalle, mientras que los otros vecinos podrán ser del mismo nivel o de uno más fino.

Las reglas propuestas para seguir explorando el árbol y evitar roturas en la malla serán las siguientes:

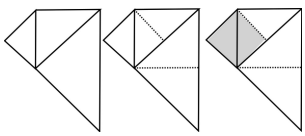


Figura 14.14: Partición recursiva hasta encontrar un diamante para el nodo de la izquierda.

- Si un nodo es parte de un diamante, partir el nodo y el vecino base.
- Si se está en el borde de una malla, partir el nodo.
- Si no forma parte de un diamante, partir el nodo vecino de forma recursiva hasta encontrar uno antes de partir el nodo actual (figura 14.14).

El algoritmo en tiempo de ejecución recorrerá el árbol utilizando alguna métrica para determinar cuándo tiene que profundizar en la jerarquía, y cómo ha de realizar las particiones cuando lo haga dependerá de las reglas anteriores.

Realizar en tiempo real todos estos cálculos es muy costoso. Para reducir este coste, se puede dividir el terreno en *arrays* de BTTs y sólo recalcular el árbol cada ciertos *frames* y cuando la cámara se haya movido lo suficiente para que cambie la vista por encima de un umbral.

Otra aproximación para optimizar el algoritmo es utilizar el *frustum* para determinar qué nodos hay que reconstruir de nuevo, marcando los triángulos que quedan dentro o fuera, o parcialmente dentro. Sólo estos últimos necesitarán atención para mantener el árbol actualizado. A la hora de representarlo, lo único que habrá que hacer es mandar los nodos marcados como dentro, total o parcialmente.

14.4.3. *Chunked* LODs

Ulrich [Ulr02] propone un método para representar grandes extensiones de tierra. En la demo del SIGGRAPH 2002 incluye un terreno que cubre $160Km^2$. Este método tiene su partida en una imagen muy grande, por ejemplo obtenida de un satélite, lo que hace el método ideal para la implementación de simuladores de vuelo.

Para almacenar la información se utilizará un *quadtree*. Se comenzará con una imagen potencia de dos en el nodo raíz, que corresponderá a una imagen de muy baja resolución del terreno completo. Según se vaya profundizando, los 4 subnodos hijos contendrán imágenes del mismo tamaño pero con la calidad resultante de hacer un zoom a los cuatro cuadrantes de la misma (figura 14.15).

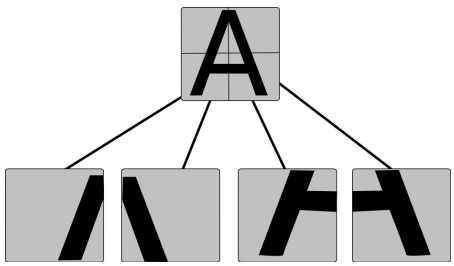


Figura 14.15: Nodo y subnodos de las texturas en un *quadtree* para Chunked LODs.

A cada uno de los nodos se le añade información acerca de la pérdida del nivel de detalle se produce al subir un nivel en la jerarquía. Si las hojas contienen imágenes de 32x32 píxeles, los pedazos de terreno contendrán 32x32 vértices. El árbol de mallas también forma parte del preproceso normalmente, partiendo de una malla muy grande y con mucha resolución y partiéndola en trozos más pequeños.

Para renderizar el *quadtree* bastará con recorrerlo realizando *clipping* jerárquico, utilizando algún umbral para comparar con un valor obtenido de computar la distancia a la cámara, el error que produce un nivel determinado y la proyección de perspectiva que se está utilizando. Si la comparación sugiere que el nivel de detalle debería incrementarse en esa región, se procederá a encolar un nivel más bajo del árbol recursivamente.

Ulrich propone unir los *chunk*s simplemente prolongando un poco sus fronteras con unos faldones. Como la proyección de la textura dependerá sólo de la X y la Z, las uniones serán prácticamente invisibles. Este algoritmo se beneficia de un *quadtree* para texturizar el terreno. Como contrapartida, el principal problema del mismo es la gran cantidad de memoria que necesita (la demo del SIGGRAPH, más de 4GB), lo que hace que sea necesario prestar especial atención a la misma, cargando en memoria los nuevos pedazos cuando sean necesarios, descartando los que no se necesitan hace tiempo.

14.4.4. Terrenos y GPU

En su libro, Dalmau propone una aproximación diferente a la representación de terrenos utilizando simplemente la GPU. La premisa de la que parte el algoritmo es mantener a la CPU completamente desocupada, pudiendo ser utilizada esta para otra labores como para la inteligencia artificial o para el cálculo de las colisiones o las físicas.

De esto modo sugiere que la geometría del terreno tendrá que almacenarse en algún modo en el que la GPU pueda acceder a ella sin la intervención de la CPU, seleccionando bloques de 17x17 vértices (512 triángulos), que serán analizados e indexados para maximizar el rendimiento. Además, como diferentes bloques compartirán vértices, estos se almacenarán sólo una vez y se indexarán de forma eficiente. Así, la CPU sólo tendrá que determinar los bloques visibles y mandar esta información a la GPU para que los pinte.

A esta técnica se le puede sumar el uso de *bounding boxes* para cada bloque de terreno y la construcción de un *PVS* o incluso implementar alguna política de LOD (que Dalmau define como no necesaria excepto en GPU con limitaciones muy restrictivas de número de triángulos por segundo).

14.4.5. *Scenegraphs* de Exteriores

Una escena de exteriores es mucho más grande que una de interiores. La cantidad de datos que es necesario manejar es en consecuencia gigantesca en comparaciones con escenas mucho más pequeñas. Aunque se tenga una buena política de nivel de detalle, hay que tener en cuenta que el número de triángulos totales que se tendrán que manejar es enorme.

Para implementar un grafo de escena de exteriores es importante tener en cuenta que:

- Cada objeto sólo tendrá una instancia, y lo único que se almacenará aparte de esta serán enlaces a la misma.
- Es necesario implementar alguna política de nivel de detalle, puesto que es imposible mostrar por pantalla (e incluso mantener en memoria) absolutamente todos los triángulos que se ven en una escena.
- Se necesita una rutina muy rápida para descartar porciones no visibles del terreno y del resto de objetos de la escena.

Mientras que para almacenar un terreno y las capas estáticas que lleva encima es muy buena opción utilizar un *quadtree*, será mejor utilizar alguna tabla de tipo rejilla para almacenar la posición de los objetos dinámicos para determinar cuáles de ellos se pintar en cada fotograma de manera rápida.

¿Reconoce alguna de estas técnicas en algún juego o aplicación que haya utilizado recientemente?

14.5. Exteriores y LOD en OGRE

OGRE da soporte a diferentes estilos de escenas. El único *scene manager* que es de uso exclusivo para interiores es `ST_INTERIOR` (el gestor de portales parece estar abandonado en los ejemplos de ogre 1.7.3). El resto gestores está más o menos preparado para escenas de exteriores. Estos son:

- **`ST_GENERIC`** - Gestor de propósito general, adecuado para todo tipo de escenas, pero poco especializado. Se utiliza un *octree* para almacenar los objetos.
- **`ST_EXTERIOR_CLOSE`** - Gestor de terrenos antiguo de OGRE. Soportado hasta la versión 1.7, deprecado en la 1.8.
- **`ST_EXTERIOR_REAL_FAR`** - Gestor que permite dividir la escena en un conjunto de páginas. Sólo se cargarán las páginas que se usen en un momento determinado, permitiendo representar escenas muy grandes de cualquier tamaño. Cada página tiene asociado un mapa de altura, y se pueden aplicar diferentes texturas a la malla generada según la altura.

14.5.1. Terrenos

OGRE soporta la creación de terrenos utilizando mapas de altura, que se pueden cargar en diferentes páginas y cubrir grandes extensiones en una escena.

En estos momentos OGRE está cambiando de gestor de terrenos. El gestor nuevo utiliza *shaders* que dependen del plugin *CgProgramManager*, válido únicamente para tarjetas gráficas NVIDIA. Esta dependencia se da sólo utilizando en sistema de render basado en OpenGL, porque lo mientras en las demos de OGRE de Windows funciona perfectamente con DirectX, en GNU/Linux hará falta cargar este *plugin*. En Debian, ya no se distribuye este plugin con lo que será necesario compilarlo desde cero.

Un terreno de OGRE implementa una política de nivel de detalle continua. El nuevo gestor implementa una variante de *Chunked-LODS* y permite cargar mapas muy grandes, compuestos por diferentes mapas de altura.

CG Plugin

Para compilar el plugin *CgManager* de OGRE en Debian, habrá que instalar antes *nvidia-cg-toolkit*, que es privativo.

A continuación se muestra un ejemplo de uso del nuevo terreno de OGRE. El *Scene Manager* que utiliza es el genérico. El mapa de altura que cargará por secciones en el mismo.

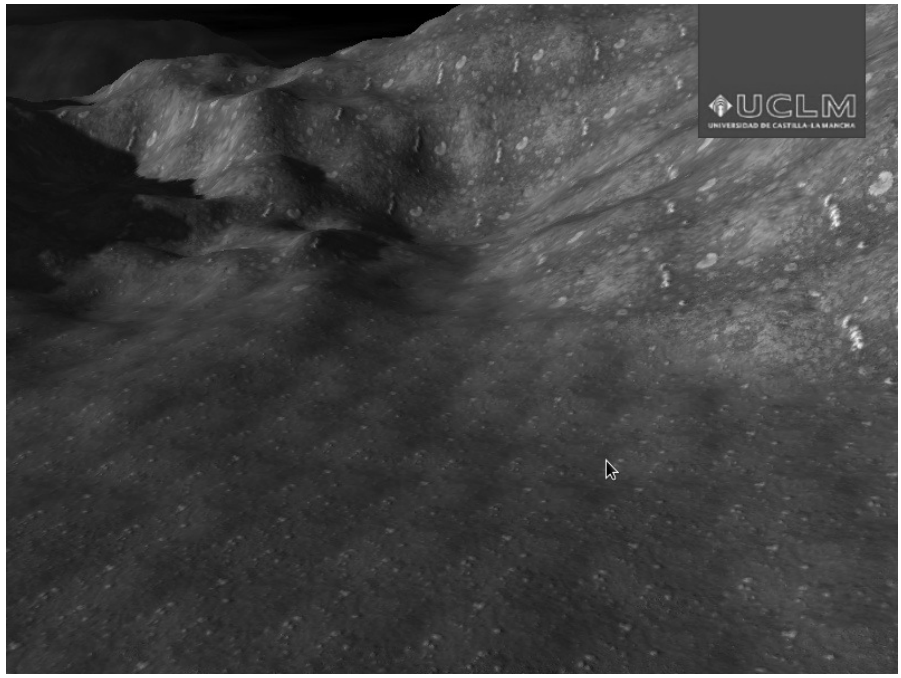


Figura 14.16: Pantallazo de terreno en OGRE

En la lineas [3–5] se configura niebla en la escena, usando un color oscuro para hacer que parezca oscuridad. En las líneas [7–14] se configura una luz direccional que se utilizará en el terreno. A partir de ella se calcularán las sombras en el mismo. En la línea [16] se configura la luz ambiental de la escena. En la [19] se crea un nuevo objeto de configuración del terreno, que se rellenará más adelante. En la [20] se crea un objeto agrupador de terrenos. Este objeto es el responsable de agrupar diferentes pedazos del terreno juntos para que puedan ser representados como proceda. En la línea [26] se configura el error máximo medido en píxeles de pantalla que se permitirá al representar el terreno. En las líneas [28–31] se configura la textura que compodrá con él, dibujando las sombras. Primero se determina la distancia de representación de las luces, luego se selecciona la dirección de las sombras, que parte de una luz direccional y luego se configuran el valor ambiental y difuso de iluminación.

Listado 14.1: Ejemplo de terreno en OGRE. Basado en OGRE3D WIKI

```
1  void MyApp::createScene()
2  {
3    _sMgr->setFog(Ogre::FOG_LINEAR,
4                  Ogre::ColourValue(0.1, 0.1, 0.1),
5                  0.5, 2000, 5000);
6
7    Ogre::Vector3 lightdir(0.55, -0.3, 0.75);
8    lightdir.normalise();
9
10   Ogre::Light* light = _sMgr->createLight("DirLight");
11   light->setType(Ogre::Light::LT_DIRECTIONAL);
12   light->setDirection(lightdir);
13   light->setDiffuseColour(Ogre::ColourValue::White);
14   light->setSpecularColour(Ogre::ColourValue(0.4, 0.4, 0.4));
15
16   _sMgr->setAmbientLight(Ogre::ColourValue(0.2, 0.2, 0.2));
17
18
19   _tGlobals = OGRE_NEW Ogre::TerrainGlobalOptions();
20   _tGroup = OGRE_NEW Ogre::TerrainGroup(_sMgr,
21                                         Ogre::Terrain::ALIGN_X_Z,
22                                         513, 12000.0f);
23
24   _tGroup->setOrigin(Ogre::Vector3::ZERO);
25
26   _tGlobals->setMaxPixelError(8);
27
28   _tGlobals->setCompositeMapDistance(3000);
29   _tGlobals->setLightMapDirection(light->getDerivedDirection());
30   _tGlobals->setCompositeMapAmbient(_sMgr->getAmbientLight());
31   _tGlobals->setCompositeMapDiffuse(light->getDiffuseColour());
32
33   Ogre::Terrain::ImportData& di;
34   di = _tGroup->getDefaultImportSettings();
35
36   di.terrainSize  = 513;
37   di.worldSize    = 12000.0f;
38   di.inputScale   = 600;
39   di.minBatchSize = 33;
40   di.maxBatchSize = 65;
41
42   di.layerList.resize(3);
43   di.layerList[0].worldSize = 100;
44   di.layerList[0].textureNames.push_back("dirt_diff_spec.png");
45   di.layerList[0].textureNames.push_back("dirt_normal.png");
46   di.layerList[1].worldSize = 30;
47   di.layerList[1].textureNames.push_back("grass_diff_spec.png");
48   di.layerList[1].textureNames.push_back("grass_normal.png");
49   di.layerList[2].worldSize = 200;
50   di.layerList[2].textureNames.push_back("growth_diff_spec.png");
51   di.layerList[2].textureNames.push_back("growth_normal.png");
52
53
54   Ogre::Image im;
55   im.load("terrain.png",
56           Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
57
58   long x = 0;
59   long y = 0;
60
61   Ogre::String filename = _tGroup->generateFilename(x, y);
62   if (Ogre::ResourceGroupManager::getSingleton().resourceExists(
         _tGroup->getResourceGroup(), filename))
63   {
```

```
64      _tGroup->defineTerrain(x, y);
65    }
66    else
67    {
68      _tGroup->defineTerrain(x, y, &im);
69      _tsImported = true;
70    }
71
72    _tGroup->loadAllTerrains(true);
73
74    if (_tsImported)
75    {
76      Ogre::TerrainGroup::TerrainIterator ti;
77      ti = _tGroup->getTerrainIterator();
78      while(ti.hasMoreElements())
79      {
80        Ogre::Terrain* t = ti.getNext()->instance;
81        initBlendMaps(t);
82      }
83    }
84
85    _tGroup->freeTemporaryResources();
86
87    Ogre::Plane plane;
88    plane.d = 100;
89    plane.normal = Ogre::Vector3::NEGATIVE_UNIT_Y;
90
91    _sMgr->setSkyPlane(true, plane,"Skybox/SpaceSkyPlane",
92                       500, 20, true, 0.5, 150, 150);
93
94  }
95
96  void MyApp::initBlendMaps(Ogre::Terrain* t)
97  {
98    Ogre::TerrainLayerBlendMap* blendMap0 = t->getLayerBlendMap(1);
99    Ogre::TerrainLayerBlendMap* blendMap1 = t->getLayerBlendMap(2);
100
101   Ogre::Real minHeight0  = 70;
102   Ogre::Real fadeDist0   = 40;
103   Ogre::Real minHeight1  = 70;
104   Ogre::Real fadeDist1   = 15;
105
106   float* pBlend1 = blendMap1->getBlendPointer();
107   for (Ogre::uint16 y = 0; y < t->getLayerBlendMapSize(); ++y) {
108     for (Ogre::uint16 x = 0; x < t->getLayerBlendMapSize(); ++x){
109       Ogre::Real tx, ty;
110
111       blendMap0->convertImageToTerrainSpace(x, y, &tx, &ty);
112       Ogre::Real height = t->getHeightAtTerrainPosition(tx, ty);
113       Ogre::Real val = (height - minHeight0) / fadeDist0;
114       val = Ogre::Math::Clamp(val, (Ogre::Real)0, (Ogre::Real)1);
115       val = (height - minHeight1) / fadeDist1;
116       val = Ogre::Math::Clamp(val, (Ogre::Real)0, (Ogre::Real)1);
117       *pBlend1++ = val;
118     }
119   }
120   blendMap0->dirty();
121   blendMap1->dirty();
122   blendMap0->update();
123   blendMap1->update();
124 }
```

En [34] se obtiene la instancia que configura algunos parámetros del terreno. El primero es el tamaño del terreno, que corresponde al número de píxeles del mapa de altura. El segundo el tamaño total del mundo, en unidades virtuales. El tercero (`inputScale`) corresponde a la escala aplicada al valor del píxel, que se transformará en la altura del mapa. Las dos siguientes corresponden al tamaño de los bloques de terrenos que se incluirán en la jerarquía (diferentes LOD). Estos tres últimos valores tendrán que ser del tipo $2^n + 1$. El atributo `layerList` contiene un vector de capas, que se rellenará con las texturas (color y mapa de normales en este caso). El atributo `worldSize` corresponde a la relación de tamaño entre la textura y el mundo. En [55] se carga la imagen con el mapa de altura.

Las siguientes líneas son las que asocian el mapa de altura con el terreno que se va a generar. En este ejemplo sólo se genera el subterreno (0,0) con lo que sólo se cargará una imagen. En las siguientes líneas [61–70] se define el terreno, en este caso, sólo para el *slot* (0,0), aunque el ejemplo está preparado para ser extendido fácilmente y añadir la definición algunos más. La llamada `defineTerrain()` expresa la intención de crear ese pedazo de terreno con la imagen que contiene el mapa de altura. La ejecución de este deseo se realiza en la linea [72]. Sin esta llamada, el ejemplo no funcionaría puesto que se ejecutaría el siguiente paso sin haber cargado (generado) el terreno.

En el bloque [74–83] se crea la capa de *blending*, esto es, la fusión entre las tres texturas del terreno (habrá dos mapas de *blending*). Esta operación se realiza en el método `initBlendMaps` [96–124]. Dependiendo de la altura a la que corresponda un píxel de las imágenes a fusionar (el tamaño ha de ser coherente con el mapa de altura) así será el valor de *blending* aplicado, con lo que se conseguirá que cada altura presente una textura diferente.

En las líneas [87–89] se añade un plano, y justo debajo, en la [91] se usa como un *Skyplane*.

14.5.2. *Skyboxes*, *skydomes* y *skyplanes*

Una de las características de una escena de exteriores es que muy probablemente se llegue a ver el horizonte, una gran parte del cielo, o si estamos en un entorno marino o espacial, el abismo o las estrellas. Representar con alto de nivel de detalle los objetos del horizonte es practicamente imposible. Una de las razones es que probablemente alguna de las políticas de *culling* o de nivel de detalle hará que no se pinte ningún objeto a partir de una distancia determinada de la cámara.

La solución adopotada por muchos desarrolladores y que está disponible en OGRE es utilizar imágenes estáticas para representar lo que se ve en el horizonte. En OGRE existen tres formas de representar el entorno que contiene a todos los objetos de la escena. Es decir, no habrá ningún objeto que quede fuera de las imágenes que representan el horizonte. Los objetos que brinda OGRE son de tipo:

- **skybox** - Como su nombre indica es una caja. Normalmente son cubos situados a una distancia fija de la cámara. Así, será necesario proporcionar seis texturas, una para cada uno de sus caras.
- **skydome** - Corresponde con una cúpula de distancia fija a la cámara. Con una textura es suficiente. Como contrapartida, una *skydome* sólo cubre la mitad de una esfera sin distorsionarse y, mientras una *skybox* posee un suelo, una *skydome* no.
- **skyplane** - El cielo está representado por un plano fijo a la cámara, con lo que parecerá infinito. Con una sóla textura es suficiente.

Ejemplo de *Skybox*

Para utitlizar una *skybox* primero es necesario definir un material:

```
material SkyBoxCEDV
{
  technique
  {
    pass
    {
      lighting off
      depth_write off

      texture_unit
      {
        cubic_texture cubemap_fr.jpg cubemap_bk.jpg cubemap_lf.jpg\
           cubemap_rt.jpg  cubemap_up.jpg cubemap_dn.jpg separateUV
        tex_address_mode clamp
       }
     }
   }
}
```

`setSkyBox()`

Esta llamada acepta más parámetros, si fuera necesario definir una orientación tendría que usarse un quinto parámetro (el cuarto es un booleano que indica que la caja se dibuja antes que el resto de la escena.

En este material se deshabilita la escritura en el *buffer* de profundidad (`depth_write off`) para que la representación del cubo no afecte a la visibilidad del resto de los objetos. También se ha de representar sin iluminación (`lighting off`). Con `cubic_texture` se determina que la textura será cúbica. Existen dos formas de declarar qué imágenes se usarán para la misma. La del ejemplo anterior consiste en enumerar las seis caras. El orden es:

```
<frontal> <posterior> <izquierda> <derecha> <arriba> <abajo>
```

Otra forma es utilizar la parte común del nombre de los archivos que queda delate de "_". En el caso anterior, sería `cubemap.jpg`. El último parámetro puede ser `combinedUVW`, si en una misma imagen están contenidas las seis caras, o `separateUV` si se separan por imágenes. La primera forma utilizará coordenadas de textura 3D, la segunda usará coordenadas 2D ajustando cada imagen a cada una de las caras. Usando `tex_address_mode clamp` hará que los valores de coordenadas de texturizado mayores que 1.0 se queden como 1.0.

Para utilizar una *Skybox* simplemente habrá que ejecutar esta línea:

```
1 sceneManager->setSkyBox(true, "SkyBoxCEDV", 5000);
```

Como cabría esperar, la *skybox* se configura para el gestor de escenas. Ésta se encontrará a una distancia fija de la cámara (el tercer parámetro), y podrá estar activada o no (primer parámetro).

Ejemplo de *SkyDome*

Aunque sólo se utiliza una textura, realmente una *SkyDome* está formada por las cinco caras superiores de un cubo. La diferencia con una *skybox* es la manera en la que se proyecta la textura sobre dichas caras. Las coordenadas de texturizado se generan de forma curvada y por eso se consigue tal efecto de cúpula. Este tipo de objetos es adecuado cuando se necesita un cielo más o menos realista y la escena no va a contener niebla. Funcionan bien con texturas repetitivas como las de nubes. Una curvatura ligera aportará riqueza a una escena grande y una muy pronunciada será más adecuada para una escena más pequeña donde sólo se vea pedazos de cielo de manera intermitente (y el efecto exagerado resulte atractivo).

Un ejemplo de material es el siguiente:

```
material SkyDomeCEDV
{
[...]
      texture_unit
      {
        texture clouds.jpg
        scroll_anim 0.15 0
      }
[...]
}
```

El resto del mismo sería idéntico al ejemplo anterior. Con `scroll_anim` se configura un desplazamiento fijo de la textura, en este caso sólo en el eje X.

La llamada para configurar una *skydome* en nuestra escena en un poco más compleja que para una *skybox*.

```
1 sceneManager->setSkyDome(true, "SkyDomeCEDV", 10, 8, 5000);
```

Donde el tercer parámetro es la curvatura (funciona bien para valores entre 2 y 65), y el cuarto es el número de veces que se repite la textura en la cúpula.

Ejemplo de *SkyPlane*

Un *skyplane* es a priori la más simple de las representaciones. Como su nombre indica está basado en un plano, y la creación de uno es necesaria. Aun así también es posible aplicarle algo de curvatura, siendo más adecuado para escenas con niebla que una cúpula. El material podría ser el mismo que antes, incluso sin utilizar animación. Se ve un ejemplo del uso del mismo en el ejemplo anterior del terreno.

Se propone crear una escena con cada uno de las tres técnicas anteriores. Si fuera posible, utilice el ejemplo de terreno anterior.

14.5.3. LOD : Materiales y Modelos

OGRE da soporte a la representación con diferentes niveles de detalle tanto en materiales como en modelos.

Materiales

En los materiales no sólo se remite a las texturas sino a todas las propiedades editables dentro del bloque `technique`. Para configurar un material con soporte para LOD lo primero es configurar la estrategia (`lod_strategy`) que se va a utilizar de las dos disponibles:

- **`Distance`** - Basado en la distancia desde la cámara hasta la representación del material. Se mide en unidades del mundo.
- **`PixelCount`** - Basado en el número de píxeles del material que se dibujan en la pantalla para esa instancia.

Tras esto, lo siguiente es determinar los valores en los que cambiará el nivel de detalle para dicha estrategia. Se usará para tal labor la palabra reservada `lod_values` seguida de dichos valores. Es importante destacar que tendrán que existir al menos tantos bloques de `technique` como valores, ya que cada uno de ellos estará relacionado con el otro respectivamente. Cada uno de los bloques `technique` debe tener asociado un índice para el nivel de detalle (`lod_index`). Si no aparece, el índice será el 0 que equivale al mayor nivel de detalle, opuestamente a 65535 que corresponde con el menor. Lo normal es sólo tener algunos niveles configurados y probablemente jamás se llegue a una cifra tan grande. Aun así, es importante no dejar grandes saltos y hacer un ajuste más o menos óptimo basado en las pruebas de visualización de la escena. Es posible que varias técnicas tengan el mismo índice de nivel de detalle, siendo OGRE el que elija cuál es la mejor de ellas según el sistema en el que se esté ejecutando.

Un ejemplo de material con varios niveles de detalle:

```
material LOD_CEDV
{
  lod_values 200 600
  lod_strategy Distance

  technique originalD {
    lod_index 0
    [...]
  }
  technique mediumD {
    lod_index 1
    [...]
  }

  technique lowD {
    lod_index 2
    [...]
  }

  technique shaders {
    lod_index 0
    lod_index 1
    lod_index 2
    [...]
  }
}
```

Nótese que la técnica `shaders` afecta a todos los niveles de detalle.

Como ejercicio se propone construir una escena que contenga 1000 objetos con un material con tres niveles de detalle diferente. Siendo el nivel 0 bastante complejo y el nivel 3 muy simple. Muestre los fotogramas por segundo y justifique el uso de esta técnica. Se sugiere mostrar los objetos desde las diferentes distancias configuradas y habilitar o deshabilitar los niveles de detalle de baja calidad.

Modelos

OGRE proporciona tres formas de utilizar LOD para mallas. La primera es una donde modelos con menos vértices se generan de forma automática, usando `progressiveMesh` internamente. Para utilizar la generación automática será necesario que la malla que se cargue no tenga ya de por sí LOD incluido (por ejemplo, la cabeza de ogro de las demos ya contiene esta información y no se podrá aplicar). Una vez que la entidad está creada, se recuperará la instancia de la malla contenida dentro.

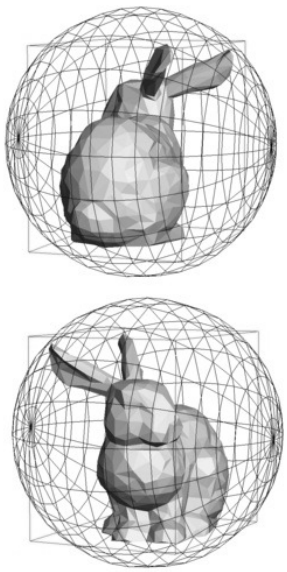


Figura 14.17: Ejemplo de *bounding-sphere*(MATH IMAGES PROJECT)

> ¿Podría analizar los niveles de detalle incluidos en la cabeza de ogro de las demos de OGRE? Use `OgreXMLConverter` para obtener una versión legible si es necesario.

Listado 14.2: Generación de niveles de detalle para una malla

```
1  Entity *entidad = sceneManager->createEntity("objeto", "objeto.mesh
       ");
2
3  Ogre::MeshPtr mesh = entidad->getMesh();
4
5  Ogre::Mesh::LodDistanceList lodDvec;
6
7  lodDvec.push_back(50);
8  lodDvec.push_back(100);
9  lodDvec.push_back(150);
10 lodDvec.push_back(200);
11
12 mesh->generateLodLevels(lodDList, ProgressiveMesh::
       VertexReductionQuota::VRQ_PROPORTIONAL, 0.1);
```

En la línea [3] del ejemplo anterior se crea una lista de distancias para el nivel de detalle. Esta lista no es más que un vector de reales que determina los valores que se utilizarán según la estrategia elegida para esa entidad. Por defecto corresponde con la distancia y en ese caso los valores corresponden a las raíces cuadradas de la distancia en la que se producen cambios. La estrategia se puede cambiar con `setLodStrategy()` usando como parámetro un objeto de tipo base `lodStrategy`, que será `DistanceLodStrategy` o `PixelCountLodStrategy`. Estas dos clases son *singleton*s, y se deberá obtener su instancia utilizando `getSingleton()`. La segunda de ellas corresponde con la política basada en el número de píxeles que se dibujan en la pantalla de una esfera que contiene al objeto (*bounding-sphere*). En la línea [12] se llama a la función que genera los diferentes niveles a partir de la malla original. El segundo parámetro es corresponde con el método que se usará para simplificar la malla:

- **`VRQ_CONSTANT`** - Se elimina un número dijo de vértices cada iteración.
- **`VRQ_PROPORTIONAL`** - Se elimina una cantidad proporcional al número de vértices que queda en la malla.

El tercero corresponde con el valor asociado al método y en el caso de ser un porcentaje tendrá que estar acotado entre 0 y 1.

La segunda forma de añadir LOD a una entidad de OGRE es hacerlo de manera completamente manual. Será necesario crear diferentes mallas para el mismo objeto. Esto se puede llevar a cabo en un programa de edición 3D como Blender. Se empieza con el modelo original, y se va aplicando algún tipo de simplificación, guardando cada uno de los modelos obtenidos, que serán usados como mallas de diferente nivel de detalle para la entidad que represente a dicho modelo.

Listado 14.3: Generación de niveles de detalle para una malla

```
Ogre::MeshPtr m =
  Ogre::MeshManager::getSingleton().load("original.mesh",
   Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);

m->createManualLodLevel(250, "lod2.mesh");
m->createManualLodLevel(500, "lod3.mesh");
m->createManualLodLevel(750, "lod3.mesh");

Ogre::Entity* objeto = sceneManager->createEntity("objetoConLOD",
                                                  m->getName() );

Ogre::SceneNode* n = sceneManager->getRootSceneNode()->
    createChildSceneNode();
n->attachObject(objeto);
```

La tercera forma es que el archivo que se carga ya tenga la información de nivel de detalle. Esta forma es bastante habitual, ya que el programa `OgreXMLConverter` acepta los siguientes parámetros relacionados con en nivel de detalle cuando se convierte un XML en un archivo *mesh* de OGRE:

OgreXMLConverter

Esta aplicación puede convertir desde el formato XML a *mesh* y viceversa.

- **`-l <lodlevels>`** - El número de niveles de detalle.
- **`-v <lodvalue>`** - El valor asociado a la reducción de detalle.
- **`-s <lodstrategy>`** - La estrategia a usar. Puede ser *Distance* o *PixelCount*.
- **`-p <lodpercent>`** - Porcentaje de triángulos reducidos por iteración.
- **`-f <lodnumtrs>`** - Número fijo de vértices para quitar en cada iteración.

Cuando se cargue un archivo *mesh* creado de este modo, se utilizará de forma automática el número de niveles de detalle configurados.

Se propone la construcción de una escena con unos 1000 objetos utilizando cualquiera de las dos primeras formas programáticas de añadir LOD. Justifique mediante el número de *frames* por segundo el uso de este tipo de técnicas. Compare la velocidad usando y sin usar LOD.

14.6. Conclusiones

En los dos últimos capítulos se ha realizado una introducción a algunas de las técnicas utilizadas para aumentar el rendimiento de las representaciones en tiempo real. Sin algunas de ellas, sería del todo imposible llevar a cabo las mismas. Si bien es cierto que cualquier motor gráfico actual soluciona este problema, brindando al programador

las herramientas necesarias para pasar por alto su complejidad, se debería conocer al menos de forma aproximada en qué consisten las optimizaciones relacionadas.

Saber elegir la técnica correcta es importante, y conocer cuál se está utilizando también. De esto modo, será posible explicar el comportamiento del juego en determinadas escenas y saber qué acciones hay que llevar a cabo para mejorarla.

Por desgracia, OGRE está cambiando alguno de los gestores de escena y se está haciendo evidente en la transición de versión que está sucediendo en el momento de escribir esta documentación.

La forma de realizar optimizaciones de estos tipos ha cambiado con el tiempo, pasando de utilizar el ingenio para reducir las consultas usando hardware genérico, a usar algoritmos fuertemente apoyados en la GPU.

Sea como sea, la evolución de la representación en tiempo real no sólo pasa por esperar a que los fabricantes aceleren sus productos, o que aparezca un nuevo paradigma, sino que requiere de esfuerzos constantes para crear un código óptimo, usando los algoritmos correctos y eligiendo las estructuras de datos más adecuadas para cada caso.

Bibliografía

[AMH08] E. Akenine-Möller, T. Haines and N. Hoffman. *Real-Time Rendering*. AK Peters, Ltd., 2008.

[AMHH08] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering*. AK Peters, 3^{rd} edition, 2008.

[CLRS09] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. 2009.

[Dal04] D.S.C. Dalmau. *Core techniques and algorithms in game programming*. New Riders Pub, 2004.

[Dal06] Chris Dallaire. Binary triangle trees for terrain tile index buffer generation. 2006.

[DB00] W.H. De Boer. Fast terrain rendering using geometrical mipmapping. *Unpublished paper, available at http://www. flipcode. com/articles/article geomipmaps. pdf*, 2000.

[DDSD03] X. Décoret, F. Durand, F.X. Sillion, and J. Dorsey. Billboard clouds for extreme model simplification. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 689–696. ACM, 2003.

[DWS+97] M. Duchaineau, M. Wolinsky, D.E. Sigeti, M.C. Miller, C. Aldrich, and M.B. Mineev-Weinstein. Roaming terrain: real-time optimally adapting meshes. In *Visualization'97., Proceedings*, pages 81–88. IEEE, 1997.

[Ebe05] D.H. Eberly. *3D Game Engine Architecture*. Morgan Kauffmann, 2005.

[Eri05] C. Ericson. *Real-time collision detection*, volume 1. Morgan Kaufmann, 2005.

[Fer04] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[FK03] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Gro12] Khronos Group. *COLLADA Format Specification*. http://www.khronos.org/collada/, 2012.

[Hop98] H. Hoppe. Efficient implementation of progressive meshes. *Computers & Graphics*, 22(1):27–36, 1998.

[Ibe12] IberOgre. *Formato OGRE 3D*. http://osl2.uca.es/iberogre/index.php/Página_Principal, 2012.

[Inf02] InfiniteCode. *Quadtree Demo with source code*. http://www.infinitecode.com/?view_post=23, 2002.

[Jun06] G. Junker. *Pro Ogre 3D Programming*. Apress, 2006.

[Ker10] F. Kerger. *Ogre 3D 1.7 Beginner's Guide*. Packt Publishing, 2010.

[LG95] D. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 105–ff. ACM, 1995.

[McS03] M. McShaffry. *Game coding complete*. "Paraglyph Publishing, 2003.

[Noc12] Assembla NocturnaIR. *El Formato Collada*. http://www.assembla.com/wiki/show/reubencorp/El_formato_COLLADA, 2012.

[NVi01] ATI NVidia. Arb occlusion query, 2001.

[Pip02] E. Pipho. *Focus on 3D models*, volume 1. Course Technology, 2002.

[Ree83] William T. Reeves. Particle systems - a technique for modelling a class of fuzzy objects. *ACM Transactions on Graphics*, 2:91–108, 1983.

[Shr09] D. Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1 (7th Edition)*. Addison-Wesley, 2009.

[SM09] P. Shirley and S. Marschner. *Fundamentals of Computer Graphics*. AK Peters, 3^{rd} edition, 2009.

[TPP08] T. Theoharis, G. Papaioannou, and N. Platis. *Graphics and Visualization: Principles & Algorithms*. AK Peters, 2008.

[TS91] S.J. Teller and C.H. Séquin. Visibility preprocessing for interactive walkthroughs. In *ACM SIGGRAPH Computer Graphics*, volume 25, pages 61–70. ACM, 1991.

[Ulr02] T. Ulrich. Rendering massive terrains using chunked level of detail control. *SIGGRAPH Course Notes*, 3(5), 2002.

[WN04] Wang and Niniane. Let there be clouds! *Game Developer Magazine*, 11:34–39, 2004.

[ZMHHI97] H. Zhang, D. Manocha, T. Hudson, and K.E. Hoff III. Visibility culling using hierarchical occlusion maps. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 77–88. ACM Press/Addison-Wesley Publishing Co., 1997.

Este libro fue maquetado mediante el sistema de composición de textos LaTeX utilizando software del proyecto GNU.

Ciudad Real, a 10 de Julio de 2012.