Videojuegos 3D

Jesús Alonso Alonso

P07/B0053/02687

UOC
www.uoc.edu

Índice

Introducción

En este módulo trataremos los aspectos involucrados en la realización de videojuegos 3D. En primer lugar, veremos las diferencias existentes entre las versiones de juegos 2D y 3D, el concepto y las características de un *game engine*, así como un número considerable de ejemplos. Continuaremos con un nuevo apartado de física de mayor complejidad y veremos dos ejemplos de motores físicos. Posteriormente, trataremos el punto más denso del módulo, la programación gráfica, por lo que veremos las diferentes tecnologías ligadas al desarrollo de videojuegos de carácter profesional. Con todo ello, seremos capaces de implementar un juego de complejidad considerable en su versión 3D.

La tecnología que utilizaremos a tal efecto será idéntica que en el módulo anterior, esto es, Visual Studio .net como herramienta de desarrollo y las librerías GLUT y OpenGL.

Objetivos

Al finalizar este módulo, seremos capaces de:

1. Entender las diferencias claves entre videojuegos 2D y 3D.

2. Conocer los aspectos involucrados en un *game engine* y las diferentes soluciones.

3. Entender las diferentes técnicas para la detección de colisiones en mundos 3D y el conocimiento de algunos motores físicos.

4. Implementar un videojuego 3D partiendo desde cero.

5. Conocer la API gráfica OpenGL con mayor profundidad.

6. Entender las diferentes técnicas gráficas asociadas al mundo de los videojuegos como navegación, interacción, visibilidad, LOD, *shaders*, etc.

1. Videojuegos 3D

Para quien programa, el concepto de videojuego 3D puede llegar a suscitar gran dificultad, o complejidad añadida, a lo que sería un juego en dos dimensiones. Evidentemente esto es así, y sin embargo, pueden llegar a tener mucho en común, pues dos versiones de un juego en 2D y 3D pueden llegar a compartir mucho código.

1.1. Videojuegos 2D frente a 3D

Con el objetivo de identificar cuáles van a ser las diferencias fundamentales para afrontar un videojuego 3D, veremos unas cuantas ilustraciones de juegos en ambas versiones.

De izquierda a derecha, Mario BROS y Super Mario BROS 64

En estas dos ilustraciones tenemos a un mismo protagonista, Mario, en un juego de plataformas pero en diferentes versiones. La primera diferencia es gráfica: en una tratamos con *sprites*, mientras que en la otra contamos con modelos 3D y algún que otro *sprite* para la interfaz gráfica, por ejemplo.

Para tratar el mundo, en la versión 2D nos bastará con un fichero de texto en el cual se defina nuestro *tile based map*, mientras que, en el segundo, resultará casi imprescindible un editor de niveles. Para el caso 3D, en el apartado físico, ahora necesitaremos un modelo de subdivisión espacial, y para las colisiones entre objetos o lanzamiento de objetos, tan sólo habrá que añadir la tercera coordenada a las ecuaciones vistas en el anterior módulo.

En cambio, en lo que concierne al apartado de lógica e inteligencia artificial, si bien en general un mejor aspecto visual suele venir acompañado de una mejora en otros aspectos, ambas versiones podrían ser exactamente iguales.

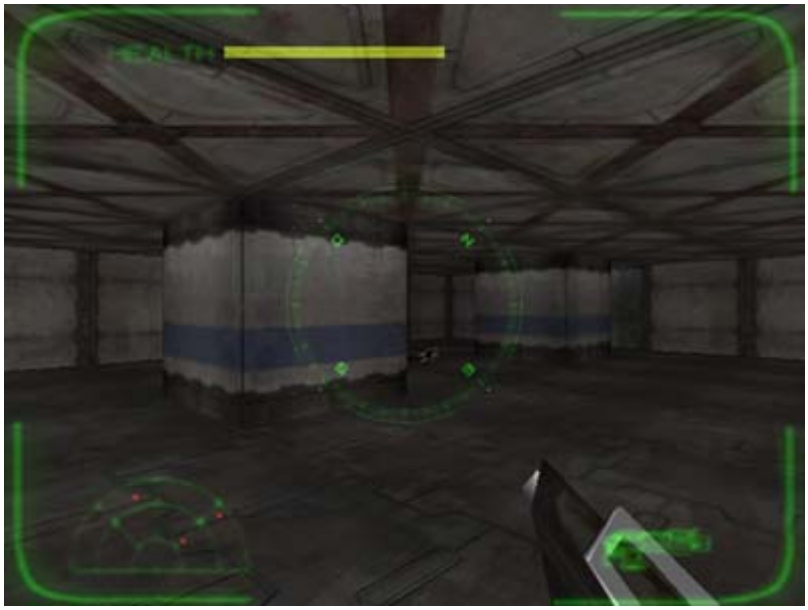
El casi imprescindible editor de niveles que hemos mencionado con anterioridad al hablar del tratamiento del escenario, viene dado por un motivo que puede resultar sorprendente. Es posible dar cabida al motor propio de los juegos 2D basado en *tiles* para desarrollar un videojuego en su versión en tres dimensiones. ¿Cómo?

La idea del *Tile Based Engine* es codificar todo un escenario siguiendo una estrategia que sea sencilla (es decir, que no suponga un coste de CPU considerable) y eficiente (o sea, recursos físicos de memoria y espacio de disco necesarios mínimos). Pues bien, podemos aplicar la misma idea cuando nuestro propósito es trabajar con un mundo 3D. Para ello, será necesario realizar la correspondencia *tile* (2D) - cubo (3D). Con esta idea, la implementación de un *Tile Based Engine* vista en el módulo anterior y unas nociones de geometría y mapeado de texturas para conseguir un mayor realismo visual, ya estaríamos en disposición de realizar nuestro primer videojuego 3D. Sin embargo, la aplicación de esta técnica supone la confección de mundos limitados a primitivas en formas de cubos (u otras).

Contenido complementario

Un *Tile Based Engine* puede servirnos para hacer un videojuego 3D. Para ello, deberemos tratar el concepto anterior de identificador de *tile* a un modelo 3D como puede ser un cubo. Diferentes ID se correlacionarán con cubos con diferentes texturas.

Ejemplo de videojuego 3D implementado mediante un *Tile Based Engine*



Realizado por Pere Alsina, ex alumno del curso Iniciación a los Videojuegos de la UPC.

En este caso, se aprecia que existen diferencias sustanciales, aunque también tienen lugar aspectos calcados.

En las siguientes imágenes, se ven estas diferencias entre dos videojuegos del género RTS (*Real Time Strategy*, estrategia en tiempo real): Warcraft II y Warcraft III de la compañía Blizzard.

De izquierda a derecha, Warcraft II y Warcraft III

La principal diferencia es, nuevamente, el aspecto gráfico. Imágenes 2D y modelos 3D, respectivamente, caracterizan cada uno de los casos. Sin embargo, tienen en común la interfaz gráfica, la lógica del juego y la inteligencia artificial de los personajes, tanto para los comportamientos basados en reglas y máquinas de estados (FSM, *Finite State Machina*), como para el paradigma del *pathfinding* (búsqueda de caminos).

De arriba a abajo y de izquierda a derecha, FIFA 95, PES, Yahoo Billar *online* y Foo Billard

En estas imágenes, que provienen de videojuegos deportivos, podemos apreciar cómo se repite el mismo patrón a la hora de estudiar las diferencias entre los videojuegos 2D y 3D. El aspecto gráfico los diferencia claramente y existen otros apartados como la física, la inteligencia artificial o la lógica que pueden llegar a compartirse en gran medida.

En la siguiente tabla, mostramos los principales aspectos involucrados en el desarrollo de un videojuego, especificando cómo los encontramos en cada una de las dos versiones posibles.

Aspectos	2D	3D
Cámara	Lateral, perpendicular	1.ª, 3.ª persona, vistas, *traveling*
Física	*Bound boxes*, *bounding circles*, *pixel perfect*, *tiles*	AABB, OBB, K-d trees, BSP, Octrees
Editor	*Tiles*	Niveles 3D
Diseño gráfico	*Sprites*	Modelos 3D y texturas
Computación gráfica	Parallax, efectos sencillos	Visibilidad, sistemas de partículas, iluminación, texturas, *shaders*
Animación	*Sprites*	*Key frame animation*, *Skeletal animation*
Cutscenes	Texto + imágenes	Película de animación 3D
Sonido	Clásico	Sonido 3D

1.2. *Game engines*

La complejidad gráfica añadida a la hora de tratar con escenarios 3D, hace necesaria la creación de un módulo gráfico independiente lo bastante potente para poder tratar toda la información que hay en él, así como su visualización en tiempo real. De esta necesidad surge el concepto de *Graphics Engine* (motor gráfico).

Si enriquecemos este módulo con elementos como física, redes o sonido, por ejemplo, tendremos lo que llamamos un *game engine* (motor de videojuego). De hecho, estos otros modelos también pueden presentarse de forma independiente dando lugar a un *Physics Engine* (motor físico) u otras capas o librerías como pueden ser una *Networking Layer* (capa de red) para el desarrollo de videojuegos en línea.

En este apartado, trataremos los módulos involucrados en la confección de un *game engine* y mostraremos algunas soluciones privadas y libres a modo de ejemplo.

1.2.1. Definición

Un *game engine* es el sistema operativo de un videojuego, el núcleo de un videojuego. Se encarga de dirigir y coordinar cada uno de los aspectos tecnológicos ligados a él. Capta eventos de entrada, computa física (colisiones, proyectiles), reproduce sonidos, simula IA, pinta, etc.

La dificultad que entraña realizar un motor de videojuego viene fundamentada por sus principales objetivos: eficiencia y venta. Por eso, casi todos los estudios desarrolladores han confeccionado su propio motor para una consecuente eficiencia de desarrollo en siguientes productos. Además, el motor puede ser vendido a otros estudios, obteniendo un beneficio económico añadido.

Al realizar un motor de videojuego, debemos dar respuesta a varios factores que definirán y limitarán su alcance. Los principales factores involucrados en el desarrollo de un motor son:

Funcionalidades

Las funcionalidades (*features*) que puede brindar un motor es una de las decisiones más importantes que habrá que tomar. Serán diferentes según el género del videojuego y deberá decidirse hasta qué nivel serán ofrecidas. Entre ellas cabrá especificar diferentes aspectos como, por ejemplo, los siguientes:

- Desde el punto de vista gráfico, qué tipo de **escenarios** podremos tratar: interiores (*indoors*), exteriores (*outdoors*) o genéricos; y con qué tipo de recursos y **formatos** trabajaremos para texturas o modelos 3D.

- Aspectos de **inteligencia artificial** brindados: *scripting, behaviours* (comportamientos), búsqueda de caminos, etc.

- Qué temas de **física** abarcaremos: estructuras de descomposición espacial como BSP o *octrees*, reacción a impactos, fuerzas elásticas e inelásticas, simulaciones, etc.

Ved también

En el apartado "Física" se verán con detalle cada una de las funcionalidades.

Como hemos dicho, un *game engine* puede plantearse para dar soporte a una solución en concreto o para acaparar un dominio mayor. A continuación, mostraremos dos videojuegos totalmente diferentes para detallar qué funcionalidades debería aportar cada uno de los motores sobre los que funcionarán.

De izquierda a derecha, *Need for speed* y *Devil may cry*

En el caso del videojuego *Ned for speed*, tenemos un videojuego del género motor y sus necesidades vienen dadas por su género. El apartado gráfico y físico deberá proporcionar una rápida respuesta para conseguir una fluida visualización en tiempo real. Otros aspectos, como la técnica del *motion blur,* ayudarán a tal finalidad y a ofrecer una mayor inmersión en cuanto a velocidad se refiere en el juego.

En el caso del videojuego *Devil may cry*, tenemos otros aspectos principales, como la colocación y el tratamiento de la cámara, un algoritmo más preciso para la detección de colisiones para interiores, múltiples y variados efectos especiales de explosiones y disparos o el tratamiento de las animaciones de los modelos y la interacción entre ellos.

Optimizaciones

Para cada tipo de juego tenemos diferentes optimizaciones posibles. Un motor de videojuego incluirá más o menos optimizaciones, según el abanico de tipos de juegos al que pueda dar soporte. Generalmente todos ellos contemplan una serie de mejoras (respuesta en tiempo) en los aspectos matemáticos que conciernen a la aritmética, geometría o cálculos numéricos. Asimismo, en todos ellos suele aparecer el gestor de memoria y el tratamiento de ficheros.

Sin embargo, pueden existir grandes diferencias cuando observamos todas y cada una de las funcionalidades que incorporan, así como la plataforma en cuestión a la que se dirigirá. Sin ir más lejos, pensemos en las consolas que tan sólo cuentan con dispositivos ópticos. La lectura de los datos es secuencial y por ello lenta. Por lo tanto, serán necesarias técnicas para el tratamiento de recursos específicos para ellas.

A continuación, estudiemos dos juegos bien diferentes para corroborar las optimizaciones de las que deberán estar provistos cada uno de ellos.

De izquierda a derecha: *Doom III* y *Comandos 3*

En el caso del juego de la serie de ID Software, *Doom III*, el factor más relevante es la inmersión que se obtiene con el realismo gráfico. Los aspectos de texturizado e iluminación que lo hacen posible se ofrecen en un alto grado de calidad, de forma transparente (juego fluido) al usuario, aunque sea necesaria una lógica potencia hardware.

El otro juego que podemos observar, *Comandos 2* de Pyro Studios, en cambio, necesita ser eficiente en otro aspecto. La calidad gráfica viene dada en forma de pre-procesos a partir de *renders* de escenarios previamente realizados. Por eso, este factor no es fundamental para la visualización en tiempo real: en este caso, el gran número de personajes que vagabundean o patrullan por el mundo, así como los que ordenamos nosotros mismos, los que pueden llevarse el mayor tanto por ciento de la CPU. El algoritmo utilizado para el *pathfinding* (búsqueda de caminos) estará ejecutándose, continuamente, para cada uno de los personajes que aparecen en el juego. Por ello, uno de los principales esfuerzos de optimización deberá enfocarse a este aspecto de la inteligencia artificial.

Hardware

La continua evolución del hardware es el factor más importante que nos define la vida útil de un motor de videojuego. Concretamente nos referimos a las tarjetas gráficas, al gran progreso del rendimiento de las GPU, que se vio iniciado por la Voodoo, la primera tarjeta gráfica aceleradora 3D desarrollada por 3DFX, en octubre del año 1996. A esta empresa le siguieron las que unos años más tarde se convertirían en sus más grandes competidores, NVIDIA y ATI y que, de hecho, abarcan el mercado actual.

En términos generales, podemos indicar que las ventajas gráficas son el elemento que más evoluciona del PC, duplican su potencia cada seis meses (lo cual supone un incremento tres veces mayor al de la CPU) y, de hecho, el papel de la GPU, su innovación en términos históricos, es tan sólo comparable con la llegada de la memoria *caché* (1980).

En esta línea, la especificación de los motores suele incluir el soporte y la versión a *píxel shaders*, *vertex shaders*, así como los nuevos *geometry shaders*. Además, en la mayoría de casos, muchos efectos implementados por hardware por medio de los *shaders*, son ofrecidos de una manera más automatizada al desarrollador o en forma de múltiples ejemplos con la propia SDK.

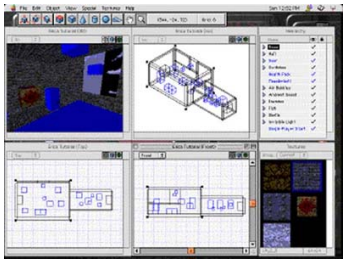
Software

Son muchos los procesos que es necesario repetir en la mayoría de desarrollos. Diseño de escenarios exteriores, interiores o modelos, son algunos ejemplos que constatan este hecho.

Ante esta necesidad, surge la motivación para la creación de un software íntimamente ligado a un motor de videojuego. De esta manera, la comunicación entre motor y recursos se ve claramente favorecida en términos de eficiencia de producción: tiempo y calidad.

Los paquetes o programas típicos que suelen estar ligados a un motor de videojuego son el editor de niveles y el editor de modelos. Otros que pueden aparecer son: editor de sistemas de partículas, editor de *shaders* y editor de terrenos.

- ***Level editor*** (editor de niveles): herramienta a partir de la cual se diseñan los escenarios que contendrá el juego. En éstos se define la configuración de:



Quake level editor

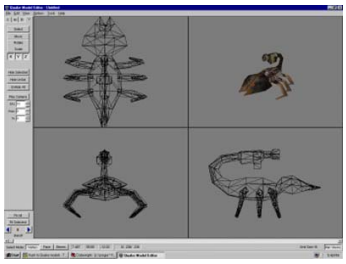
- El mundo físico estático con el que interaccionará el usuario: edificios, puentes, rocas, etc.

- Los elementos dinámicos del juego: personajes, animales, enemigos, vehículos, etc.

- Elementos decorativos: cielo, cascadas, lluvia, fuego, etc.

- Datos específicos: pueden hacer referencia a los diferentes puntos de las rutas de las patrullas de los elementos dinámicos o, puntos clave donde sucederá algo en especial como, por ejemplo, una teletransportación o la aparación de algun ente.

- **Editor de modelos**. Software a partir del cual se realiza la mayor parte de lo que serán todos los recursos gráficos 3D. Existen muchas herramientas genéricas en el mercado a tal efecto; Softimage, Maya, 3D Studio o Blender son algunos ejemplos. Sin embargo, en la mayoría de estudios es necesario establecer una comunicación más directa y nítida entre el recurso que generan y su inclusión en el videojuego. Con esta finalidad, y cuando un *pluggin* para la herramienta en cuestión no es suficiente, se suelen realizar soluciones específicas y privadas.

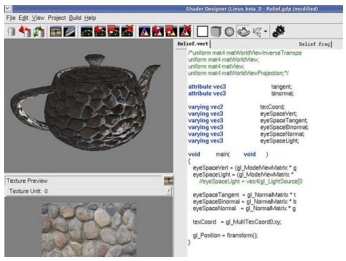


Quake Model Editor

- ***Particle system editor*** (editor de sistemas de partículas). Software a partir del cual se confecciona la lógica de los efectos especiales que se realizarán vía sistemas de partículas, así como el *tunning* de parámetros pertinentes como texturas, velocidades, rangos de colores, tamaños, tiempo de vida, fuerzas, emisores, etc.

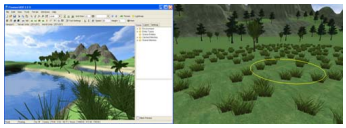
Ogre Particle System Editor

- ***Shader designer*** (editor de *shaders*). Herramienta para el diseño de materiales con *shaders*, haciendo uso de la potencia actual de las tarjetas gráficas. En tiempo real, y jugando por medio de parámetros, podemos apreciar qué tal resulta el diseño del *shader* que estamos realizando.



OpenGL Shader Designer

- ***Terrain editor World editor*** (editor de mundos y/o terrenos). Mediante esta aplicación, diseñaremos el escenario sobre el cual tendrá lugar el videojuego. Existen editores de terrenos y otras aplicaciones más complejas en las cuales puedes ubicar diferentes elementos naturales como árboles, vegetación, rocas o edificios e incluso personajes, dando lugar a un editor de mundos.



FreeWorld3D 2.0: Terrain and World Editor

Además de los paquetes que pueden venir acompañados de un *game engine*, u ofrecerse como aplicaciones independientes con soporte de importación y exportación a múltiples formatos de archivos, existen otras soluciones, como los últimos motores de videojuegos que han salido a la luz; por ejemplo, la segunda versión de CryEngine desarrollado por Crytek.

En estos motores de videojuegos, el software que los acompaña es una herramienta, de sorprendente complejidad, que incorpora las posibles configuraciones de cada uno de los aspectos involucrados que puede haber en un juego: escenarios, personajes, efectos especiales, etc. Además, en cualquier momento el editor nos permite entrar en modo juego, de manera que podemos navegar en tiempo real por el escenario recreado e interactuar con él.

CryEngine 2.0 editor

1.2.2. Arquitectura

En la arquitectura de un motor de videojuego, podemos distinguir los siguientes **módulos principales**:

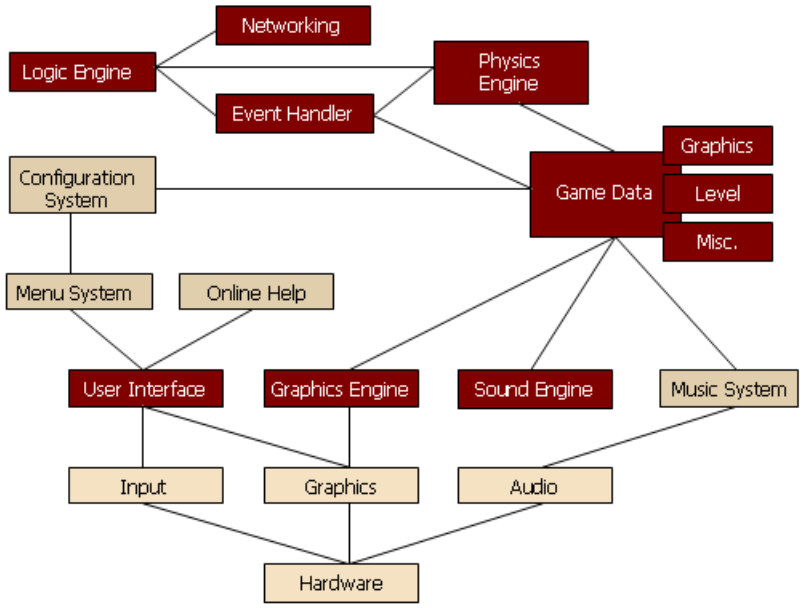
- *Graphics Engine*: motor gráfico
- *User interface*: interfaz de usuario
- *Event handler*: tratamiento de eventos
- *Data engine*: niveles, gráficos, etc.
- *Dynamics system*: física
- *Logic engine*: *scripting*, comportamientos
- *Sound engine*: sonido 3D y efectos especiales
- *Networking*: capa de red
- Hardware *abstraction layers*: capa para la abstracción del hardware

Además, podemos encontrar otros **módulos secundarios** como pueden ser:

- *Game configuration system*: sistema de configuración
- *Text system*: fuentes y texto
- *Menu system*: sistema de menús
- *Help system*: sistema de ayuda
- *Music system*: música

La estructura en la cual podemos ver cómo se relacionan los módulos citados se muestra en el siguiente esquema.

Arquitectura de un *game engine*



Los cuatro módulos que se encuentran en la parte inferior tratan la dependencia con el hardware; de color más oscuro podemos apreciar los módulos principales; y de color más claro, los secundarios.

A continuación, trataremos las principales características que tienen lugar en cada módulo por separado.

Graphics Engine

Graphics Engine es el módulo más importante del motor por su complejidad y la cantidad de puntos que trata, como, por ejemplo, los siguientes:

- Almacenamiento de objetos, la técnica utilizada para guardar y tratar la información geométrica (vértices): *vertex arrays, display lists, vertex buffer objects*, etc.
- Tipo de formatos para escenarios o modelos: 3ds, max, obj, mdl, md2, md3, ms3d, pud, pk3, pk4, raw, etc.
- Técnicas para recrear el cielo envolvente: SkyBox, SkyPlane, SkyDome.
- Técnicas para recrear terrenos: *heightmaps*.
- Técnicas de visibilidad: *portal culling*, BSP, *octree*, etc.
- Técnicas de iluminación: *flat shading, vertex shading, píxel shading, cel shading, lightmaps*, etc.
- Texturas: compresión, *mipmapping, multitexturing, bum mapping*, filtros anisotrópicos, etc.
- Efectos de niebla: *fading in distance* o *volumetric fog*.
- *Anti-aliasing*: eliminación de aristas dentadas, puntiagudas.
- Soporte a *pixel shaders*, *vertex shaders*, *geometry shaders*.
- Sistemas de partículas: efectos especiales.
- Modelado y animación de personajes: esqueletos, *blending animation*.
- Cinemática: inversa, directa.
- LODs (*levels of detail*): niveles de detalles para terrenos, modelos, vegetación, etc.

El algoritmo de visibilidad del Portal Culling es un punto que puede tratar un Graphics Engine para trabajar con escenarios *indoors*.

Physics Engine

Physics Engine es el módulo encargado de la detección de colisiones, la reacción a colisiones y la simulación del movimiento de aquellas entidades que pueden ser sometidas a fuerzas externas.

La detección de colisiones suele realizarse en dos niveles:

- En primer lugar, tenemos el mundo divido por descomposición espacial, mediante el cual ubicamos y mantenemos actualizadas las posiciones donde cada elemento está presente dentro de todo el escenario. Algoritmos que pueden ser utilizados con este objetivo son:

- BSP (*Binary Space Parttition*): subdivisión espacial mediante una estructura de datos de árbol binario.

- *Quadtree* / *Octree*: subdivisión espacial homogénea utilizando árboles de 4 u 8 hijos, según queramos realizar un tratamiento 2D (o mejor dicho en un plano en cuestión, típicamente y = 0) o 3D.

- K-d *tree*: subdivisión binaria del espacio en dos dimensiones, tomando, en cada nodo, un eje de división que alternamos entre x e y a medida que descendemos por el árbol.

- En un segundo término, tiene lugar la detección de colisiones confrontando las entidades en cuestión entre sí, haciendo uso de volúmenes envolventes. Las estructuras más comunes utilizadas a tal efecto son:

- AABB (*Axis Aligned Bounding Boxes*): cubos envolventes alineados a los ejes.

- OBB (*Oriented Bound Boxes*): cubos envolventes orientados en cualquier dirección.

- *Bounding Sphere*: mediante esferas envolventes.

- Otros objetos envolventes: soluciones específicas mediante, por ejemplo, elipsoides.

En lo que respecta a reacciones por colisiones o fuerzas externas, tenemos las leyes físicas ya expuestas simulando física newtoniana, velocidades, fricción, inercia, campos magnéticos, etc. Por otro lado, también tienen lugar aquellos objetos que mantienen nexos de unión con otros (*joins*) y que, al ser sometidos a cualquier física, responden provocando una reacción en cadena entre ellos.

Ved también

Podéis consultar las leyes físicas relacionadas con reacciones por colisiones o fuerzas externas en el módulo "Videojuegos 2D".

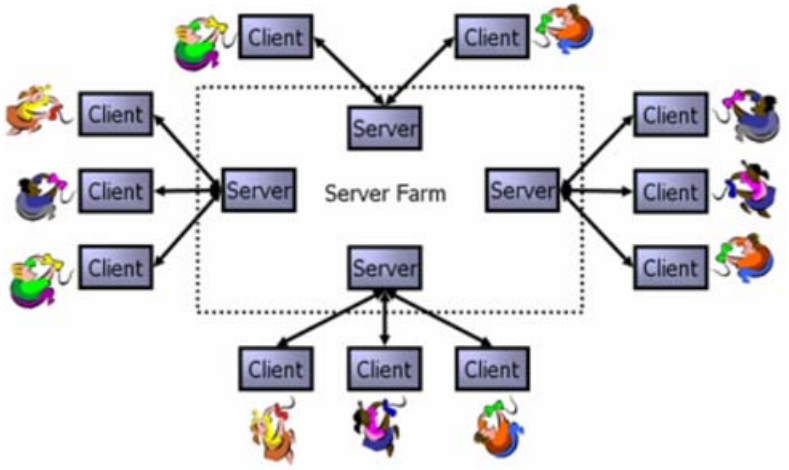
Captura de *Sumotori Dreams*, juego *freeware* de físicas

Sound System

Sound System es el módulo encargado de reproducir la música y los efectos de sonido de un videojuego. En este subsistema, se nos ofrece la reproducción de varios formatos, como mp3, wav, ogg, midi, etc. La técnica del Sonido 3D se brinda para dar una mayor inmersión, escuchar aquellos efectos que nos son cercanos con mayor intensidad y por el altavoz adecuado.

Networking

Hoy por hoy, la capa de red en los videojuegos tiene una marcada presencia. La mayoría de éstos ofrecen el servicio en línea para añadir mayor entretenimiento; incluso, para algunos, es un medio imprescindible. En este ámbito, según necesidades específicas, se tratan arquitecturas cliente-servidor, P2P, granjas, *grids* o híbridas.



Modelo Server Farm

Estos modelos dan respuestas diferentes para cada tipo de juego en línea, según los siguientes factores:

- Consistencia y sincronización: todos los jugadores en un mismo instante deben ver el mismo estado del mundo del juego.

- Baja latencia (*lag*): el retardo entre jugadores, que debe ser pequeño, ayuda a la consistencia.

- Escalabilidad: juego capaz de soportar un gran número de jugadores sin problemas, y escalable en tiempo real, conforme los usuarios entran y salen del juego.

- Rendimiento: suficientes recursos (por ejemplo, el ancho de banda) disponibles para que el juego corra suavemente.

- Fiabilidad: ante situaciones adversas, que el juego pueda continuar sin mayores problemas.

- Seguridad: registro/puntuaciones de jugadores, control de *cheating* (trampas) y *hackers*.

La información que se envía puede ser vía protocolo TCP (la información indispensable) o UDP (el resto de la información). Una guía general para cualquier tipo de juego en red debe contemplar los siguientes puntos:

- Sólo enviar lo necesario (el **qué**): estado del juego y cambios de estado, movimiento y acciones de personajes, colocación de objetos, condiciones de nivel, etc.

- Sólo enviar **cuando** es necesario: en lugar de hacer constar cada cambio independientemente, recolectamos los que ocurran en un pequeño intervalo de tiempo, enviando varios cambios de estado en un solo paquete.

- Sólo enviar **donde** sea necesario: cada cambio de estado no debe ser enviado a todos los jugadores, sino sólo a los que vayan a verse afectados. Por ejemplo, si los jugadores no se ven entre sí, incrementa complejidad, pero obtenemos grandes beneficios.

Script Engine

Script Engine se encarga de gestionar toda la información especificada del juego, ya sea referente a datos o a la lógica del juego (todo el sistema de reglas y comportamientos de entidades propios de la Inteligencia Artificial), que es almacenada externamente al código fuente de la aplicación.

La técnica del *scripting* aporta muchas ventajas al desarrollo de un videojuego. A continuación, enumeramos algunas de ellas:

- Los *Game designers* pueden escribir código con menor esfuerzo.
- Minimiza el trabajo de programación.
- No es necesario compilar ni montar: "Edit-test fast cycle".
- Los *scripts* pueden ser cambiados de manera mucho más fácil que el código.
- El *scripting* hace más fácil la definición de objetos y comportamientos, el tratamiento de los eventos de entrada, el diseño de la interfaz gráfica de usuario, el *testeo* y el *debug*.
- Proporciona un acceso a artistas y guionistas que pueden experimentar con nuevas ideas y *tunning*.

De hecho, gracias a todos estos factores, podemos decir que la técnica del *scripting* hace posible la existencia de los ***Mods***, concepto que proviene de la palabra del inglés "modifications". Los cambios y las nuevas funcionalidades se pueden realizar mediante herramientas de edición sencillas que han tenido una gran presencia, sobre todo en los videojuegos FPS (*first person shooters*) y RTS (*real-time strategy*).

Estas modificaciones suelen incluir nuevos ítem, armas, personajes, enemigos, modelos, modos, texturas, niveles y nuevas misiones. Desde un punto de vista empresarial, podemos decir que, cuanto más *moddable* (modificable) sea un juego, tanta más gente lo comprará y de mayor longevidad será su presencia en el mercado.

Batman Doom Mod

Existen diferentes niveles de *scripting*, como podemos apreciar en la siguiente lista:

- Level 0, *Hard-coded* AI: todo en el código fuente.
- Level 1, *Data Specified in Files*: estadísticas y configuraciones especificadas en ficheros aparte.

- Level 2, *Scripted Cutscenes*: secuencias de escenas no interactivas.

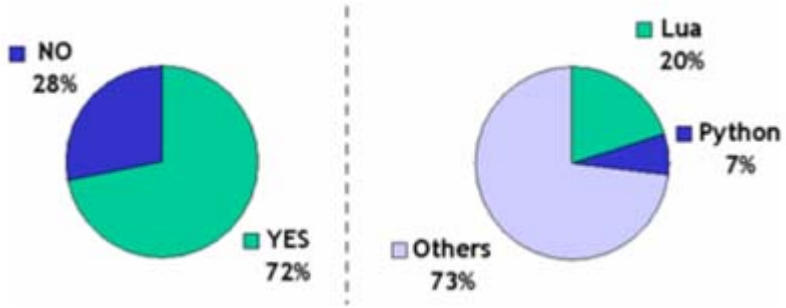
- Level 3, *Trigger System*: lógica poco pesada descrita en *scripts* utilizando la técnica de los *triggers* propia de las bases de datos.

- Level 4, *Heavy Script Logic*: se describe toda la lógica.

- Level 5, *Everything in Scripts*: todo implementado vía *scripts*.

De entre todos los lenguajes de *script* existentes en el mundo de la programación, el más utilizado es Lua. Su sencillez y potencia corroboran este hecho. A continuación, presentamos los resultados de una encuesta que, aunque es de hace unos años, aporta una visión del panorama. Cabe precisar que la opción "otros" corresponde, en su gran mayoría, a soluciones privadas de cada estudio.

Uso de los lenguajes de *script* en los videojuegos



Fuente: www.gamedev.net, septiembre 2003

Inteligencia Artificial

El aumento del rendimiento de las tarjetas gráficas de los últimos años ha hecho posible que se haya incrementado el porcentaje de potencia dedicado a los cálculos de comportamientos y demás aspectos de la IA. Precisamente, podemos dividir los aspectos ligados a la Inteligencia Artificial en dos grandes grupos:

- Las técnicas robustas. Engloba los siguientes aspectos:
  – FSM (*Finite Sate Machine*) o máquinas de estados
  – Sistemas de reglas
  – *Pathfinding*: búsqueda de la ruta óptima
  – Dinámicas de grupos
  – *Scripting*

El *pathfinding* en juegos RTS como Warcraft II, es imprescindible para el movimiento de tropas.

- Las técnicas experimentales. Engloba los siguientes aspectos:
  – Sistemas de aprendizaje
  – Algoritmos genéticos
  – Redes neuronales

1.2.3. Construir o comprar

Un *game engine* es un producto software muy complejo. Por tanto, como estudio de desarrollo, la reflexión que deberemos realizar acerca de si construir o comprar un motor deberá efectuarse con máxima precaución y detenimiento.

La compra de un motor debería responderse o justificarse respondiendo a los siguientes puntos:

- *Features*: qué funcionalidades y hasta qué nivel son ofrecidas, módulos que incorpora, tipos de formatos, etc.
- Nivel de soporte: documentación de la SDK, ejemplos, servicio de ayuda/consultoría.
- Vida: limitaciones, ampliable, modificable.
- Precio: suelen ser muy caros, centenares de miles de euros.

La construcción de un propio *game engine,* en cambio, puede venir motivada por los siguientes puntos:

- Funcionalidades a gusto: nosotros decidimos el alcance de nuestro motor y, aunque sea una solución muy específica, puede ser perfecto para nuestras pretensiones.

- Experiencia: la estructura y el diseño modular de un motor es todo un reto para un programador o un equipo de programadores.

- Reutilización y adaptación: para próximas versiones del juego u otros de semejante género o prestaciones gráficas, físicas u otros aspectos.

1.2.4. Ejemplos

La idea de motor de videojuego es un concepto que se ha tratado desde hace muchos años. En este contexto, cabe situar un ejemplo histórico muy conocido en su época: SCUMM de LucasFilms. Empezaremos la muestra de ejemplos de motores con este caso.

SCUMM son las siglas de "Script Creation Utility for Maniac Mansion". Se trata de un motor de videojuego de aventuras gráficas, desarrollado por Aric Wilmunder y Ron Gilbert, y que dio luz, a través del juego que se incluye en las propias siglas, Maniac Mansion en 1988.

Este motor estaba compuesto por una serie de subsistemas:

- SPUTM: el nombre real del *engine*.
- SCUMM: el lenguaje de *scripting* que dio fama al motor y que hizo que la gente lo conociera con este nombre.
- IMUSE: el sistema de control MIDI, que permitía música dinámica.
- SMUSH: formato de *compression* y reproductor de vídeo.
- INSANE: el sistema del gestor de eventos utilizado (versiones 7 y posteriores).
- MMUCAS: el gestor de memoria utilizado en The Curse of Monkey Island (versión 8).

Una de las ventajas de la confección de un motor de videojuego es la fácil adaptabilidad que puedes conseguir con él, al tratarse de una solución específica con un propósito bien marcado. En este caso, dedicado a implementar las funcionalidades que ha de servir una aventura gráfica.

SCUMM fue un motor que sufrió continuas mejoras y modificaciones, dando lugar a un gran número de versiones a medida que el estudio presentaba en el mercado sus aventuras gráficas.

Versiones SCUMM

A continuación, mostramos las sucesivas versiones que conformaron el histórico motor y en qué juegos se utilizaron. En total, el motor tuvo 8 versiones durante un periodo de 10 años.

v1:	Maniac Mansion (c64)

v1:	Zak McKracken and the Alien Mindbenders (C64)
v2:	Maniac Mansion
v2:	Zak McKracken and the Alien Mindbenders
v3:	Indiana Jones and the Last Crusade
v3:	Zak McKracken and the Alien Mindbenders (256 - FmTowns)
v3.0.22:	Indiana Jones and the Last Crusade (256)
v3.5.37:	Loom
v3.5.40:	Loom (alt. version)
v4.0.67:	The Secret of Monkey Island (EGA)
v5.0.19:	The Secret of Monkey Island (VGA Floppy)
v5.1.42:	LOOM (256 color CD version)
v5.2.2:	Monkey Island 2: LeChuck's revenge
v5.3.06:	The Secret of Monkey Island (VGA CD)
v5.5.2:	Indiana Jones 4 and the Fate of Atlantis (DEMO)
v5.5.20:	Indiana Jones 4 and the Fate of Atlantis
v6.3.0:	Sam & Max (DEMO)
v6.3.9:	Day Of The Tentacle (DEMO)
v6.4.2:	Day Of The Tentacle
v6.5.0:	Sam & Max
v7.3.5:	Full Thottle
v7.5.0:	The DIG
v8.1.0:	Curse of Monkey Island

En las siguientes ilustraciones, observamos el aspecto de cuatro de estos juegos que marcaron una época. Podemos apreciar cómo todas ellas obedecen a una misma estructura, cuerpo o esqueleto. En todas ellas aparecen una serie de acciones, un inventario, un sistema de navegación y de interacción muy semejante. La reutilización es la principal ganancia de un motor.

De izquierda a derecha y de arriba abajo: *Maniac Mansion* (1988), *The Secret of Monkey Island* (1990), *Indiana Jones and the Fate of Atlantis* (1992) y *Day of the Tentacle* (1993)

Actualmente, existe una gran variedad de *engines* comerciales y no comerciales. A éstos hay que añadir otro grupo de *middlewares*, soluciones que tratan temas como puede ser la portabilidad multiplataforma automática (*porting*), o un módulo en concreto como puede ser la física. Veamos unos cuantos ejemplos.

Ejemplos de *engines*

- ***Engines comerciales***: Cry Engine, Unreal Engine, Doom III, LithTech, Serious Sam, Torque Engine, Aurora.
- ***Engines no comerciales***: Irrlicht, Nebula Device, Fly3D, Delta3D, Crystal Space, Ogre.
- ***Middlewares comerciales***: Renderware, MathEngine, Havoc (física), WorldToolKit.

Entraremos un poco más en detalle en lo que respecta a los *open source game engines*. Para ello, daremos algunos datos de cada uno de los motores libres mencionados con anterioridad, describiremos alguna de sus principales características y mostraremos algunas imágenes de lo que se ha llegado a hacer con ellos.

Lectura recomendada

Recomendamos la lectura del siguiente enlace, en el que se confrontan dos motores de videojuego libres de considerable prestigio: **Ogre3D** y **Crystal Space**.

http://www.arcanoria.com/CS-Ogre.php

Irrlicht

Descripción	Funcionalidades principales
• "Open source high performance realtime 3D engine", desde el 2002 • Versión actual: 1.4 (Octubre 2007) • Autor: Nikolaus Gebhardt (ahora equipo) • Lenguaje: C++ • Plataforma: Linux, Mac, Windows • API gráfica: OpenGL 2.0, DirectX 8.1 y 9.0 • Web: http://irrlicht.sourceforge.net	• Fácil de entender y utilizar • *Vertex/Pixel shader* • Animación de personajes mediante esqueleto y morfología • Efectos de partículas • Sombras en tiempo real • GUI de fácil uso • Importa modelos: obj, 3ds, md3d, bsp, md2, etc. • Importa texturas: bmp, jpg, png, psd, tga, etc. • Detección de colisiones • Lectura directa de .zip • Parser XML • Editor 3D en tiempo real

Muestra de imágenes de proyectos utilizando Irrlicht

Nebula Device

Descripción	Funcionalidades principales
• "Open source realtime 3D game / visualization engine", desde 1998 • Versión actual: 2.0 (2002) con dependencias actualizadas (2007) • Autor: Radon Labs • Lenguaje: C++ • Plataforma: Windows • API gráfica: DirectX 9 • Web: http://nebuladevice.cubik.org/	• *Shaders* • *Scripting* • Física • Iluminación: *vertex*, *pixel*, *lightmap*, *gloss map* • Textura: *bump mapping*, *multitexturing*, *mip map* • Escenas: *octree*, *portal*, *culling* • Animación: *keyframe*, *skeletal*, *blending* • Modelos: obj, n3d, n3d2 • Terrenos: CLOD • *Networking* • Sonido • Efectos especiales: fuego, humo, destellos

Muestra de imágenes de proyectos utilizando Nebula Device 2

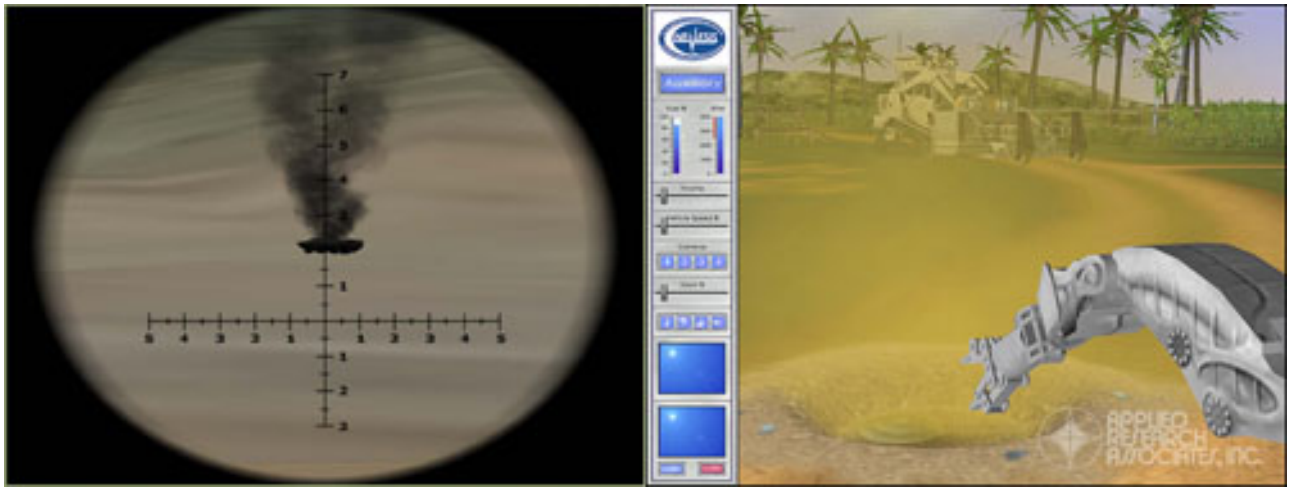
Fly3D

Descripción	Funcionalidades principales
• "Free and open source 3D game engine", desde el 2000 • Versión actual: 2.10 (enero 2003) • Autor: Fabio Policarpo • Origen didáctico: libro 3D Games • No se continúa desarrollando • Lenguaje C++ • Plataforma: Windows • API gráfica: OpenGL (*render*) y DirectX (*input*, *sound*) • Web: http://fabio.policarpo.nom.br/fly3d/	• Editores: *Shaders* y creación de mundos. • Física: Física básica, Detección de Colisiones, Cuerpos rígidos • Iluminación: Por vértice, por píxel, volumétrica • Sombras: Generación de sombras • Texturas: Básicas, Multitexturas, *bumpmapping* • Escenas: BSP y PVS • Animación: Por *keyframes*, por esqueletos • Efectos especiales: *Environment Mapping*, *Lens Flares*, *Billboarding*, Sistemas de partículas, *Depth of Field*, *Motion Blur*, Cielo, Agua • Terreno: CLOD • *Networking* • Sonido • Inteligencia artificial: A*

Muestra de imágenes de proyectos utilizando Fly3D

Delta3D

Descripción	Funcionalidades principales
• "Open source gaming & simulation engine backed by the U.S. Military", desde el 2004 (v0.1) • Versión actual: 1.5 (julio 2007) • Origen: US Navy, después fin docente • Plataforma: Windows, Linux • API gráfica: OpenGL 2.0 • Web: http://sourceforge.net/projects/delta3d	• *Shaders* • Animación de personajes • Física • Sonido • *Scripting* • Terrenos • Sistemas de partículas: humo, explosiones • *Networking* • Múltiples formatos de ficheros • Editor 3D de escenarios • Incluye los siguientes paquetes: Open Scene Graph, Open Dynamics Engine, Character Animation Library y OpenAL



Muestra de imágenes de proyectos utilizando Delta3D

Crystal Space

Descripción	Funcionalidades principales
• "Open source 3D software development kit", desde 1997 • Versión actual: 1.2 (octubre 2007) • Autor: Jorrit Tyberghein • Lenguaje: C++ • Plataforma: Windows, Unix, Mac • API gráfica: OpenGL • Web: www.crystalspace3d.org • Proyecto muy vivo a través de foro	• Animación: esqueletos • Modelos: MD3, MDL, CS • Reproducción Divx 4 • Iluminación: estatica, dinámica, pseudo, halos, sombras • Sonido • Terrenos • Sistemas de particulas (fuego, explosiones, lluvia, etc.) • Deteccion de colisiones y fisica • *Scripting* (java, *phyton*, *Perl*) • Sistema de *debug* • Muchas librerías adicionales

Muestra de imágenes de proyectos utilizando Crystal Space

Ogre3D

Descripción	Funcionalidades principales
• Object-Oriented Graphics Rendering Engine, 2001, actualmente v1.2 • Autor: Steve Streeting (comunidad) • Lenguaje: C++ • Plataforma: Linux, Mac, Windows • API gráfica: DirectX y OpenGL • Web: www.ogre3d.org • Sólo motor gráfico • Comunidad, diferentes estatus según contribución, tiempo y calidad aportaciones: Core Team member, MVP (Most Valued Person), Add-on Developer y User	• Escenas paginadas • *Shaders*: ensamblador, Cg, HLSL, GLSL • *Scripting* • LOD • Gestión de memoria y archivos (directorios, zips, pk3) • Sistemas de partículas, *billboarding* • Skybox, SkyPlane o SkyDome • Terrenos • Herramientas: exportación modelos 3d, visor, editor de partículas • Facilidad de inclusión y turiales para la utilización de otras librerías o módulos

Muestra de imágenes de proyectos utilizando Ogre3D

2. Física

2.1. Detección de colisiones

Existen dos técnicas principales para hacer eficiente el cálculo de la detección de colisiones que no son excluyentes y, en la mayoría de los casos, suelen combinarse:

- A partir de volúmenes envolventes.
- A partir de la división del espacio siguiendo una estructura determinada.

El objetivo es la detección de superposiciones de todos los elementos que tienen lugar en el escenario entre sí, y entre los diferentes elementos decorativos o interactivos del mismo; todo ello en un tiempo mínimo, para garantizar un juego fluido.

Esto también afecta a la computación gráfica, concretamente al apartado de la visibilidad. El motivo es simple, pues tan sólo es necesario comprobar las colisiones entre los objetos cercanos o visibles entre ellos y, a la hora de visualizar los elementos, sólo es necesario enviar a pintar aquellos objetos o modelos que son visibles al observador. Es decir, los que colisionan con su campo de visión.

Estos dos campos unidos por un mismo problema, al que hay que responder con la máxima brevedad posible, han hecho posible una ardua investigación iniciada hace muchos años. Por eso, se han realizado y de hecho continúan llevándose a cabo publicaciones al respecto y, hoy en día, las soluciones existentes son, como mínimo, notablemente satisfactorias. Incluso se ha llegado a adaptar este problema al hardware de la tarjeta gráfica, para que sea realizado en un coste en tiempo todavía menor.

2.1.1. Volúmenes envolventes

Comprobar la colisión entre dos objetos es muy costoso, sobre todo cuando están formados por miles de polígonos. Para minimizar este coste, previamente suele comprobarse la colisión entre sus volúmenes envolventes, de manera que, sólo si éstos colisionan, realizamos un test de colisión refinado entre objetos o respondemos de forma satisfactoria, con el consecuente error de precisión.

Un volumen envolvente está formado por un volumen simple que contiene uno o más objetos de naturaleza compleja. La idea consiste en utilizar un test de colisión entre volúmenes envolventes, menos costoso que el propio test de colisión entre objetos.

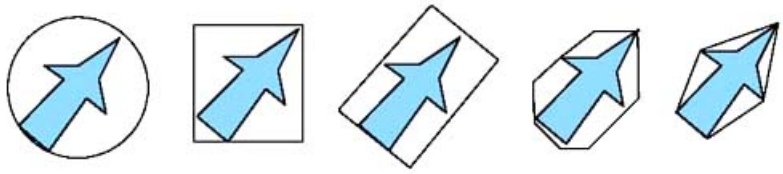
Para que sea útil, un volumen envolvente ha de cumplir las siguientes propiedades:

- Debe ajustarse al objeto tanto como sea posible.
- El cálculo de colisión entre volúmenes envolventes debe ser poco costoso en comparación con el test de colisión entre objetos.
- El cálculo del volumen envolvente debe ser poco costoso, sobre todo si éste debe recalcularse con cierta regularidad.
- El volumen envolvente debe poder rotarse y transformarse fácilmente.
- Debe representarse utilizando poca cantidad de memoria.

Los volúmenes envolventes más representativos son:

- Las esferas envolventes
- Las cajas envolventes alineadas con los ejes
- Las cajas envolventes orientadas
- Los k-DOPs

Tipos de volúmenes envolventes



De izquierda a derecha, esfera, caja envolvente alineada con ejes, caja envolvente orientada y k-Ddop y envolvente convexa

Esfera envolvente

Se le suele llamar **BS** (*Bounding sphere*). Una esfera envolvente se beneficia de un test de colisión poco costoso. Además, tiene la ventaja de ser un volumen invariante ante rotaciones y de necesitar poco espacio de almacenamiento.

El inconveniente de este tipo de volumen envolvente es que no se ajusta a determinados objetos, siendo difícil obtener la esfera que mejor se ajuste a un objeto dado.

Caja envolvente alineada con los ejes

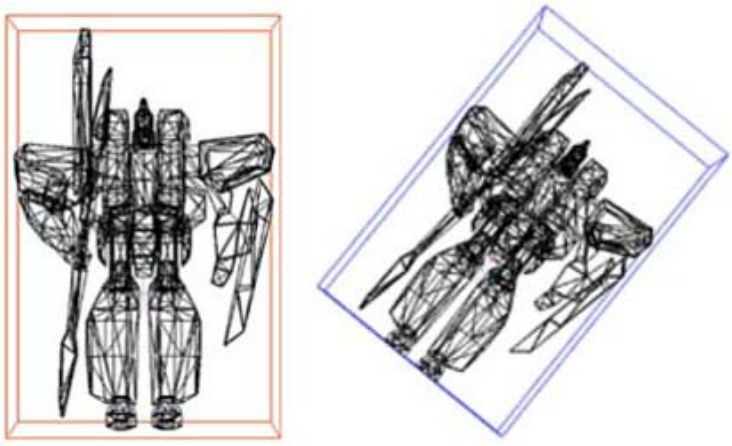
Normalmente conocida como **AABB** (*axis aligned bounding box*), es una caja rectangular cuyas caras están orientadas de manera que sus normales son paralelas a los ejes coordenados. La ventaja de este tipo de volumen es que el test de colisión es muy rápido, consistiendo en la comparación directa de los valores de coordenadas.

Sin embargo, no se produce un ajuste apropiado para muchos objetos. Además, actualizar este volumen envolvente suele ser más costoso que otros cuando los objetos rotan.

Caja envolvente orientada

También conocida como **OBB** (*oriented bounding box*), este tipo de volumen envolvente es representado por una caja rectangular con una orientación arbitraria, aunque suelen utilizarse determinadas direcciones prefijadas.

Al contrario que las AABB, este tipo de volumen envolvente tiene un test de colisión bastante costoso, basado en el teorema del eje separador. Sin embargo, su uso viene justificado por su mejor ajuste a los objetos y su rápida actualización en las rotaciones. Además, es necesario guardar mucha más información para representar este volumen, normalmente bajo la forma del centro de la caja, tres longitudes y una matriz de rotación.

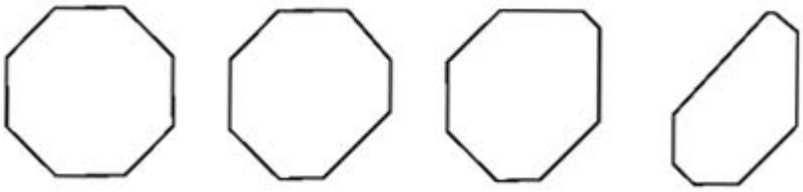


Ejemplo de AABB y ejemplo de OBB

Politopo de orientación discreta

Es un volumen envolvente basado en la intersección de "slabs". Un *slab* es la región infinita del espacio delimitada por dos planos paralelos. Este volumen conocido como **k-DOP** (*discrete orientation polytope*) es un politopo convexo definido por *slabs* cuyas normales pertenecen a un conjunto de direcciones predefinidas y comunes para todos los k-DOP. Las componentes de los vectores normales están restringidas al conjunto de valores {-1,0,1}, y éstos no están normalizados.

Ejemplo de k-DOP



Los distintos *slabs* son ajustados para adaptarse a un objeto, con lo que cambia la forma del volumen envolvente. Los planos que forman un *slab* siempre deben tener el mismo ángulo con la vertical que la configuración original.

Un AABB es un 6-DOP, pues está formado por 3 *slabs*. Evidentemente, mientras mayor sea el número de *slabs* de un k-DOP mejor se ajusta éste a los objetos, siendo necesario comprobar k/2 intervalos para determinar la colisión de volúmenes envolventes.

La actualización de un k-DOP es algo más costosa que un AABB debido a su mayor número de *slabs*, pero proporcional a k. El almacenamiento de un k-DOP depende también del número de *slabs,* pero es poco costoso debido, sobre todo, a que estos *slabs* están restringidos a ciertas configuraciones. Este volumen tiene la ventaja adicional de que siempre se puede modificar el tipo de ajuste a un objeto cambiando el número de *slabs* que se han de utilizar.

Otros tipos de volúmenes

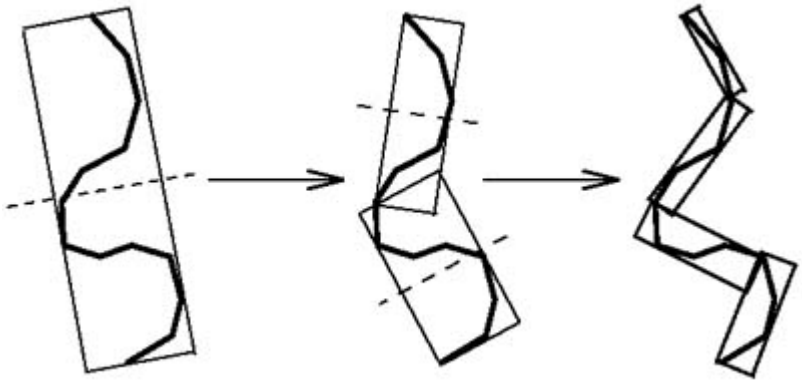
Existen muchos otros tipos de volúmenes envolventes que los aquí expuestos, como conos, cilindros, elipsoides, etc.

2.1.2. Jerarquías de volúmenes envolventes

Utilizar volúmenes envolventes puede disminuir el tiempo de cálculo de colisión entre objetos. Sin embargo, el número de parejas de objetos sobre el que realizar este test no varía, por lo que el tiempo de ejecución es el mismo aunque reducido por una constante.

Uniendo los volúmenes envolventes a una jerarquía, podemos reducir esta complejidad a logarítmica, construyendo para ello una jerarquía de volúmenes envolventes. Para n objetos podemos construir una jerarquía de volúmenes envolventes en forma de árbol de la siguiente manera.

En primer lugar, obtenemos los volúmenes envolventes de los objetos individuales y los colocamos como nodos hoja del árbol, que representa dicha jerarquía. Estos nodos son agrupados utilizando un nuevo volumen envolvente, de manera recursiva, hasta obtener un volumen envolvente que agrupe todos los objetos (representado por la raíz del árbol). Con esta jerarquía, se comprobaría la colisión entre hijos de un nodo sólo si se produce colisión entre los nodos padre.



Construcción de un OBBTree

En una jerarquía de volúmenes envolventes no es necesario que un volumen padre englobe los volúmenes hijos, sino que simplemente debe englobar los dos objetos que representan los volúmenes envolventes. Además, dos volúmenes envolventes del mismo nivel pueden tener partes comunes (intersecciones).

Una jerarquía de volúmenes envolventes puede utilizarse para organizar los objetos de una escena, o bien pueden utilizarse jerarquías de volúmenes envolventes individuales para cada uno de los objetos. Suelen usarse para objetos poliédricos no convexos, de manera que se dividen en componentes convexas. Debemos tener en cuenta que esta partición en componentes convexas no suele ser trivial.

Una jerarquía de volúmenes envolventes debe tener una serie de propiedades:

1) La geometría contenida en los nodos de cualquier sub-árbol debería estar próxima entre sí.

2) Cada nodo de la jerarquía debería tener el mínimo volumen posible.

3) Debemos considerar en primer lugar los nodos cercanos a la raíz de la jerarquía, pues eliminar uno de estos nodos es siempre mejor que hacerlo a mayor profundidad del árbol.

4) El solapamiento entre nodos del mismo nivel debería ser mínimo.

5) La jerarquía debería ser equilibrada en cuanto a su estructura y su contenido.

6) En aplicaciones en tiempo real, la determinación de colisión utilizando la jerarquía no debe ser mucho peor que el caso medio. Además, la jerarquía ha de generarse de forma automática, sin la intervención de un usuario.

Algunas de estas descomposiciones, mediante jerarquías de volúmenes envolventes, incluyen, entre otros: AABB-Trees, OBB-Trees, Box-Trees, Sphere-Trees, k-DOP trees, Shell-Trees, R-Trees y QuOSPO-Trees.

2.1.3. Volúmenes envolventes a distinto nivel de detalle

Podemos utilizar distintos volúmenes envolventes, cada uno con mayor nivel de ajuste o de detalle que el anterior y formar una sucesión de volúmenes envolventes. La proximidad entre objetos se obtiene de la misma forma para todos los niveles de este conjunto. Cuando no se puede determinar de forma exacta si dos objetos colisionan, se pasa al siguiente nivel de detalle.

Volúmenes envolventes a distinto nivel de detalle

Esta técnica implica un mayor coste de almacenamiento debido a la multiplicidad de volúmenes envolventes y un mayor número de test, cuando se produce colisión entre objetos. Además, la actualización de un mayor número de volúmenes es mucho más costosa. Sin embargo, esta variabilidad en el ajuste de los volúmenes envolventes permite realizar test de colisión entre volúmenes envolventes, en menor número, cuanto más ajustado al objeto sea el volumen (y más costoso), y mayor número de test entre volúmenes envolventes con menor ajuste (y menor coste).

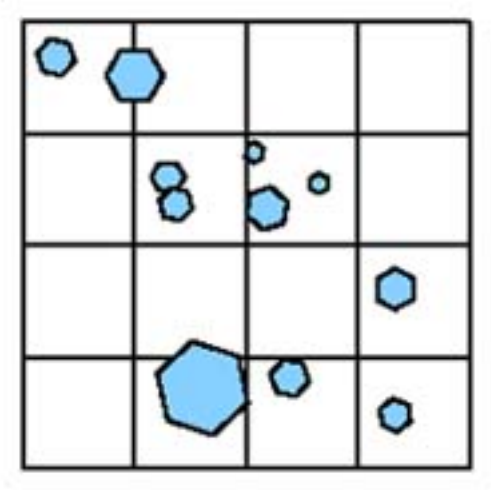
2.1.4. Descomposición del espacio

La descomposición espacial es una técnica utilizada para descartar pares de objetos. Se utiliza para determinar en qué zona del espacio se encuentra un objeto o parte del mismo (incluidas las características de un poliedro), y realizar un test de colisión entre objetos (o partes) sólo si ocupan la misma zona del espacio.

A continuación, veremos algunos métodos de descomposición espacial utilizados en detección de colisiones:

Rejilla de vóxeles

Una rejilla es una partición del espacio en celdas rectangulares y uniformes, es decir, todas las celdas tienen el mismo tamaño. Cada objeto se asocia con las celdas en las que se encuentra. Para que dos objetos colisionen deben ocupar una misma celda; sólo debe comprobarse la colisión entre objetos que ocupen la misma celda.



Rejilla utilizada en la detección de colisiones

Averiguar la celda a la que pertenece una coordenada consiste en dividir entre el tamaño de la celda, lo cual es simple y eficiente. Además, las celdas adyacentes a una dada se pueden localizar fácilmente. Esta simplicidad ha hecho que las rejillas se utilicen comúnmente como método de descomposición espacial.

En términos de eficiencia, uno de los aspectos más importantes de los métodos basados en rejillas consiste en elegir el tamaño apropiado de las celdas. Hay cuatro situaciones en las que el tamaño de las celdas puede ser un problema:

- La rejilla es muy fina. Si las celdas son muy pequeñas, el número de celdas que se han de actualizar será alto y, por tanto, costoso en términos de espacio y tiempo empleados.

- La rejilla es demasiado dispersa en relación con el tamaño de los objetos. Si los objetos son pequeños y las celdas grandes, habrá muchos objetos por celda. En esta situación, puede ser necesario realizar muchos pares de test de colisión entre los numerosos objetos clasificados en una celda.

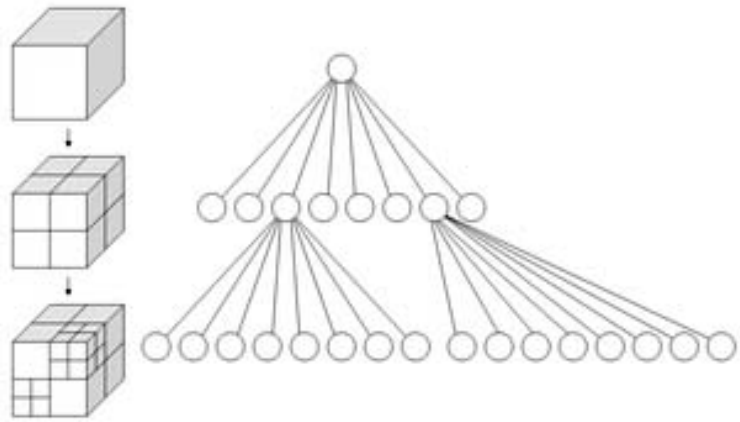
- La rejilla es demasiado dispersa en relación con la complejidad de los objetos. En este caso, el tamaño de las celdas es adecuado al tamaño de los objetos. Sin embargo, el objeto es demasiado complejo. Las celdas deben reducirse de tamaño, y los objetos descomponerse en piezas más pequeñas.

- La rejilla es demasiado fina y dispersa. Si los objetos son de tamaños muy dispares, las celdas pueden ser muy grandes para los objetos pequeños y viceversa.

Normalmente, el tamaño de celda de una rejilla se ajusta para que acomode el objeto de mayor tamaño rotando un ángulo arbitrario. Así se asegura que el número máximo de celdas que ocupa un objeto sea de cuatro en 2D y de ocho en 3D. De esta forma, se reduce el esfuerzo necesario para insertar o actualizar los objetos en la rejilla, así como el número de test de colisiones que realizar.

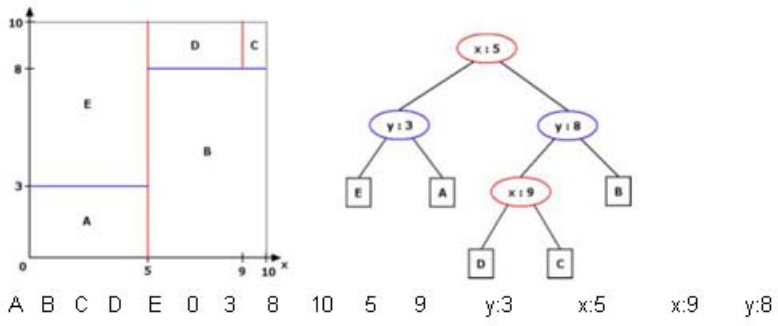
Octrees y *k-d trees*

Un ***octree*** es una partición jerárquica del espacio mediante *vóxeles* alineados con los ejes. Cada nodo padre se divide en ocho hijos. El nodo raíz suele representar el volumen del espacio completo que se ha de representar (la escena). Este volumen se divide en ocho sub-volúmenes, tomando la mitad del volumen en los ejes x, y, z. Estos ocho volúmenes se dividen de forma recursiva de la misma manera. El criterio de parada puede consistir en alcanzar una profundidad máxima en el árbol generado o un volumen mínimo por nodo.



Estructura de un *octree*

En un ***k-d tree*** cada región es dividida en dos partes por un plano alineado con los ejes. Un *k-d tree* puede realizar divisiones uniformes del espacio o variables (existen también *octrees* con divisiones variables). Las divisiones variables necesitan mayor almacenamiento para la estructura pero se adaptan mejor a los objetos.

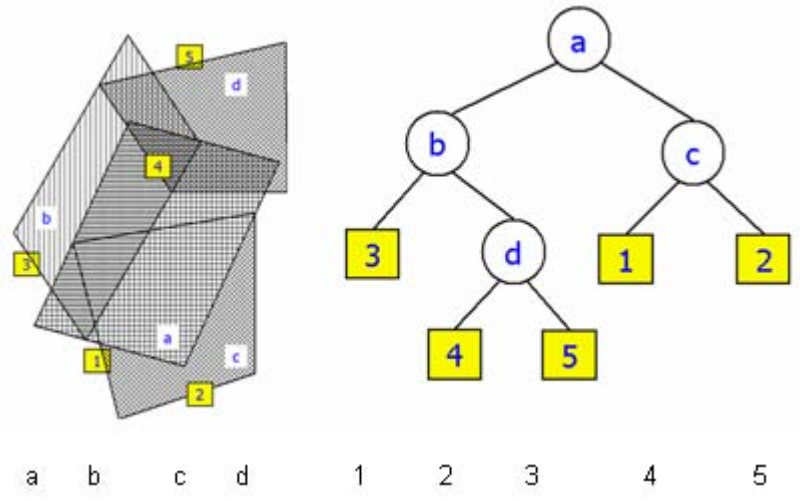


Ejemplo de *k-d tree*

El hecho de utilizar una subdivisión jerárquica del espacio permite la adaptación a las densidades locales del modelo. En regiones en las que hay un gran número de objetos, podemos dividir más que en regiones en las que hay menos objetos. En aplicaciones de detección de colisión entre objetos en movimiento, una celda puede subdividirse cuando entran nuevos objetos y agruparse celdas en una cuando el número de objetos disminuye.

BSP *trees*

Un árbol de partición binaria del espacio es una estructura jerárquica que subdivide el espacio en celdas convexas. Cada nodo interno del árbol divide la región convexa asociada con el nodo en dos regiones, utilizando para ello un plano con orientación y posición arbitrarias. Es análogo a un *k-d tree* variable, sin la restricción de que los planos estén orientados de modo ortogonal con los ejes.



Ejemplo de BSP: visualización de los planos de corte y de la estructura de árbol resultante

Un árbol BSP puede utilizarse como esquema de representación de objetos o como partición del espacio. Normalmente, se utiliza como partición del espacio y presenta el problema de cómo elegir los planos sobre los que dividir el espacio para obtener una descomposición óptima.

Lectura complementaria

Para abordar cómo elegir los planos sobre los que dividir el espacio para obtener una composición óptima, podéis consultar:

B. Naylor (1993). "Constructing good partitioning trees". *Graphics Inter*face (pág. 181-191). Disponible en http://www.graphicsinterface.org/pre1996/93-Naylor.pdf

2.2. Motores de física

En esta sección, estudiaremos dos motores de física: Newton Game Dynamics y Open Dynamics Engine (ODE). La elección de éstos viene determinada por el público objetivo al que nos dirigimos, en un contexto docente y a la potencia de ambos.

2.2.1. Newton Game Dynamics

Newton Game Dynamics es una solución integrada para la simulación, en tiempo real, de entornos físicos. Esta API proporciona la gestión de escenas, detección de colisiones, un comportamiento dinámico y, todo ello, de una manera rápida, estable y fácil de usar.

Actualmente, se encuentra en la versión 1.53 publicada en el mes de mayo del 2006; se puede obtener de manera gratuita, y cuenta con una peculiar licencia propia (ver fichero license.txt). Está disponible para las plataformas Windows, Linux y Mac.

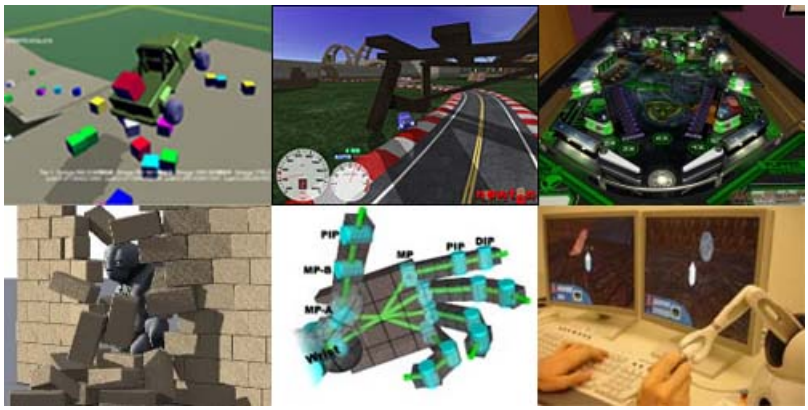
Como podemos apreciar en los diferentes proyectos que se han realizado con Newton, no se trata de una librería para un uso específico en el campo de los videojuegos. Está pensada para poder simular en tiempo real cualquier proyecto donde tenga lugar la física.

La integración de Newton en cualquier proyecto es más o menos sencilla y, en principio, basta con conocer los principios físicos para producir comportamientos realistas.

Web

Enlaces imprescindibles para conocer más acerca de Newton Game Dynamics:

- Web Oficial:
  http://www.newtondynamics.com/index.html
- Wiki no oficial:
  http://walaber.com/newton_wiki/



Muestras de proyectos de todo tipo realizados con Newton

2.2.2. Open Dynamics Engine (ODE)

A continuación, mostraremos un segundo ejemplo de motor de físicas de manera mucho más extensa, pues éste resulta de especial interés. El motivo es simple: la mayoría de proyectos que utilizan ODE a menudo son integrados con uno de los motores libres que mencionamos anteriormente, Ogre. Tanto es así que existe un considerable número de ejemplos y manuales de cómo llevar a cabo esta integración.

Descripción

ODE es un motor que permite simular físicas en cuerpos rígidos, y está especialmente indicado para detectar colisiones y fricción entre objetos, simular vehículos, paisajes y personajes. Es independiente de la plataforma y de la API gráfica que se use, y está implementado en C++.

ODE cuenta con dos licencias para escoger:

- BSD: permite hacer modificaciones de la librería y cambiar su licencia.
- GPL: obliga a incluir el código fuente en la distribución de la librería. No puedes cambiar la licencia.

La primera versión disponible de la librería (v0.01) data de abril del 2001. Su principal autor es Russell Smith, quien tiene el copyright, aunque detrás de él hay una importante comunidad que ayuda en el desarrollo de la librería. Actualmente, se encuentra en su versión 0.9 desde el pasado mes de septiembre del 2007.

Web

Enlaces imprescindibles para conocer más acerca de Open Dynamics Engine:

- Web Oficial:
  http://www.ode.org/
- Presentación:
  http://www.ode.org/slides/parc/dynamics.pdf
- Guía de usuario:

http://www.ode.org/ode-latest-userguide.pdf

- Wiki OgreODE:
  http://www.ogre3d.org/wiki/index.php/OgreODE

ODE está especializado en la simulación de físicas para cuerpos rígidos, que son creados mediante objetos unidos entre sí por varios tipos de uniones. Por lo tanto, podremos simular, por ejemplo, vehículos terrestres que tienen las ruedas unidas al chasis o animales terrestres pues tienen las extremidades unidas al cuerpo.

Está diseñado para la simulación en tiempo real, enfocado más en tener una buena respuesta en tiempo que en conseguir unos resultados exactos. De todas maneras, es un parámetro de entrada que podemos modificar a nuestro gusto.

Su sistema de colisiones trabaja con objetos de formas simples, como la esfera, el cubo, el cilindro, el plano, el rayo y mallas triangulares. Lo más común es utilizar esferas y cubos, ya que son los objetos con los cuales se pueden obtener test más rápidos.

Muestras de proyectos de todo tipo realizados con ODE

Desarrollo

Veamos cómo podemos llevar a cabo un proyecto utilizando ODE. Seguiremos los siguientes puntos:

- Estructura y conceptos básicos
- Construcción del mundo dinámico
- Creación de cuerpos y estados
- Creación y unión de *joins*
- Clases de geometría
  - El *loop*
  - Destrucción del mundo

1) Estructura y conceptos básicos

Los proyectos que utilizan ODE mantienen la siguiente **estructura**:

- Creación del mundo dinámico
- Creación de los cuerpos dentro del mundo dinámico
- Introducción de los estados de los cuerpos
- Creación de las diferentes *joins* en el mundo dinámico
- Unir las *joins* a los cuerpos
- Introducir las propiedades de las diferentes *joins*
- Creación de las colisiones del mundo y de las diferentes geometrías
- Creación de un grupo común para sostener las *joins* de contacto
- El *loop*
- Aplicación de las fuerzas que afectan a los cuerpos
- Enlazar las *joins* necesarias
- Llamar a la detección de colisiones
- Creación de una *join* de contacto por cada punto de colisión e introducirlas en el grupo de la conexión de contacto
- Cálculo de la simulación
- Eliminación de todas las *joins*
- Destrucción del mundo

Los **objetos principales** de la librería ODE son los siguientes:

- dWorld: Mundo dinámico
- dSpace: Referente a las colisiones del mundo
- dBody: Cuerpo rígido
- dGeom: Geometría de las colisiones
- dJoin: Conexiones
- dJoinGroup: Grupo de conexiones

De especial interés son los siguientes dos **conceptos** con los que trabaja ODE:

- **ERP**, *Error Reduction Parameter*. Tiene lugar cuando tenemos dos objetos concatenados y permite definir el nivel de exactitud de contacto entre éstos. Reduce el número de errores a la hora de calcular la dinámica que pueda haber entre sendos objetos. Su valor oscila en el rango [0..1] de tal manera que:

- Si tenemos un ERP=0 no se aplica ningún tipo de corrección.

- Un ERP=1 hace que la física sea demasiado imprecisa.

- Por lo general, se le suele dar un valor entre 0.1 y 0.8.

- **CFM**, *Constraint Force Mixing*. Este atributo hace que los cuerpos puedan ser más o menos duros, para la interacción con el mundo y los demás cuerpos que lo conforman.

- Un CFM=0 hará que el cuerpo sea duro, simulando lo que podría ser una roca, por ejemplo.

- Un CFM>0 otorgará cierta flexibilidad y nos puede servir para simular por ejemplo un objeto de plástico.

2) Creación del mundo dinámico

El objeto mundo (llamado dWorld) es un contenedor de los diferentes cuerpos y enlaces rígidos. Los cuerpos de diferentes mundos no pueden interactuar entre ellos. Algunos de los parámetros más importantes a la hora de crear el mundo son: la gravedad, la fuerza que actúa sobre los objetos y los factores ERP y CFM. Veamos el código oportuno que nos brinda esta librería para trabajar con estos conceptos.

Crear un nuevo **mundo vacío** y obtener su identificador.

```
dWorldID dWorldCreate();
```

Introducir la **gravedad** del mundo. Por ejemplo, para el caso de la gravedad terrestre, sería dWorldSetGravity (dWorldID, 0.0, 0.0, -9.5).

```
void dWorldSetGravity (dWorldID, dReal x, dReal y, dReal z);
```

Crear una **fuerza de impulso** en el mundo que afecte a los diferentes cuerpos.

```
void dWorldImpulseToForce(dWorldID, dReal stepsize,
                          dReal ix,dReal iy,dReal iz, dVector3 force);
```

Fijar el valor global de **ERP**, por defecto es 0.2.

```
void dWorldSetERP (dWorldID, dReal erp);
```

Fijar el valor global de **CFM**, su valor suele oscilar entre 10^{-9} y 1. Por defecto este es de 10^{-5}.

```
void dWorldSetCFM (dWorldID, dReal cfm);
```

ODE permite computar la **física** de dos maneras diferentes:

- De **forma precisa** pero consumiendo mucha memoria:

```
void dWorldStep (dWorldID, dReal stepsize);
```

- De forma **no tan precisa** pero mucho más rápida y gastando menos recursos:

```
void dWorldQuickStep (dWorldID, dReal stepsize);
```

3) Creación de cuerpos y estados

En este paso, se construyen todos los cuerpos y se indican sus características iniciales dentro del mundo.

Crear un **cuerpo** dentro de un mundo que cumplirá las reglas físicas que éste contenga. Por defecto, sus coordenadas en el espacio serán (0, 0, 0):

```
dBodyID dBodyCreate (dWorldID);
```

Fijar los **valores** de posición, rotación, velocidad lineal y angular del cuerpo.

```
void dBodySetPosition (dBodyID, dReal x, dReal y, dReal z);
void dBodySetRotation (dBodyID, const dMatrix3 R);
void dBodySetQuaternion (dBodyID, const dQuaternion q);
void dBodySetLinearVel (dBodyID, dReal x, dReal y, dReal z);
void dBodySetAngularVel (dBodyID, dReal x, dReal y, dReal z);
```

Fijar la **masa del cuerpo**.

```
void dBodySetMass (dBodyID, const dMass *mass);
```

Introducir las **fuerzas** que afectan al cuerpo.

```
void dBodySetForce (dBodyID b, dReal x, dReal y, dReal z);
void dBodySetTorque (dBodyID b, dReal x, dReal y, dReal z);
```

Añadir **diferentes tipos de fuerzas** que afectarán al cuerpo.

```
void dBodyAddForce (dBodyID, dReal fx, dReal fy, dReal fz);
void dBodyAddTorque (dBodyID, dReal fx, dReal fy, dReal fz);
void dBodyAddRelForce (dBodyID, dReal fx, dReal fy, dReal fz);
void dBodyAddRelTorque (dBodyID, dReal fx, dReal fy, dReal fz);
void dBodyAddForceAtPos (dBodyID, dReal fx, dReal fy, dReal fz,
                         dReal px, dReal py, dReal pz);
void dBodyAddForceAtRelPos (dBodyID, dReal fx, dReal fy, dReal fz,
                            dReal px, dReal py, dReal pz);
void dBodyAddRelForceAtPos (dBodyID, dReal fx, dReal fy, dReal fz, dReal px,
                            dReal py, dReal pz);
void dBodyAddRelForceAtRelPos (dBodyID, dReal fx, dReal fy, dReal fz, dReal px,
```

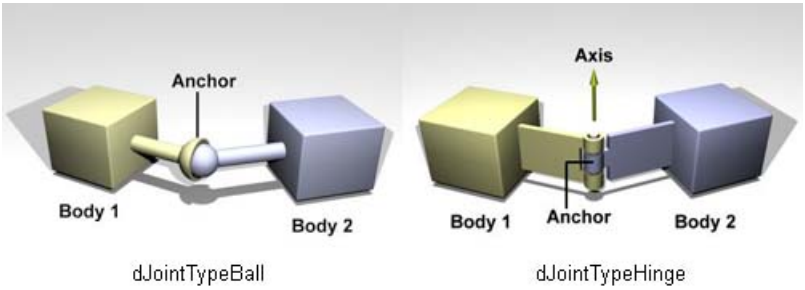
```
                                    dReal py, dReal pz);
```

Un cuerpo puede estar activado o desactivado. Mientras esté desactivado no le afecta la física del mundo. Si un objeto desactivado esta conectado con otro activado, automáticamente éste se activará. Por defecto todos los objetos al crearse están activados.

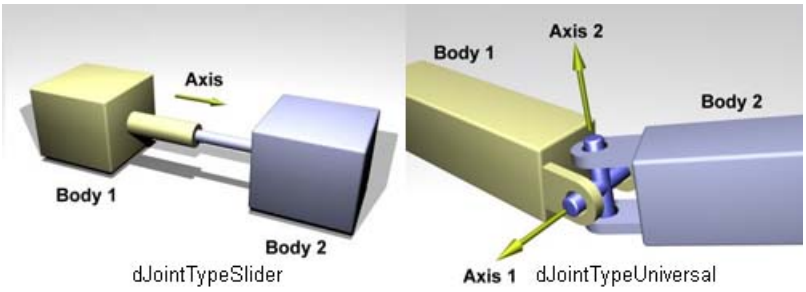
```
void dBodyEnable (dBodyID);
void dBodyDisable (dBodyID);
```

4) Creación y unión de *joins*

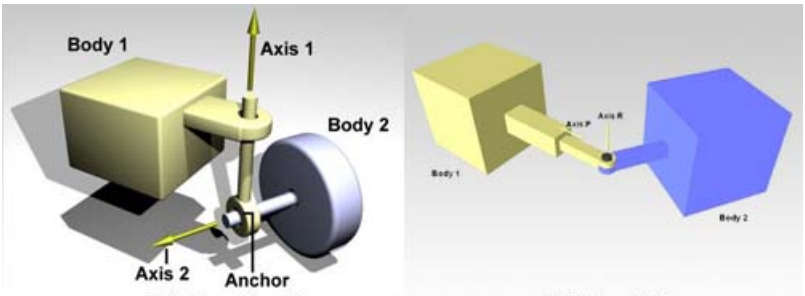
ODE nos proporciona diferentes tipos de *joins*. Las enumeramos a continuación mostrando, para cada una de ellas, una figura ilustrativa.



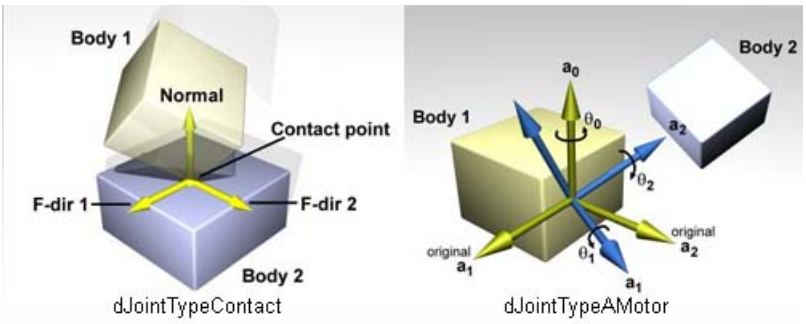
dJointTypeBall dJointTypeHinge



dJointTypeSlider dJointTypeUniversal



dJointTypeHinge2 dJointTypePR



A continuación, describimos las funciones que permiten crear una nueva *join* de un tipo determinado. Inicialmente no tendrá ningún efecto en la simulación; para ello será necesario enlazarla a algún cuerpo. La *join* del contacto será inicializada con la estructura del atributo dContact.

```
dJointID dJointCreateBall(dWorldID, dJointGroupID)
dJointID dJointCreateHinge(dWorldID, dJointGroupID)
dJointID dJointCreateSlider(dWorldID, dJointGroupID)
dJointID dJointCreateContact(dWorldID, dJointGroupID, const dContact *)
dJointID dJointCreateUniversal(dWorldID, dJointGroupID)
dJointID dJointCreateHinge2(dWorldID, dJointGroupID)
dJointID dJointCreateFixed(dWorldID, dJointGroupID)
dJointID dJointCreateAMotor(dWorldID, dJointGroupID)
dJointID dJointCreatePlane2d(dWorldID, dJointGroupID)
dJointID dJointCreatePR(dWorldID, dJointGroupID)
```

Para unir las *joins* a los cuerpos, asociamos una de las conexiones creadas anteriormente a dos cuerpos.

```
void dJointAttach (dJointID, dBodyID b1, dBodyID b2);
```

5) Clases de geometría

El motor de física ODE acepta las siguientes clases de geometría: Sphere, Box, Plane, Cylinder, Ray y Triangle Mesh.

También permite que el usuario defina sus propias clases geométricas. Para definir una geometría personalizada, debemos modificar la librería o usar la función dCreateGeomClass. En el caso que queramos crear nuestra propia geometría, deberemos definir su comportamiento redefiniendo las funciones de colisión propias para cada una de ellas.

No obstante, no es estrictamente necesario crear una clase de geometría propia para trabajar con objetos complicados, ya que, como hemos explicado anteriormente, es posible combinar varios objetos para representar demás complejos.

6) *Loop*

Durante el *loop*, ODE mira si algún objeto está en contacto con otro y calcula el comportamiento de éstos. Como podemos ver en la siguiente tabla, no todos los objetos reaccionan al chocar con otro.

	Ray	Plane	Sphere	Box	Capsule	Cylinder	Trimesh[3]	Convex	Heightfield
Ray	No[2]	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Plane	–	No[2]	Yes	Yes	Yes	Yes	Yes	Yes	No[2]
Sphere	–	–	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Box	–	–	–	Yes	Yes	Yes	Yes	No	Yes
Capsule	–	–	–	–	Yes	No	Yes	No	Yes
Cylinder	–	–	–	–	–	No	Yes	No	Yes
Trimesh[3]	–	–	–	–	–	–	Yes	No	Yes
Convex	–	–	–	–	–	–	–	Yes	Yes
Heightfield	–	–	–	–	–	–	–	–	No[2]

1. Esta funcionalidad tan sólo existe en versión SVN pendiente de revisar y no está incluida en la versión oficial.
2. No está planificado el hecho de ser implementada, pues no tiene mucho sentido en una simulación física.
3. No está habilitada por defecto, se necesita activar el *flag dTRIMESH_ENABLED*.

El método encargado de ver los cuerpos que chocan es nearCallback. Se recomienda sobrescribirlo para indicar qué tipo de reacción se quiere que tengan los cuerpos.

7) Destrucción del mundo

La siguiente función destruye un cuerpo, pero no sus conexiones con el resto de cuerpos, lo cual puede conllevar errores si los demás cuerpos asociados a éste no se eliminan.

```
void dBodyDestroy (dBodyID);
```

Destruir el mundo y todas sus conexiones.

```
void dWorldDestroy (dWorldID);
```

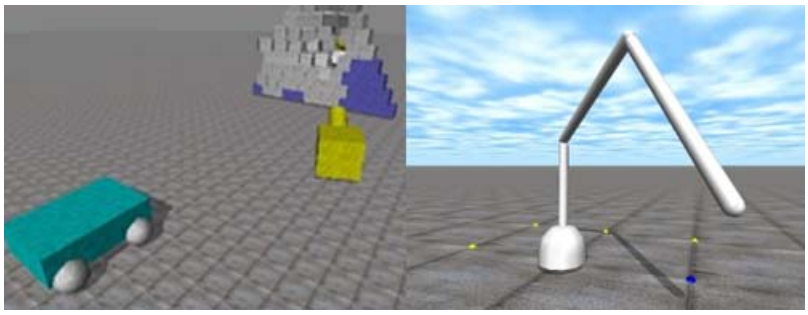
Lectura recomendada

Para más información se recomienda utilizar el manual de la página oficial:
http://opende.sourceforge.net/wiki/index.php/Manual

Ejemplos

Si nos descargamos el código fuente de la librería, nos encontraremos con una grata sorpresa. Desde la página web principal no es posible encontrar proyectos sencillos que utilicen ODE. Sin embargo, dentro del fichero donde se incluye el código fuente, hay varios proyectos sencillos que pueden sernos de gran ayuda.

Tan sólo es necesario ubicarse en la carpeta "tests". Para realizar la simulación gráfica, los proyectos se basan en una librería llamada Drawstuff con la cual, en principio, no es necesario tener nociones de programación gráfica.



Proyectos de ejemplos sencillos que utilizan ODE que podemos encontrar junto a la distribución del código fuente.

3. Programación gráfica 3D

3.1. Transferencia de datos

Hasta ahora hemos descrito rutinas sencillas de OpenGL mediante las cuales éramos capaces de recrear una escena sencilla. Por una parte, hemos definido objetos 3D a partir de primitivas, utilizando comandos de vértices y colores. Más tarde, comentamos la posibilidad de utilizar *display lists* para que el envío de la información fuera más rápido.

Todo esto lo hemos hecho sin diferenciar y dejar claro, explícitamente, que existen diferentes maneras de realizar la transferencia de datos a la tarjeta gráfica. Ahora que nos encontramos en un apartado de programación gráfica de mayor complejidad, es el momento adecuado para exponerlo.

La transferencia de datos es uno de los temas principales que preocupa a la programación de escenarios virtuales 3D. El motivo es simple, pues en la mayoría de los casos nos interesará una navegación en tiempo real y, por ello, el trabajo con toda la información deberá realizarse con especial atención. En el caso de los videojuegos este factor nos resulta imprescindible: gestionar grandes cantidades de triángulos de manera que los resultados se muestren de una manera fluida y continua. A pesar del incremento del ancho de banda de las tarjetas gráficas de los últimos años, actualmente, la transferencia de datos puede constituir un *bottleneck* (cuello de botella) para el *renderizado* de escenas.

Las diferentes posibilidades que se nos presentan son:

- Modo inmediato clásico
- *Display Lists*
- *Vertex Array* (VAR)

Otras estructuras de datos con las que las tarjetas gráficas pueden operar internamente son:

- *Vertex Buffer Object* (VBO)
- *Pixel Buffer Object* (PBO)
- *Frame Buffer Object* (FBO)

A continuación, analizaremos en detalle las propuestas presentadas.

3.1.1. Modo inmediato clásico

Este sistema es, por descontado, el más cómodo para el programador pero también el más ineficiente. Trabaja con vértices y primitivas y se identifica con el modelo de vista del cliente. Recordemos el aspecto de este modo con el siguiente código:

```
glBegin(GL_QUADS);
   glVertex3f(-1.0, 1.0, 0.0);
   glVertex3f( 1.0, 1.0, 0.0);
   glVertex3f( 1.0, -1.0, 0.0);
   glVertex3f(-1.0, -1.0, 0.0);
glEnd();
```

La estructura Begin/End es ineficiente, y para entender la lentitud que supone, expondremos una comparación con lo que es el tratamiento de ficheros que todos conocemos. Esto es, cuando estamos operando con archivos y realizamos operaciones de lectura y escritura.

Si este proceso lo realizamos *byte* a *byte*, la operación es mucho más lenta que si lo hacemos utilizando *buffers*. Pues precisamente radica en este hecho la corroboración de lo que acabamos de decir del modo inmediato. Cada operación de vértice que tenemos en el ejemplo anterior supone lo que, en sistema de ficheros, sería una llamada al sistema pero, en este caso, es con la tarjeta gráfica.

3.1.2. *Display Lists*

Este método es un mecanismo para encapsular comandos de pintado y que, siguiendo el ejemplo anterior, agrupa toda una serie de información realizando una sola transferencia a la tarjeta gráfica. El aspecto que tenía para el programador era el siguiente:

```
glNewList (identifier, GL_COMPILE);
   ...
glEndList ();
```

Como dijimos antes, es útil para describir objetos jerárquicamente y por lo general permitir optimizar escenarios mediante la filosofía: "Create once, use many".

3.1.3. *Vertex Arrays*

Las *display lists* son útiles para definir modelos estáticos. Si éstos sufren modificaciones, y quisiéramos seguir tratándolos de esta manera, sería necesario rehacer cada vez la lista. Ante la necesidad de trabajar de manera eficiente con objetos animados o dinámicos surgen los *vertex array*.

Este modo integra coordenadas de vértices, colores, normales y otros datos en grandes *arrays*. La segunda funcionalidad que nos brinda es la posibilidad de enviar toda la información o subconjuntos de ésta.

Cuantos más valores especificamos por vértice (posición, color, normal, etc.), más llamadas OpenGL son necesarias y, por tanto, necesitamos más tiempo. Utilizando VAR pasamos un puntero a OpenGL y decidimos qué vértices pintamos y cómo. A continuación, presentamos las funciones que nos permiten trabajar con VAR.

Tenemos un *array* para cada tipo de información que queremos tratar. Para **habilitar** y **deshabilitar** cada uno de éstos contamos con las siguientes funciones:

```
void glEnableClientState (GLenum cap)
void glDisableClientState (GLenum cap)
   cap: GL_VERTEX_ARRAY
        GL_NORMAL_ARRAY
        GL_COLOR_ARRAY
        GL_INDEX_ARRAY
        GL_TEXTURE_COORD_ARRAY
        GL_EDGE_FLAG_ARRAY
```

Para definir estos *arrays* contamos con el siguiente comando:

```
void glVertexPointer (GLint sizei, GLenum type, GLsizei stride, const GLvoid *pointer)
      glNormalPointer (GLint sizei, GLenum type, GLsizei stride, const GLvoid *pointer)
      glColorPointer (GLint sizei, GLenum type, GLsizei stride, const GLvoid *pointer)
      glIndexPointer(GLint sizei, GLenum type, GLsizei stride, const GLvoid *pointer)
      glTexCoordPointer (GLint sizei, GLenum type, GLsizei stride, const GLvoid *pointer)
      glEdgeFlagPointer(GLint sizei, GLenum type, GLsizei stride, const GLvoid *pointer)
```

Los parámetros tienen el siguiente significado:

- *sizei*: número de componentes
- *type*: GL_BYTE, GL_INT, GL_FLOAT
- *stride*: *offset* entre vértices consecutivos (tamaño del *vertex struct*)
- *pointer*: puntero a la primera coordenada del primer vértice del *array*

Para enviar a *renderizar* la información, tenemos diferentes posibilidades. Para enviar un elemento de un *array* se utiliza el siguiente comando. El parámetro corresponde al vértice i-ésimo.

```
void glArrayElement (GLint i)
```

Para enviar las primitivas que tenemos en los datos de un *array* procedemos de la siguiente manera:

```
void glDrawArrays (GLenum mode, GLint first, GLsizei count)
```

A continuación, exponemos el significado de los parámetros y el procedimiento de la acción mediante el siguiente código:

```
if (mode or count is invalid)
{
   generate appropiate error
}
else
{
   int i;
   glBegin (mode);
      for( i=0; i<count; i++)
      {
         glArrayElement (first + i);
      }
   glEnd();
}
```

La siguiente función nos permite enviar la información a partir de índices.

```
void glDrawElements (GLenum mode, GLsizei count, GLenum type, const GLvoid * indices)
```

Mostramos el significado de la función y de sus parámetros, de nuevo, mediante el siguiente código:

```
if (mode or count is invalid)
{
   generate appropiate error
{
else
{
   int i;
   glBegin (mode);
      for( i=0; i<count; i++)
      {
         glArrayElement (indices[i]);
      }
   glEnd();
}
```

3.1.4. *Buffer objects*

Como hemos introducido anteriormente, al grupo de los *buffer objects* pertenecen:

- *Vertex Buffer Object* (VBO)
- *Pixel Buffer Object* (PBO)
- *Frame Buffer Object* (FBO).

La característica principal que la diferencia de los anteriores es que, mediante estas estructuras, el volcado de la información a la tarjeta gráfica es implícito pues, en todo momento, estamos trabajando en ella.

Web

A continuación, se exponen los enlaces, donde podemos encontrar la especificación para cada uno de los siguientes *buffer objects*:

- ***Vertex Buffer Object*** (VBO):
  http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_buffer_object.txt
  http://www.spec.org/gpc/opc.static/vbo_whitepaper.html
- ***Pixel Buffer Object*** (PBO):
  http://oss.sgi.com/projects/ogl-sample/registry/EXT/pixel_buffer_object.txt
- ***Frame Buffer Object*** (FBO):
  http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt
  http://download.nvidia.com/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_FrameBuffer_Object.pdf

3.2. Iluminación

Para tratar el tema de la iluminación, primero haremos una pequeña introducción de los conceptos teóricos que involucra. Acto seguido, se entrará en detalle en cada uno de ellos y, paralelamente, veremos cómo OpenGL trata esta información. Para finalizar, mostraremos unos ejemplos prácticos.

La iluminación es un aspecto importante en la composición de la escena. Como en la realidad, la iluminación es un factor relevante que contribuye al resultado estético y a la calidad visual del trabajo. Por eso, puede ser un arte difícil de dominar. Los efectos de iluminación pueden contribuir en gran medida al humor y la respuesta emocional generada por la escena, algo que es bien conocido por fotógrafos y técnicos de iluminación teatral.

El modelo de iluminación de OpenGL se basa en sus componentes: ambiental, difusa y especular. Veamos en detalle cada una de ellas:

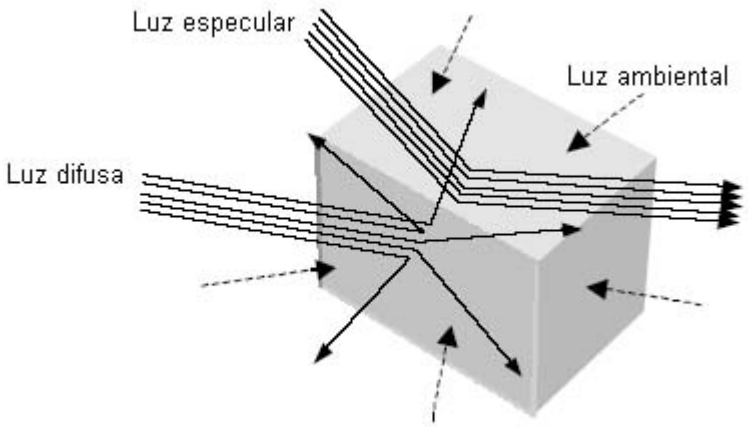
- La luz **ambiental** es aquella que no proviene de una dirección concreta, sino que incide sobre todas las partes del objeto de igual manera. Aunque realmente toda la luz proceda de algún foco, debido a la reflexión sobre los demás objetos que se encuentren en la escena, siempre hay alguna

componente de esa luz que incide sobre todas las partes del objeto, aunque no estén expuestas a ella directamente.

- La luz **difusa** es la que proviene de una dirección particular pero es reflejada en todas direcciones. Esta luz sólo afecta a aquellas partes del objeto en las que la luz incide. De hecho, estas partes se mostrarán más brillantes que las demás.

- La luz **especular** es la que procede de una dirección concreta y se refleja en una única dirección, de manera que produce brillos intensos en ciertas zonas. Es el caso, por ejemplo, de los objetos metálicos.

Gráficamente, esta información puede expresarse mediante la siguiente ilustración.



Cada foco de luz que incluyamos en la escena tiene tres componentes: luz ambiental, luz difusa y luz especular. La proporción de cada componente determina el tipo de luz.

Ejemplo

Un láser está casi totalmente formado por luz especular. Las componentes difusa y ambiental son mínimas y se deben a la presencia de polvo en el ambiente. Una bombilla, sin embargo, tiene una parte grande de luz ambiental y difusa, y una parte mínima de luz especular.

Cuando se define un foco de luz, no sólo es necesario definir las tres componentes de la luz, sino también el color de cada componente (RGB).

El aspecto final de los objetos diseñados no depende únicamente de la luz que reciban, sino también del **material** del que estén hechos. El material, además del color, posee propiedades de **reflectancia** de la luz, que marcan cómo se refleja cada componente de la luz. Así, un material que tenga alta reflectancia

para las componentes ambiental y difusa de la luz, pero baja para la especular, se iluminará principalmente de manera ambiental y difusa, por muy especular que sea la luz con el que lo iluminemos.

El cálculo del color final se realiza multiplicando los valores de cada componente de la luz por la reflectancia correspondiente a cada una de ellas. En el caso de las componentes difusa y especular intervienen, además, los ángulos de incidencia de la luz sobre cada superficie del objeto. Para obtener el valor final basta con sumar los valores obtenidos para cada color (si se sobrepasa el valor 1 se toma ese color como 1). Veamos un ejemplo práctico:

Ejemplo práctico de cálculo de color final

Luz	**Componente**	**Rojo**	**Verde**	**Azul**
	Ambiental	0.7	0.6	0.1
	Difusa	0.5	0.4	0.1
	Especular	1.0	0.3	0.0
Material	**Reflectancia**	**Rojo**	**Verde**	**Azul**
	Ambiental	0.4	0.1	0.8
	Difusa	1.0	0.2	0.7
	Especular	0.9	0.1	0.1
Valor final	**Componente**	**Rojo**	**Verde**	**Azul**
	Ambiental	0.28	0.06	0.08
	Difusa*	0.3	0.05	0.04
	Especular*	0.6	0.02	0.0
	Valor final	**0.98**	**0.13**	**0.12**

Obsérvese que estos valores no son el producto de los anteriores. Se debe a que en estos casos interviene también el ángulo de incidencia de la luz.

A continuación veremos cómo añadir luz a una escena definida con OpenGL. Nosotros debemos definir únicamente las luces y los materiales; OpenGL se encarga de realizar todos los cálculos.

El primer paso es activar la iluminación. Para **activar/desactivar** el uso de las luces utilizamos dos funciones que ya hemos visto anteriormente: glEnable y glDisable con el valor GL_LIGHTING.

```
glEnable (GL_LIGHTING);
glDisable (GL_LIGHTING);
```

Las luces se encuentran en una posición del espacio y brillan en una dirección. Para colocar las luces en una posición será necesario definir no sólo las componentes de la luz, sino también su posición y la dirección en la que emiten luz. OpenGL permite colocar hasta GL_MAX_LIGHTS luces diferentes en la escena, cada una con la posición que ocupa y la dirección en la que emite la luz (la constante GL_MAX_LIGHTS siempre es igual o superior a 8 y depende de la implementación). Se nombran como GL_LIGHT0, GL_LIGHT1, etc.

Además de la posición y dirección, debemos especificar de qué tipo de luz se trata (difusa, ambiental o especular), su color y otros aspectos más complejos que veremos detenidamente.

La función que utilizaremos para fijar todos los parámetros de la luz es glLight, que tiene varias versiones, aunque sólo utilizaremos dos:

```
void glLightfv (GLenum luz, GLenum valor, const GLfloat *parametro)
void glLigntf (GLenum luz, GLenum valor, GLfloat parametro)
```

El parámetro "luz" indica cuál de las luces estamos modificando. Puede valer GL_LIGHT0, GL_LIGHT1, ..., GL_LIGHT7.

El segundo parámetro "valor" especifica qué parámetro de la luz estamos fijando. Puede valer GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_POSITION, GL_SPOT_DIRECTION, GL_SPOT_EXPONENT, GL_SPOT_CUTOFF, GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION o GL_QUADRATIC_ATTENUATION.

El último parámetro es un vector de cuatro componentes o valor real, cuyo significado depende del argumento anterior. Veamos cada caso:

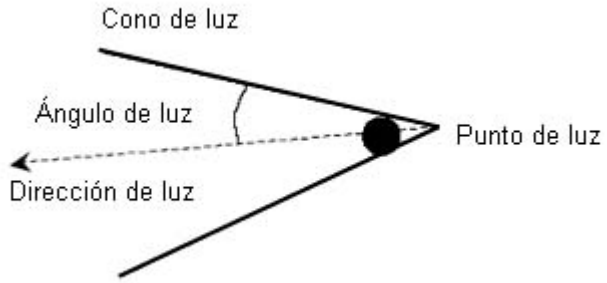
- GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR. Indica la composición RGBA de la luz.

- GL_POSITION. Vector de cuatro componentes: las tres primeras indican la posición (x,y,z) del punto de luz, y la cuarta componente puede valer 1.0, para indicar que la luz se encuentra en la posición indicada, u otro valor para indicar que se encuentra en el infinito y que sus rayos son paralelos al vector indicado por (x,y,z).

- GL_SPOT_DIRECTION. Vector de tres componentes que indica la dirección de los rayos de luz. Este vector no necesita ser normalizado y se define en coordenadas del observador. Si no se indica, la luz brilla en todas direcciones.

- GL_SPOT_CUTOFF. Las luces direccionales brillan formando un cono de luz. Esta opción nos sirve para fijar el ángulo de apertura del cono de luz. La versión adecuada en este caso es la segunda, y el último argumento

indica el ángulo desde el punto central del cono (marcado por la dirección de la luz) y el lado del cono. Este ángulo debe valer entre 0.0º y 90.0º. Por defecto, el ángulo es 180.0º, es decir, si no se indica, la luz se emite en todas direcciones.



- GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION y GL_QUADRATIC_ATTENUATION. Se utilizan para añadir atenuación a la luz. Para dar mayor realismo a las escenas, a las luces se le puede añadir un factor de atenuación, de manera que la luz es menos intensa conforme los objetos se alejan del foco de luz, como ocurre en una escena real donde la presencia de partículas en el aire hace que la luz no llegue a los objetos más alejados. El factor de atenuación (fa) se define como:

$$fa = \frac{1}{k_c + k_l d + k_q d^2}$$

Donde d es la distancia, kc es la atenuación constante, kl la atenuación lineal y kq la atenuación cuadrática. En los tres casos, se utiliza la segunda versión de la función y el tercer parámetro sirve para fijar los tres tipos de atenuación. Por defecto kc vale 1, kl vale 0 y kq vale 0. Lógicamente, la atenuación no tiene sentido cuando se trata de luces situadas en el infinito.

Veamos un ejemplo en el que se coloca una luz en la posición (50, 50, 50), con componente difusa de color verde y especular de color azul, pero sin componente ambiental. La luz brilla en dirección al punto (25, 100, 50) y el cono de luz tiene una amplitud de 30 grados.

```
// Parámetros de la luz
GLfloat luzDifusa[]    = {0.0, 1.0, 0.0, 1.0);
GLfloat luzEspecular[] = {0.0, 0.0, 1.0, 1.0);
GLfloat posicion[]     = {50.0, 50.0, 50.0, 1.0};

// Vector entre (50,50,50) y (25,100,50)
GLfloat direccion[]    = {-25.0, 50.0, 0.0}
```

```
// Activar iluminación
glEnable (GL_LIGHTING);

// Fijar los parámetros del foco de luz número 0
glLightfv (GL_LIGHT0, GL_DIFFUSE, luzDifuse);
glLightfv (GL_LIGHT0, GL_SPECULAR, luzEspecular);
glLightfv (GL_LIGHT0, GL_POSITION, posicion);
glLightfv (GL_LIGHT0, GL_SPOT_DIRECTION, direccion);
glLightf  (GL_LIGHT0, GL_SPOT_CUTOFF, 30.0);

// Activar el foco de luz número 0
glEnable (GL_LIGHT0);
```

El otro aspecto fundamental que afecta a la visualización es el material del que está hecho el objeto. Definir el material supone fijar las reflectancias para cada tipo de luz, como hemos comentado anteriormente. La forma de definir las características del material es utilizando la función glMaterial.

```
void glMaterialf (GLenum cara, GLenum valor, GLfloat parametro)
void glMaterialfv (GLenum cara, GLenum valor, const GLfloat *parametro)
```

El primer parámetro "cara" es la cara del polígono a la que asignamos el material. Puede valer GL_FRONT (cara exterior), GL_BACK (cara interior) o GL_FRONT_AND_BACK (ambas caras).

El siguiente parámetro "valor" es la característica que estamos fijando. Puede valer GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS o GL_AMBIENT_AND_DIFFUSE.

Como antes, "parámetro" será un valor real o vector de reales que indican el valor correspondiente a la característica elegida:

- GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR y GL_AMBIENT_AND_DIFFUSE. Valor RGBA de la reflectancia.

- GL_SHININESS. Sirve para fijar el tamaño del brillo para la luz especular. Se utiliza la primera versión, y el tercer parámetro es un real que puede valer entre 0.0 y 128.0. Cuanto mayor es el valor, tanto más concentrado y brillante es el punto.

- GL_EMISSION. Con esta opción podemos hacer que un objeto emita su propia luz. La versión utilizada es la segunda, y el vector de reales indica las componentes RGBA de la luz emitida por el objeto. Hemos de tener en cuenta que, aunque un objeto emita luz, no ilumina a los demás (no funciona como un punto de luz), con lo que es necesario crear un punto

de luz en la misma posición para conseguir el efecto de que ilumine al resto de objetos.

La función glMaterial afecta a todos los objetos que se definan a continuación. Por lo tanto, para cambiar de material hemos de redefinir de nuevo las características que queramos alterar del mismo. Una forma más sencilla de hacerlo es utilizar el *tracking* o dependencia de color. Esta técnica consiste en actualizar las propiedades del material utilizando la función glColor.

Para utilizar el *tracking* de color el primer paso es activarlo utilizando la función glEnable:

```
glEnable (GL_COLOR_MATERIAL)
```

A continuación, es necesario especificar qué propiedades del material van a depender del color. Esto se realiza utilizando la función glColorMaterial:

```
void glColorMaterial (GLenum cara, GLenum modo)
```

Los valores que pueden tomar estos parámetros son casi los mismos que para la función glMaterial. El primero especifica a qué cara afecta el *tracking* de color (GL_FRONT, GL_BACK o GL_FRONT_AND_BACK) y el segundo indica qué característica va a depender del color (GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION o GL_AMBIENT_AND_DIFFUSE).

A partir del momento en que se llama a la función glColorMaterial, la propiedad elegida pasa a depender del color que se fije. Por ejemplo, en el siguiente código se fija la reflectancia de la luz ambiental y difusa como dependiente del color, de manera que al alterar el color, lo que hacemos es alterar esa reflectancia.

```
// Activar tracking de color
glEnable (GL_COLOR_MATERIAL);
// Fijar la reflectancia ambiental y difusa en la cara
// exterior como dependiente del color
glColorMaterial (GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
...
// Al alterar el color, estamos alterando también la
// reflectancia ambiental y difusa del material
glColor 3f (0.2, 0.5, 0.8);
...
// Desactivar el tracking de color
glDisable (GL_COLOR_MATERIAL);
```

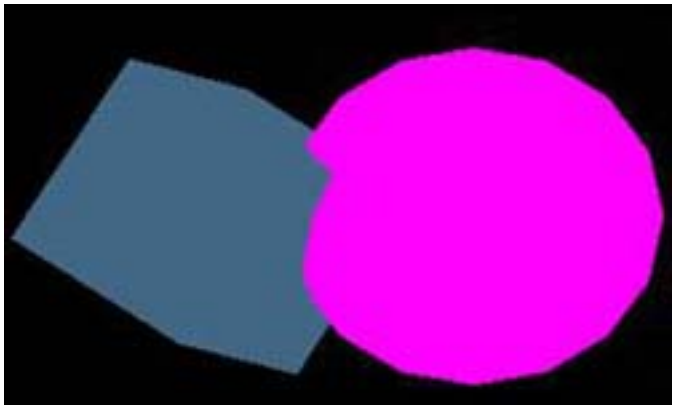
El *tracking* de color es más rápido que el cambio de material, por lo que es recomendable siempre que sea posible, especialmente cuando se producen muchos cambios de color pero no de otras propiedades del material.

3.2.1. Ejemplos de iluminación

A continuación, mostremos un breve resumen de las nociones explicadas mediante diferentes ejemplos. En ellos, se acompaña el código necesario y como la imagen resultante.

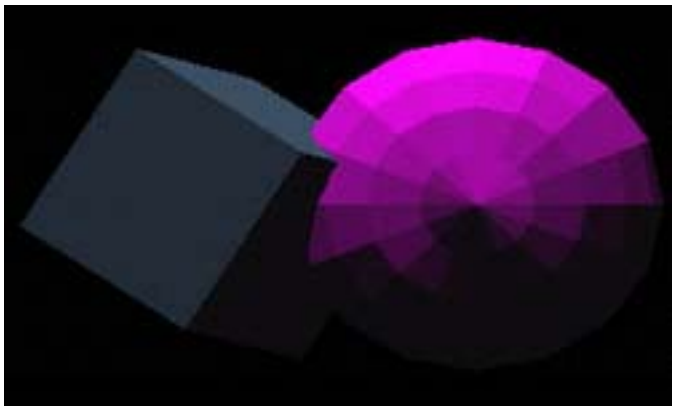
- Iluminación constante usando luz ambiente

```
GLfloat ambiente[] = {1.0, 1.0, 1.0, 1.0};
glShadeModel(GL_FLAT);
glEnable(GL_COLOR_MATERIAL);
glLightfv (GL_LIGHT0, GL_AMBIENT, ambiente);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_DEPTH_TEST);
```



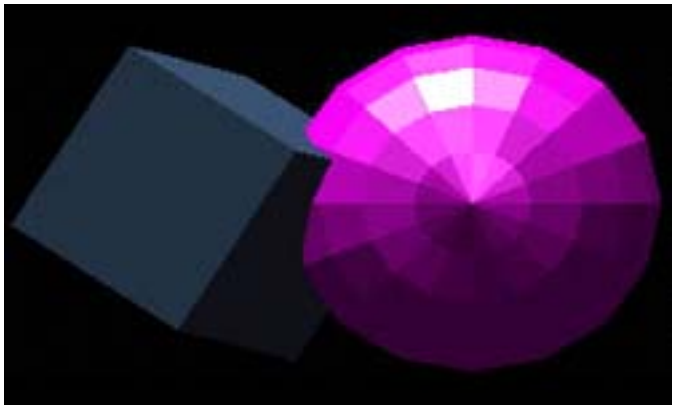
- Iluminación constante usando reflexión difusa

```
GLfloat difusa[] = { 1.0, 1.0, 1.0, 1.0};
GLfloat position[] = { 0.0, 3.0, 1.0, 0.0 };
glShadeModel(GL_FLAT);
glColorMaterial(GL_FRONT_AND_BACK, GL_DIFFUSE);
glEnable(GL_COLOR_MATERIAL);
glLightfv (GL_LIGHT0, GL_DIFFUSE, difusa);
glLightfv (GL_LIGHT0, GL_POSITION, position);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_DEPTH_TEST);
```



- Iluminación constante con reflexión especular

```
GLfloat especular[] = { 1.0, 1.0, 1.0, 1.0};
GLfloat position[] = { 0.0, 3.0, 1.0, 0.0 };
glShadeModel(GL_FLAT);
glEnable(GL_COLOR_MATERIAL);
glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR, especular);
glMaterialf (GL_FRONT_AND_BACK, GL_SHININESS, 20.0);
glLightfv (GL_LIGHT0, GL_SPECULAR, especular);
glLightfv (GL_LIGHT0, GL_POSITION, position);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_DEPTH_TEST);
```

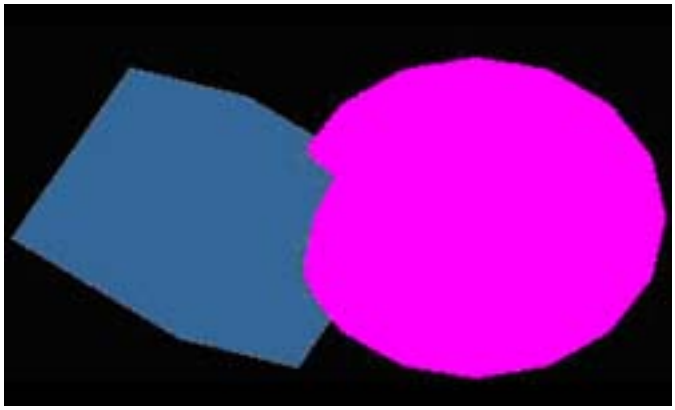


En los ejemplos que acabamos de ver, el modelo de sombreado que hemos utilizado, es decir, el método que hemos utilizado para rellenar de color los polígonos, ha sido GL_FLAT y lo hemos indicado cuando hemos invocado la función glShadeModel.

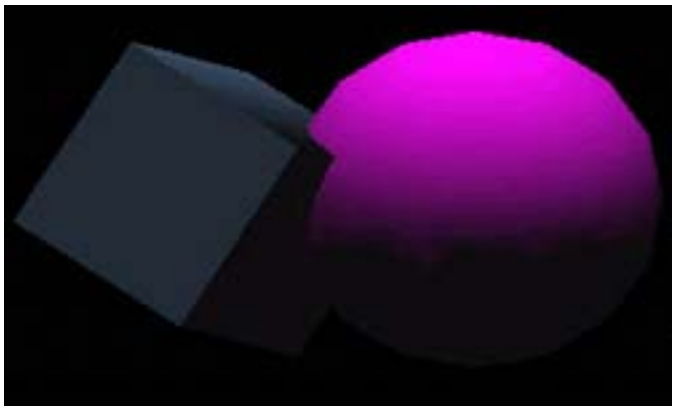
Si realizamos los mismos ejemplos, pero ahora utilizando el modelo de Gouraud, interpolando los colores de cada vértice, obtenemos los siguientes resultados. En el código tan sólo es necesario invocar a la función comentada con el parámetro GL_SMOOTH tal y como podemos apreciar en la siguiente línea de código.

```
glShadeModel(GL_SMOOTH);
```

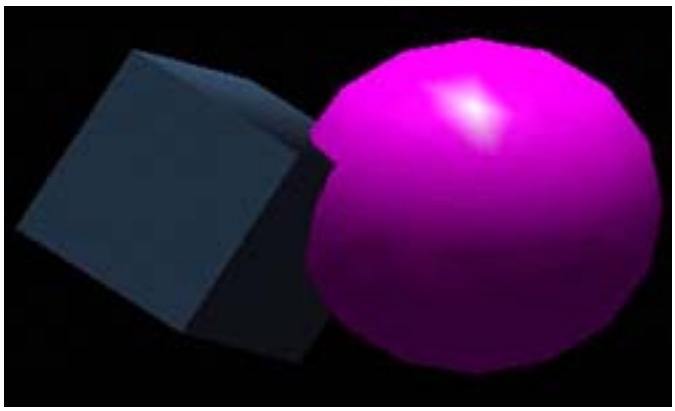
- Método de Gouraud usando luz ambiente



- Método de Gouraud usando reflexión difusa



- Método de Gouraud usando reflexión especular

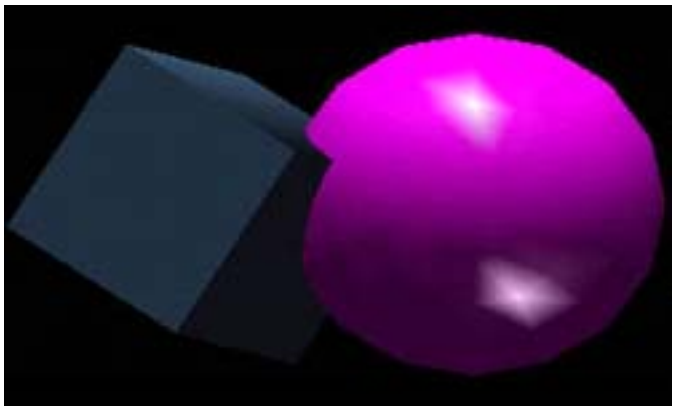


Para terminar, mostramos un ejemplo en el cual aparezcan dos fuentes de luz diferentes.

- Dos fuentes y método de Gouraud usando reflexión especular:

```
GLfloat especular[] = { 1.0, 1.0, 1.0, 1.0};
```

```
GLfloat position[] = { 0.0, 3.0, 1.0, 0.0 };
GLfloat position2[] = { 1.0, -2.0, 1.0, 0.0 };
glShadeModel(GL_SMOOTH);
glEnable(GL_COLOR_MATERIAL);
glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR, especular);
glMaterialf (GL_FRONT_AND_BACK, GL_SHININESS, 120.0);
glLightfv (GL_LIGHT0, GL_SPECULAR, especular);
glLightfv (GL_LIGHT0, GL_POSITION, position);
glLightfv (GL_LIGHT1, GL_SPECULAR, especular);
glLightfv (GL_LIGHT1, GL_POSITION, position2);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_LIGHT1);
glEnable(GL_DEPTH_TEST);
```



3.3. Texturas

En este apartado, veremos los conceptos de *multitexturing*, *mipmapping* y los diferentes filtros de textura existentes.

3.3.1. *Multitexturing*

El *multitexturing* o la aplicación de más de una textura es una técnica que permite *mapear* más de una textura sobre una primitiva, y que necesitaremos más adelante, cuando tratemos los temas de mapas de detalle y los *lightmaps*.

Para poder trabajar con multitexturas en OpenGL son necesarios los siguientes pasos:

- Empezamos con un proyecto que nos permita cargar texturas.

- Añadimos la cabecera "glext.h" a nuestro proyecto. Podemos descargar este archivo desde:
  http://oss.sgi.com/projects/ogl-sample/sdk.html

- Declaramos dos nuevos punteros a funciones con "NULL" como valor inicial.

```
PFNGLMULTITEXCOORD2FARBPROC glMultiTexCoord2fARB=NULL;
PFNGLACTIVETEXTUREARBPROC   glActiveTextureARB=NULL;
```

- Declaramos un vector con dos identificadores de texturas.

```
UINT TextureArray[2];
```

- Activar *multitexturing* y definir los dos nuevos punteros a funciones.

```
glActiveTextureARB = (PFNGLACTIVETEXTUREARBPROC)
             wglGetProcAddress("glActiveTextureARB");
glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC)
             wglGetProcAddress("glMultiTexCoord2fARB");
```

- Cargamos las texturas.

```
LoadTexture(TextureArray,"texture/image1.bmp", 0);
LoadTexture(TextureArray,"texture/image2.bmp", 1);
```

- Asignamos nuestro ID de textura a un nuevo identificador de multitextura.

```
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, TextureArray[0]);
glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, TextureArray[1]);
```

- Cuando indicamos nuestras coordenadas de textura, indicamos además nuestro identificador de multitextura. Notad que realizamos esto para ambas texturas.

```
glBegin(GL_QUADS);
   // top left vertex
   glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 1.0f);
   glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 1.0f);
   glVertex3f(-1, 1, 0);

   // bottom left vertex
   glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 0.0f);
   glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 0.0f);
   glVertex3f(-1, -1, 0);

   // bottom right vertex
```

```
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0f, 0.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0f, 0.0f);
    glVertex3f(1, -1, 0);

    // top right vertex
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0f, 1.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0f, 1.0f);
    glVertex3f(1, 1, 0);
glEnd();
```

3.3.2. *Mipmapping*

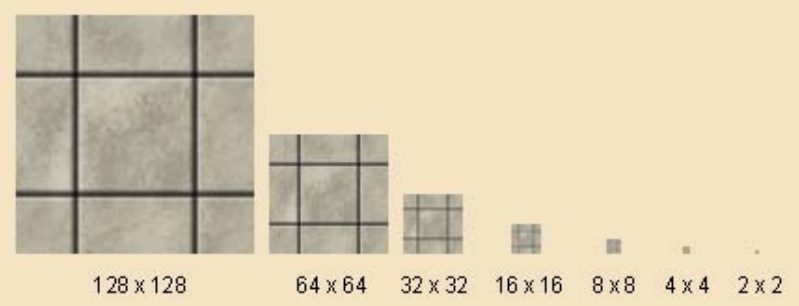
Este concepto proviene de las siglas MIP. Esta técnica surge para solventar el problema de la reducción de la calidad visual en la aplicación de texturas sobre polígonos distantes y por ello más pequeños en pantalla.

MIP

MIP son las siglas de la expresión latina *multum in parvo*, cuyo significado es "mucho en poco espacio".

La solución se basa en contar con muestras de texturas de diferentes tamaños para utilizar una u otra según convenga. Este versionado de texturas se realiza reduciendo el área por 0.25 hasta obtener la versión más pequeña posible.

El resultado de hacer esto sobre una textura es el que se muestra en la siguiente imagen:



Esta técnica no supone un problema de espacio, ya que tan sólo implica el consumo de una tercera parte más.

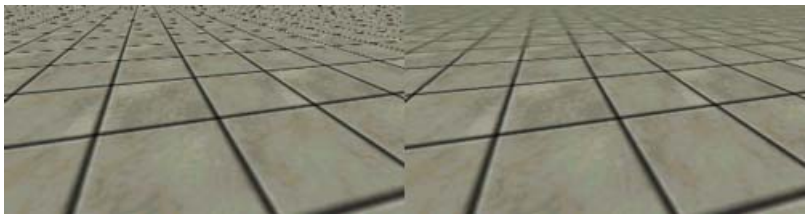
OpenGL nos brinda la posibilidad de trabajar con esta técnica automáticamente, si, a la hora de cargar la textura, en vez de utilizar el método glTexImage2D, empleamos el siguiente:

Ved también

Podéis ver el método glTexImage2D en el módulo "Videojuegos 2D".

```
GLint gluBuild2DMipmaps(GLenum target,
                        GLint format, GLsizei width, GLsizei height,
                        GLenum format, GLenum type, const void *data )
```

Los parámetros tienen el mismo significado que los de la función que vimos con anterioridad.



La diferencia entre aplicar o no *mipmapping* es evidente en la calidad del resultado.

3.3.3. Filtros de textura

Existen dos tipos de filtros para textura:

- Isotrópicos: independiente de la posición del observador
- Anisotrópicos: dependiente

Filtro isotrópico

En el grupo de filtros isotrópicos encontramos el *bilinear filtering* y el *trilinear filtering*. Describimos su funcionamiento y exponemos qué es necesario hacer para utilizarlos con OpenGL. La configuración de éstos se basa en el parámetro de minificación de la función glTexParameteri.

- *Bilinear* sin *mipmapping*. Elige como valor para un píxel el valor medio del *array* 2x2 de *texels* más cercano al centro del píxel.

```
glTexParameteri(GL_TEXTURE_2D, GL_MIN_FILTER, GL_LINEAR);
```

- *Bilinear* con *mipmapping*. Elige la textura más cercana en tamaño y procede como en el caso anterior.

```
glTexParameteri(GL_TEXTURE_2D, GL_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
```

- *Trilinear*. Elige las dos texturas más cercanas, las interpola, y aplica el filtro GL_LINEAR.

```
glTexParameteri(GL_TEXTURE_2D, GL_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
```

Filtro anisotrópico

A diferencia de los filtros isotrópicos que acabamos de apuntar, el filtro anisotrópico tiene en cuenta la orientación del polígono donde va a tener lugar el mapeo de la textura. Se suele hacer referencia a él mediante las siglas AF, *anisotropic filtering*.

Este método sirve para incrementar la calidad de la imagen de las texturas en superficies que son lejanas y que estén dispuestas en ángulos abruptos (pendientes) respecto a la cámara.

Como hacen los filtros *bilinear* y *trilinear*, elimina los efectos de *aliasing*, pero introduce menos distorsión (*blur*) y preserva un mayor detalle. Desde el punto de vista computacional, es muy caro, y recientemente se ha convertido en una funcionalidad estándar para las tarjetas gráficas.

Cuando se aplica la textura, este filtro escoge varios píxeles alrededor del punto central, pero de una muestra sesgada conforme a la vista perspectiva. De este modo, las partes más distantes de la textura contribuirán con menos muestras y las más cercanas con mayor número.

Las expresiones del tipo AF 2x, AF 4x, AF 8x, etc. hacen referencia a este filtro y el número indica la diferencia de calidad respecto a la imagen original.

Diferencias entre los filtros isotrópicos (*bilinear*, *trilinear*) y anisotrópicos (2x, 4x, 8x)

3.4. *Blending* y *alpha test*

Hasta ahora, hemos utilizado triángulos para describir superficies y siempre hemos asumido que eran opacas. Para hacer superficies transparentes emplearemos la técnica del ***blending***.

Algunas superficies transparentes que podemos querer simular son: ventanas, plástico, vidrio de color, agua, etc.

Utilización del *blending* para simular la transparencia del agua en el juego *World of Warcraft*.

Para trabajar con *blending* en OpenGL tenemos las siguientes funciones.

Activar *blending*:

```
glEnable( GL_BLEND )
```

Para controlar el color y el valor *alpha* entre los objetos fuente y destino se utiliza la siguiente función:

```
void glBlendFunc (GLenum sfactor, GLenum dfactor)
   sfactor: source blending factor
   dfactor: destination blending factor
```

Las diferentes posibilidades que tenemos a escoger, para definir los dos factores de esta función, se presentan en la siguiente tabla.

Constant Value	***Relevant Factor***	***Computed Blend Factor***
GL_ZERO	source or destination	(0,0,0,0)
GL_ONE	source or destination	(1,1,1,1)
GL_DST_COLOR	source	(Rd,Gd,Bd,Ad)
GL_SRC_COLOR	destination	(Rs,Gs,Bs,As)
GL_ONE_MINUS_DST_COLOR	source	(1,1,1,1) - (Rd,Gd,Bd,Ad)
GL_ONE_MINUS_SRC_COLOR	destination	(1,1,1,1) - (Rs,Gs,Bs,As)
GL_SRC_ALPHA	source or destination	(As,As,As,As)
GL_ONE_MINUS_SRC_ALPHA	source or destination	(1,1,1,1) - (As,As,As,As)
GL_DST_ALPHA	source or destination	(Ad,Ad,Ad,Ad)

Constant Value	*Relevant Factor*	*Computed Blend Factor*
GL_ONE_MINUS_DST_ALPHA	source or destination	(1,1,1,1) - (As,As,As,As)
GL_SRC_ALPHA_SATURATE	source	(f,f,f,1): f-min(As,1-Ad)

Una vez establecidos los factores fuente (A) y destino (B), el color final se calcula como sigue:

ColorFinal = A * ColorSource + B * ColorDestination

Las opciones más comunes son: GL_ONE, GL_ZERO, GL_SRC_ALPHA y GL_ONE_MINUS_SRC_ALPHA. Ejemplos:

```
// A = 1, B = 0
glBlendFunc (GL_ONE, GL_ZERO);
// A = 1-AlphaSource, B = AlphaSource
glBlendFunc (GL_ONE_MINUS_SRC_ALPHA, GL_SRC_ALPHA);
// A = AlphaSource, B = 1
glBlendFunc (GL_SRC_ALPHA, GL_ONE);
```

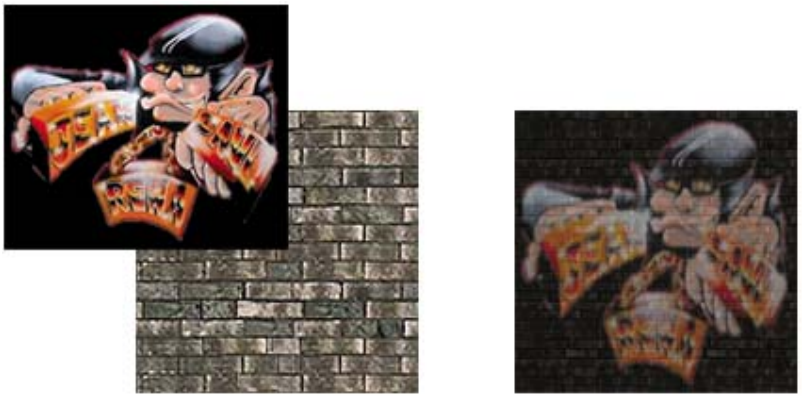


Imagen resultante aplicando la función de *blending*:
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

El ***alpha test*** se utiliza para descartar píxeles teniendo en cuenta su valor *alpha*. Se usa, sobre todo, para definir la fauna vegetal de un escenario: árboles, plantas, césped, etc.

Esta funcionalidad tiene asociada una técnica muy conocida: la recreación de fauna vegetal mediante diferentes planos orientados de manera homogénea, como muestra la siguiente ilustración.

Tres planos correctamente orientados simulando el modelo 3D de un árbol haciendo uso del *alpha test*.

Igual que en el caso anterior, en OpenGL primero es necesario activar el test y, posteriormente, definir la función de *alpha* tal y como mostramos a continuación.

Activar *alpha test*:

```
glEnable ( GL_ALPHA_TEST );
```

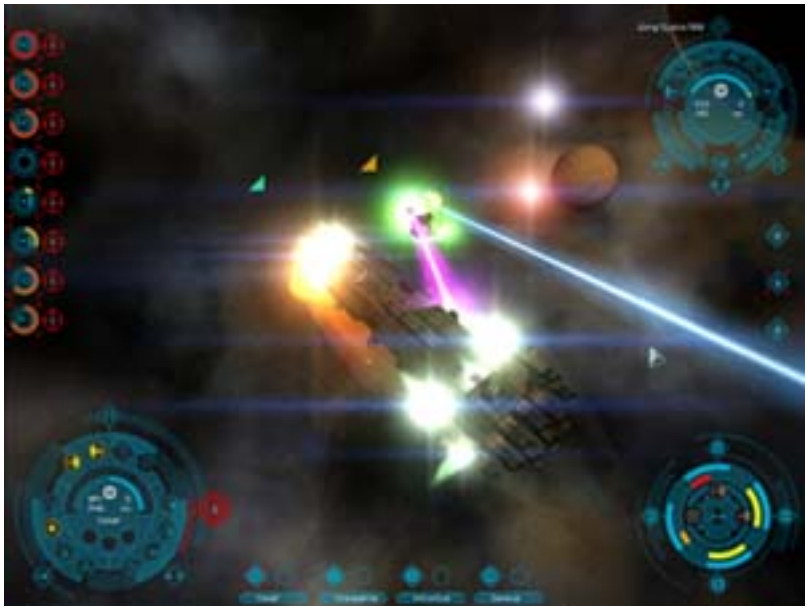
Establecer función del *alpha test*:

```
void AlphaFunc ( enum func, clampf ref );
   func: GL_NEVER, GL_LESS, GL_EQUAL, GL_LEQUAL,
         GL_GREATER, GL_NOTEQUAL, GL_GEQUAL, GL_ALWAYS
   ref: valor
```

Por ejemplo, para descartar todo aquel píxel que provenga de una compenente *alpha* superior a 0.5, procederíamos de la siguiente manera:

```
glAlphaFunc ( GL_GREATER, 0.5f );
```

La vegetación es abundante en según que juegos, como es el caso de *Crysis*, donde el *alpha test* tiene una función importante.



Blending y *alpha test* combinados para ofrecer una IGU atractiva en el videojuego *Imperium Galactica*.

3.5. Navegación 3D

A la hora de confeccionar un videojuego 3D, es imprescindible implementar un sistema de navegación a partir del cual podamos simular una vista en primera persona, en tercera persona o una cámara arbitraria. Por tanto, será necesario implementar la cámara que nos permitirá viajar por todo el escenario.

Para poder llevar a cabo esta tarea, es necesario que nos hagamos la siguiente pregunta: ¿Qué significar navegar por un mundo virtual?

A primera vista, nos puede parecer un tanto complicado qué responder e incluso por qué pensar que la clave está en hacernos esta pregunta. Vayamos un poco más allá y propongámonos otras preguntas que, sin duda, nos empezarán a despejar dudas para volver a la primera pregunta.

- Cuándo nos desplazamos hacia adelante (a través) en un escenario, ¿qué le sucede a los elementos que teníamos delante?

- Cuando nos movemos lateralmente en una dirección, ¿qué pasa con lo que estamos viendo?

- Y por último, si nosotros rotamos en una dirección, ¿cómo se ve afectado el resto del escenario respecto a cómo lo veíamos antes?

Si hacemos el ejercicio práctico y físico de recrearnos estas cuestiones llegaremos a la siguiente conclusión: el mundo se mueve y rota en dirección contraria a cómo lo hacemos nosotros. Por ejemplo, si nosotros no desplacemos a la derecha, lo que en realidad está sucediendo es que el mundo se está trasladando hacia la izquierda. He aquí la clave de la navegación.

> Implementar un sistema de navegación implica desarrollar la idea de que el mundo se mueve y rota en dirección contraria a como lo hace el usuario.

Con esta idea, y los conceptos adquiridos de programación gráfica 2D respecto a las transformaciones geométricas, ya estamos en disposición de implementar nuestro sistema de navegación.

Llevar a cabo la idea de mover todo el escenario respecto a la manera contraria como lo hace el observador, implicará el siguiente añadido a nuestro código. Antes de *renderizar* nuestro escenario, será necesario ubicarlo (transformarlo) según cómo lo queramos ver, esto es, desde donde estemos y hacia dónde estemos mirando. Por tanto, las líneas de código a las que estamos referenciando podrían ser, por ejemplo, las que describimos a continuación:

```
glRotatef( -RotatedX, 1.0, 0.0, 0.0);
glRotatef( -RotatedY, 0.0, 1.0, 0.0);
glRotatef( -RotatedZ, 0.0, 0.0, 1.0);
glTranslatef( -Position.x, -Position.y, -Position.z );
```

Tal y como acabamos de decir, rotamos y trasladamos en dirección contraria a como no hemos movido.

Exponemos, a continuación, lo que podría ser una clase *Camera* que implementa un sistema de navegación en primera persona.

```
class cCamera
{
private:
   cVector3D Position;
   cVector3D ViewDir;
   bool      ViewDirChanged;
   GLfloat   RotatedX, RotatedY, RotatedZ;
   int       width, height;
   float     speed;

public:
   void Init(int w,int h,float s);
   void Look()
   void Rotate (cVector3D Angles);
   void Update (bool keys[],int mouseX, int mouseY);
   void GetPosition(cVector3D *pos);
   void GetRotated(cVector3D *rot);
   void SetPosition(cVector3D pos);
   void MoveForwards(GLfloat distance);

private:
   void GetViewDir (void);
   void RotateX(GLfloat angle);
   void RotateY(GLfloat angle);
   void RotateZ(GLfloat angle);

   void StrafeRight(GLfloat distance);
}
```

Como podemos ver, los atributos principales son la posición y la rotación, que serán los valores acumulados por el usuario mediante teclado y ratón. Los valores anchura y altura corresponden a la ventana y tienen razón de ser para controlar la rotación con el *mouse*.

El parámetro velocidad nos servirá para parametrizar el avance de la cámara. Es decir, será el tamaño del "paso".

```
void cCamera::Init(int w,int h,float s)
{
   Position = cVector3D(0.0,0.0,0.0);
   ViewDir  = cVector3D(0.0,0.0,-1.0);

   ViewDirChanged = false;
   RotatedX = RotatedY = RotatedZ = 0.0;
```

```
    //Screen
    width = w;
    height = h;
    //Step length
    speed = s;
}
```

La función que será necesario invocar siempre antes de cualquier orden de *renderizado* colocará la cámara para que simule la navegación en el escenario en cuestión.

```
void cCamera::Look()
{
    glRotatef(-RotatedX , 1.0, 0.0, 0.0);
    glRotatef(-RotatedY , 0.0, 1.0, 0.0);
    glRotatef(-RotatedZ , 0.0, 0.0, 1.0);
    glTranslatef(-Position.x,-Position.y,-Position.z );
}
```

Funciones consultoras e inicializadoras.

```
void cCamera::GetPosition(cVector3D *pos)
{
    *pos = Position;
}
void cCamera::SetPosition(cVector3D pos)
{
    Position.x = pos.x;
    Position.y = pos.y;
    Position.z = pos.z;
}
```

```
void cCamera::GetRotated(cVector3D *rot)
{
    *rot = cVector3D(RotatedX,RotatedY,RotatedZ);
}
void cCamera::Rotate(cVector3D v)
{
    RotatedX = v.x;
    RotatedY = v.y;
    RotatedZ = v.z;
    ViewDirChanged = true;
}
```

A continuación, la función que actualiza los parámetros de navegación a partir de la información introducida mediante ratón y teclado:

- Con el **ratón** comprobamos la posición respecto al centro y le damos un significado de rotación en el eje X (si nos movemos verticalmente) o Y (si nos hemos movido horizontalmente) según sea el caso.

- A partir del **teclado** implementamos tanto el movimiento hacia delante y hacia atrás como el lateral.

```
void cCamera::Update(bool keys[],int mouseX,int mouseY)
{
   int middleX, middleY;
   float angle;

   // Move the camera's view by the mouse
   middleX = width  >> 1;
   middleY = height >> 1;

   if(mouseX!=middleX)
   {
      angle = (middleX - mouseX) / 50.0f;
      RotateY(angle);
   }

   if(mouseY!=middleY)
   {
      angle = (middleY - mouseY) / 50.0f;
      RotateX(angle);
   }

   // Move the camera's view by the keyboard
   if(keys[GLUT_KEY_UP]) MoveForwards(-speed);
   if(keys[GLUT_KEY_DOWN]) MoveForwards( speed);
   if(keys[GLUT_KEY_LEFT]) StrafeRight(-speed);
   if(keys[GLUT_KEY_RIGHT]) StrafeRight( speed);
}
```

Demás operaciones que implementan las rotaciones y traslaciones y que son invocadas en la función de actualización de la cámara que acabamos de ver.

```
void cCamera::GetViewDir(void)
{
   cVector3D Step1, Step2;
   //Rotate around Y-axis:
   Step1.x = cos( (RotatedY + 90.0) * PIdiv180);
   Step1.z = -sin( (RotatedY + 90.0) * PIdiv180);
   //Rotate around X-axis:
   double cosX = cos (RotatedX * PIdiv180);
   Step2.x = Step1.x * cosX;
```

```
    Step2.z = Step1.z * cosX;
    Step2.y = sin(RotatedX * PIdiv180);
    //Rotation around Z-axis not implemented, so:
    ViewDir = Step2;
}
```

```
void cCamera::RotateX (GLfloat angle)
{
    RotatedX += angle;
    ViewDirChanged = true;
}
```

```
void cCamera::RotateY (GLfloat angle)
{
    RotatedY += angle;
    ViewDirChanged = true;
}
```

```
void cCamera::RotateZ (GLfloat angle)
{
    RotatedZ += angle;
    ViewDirChanged = true;
}
```

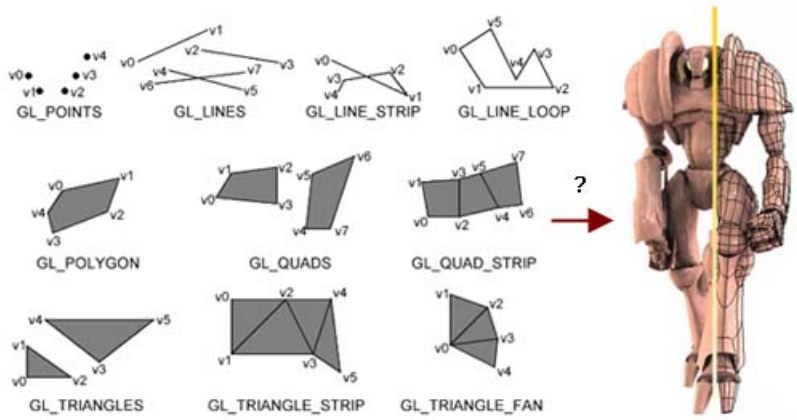
```
void cCamera::MoveForwards( GLfloat distance )
{
    if (ViewDirChanged) GetViewDir();
    cVector3D MoveVector;
    MoveVector.x = ViewDir.x * -distance;
    MoveVector.y = ViewDir.y * -distance;
    MoveVector.z = ViewDir.z * -distance;
    Position.Add(MoveVector);
}
```

```
void cCamera::StrafeRight ( GLfloat distance )
{
    if (ViewDirChanged) GetViewDir();
    cVector3D MoveVector;
    MoveVector.z = -ViewDir.x * -distance;
    MoveVector.y = 0.0;
    MoveVector.x = ViewDir.z * -distance;
    Position.Add(MoveVector);
}
```

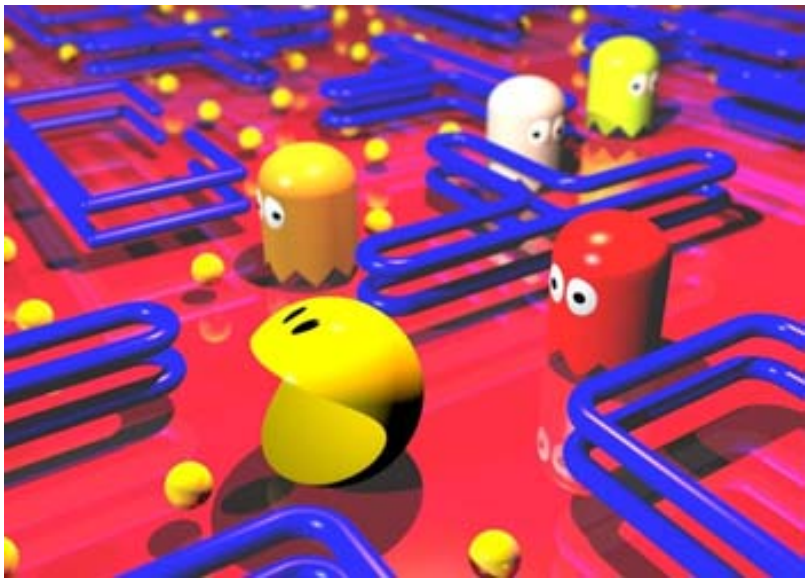
3.6. Integración de modelos 3D

3.6.1. Motivación

Para quien programa, el desarrollo de videojuegos profesionales conlleva la integración de recursos elaborados por los perfiles de modelador y animador 3D. Como bien refleja la siguiente ilustración, a primera vista parece bastante complicado que un programador con las primitivas con las que cuenta pueda elaborar cierto material 3D de calidad.

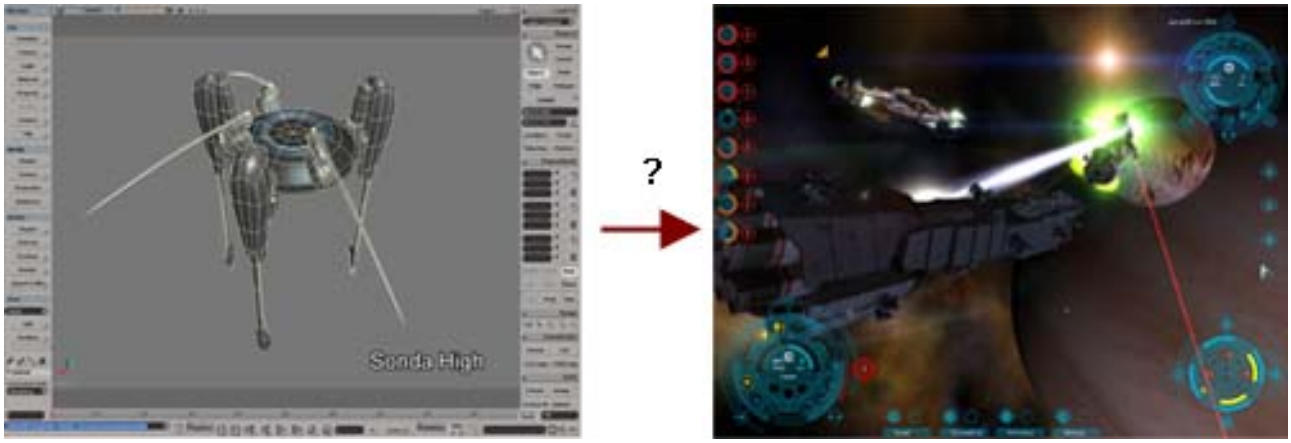


Sin embargo, hay tipos de videojuegos en los que, posiblemente, el programador podría ser autosuficiente: aquellos videojuegos donde los elementos gráficos 3D son describibles, desde el punto de vista procedural, o haciendo uso de jerarquías mediante modelos básicos. Todo ello con sus inconvenientes: posible pero muy costoso, difícil de reutilizar, modificar, ampliar, etc.



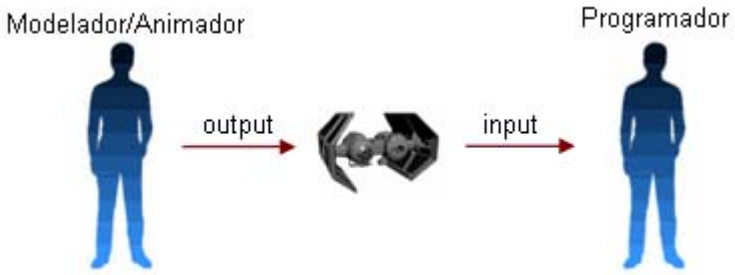
Ejemplo de tipo de juego donde un programador podría valerse por sí mismo sin la ayuda de un diseñador 3D.

Y la misma cuestión podría plantearse un diseñador: ¿Cómo podría apañárselas él para confeccionar un videojuego tan sólo a partir de recursos gráficos 3D? Algo, de nuevo, claramente representativo mediante la siguiente imagen:



Un diseñador dependerá de un programador a menos que opte por hacer *modding* sobre un juego preexistente a partir de un "Mod Toolkit" u otras herramientas semejantes. Con pequeñas nociones de *script,* podría utilizar una herramienta del tipo *Game maker*.

Por tanto, es evidente que entre estos dos mundos tiene que haber un canal de comunicación del que ambas partes saldrán beneficiadas. Como muestra la siguiente ilustración, el trabajo del modelador será la entrada de datos para el programador que le dará vida, ubicará en el mundo, etc.



3.6.2. Estructura y formatos

La estructura de los modelos 3D varía según el formato. En éstos podemos ver contemplados diferentes elementos, según la complejidad del mismo: vértices, materiales, texturas, esqueleto, animaciones, etc.

El número de formatos posibles de modelos 3D es realmente increíble. Hay que tener en cuenta que no tan sólo la industria del videojuego contribuye a ello, sino que tenemos formatos específicos para el mundo del CAD, estructu-

ras moleculares, arquitectura, etc. Por lo que respecta al contexto de los videojuegos, por ejemplo, podemos citar: XSI, 3DS, MAX, OBJ, MS3D, MDL, MD2, Blend, etc.

Aparte de estos formatos, hay dos líneas existentes que se crearon hace pocos años que pretenden consolidarse como un estándar: FBX y Collada.

A continuación, veremos algunas características de los siguientes formatos:

- Wavefront OBJ, por ser uno de los primeros que existieron
- Milkshape MS3D: muy sencillo y de gran importancia por el abanico de exportadores e importadores con el que cuenta
- Half Life MDL: ejemplo de solución privada
- FBX: ejemplo de formato que pretende consolidarse como un estándar

Wavefront OBJ

Wavefront OBJ es uno de los formatos más utilizados. Casi todas las herramientas importan y exportan a este formato. Permite trabajar con geometría poligonal, curvas y superficies.

Web

Podemos encontrar la sintaxis de este modelo en:
http://www.csit.fsu.edu/~burkardt/data/obj/obj_format.txt

Algunas de sus especificaciones son:

- Comentarios.

```
# some text
```

- Geometría de un vértice posicionado en el espacio. El primer vértice listado se indexa con el valor 1 y, los vértices subsiguientes, son numerados secuencialmente.

```
v float float float
```

- Coordenada de textura. La primera coordenada de textura es indexada con el valor 1 y, al igual que con los vértices, las siguientes consecutivamente.

```
vt float float
```

- La normal del vértice. Se indexan de la misma manera que en los dos casos anteriores.

```
vn float float float
```

- Una cara poligonal. Los números son índices dentro del vector de las posiciones de los vértices, sus coordenadas de texturas y sus normales respectivamente.

```
f int int int ...
f int/int int/int int/int ...
f int/int/int int/int/int int/int/int ...
```

Con esta sintaxis presentada seremos capaces de entender el siguiente ejemplo. Mostramos el código y la imagen resultante.

```
# texture file name
mtllib master.mtl
# vertex position
v 0.000000 2.000000 0.000000
v  0.000000 0.000000 0.000000
v  2.000000 0.000000 0.000000
v  2.000000 2.000000 0.000000
# texture coordinates
vt 0.000000 1.000000
vt 0.000000 0.000000
vt 1.000000 0.000000
vt 1.000000 1.000000
# use texture
usemtl wood
# face (square)
f 1/1 2/2 3/3 4/4
```

Ejemplo de modelo OBJ

Web

Si queremos precisar de un cargador de modelos de formato OBJ, una posible solución es dirigirse a la página web de Nate Robins, que está provista de un cargador a tal efecto, que permite leer, guardar y manipular OBJ. Es necesario obtener el fichero glm.c; allí tenemos la clase GLMmodel.

http://www.xmission.com/~nate/tutors.html

Milkshape (MS3D)

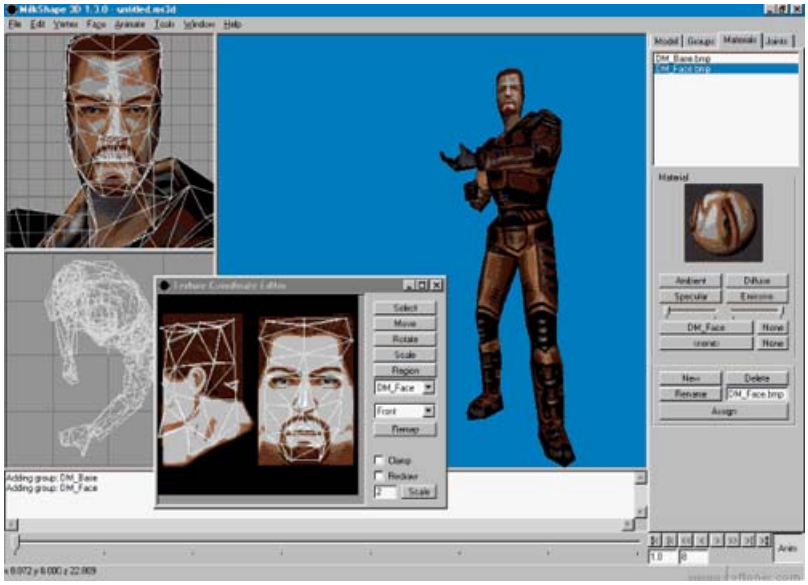
Milkshape es un editor muy sencillo y simple en prestaciones, pero que destaca por el gran número de *plugins* exportadores e importadores que integra. La versión actual es la 1.8.2 y es del octubre del 2007.

Web

La página web de Milkshape es:

http://chumbalum.swissquake.ch/index.html

Como podemos ver en la siguiente figura, el aspecto de esta herramienta es bastante sencillo. Eso hace que sea fácil de utilizar y aunque no seamos grandes diseñadores, podamos contar con un modelo 3D con relativa facilidad.



Herramienta de diseño Milkshape 3D

A continuación mostramos los diferentes formatos para los cuales contiene algún *plugin* de exportación o importación: Half-Life SMD, Half-Life MDL, Quake MDL, Quake II MD2, Quake III: Arena MD3, Unreal/UT Engine 3D, Unreal/UT Engine Skeletal Mesh PSK, Vampire: the Masquerade NOD, Genesis3D 1.0 BDY, Genesis3D 1.0 MOT, Genesis3D 1.0 ACT, Serious Sam MDL, Serious Sam LWO/SCR, Max Payne KF2, Max Payne KFS, Max Payne SKD, The Sims SKN, Wavefront OBJ, 3D Studio ASC, LightWave LOW, LightWave 6.5x LOW, AutoCAD DXF, POV-Ray INC, VRML1 WRL, VRML97 WRL, Autodesk 3DS, RAW Triangles, RenderMan RIB, Lithtech ABC v11, v12 (NOLF), Lithium UnWrapper, Playstation TMD, BioVision Motion Capture BVH, Nebula Script, Jedi Knight 3DO, GenEdit 3DT y DirectX 8.0.

En este formato, la estructura sigue los siguientes puntos:

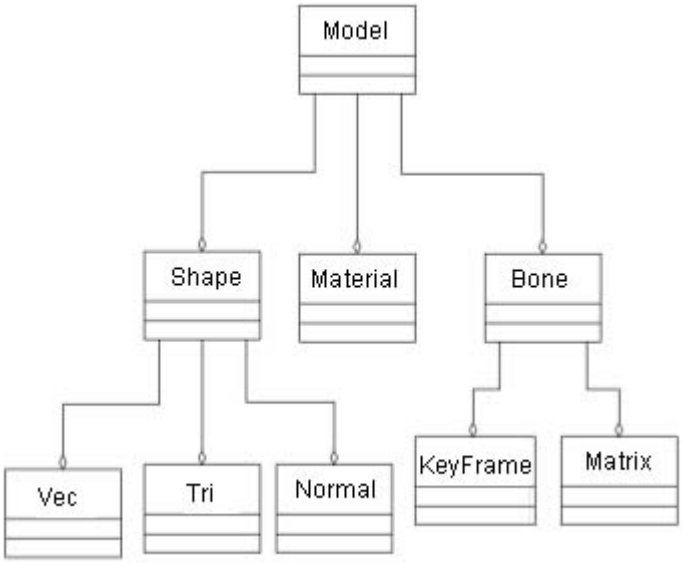
- Un modelo consiste en una o más mallas también conocidas como formas o grupos.

- Una forma tiene vectores (Vec) y triángulos (Tri). También tiene aquellos vectores normales que sean necesarios para la iluminación.

- Un material se puede aplicar a una forma para fijar el color, la reflexividad, o la textura.

- El esqueleto de un modelo se define como un conjunto de huesos (*bones*). La animación de cada hueso es definida por un número de fotogramas clave (*key frame*).

La relación que tiene lugar entre estas diferentes entidades queda reflejada en la siguiente imagen:



La propia herramienta nos permite exportar un modelo a un proyecto escrito en C utilizando como API gráfica OpenGL listo para compilar y ejecutar. Todo ello gracias al *plugin*: msOpenGLCppExporter.dll

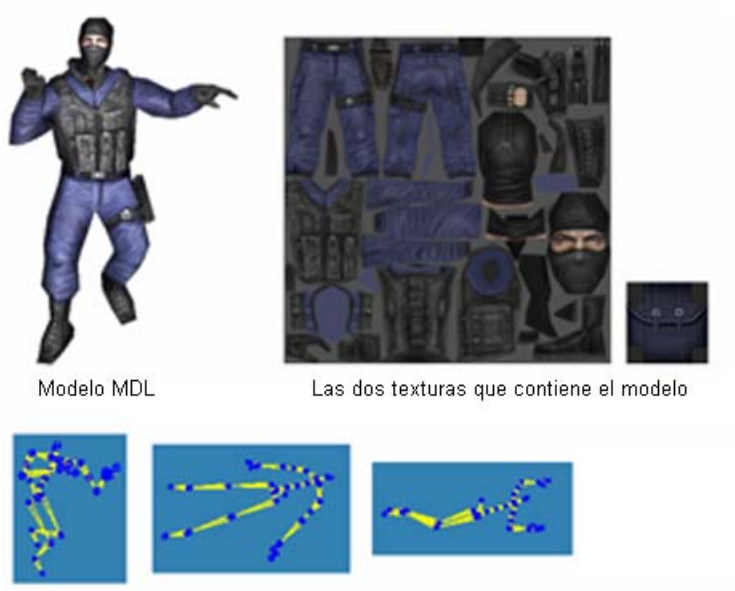
Half-Life MDL

En esta estructura de modelo tenemos empaquetados varios ficheros:

- Varios ficheros SMD que contienen las diferentes secuencias de las animaciones.

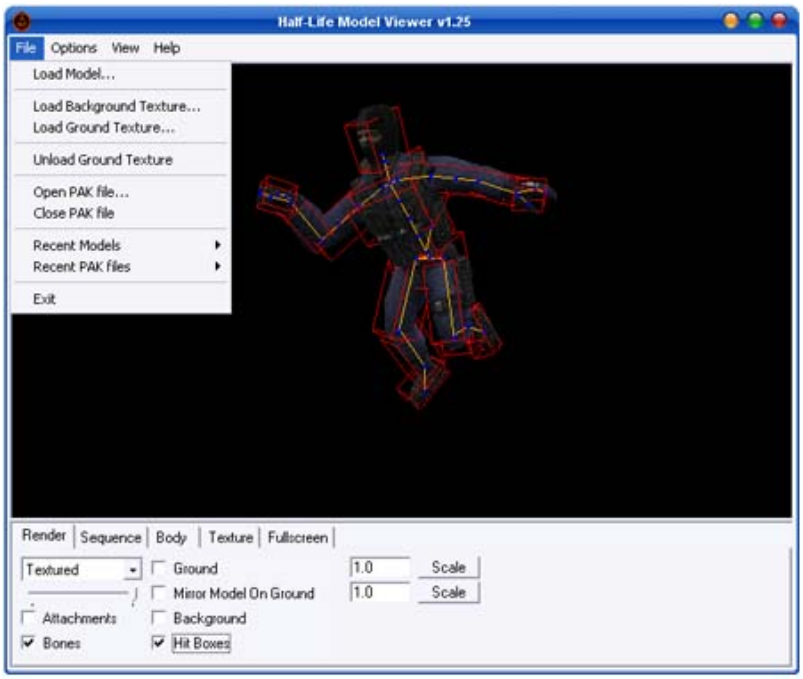
- El fichero QC, en el cual tiene lugar la definición del modelo. La compilación de este fichero genera el archivo empaquetado MDL.

- Los *bitmaps* necesarios para las texturas.



Ejemplo de modelo MDL

Para este modelo, existe el Hal-Life Model Viewer, a partir del cual podemos apreciar todas las características del modelo: animaciones, esqueletos, *bounding boxes*, etc. A continuación, mostramos una imagen de la aplicación en marcha.



Half-Life Model Viewer

FBX

Alias FBX es un formato independiente de plataforma que soporta la mayoría de elementos 3D, así como 2D, vídeo y audio. Ofrece facilidades para la conversión de animaciones y geometría por medio de múltiples estructuras de datos. Incorpora una SDK libre, fácil de utilizar y escrita en C++.

Web

Los principales enlaces de interés sobre FBX son:

www.autodesk.com/fbx

http://images.autodesk.com/adsk/files/fbx_whitepaper.pdf

http://images.autodesk.com/adsk/files/FBXSDKOverview.pdf

FBX, además, incorpora un visor a partir de un *plugin* para el reproductor QuickTime, llamado Apple QuickTime Plug-in, a partir del cual es posible navegar sobre nuestro modelo y visualizar sus principales atributos por separado o conjuntamente.

3.6.3. ¿Qué estrategia hay que seguir?

Una vez visto el apartado de motivación y algunos ejemplos de formatos, la siguiente cuestión que debemos abordar es decidirnos por una posibilidad. Se nos presentan los siguientes casos, ordenados de mayor a menor dificultad para el programador:

- **Hacer el propio formato**. Es la opción que sigue un estudio profesional con experiencia en el mercado. Los pasos a seguir serán:
  - Construir *plug-in* y añadirlo a la herramienta 3D (o construir el propio editor 3D)
  - Construir *parser*, cargador y visualizador

- **Trabajar con un formato conocido**. Las propias herramientas de modelado 3D suelen tener su propio SDK. Existe un problema a tener en cuenta: normalmente existe poca documentación y pocos ejemplos. Si escogemos esta opción, deberemos construir una librería que ataque a la SDK del programa de edición 3D.

- **Utilizar motor**. Un motor suele trabajar con varios tipos de formatos e incluso puede incorporar un *plugin* para herramientas de diseño 3D conocidas. Ante esta situación, deberemos preocuparnos de si los formatos que soporta son de nuestro interés.

- **Trabajar con librería que utiliza formato conocido**. Es la opción más cómoda de todas. Estas librerías suelen hacer referencia a formatos de modelos de videojuegos de cierta antigüedad, cuya empresa propietaria ha decidido liberar.

3.6.4. Programación con modelos 3D

Una vez solventado el punto anterior, podemos empezar a dar vida a nuestros modelos dentro de nuestro videojuego. Para incluirlos será necesario instanciarlos dentro de nuestro código y pintarlos cuando sea necesario en aquel lugar del mundo donde se encuentren situados.

- **Creación e inicialización**. Primero crearemos nuestros objetos modelos y los iniciaremos como se muestra en el siguiente ejemplo de código.

```
cModelBFX Model;
Model = new cModelFBX();
Model->Load("personaje.fbx");
```

- **Visualización**. Mostraremos nuestros personajes cuando sean visibles a la cámara, en aquel punto donde se encuentren situados y correctamente orientados.

```
if(visible)
{
   glPushMatrix();
      glTranslatef(x, y, z);
      glRotatef(roty, 0, 1, 0)
      Model->Render();
   glPopMatrix();
}
```

- **Animación**. Si el modelo en cuestión es animado tendremos que avanzar los *frames* de la animación pertinentes. Esto puede realizarse de manera constante o teniendo en cuenta el *frame-rate* del videojuego.

```
Model->AdvanceFrame(fps);
```

- **Control de estado**. Si el modelo es animado, deberemos controlar la animación que proceda en cada momento. Así, por ejemplo, el siguiente código ilustra el control para las animaciones de correr e *idle* (cuando estamos parados).

```
if(keys[GLUT_KEY_UP])
{
   if(GetState()==STOP)
   {
      Model->SetSequence(RUN);
      SetState(RUN);
   }
}
```

```
else
{
   Model->SetSequence(STOP);
   SetState(STOP);
}
```

- **Varios**. Ligado a la gestión de los modelos, encontramos involucrados otros aspectos como son:
  - La detección de **colisiones**: a partir de modelos envolventes o jerarquías de éstos, ya sea para temas de colisiones o visibilidad.
  - **Sombras**: por software o por tarjeta gráfica.
  - ***Shaders***: que permiten a la tarjeta gráfica dotar de mayor realismo los modelos a partir de técnicas como *normal mapping*, *parallax mapping*, *glow*, efectos de agua, etc.
  - ***Attachments***: para ligar objetos a modelos como pueden ser armas o demás objetos.
  - **Sistemas de partículas**: que aparecen íntimamente ligados al modelo o cuando sucede algún evento en cuestión como, por ejemplo, al realizar una magia.

Detección de colisiones en Quake III Arena

De izquierda a derecha, Shaders en Unreal Tournament y Sombras en Doom III

Sistemas de Partículas en World of Warcraft

3.7. Selección

La selección de objetos, en un escenario 3D mediante el ratón, se realiza utilizando la técnica del ***picking***. La API OpenGL proporciona un mecanismo para detectar qué objetos se encuentran en la posición del ratón o en una region cuadricular de la ventana. Enumeramos los pasos involucrados en esta tarea.

- Obtener las coordenadas del ratón en la ventana.
- Entrar en el modo selección.
- Redefinir el área de visualización, de tal manera que tan sólo una pequeña área de la pantalla alrededor del ratón sea *renderizada*.
- *Renderizar* la escena, ya sea con todas las primitivas o tan sólo aquellas que consideremos relevantes para la operación de *picking*.
- Salir del modo de selección e identificar los objetos que fueron *renderizados* en la pequeña área de la pantalla.

Para identificar los objetos *renderizados* previamente, deberán haber sido etiquetados. OpenGL nos da la posibilidad de dar un nombre a primitivas o a conjuntos de primitivas. Cuando nos encontramos en el modo de selección, un modo de *renderizado* especial de OpenGL, los objetos no son *renderizados* en el *framebuffer*. En lugar de ello, las etiquetas de los objetos y la información de profundidad son almacenadas en un *array*. Para aquellos objetos que no hayan sido nombrados, tan sólo se almacena la información de profundidad.

En el modo de selección, OpenGL devuelve un *hit* cuando una primitiva se encuentra dentro del volumen de visualización. Este conjunto de *hits*, junto a la información de profundidad de la primitiva en cuestión, es almacenado en el *buffer* de selección. Posteriormente, podremos tratar esta información y detectar los objetos más cercanos al usuario.

Con todo lo visto, las tareas del programador radican en estos cuatro puntos:

- La gestión de la pila de nombres
- El modo de selección
- El proceso de los *hits*
- Lectura de *hits*

Veamos a continuación las llamadas que OpenGL nos brinda para cada una de estas etapas.

3.7.1. La pila de nombres

- Inicializar la pila de nombres. Debe ser llamada antes de cualquier *push*.

```
void glInitNames(void);
```

- Añadir un nombre en la cima de la pila.

```
void glPushName(GLuint name);
```

- Suprimir el nombre de la cima de la pila.

```
void glPopName();
```

- Reemplazar la cima de la pila con el nombre dado. Viene a ser lo mismo que hacer `glPopName()` y después `glPushName(name)`.

```
void glLoadName(GLunit name);
```

Veamos un ejemplo de funcción que se encarga de hacer el *renderizado* de objetos, teniendo en cuenta que estamos en el modo selección.

```
#define BODY 1
#define HEAD 2
...
void RenderInSelectionMode()
{
   glInitNames();

   glPushName(BODY);
   DrawBody();
```

```
    glPopName();

    glPushName(HEAD);
    DrawHead();
    DrawEyes();
    glPopName();

    DrawGround();
}
```

3.7.2. El modo selección

Las acciones relacionadas con el modo selección son:

- Configurar el *buffer* donde obtendremos el resultado de la selección. Es necesario llamar a esta función antes de entrar en el modo selección.

```
void glSelectBuffer (GLsizei size, GLuint *buffer);
   size: tamaño del buffer
   buffer: los datos seleccionados
```

- Escoger el modo de *renderizado*.

```
GLint glRenderMode (GLenum mode);
   mode: GL_SELECT para entrar en modo selección
         GL_RENDER para entrar en el modo normal
```

- Redefinir el volumen de visión a aquella pequeña parte donde el puntero del ratón esté ubicado. Guardamos la matriz de proyección original y definimos la nueva área con la función `gluPickMatrix`.

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();

void gluPickMatrix(GLdouble x, GLdouble y, GLdouble witdth,
                   GLdouble height, Glint viewport[4]);
   x,y: posición del cursor (coordenadas de ventana)
   width, height: tamaño de la región de picking
   viewport: nuestro viewport
```

De nuevo, presentamos un ejemplo con los pasos comentados.

```
#define BUFSIZE 512
GLuint select_buf[BUFSIZE];

void StartPicking(int cursorX, int cursorY)
```

```
{
   GLint viewport[4];
   glSelectBuffer(BUFSIZE,select_buf);
   glRenderMode(GL_SELECT);

   glMatrixMode(GL_PROJECTION);
   glPushMatrix();
   glLoadIdentity();

   glGetIntegerv(GL_VIEWPORT, viewport);
   gluPickMatrix(cursorX, viewport[3]-cursorY, 5, 5, viewport);
   gluPerspective(45, ratio, 0.1, 1000);
   glMatrixMode(GL_MODELVIEW);
   glInitNames();
}
```

3.7.3. Proceso de los *hits*

Las acciones relacionadas con el proceso de *hits* son:

- Recuperar la matriz de proyección original.

```
glMatrixMode(GL_PROJECTION);
glPopMatrix();
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

- Para procesar los posibles *hits* que se hayan producido, es necesario volver al modo de *render* normal. Esto puede efectuarse mediante la llamada siguiente, que nos devuelve el número de *hits* que han tenido lugar durante el *renderizado* en modo selección.

```
hits = glRenderMode(GL_RENDER);
```

- El último paso consiste en el proceso de los *hits*.

```
if (hits != 0)
{
   ProcessHits(hits, select_buf);
}
```

3.7.4. Lectura de *hits*

Llegados a este punto, la tarea que nos queda por realizar es leer e interpretar el resultado. Para ello, deberemos centrarnos en la estructura del *buffer* de selección, en el que tenemos almacenados los diferentes *hits* que se han producido

en orden secuencial. Esta información será variable en tamaño, ya que pueden contener diferente número de nombres. El *buffer* contendrá una serie de *hit records* cuya estructura es la que presentamos a continuación:

- El primer campo nos indica el número de nombres que contiene.
- El segundo y tercer campo se corresponden a la mínima y máxima profundidad del *hit*.
- A continuación, tenemos la secuencia de nombres del *hit*.

Ejemplo del contenido de un *buffer* de selección

Contenido del *buffer*	Descripción
0	0 nombres para el primer *hit*
Valor Z1 min	Profundidad mínima del primer *hit*
Valor Z1 max	Profundidad máxima del primer *hit*
1	1 nombre en el segundo *hit*
Valor Z2 min	Profundidad mínima del segundo *hit*
Valor Z2 max	Profundidad máxima del segundo *hit*
6	Nombre del segundo *hit*
2	2 nombres en el tercer *hit*
Valor Z3 min	Profundidad mínima del tercer *hit*
Valor Z3 max	Profundidad máxima del tercer *hit*
2	Primer nombre del tercer *hit*
5	Segundo nombre del tercer *hit*

Para obtener el objeto seleccionado más cercano al observador, deberemos trabajar con el campo de información de la profundidad. La siguiente función nos devuelve el nombre del objeto más cercano.

```
void SelectHitClosest (GLint hits, GLuint buffer[])
{
   unsigned int i, j;
   GLuint names, *ptr, minZ,*ptrNames, numberOfNames;

   printf ("hits = %d\n", hits);
   ptr = (GLuint *) buffer;
   minZ = 0xffffffff;
```

```
    for (i = 0; i < hits; i++)
    {
       names = *ptr;
       ptr++;
       if (*ptr < minZ)
       {
          numberOfNames = names;
          minZ = *ptr;
          ptrNames = ptr+2;
       }

       ptr += names+2;
    }
    printf ("The closest hit names are ");
    ptr = ptrNames;
    for (j = 0; j < numberOfNames; j++,ptr++)
    {
       printf ("%d ", *ptr);
    }
    printf ("\n");
}
```

3.8. Escenarios

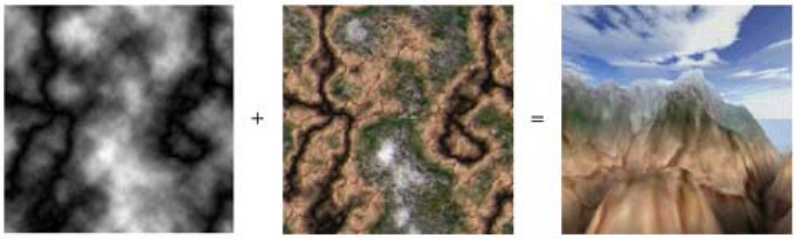
A continuación, veremos las principales técnicas que podemos apreciar en lo que respecta a la programación de escenarios exteriores (***outdoors***) e interiores (***indoors***). Utilizaremos el primero de ellos para mostrar el tratamiento de terrenos, simulación del cielo, el agua y la niebla. En el segundo caso, trataremos el tema de iluminación pre-calculada mediante *lightmaps*.

3.8.1. Terrenos

En la mayoría de casos, los terrenos son modelados mediante mapas de altura, más conocidos como *heightmaps* o también *heightfields* o *digital elevation model (DEM)*. Es una técnica sencilla de generación de terrenos que permite obtener resultados muy interesantes, formada por los siguientes elementos:

- Por un lado, tenemos un fichero, llamado mapa de alturas, compuesto por unos gradientes que, por lo general, oscilan entre los valores 0 y 255 y que, normalmente, tiene la extensión "raw".
- Por otro lado, contamos con una textura que nos proporcionará el aspecto gráfico.
- Por último, hay un algoritmo que interpreta este mapa y genera una malla de vértices conectados correctamente, formando polígonos triangulares o rectangulares.

Representación de terrenos mediante *Heightmaps*



La implementación de esta técnica en OpenGL puede llevarse a cabo mediante:

- GL_QUADS (polígonos cuadriláteros)
- GL_TRIANGLE_STRIP (concatenación de triángulos)

Implementación con GL_QUADS

Para la primera con GL_QUADS, es necesario hacer una lectura secuencial de cada uno de los valores de nuestro mapa de alturas, y formar un "quad" con los 3 valores de sus vecinos siguientes.

Una posible implementación en OpenGL podría ser la siguiente:

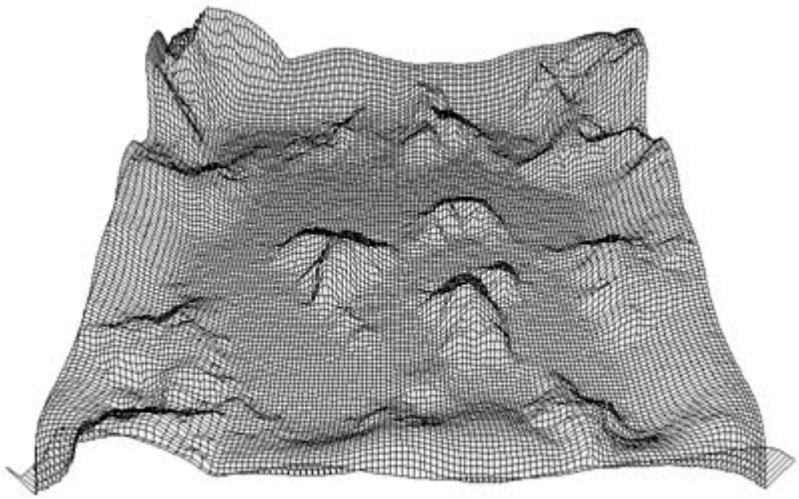
```
glBegin( GL_QUADS );

for ( X = 0; X < MAP_SIZE; X += STEP_SIZE )
{
   for ( Y = 0; Y < MAP_SIZE; Y += STEP_SIZE )
   {
      // bottom left vertex
      x = X;
      y = Height(X,Y);
      z = Y;
      glVertex3i(x, y, -z);

      // top left vertex
      x = X;
      y = Height(X,Y+STEP_SIZE);
      z = Y + STEP_SIZE ;
      glVertex3i(x, y, -z);

      // top right vertex
      x = X + STEP_SIZE;
      y = Height(X+STEP_SIZE,Y+STEP_SIZE);
      z = Y + STEP_SIZE ;
      glVertex3i(x, y, -z);

      // bottom right vertex
```

```
        x = X + STEP_SIZE;
        y = Height(X+STEP_SIZE,Y);
        z = Y;
        glVertex3i(x, y, -z);
    }
}

glEnd();
```

La función "height" es la encargada de obtener el valor del mapa de alturas dada una determinada posición. Para ello, es necesario haber cargado el fichero *raw* en un *array*. Su implementación se muestra a continuación.

```
int Height(int x, int y)
{
    int index,value;

    index = x + (y * MAP_SIZE);
    value = HeightMap[index];

    return value;
}
```

Un posible resultado que se puede obtener para un cierto mapa de alturas, siguiendo este algoritmo, es el siguiente:



Implementación de un *heightmap* mediante GL_QUADS

Implementación con GL_TRIANGLE_STRIP

La implementación con GL_TRIANGLE_STRIP consiste en la concatenación de triángulos (*triangle strip*). El proceso es semejante al de la implementación con GL_QUADS, puesto que un polígono formado por 4 aristas puede descompo-

nerse en 2 triángulos dividiéndolo por la mitad, sin ser necesario ningún cálculo adicional. Tan sólo es necesario hacer una lectura correcta de los vértices en lo que al orden se refiere.

Sin embargo, dado que los triángulos se forman por concatenación de los vértices anteriormente creados, es necesario un tratamiento especial para aquellos triángulos que se forman en los extremos. Para ello, debemos tratar la lectura secuencial de los valores del mapa de alturas una vez en una dirección, y a la siguiente, en el sentido contrario.

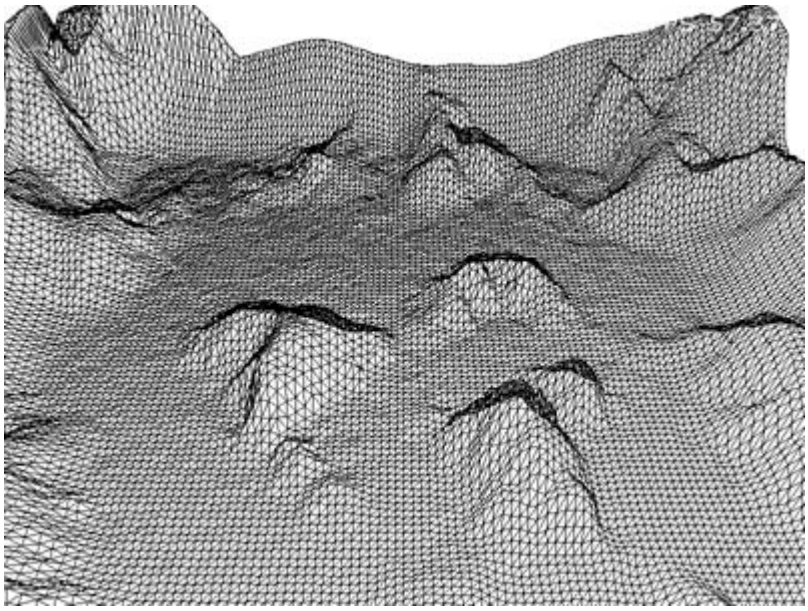
La implementación de esta técnica se desarrolla como se puede apreciar en el siguiente código.

```
glBegin( GL_TRIANGLE_STRIP );

for ( X = 0; X <= MAP_SIZE; X += STEP_SIZE )
{
   if(bSwitchSides)
   {
      for ( Y = MAP_SIZE; Y >= 0; Y -= STEP_SIZE )
      {
         // bottom left vertex
         x = X;
         y = Height(X, Y );
         z = Y;
         glVertex3i(x, y, -z);

         // bottom right vertex
         x = X + STEP_SIZE;
         y = Height(X + STEP_SIZE, Y );
         z = Y;
         glVertex3i(x, y, -z);
      }
   }
   else
   {

      for ( Y = 0; Y <= MAP_SIZE; Y += STEP_SIZE )
      {
         // bottom right vertex
         x = X + STEP_SIZE;
         y = Height(X + STEP_SIZE, Y );
         z = Y;
         glVertex3i(x, y, -z);

         // bottom left vertex
         x = X;
```

```
            y = Height(X, Y );
            z = Y;
            glVertex3i(x, y, -z);
         }
      }
      // Switch the direction the column renders
      bSwitchSides = !bSwitchSides;
   }

   glEnd();
```

La función "height" es similiar a la implementación con GL_QUADS.

El resultado de esta implementación es el siguiente:



Implementación de un *heightmap* mediante GL_TRIANGLE_STRIP

3.8.2. El cielo

Las técnicas más utilizadas para simular un cielo son:

- ***SkyBox***, a partir de un cubo envolvente.
- ***SkyDome***, en el que la figura geométrica es medio elipsoide formando una especie de cúpula.
- ***SkyPlane***, a partir de un plano.

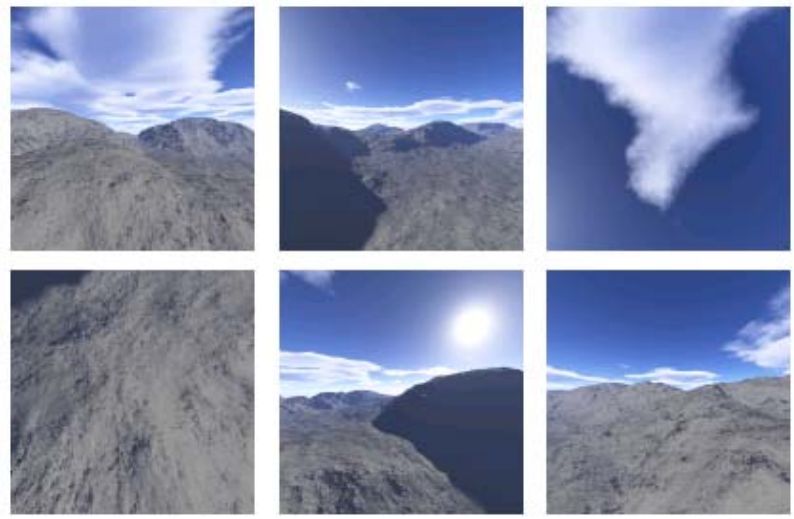
Todas ellas son similares, pues escogen una figura geométrica que envolverá todo el escenario donde tendrá lugar el juego y se mapea una textura sobre él.

Significado de *SkyBox*

La traducción directa de este término que podemos hacer del inglés, nos pueda dar una idea bien clara de que consiste esta técnica: "caja" y "cielo".

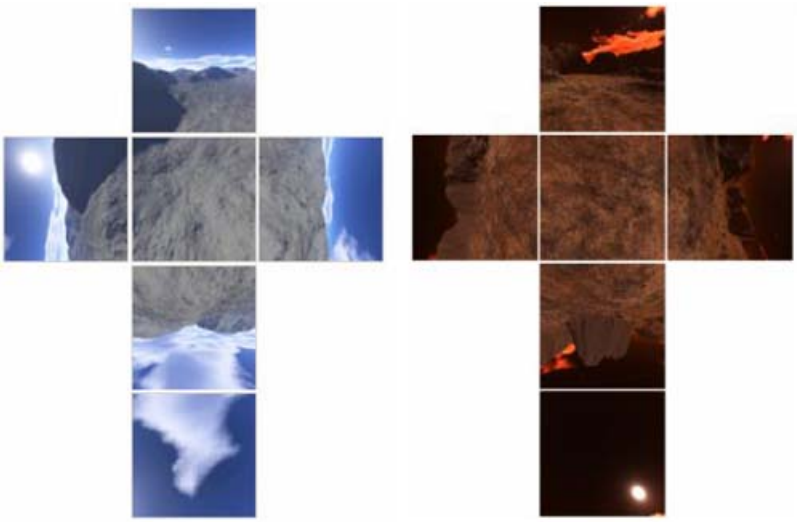
Veamos la implementación de un *SkyBox*. Esta técnica consiste en generar una caja, o mejor dicho un cubo, que contendrá en cada una de sus caras la textura de lo que podría ser, globalmente, una bóveda celestial. De esta manera, serán

necesarias 6 texturas, una para cada cara del cubo, predispuestas de tal manera que exista una continuidad entre las caras que serán colindantes, como se aprecia en los siguientes ejemplos:



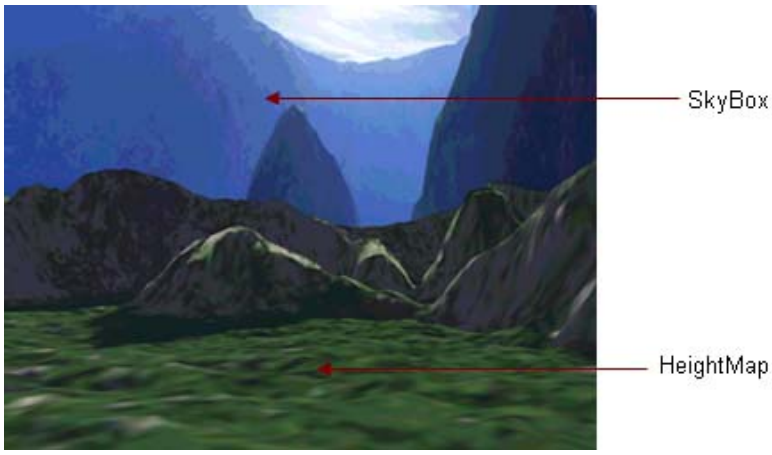
De izquierda a derecha y de arriba a abajo: bottom.bmp, forward.bmp, up.bmp, down.bmp, left.bmp y right.bmp

A partir de estas texturas, tan sólo es necesario aplicarlas debidamente sobre cada una de las caras de nuestro cubo. En las siguientes ilustraciones, podemos observar el efecto de utilizar las anteriores texturas, así como el de otro ejemplo.



Aspecto de un *SkyBox* aplanado. A la izquierda, *SkyBox* formado a través de las texturas anteriores. A la derecha, otro ejemplo con otras texturas.

Si juntamos las dos técnicas que acabamos de presentar para escenarios, *heightmaps* y *skybox*, el resultado obtenido es el siguiente:



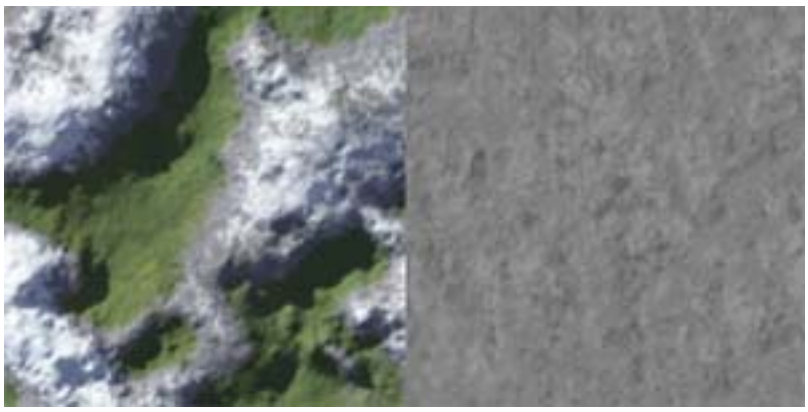
Heightmap con *SkyBox*

3.8.3. Mapas de detalle

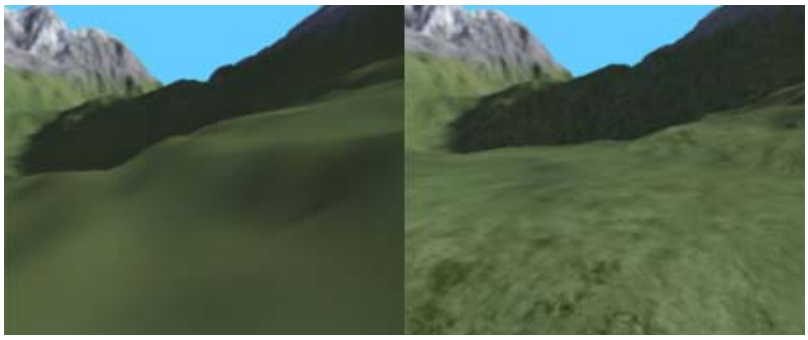
Los mapas de detalle o *detail maps* son una técnica que sirve para dotar de mayor definición y realismo el aspecto de un modelo 3D.

Las texturas que aplicamos sobre los modelos no pueden ser enormes, ya que puede provocar que el aspecto gráfico del modelo quede poco definido. La utilización de una textura base sobre el modelo, acompañada de una segunda que se replica por todo éste, es una buena solución. Comprobemos este hecho en las siguientes ilustraciones.

Izquierda, terreno original; derecha, terreno con *detail map*



Izquierda, textura base; derecha, textura detalle



Izquierda, terreno original; derecha, terreno con *detail map*

Observación

La implementación de esta técnica mediante OpenGL cabría realizarla utilizando multitexturas.

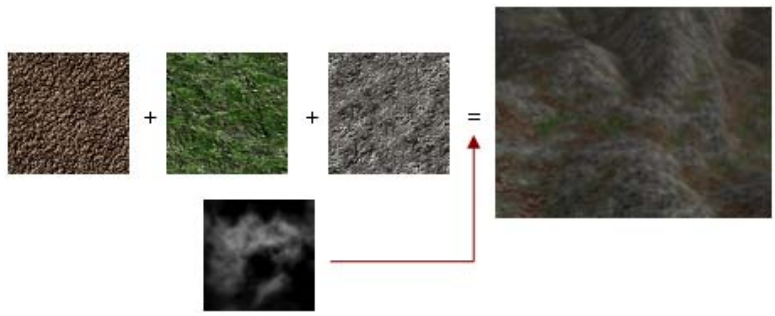
Por otro lado, es necesario abordar cómo realizar el mapeo de la segunda textura, la textura de detalle, repetida varias veces. Para ello, utilizaremos la matriz de textura a la hora de definir las diferentes matrices con las que contaba OpenGL: matriz de proyección, matriz del modelo de vista y matriz de textura.

Para ello, es necesario entrar en el modo de definición de la matriz textura y decir cómo se aplicará. En nuestro caso, repetir varias veces una textura significa escalar, para cada uno de los ejes, el valor de repetición. El siguiente ejemplo muestra cómo debemos proceder.

```
glMatrixMode(GL_TEXTURE);
   glLoadIdentity();
   glScalef(detail_level, detail_level, 1.0f);
glMatrixMode(GL_MODELVIEW);
```

La aplicación de multitexturas sobre terrenos puede darnos otras ventajas. En este sentido, aplicando o no una textura de detalle, podemos mapear un tipo de textura u otro según la altura que marque el *heightmap*. Asimismo, podemos hacer que, según la altura, tenga mayor presencia una textura u otra haciendo uso del valor *alfa* de la componente del color.

La figura que mostramos a continuación ilustra un ejemplo haciendo uso de esta última estrategia.



Terreno mapeado con multitexturas según alturas

3.8.4. La niebla

La niebla es un efecto muy utilizado en videojuegos para crear ambientes. Este concepto, que a priori puede parecer un mero elemento decorativo, conlleva varias ventajas:

- Esconde el final de la escena
- Hace que el mundo parezca mayor
- Suaviza cambios bruscos en el plano zfar
- Permite reducir el plano zfar

Los usos más comunes son la simulación de humos, la neblina, áreas de acción, etc.

Matemáticamente, la niebla interfiere en el color de la siguiente manera:

$$C = f \cdot Ci + (1- f) \cdot Cf$$

Donde

f: factor de niebla, puede ser lineal, exponencial y exponencial al cuadrado
Ci: color inicial
Cf: color de la niebla

A continuación, describimos cómo la API OpenGL permite simular la niebla.

Activar niebla:

```
glEnable(GL_FOG)
```

Configurar niebla:

```
glFogf(name, param)
   GL_FOG_MODE: GL_LINEAR, GL_EXP (def), GL_EXP2
   GL_FOG_DENSITY (por defecto: 1.0)
   GL_FOG_START (por defecto: 0.0)
   GL_FOG_END (por defecto: 1.0)
   GL_FOG_INDEX
   GL_FOG_COLOR (por defecto: {0,0,0,0})
```

El factor de niebla lineal, exponencial o exponencial al cuadrado se describe mediante las siguientes fórmulas matemáticas:

GL_LINEAR

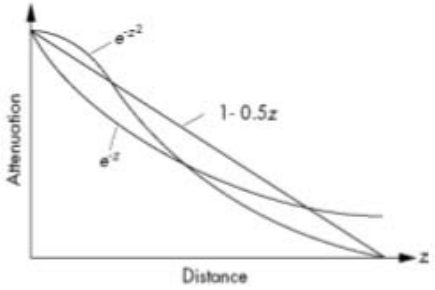
$$f = \frac{end - z}{end - start}$$

GL_EXP

$$f = e^{(-density \cdot z)}$$

GL_EXP2

$$f = e^{(-density \cdot z)^2}$$



Tipos de niebla con OpenGL

Veamos tres casos prácticos sobre un mismo escenario y, en cada uno de ellos, utilizando un factor diferente de niebla:

El código que genera cada una de estas ilustraciones es el siguiente:

```
// a) GL_LINEAR
GLfloat color[4] = {0.7, 0.7, 0.7, 1.0};
glFogfv(GL_FOG_COLOR, color);
glFogf(GL_FOG_START, 0.5);
glFogf(GL_FOG_END, 2.0);
glFogi(GL_FOG_MODE, GL_LINEAR);
```

```
// b) GL_EXP
GLfloat color[4] = {0.7, 0.7, 0.7, 1.0};
glFogfv(GL_FOG_COLOR, color);
glFogf(GL_FOG_DENSITY, 1.0);
glFogi(GL_FOG_MODE, GL_EXP);
```

```
// c) GL_EXP2
GLfloat color[4] = {0.7, 0.7, 0.7, 1.0};
glFogfv(GL_FOG_COLOR, color);
glFogf(GL_FOG_DENSITY, 1.0);
glFogi(GL_FOG_MODE, GL_EXP2);
```

Ejemplo de escenario con niebla

Existe otro tipo de niebla, llamada niebla volumétrica, más comúnmente conocida como *volumetric fog*. Ésta nos permite ubicar zonas de niebla en el escenario ocupando un volumen en concreto. Es de especial interés cuando queremos colocar niebla que tan sólo tenga presencia en diferentes áreas del terreno o, por ejemplo, para marcar zonas de interés.

Ejemplos de niebla volumétrica

Para trabajar con niebla volumétrica con OpenGL, es necesario el uso de extensiones, como en el caso de las multitexturas.

Para comprobar que nuestra tarjeta viene provista de esta funcionalidad, deberemos tener la extensión llamada "GL_EXT_fog_coord".

Acto seguido, deberemos decir a OpenGL que llevaremos a cabo niebla basada en coordenadas de vértices invocando la siguiente instrucción:

```
glFogi (GL_FOG_COORDINATE_SOURCE_EXT, GL_FOG_COORDINATE_EXT);
```

Ahora, antes de cada vértice, análogamente a como hacemos con las coordenadas de textura, será necesario explicitar la intensidad de niebla que tendrá asociado ese vértice mediante la siguiente función:

```
glFogCoordfEXT ( value )
```

El valor de la intensidad de esta niebla dependerá del valor que hayamos introducido previamente mediante los atributos GL_FOG_START y GL_FOG_END vistos con anterioridad, como si estuviéramos implementando una niebla de factor lineal.

Así, por ejemplo, ante una inicialización de la niebla volumétrica de este estilo:

```
float fogColor[4] = {0.8f, 0.8f, 0.8f, 1.0f};
glEnable(GL_FOG);
glFogi(GL_FOG_MODE, GL_LINEAR);
glFogfv(GL_FOG_COLOR, fogColor);
glFogf(GL_FOG_START, 0.0);
glFogf(GL_FOG_END, 50.0);
```

una llamada a la función con el siguiente parámetro:

```
glFogCoordfEXT(25.0);
glVertex3f(1.0,1.0,1.0);
```

hará que el vértice en cuestión quede atenuado un 50% por el color de la niebla.

Siguiendo con el escenario del ejemplo anterior, que contenía un *heightmap* con su *SkyBox,* si aplicamos niebla volumétrica según la altura de los vértices, el resultado que podemos obtener es el siguiente:

Aplicación de niebla volumétrica según la altura de los vértices

3.8.5. El agua

El agua es un elemento presente en la mayoría de videojuegos cuya acción tiene lugar en exteriores. La implementación de este elemento natural es bastante sencilla, pues implica la creación de un plano que situaremos a lo largo

de todo el escenario, a la altura que se desee, y el mapeo de una textura dinámicamente. Para dotar al agua de este dinamismo, basta con hacer que las coordenadas de texturas modifiquen su valor de la siguiente manera:

```
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, id);

glBegin(GL_QUADS);
   glTexCoord2f( 0.0f, 1.0 + move);
   glVertex3f( 0.0f, water_height, 0.0f);
   glTexCoord2f( 0.0f, 0.0f + move);
   glVertex3f( 0.0f, water_height, MAP_SIZE);
   glTexCoord2f( 1.0f, 0.0f + move);glVertex3f(MAP_SIZE, water_height, MAP_SIZE);
   glTexCoord2f( 1.0f, 1.0f + move);glVertex3f(MAP_SIZE, water_height, 0.0f);
glEnd();

glDisable(GL_TEXTURE_2D);
move += 0.00005f;
```

Con esta implementación, obtenemos un resultado como el que mostramos a continuación. Nótese la ganancia visual producida al acompañar una niebla volumétrica azulada sobre el plano del agua.

Efecto de agua mediante coordenadas de texturas dinámicas

Este tipo de agua puede ser satisfactorio, si bien es la técnica más sencilla de todas. Podemos dotar a nuestro mar de un mayor realismo si añadimos la **reflexión** de los elementos que tienen lugar en el escenario sobre ella.

La estrategia para implementar este efecto espejo consiste en realizar los siguientes pasos:

1) Definir un plano de *renderizado* donde volcaremos la información que constituirá el reflejo.
2) Trasladar los objetos a la altura necesaria para que el reflejo tenga continuidad con los elementos no reflejados.

3) Invertir los objetos a pintar en el eje Y para que el efecto espejo sea un hecho.
4) Enviar a pintar los objetos que quedarán reflejados en el agua.
5) Desactivar el plano de *renderizado*.

Es necesario conocer dos funciones de OpenGL para poder tratar el *renderizado* sobre un plano. Son las siguientes:

- Definir un plano de corte. El argumento "equation" apunta a los cuatro coeficientes de la ecuanción del plano Ax+By+Cz+D=0. El argumento "plane" es GL_CLIP_PLANEi donde el valor i está entre 0 y 5 especificando cuál de los 6 planos de corte está siendo definido.

```
void glClipPlane( GLenum plane, const GLdouble *equation);
```

- Activar el *clipping* del plano.

```
glEnable( GLenum plane );
```

Ahora exponemos la implementación de los puntos anteriores en OpenGL:

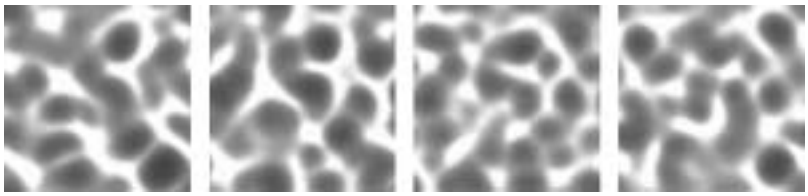
```
double reflect_plane[]={0.0f,-1.0f,0.0f, water_height};
glEnable(GL_CLIP_PLANE0);
glClipPlane(GL_CLIP_PLANE0, reflect_plane);

glPushMatrix();
   glTranslatef(0.0f, water_height*2.0f, 0.0f);
   // flip the terrain upside-down (-1 flips it).
   glScalef(1.0, -1.0, 1.0);
   // render
   SkyBox->Render();
   Terrain->Render();
glPopMatrix();

glDisable(GL_CLIP_PLANE0);
```

La implementación de la reflexión nos permite obtener un resultado como el que mostramos a continuación:

Efecto de agua con reflexión mediante *clipping*

En los dos casos que hemos tratado, estamos abarcando la visualización que queremos obtener desde el exterior. ¿Y desde el interior? Si el usuario se introduce dentro del agua, deberemos tratar de otra manera su simulación. Para ello, podemos hacer uso de una simulación de **cáusticas**, mediante las cuales, otorgaremos vida a nuestra agua y el factor de inmersión será mucho mayor.

Para hacer la simulación del agua reflejada en el terreno desde dentro de ella, nos apoyaremos en una serie de *bitmaps* que contendrán la animación de las cáusticas de manera precalculada. Estos *bitmaps* tendrán el aspecto siguiente. Mostramos 4 de las 32 imágenes que contiene el juego de pruebas.



Sprites que forman la animación de las cáusticas

El algoritmo que simulará el interior del agua deberá efectuar los siguientes pasos:

1) Control de animación de las cáusticas y elección de la textura pertinente.

2) Al igual que hicimos con la textura de detalle, ahora tendremos una tercera textura que será *renderizada* repetidas veces.

3) Tan sólo *renderizamos* la parte del terreno que no sobresale del agua con cáusticas.
La implementación queda como sigue:

```
// Activate the third texture ID and bind the caustic
// texture to it
glActiveTextureARB(GL_TEXTURE2_ARB);
glEnable(GL_TEXTURE_2D);
```

```
// Bind the current caustic texture to our terrain
glBindTexture(GL_TEXTURE_2D, tex_caust[caust_seq].GetID());

// Animation
caust_delay--;
if(caust_delay == 0)
{
   caust_seq++;
   caust_seq %= CAUSTIC_NUM;
   caust_delay = CAUSTIC_DELAY;
}

// Scale the caustic texture
glMatrixMode(GL_TEXTURE);
   glLoadIdentity();
   glScalef(CAUSTIC_SCALE, CAUSTIC_SCALE, 1.0f);

// Model view matrix
glMatrixMode(GL_MODELVIEW);
// Clip the top part of terrain
double plane[4] = {0.0, -1.0, 0.0, water_height};
glEnable(GL_CLIP_PLANE0);
glClipPlane(GL_CLIP_PLANE0, plane);

// Render the bottom of the terrain with caustics
Terrain->Render();
// Turn clipping off
glDisable(GL_CLIP_PLANE0);

// Reset the texture matrix
glMatrixMode(GL_TEXTURE);
   glLoadIdentity();
glMatrixMode(GL_MODELVIEW);

// Turn the third multi-texture pass off
glActiveTextureARB(GL_TEXTURE2_ARB);
glDisable(GL_TEXTURE_2D);
```

El resultado obtenido es este:

Vista desde dentro del agua con cáusticas

3.8.6. *Lightmaps*

La técnica de los mapas de iluminación o *lightmapping* nos permite simular la iluminación con un coste muy bajo. Un *lightmap* es una textura a la cual le damos un uso diferente. En esta textura, tenemos una zona más blanquinosa y otra más oscura. La zona negra nos proporciona el aspecto del efecto de una sombra, y la blanca el de la luz.

Estos mapas de iluminación vienen precalculados mediante la herramienta de diseño 3D de tal manera que utilizarlas (iluminar), consiste en añadir una multitextura más; sin embargo, tienen el inconveniente de que las luces deberán permanecer estáticas.

Escenario del Quake sin *lightmapping*

Escenario del Quake con *lightmapping*

Para poder llevar a cabo la implementación, será necesario trabajar con multitexturas, como ya hemos explicado anteriormente. El aspecto de las texturas de los *lightmaps* es como el que se muestra a continuación:

Lightmaps de una habitación

Y la escena *renderizada* en la cual se utilizan estos *lightmaps* es la siguiente:

Habitación iluminada con *lightmaps*

3.9. Sistemas de partículas

Los sistemas de partículas son una técnica de modelado de objetos difusos (*fuzzy objects*). Están compuestos por una serie de partículas individuales que tienen ciertos atributos, como velocidad, posición, color, tiempo de vida, etc., y son utilizados para modelar sistemas complejos como planetas, fuegos artificiales, cuero cabelludo, humo, fuego, etc.

Los sistemas de partículas ofrecen una solución para modelar objetos amorfos, dinámicos y fluidos. Son la base de los efectos especiales de los videojuegos, así como de las películas de hoy en día. Por tanto, pueden constituir un punto muy a tener en cuenta dentro de un presupuesto.

Ejemplos de sistemas de partículas

Las características principales de un sistema de partículas son:

- Permite que un objeto sea representado como una **nube de primitivas** (partículas) que definen su volumen en lugar de por polígonos que definirían su frontera.

- Es **dinámico**; las partículas cambian de forma y se mueven con el transcurso del tiempo.

- Es un objeto **no determinista**; su forma no está completamente definida.

3.9.1. Modelo básico

El modelo básico de un sistema de partículas sigue los siguientes puntos:

1) Generación: nuevas partículas son generadas e introducidas al sistema.
2) Atributos: a cada nueva partícula se le asigna sus propios atributos.
3) Extinción: todas aquellas partículas preexistentes que superan su tiempo de vida son eliminadas.
4) Lógica: el resto son desplazadas y transformadas según sus atributos dinámicos.
5) Visualización: se visualiza una imagen con las partículas en el *frame buffer*, a menudo utilizando algoritmos con un propósito especial.

Las partículas pueden ser tanto imágenes 2D como objetos 3D (cubos, esferas, modelos, etc.). He aquí varios ejemplos:

Ejemplos de partículas

A continuación, veremos cada una de las etapas que acabamos de mencionar.

Generación

Las partículas son creadas mediante procesos que incorporan un elemento de aleatoriedad. El control del número de partículas creadas puede llevarse a cabo en estas dos direcciones; por ejemplo:

- Número de partículas generadas por *frame*
  Npartsf = MeanPartsf + Rand() X VariancePartsf

- Generar un cierto número de partículas según área en pantalla

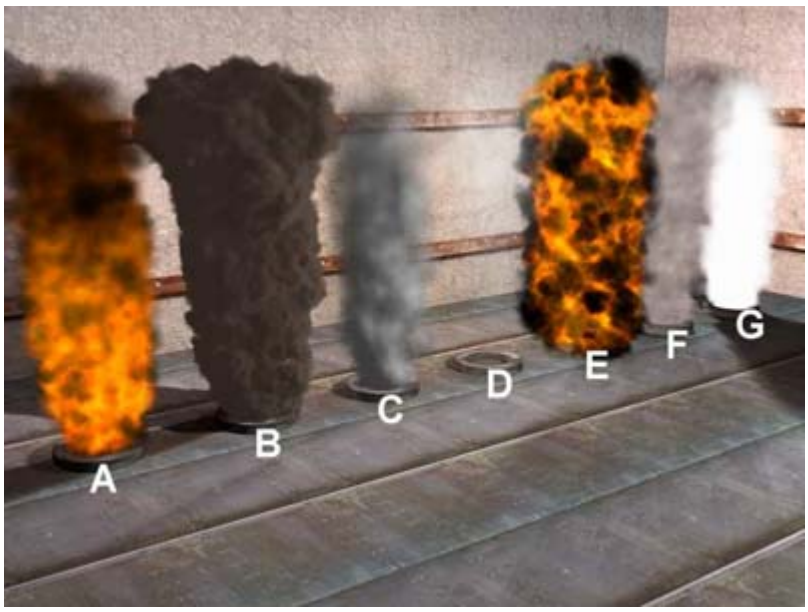
Npartsf = (MeanPartsSAf + Rand() X VariancePartsSAf) X ScreenArea

Con este método, el número de partículas depende del tamaño del objeto en pantalla.

Atributos

Inicializamos los atributos que pueda tener una partícula: posición inicial, velocidad inicial, tamaño inicial, color inicial, transparencia inicial, forma, tiempo de vida, fuerza, etc.

Estos atributos marcarán el aspecto del sistema de partículas. Siendo la lógica del sistema de partículas idéntica a otra, según los valores de inicialización podemos tener resultados muy diferentes, como muestra la siguiente ilustración.



Sistemas de partículas con mismos atributos pero valores diferentes.

Extinción

Cuando generamos una partícula se le asocia un tiempo de vida en *frames*. El tiempo de vida se ve decrementado a cada *frame* y cuando alcanza el valor cero es eliminada.

La eliminación de las partículas no supone un problema en el aspecto visual, dado que su tiempo de vida suele corresponderse a su nivel de transparencia.

Como puede apreciarse en la imagen anterior, la eliminación de una partícula no contribuye demasiado (o nada) al aspecto final de la imagen, dado que el tiempo de vida de la partícula suele ser su nivel de transparencia.

Lógica

Actualizamos la posición de cada partícula según su velocidad, posibles fuerzas externas, así como la propia lógica de la partícula que puede estar descrita mediante diferentes estados transitorios. Los demás atributos, como el color, la velocidad o el nivel de transparencia son modificados gradualmente. Se disminuye el tiempo de vida.

La lógica de un sistema de partículas formado por un banco de peces en el videojuego *WoW* consiste en describir trayectorias circulares con cierta irregularidad.

Visualización

Toda partícula viva es *renderizada* de manera acorde a sus atributos:

- Para el caso de partículas que sean modelos 3D, el *renderizado* es el convencional.
- Para los casos donde la partícula físicamente se corresponda con una imagen 2D, pueden ser necesarios tratamientos auxiliares como es la orientación a la cámara del polígono, donde se mapeará dicha textura.

Partículas 2D orientadas a la cámara en Halo 3

3.9.2. Integración con modelos 3D

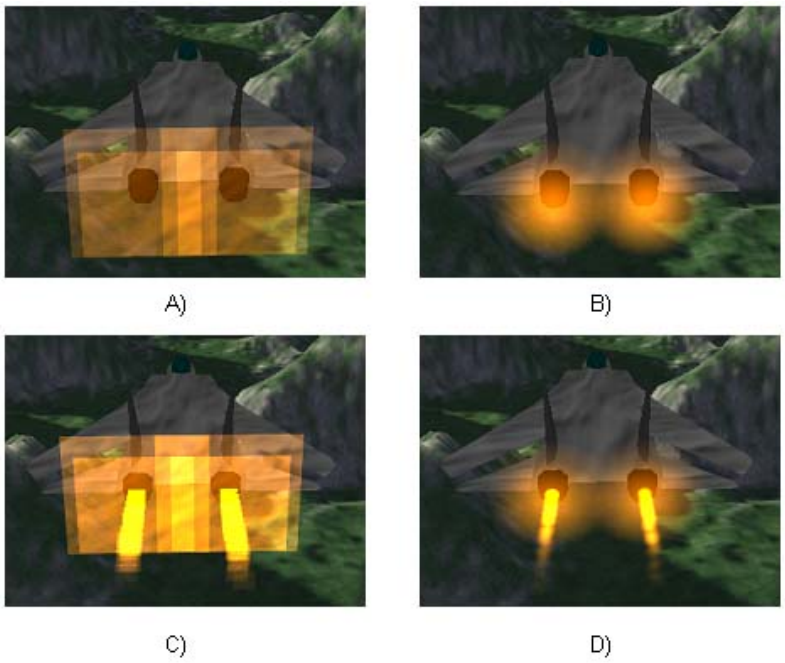
Rara vez podemos encontrar un sistema de partículas que no está asociado a algún objeto o personaje, pues son casos aislados, como la lluvia o las estrellas, por ejemplo. En la mayoría de los casos, un sistema de partículas nace (es emitido) de un modelo 3D como puede ser la leña, en el caso de una hoguera, armas o brazos, para el caso de disparos o magias, humo que sale de un objeto quemado o un tubo de escape, etc.

Para estos casos, no tenemos un solo sistema de partículas, sino una estructura algo más compleja para que este sistema tenga una continuidad visual desde el emisor y no seamos capaces de ver, a primera vista, de dónde surge.

Para ello, es necesario crear una base a partir de varios polígonos que, mediante transparencias dinámicas y transiciones de color, den vida a esa conexión entre modelo y sistema de partículas. Éste último, además, puede verse descrito a partir de una jerarquía de él mismo (varias copias) o coexistir con otros, ofreciendo un enriquecimiento visual.

Mostremos el ejemplo del turbo de una nave espacial, formado por dos sistemas de partículas idénticos y cuatro polígonos rectangulares que desempeñarán la función de base.

Ejemplo de sistema de partículas ligado a un modelo: nave y turbo



Base formada por 4 *quads* con *blending* (2 por cada) (A y B): sin textura (A) y con textura (B); base y dos sistema de partículas (1 por cada) (C y D): sin textura (C) y con textura (D).

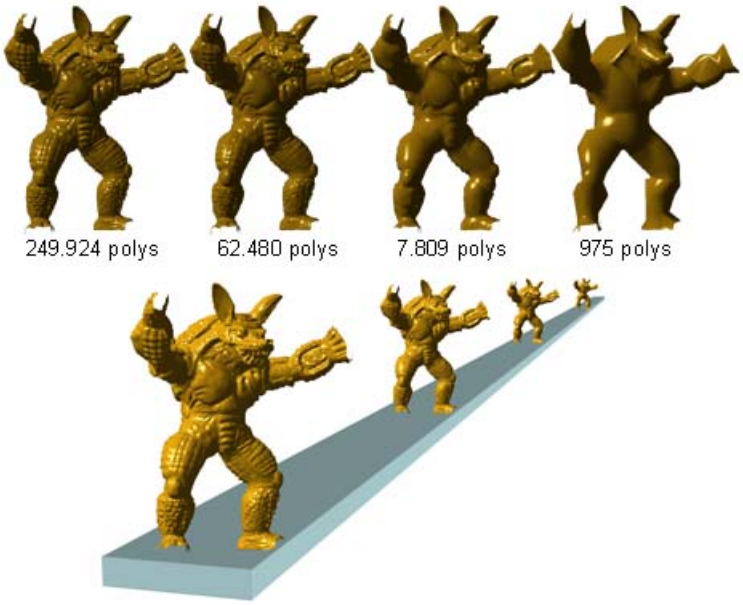
3.10. LOD's

La idea de los niveles de detalle o LOD (*levels of detail)* surge cuando nos encontramos en un contexto en el cual debemos trabajar con conjuntos de geometría muy complejos y éstos deben ser *renderizados* en tiempo real. Es decir, manteniendo un buen *frame rate* (mínimo o superior a 30 imágenes por segundo). Este es un contexto que es fácil encontrarse en el caso de los videojuegos y otras aplicaciones gráficas interactivas.

Conviene formularse la siguiente pregunta: si algo está lejos, ¿por qué malgastar número de polígonos en ello? La solución que se nos presenta nos lleva a una simplificación de la geometría poligonal para objetos pequeños, distantes o menos importantes. Ésta es precisamente la técnica de los niveles de detalle; podemos encontrarla referenciada con otras nomenclaturas, como pueden ser:

- *Polygonal simplification*
- *Geometric simplification*
- *Mesh reduction*
- *Decimation*
- *Multiresolution modeling*

Afinar entre fidelidad y rendimiento es un asunto recurrente en la computación gráfica:



La reducción del número de polígonos puede llegar a ser muy considerable mediante diferentes LOD de un modelo.

3.10.1. Técnicas

Existen diferentes técnicas para abordar la reducción de polígonos:

Impostors–billboard

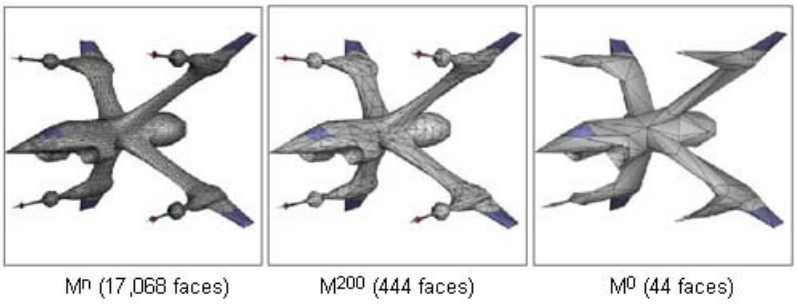
Impostors-billboard consiste en tener versiones que se corresponden a vistas 2D orientadas del objeto. La reducción es la más drástica posible, pues el modelo de representación escogido cuenta con tan sólo unos pocos triángulos y una textura. Es una solución que suele escogerse para la representación de fauna vegetal y de nubes.

Discrete LOD

Al grupo Discrete LOD pertenecen los niveles de detalles estáticos. Esta técnica consiste en almacenar varias versiones (típicamente tres) diferentes del modelo, las llamadas baja, media y alta (*low*, *medium*, *high*) y escoger cuál enviar a pintar según un criterio como, por ejemplo, la distancia.

Las **ventajas** de esta técnica son:

- El modelo de programación que hay detrás es el más simple.
- Separa los conceptos de simplificación y *renderizado*.
- No hay que calcular casi nada en tiempo real, tan sólo situarlos.
- Cada LOD es fácilmente compilable en *triangle strips*, *display lists*, *vertex arrays*, etc.



Tres niveles de detalle: alta, media y baja

En cambio, esta técnica tiene una serie de **desventajas**:

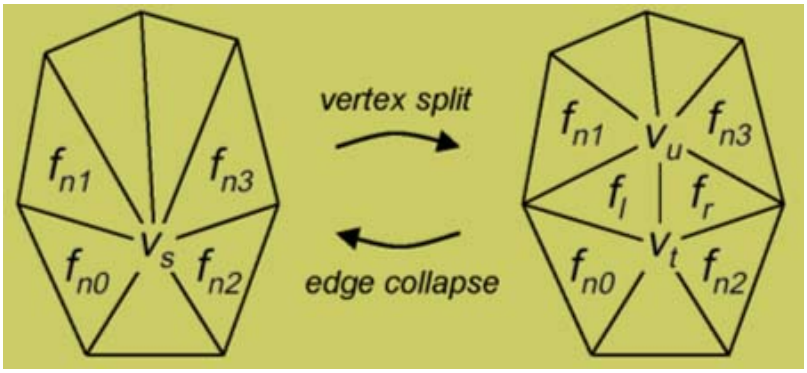
- No depende del tamaño del modelo y, por ello, si nos encontramos ante situaciones de modelos grandes o pequeños, la solución puede no adaptarse a necesidades específicas.

- Surge el *popping effect,* que es el cambio brusco que puede tener lugar justo en el momento que cambiamos de visualización entre niveles de detalle diferentes.

Continuos LOD (CLOD)

Para la técnica de Continuos LOD (CLOD), es necesario crear una estructura de datos desde la cual pueda obtenerse un nivel de detalle determinado en tiempo de ejecución. Ajusta el detalle gradual e incrementalmente, con lo que obtenemos un *morphing* suave entre diferentes niveles de poligonización.

Es la técnica más costosa, dado que los cálculos se realizan en tiempo de ejecución; sin embargo, existen publicaciones que ofrecen buenos resultados, como el *paper* "Progressive Meshes", de Hughe Poppe, basado en las operaciones de *vertex split* (división de vértices) y *edge collapse* (unión de aristas) que podemos apreciar en la siguiente ilustración:



Las operaciones *vertex split* y *edge collapse* asociadas a la técnica de los niveles de detalle continuos.

Las **ventajas** de esta técnica son notorias:

- Obtención de transiciones suaves.
- Reduce los "visual pops".

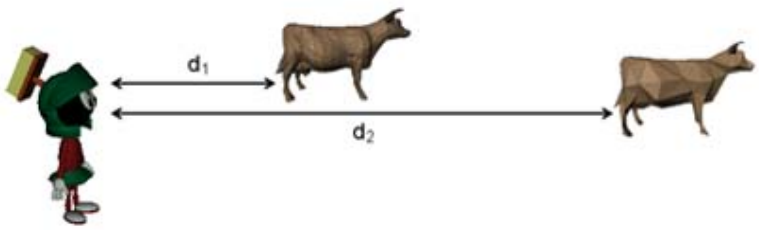
- El LOD es exacto y por ello la utilización de polígonos es óptima.
- Hace posible el "view-dependent LOD", nivel de detalle dependiente del punto de visión.
- Permite una simplificación drástica.
- Soluciona el problema con los modelos grandes.
- Para los modelos pequeños se puede solucionar con "Hierarchical LOD".

3.10.2. Criterios

La elección de un nivel de talle u otro puede venir dado por determinados factores, como la distancia, el tamaño en pantalla, las condiciones medioambientales o el *frame-rate*.

Distancia

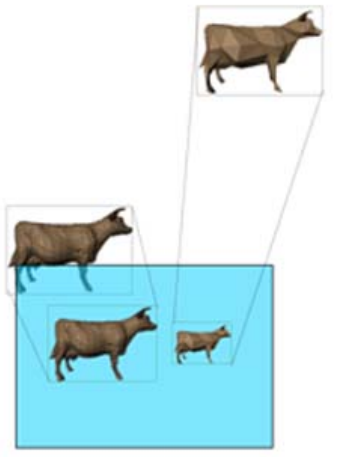
Es el criterio más obvio. Escogemos un modelo de detalle dependiendo de la distancia de éste al observador, tal y como ilustra la siguiente imagen:



La distancia puede ser un buen criterio para definir el nivel de detalle necesario para visualizar un modelo.

Tamaño

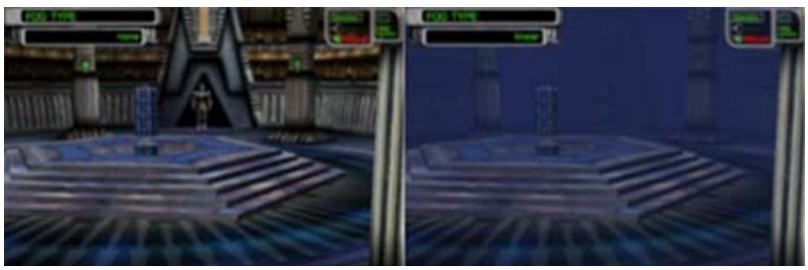
El tamaño en píxeles que ocupará en pantalla el modelo *renderizado* es otro criterio posible que nos puede justificar la elección de un nivel u otro.



El criterio del tamaño en pantalla es otra posible solución.

Condiciones medioambientales

Las condiciones que se recreen en la escena donde colocaremos los modelos pueden ser concluyentes en la elección de un nivel de detalle. Un ambiente con mucha niebla, por ejemplo, hará innecesaria la representación fiel de un modelo.



En escenarios donde las condiciones de ambiente inviten a ello, no es necesario un gran nivel de detalle, pues será imperceptible para el usuario.

Frame-rate

El *Frame-rate* se basa en optimizar según el valor del *framerate*, es decir:

- Maximizar: Σ Benefit (Object, Lod, Algorithm)
- Sujeto a: Σ Cost (Object, Lod, Algorithm) ≤ TargetFrameRate

3.10.3. Terrain LOD

Los terrenos son uno de los ámbitos de utilización de niveles de detalles más necesitados. Hacen uso de terrenos: simuladores de vuelo, videojuegos basados en *outdoors*, *geographic information systems* (GIS), aplicaciones de turismo virtual, etc.

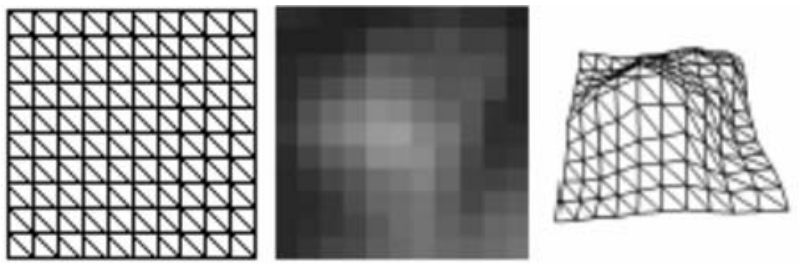
La gestión de terrenos es muy compleja, dado que trabajamos con grandes bases de datos y, simultáneamente, se trata geometría muy cercana y lejana. Por ello, es necesario aplicar técnicas de *view-dependent* LOD (niveles de detalles según la posición del observador), así como memoria virtual (*out-of-core*).

Éste es un ámbito sobre el cual se ha investigado mucho; actualmente, las estructuras que se han utilizado son: *regular grids*, TIN, *bintrees* o *quadtrees* y otras estrategias específicas como el *geomipmapping* o soluciones basadas en la GPU (*graphics process unit*). A continuación, mostraremos cada una de estas soluciones.

Regular grids

Las *regular grids* son estructuras que contienen un *array* uniforme de alturas. Son simples de almacenar y manipular y suelen estar codificadas en formatos *raster* (DEM, GeoTIFF, RAW). Las principales características que podemos citar de este modelo son:

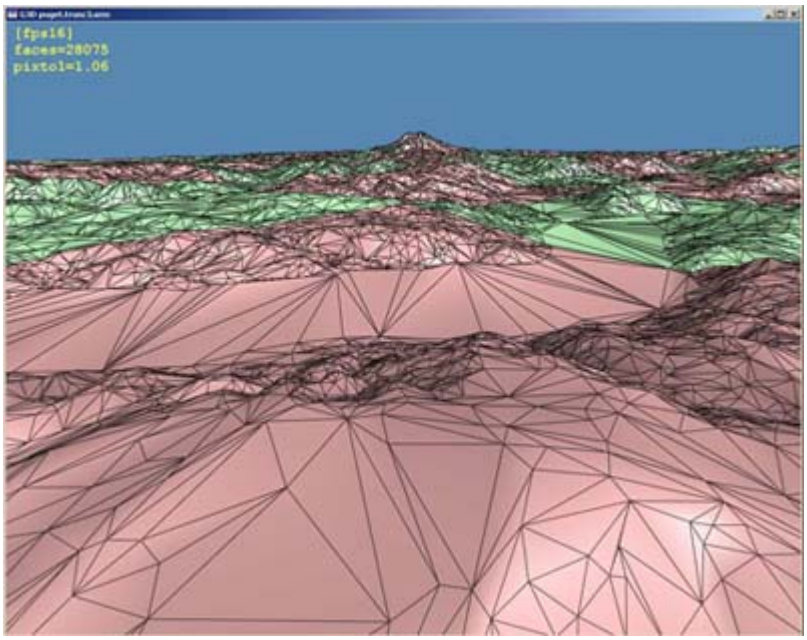
- Fácil de interpolar para obtener elevaciones
- Poca memoria/disco (sólo valores y)
- *Culling* y detección de colisiones fácil
- Utilizado por la mayoría de implementadotes
  El aspecto visual de este modelo es el que mostramos en la siguiente figura:



De izquierda a derecha: vista área del *grid*, mapa de alturas y vista 3D

Triangular irregular networks (TIN)

En el modelo de las TIN se requieren pocos polígonos para obtener precisión. En las zonas montañosas, necesitamos más muestreos que en zonas planas y, a diferencia de las *regular grids*, podemos representar cualquier tipo de formación terrenal: cordilleras, valles, *overhang* y cuevas.



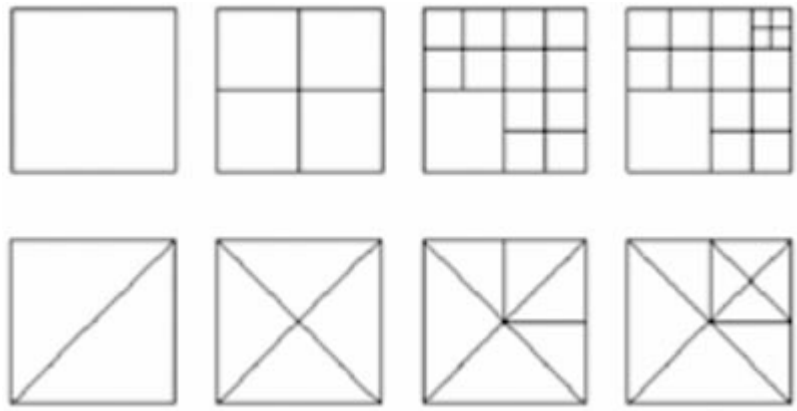
Ejemplo de representación de terreno mediante un TIN

Bintree y *quadtree*

Existen diferentes terminologías para hacer referencia a estas dos técnicas:

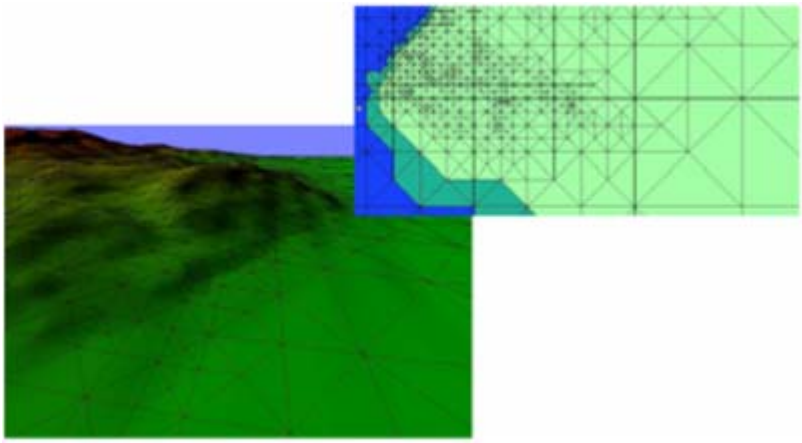
- *Binary triangle tree* (*bintree*, *bintritree*, BTT)
- *Right triangular irregular networks* (RTIN)

Ambas técnicas se basan en la división del terreno mediante un árbol binario o cuaternario según sea el caso. En éste, encontramos que un nodo es subdivido si es necesario precisar de un mayor nivel de detalle. Todo ello teniendo en cuenta que la distancia entre nodos vecinos nunca puede ser superior a un nivel. Las siguientes figuras ilustran ambos casos.



Estructura de subdivisión de un *bintree* y un *quadtree*

Veamos ahora un ejemplo resultante de modelar un terreno mediante la técnica del *bintree*.



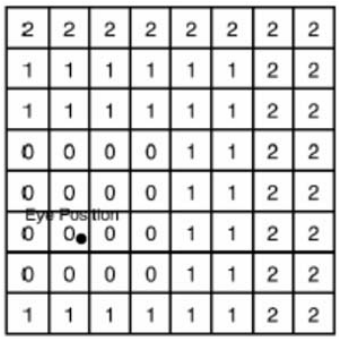
Estructura arborescente del *bintree* resultante y su *renderizado* 3D

Geomipmapping

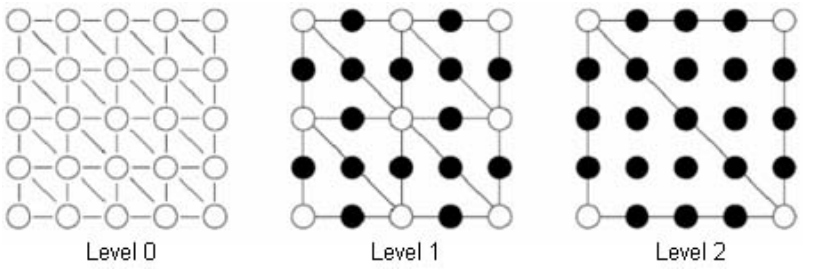
El *geomipmapping* es un mecanismo de nivel de detalle que tan sólo considera la posición del observador y no tiene en cuenta las características del terreno. Está basado en la técnica *mipmapping* (reducción del efecto del *aliasing* mediante la construcción de una pirámide de imágenes de distinto nivel de detalle).

El proceso consiste en dividir el terreno en *patches* (trozos) y clasificarlos en niveles de detalle en función de su cercanía respecto al observador.

Una vez realizada la división, el algoritmo de visualización realiza una toma selectiva de vértices (basada en *strips* de triángulos) según el nivel de detalle del trozo en cuestión. En la imagen siguiente se muestran de color blanco los vértices que serán enviados a pintar.

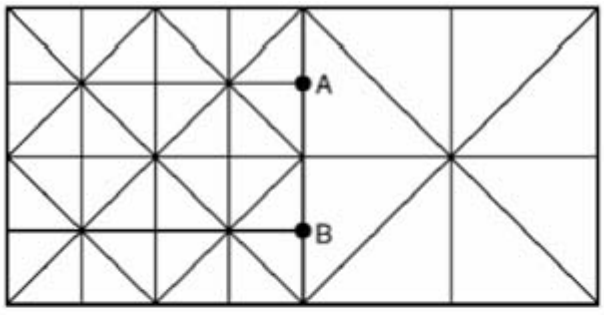


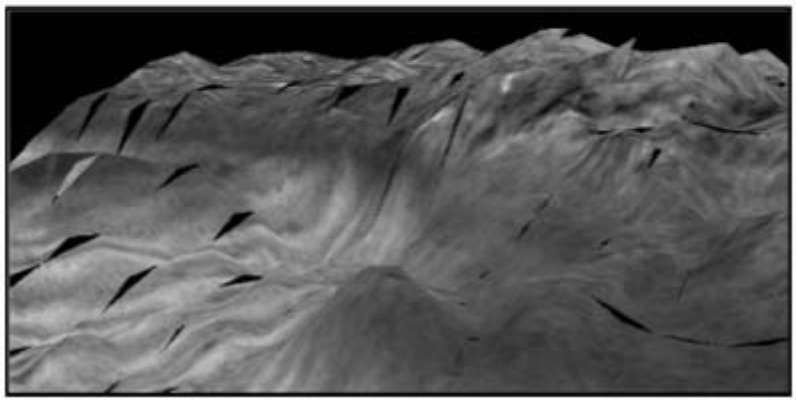
División del terreno del *geomipmapping* y clasificación de *patches* teniendo en cuenta la posición del observador.



Toma selectiva de vértices del *geomipmapping* según el nivel de detalle.

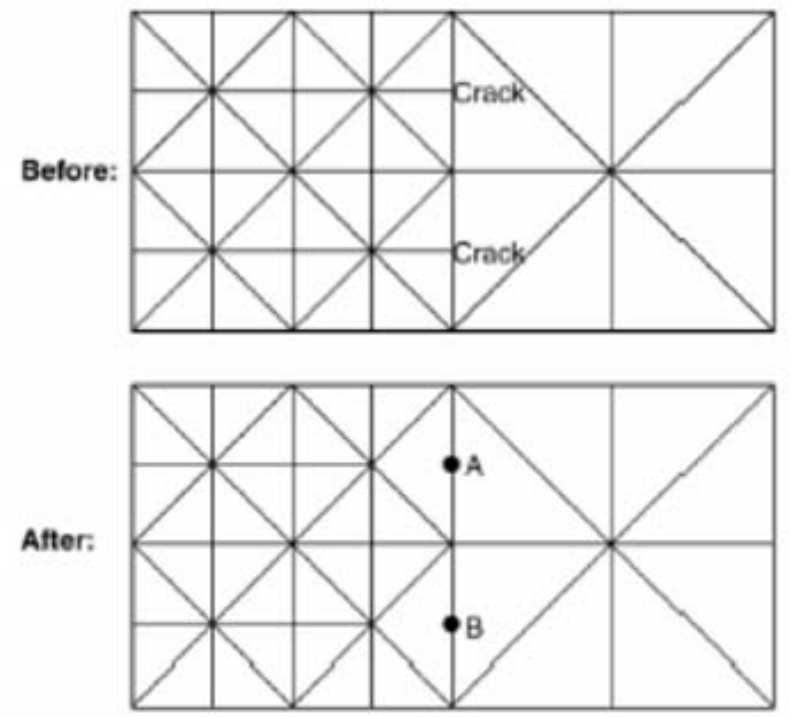
Al realizar esta operación surge un problema: aparecen *T-junctions* en las fronteras entre trozos de distintos niveles de detalle (3 triángulos tienen en A alturas distintas). Esto provoca el error de visualización que mostramos a continuación:





Aparición de agujeros en las zonas donde encontramos *T-junctions*, como en los casos A y B.

Para solucionar estos casos, el algoritmo del *geomipmapping* elimina estos vértices. La estructura resultante queda como sigue:

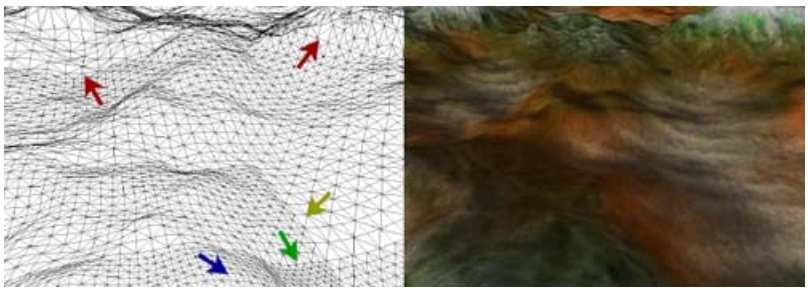


Para evitar la aparición de agujeros (*cracks*), eliminamos los vértices A y B.

Para determinar el LOD de cada *patch*, cada vez que el usuario se mueva por el terreno, el proceso de actualización deberá:

- Calcular para cada *patch* la distancia desde el observador a su centro.
- Establecer relación entre LOD y dicha distancia (tabulada, proporcional, criterios de error en pantalla etc.).

Para terminar, mostraremos un terreno implementado utilizando la técnica del *geomipmapping*. En la imagen de la izquierda, podemos apreciar los diferentes niveles de detalle, y a la derecha, el aspecto del terreno texturizado.



Terreno implementado con *geomipmapping*

3.11. Visibilidad

El concepto de visibilidad en aplicaciones gráficas interactivas es de especial interés, pues es fundamental para obtener una visualización fluida a partir de un buen *frame-rate*.

En un escenario virtual aparecen múltiples elementos interactivos, estáticos o dinámicos, así como otros elementos como terrenos o edificios. Ahora bien, rara vez van a ser visibles todos ellos al mismo tiempo y, por tanto, no es necesario enviar a pintar siempre todos estos elementos. La visibilidad trata de dar respuesta a la siguiente cuestión: no enviar a pintar aquello que no va a ser visible.

Existen dos vías a partir de las cuales podemos decidir la visibilidad de un elemento:

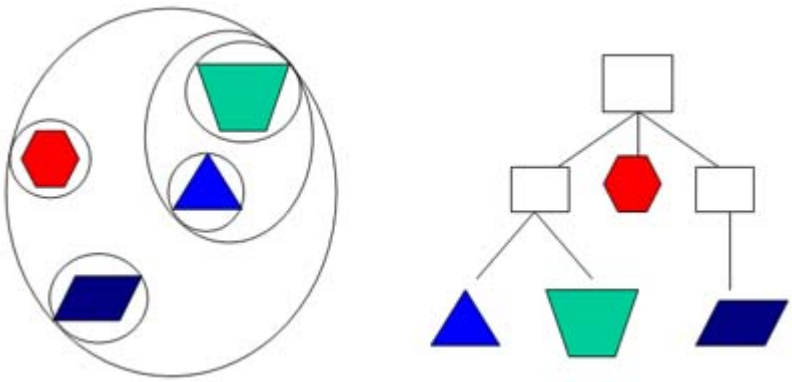
- ***Spatial data structures***. A partir de estructuras de datos de descomposición espacial, como se ha visto en el apartado de física. Las estructuras que por lo general se utilizan son: BVH (*bounding volumes hierarchies*), BSP (*binary space partition*), *k-d trees* y *octrees*.

- ***Culling***. Eliminación de aquellos elementos que no serán visibles al observador mediante técnicas específicas como son: *backface culling*, *frustum culling*, *occlusion culling* o *portal culling*.

3.11.1. *Spatial data structures*

Anteriormente, se han explicado las posibles estructuras utilizadas para organizar la geometría en un espacio n dimensional (2D y 3D), las cuales permiten acelerar consultas para la detección de colisiones o para el tema que nos concierne en el actual apartado, la visibilidad.

En estas estructuras, cada nodo contiene el volumen envolvente correspondiente a la geometría de un sub-árbol y la geometría se encuentra en los nodos hoja.

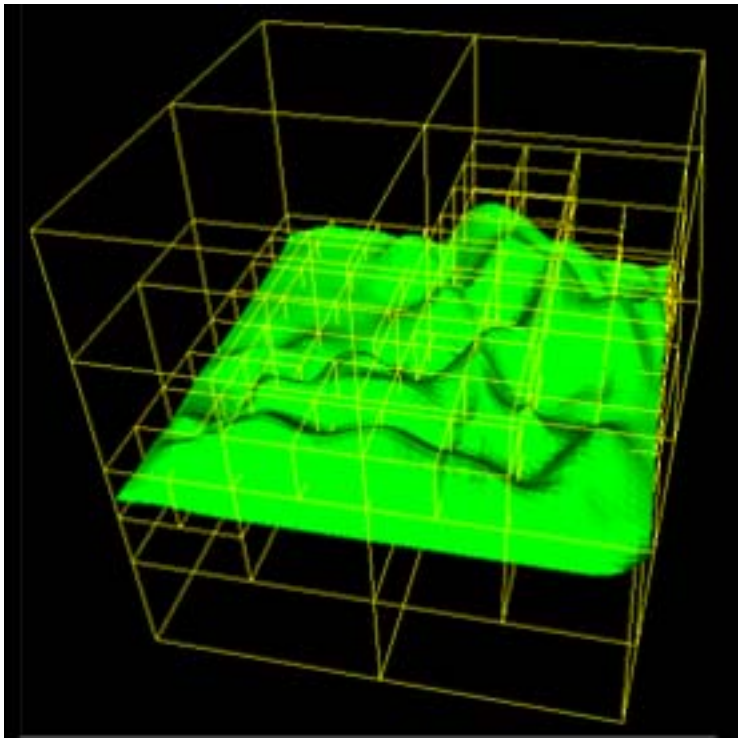


BVH con la información almacenada en los nodos hoja

La subdivisión del espacio se realiza mientras se da alguna de estas circunstancias:

- Existe algún elemento.

- El número de primitivas por cada nodo hoja es mayor a un determinado valor (*threshold*).



Octree subdivido teniendo en cuenta el número de vértices por nodo hoja.

Dependiendo de las situaciones, los algoritmos más convenientes pueden variar:

- **Interiores** (*indoors*). El BSP nos permite subdividir de manera natural las habitaciones de un edificio. Los planos de subdivisión serán precisamente las paredes divisorias que podamos tener.

- **Exteriores** (*outdoors*). Los *octrees* permiten agrupar o dividir la geometría según donde haya falta con independencia del escenario. Aplicar un BSP en este contexto no tiene tanto sentido, dado que los planos divisorios no tienen la correspondecia explícita del caso anterior.

3.11.2. *Culling*

El *visibility culling* da respuesta a la ocultación de la geometría desde diferentes puntos de vista:

- *Backface culling*: eliminación de las caras ocultas; viene implementado por la API.

- *Frustum culling*: eliminación de los objetos que no son visibles por nuestro *frustum*, o cono de visión, definido por la cámara.

- *Occlusion culling*: eliminación de objetos que son tapados por otros.

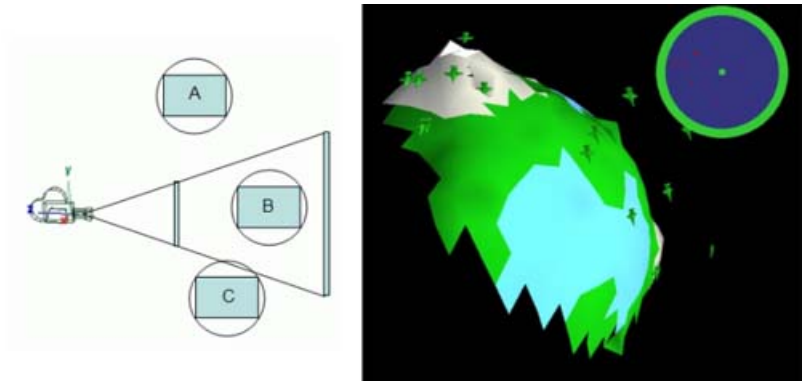
- *Portal culling*: selección de las habitaciones visibles a partir de portales de visión.

La operación de *culling* nos devuelve lo que se conoce como un PVS (*potential visibility set*), el conjunto de objetos visibles. A continuación, veremos las técnicas citadas.

Frustum culling

Esta técnica consiste en seleccionar aquellos objetos que podamos ver a partir de nuestra pirámide de visión (*frustrum*). Para ello, será necesario construir los 6 planos que conforman nuestro *frustum* y hacer test sencillos *in*/*out*. Para los objetos de la escena, son suficientes las estructuras *bounding spheres* o *bounding cubes*.

En la siguiente imagen, podemos apreciar a la izquierda la idea conceptual, y a la derecha, la imagen resultante 3D en un contexto determinado.



Eliminación de objetos que pintar mediante *frustum culling*

A continuación, presentamos el código necesario para obtener los 6 planos del *frustum* en OpenGL, así como la operación consultora que nos permitirá decidir si un punto es interior a éste.

```
float f[6][4];// Frustum

void ExtractFrustum()
{
   float p[16]; // Projection Matrix
   float m[16]; // Modelview Matrix
   float c[16]; // Clipping
   float t;

   // Get the current PROJECTION matrix from OpenGL
   glGetFloatv(GL_PROJECTION_MATRIX, p);
   // Get the current MODELVIEW matrix from OpenGL
```

```
glGetFloatv(GL_MODELVIEW_MATRIX, m);

// Multiply projection by modelview)
c[0] = m[0]*p[0]+m[1]*p[4]+m[2]*p[8]+m[3]*p[12];
c[1] = m[0]*p[1]+m[1]*p[5]+m[2]*p[9]+m[3]*p[13];
c[2] = m[0]*p[2]+m[1]*p[6]+m[2]*p[10]+m[3]*p[14];
c[3] = m[0]*p[3]+m[1]*p[7]+m[2]*p[11]+m[3]*p[15];
c[4] = m[4]*p[0]+m[5]*p[4]+m[6]*p[8]+m[7]*p[12];
c[5] = m[4]*p[1]+m[5]*p[5]+m[6]*p[9]+m[7]*p[13];
c[6] = m[4]*p[2]+m[5]*p[6]+m[6]*p[10]+m[7]*p[14];
c[7] = m[4]*p[3]+m[5]*p[7]+m[6]*p[11]+m[7]*p[15];
c[8] = m[8]*p[0]+m[9]*p[4]+m[10]*p[8]+m[11]*p[12];
c[9] = m[8]*p[1]+m[9]*p[5]+m[10]*p[9]+m[11]*p[13];
c[10]= m[8]*p[2]+m[9]*p[6]+m[10]*p[10]+m[11]*p[14];
c[11]= m[8]*p[3]+m[9]*p[7]+m[10]*p[11]+m[11]*p[15];
c[12]= m[12]*p[0]+m[13]*p[4]+m[14]*p[8]+m[15]*p[12];
c[13]= m[12]*p[1]+m[13]*p[5]+m[14]*p[9]+m[15]*p[13];
c[14]= m[12]*p[2]+m[13]*p[6]+m[14]*p[10]+m[15]*p[14];
c[15]= m[12]*p[3]+m[13]*p[7]+m[14]*p[11]+m[15]*p[15];

// RIGHT plane
f[0][0] = c[3] - c[0];
f[0][1] = c[7] - c[4];
f[0][2] = c[11] - c[8];
f[0][3] = c[15] - c[12];

// LEFT plane
f[1][0] = c[3] + c[0];
f[1][1] = c[7] + c[4];
f[1][2] = c[11] + c[8];
f[1][3] = c[15] + c[12];

// BOTTOM plane
f[2][0] = c[3] + c[1];
f[2][1] = c[7] + c[5];
f[2][2] = c[11] + c[9];
f[2][3] = c[15] + c[13];

// TOP plane
f[3][0] = c[3] - c[1];
f[3][1] = c[7] - c[5];
f[3][2] = c[11] - c[9];
f[3][3] = c[15] - c[13];

// FAR plane
f[4][0] = c[3] - c[2];
f[4][1] = c[7] - c[6];
```

```
      f[4][2] = c[11] - c[10];
      f[4][3] = c[15] - c[14];

      // NEAR plane
      f[5][0] = c[3] + c[2];
      f[5][1] = c[7] + c[6];
      f[5][2] = c[11] + c[10];
      f[5][3] = c[15] + c[14];

      for(int i=0;i<6;i++) {
         normalize(f[i]);
      }
   }
```

```
bool PointInFrustum(float x, float y, float z)
{
   for (int p = 0; p < 6; p++)
   {
      if (f[p][0]*x+f[p][1]*y+f[p][2]*z+f[p][3] <= 0)
         return false;
   }
   return true;
}
```

Occlusion Culling

Los modelos complejos de gran profundidad *renderizan* muchos píxeles que, al final, serán descartados durante el test del *z-buffer*. Transformar vértices y *rasterizar* primitivas que son ocluidas por otros polígonos reduce el *frame-rate* de la aplicación y, sin embargo, no aporta información a la imagen resultante.

Los algoritmos de *occlusion culling* intentan identificar estos polígonos que no van a ser visibles y descartarlos, de tal manera que no sean enviados a pintar. Estos algoritmos pretenden determinar las superficíes visibles y no visibles mediante una granularidad mayor que el test por medio de píxeles.

Con esta finalidad, se han creado algoritmos de *occlusion culling* en dos direcciones:

- Los que trabajan en **espacio-objeto**
- Los que trabajan en **espacio-imagen**

Lecturas complementarias

Sobre los algoritmos que trabajan en espacio-objeto, podéis consultar: **S. Coorg; S. Teller** (1996). "A spatially and temporally coherent object space visibility algorithm". *Technical Report TM* (núm. 546) y **D. Luebke; C. Georges** (1995). "Portals and mirrors: Simple, fast evaluation of potentially visible sets". En: *Proceedings of the 1995 symposium on Interactive 3D Graphics* (pág. 105).

Sobre los algoritmos que trabajan en espacio-imagen, podéis consultar: **H. Zhang; D. Manocha; T. Hudson; K. Hoff III** (1997). "Visibility culling using hierarchical occlusion maps". En: T. Whitted (ed.). *Computer Graphics (SIGGRAPH '97 Proceedings)* (vol. 24, pág. 77-88).

En general, el comportamiento de un algoritmo de oclusión que trabaja en espacio-objeto puede ser descrito mediante el siguiente pseudo-código:

```
function OcclusionCulling (G: input graphics data)
   Or: occlusion hint
   Or = empty
   foreach object g in G
      if (IsOccluded(g, Or))
         Skip g
      else
         Render (g)
         Update (Or)
      end if
   end foreach
end function
```

Con este código en mente, y sin entrar en más detalle, mostremos las cuestiones que se plantean:

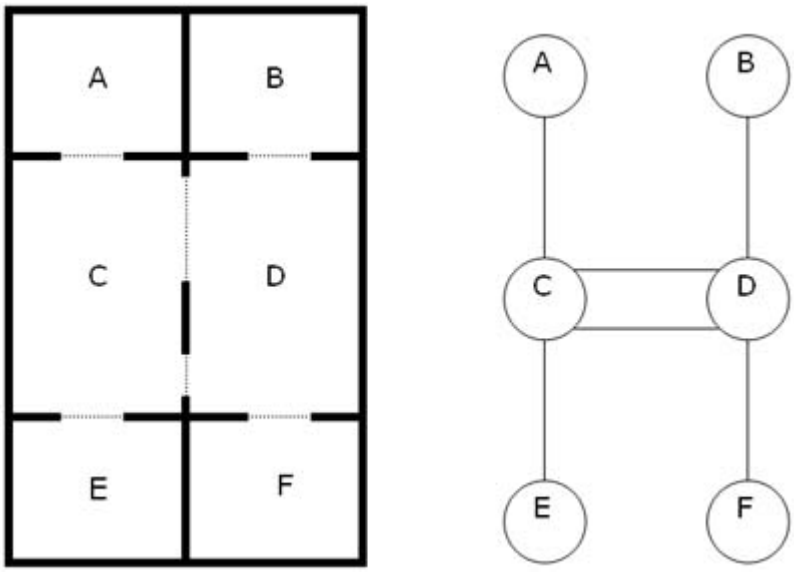
- El desarrollo de la función IsOcludded
- La estructura de datos que representa Or
- Rápida actualicación de Or cuando sea necesario

Portal culling

Esta técnica divide el mundo en celdas (por ejemplo, habitaciones) y éstas son conectadas mediante portales (por ejemplo, ventanas, puertas). Un grafo de adyacencias describe las conexiones.

La estrategia que sigue es la siguiente. En primer lugar, pintamos la celda donde nos encontramos. Para cada portal de la celda, comprobamos si se encuentra en el *frustrum* de visión. Si la respuesta es afirmativa, sabemos que podemos ver otra celda en el otro lado del portal y actuamos con recursividad. Si no es así, quiere decir que no podemos ver más celdas conectadas a este portal y el algoritmo acaba.

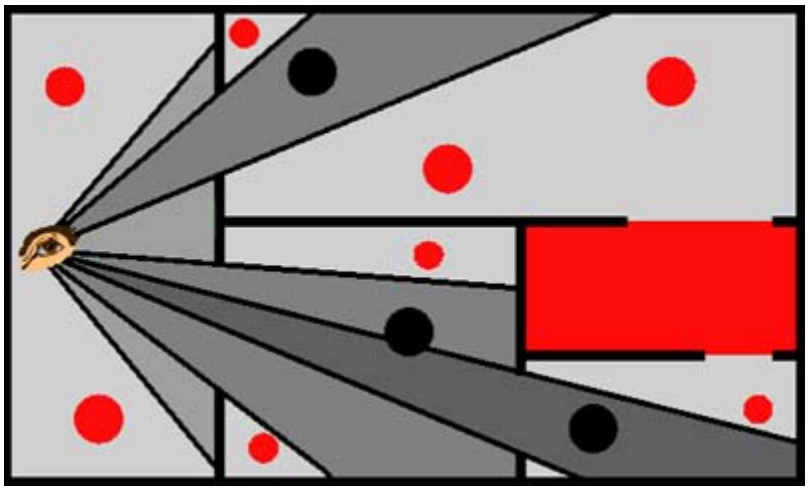
La siguiente imagen muestra un escenario de ejemplo y el grafo de adyacencias que origina.



Grafo de adyacencias para el algoritmo de *portal culling*

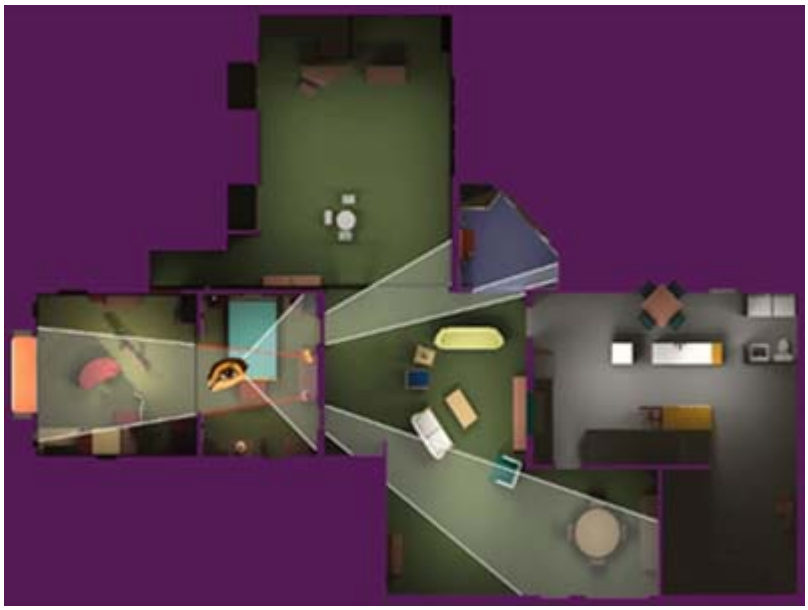
En esta otra ilustración, se muestra el proceso mediante el cual *portal culling* determina qué objetos pintar y sobre qué portales hacer recursividad. Como podemos apreciar, empezamos pintando los objetos que aparecen dentro de nuestro *frustum* de visión. Proseguimos de forma recursiva por cada una de las habitaciones a las que podemos llegar, siguiendo los portales que también se encuentran dentro del área de la zona de visión.

De color rojo tenemos los objetos que no serán pintados y las habitaciones que no consultaremos. En color gris, tenemos los portales sobre los cuales nos movemos con mayor intensidad según el índice de recursividad. Por último, de color negro, aparecen los objetos que serán enviados a *renderizar*.



Proceso del dibujado con *portal culling*

Para acabar, enseñaremos un escenario 3D complejo en el que podremos ver, en primer lugar, el tratamiento de portales que tendrá lugar y, en una segunda imagen, el resultado final que se obtiene. Prestemos especial atención a las funciones que tienen las puertas, ventanas y espejos como portales.



Vista área de los portales

Resultado del *renderizado*

3.12. Shaders

Un *shader* es un programa para la tarjeta gráfica, un conjunto de instrucciones escritas en un lenguaje de programación propio (*shading lenguaje*), capaces de ser ejecutadas por la GPU (el procesador gráfico).

Su uso esta vinculado a tres tipos de tareas:

- Aplicar efectos de representación visual para conseguir un mayor realismo.

- Delegar tarea de procesado gráfico a la tarjeta gráfica en lugar de realizarla vía software.

- Otros usos, fuera del contexto gráfico, como puede ser la detección de colisiones (filosofía GPGPU, *general purpose gaphics process unit*).

Los lenguajes más comunes para la programación de *shaders* son:

- **GLSL** (OpenGL Shading Language): usado con OpenGL y libre.
- **Cg** (C for graphics): propiedad de la empresa Nvidia.
- **HLSL** (*high level shading languaje*): usado con DirectX y propiedad de Microsoft.

Según el momento en que se aplica el código a la representación a lo largo del *pipeline* de dibujado, se diferencian 3 tipos de *shaders*:

- ***Vertex shaders***: actúa sobre las coordenadas, color, textura, etc. de un vértice.
- ***Geometry shaders***: es capaz de generar nuevas primitivas dinámicamtente.
- ***Pixel shaders***: actúa sobre el color de cada píxel (fragmento para ser mas preciso).

A continuación, mostraremos algunos ejemplos de *shaders*.

3.12.1. *Toon shader*

Toon shader es un programa que permite simular gráficos con un aspecto de dibujos animados. La implementación consiste en los siguientes pasos:

- *Vertex shader*: calcula la intensidad que deberá tener el color a partir de la normal y la posición del observador.
- *Fragment shader*: discretiza el valor intensidad en un rango de tres o cuatro valores de color.

Ejemplo de *toon shader* aplicado sobre el modelo de una tetera

A modo de ejemplo, veamos está implementación en GLSL y la visualización que obtenemos.

El código del *vertex shader* es el siguiente:

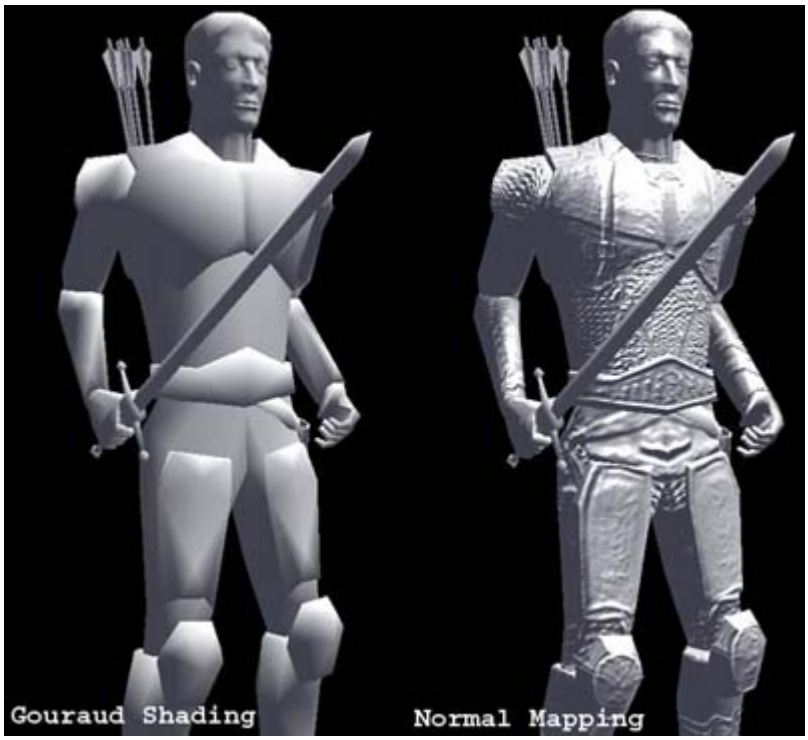
```
varying vec3 lightDir,normal;

void main()
{
   lightDir=normalize(vec3(gl_LightSource[0].position));
   normal = gl_NormalMatrix * gl_Normal;
   gl_Position = ftransform();
}
```

Y el *fragment shader* es:

```
varying vec3 lightDir,normal;

void main()
{
   float intensity;
   vec4 color;
   vec3 n = normalize(normal);

   intensity = dot(lightDir,n);
   if (intensity > 0.95)
      color = vec4(1.0,0.5,0.5,1.0);
   else if (intensity > 0.5)
      color = vec4(0.6,0.3,0.3,1.0);
   else if (intensity > 0.25)
      color = vec4(0.4,0.2,0.2,1.0);
   else
      color = vec4(0.2,0.1,0.1,1.0);
   gl_FragColor = color;
}
```

3.12.2. *Normal mapping*

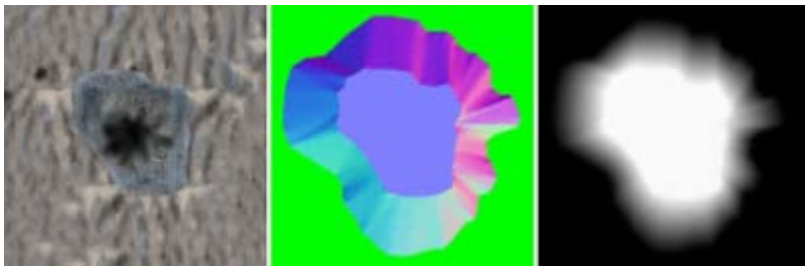
El *normal mapping* es una técnica mediante la cual podemos similar relieves en polígonos planos a partir de unos ficheros llamados *normal maps* (mapas de normales). Con una textura más y sin añadir geometría (vértices), gracias al *shader* en cuestión, somos capaces de conseguir resultados como el siguiente.



Como puede apreciarse, la diferencia es notoria, y conseguimos un realismo mucho mayor a un coste negligible realizado por la tarjeta gráfica. Veamos, con un par de ejemplos, el aspecto de un *normal map*.

3.12.3. *Parallax mapping*

Parallax mapping es una técnica que, además de un *normal map*, cuenta con un mapa de alturas. A partir de éste podemos modificar las coordinadas de textura y simular, por ejemplo, grandes concavidades (inexistentes geométricamente hablando).



De izquierda a derecha: textura original, *normal map* y mapa de alturas

Como veremos en las siguientes imágenes, extraidas del videojuego *F.E.A.R.* desarrollado por Monolith Productions y distribuido por Vivendi Universal, la técnica del *parallax mapping* resulta bastante efectiva. Sin embargo, si el punto de vista está situado perpendicularmente al plano sobre el cual tiene lugar el efecto, puede apreciarse "la trampa".

Parallax mapping en *F.E.A.R*

3.12.4. *Glow*

Este efecto permite simular el efecto *glow*. En este caso, la tarea del *shader* radica en realizar un *motion blur* en dirección horizontal y vertical en aquellas zonas donde interese. Veamos aquí una escena donde predomina el uso de esta técnica.

Efecto GLOW en paredes y paneles

Bibliografía

Coorg, S.; Teller, S. (1996). "A spatially and temporally coherent object space visibility algorithm". *Technical Report TM* (núm. 546).

Jacobs, J. (2007). *Game Programming Gems*. Charles River Media.

Luebke, D.; Georges, C. (1995). "Portals and mirrors: Simple, fast evaluation of potentially visible sets". En: *Proceedings of the 1995 symposium on Interactive 3D Graphics* (pág. 105).

Naylor, B. (1993). "Constructing good partitioning tres". *Graphics Interface* (pág. 181-191). Disponible en: http://www.graphicsinterface.org/pre1996/93-Naylor.pdf

Richard, S.; Wright, Jr.; Lipchak, Benjamin (2005). *OpenGL SuperBible*. Anaya Multimedia

Rost, R. J. (2006). *OpenGL Shading Language*. Boston, MA: Addison-Wesley.

Watt, A.; Policarpo, F. (2001). *3D Games: Real-Time Rendering and* Software *Technology* (2 vol.). Boston, MA: Addison Wesley.

Wright R. S.; Lipchak, Benjamin (2005). *OpenGL SuperBible*. Madrid: Anaya Multimedia.

Zhang, H.; Manocha, D.; Hudson, T.; Hoff III, K. (1997, agosto). "Visibility culling using hierarchical occlusion maps". En: T. Whitted (ed.). *Computer Graphics (SIGGRAPH '97 Proceedings)* (vol. 24, pág. 77-88).