Introducción a la Programación Gráfica con OpenGL

Oscar García – Alex Guevara

Escola Tècnica Superior d'Enginyeria Electrònica i Informàtica La Salle

Enero 2004

Índice

Introducción

OpenGL es una librería gráfica escrita originalmente en C que permite la manipulación de gráficos 3D a todos los niveles. Esta librería se concibió para programar en maquinas nativas Silicon Graphics bajo el nombre de GL (**Graphics Library**). Posteriormente se considero la posibilidad de extenderla a cualquier tipo de plataforma y asegurar así su portabilidad y extensibilidad de uso con lo que se llego al termino **Open Graphics Library**, es decir, OpenGL.

La librería se ejecuta a la par con nuestro programa independientemente de la capacidad gráfica de la maquina que usamos. Esto significa que la ejecución se dará por software a no ser que contemos con hardware gráfico especifico en nuestra máquina. Si contamos con tarjetas aceleradoras de vídeo, tecnología MMX, aceleradoras 3D, pipelines gráficos implementados en placa, etc. por supuesto gozaremos de una ejecución muchísimo mas rápida en tiempo real.

Así esta librería puede usarse bajo todo tipo de sistemas operativos e incluso usando una gran variedad de lenguajes de programación. Podemos encontrar variantes de OpenGL para Windows 95/NT, Unix, Linux, Iris, Solaris, Delphi, Java e incluso Visual Basic. No obstante, su uso mas extenso suele ser el lenguaje C o C++.

Por supuesto no podemos desligar el mundo de los gráficos de OpenGL y por tanto se intentará que este sea un **curso de "gráficos usando OpenGL"**, mas que una larga mención de funciones y constantes integradas en esta librería.

Trabajaremos con **C** y por tanto nos referiremos siempre a ejemplos codificados según este lenguaje. Simplemente necesitaremos las librerias adecuadas y un compilador cualquiera de Ansi C, es decir, el estándar de este lenguaje.

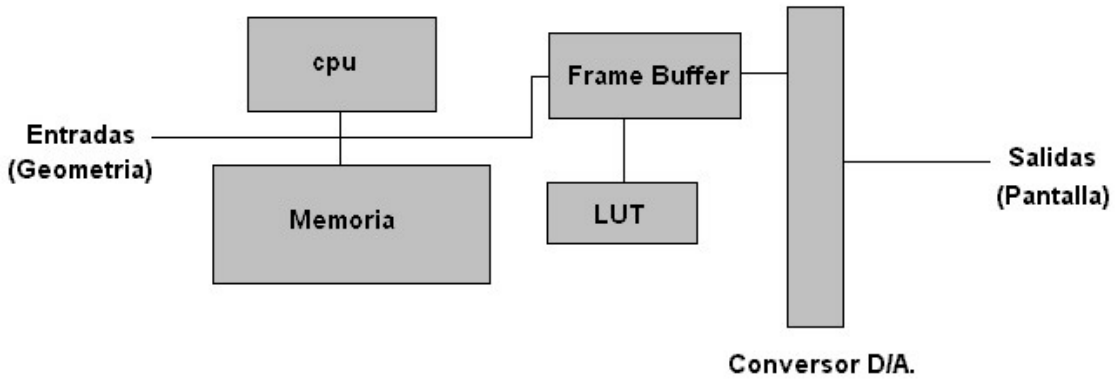
Algo de álgebra lineal también hará falta para entender que es lo que ocurre internamente y como podemos operar con nuestra geometría para lograr los efectos deseados.

1. Conceptos previos.

1.2. Sistemas gráficos. Dispositivos y elementos.

Tal como comentamos en el capítulo de introducción, es inviable hablar de cómo programar con OpenGL sin, a la par, mencionar determinados conceptos. Por tanto, primero debemos entender que es un "pipeline" gráfico, que elementos lo configuran y cuales son sus entradas y salidas.

Un sistema gráfico típico se compone de los siguientes elementos físicos:



Estas son las características generales de cada uno de los elementos:

- **Entradas :** todo aquello que nuestro programa ha calculado y desea dibujar. En definitiva es el "nuevo estado" de nuestro mundo tras algún evento que lo ha hecho cambiar como por ejemplo que la cámara se haya movido o alejado de la escena.
- **Procesador (CPU):** máximo administrador del sistema, se encargará de gestionar la comunicación entre todos los módulos. Realizara operaciones según se le pida con ayuda de una/s ALU/s (Unidades aritméticas) y consultara la memoria cuando le sea necesario. En sistemas dedicados, y por tanto especializados en gráficos, podemos tener diversas CPU's trabajando en paralelo para asegurar un buen rendimiento en tiempo real (Calcular y dibujar a la vez).
- **Memoria :** elemento indispensable y bastante obvio como el anterior. En nuestro caso nos interesara una cierta franja de memoria, el "frame buffer".
- **Frame Buffer :** Zona de memoria destinada a almacenar todo aquello que debe ser dibujado. Antes de presentar la información por pantalla, esta se recibe en el frame buffer. Por tanto, nuestro programa OpenGL escribe en esta área de memoria y automáticamente envía su contenido a la pantalla después.
- **Look Up Table (LUT):** esta "tabla" contiene todos los colores que tenemos disponibles en nuestro sistema. A algunos os parecerá quizá más familiar el termino "paleta" para referiros a la LUT. En sistemas "indexados" cada color tiene un identificador en la tabla y puedes referirte a él usándolo en tu programa.

Así si deseamos dibujar un polígono de color rojo, modificaremos su atributo de color dándole a este el valor del identificador que tiene el color rojo en la LUT. Generalmente no se utiliza ya que no se trabaja en color indexado, sino con color real (RGB)

- **Conversor D/A:** la información contenida en el frame buffer a nivel de bit es digital y por tanto debe convertirse a su homónimo analógico para poder ser procesada por un CRT (Tubo de rayos catódicos) y proyectada en la pantalla. No profundizaremos mas en este tema pues no es el objetivo analizar el funcionamiento de un monitor.
- **Salidas :** tras el conversor ya disponemos de información analógica para ser visualizada en nuestra pantalla.

Antes de continuar aclarar el termino **píxel** (picture element). Un píxel es la unidad mínima de pantalla y los encontramos dispuestos en filas en cualquier monitor o televisor de manera que el conglomerado de pixeles con sus colores asociados da lugar a la imagen.

El frame buffer se caracteriza por su **resolución** y por su **profundidad**.

La resolución viene dada por el producto *ancho* x *alto*, es decir, el número de filas multiplicado por el número de columnas análogamente a las resoluciones que nuestro monitor puede tener (640 x 480, 800 x 600 ...).

La profundidad es el número de bits que utilizamos para guardar la información de cada píxel. Este número dependerá de la cantidad de colores que deseemos mostrar en nuestra aplicación. Típicamente si queremos "**color real**" necesitaremos ser capaces de mostrar 16,7 millones de colores simultáneamente que es la capacidad aproximada de nuestro sistema visual. En este caso y suponiendo una resolución de 800 x 600 pixeles en pantalla necesitaremos:

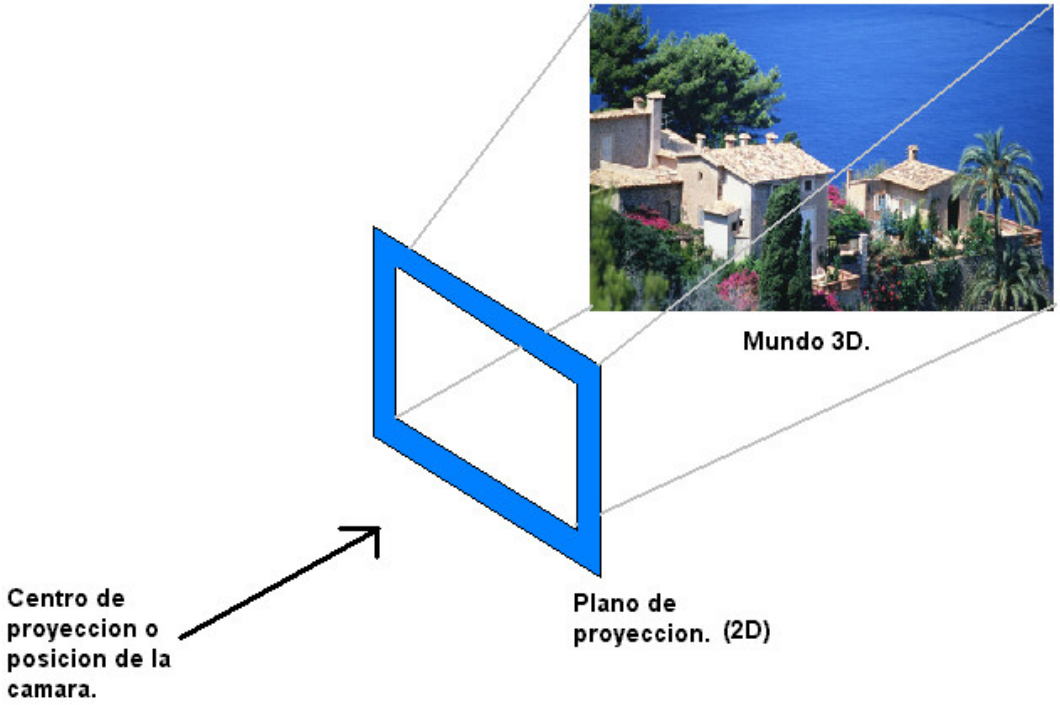
```
800 píxeles/fila x 600 filas x 24 bits/píxel = 1.37 Megabytes
                de memoria para el frame buffer
```

Dado que color real implica 256 posibles valores de rojo, 256 de verde y 256 de azul por píxel y esto implica un byte/píxel para cada una de estas componentes, es decir, 3 bytes por píxel.

La librería OpenGL, por si sola, se encargará de reservar este espacio de memoria, sin que nosotros tengamos que ocuparnos de ello. Nuestra librería o **API** (Application Programming Interface) , en este caso OpenGL, contactara directamente con los elementos de la figura anterior a medida que lo crea necesario. Por tanto, esto nos será totalmente transparente y no deberá preocuparnos.

1.3. El modelo "cámara sintética".

OpenGL utiliza este modelo semántico para interpretar una escena que debe ser dibujada. Básicamente se trata de imaginar un objeto situado en un determinado lugar y filmado por una cámara, tan sencillo como esto. Veamos el siguiente diagrama:



Contamos con un "mundo 3D" que estamos observando desde una determinada posición. Podéis pensar en el observador como vosotros mismos mirando hacia vuestro mundo virtual o bien como una cámara que lo esta filmando. Ese punto es el **centro de proyección** tal y como se observa en la figura. Evidentemente el mundo es tridimensional pero su proyección en un plano (**plano de proyección**) es bidimensional. Este plano es nuestro frame buffer antes de dibujar o bien la pantalla del monitor después de haberlo hecho.

Por tanto, queda claro que pasamos de coordenadas del mundo en 3D a coordenadas de pantalla en 2D y por lo tanto necesitamos **proyectar**.

El modelo "**cámara sintética**" se compone pues de los siguientes elementos:

- Necesitamos **luces** que iluminen nuestro mundo 3D. Para ello será necesario que especifiquemos sus **localizaciones**, sus **intensidades** y sus **colores**.
- Necesitamos una **cámara** que "filme" nuestro mundo virtual y lo muestre por pantalla. Esta cámara es nuestro "punto de vista" del mundo a cada momento. Se caracteriza por su **posición** en el mundo, su **orientación** y su **apertura** o "**campo visual**". El campo visual es la "cantidad" de mundo que la cámara alcanza a ver. Mirad la figura anterior e imaginad que acercamos el plano de

proyección a nuestro mundo 3D. En este caso estaremos disminuyendo el campo visual dado que será menos "porción" del mundo la que se proyectará.

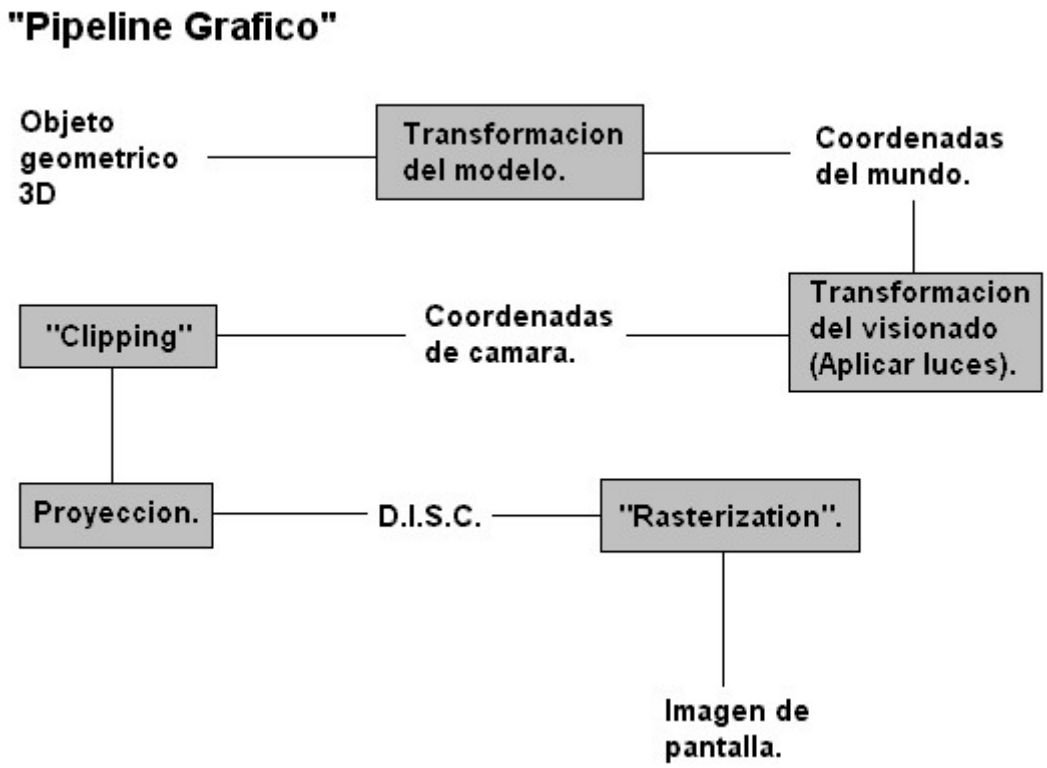
- Por ultimo necesitamos **objetos** que formen parte de nuestro mundo y que precisamente serán los "filmados" por nuestra cámara. Se caracterizaran por sus atributos de color, material, grado de transparencia, etc.

Es muy importante que notéis la independencia que todos estos elementos tienen entre sí. La cámara es independiente de los objetos puesto que estos están en una posición y ella en otra. Manejaremos los objetos (**Geometría**) por un lado y la cámara por otro de manera que los primeros "actuaran" en nuestro mundo y la segunda filmará desde una determinada posición con un determinado ángulo.

1.4. Arquitecturas gráficas

Hablemos ahora del **pipeline gráfico**, elemento que marcara la pauta de actuación para OpenGL e incluso para vuestras manos en el momento de programar.

La geometría que deseáis dibujar será la entrada de vuestro pipeline gráfico y como salida tendréis una imagen en pantalla. Pero lo que ocurre con todos esos puntos, vectores y polígonos desde que entran hasta que salen lo vemos en la siguiente figura:



El punto de entrada de nuestra geometría es el superior izquierdo y a partir de ahí solo tenemos que seguir las líneas que conectan los diferentes módulos.

Analicemos cada uno de los módulos:

- **Objeto geométrico:** nuestro mundo se compone de puntos, líneas, polígonos, etc. en definitiva, primitivas. Inicialmente estos objetos tienen unos atributos que nosotros fijamos pero pueden ser movibles o fijos, deformables o rígidos, y, por tanto, debemos ser capaces de trasladarlos, rotarlos y escalarlos antes de dibujarlos en pantalla.
- **Transformación del modelo:** este módulo es el encargado de trasladar, rotar, escalar e incluso torcer cualquier objeto para que sea dibujado en pantalla tal y como debe estar dispuesto en el mundo. OpenGL realiza estas funciones multiplicando a nuestra geometría (vértices) por varias matrices, cada una de las cuales implementa un proceso (rotar, trasladar...) Nosotros usaremos las funciones de OpenGL para “crear” estas matrices y "operarlas" con nuestra geometría.

- **Coordenadas del mundo:** tras haber transformado nuestros vértices, ya sabemos las posiciones de todos los objetos en nuestro mundo. No son relativas a la cámara, recordad que precisamente he incidido en la independencia cámara/mundo. Son posiciones referidas a un sistema de coordenadas que definiremos única y exclusivamente para el mundo que estamos creando.
- **Transformación del visionado:** ahora es cuando necesitamos saber "como" se ven esos objetos, ya posicionados correctamente en el mundo, desde nuestra cámara. Por tanto, los "iluminamos" para que sean visibles y tomamos sus posiciones tal y como se ven desde la cámara.
- **Coordenadas de cámara**: tras aplicar luces ya sabemos cuales son las coordenadas de todos los objetos respecto de nuestra cámara, es decir, como los vemos nosotros desde nuestra posición en el mundo.
- **Clipping:** el clipping consiste en recortar (ocultar) todo aquello que "está" pero "no se ve" desde la cámara. Es decir, si todo aquello que la camara no puede ver, por que no entra en su ángulo de visión, se elimina, a ese proceso se denomina Clipping.
- **Proyección :** Pasamos de coordenadas 3D del mundo a coordenadas 2D de nuestro plano de proyección.
- **D.I.S.C.** (Device Independent Screen Coordinates): tras proyectar tenemos lo que se llaman "coordenadas de pantalla independientes de dispositivo". Las coordenadas que tenemos calculadas en este momento todavía no se han asociado a ningún tipo de pantalla o monitor. En este punto del pipeline no sabemos si nuestro monitor es de 15 pulgadas y de resolución 800 x 600 o si es de 19 pulgadas y de resolución 1024 x 1480. De hecho esto no tenemos porque controlarlo nosotros, nos será transparente. Se trata de asociar la imagen recortada 2D que se encuentra en el frame buffer con los píxeles de la pantalla. Según la resolución de pantalla sea mayor o menor, un punto de nuestro mundo ocupara mas o menos píxeles.
- **Rasterización** : este proceso se conoce también como "scan conversion". Finalmente asociamos todos nuestros puntos a píxeles en pantalla. Tras esto solo falta iluminar los fósforos de nuestra pantalla con energía suficiente para que veamos lo que esperamos. Evidentemente esto es un proceso totalmente físico llevado a cabo en el tubo de rayos catódicos del monitor. No actuaremos pues sobre él.
- **Imagen de pantalla:** Aquí termina nuestro proceso. Ya tenemos la imagen de lo que nuestra cámara ve en el monitor.

El pipeline gráfico puede implementarse vía software o hardware. En maquinas dedicadas, por ejemplo Silicon Graphics, todos los módulos están construidos en la placa madre de manera que el sistema es muy rápido. En sistemas mas convencionales como PC o Mac, todo se realiza vía software y por tanto es mas lento, siempre y cuando no tengamos instalada una tarjeta acceleradora gráfica.

De todas formas lo interesante de OpenGL es que funcionará independientemente de la implementación del pipeline. No tendréis que cambiar el código si cambiáis de plataforma, funcionara igual. Lo único es que según el sistema conseguiréis más o menos velocidad.

2. Interfície OpenGL. Primitivas y atributos.

Empecemos con algunos comandos básicos de OpenGL. OpenGL tiene sus propios tipos en cuanto a **variables** se refiere. Así aunque podemos usar los típicos (`int, float, double`), nos acostumbraremos a los definidos por esta librería (`GLint, GLfloat, GLdouble`) Como veis son los mismos pero con el prefijo "**GL**" delante. Se declaran exactamente igual que cualquier variable en C y tienen casi las mismas propiedades. Usémoslos porque el funcionamiento general del sistema será más óptimo.

Las **funciones** OpenGL empiezan con el prefijo "**gl**", en minúsculas. Veamos un ejemplo. Supongamos que queremos crear un vértice, es decir, un punto:

Usaremos:

```
glVertex3f( 0.0 , 0.0 , 0.0 );
```

o:

```
GLfloat vértice[3] = { 0.0, 0.0, 0.0 };

glVertexfv( vértice );
```

Ambas funciones crean un vértice situado en el origen de coordenadas del mundo, es decir, `x = y = z = 0.0`. En el primer caso el nombre de la función termina en "3f" (3 floats). Esto significa que vamos a especificar el vértice con 3 valores o variables de tipo real, o sea float. En cambio en el segundo caso tenemos "fv" (float vector). Estamos indicando a OpenGL que el vértice lo daremos mediante un array/vector de floats. Precisamente este es el array que defino justo antes de llamar a la función.

Trabajamos en 3D y especificamos las coordenadas del vértice en este orden: X, Y, Z. Si deseamos trabajar en 2D solo tenemos que hacer una coordenada igual a 0.0, normalmente la Z.

Ya que estamos puestos vamos a definir nuestro primer polígono, un triángulo. OpenGL tiene varios tipos definidos de manera que nos facilita la creación de polígonos simples. Vamos allá:

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

Este código crea un triángulo situado en el plano XY ya que observareis que los valores de Z son todos 0.0 para los tres vértices que lo forman. Sus tres vértices se encuentran en las posiciones (-1.0, 0.0), (1.0, 0.0) y (0.0, 1.0) según la forma (X, Y).

Un polígono se encapsula entre las funciones **glBegin** y **glEnd**. El parámetro que recibe la primera sirve para decirle a OpenGL que tipo de polígono deseamos crear, en nuestro caso un triángulo. **GL_TRIANGLES** es una constante ya definida en la librería.

Por claridad es conveniente tabular (indentar) el código entre **glBegin** y **glEnd** tal y como veis en el ejemplo. Cualquier programa OpenGL que examinéis seguirá esta convención si esta bien estructurado.

Vamos a definir alguno de los atributos de nuestro triángulo, por ejemplo su color. Usamos:

```
glColor3f( 0.5, 0.5, 0.5 );
```

donde los tres valores (floats) que se le pasan a la función glColor son por orden, la cantidad de rojo (**Red**) que deseamos, la cantidad de verde (**Green**) y la cantidad de azul (**Blue**). Es el llamado sistema **RGB** que muchos conoceréis sobradamente. Aplicando una cantidad de cada color conseguimos el tono deseado (Teoría **aditiva** del color). Los valores de cada color deben estar entre 0.0 (No aplicar ese color) y 1.0 (Aplicar ese color en su máxima intensidad). Por tanto:

```
glColor3f( 0.0, 0.0, 0.0 ); //se corresponde con el color NEGRO
```

mientras que...

```
glColor3f( 1.0, 1.0, 1.0 ); //se corresponde con el BLANCO
```

y de esta manera si queremos definir un triángulo blanco obraremos así:

```
glColor3f( 1.0, 1.0, 1.0 );
glBegin( GL_TRIANGLES );
    glVertex3f( -1.0, 0.0, 0.0 );
    glVertex3f( 1.0, 0.0, 0.0 );
    glVertex3f( 0.0, 1.0, 0.0 );
glEnd( );
```

...de manera que primero especificamos el color y TODO lo que dibujemos a partir de este momento será de ese color, en este caso el triángulo.

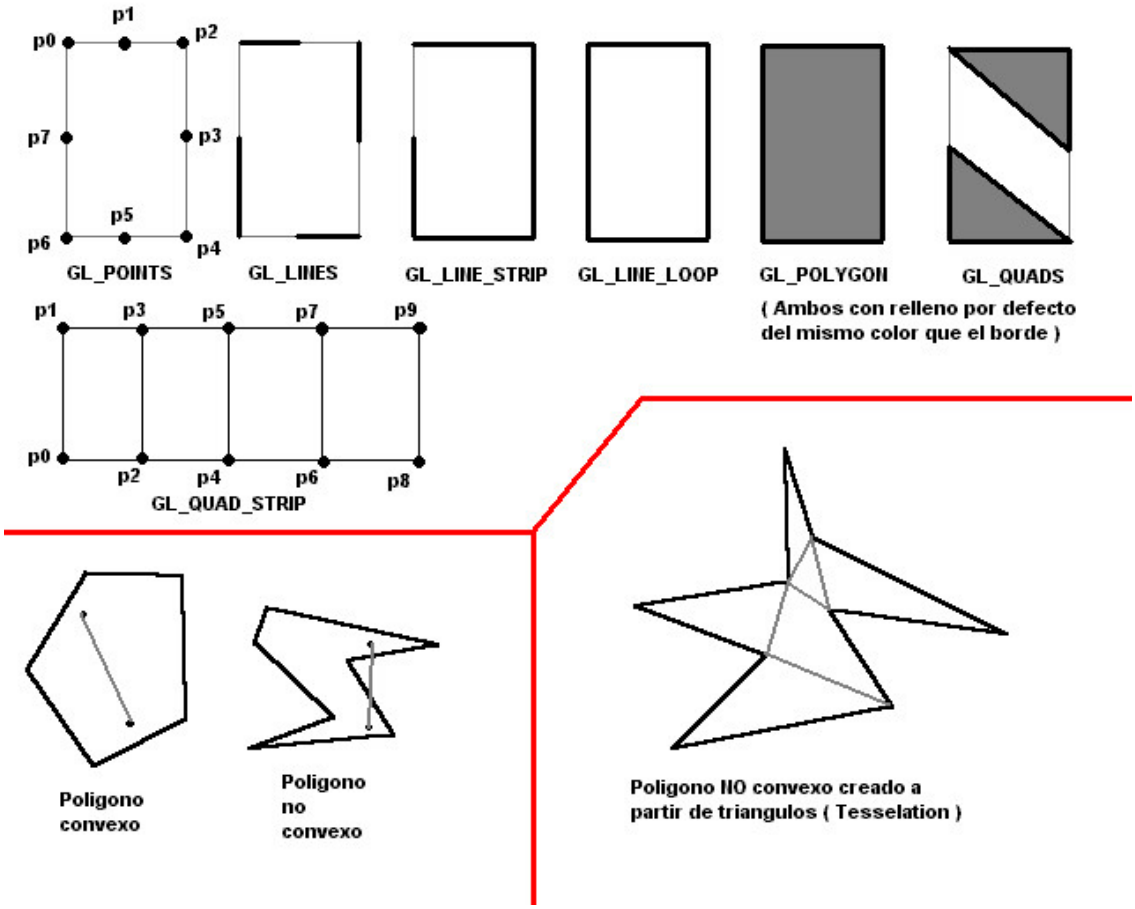
Al ejecutar cada función glVertex, el vértice en cuestión "entra" en el pipeline y se traslada a la posición que hemos especificado para él. Entonces se "**mapea**" en el frame buffer (representación en memoria del plano de proyección 2D) pasando posteriormente a la pantalla de nuestro monitor. Así en este caso tenemos 3 vértices que sucesivamente entran en el pipeline uno detrás de otro.

Algo muy importante que debéis tener en cuenta. Cuando definàis un polígono a base de sus vértices deberéis seguir un orden concreto. Si no lo hacéis, OpenGL no asegura que la representación en pantalla sea la correcta. La convención es crear los vértices siguiendo el polígono según el s**entido antihorario** de las manecillas del reloj. OpenGL supone la cara "a dibujar" del polígono como la que se define de esta manera.

Respecto a las constantes que podemos usar en la función **glBegin** tenemos entre otras:

- **GL_POINTS :** para que todos los vértices indicados entre ambas funciones se dibujen por separado a modo de puntos "libres".
- **GL_LINES :** cada dos vértices definidos, se traza automáticamente una línea que los une.
- **GL_POLYGON :** se unen todos los vértices formando un polígono.
- **GL_QUADS :** cada 4 vértices se unen para formar un cuadrilátero.
- **GL_TRIANGLES :** cada 3 vértices se unen para formar un triángulo.
- **GL_TRIANGLE_STRIP :** crea un triángulo con los 3 primeros vértices. entonces sucesivamente crea un nuevo triángulo unido al anterior usando los dos últimos vértices que se han definido y el actual.
- **GL_QUAD_STRIP :** igual que el anterior pero con cuadriláteros.

De hecho los tipos mas usados son los tres primeros mientras que el resto pueden sernos de ayuda en casos determinados. Veamos la figura:



En la figura se observa como usando diferentes tipos (**GL_POINTS**, **GL_LINES**, etc.) conseguimos diferentes objetos en pantalla dados unos vértices

creados según el orden que veis en la figura (primero p0, luego p1, etc.) para los 6 rectángulos de la primera fila. Es decir:

```
glBegin( TIPO );
    glVertex3f( p0x, p0y, p0z );
    glVertex3f( p1x, p1y, p1z );
    glVertex3f( p2x, p2y, p2z );
    glVertex3f( p7x, p7y, p7z );
glEnd();
```

Lo que OpenGL dibujará según el caso es lo que se encuentra en color negro en la figura, no el trazado gris que solo lo he puesto como referencia.

En la segunda fila de la figura vemos como crear sucesivos "polígonos de cuatro lados" (no tienen porque ser cuadrados o rectángulos, eso depende de en qué posiciones situemos los vértices) con un lado en común de dos a dos. Hay mas variantes que podréis encontrar en las especificaciones de OpenGL aunque no suelen usarse mas que para aplicaciones muy concretas.

Los polígonos especificados por una serie de puntos deben ser **convexos** para su correcto trazado, y coloreado, por parte de OpenGL. Que quiere decir esto?, pues bien se define a un polígono como convexo si cogiendo dos puntos cualesquiera de su interior y trazando la línea que los une, todos los puntos que pertenecen a la línea se encuentran dentro del polígono. En la anterior figura tenéis un ejemplo de esto.

Por tanto asegurad siempre que definís polígonos convexos, de lo contrario los resultados son simplemente impredecibles y pocas veces acertados. Entonces, ¿como dibujamos polígonos que no sean convexos? Lo que normalmente se hace es dibujar un polígono complejo, convexo o no, a base de muchos triángulos. Es lo que se denomina "**Tesselation**". Todo polígono puede descomponerse en triángulos y, por tanto, a partir de estos somos capaces de dibujar cualquier cosa. También lo tenéis en la figura explicitado gráficamente.

Usando GLU o GLUT podemos crear con una sola llamada objetos mas complejos como es el caso de una esfera:

- Usando GLU con:

```
void gluSphere( GLUquadricObj *qobj, GLdouble radius, GLint slices,
GLint stacks );
```

- Usando GLUT con:

```
void glutSolidSphere(GLdouble radius, GLint slices, GLint stacks);
```

En cuanto programemos lo veréis mas claro pero de hecho no dejan de ser funciones en C de las de siempre. Con **radius** definimos el radio que queremos para la esfera, con **slices** y **stacks** la "partimos" en porciones como si de un globo terráqueo se tratara (paralelos y meridianos) de manera que a mas particiones, mas calidad tendrá ya que OpenGL la aproximara por mas polígonos. En cuanto a ***qobj** en la primera función, se trata de definir primero un objeto de tipo quadric con otra función de GLU (gluNewQuadric) para después asociar la esfera al susodicho objeto. No os preocupéis

porque no usaremos demasiado esta filosofía. De momento quedaros con la idea que es lo que me interesa.

A alguien se le ocurrirá probablemente pensar que en estas funciones no se define la posición de la esfera en el mundo. Es cierto y por lo tanto previamente a crearla, y por tanto dibujarla, deberemos especificar **donde** queremos que OpenGL la dibuje. Lo podemos especificar con:

glTranslatef(posX, posY, posZ): función que nos traslada posX unidades en el eje X, posY en el eje Y ... y si tras esto llamamos a glutSolidSphere, ya tendremos a nuestra esfera dibujada en el punto que deseamos.

Bueno hablemos un poco de los **atributos**. Una cosa es el tipo de la primitiva a dibujar y otra es **como** se dibuja, es decir, con que color de borde, con que color de relleno, con que ancho de borde, con que ancho por punto, etc. Estos son algunos de los atributos que pueden especificarse. Algo importante, en OpenGL los atributos no se pueden definir explícitamente para cada objeto sino que lo usual es definir unos ciertos atributos generales de manera que todo lo que se dibuje a partir de ese momento seguirá esas especificaciones. Si deseamos cambiar los atributos deberemos hacerlo en otra parte del programa de manera que la forma de dibujar será diferente a partir de esa línea. Veamos algunas funciones de ejemplo referidas a atributos:

- **glClearColor(0.0, 0.0, 0.0, 0.0):** esta es algo genérica y se refiere al color con el cual debe de "resetearse" el frame buffer cada vez que redibujemos toda la escena de nuevo. En este caso el "**fondo**" de nuestra ventana será como el fijado por esta función en el frame buffer, de color negro. El cuarto parámetro de la función se refiere al valor de "**alpha**" en cuanto al color. Veremos mas adelante que el valor de alpha permite variar el **grado de transparencia** de un objeto.
- **glColor3f(1.0, 0.0, 0.0):** En este caso definimos que todo lo que se dibuje desde este momento será de color rojo. Recordad que el orden de parámetros es Red, Green, Blue (RGB).
- **glPointSize(2.0):** con esta llamada definimos que cada uno de nuestros puntos deberá tener un grosor de dos pixeles en el momento de ser trasladado a pantalla.
- **glNormal3f(1.0, 0.0, 0.0);** cuando operemos con **luces** veremos que para cada cara de un polígono hay que asociar un vector o **normal**. Esta función define como normal a partir de este momento a un vector definido desde el origen con dirección positiva de las X. El orden de los parámetros es X, Y, Z.
- **glMaterialfv(GL_FRONT, GL_DIFFUSE, blanco);** por ultimo y también referido al tema de las **luces**. Cada objeto será de un material diferente de manera que los reflejos que en el se produzcan debidos a las luces de nuestra escena variaran según su rugosidad, transparencia, capacidad de reflexión ... en este caso definimos que todas las caras principales (FRONT, son las caras "que se ven" del polígono) de los objetos dibujados a partir de ahora tendrán una componente difusa de color blanco (asumiendo que el parámetro "blanco" es un vector de reales que define este color). La componente difusa de un material define que color tiene la luz que este propaga en todas las direcciones cuando sobre el incide un rayo luminoso. En este caso se vería luz blanca emanando del objeto/s dibujados a partir de ahora. Por supuesto antes hay que definir luces en

nuestra escena y activarlas. Lo veremos mas adelante en el apartado de iluminación.

En cuanto a los **colores**, trabajaremos con la convención RGBA, es decir, grado de rojo, verde, azul y transparencia. Los valores para cada componente se encontraran siempre dentro del intervalo [0, 1]. Si nos pasamos o nos quedamos cortos numéricamente hablando, OpenGL adoptara como valor 1 o 0 según sea el caso. Es decir, que redondea automáticamente aunque hay que evitarle cálculos innecesarios y prever valores correctos.

En el caso del valor para alpha, 0 significa transparencia total y de aquí podemos usar cualquier valor hasta 1, objeto totalmente opaco.

Que decir del **texto**.....no nos es de mucho interés ahora mismo que lo que estamos deseando es empezar a dibujar polígonos 3D como locos pero comentare un par de cosas. OpenGL permite dos modos de texto: **Stroke Text** y **Raster Text**.
En el primer caso cada letra del alfabeto es una nuevo polígono literalmente construido a partir de primitivas de manera que puede tratarse tal y como si de un objeto cualquiera se tratara. Es pesado pues tenemos que dibujar literalmente cada letra a base de primitivas pero interesante en cuanto a posibilidades pues podremos escalar, rotar, trasladar.....en fin todo lo que nos es posible aplicar a un polígono.
En cuanto al segundo caso tenemos un ejemplo de simplicidad y rapidez. Es el típico alfabeto creado a partir de bloques de bits (**BitMaps**), de manera que tendremos una rejilla sobre la que un 1 significara lleno y un 0 vacío. De esta forma:

```
1 0 0 0 1
1 0 0 0 1
1 0 0 0 1
1 1 1 1 1
1 0 0 0 1
1 0 0 0 1
1 0 0 0 1
```

...se corresponde con la letra H.

Así podremos crearnos fácilmente un alfabeto y posicionar directamente en el frame buffer sendas letras. Para ello es necesario conocer como acceder directamente, al nivel de bit, al frame buffer y esto aun nos tomara algo de tiempo.

3. Transformaciones.

3.1. Coordenadas Homogéneas

Todos estamos acostumbrados a utilizar coordenadas cartesianas para representar los vértices que definen a nuestra geometría. Es decir un punto es algo así:

P = (x, y, z)

y representa una determinada localización en un espacio 3D.

Pero cuando programamos Gráficos hablamos de **puntos** y de **vectores** y pueden confundirse en cuanto a representación. Si entendemos que un vector es una resta entre dos puntos:

Vector v = Punto1 - Punto2 = (x1, y1, z1) - (x2, y2, z2) = (a, b, c)

Obtenemos el mismo aspecto que un punto.

Por otra parte trabajaremos **modelando geometría para luego transformarla**: trasladándola a otra posición, rotándola respecto de un eje, escalándola para cambiar su tamaño, etc. Estas son las llamadas **transformaciones afines/rígidas/lineales**. Dado que operamos usando matrices para efectuar estas transformaciones necesitamos modificarlas ligeramente por dos motivos:

· Para que no alteren de igual forma a un vector y a un punto, lo cual sería incorrecto.
· Para poder efectuar algunas transformaciones afines como la traslación, imposibles de efectuar multiplicando matrices si no se usan coordenadas homogéneas.

Es muy sencillo convertir un vector o un punto **cartesiano** a su representación **homogénea**. De hecho lo que se hace es añadir una nueva coordenada a las típicas XYZ. **Añadimos la componente W** de esta forma:

Punto **P1 = (x1, y1, z1)** en cartesianas es **P1 = (x1, y1, z1, w1)** en homogéneas.

Vector **v = (a, b, c)** en cartesianas es **v = (a, b, c, w)** en homogéneas.

Los valores típicos para la nueva componente son:

· **W = 1**, cuando tratemos con puntos.
· **W = 0**, cuando sean vectores.

Por tanto, el caso anterior queda modificado de la siguiente manera:

Punto **P1 = (x1, y1, z1, 1)** en homogéneas.
Vector **v = (a, b, c, 0)** en homogéneas.

Veremos más adelante en este capítulo como este convenio nos permite operar transformando un punto en otro punto, y un vector en otro vector, sin tener que efectuar una operación diferente en cada caso.

En el caso que **W** sea diferente que 1 ó 0, tendremos que efectuar una sencilla operación para transformar un punto homogéneo en uno cartesiano.

Si tenemos el punto:

$$\mathbf{P1 = (x1, y1, z1, w1)}$$

en homogéneas, entonces en cartesianas el punto es

$$\mathbf{P1 = (x1/w1, y1/w1, z1/w1).}$$

Es decir, que **normalizamos** cada una de las componentes XYZ del punto por su componente W. En el caso de W = 1 no hay que hacer nada pues la división es obvia, pero puede pasar que nos interese variar W y entonces no podremos usar las XYZ hasta haberlas normalizado según os acabo de explicar.

Por lo tanto, podemos decir que:

$$\mathbf{P = (1, 2, 3, 1) = (2, 4, 6, 2) = (5, 10, 15, 5)}$$

Una observación importante: podemos transformar de dos maneras distintas pero totalmente equivalentes. Es exactamente lo mismo transformar **un vértice respecto de un sistema de referencia** que transformar, en orden inverso, **el sistema de referencia dibujando después el vértice en éste**.

Las transformaciones afines se llaman así porque **conservan las lineas**, como podemos ver en la figura:

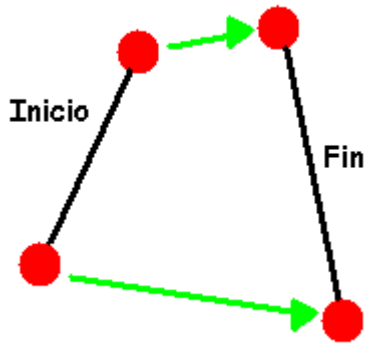


Figura 1. Las transformaciones afines conservan las lineas.

Se observa que tan sólo transformando los vértices y uniéndolos de nuevo obtendremos como resultado final la línea que los une transformada. De esta forma queda claro que sólo tendremos que aplicar transformaciones a los puntos, vértices, de nuestra geometría, uniéndolos después con segmentos rectos.

3.2. Escalar

Gracias a la función de escalado podemos **aumentar/disminuir** un objeto en cuanto a tamaño. Tan sólo tendremos que multiplicar a cada uno de sus vértices por la matriz de escalado, uniéndolos después con líneas tal y como estaban al inicio:

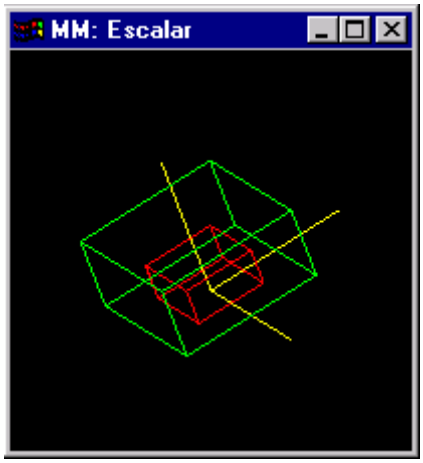


Figura 2. Escalar un cubo.

En el ejemplo se observa el efecto claramente. El cubo original es el de color rojo y como veis está centrado en el origen. El cubo escalado es el de color verde. Es 2 veces mayor que el original al haber efectuado un escalado de 2.0 sobre todas sus componentes. La matriz que se ha aplicado a cada uno de los vértices del cubo la tenéis en la figura 9.

3.3. Trasladar

Esta es precisamente una transformación afín imposible de realizar en cartesianas si no se incluye una suma de matrices. Pero nosotros no queremos sumar, tan sólo multiplicar. Y es que la mayoría de los "pipeline's" gráficos implementados vía hard en aceleradoras 3D o tarjetas de video esperan recibir matrices para concatenarlas y multiplicarlas.

Si intentamos aplicar la transformación que vemos sin usar coordenadas homogéneas, es decir, con un vector y una matriz de 3 x 3, veremos como es imposible hacerlo. Necesitamos “sumar” algo al resultado para lograrlo.

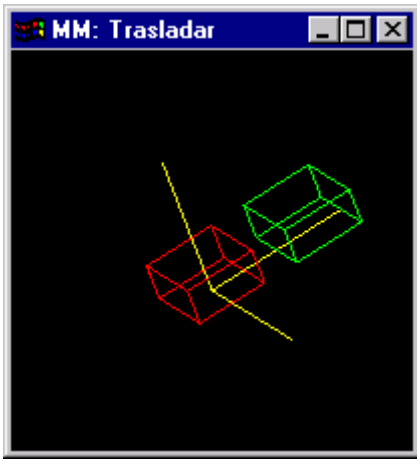


Figura 3. Trasladar un cubo.

Al igual que anteriormente, el cubo original es de color rojo y está centrado en el origen. Al cubo verde se le ha aplicado una traslación de 30 unidades positivas siguiendo la dirección del eje X. Obviamente el tamaño se conserva pues no hemos aplicado ningún escalado. La matriz que implementa esta transformación la tenéis en la figura 9.

3.4. Rotar

La rotación debe realizarse siempre **alrededor de un determinado eje de referencia**. Podemos rotar alrededor del eje X, del eje Y o del eje Z, y según el caso la matriz a aplicar será una o será otra.

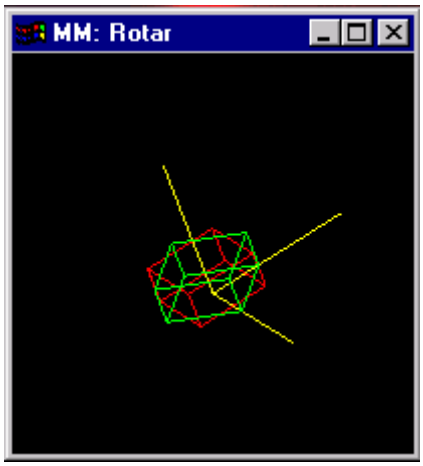


Figura 4. Rotar un cubo.

Seguimos con el mismo criterio que antes. El cubo rojo sigue estático en el origen. El cubo verde tiene exactamente las mismas dimensiones pero se ha rotado 45 grados alrededor del eje vertical, que en este caso es el eje Y y no el Z al que tanto nos hemos acostumbrado desde siempre. La matriz a emplear está en la figura 9.

Respecto al cambio del eje Y con el Z, hemos de pensar que en gráficos siempre se utiliza la Z como unidad de profundidad, y no de altura. Incluso más adelante hablaremos del "Z buffer", o buffer de profundidad de unos políguonos respecto de los otros para que no se solapen (para eliminar las superfícies ocultas)

Otro aspecto a considerar, es el signo del ángulo cuando vamos a efectuar a una rotación. Existe una convención establecida para cuando hablemos de ello:

"***El ángulo de rotación se define como positivo si supone girar en dirección contraria a las manecillas del reloj (CCW-Counter Clockwise) al mirar el eje sobre el que se rota de fuera hacia a dentro (mirar hacia el origen)***"

Es decir:

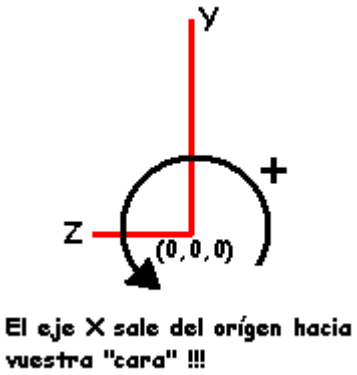


Figura 5. Sentido positivo de rotación.

En ella podéis ver como si miramos hacía el origen a través del eje de las X, una rotación contra-reloj es la indicada. Pues esa rotación se considera positiva. Así pues si digo que voy a rotar 30 grados CCW alrededor del eje X me refiero a que rotaré 30 grados siguiendo la dirección y el eje de la figura.

3.5. Deformar

Es la llamada transformación de "**Shearing**". Consiste en hacer que alguna de las componentes de un vértice **varíe linealmente** en función de otra. Me explico, se trata por ejemplo de alterar el valor de la X y de la Y en función del de la Z. Se consiguen efectos de **distorsión** muy interesantes para ciertas animaciones. Aquí tenemos las matrices a aplicar:

$$\begin{bmatrix} xt \\ yt \\ zt \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ Sxy & 1 & 0 & 0 \\ Sxz & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} xt \\ yt \\ zt \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & Syx & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & Syz & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} xt \\ yt \\ zt \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & Szx & 0 \\ 0 & 1 & Szy & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Figura 6. Matrices de deformación o "shearing".

El punto resultante es el de la izquierda. A la derecha tenemos la matriz a aplicar y un punto genérico al cuál se aplica ésta. En el primer caso variamos las componentes Y y Z en función de X, en el segundo X y Z en función de Y y en el tercero X e Y en función de Z.

Se entiende que los valores Sxy, Sxz, Syx, etc son **escalares reales**, es decir **números**, que vosotros mismos deberéis escoger para conseguir el efecto deseado.

3.6. Concatenación de transformaciones

Ahora supongamos que deseamos aplicar múltiples transformaciones a un determinado objeto geométrico. Para hacerlo tenemos que concatenar una detrás de otra todas las matrices por las que sus vértices deben multiplicarse. Para cada transformación creo una matriz, las multiplico todas y obtengo una matriz resultante más o menos compleja. Esa es la matriz que aplicaré a mis vértices para que se vean afectados "de golpe" por todas las transformaciones.

Hemos de tener en cuenta una propiedad: **La multiplicación de matrices no es conmutativa,** es decir que si A y B son matrices, es distinto A · B que B · A.

Lo cuál equivale a decir que **el orden de las matrices afecta al resultado final**, es decir, a la posición y orientación de nuestro objeto geométrico 3D. Una prueba gráfica es la siguiente figura:

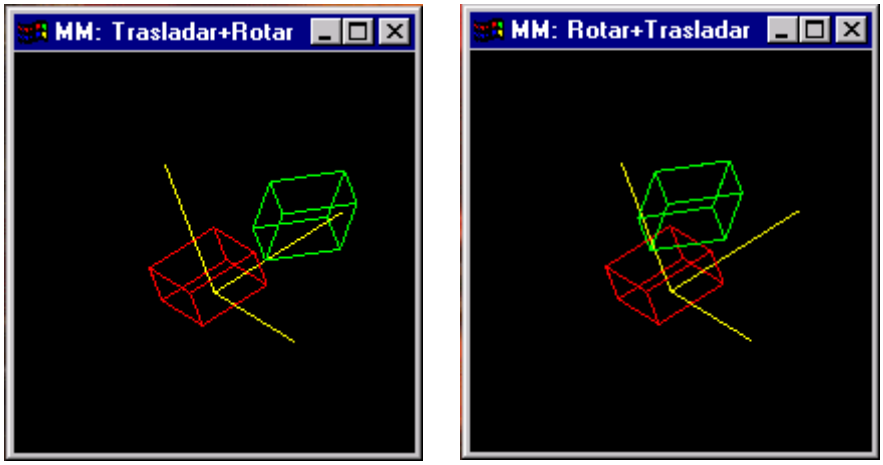


Figura 7. El orden afecta al resultado.

Se observa con toda claridad que el resultado de aplicar las mismas dos transformaciones pero con el orden cambiado da como resultados dos bien distintos. Como siempre el cubo rojo y centrado es el inicial y el verde el resultado final. En el primer caso hemos aplicado una traslación de 30 a lo largo de X y después una rotación de 45º alrededor del Y. En el segundo caso primero rotamos los 45º alrededor del mismo eje y después trasladamos 30 unidades siguiéndo el eje X.

Tenemos ambos casos en la figura 9.

3.7. Premultiplicación y postmultiplicación

Existen **dos convenciones** en cuanto a uso de transformaciones geométricas: la de **Robótica / Ingeniería y la de Gráficos**. En ambos casos se realizan exactamente las mismas operaciones pues tanto puedo querer mover un brazo robot como un personaje sobre mi juego 3D. Pero en cada caso se sigue una metodología distinta.

En la convención de Gráficos, que es la que yo he estado asumiendo durante todo el artículo y en concreto en la figura 9, se **postmultiplican** las matrices. Es decir, que los puntos se toman como vectores en columna que se multiplican a las matrices por la derecha. Y, además el orden de las transformaciones, de primera a última a aplicar, es de derecha a izquierda.

En cambio en Robótica se utilizan vectores de tipo fila, o renglón, que se multiplican por la izquierda. Las matrices se ordenan de izquierda a derecha en cuanto a orden de las transformaciones. Es decir, se **premultiplica.**

Aquí tenéis gráficamente lo que hemos explicado escribiendo:

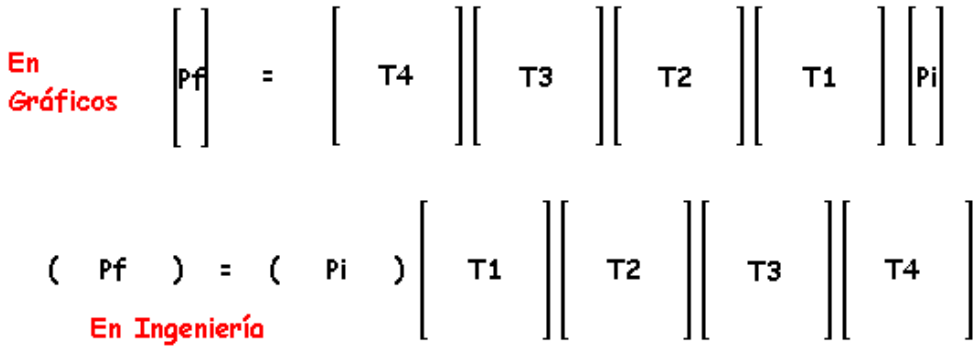


Figura 8. Premultiplicar y postmultiplicar.

Dónde **Pf** es el punto transformado final, **Pi** el inicial del que parto, **T1** la primera transformación a aplicar, **T2** la segunda y así sucesivamente.

Pero ojo que en ambos casos tenemos que multiplicar las matrices como siempre nos han enseñado, es decir, de izquierda a derecha. Sólo hay que fijarse en la convención que se usa porque eso define que forma tienen nuestros puntos, por que lado los he de multiplicar y en que orden debo ir añadiendo las transformaciones.

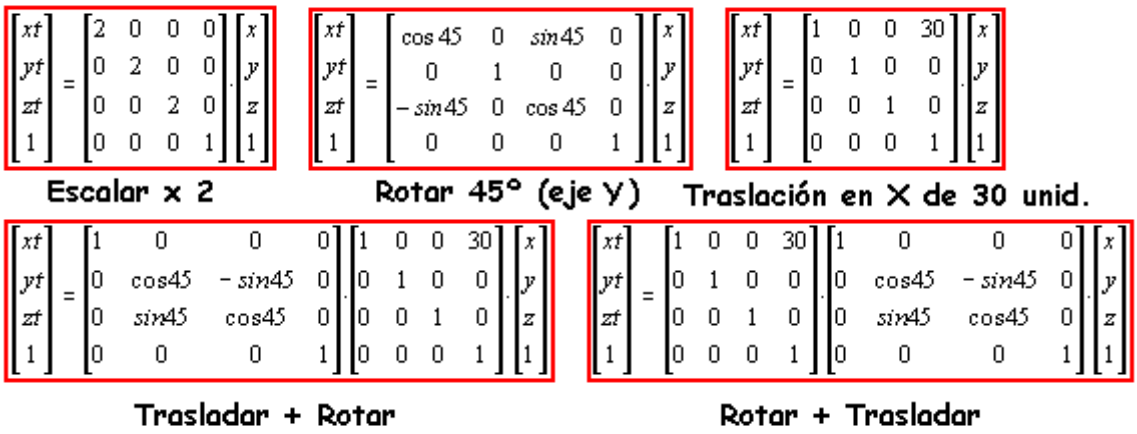


Figura 9. Ejemplos presentados.

En la figura siguiente tenemos las matrices de transformaciones genéricas que podremos aplicar.

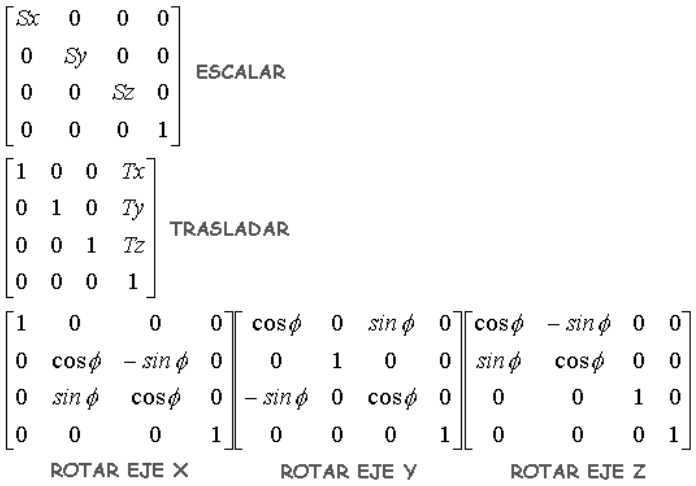


Figura 10. Matrices genéricas.

3.8. Implementación

En cualquier aplicación o paquete gráfico, también en OpenGL por supuesto, toda la geometría se ve afectada por la **CTM (Current Transformation Matrix) o matriz de transformación actual**. Esta guarda la información sobre todas las matrices que se han ido acumulando. Cualquier vértice que pase por el "pipeline" será multiplicado por esta matriz y consecuentemente transformado.

En OpenGL la CTM se compone de dos matrices. La **"Model-View matrix" o matriz de transformación** y la **"Projection matrix" o matriz de proyección**. Ambas se concatenan y de su producto se crea la CTM para el "pipeline" que controla OpenGL. De la segunda ya profundizaremos más adelante pues se encarga de la conversión 3D (mundo virtual) a 2D (pantalla) es decir, de la proyección. La primera nos interesa mucho ahora pues almacena todas las transformaciones afines que definamos en nuestro código.

Lo primero que debe hacerse es **inicializar** la matriz. Esto se hace cargando en ella la matriz **identidad** que es el **elemento neutro de la multiplicación** de matrices. Con esto nos aseguramos de "limpiarla" por completo antes de añadir transformaciones. Si no lo hiciéramos correríamos el riesgo de añadir transformaciones a otras ya existentes con lo cuál el resultado en pantalla no seria el deseado.

Esto lo hacemos con:

```
glMatrixMode(GL_MODELVIEW); // Activamos la matriz de transformación
glLoadIdentity(); // Cargamos la matriz identidad
```

Una vez hecho esto podemos acumular transformaciones sucesivas mediante las funciones:

· Escalar según sean los factores sx, sy y sz.

```
glScalef(GLfloat sx, GLfloat sy, GLfloat sz);
```

· Trasladar según los factores tx, ty y tz.

```
glTranslatef(GLfloat tx, GLfloat ty, GLfloat tz);
```

· Rotar el ángulo según el eje que define el vector (vx, vy, vz)

```
glRotatef(GLfloat angulo, GLfloat vx, GLFloat vy, GLfloat vz);
```

En el caso de la rotación podemos indicar cualquier eje. Lo típico es el eje X o (1,0,0), el Y o (0,1,0) o bien el Z que es (0,0,1).

A medida que vamos definiendo transformaciones se acumulan postmultiplicando en la matriz de transformación. Queda por tanto claro que OpenGL utiliza la convención de Gráficos como era de esperar. Cuidado porque **la regla en esta librería es que la transformación que se ha definido última será la primera en aplicarse a la geometría**. Eso quiere decir que si tengo 3 líneas de código tal que:

```
GlScalef(…)
GlRotatef(…)
GlTranslatef(…)
```

...la primera en aplicarse será la transformación de la tercera línea, seguida por la de la segunda y finalizando con la primera. Eso "altera" un poco nuestra idea de ejecución secuencial al programar estructuradamente pero es así con OpenGL y debe tenerse en cuenta.

3.8.1. Concepto de "pila" o "stack"

La matriz de transformación, la "model-view" debe entenderse como una pila. Cada transformación que añadimos entra a la pila como la última y por tanto al salir será la primera. Podemos salvar el estado de la pila en cualquier momento para recuperarlo después. Esto la haremos mediante las funciones:

```
glPushMatrix( ); // Salvamos el estado actual de la matriz
glPopMatrix( ); // Recuperamos el estado de la matriz
```

Esto nos servirá en el caso de que tengamos que aplicar algunas transformaciones a una pequeña parte de la geometría. El resto no debiera verse afectado por esos cambios. Lo que se hace es definir las transformaciones generales que afectan a todos. Entonces se salva la matriz y se añaden otras. Se dibuja la geometría "especial" y inmediatamente después se recupera la matriz. Ahora podemos dibujar todo el resto, ya que no se verá afectado por las transformaciones que hayamos definido entre el `glPush()` y el `glPop()`

Aquí vemos un pequeño ejemplo

```
(...)
glRotatef... // afectará a toda la geometría que dibuje a partir de ahora
glTranslatef... // afectará a toda la geometría que dibuje a partir de ahora
glPushMatrix( );  // salvo el estado actual de la matriz, es decir, las 2
                  // transformaciones anteriores
glTranslatef... // afectará a sólo a la geometría que dibuje antes del glPop
glScalef...       //afectará a sólo a la geometría que dibuje antes del
                  //glPop
dibujo_geometría_específica( ); // Render de la geometría que pasará
                                // por 4 transformaciones
glPopMatrix( ); // recupero el estado de la matriz anterior
dibujo_el_resto( ); // Render de la geometría que pasará por 2 transformaciones
(...)
```

3.8.2. Crear matrices "a medida"

Por último comentar que también podemos crearnos matrices "a mano" para después pasarlas a la matriz de transformación de OpenGL. No disponemos tan sólo de las funciones de traslación, rotación... que os he comentado sinó que también podemos usar:

```
glLoadMatrixf(puntero_a_matriz);
glMultMatrixf(puntero_a_matriz);
```

En el primer caso sustituimos a la matriz actual con la que le pasamos precalculada por nosotros mismos. En el segundo caso multiplicamos a lo que ya haya en la matriz por lo que nosotros pasamos.

El puntero a una matriz se asume como variable del tipo:

```
GLfloat M[16];
```

o

```
GLfloat M[4][4];
```

Es importante saber que OpenGL asume que la matriz que se le pasará está definida por columnas, es decir:

```
|a0   a4    a8    a12|
|a1   a5    a9    a13|
|a2   a6    a10   a14|
|a3   a7    a11   a15|
```

Primero definimos a0, después a1, a2, a3, a4 ... y así sucesivamente.

4. Proyecciones.

Tras las transformaciones homogéneas para nuestra geometría ya somos capaces de modelar y situar objetos a lo largo y ancho de nuestra escena virtual. ¿Pero cómo hacer que aparezcan en nuestra ventana del Sistema Operativo? La ventana es 2D y en cambio está mostrando geometria 3D dando sensación de que hay profundidad.

Tenemos que proyectar la geometría en un **plano de proyección**. Como plano es obviamente 2D y suele situarse por convención en **Z = 0 (plano XY)**. La idea es proyectar primero para "discretizar" después. Por discretizar entendemos pasar del mundo real (float o double) al entero (integer) que es el de los píxels de nuestro monitor.

4.1. Tipos de proyección

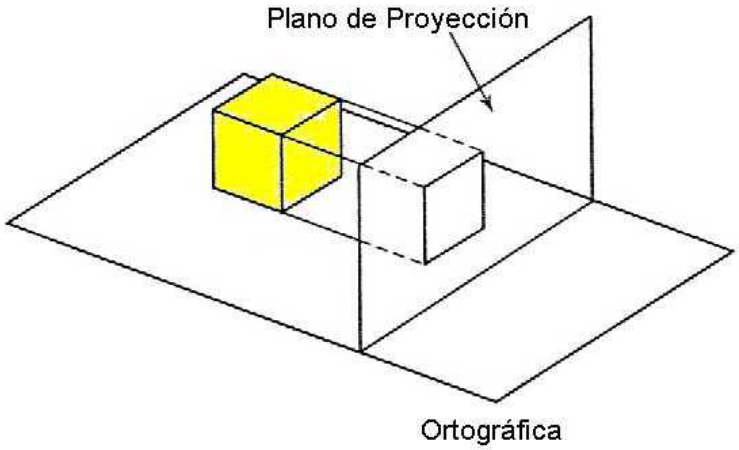
No comentaremos todas las existentes, no será necesario. Nos centraremos en las **proyecciones planares**. En éstas se define una **dirección de visión** (**viewing**) que va desde el observador hasta el objeto en cuestión. Esta dirección se establece por medio de **proyectores** (líneas) que cortan el plano de proyección generando así la imagen 2D que finalmente aparecerá en pantalla. Como primera gran clasificación de las proyecciones planares podemos hablar de:

- Proyecciones planares
  - Paralelas
    - Oblicua
    - Ortográfica
  - Perspectiva
    - 1 punto
    - 2 puntos
    - 3 puntos

Nosotros nos centraremos en dos de estas proyecciones. La ortográfica y la perspectiva de un sólo punto.

4.2. Proyección ortográfica

Como proyección paralela que es, cuenta con proyectores paralelos entre ellos. El **centro de proyección** (**COP**) se encuentra en el infinito. En el caso de la ortográfica, los proyectores son perpendiculares al plano de proyección. Lo observamos mejor en la figura:



Este tipo de proyección **no preserva las dimensiones reales** de los objetos según la distancia hasta ellos. Es decir, que si nos acercamos o alejamos de ellos no se producen cambios de tamaño, con lo cual el realismo no es total. Se utiliza tradicionalmente en proyectos de ingeniería del tipo de programas CAD/CAM.

Los parámetros a especificar son las dimensiones de la caja (Xmin, Xmax, Ymin, Ymax, Zmin, Zmax). A los valores **MAX** y **MIN** también se les denomina **FAR** o **BACK** y **NEAR** o **FRONT**.

En OpenGL la podemos definir de la siguiente forma:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(x_min, x_max, y_min, y_max, z_min, z_max);
```

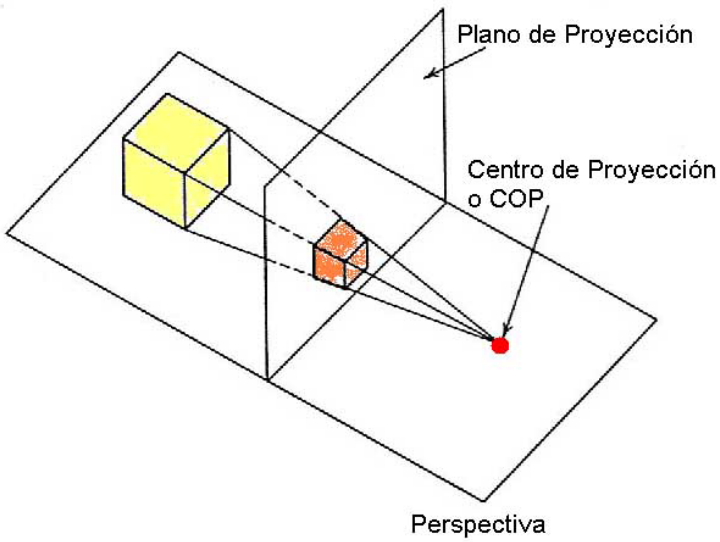
la última función puede reemplazarse por:

```
gluOrtho2D(x_min, x_max, y_min, y_max);
```

si se aceptan los valores por defecto de Zmin = -1.0 y Zmax = 1.0.

4.3. Proyección perspectiva

Esta es la que utilizaremos para dotar del mayor realismo a nuestras aplicaciones. Las proyecciones perspectiva preservan las dimensiones reales de los objetos si nos acercamos / alejamos de ellos. Por tanto el efecto visual es justo el que necesitamos en casos de apariencia real.



En la figura tenemos una proyección perspectiva con un sólo COP o **punto de fuga**. Todos los proyectores emanan de él y se dirigen hasta el objeto intersectando el plano de proyección. Como podemos observar los proyectores no son paralelos entre ellos tal.

En OpenGL la podemos definir así:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective( FOV en grados, Relación de Aspecto, z_near, z_far);
```

Los parámetros que tenemos que especificar son:

· **FOV** en grados. Es el "**field of view**" o campo visual. Se refiere al ángulo de abertura vertical.
· **Relación de Aspecto** o "**aspect ratio**". Es el cociente entre la anchura (width) y la altura (height) del plano de proyección deseado.
· Los valores **NEAR** y **FAR** del volúmen de visualización perspectiva. Idem que en ortográfica.

Podemos definir también una proyección perspectiva usando la función:

```
glFrustrum(x_min, x_max, y_min, y_max, z_min, z_max);
```

En este caso OpenGL calculará el "**frustrum piramidal**", es decir, el volúmen de visualización perspectiva más idóneo. Son simplemente dos maneras distintas de acabar creando lo mismo. A cada uno con su elección, la que os parezca más intuitiva y cómoda.

Las distancias NEAR y FAR son siempre positivas y medidas desde el COP hasta esos planos, que serán obviamente paralelos al plano Z = 0. Dado que la cámara apunta por defecto en la dirección negativa de Z, el plano de front (near) estará realmente situado en z = -zmin mientras que el de back (far) estará en z = -zmax.

Hemos de pensar que no hay que proyectar toda la geometría existente sinó tan sólo la que vé la cámara desde su posición y orientación. Tenemos que acotar en este sentido y es por eso que además de definir un plano de proyección y una forma de proyectar, creamos un volúmen de visualización finito con unas fronteras bién marcadas. Todo aquello que no se encuentre dentro del volúmen será rechazado y no proyectado dado que no debe verse.

4.4. La Cámara

La cámara son nuestros ojos virtuales. Todo lo que ella vea será proyectado, discretizado y finalmente mostrado en nuestra ventana del sistema operativo. Podemos imaginar que de la cámara emana el volúmen de visualización de forma que se traslada con ella. Los parámetros a definir en cuanto a la cámara son:

Posición XYZ en el mundo 3D. Al igual que un objeto cualquiera, la cámara debe posicionarse. En un juego arcade 3D típico la cámara nos da la visión frontal del mundo y se va moviendo a nuestro antojo con las teclas o el ratón. Cada vez que modificamos la posición varían las coordenadas XYZ de cámara.

Orientación. Una vez situada debe orientarse. Yo puedo estar quieto pero girar la cabeza y mirar hacía donde me venga en gana.

Dirección de "AT". Define hacia dónde estoy mirando, a qué punto concretamente.

En OpenGL lo tenemos fácil con:

```
gluLookAt(eyeX, eyeY, eyeZ, atX, atY, atZ, upX, upY, upZ);
```

Esta es la función que determina dónde y cómo está dispuesta la cámara. La posición de la cámara no tiene nada que ver con la proyección que hayamos establecido. La proyección se define sólo una vez, típicamente al principio del programa en una función inicializadora, mientras que la cámara se mueve continuamente según nos interese. Añadir a esto que la matriz que se modifica al llamar a esta función no tiene que ser GL_PROJECTION sinó GL_MODELVIEW. OpenGL calculará todas las transformaciones que aplicará al mundo 3D para que manteniendo la cámara en el origen de coordenadas y enfocada en la dirección negativa de las Z's nos dé la sensación de que lo estamos viendo todo desde un cierto lugar. Para ello recordad siempre activar esta matriz antes de llamar a gluLookAt de esta forma:

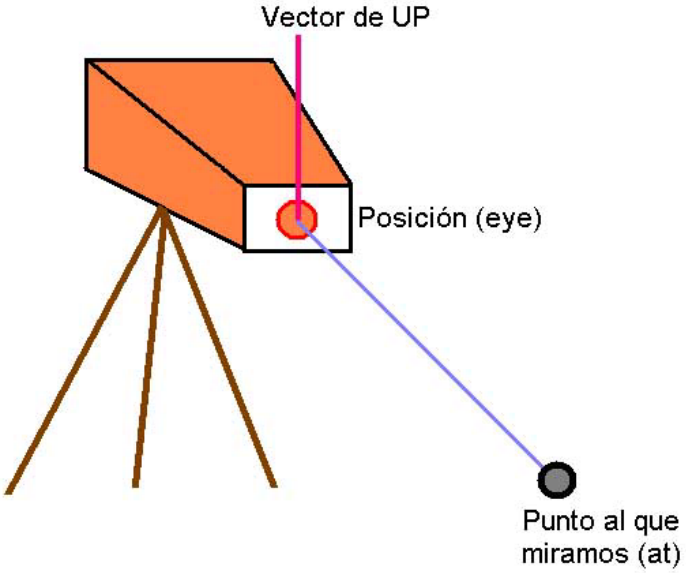
```
glMatrixMode(GL_MODELVIEW);
```

En cuanto a los parámetros que demanda la función son los siguientes:

· **Coordenadas del "eye"**. Es literalmente la posición XYZ dónde colocar la cámara dentro del mundo.

· **Coordenadas del "at"**. Es el valor XYZ del punto al que queremos que mire la cámara. Un punto del mundo obviamente.

· **Coordenadas del vector "up"**. Si, es un vector y no un punto. Con él regularemos la orientación de la cámara. Este ha de ser el vector que "mira hacia arriba" si entendemos que el vector que mira hacia delante es el que va del "eye" hasta el "at". Variando el "up" variamos la orientación:



5. La librería GLUT

En este apartado veremos el funcionamiento de la librería GLUT. Esta librería, que se sirve de funciones de OpengGL, añade funcionalidades tales como creación de ventanas, control de eventos de entrada, creación de menús, etc. Su simplicidad de uso la hace idónea para aplicaciones pequeñas, donde tan solo queremos experimentar con OpenGL, sin tener que complicar el código para crear ventanas o controlar el teclado con la API del Sistema Operativo.

A continuación veremos un pequeño codigo fuente de un programa, mientras lo analizamos veremos la manera de funcionar de GLUT.

```
/* Copyright (c) Oscar García Panyella 1998, todos los derechos
reservados.
 *      Curso      de      OpenGL      para      Macedonia      Magazine.
 *                              Primer                              ejemplo.
 */
/*           Incluimos            las            librerías            */
#include <GL/glut.h
/*     Ancho     de     la     ventana     de     visualización     */
#define                               ANCHO                               400
/*     Alto     de     la     ventana     de     visualización     */
#define                                ALTO                               400
/* Coordenada X del origen de la ventana, esquina superior izquierda
*/
#define                              ORIGENX                              100
/* Coordenada Y del origen de la ventana, esquina superior izquierda
*/
#define ORIGENY 100
/*
 *Parámetros             iniciales             del             programa.
 */
void                                                          inicio(void)
{
        /*     Activamos     la     matriz     de     proyección.     */
                                              glMatrixMode(GL_PROJECTION);
      /*   "Reseteamos"   esta   con   la   matriz   identidad.   */
                                                        glLoadIdentity();
    /* Plano de proyección igual a la ventana de visualización.
       *Volumen   de   visualización   desde   z=-10   hasta   z=10.
                                                                       */
         glOrtho(0,      ANCHO,      0,      ALTO,      -10,      10);
      /*   Activamos   la   matriz   de   modelado/visionado.    */
                                           glMatrixMode(GL_MODELVIEW);
              /*           La           "reseteamos".           */
                                                        glLoadIdentity();
    /* Nos trasladamos al centro de nuestra ventana donde siempre
dibujaremos                        el                        polígono.
     * Nos mantenemos en el plano z=5 que se encuentra dentro del
volumen                                                              de
                               *                          visualización.
                                                                       */
        glTranslatef((GLfloat)ANCHO/2,      (GLfloat)ALTO/2,      5.0);
    /* Color de fondo para la ventana de visualización, negro. */
            glClearColor(0.0,         0.0,         0.0,         0.0);
}
/*     OpenGL     llamara     a     esta     rutina     cada     vez
 *       que       tenga       que       dibujar       de       nuevo.
 *
 * Dado que rellenara el polígono de color y cada vértice es de
 * un color diferente, OpenGL rellenara el interior con una
interpolación
 * de los colores de los 4 vértices. Lo hace automáticamente.
 */
void                                                          dibujar(void)
{
    /* "Limpiamos" el frame buffer con el color de "Clear", en este
```

```
caso                              negro.                                */
                                        glClear(GL_COLOR_BUFFER_BIT);
    /* Queremos que se dibujen las caras frontales de los polígonos y
con            relleno            de            color.            */
                 glPolygonMode(GL_FRONT,                  GL_FILL);
                                          glBegin(GL_POLYGON);
          /*    Color    azul    para    el    primer    vértice    */
                   glColor3f(0.0,             0.0,             1.0);
                  glVertex3i(-100,             -100,             5);
          /*    Color    verde    para    el    segundo    vértice    */
                   glColor3f(0.0,             1.0,             0.0);
                   glVertex3i(-100,             100,             5);
          /*    Color    rojo    para    el    tercer    vértice    */
                   glColor3f(1.0,             0.0,             0.0);
                    glVertex3i(100,             100,             5);
         /*    Color    amarillo    para    el    cuarto    vértice    */
                   glColor3f(1.0,             1.0,             0.0);
                   glVertex3i(100,             -100,             5);
                                                          glEnd();
}
/*
 *                Main                del                 programa.
 */
int          main(int          argc,          char          **argv)
{
    /* Primera llamada siempre en OpenGL, por si usáramos la línea de
comandos                                                          */
                         glutInit(&argc,                         argv);
     /*  Activamos  buffer  simple  y  colores  del  tipo  RGB  */
         glutInitDisplayMode      (GLUT_SINGLE      |      GLUT_RGB);
    /* Definimos una ventana de medidas ANCHO x ALTO como ventana de
visualización                                                     */
             glutInitWindowSize           (ANCHO,           ALTO);
    /* Posicionamos la esquina superior izquierda de la ventana en el
punto                             definido                        */
           glutInitWindowPosition         (ORIGENX,         ORIGENY);
    /* Creamos literalmente la ventana y le adjudicamos el nombre que
se     observara     en     su     barra     de     titulo     */
                glutCreateWindow("Cuadrado             Multicolor");
           /*        Inicializamos        el        sistema        */
                                                       inicio();
    /* Hacemos saber a OpenGL que cada vez que sea necesario dibujar
de                                                          nuevo,
     * por ejemplo la primera vez, o al redimensionar la ventana con
el                                ratón                          o
     * en caso de provocar un "redibujado" por programa,  debe llamar
a                                                               la
                    *               función               "dibujar".
                                                                  */
                                          glutDisplayFunc(dibujar);
    /* Aquí espera el programa mientras nada ocurra, es un Loop
     * infinito que se vera turbado por las sucesivas veces que
               *          sea          necesario          redibujar.
                                                                  */
                                                    glutMainLoop();
     /*  ANSI  C  requiere  que  main  retorne  un  entero.  */
                                return                            0;
}
```

5.1. Función MAIN

Ya sabemos que cualquier programa en C se caracteriza por su función MAIN o principal, función que se llama automaticamente al iniciar la ejecución del programa. Veamos que hacemos con ella en el caso de OpenGL.

Para definir la función MAIN lo haremos como siempre,

```
int main (int argc, char **argv) {
```

Una vez declarada pasamos a la primera función de GLUT. Debemos recoger los parámetros de la línea de comandos, ARGC y ARGV, mediante **glutInit(...)**.

```
glutInit (&argc, argv);
```

Ahora hay que decirle al motor gráfico como queremos renderizar, es decir, si hay que refrescar la pantalla o no, que buffers hay que activar/desactivar y que modalidad de colores queremos usar. En este caso no tenemos doble buffer (**GLUT_DOUBLE**) y por tanto definimos un buffer de render único con la constante **GLUT_SINGLE**. Por otra parte, tenemos dos maneras de colorear, tenemos los colores indexados, con una paleta de colores, o bien tenemos la convención RGB. En nuestro caso le decimos al subsistema gráfico que cada color a aplicar será definido por tres valores numéricos, uno para el rojo (Red), otro para el verde (Green) y otro para el azul (Blue). Para esto usamos la constante **GLUT_RGB**. Sino usaríamos **GLUT_INDEX**.

```
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
```

Con buffer simple contamos tan solo con un área de memoria que se redibuja constantemente. Esto no es factible para aplicaciones donde la velocidad de render es muy alta o el contenido gráfico es elevado. En nuestro caso sólo definimos un polígono y por lo tanto nos conformaremos con el buffer simple. De todas formas casi siempre utilizaremos el buffer doble, dos zonas de memoria que se alternan de manera que se dibuja primero una y en la siguiente iteración se dibuja la otra. Ahora GLUT nos permite definir las medidas de nuestra ventana de visualización. Estamos definiendo literalmente el ANCHO y ALTO de nuestra ventana en pixels.

```
glutInitWindowSize (ANCHO, ALTO);
```

También hay que "colocar" la ventana en algún punto determinado de la pantalla. Después podremos moverla con el ratón, si no lo impedimos por código, pero siempre hay que indicar un punto de origen para la primera vez que ejecutamos la aplicación. Las coordenadas que suplimos a la función son las correspondientes al pixel que se encuentra en la esquina superior-izquierda de la ventana.

```
glutInitWindowPosition (ORIGENX, ORIGENY);
```

Una vez ya definido como "renderizar", con que medidas de ventana y en que posición física de la pantalla, crearemos la ventana. Para ello le damos un nombre cualquiera que sera el título que aparecera en esta. De hecho a la función se le pasa un array de caracteres, ya sea explícito o sea el nombre entre comillas, o bien implícito, es decir, una variable que contiene el nombre:

```
glutCreateWindow ("Cuadrado Multicolor");
```

Hasta aquí ya tenemos una ventana en pantalla, con su título y fondo negro por defecto, ya que si os fijáis no hemos especificado aún otro. Ya podemos empezar a "generar" gráficos para colocarlos en ella. Creamos una función llamada **Inicio()** que se encargará de inicializar todo lo necesario:

```
inicio ();
```

Después describiremos que hace esta función en el apartado de **Inicialización del sistema.**

Tras inicializar el sistema le decimos al subsistema gráfico cuál es la función que debe llamarse cada vez que se requiera dibujar de nuevo en pantalla. Esta función la tendremos que crear nosotros y será dónde le diremos a OpenGL qué es lo que debe dibujarse en la ventana que hemos creado para tal fin. Esta función se llamará cada vez que se produzca el evento de "render", es decir, cada vez que se cambia el tamaó de la ventana, o cada vez que se cambia de posición, o bien cuando le indiquemos nosotros.

```
glutDisplayFunc (dibujar);
```

Ahora GLUT ya sabe dónde buscar lo que ha de dibujar, y qué ha de dibujar. Ahora solo falta entrar en el bucle infinito que domina cualquier aplicación OpenGL. Con la función que sigue, que siempre se pone al final del main, le decimos a la librería que espere eternamente a que se produzcan "eventos", es decir, que hasta que no ocurra algo se mantenga a la expectativa. En nuestro caso el único evento posible es el propio "render" pues aún no hemos tratado como interactuar con el ratón, ni cómo controlar que pasa cuando se mueve la pantalla o se redimensiona.

```
glutMainLoop ();
```

Para acabar, y para cumplir con el standard ANSI C,

```
return 0;
}
```

5.2. Inicialización del sistema

Analizemos la función de inicialización Inicio(), a la cual hemos llamado en el punto anterior. Dado que no se le pasa ningún parámetro ni retorna ningún valor la declaramos tal como:

```
void inicio (void)
      {
```

Es aquí donde tendremos que definir la matriz de proyección. Recordemos que solo hace falta definirla una vez, al principio de la aplicación. Para ello activaremos la matriz de proyección,

```
      GlMatrixMode (GL_PROJECTION);
```

La inicializaremos (le asignamos la matriz identidad)

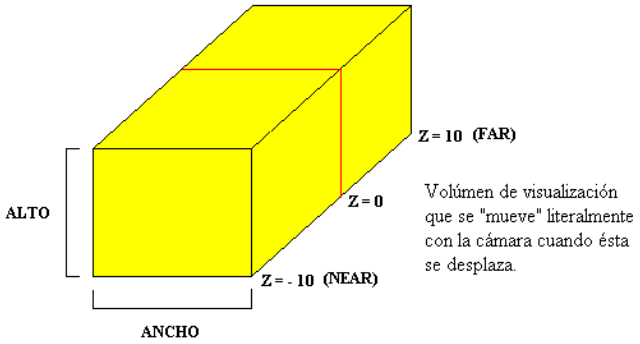
Por lo tanto cargamos ("load") la matriz identidad en la de proyección...

```
      glLoadIdentity ();
```

Ahora que tenemos la matriz de proyeccion inicializada, le diremos a OpenGL que tipo de proyeccion queremos. En este caso, usaremos la ortográfica que vimos en el capítulo anterior.

```
glOrtho (0, ANCHO, 0, ALTO, -10, 10);
```

que crea el siguiente volúmen de visualización:



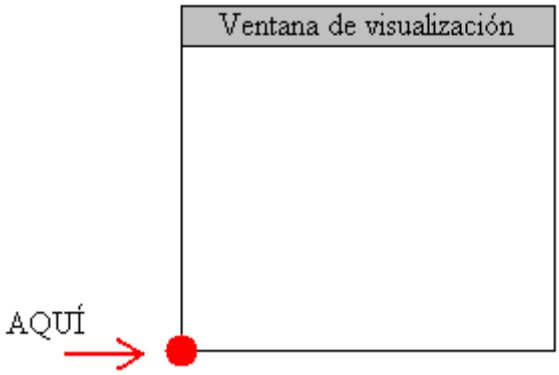
Ya hemos terminado con la matriz de proyección. El sistema ya sabe como debe proyectar en pantalla. Ahora pasemos a la matriz de modelado/visionado, es decir, la matriz que rota, escala, traslada, etc. Dado que queremos operar sobre ella la seleccionamos primero:

```
glMatrixMode (GL_MODELVIEW);
```

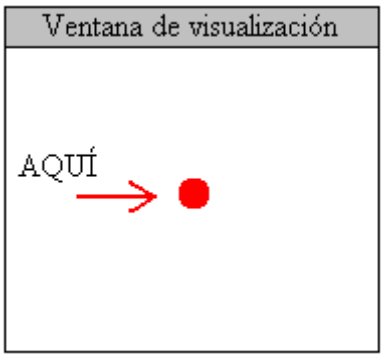
...y también la inicializamos con la matriz identidad.

```
glLoadIdentity ();
```

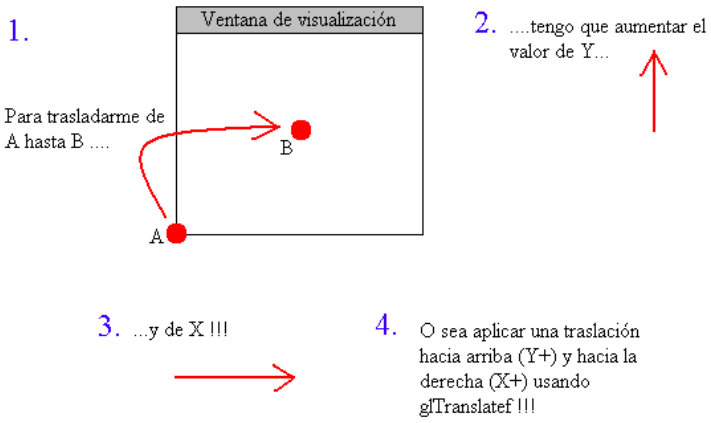
Originalmente el centro de coordenadas se asume en la esquina inferior-izquierda de la ventana, es decir:



pero nosotros lo queremos en el centro de la ventana, o sea...



y por lo tanto debemos aplicar una traslación al orígen. Después de "limpiar" la matriz de modelado/visionado, almacenaremos la traslación en ella de manera que todo aquello que requiera ser dibujado sera primero multiplicado por esta matriz y consiguientemente trasladado donde lo queremos con relación a un orígen de coordenadas en el centro de la ventana.



Para ello utilizamos en el programa la siguiente línea de código:

```
glTranslatef ((GLfloat)ANCHO/2, (GLfloat)ALTO/2, 5.0);
```

Definimos también que el centro se encuentre en:

$$Z = 5.0$$

Esto no es importante y ni lo notaremos pues recordad que la proyección ortográfica elimina la componente Z de todos los puntos a "renderizar". De todas formas, glTranslatef(...) requiere el parámetro Z, y le damos uno.

Observad que hacemos un "**casting**" de las dos primeras coordenadas. Un "casting" permite hacer conversiones directas entre formatos de variables. En este caso aviso al compilador de que las dos primeras coordenadas deben tratarse como GLfloat (numeros reales) aunque ese no sea su formato inicial, ya que espera valores GLfloat o float para sus parámetros de entrada.

Las constantes ANCHO y ALTO las hemos definido al principio del programa y indican la altura y la anchura de la ventana.

Para acabar con las inicializaciones le diremos a OpenGL con que color deseamos que se reinicialice el frame buffer cada vez que haya que volver a dibujar, o lo que es lo mismo, que color debe aparecer en todas aquellas áreas donde no dibujemos nada, es decir, el color del fondo. Definimos que queremos el color de fondo negro para nuestra ventana.

```
glClearColor (0.0, 0.0, 0.0, 0.0);
}
```

5.3. Render del sistema

Sabemos que OpenGL llamara a la función DIBUJAR cada vez que necesite "renderizar" de nuevo. En nuestro caso hacemos lo siguiente:

```
void dibujar (void)
      {
```

Una vez declarada la función, que como véis ni espera ni retorna nada, le decimos a OpenGL que restaure el frame buffer. Esto es obligado para un correcto proceso de render. De hecho estamos "reinicializando" la ventana de visualización con el color definido anteriormente en *glClearColor (0.0, 0.0, 0.0, 0.0);*. De hecho nosotros definimos el negro como valor para el fondo pero podríamos no haberlo hecho ya que se adopta este color por defecto.

```
      glClear (GL_COLOR_BUFFER_BIT);
```

Ahora le tendremos que decir a OpenGL que es lo que ha de dibujar: podemos dibujar las caras frontales, las traseras o ambas. Además podemos rellenarlas con color o no. En este segundo caso sólo se colorearan los bordes (aristas). Ya dijimos en su momento que OpenGL supone caras frontales aquellas cuyos vértices se definen en **orden contrareloj**. Casi siempre sólo dibujaremos las frontales porque las traseras no se verán, y así ahorraremos tiempo de cálculo optimizando la velocidad de ejecución. Por tanto, en este caso, decidimos dibujar sólo las caras frontales (**GL_FRONT**) y rellenadas, es decir con color en su interior ademas de en los bordes (**GL_FILL**). Para dibujar sólo las caras traseras utilizaríamos **GL_BACK** y para dibujar ambas **GL_FRONT_AND_BACK**. En el caso del color de relleno, si deseamos evitarlo coloreando solo los bordes de cada polígono usaremos la constante **GL_LINE**.

```
      glPolygonMode (GL_FRONT, GL_FILL);
```

Empecemos a definir un polígono

```
      glBegin (GL_POLYGON);
```

El primer vértice es de color azul y sus componentes en pantalla son los 3 enteros que observáis:

```
      glColor3f (0.0, 0.0, 1.0);
      glVertex3i (-100, -100, 5);
```

el segundo vértice es verde...

```
      glColor3f (0.0, 1.0, 0.0);
      glVertex3i (-100, 100, 5);
```

el tercero es rojo....

```
      glColor3f (1.0, 0.0, 0.0);
      glVertex3i (100, 100, 5);
```

y el cuarto es amarillo....

```
      glColor3f (1.0, 1.0, 0.0);
      glVertex3i (100, -100, 5);
```

y cerramos la estructura correpondiente al polígono que estabamos dibujando. Un simple cuadrado con un color distinto en cada esquina.

```
glEnd ();
}
```

Si le decimos a OpenGL que cáda vertice es de un color diferente, al dibujar el polígono interpolará los colores, de manera que si un vértice es rojo y el siguiente amarillo, pondrá en medio de ambos toda la escala de colores del rojo al amarillo, automáticamente.

Cada vez que OpenGL redibuje la escena vendrá aquí y volverá a ejecutar este código.

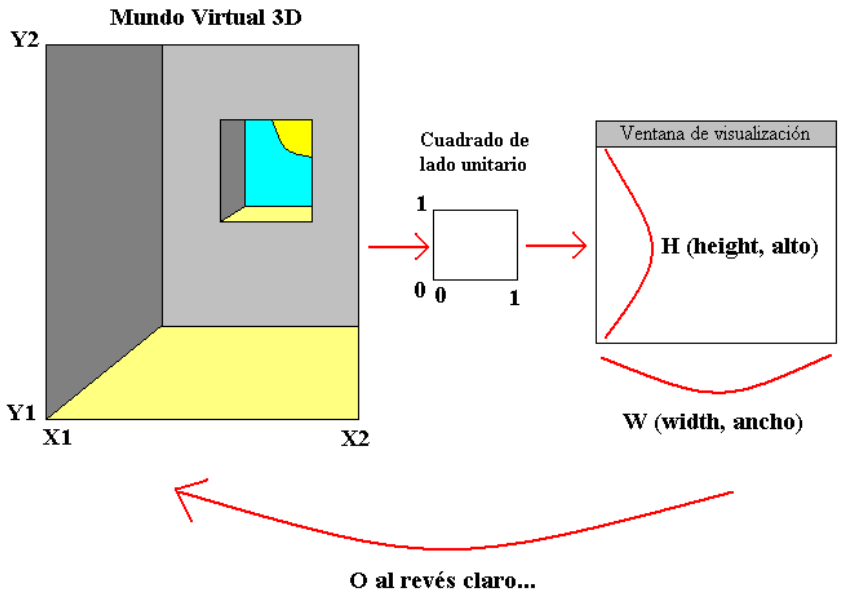
5.4. Sistema de ventanas

Gracias a GLUT podremos controlar diversas ventanas a la vez mostrando diferente información en cada una de ellas. Tan sólo definiremos una función de "display" diferente para cada ventana así como respuestas distintas según sea la interacción del usuario con ellas.

Otro aspecto importante es el de las coordenadas. No son lo mismo las coordenadas de mundo que las de ventana. Por tanto una ventana tiene coordenadas 2D que definimos como habéis visto en el programa de ejemplo (ALTO, ANCHO, ORIGENX, ORIGENY) y en cambio el mundo virtual que estamos creando suele ser 3D y no tiene porque corresponderse con ella. Entonces se produce un curioso fenomeno. ¿Qué ocurre si yo quiero recuperar las coordenadas del ratón en el momento en que se produce un "click" en la ventana, para trabajar con ellas?, es decir, imaginad que el usuario debe poder mover una esfera por el mundo simplemente seleccionándola con el ratón y arrastrándola. Si resulta que cuando se mueve el raton las coordenadas que el sistema operativo nos retorna son las de la posición de la esfera en la ventana y éstas no se correponden con las del mundo, no la podremos mover correctamente.

La solución sería un cambio de coordenadas. Cada vez que deseemos recoger información del raton para operar con ella en el mundo tendremos que convertir las coordenadas recibidas vía S.O. en coordenadas del mundo.

Se suele hacer de la siguiente forma:



Observad la figura atentamente. Podemos convertir coordenadas de mundo a pantalla o bién de pantalla a mundo. De hecho el proceso es obviamente el mismo pero en una dirección o en la contraria.

Imaginemos que queremos saber cuál es la coordenada de pantalla de un punto (Xm, Ym, Zm) cualquiera de nuestro mundo virtual. Queremos saber en que lugar de la pantalla será dibujado, en que pixel.

En primer lugar pasamos de un punto 3D en el mundo a un punto situado en una ventana imaginaria de lado unitario de manera que las coordenadas estarán acotadas entre 0 y 1. A esto se le llama "mapear coordenadas". Llamemos a las coordenadas 2D del punto en el cuadrado unitario (Xu, Yu). Entonces:

$$Xu = (Xm - X1) / (X2 - X1);$$
$$Yu = (Ym - Y1) / (Y2 - Y1);$$

donde como ya he dicho (Xm, Ym) son las coordenadas 2D del punto en el mundo y (X1,Y1), (X2, Y2) las observáis en la figura como límites de la escena real. Entonces mapeamos de coordenadas unitarias a pantalla. Llamemos a las coordenadas de pantalla (Xp, Yp):

$$Xp = W * Xu;$$
$$Yp = H * (1 - Yu);$$

donde W y H son las medidas de la ventana de visualización como también podéis ver en la figura. Y así ya sabemos en que punto de la pantalla se mapea el punto del mundo al cual nos referíamos. Si queremos pasar coordenadas de pantalla a mundo, pués sólo tenemos que repetir el proceso pero al revés.

Resumamos un ejemplo típico de úso de este concepto. Supongamos lo que dijimos, es decir, queremos mover una esfera con el ratón cada vez que hacemos "click" sobre ella o bién cuando la arrastramos con el botón derecho pulsado. En este caso tendré que:

· Tomar las coordenadas de pantalla. Veremos que eso nos lo da GLUT automáticamente.

· Convertirlas según hemos analizado, en coordenadas de mundo.

· Operar con ellas, ya convertidas, en mi programa. En este caso las podemos incrementar/decrementar según se mueva el raton hacia un lado u otro. Obtenemos nuevas coordenadas de posición para la esfera, en el mundo.

· Le decimos a OpenGL que dibuje la escena de nuevo. Repetitivamente veríamos que nuestra esfera se mueve a la vez que nuestra mano lo hace con el ratón. Es decir, se va "redibujando" en posiciones diferentes cada vez, lo que implica sensación de movimiento para nuestros ojos.

5.4.1. Concepto de Aspect Ratio

Veamos otro efecto no deseable. Primero definiremos lo que se entiende por "**aspect ratio**". El "aspect ratio" de una ventana es la relación que existe entre su anchura y su altura. Esta relación debería conservarse siempre aunque se varie el tamaño de ésta pues de lo contrario distorsionaremos el contenido y la visualización será bastante ineficiente. Controlaremos este fenómeno modificando la ventana para acomodarla al ratio correcto y lo haremos en nuestro programa cuando sea el caso, por ejemplo al producirse un redimensionado de ésta. Si se varia la altura ha de variar la anchura que corresponda o al revés.

5.4.2.Viewports

Si queremos dividir una ventana en varias porciones independientes podemos usar "**Viewports**". Un viewport es un área rectangular de la ventana de visualización. Por defecto es la ventana entera pero podemos variarlo a gusto. Se utiliza la función:

```
void glViewport (GLint x, GLint y, GLsizei w, GLsizei h)
```

...dónde (X,Y) es la esquina inferior izquierda del rectangulo o viewport. Esta coordenada debe especificarse con relación a la esquina inferior izquierda de la ventana. Claro está que W, H son la anchura y altura de nuestro viewport dentro de la ventana. Todos los valores son enteros. Tras activarlo o definirlo será dentro de éste donde dibujaremos.

6. Eventos y menús de usuario.

Un **evento** es "*algo que el usuario puede hacer*" como por ejemplo maximizar una ventana, redimensionarla, pulsar el botón izquierdo del ratón, o usar una determinada combinación de teclas. En todos estos casos deberemos "*ejecutar algo de código*" dentro de nuestro programa, si es que estaba previsto así.

En OpenGL, y gracias a GLUT, se le permite al usuario "*jugar*" pulsando botones del ratón, moviéndolo por la pantalla, apretando teclas, cambiando la ventana de la aplicación. Cada vez que éste provoque alguno de estos eventos deberemos llamar a una determinada rutina o función para que se haga cargo de la acción a tomar.

Las más comunes funciones que OpenGL llama automáticamente al detectar uno de estos eventos son:

glutMouseFunc(función de control eventos con el ratón**);**
glutMotionFunc(función de control eventos de movimiento del ratón**);**
glutReshapeFunc(función de control del cambio de tamaño de la ventana de visualización**);**
glutKeyboardFunc(función de control eventos con el teclado**);**
glutDisplayFunc(función de control del render **);**
glutIdleFunc(función que se activa cuando no hacemos nada**);**

Son las llamadas "*Callbacks*" o funciones controladoras de eventos.

Analicemos algunos casos:

6.1. El ratón

Lo más normal es querer controlar lo que debe hacerse cuando el usuario pulsa uno de sus botones. Si definimos lo siguiente en la función MAIN de nuestro programa:

```
glutMouseFunc( ControlRaton );
```

OpenGL entiende que cada vez que se pulse uno de los botones del ratón debe llamar a una rutina llamada **ControlRaton**, que por supuesto tenemos que crear y definir nosotros mismos. Lo haremos de esta forma:

```
void ControlRaton( int button, int state, int x, int y )
      {
      <código que deseemos se ejecute>
      }
```

...donde los parámetros que la función nos da (automáticamente y sin tener que hacer nada) son los siguientes:

· **button**, un entero que puede tomar los valores GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON o GLUT_RIGHT_BUTTON según el usuario haya pulsado el botón izquierdo, el del medio o el derecho, respectivamente. Estas constantes

están predefinidas y podemos usarlas sin problema en nuestro código pues GLUT las interpretará correctamente.

· **state**, puede tomar los valores GLUT_UP o GLUT_DOWN, según si se ha pulsado/soltado el correspondiente botón.

· **X** e **Y**, son las coordenadas referidas a la ventana de visualización, no al mundo virtual, en las que se pulsó/soltó el susodicho botón.

Es importante aclarar que GLUT espera que los parámetros sean éstos en el caso de este callback, y no otros. De otra manera el programa no nos compilará.

Veamos un ejemplo:

```
void ControlRaton( int button, int state, int x, int y
      {
      if (button==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
            {
      printf( "Cerramos la aplicación...\n");
      exit(0);
      }
}
```

En este caso, cuando el usuario pulse el botón izquierdo del ratón, sacaremos un mensaje diciendo que se cierra la aplicación y entonces la cerraremos. La función exit (0) pertenece a ANSI C, no a OpenGL, y provoca el fin de la ejecución.

En el caso de la función:

```
glutMotionFunc(ControlMovimientoRaton);
```

GLUT llamará a **ControlMovimientoRaton** a intervalos discretos, es decir de tanto en tanto, mientras el ratón se esté moviendo por la pantalla. La definimos así:

```
void ControlMovimientoRaton( GLsizei x, GLsizei y )
      {
      (...)
      <código que deseemos se ejecute>
      (...)
      }
```

teniendo en cuenta que **X** e **Y** son las coordenadas de pantalla por las que el ratón está pasando. Así podríamos usar esta función para indicar nuestra situación en pantalla de la siguiente forma:

```
void ControlMovimientoRaton( GLsizei x, GLsizei y )
      {
      printf( "Posición del ratón en coordenadas de ventana es:\n");
      printf( " X = %f\n", (GLfloat)GLsizei x);
      printf( " Y = %f\n", (GLfloat)GLsizei y);
      }
```

De manera que mientras movemos el ratón se imprimen estas tres líneas en la ventana desde la que hemos ejecutado el programa, una ventana de DOS o UNIX (no en la ventana de visualización, esa es para los gráficos!!!). Y claro, los valores de X e Y se irán actualizando según nos vayamos moviendo ya que se llamará a **ControlMovimientoRaton** sucesivamente.

GLsizei es un tipo de variable numérica de OpenGL comparable a un real. Fijaros en que hago un casting para convertirla en los *printf*.

6.2. El teclado

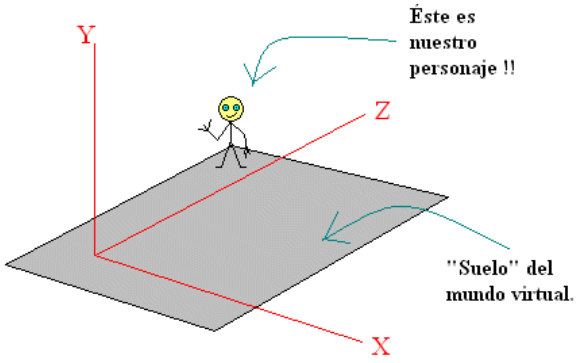
El control del teclado se realiza mediante:

```
glutKeyboardFunc( ControlTeclado );
```

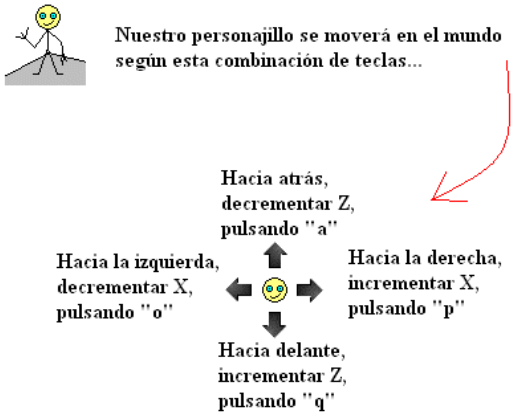
esto lo añadimos a nuestra función de MAIN y entonces definimos aparte la función de control propiamente dicha:

```
void ControlTeclado( unsigned char key, int x, int y )
      {
      (...)
      <código que deseemos se ejecute>
      (...)
      }
```

Por ejemplo supongamos que podemos "movernos" por nuestro mundo virtual. De una forma un tanto simple y primitiva, y asumiendo que el plano del suelo se corresponde con Y=0, tendríamos una situación como ésta:



y queremos movernos según esta tabla de comportamiento:



Para implementar este comportamiento necesitamos definir dos variables, XPOS y ZPOS, que contienen nuestra posición (X,Z) en el mundo. Tan sólo tendremos que incrementar/decrementar estas variables según la tecla que el usuario pulse. Por otra parte será la rutina de render (dibujado) la que nos dibujará en otra posición según nos movamos, cuando detecte que XPOS y ZPOS han cambiado.

Una primera aproximación medio codificada podría ser algo así:

```
/* Definición e inicialización de variables globales */
/* Partimos de la posición X=0 y Z=0 en el mundo */
GLfloat xpos=0, zpos=0;
/* Rutinas de Render (Dibujado) */
void DibujarMundo( ){
/*
........
........
esta rutina dibujaría todos los polígonos que componen
nuestro mundo virtual !
........
........
*/
}
void Dibujar( ){
    /* Dibujo el mundo que me rodea */
    DibujarMundo( );
    /* Representaré a mi personaje con una esfera amarilla */
    /* Activo el color amarillo */
    glColor3f(1.0, 1.0, 0.0);
    /* Las funciones referidas a matrices que se observan las
comentaré
    ampliamente en el siguiente capítulo, no os preocupéis por
ellas */
    glPushMatrix();
        /* Me trasladó a la posición concreta en el mundo */
        glTranslatef(xpos, 0.0, zpos);
        /* Dibujo una esfera de radio 2 unidades, y dividida en 16
trozos */
        glutSolidSphere(2.0, 16, 16);
    glPopMatrix();
    /* Esta función la explico más adelante en este capítulo */
    glutSwapBuffers( );
}
```

```
/* Rutina de control del teclado */
void ControlTeclado( unsigned char key, int x, int y ){
    /* Según la tecla pulsada incremento una u otra variable de
movimiento */
    switch(key){
        case "o":
            xpos++;
            break;
        case "p":
            xpos--;
            break;
        case "q":
            zpos++;
            break;
        case "a":
            zpos--;
            break;
    }
    /* Le digo a OpenGL que dibuje de nuevo cuando pueda */
    glutPostRedisplay( );
}
/* Función MAIN del programa */
int main(int argc, char** argv){
    int id;
    /* Definición típica de la ventana de visualización */
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(0, 0);
    id=glutCreateWindow("Ejemplo de control de movimiento");
    /* Definición de los Callbacks que controlaremos */
    /* Cuando haya que dibujar llamaré a ... */
    glutDisplayFunc(Dibujar);
    /* Cuando el usuario pulse una tecla llamaré a ... */
    glutKeyboardFunc(ControlTeclado);
    /* Cuando no esté haciendo nada también dibujaré ... */
    glutIdleFunc(Dibujar);
    glutMainLoop( );
    return 0;
}
```

Una cuestión importante. Al iniciar el programa, OpenGL ejecuta la función Dibujar, es decir renderiza por defecto. Cuidado porque después tenemos que forzar nosotros que se dibuje de nuevo. En nuestro caso obligamos a OpenGL a dibujar cuando se pulsa una tecla, con *glutPostRedisplay( );*. También le obligamos cuando nada esté pasando, es decir cuando el usuario no pulse nada. En ese caso se llamará a la función indicada por *glutIdleFunc*, que es precisamente *Dibujar( );*

6.3. Cambio de tamaño

El evento más importante que puede darse en nuestra ventana de visualización es un **cambio de tamaño**, es decir un *Reshape*. Para controlarlo deberemos usar:

```
glutReshapeFunc( ControlVentana );
```

que como siempre añadimos a nuestra función MAIN. Falta definir la función de control:

```
void ControlVentana( GLsizei w, GLsizei h )
      {
      <código que deseemos se ejecute>
      }
```

Los parámetros que nos llegan a la función se refieren al nuevo ANCHO (Width, w) y al nuevo ALTO (Height, h) de la ventana tras ser redimensionada por el usuario.

En esta función deberemos asegurarnos que la imagen no se distorsione, de si cambiamos o no el tamaño de lo que contiene según sus nuevas medidas, de si dibujamos todos los polígonos o por contra recortamos una parte...
Todos estos casos ya dependen de la aplicación en concreto.

6.4. MUI 

Un trabajador de SGI (Silicon Graphics) llamado Tom Davis codificó una pequeña librería a partir de GLUT llamada MUI (Micro User Interface). Lo hizo para usarla él mismo en un proyecto interno de empresa pero dada su facilidad y versatilidad de uso la incluyo de forma totalmente gratuita con GLUT.

Se trata de una serie de funciones que podemos usar fácilmente para crear ventanas con botones, barras de desplazamiento, casillas de selección y verificación....todo al estilo Motif/Windows que tanto nos gusta y sabemos manejar. Esta librería puede obtenerse juntamente con GLUT a partir de la versión 3.5 de éste.

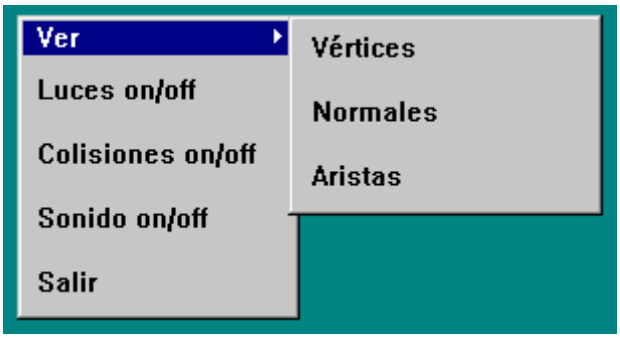
6.5. Menús

GLUT nos permite crear menús jerárquicos de varios niveles. Se activan mediante la presión de uno de los botones del ratón, el que elijamos (normalmente el derecho), y cuando estemos sobre la ventana de visualización.

Si lo que queremos son menús que cuelguen de la ventana (típicos de cualquier aplicación Windows) deberemos recurrir a algo más sofisticado como XWindows, Motif o MUI, que también los implementa.

Vamos a crear un sencillo menú asociado a la presión del botón derecho de nuestro ratón. Todo lo haremos desde el MAIN del programa:

Queremos crear este menú:



Como veis tenemos dos niveles de menú en la primera opción mientras que el resto son opciones de un sólo nivel. Esto lo codificaríamos así:

```
/* Funciones de Control del menú seleccionado */
/* Se ejecutan cuando el usuario utilice los menús */
void menu_nivel_2( int identificador){
    /* Según la opción de 2o nivel activada, ejecutaré una rutina u otra */
    switch( identificador){
        case 0:
            ControlVertices( );
           break;
        case 1:
            ControlNormales( );
            break;
        case 2:
            ControlAristas( );
            break;
    }
}
void menu_nivel_1( int identificador){
    /* Según la opción de 1er nivel activada, ejecutaré una rutina u otra */
    switch( identificador){
        case 0:
            ControlLuces( );
            break;
        case 1:
            ControlColisiones( );
            break;
        case 2:
            ControlSonido( );
            break;
        case 3:
            exit( -1 );
    }
}
int main(int argc, char** argv){
    int submenu, id;
    /* Definición de la ventana */
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(0, 0);
    id=glutCreateWindow("Ventana con menú contextual");
    /* Creación del menú */
    submenu = glutCreateMenu(menu_nivel_2);
    glutAddMenuEntry("Vértices", 0);
    glutAddMenuEntry("Normales", 1);
    glutAddMenuEntry("Aristas", 2);
    glutCreateMenu(menu_nivel_1);
    glutAddSubMenu("Ver", submenu);
    glutAddMenuEntry("Luces on/off", 0);
    glutAddMenuEntry("Colisiones on/off", 1);
    glutAddMenuEntry("Sonido on/off", 2);
    glutAddMenuEntry("Salir", 3);
    glutAttachMenu(GLUT_RIGHT_BUTTON);
```

```
    /*aquí vendrían los callbacks, el Loop, el retorno del entero, etc*/
}
```

Fijemonos en que con *glutCreateMenu*, genero un nuevo menú y además le asocio la rutina que tendrá que llamarse cuando este menú se active. En el caso del menú de primer nivel se llamará a *menu_nivel_1*, mientras que el menú de segundo nivel llamará a *menu_nivel_2*.

Al menú de primer nivel le asociamos 5 posibles opciones a activar usando *glutAddMenuEntry*. La primera, *Ver*, desplegará otro menú mientras que las restantes cuatro deben procesarse en *menu_nivel_1*. Es por eso que a cada opción se le asocia un identificador (integer), de manera que en la función *menu_nivel_1* se hace una cosa u otra dependiendo de este entero. Éste nos indica que opción se activó. Lo controlamos con un Switch de ANSI C (analizador de casos posibles).

Para el menú de segundo nivel todo es idéntico excepto que se "engancha" al de primer nivel mediante...

```
glutAddSubMenu("Ver", submenu);
```

que le dice al menú de primer nivel que la primera de sus opciones se llama *Ver* y debe llamar a un menú asociado al entero *submenu*. Mirad que este entero lo hemos asociado usando:

```
submenu = glutCreateMenu(menu_nivel_2);
```

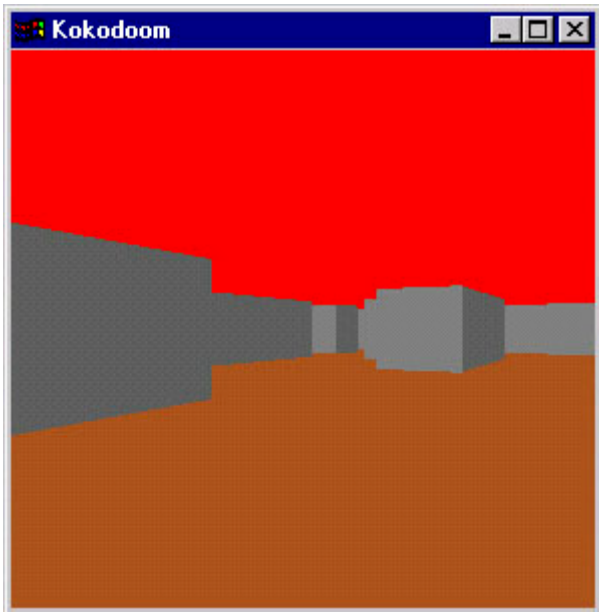
y por supuesto este segundo nivel también dispone de sus propias opciones, creadas de igual manera que antes. Por último le decimos a GLUT que este menú debe asociarse a pulsar el botón derecho del ratón con:

```
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

7. Iluminación

Hasta ahora hemos trabajado con imágenes planas (FLAT). Con esto quiero decir que cada una de las caras de nuestros objetos geométricos ha contado con un único color. Pero, respecto al realismo, es evidente que puede mejorar y mucho. En eso vamos a trabajar en este capítulo.

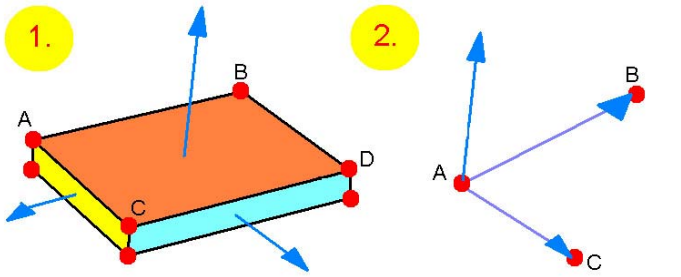
Pueden modelarse efectos de iluminación pintando las caras con distintas tonalidades de un mismo color. El efecto es muy pobre pero el resultado final ya mejora un poco. Por ejemplo en el caso de un laberinto 3D. Si los muros son cubos y los pinto con diferentes tonos de gris según la cara...dará la sensación de que hay más luz por un lado que por otro. Por ejemplo:



De qualquier manera esta es una aproximación que no tiene en cuenta la "fisica de la luz" que nos rodea.

7.1. Vectores normales a una superfície

Para iluminar una superfície (plano) necesitamos información sobre su **vector normal asociado**. De hecho en el caso de OpenGL, es necesaria la definición de un vector normal para cada uno de los vértices de nuestra geometría. Por ejemplo, en el caso de la figura:



Para obtener la normal de la superficie ABCD buscamos dos vectores pertenecientes a ésta y realizamos su producto vectorial !!!

Supongamos que tenemos un objeto 3D como el que veis. Vamos a situarnos en su cara superior, la formada por los vértices A, B, C y D. Queremos encontrar la normal de esta cara, que por lo tanto es la asociada a cada uno de los vértices que la forman. Sólo tenemos que calcular dos vectores pertenecientes a la cara y hacer su **producto vectorial** (el resultado será un vector perpendicular a ambos y por lo tanto una normal del plano).

Lo podéis ver en la figura, en el punto 2. A partir de tres vértices (A, B, C) creo dos vectores que tras su producto vectorial me generan el vector normal. Este vector ya puede asociarse a la correspondiente cara en 1.

Una vez calculada la normal tenemos que **normalizar**, es decir, dividir ese vector por su propio módulo para que sea **unitario**. De esta forma tenemos un vector normal de módulo igual a la unidad que es lo que OpenGL necesita.

Es importante el orden en que multiplicáis porque de éste depende que el vector normal apunte hacia fuera o hacia dentro de la cara. Nosotros queremos que nuestras normales apunten hacia fuera de la cara visible, ya que, como hemos visto en capítulos anteriores, le decimos a OpenGL que solo dibuje la parte visible de las caras. (FRONT).

OpenGL utilizará la normal asociada a cada vértice para evaluar la luz que incide sobre éste. Si un vértice pertenece a más de una cara, podemos o bien **promediar** para obtener unos cálculos correctos por parte de OpenGL, o bien repetir los vertices, cada uno con la normal que le corresponde. Para definir normales con OpenGL ...

```
glBegin ( GL_POLYGON ) ;
    glNormal3f ( CoordX, CoordY, CoordZ ) ;
    glVertex3f ( ... ) ;
    glVertex3f ( ... ) ;
    glVertex3f ( ... ) ;
    glVertex3f ( ... ) ;
    ...
glEnd( ) ;
```

En este caso estamos definiendo un polígono de N vértices que comparten la normal, es decir, todos tienen la misma. Si no fuera el caso lo podríamos hacer así:

```
glBegin ( GL_POLYGON ) ;
    glNormal3f ( CoordX, CoordY, CoordZ ) ;
```

```
    glVertex3f ( ... ) ;
    glNormal3f ( CoordX, CoordY, CoordZ ) ;
    glVertex3f ( ... ) ;
    glNormal3f ( CoordX, CoordY, CoordZ ) ;
    glVertex3f ( ... ) ;
    glNormal3f ( CoordX, CoordY, CoordZ ) ;
    glVertex3f ( ... ) ;
    ...
glEnd( ) ;
```

...y entonces cada vértice tendría su propia normal asociada.

En el caso de que no normalicéis los vectores normal, podéis utilizar:

```
glEnable ( GL_NORMALIZE ) ;    /* Para activar la normalización
automática */
glDisable ( GL_NORMALIZE ) ;    /* Para desactivar la normalización
automática */
```

para que OpenGL lo haga automáticamente por nosotros. No es para nada recomendable pués se carga al sistema con cálculos innecesarios que ralentizarán aún más lo que ya de por sí es computacionalmente exigente, es decir, el cálculo automático de la iluminación.

7.2. Tipos de iluminación

Ahora ya sabemos que será necesaria la especificación de una normal por vértice. Ahora vamos a ver que tipos de iluminación soporta OpenGL con lo cuál empezaremos a ver más claro el úso de estos vectores. Básicamente son tres tipos:

· Iluminación plana o **FLAT**.
· Iluminación suave o **SMOOTH** / **GOURAUD**.
· Iluminación **PHONG**.

La *iluminación plana* (FLAT) es la más simple e ineficiente. En ella todo el polígono presenta el mismo color pués OpenGL evalua sólo un color para todos sus puntos. Puede activarse mediante:

```
glShadeModel ( GL_FLAT ) ;
```

No es muy recomendable para aplicaciones dónde el realismo sea importante. Por otra parte es muy eficiente dado que los cálculos son mínimos.

En segundo lugar en cuanto a calidad tenemos a *GOURAUD*. Se activará así:

```
glShadeModel ( GL_SMOOTH ) ;
```

En este caso OpenGL si efectua cálculos de color para cada uno de los puntos del polígono. Se asocian las normales a los vértices, OpenGL cálcula los colores que éstos deben tener e implementa una **interpolación** de colores para el resto de puntos. De

esta forma ya empezamos a presenciar escenas granuladas, con degradados en la geometría. La calidad ya empieza a ser notable.

Por último la mejor de todas y por supuesto la que requiere de más proceso. En el *modelo de PHONG* se utiliza una interpolación bilineal para calcular la normal de cada punto del polígono. A diferencia del caso anterior, no se evaluan las normales sólo en los vértices y luego se interpolan colores sinó que a cada punto del polígono se le asocia una normal distinta (se interpolan las normales) con lo cuál su color asociado será muy cercano a la realidad.

OpenGL no implementa **PHONG** directamente. Así pués, deberíamos de ser nosotros los que interpolaramos normales y las asociaramos a todos los puntos. Dejaremos este caso así y nos centraremos en **GOURAUD** que ya nos dará una calidad notable y un coste computacional asumible.

Centrémonos pués. Definiremos las normales para cada vértice siempre y activaremos **FLAT** o **GOURAUD** según creamos necesario utilizando la función comentada, ¿ok?

7.3. Especificación de materiales

Antes de empezar a activar luces como unos cosacos **tenemos que definir nuestros materiales**. Para cada polígono de la escena hay que definir un material de forma que su respuesta a la incidencia de luz varíe según sea el caso.

Por tanto tenemos que decirle a OpenGL de que forma tendrá que tratar a cada trozo de geometría.

Se definen **cinco características** fundamentales para un material. Estas componentes son:

· **Reflexión difusa** (diffuse) o color de base que reflejaría el objeto si incidiera sobre él una luz pura blanca.

· **Reflexión especular** (specular), que se refiere a los "puntitos brillantes" de los objetos iluminados.

· **Reflexión ambiental** (ambient) , define como un objeto (polígono) determinado refleja la luz que no viene directamente de una fuente luminosa sinó de la escena en sí.

· **Coeficiente de brillo** o "shininess". Define la cantidad de puntos luminosos y su concentración. Digamos que variando este parámetro podemos conseguir un objeto más o menos cercano al metal por ejemplo.

· **Coeficiente de emisión** (emission) o color de la luz que emite el objeto.

Las componentes ambiental y difusa son típicamente iguales o muy semejantes. La componente especular suele ser gris o blanca. El brillo nos determinará el tamaño del punto de máxima reflexión de luz.

Se pueden especificar diferentes parámetros en cuanto a material para cada polígono. Es una tarea árdua pero lógicamente a más variedad de comportamientos más real será la escena. El funcionamiento es el normal en OpenGL. Cada vez que se llama a

la correspondiente función se activan esos valores que no cambiarán hasta llamarla de nuevo con otros. Por tanto todo lo que se "renderice" a partir de una llamada heredará esas características. La función es:

```
GLvoid glMaterialfv(GLenum face, GLenum pname,const GLfloat *params );
```

GLenum face	**GLenum pname**	**const GLfloat *params**
GL_FRONT	GL_DIFFUSE	(R, G, B, 1.0)
GL_BACK	GL_AMBIENT	(R, G, B, 1.0)
GL_FRONT_AND_BACK	GL_AMBIENT_AND_DIFFUSE	(R, G, B, 1.0)
	GL_EMISSION	(R, G, B, 1.0)
	GL_SPECULAR	(R, G, B, 1.0)
	GL_SHININESS	[0, 128]

En la tabla se observan los valores que pueden adoptar los parámetros de la función. En el caso de **face** tenemos tres posibilidades dependiendo de si la característica en cuestión debe aplicarse al lado visible (FRONT), al no visible (BACK) o a ambos. En cuanto a **pname** se define aquí cuál es la característica que vamos a definir en concreto. Las posibles son las que hemos comentado para un material. De hecho son bastante obvias si miráis las constantes que podemos usar. Por último ***params**, donde damos los valores concretos de la característica. Son tres valores, de hecho tres números reales que especifican un color RGB. Ese color define exactamente como debe verse el objeto que se renderice después en cuanto a color ambiente, difusión, componente especular, etc...

Hay una excepción en el caso de GL_SHININESS. Si usamos esta constante como segundo parámetro, el tercero tendrá que ser un número entre 0 y 128 que controlará la concentración del brillo. Por defecto este valor vale 0.

La misma función tiene también las formas glMaterialf, glMateriali y glMaterialiv. No suelen usarse por eso las versiones llamadas **escalares** (enteras) ya que sólo son útiles para definir GL_SHININESS.

Valores típicos, son los usados por defecto, son de 0.8 para las tres componentes en GL_DIFFUSE, de 0.2 para GL_AMBIENT y de 0.0 en GL_EMISSION y GL_SPECULAR. Por supuesto tendremos que retocar estos valores hasta conseguir el efecto deseado.

Mas adelante veremos el cuarto valor, el 1.0, que se refiere al valor del **canal alfa** del color RGB, que veremos mas adelante quando hablemos de transparencias.

7.4. Luces

OpenGL soporta en principio **hasta 8 luces simultaneas** en un escenario. Las luces cuentan con nombre propio del estilo GL_LIGHT0, GL_LIGHT1, GL_LIGHT2, y

así sucesivamente. Para activar / desactivar una de ellas (en este caso la cuarta luz, es decir, la número 3) :

```
glEnable ( GL_LIGHT3 ) ;
glDisable ( GL_LIGHT3 ) ;
```

También podemos activar y desactivar todo el cálculo de iluminación con:

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

Ya sabemos colocar normales en lo que dibujamos así como caracterizar de que material se supone que está hecho. Sabemos activar y desactivar fuentes de luz o los cálculos en general; no obstante no sabemos aún como poner un foco en un lugar determinado, como hacer que apunte hacía donde deseemos ni como decidir de que color es su luz.

Para ello utilizaremos la función:

```
GLvoid glLightfv(GLenum light, GLenum pname, const GLfloat *params );
```

El valor de **light** será siempre la luz a la que nos estemos refiriendo...GL_LIGHT0, GL_LIGHT1, GL_LIGHT2, etc...En cuanto a ***params**, le pasamos un array de valores RGBA reales que definen la característica en concreto. Estos valores RGBA definen el porcentaje de intensidad de cada color que tiene la luz. Si los tres valores RGB valen 1.0 la luz es sumamente brillante; si valen 0.5 la luz es aún brillante pero empieza a parecer oscura, gris... Vamos a analizar sus múltiples posibilidades.

7.4.1.Característica ambiental

Llamaremos a la función así:

```
GLvoid glLightfv ( GLenum light, GL_AMBIENT, const GLfloat *params );
```

Define la contribución de esta fuente de luz a la luz ambiental de la escena. Por defecto la contribución es nula.

7.4.2. Característica difusa

De esta forma:

```
GLvoid glLightfv ( GLenum light, GL_DIFFUSE, const GLfloat *params );
```

La componente difusa de la fuente es lo que entendemos como el color que tiene la luz. Para GL_LIGHT0 los valores RGBA por defecto valen 1.0. Para el resto de luces los valores por defecto son 0.0.

7.4.3. Característica especular

Utilizaremos:

```
GLvoid glLightfv ( GLenum light, GL_SPECULAR, const GLfloat *params );
```

Se trata de la luz que viene de una dirección particular y rebota sobre un objeto siguiendo una determinada dirección. Es la componente responsable de las zonas más brillantes en la geometría, de los "*highlights*".

Para conseguir un efecto suficientemente realista deberíamos dar a este parámetro el mismo valor que a la componente difusa. Al igual que en el caso anterior, en la primera luz los valores RGBA valen 1.0 mientras que en el resto 0.0.

7.4.4. Colocando las luces

Tenemos que especificar dónde queremos colocar cada una de las fuentes de luz. Para ello utilizamos la función de siempre:

```
GLvoid glLightfv( GLenum light, GL_POSITION, const GLfloat *params );
```

Notar el uso de la constante GL_POSITION.En este caso ***params** se corresponde con el valor de la coordenada homogénea (X, Y, Z, W) dónde colocar la luz. Si w = 0.0 se considerará que la luz se encuentra infinitamente lejos de nosotros. En ese caso su dirección se deduce del vector que pasa por el origen y por el punto (X, Y, Z). Si w = 1.0 se considera su posición con toda normalidad.

Por defecto la luz se encuentra en (0.0, 0.0, 1.0, 0.0) iluminando en la dirección negativa de las Z's. Los rayos de la luz se asumen *paralelos*.

Podemos mover una luz a gusto por una escena. Incluso podemos movernos con ella como si fuera una linterna. Para ello tan sólo tenemos que considerarla como un objeto 3D más que se ve afectado por cambios en la matriz de transformación "MODEL-VIEW" de OpenGL. Podemos rotarla, trasladarla, escalar su posición....como si de un polígono se tratara.

7.4.5. Más parámetros

Sigamos entonces con algunas propiedades más de las luces:

· Atenuación con la distancia.
· Área de cobertura.

En 1 me estoy refiriendo a la atenuación que sufre la luz a medida que se desplaza. Está claro que a más lejos esté un objeto de una fuente luminosa, menos iluminado resultará. Para modelar esto contamos con tres parámetros a definir:

· GL_CONSTANT_ATTENUATION (por defecto igual a 1.0). En la figura es "***a***".
· GL_LINEAR_ATTENUATION (por defecto igual a 0.0). En la figura es "***b***".
· GL_QUADRATIC_ATTENUATION (por defecto igual a 0.0). En la figura es "***c***".

La función que atenua la iluminación con la distancia es:

$$\frac{1}{a + b \cdot distancia + c \cdot distancia^2}$$

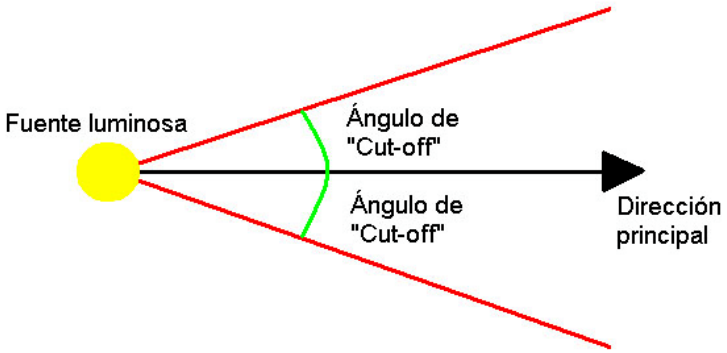
Es decir que se reduce la intensidad de la luz que llega a un determinado lugar con esta fracción, evidentemente inversamente proporcional a la distancia. Los valores los pasamos a OpenGL usando la función de siempre. Por ejemplo:

```
GLvoid glLightfv ( GL_LIGHT5, GL_CONSTANT_ATTENUATION, 0.8 ) ;
GLvoid glLightfv ( GL_LIGHT5, GL_LINEAR_ATTENUATION, 0.5 ) ;
GLvoid glLightfv ( GL_LIGHT5, GL_QUADRATIC_ATTENUATION, 0.1 ) ;
```

En este caso hemos dado más importancia a los coeficientes ***a*** y ***b*** de la función. Los hemos aplicado a la sexta de las luces (recordad que la primera es la 0).

En cuanto a 2 me estoy refiriendo al área que abarca el haz de luz que surge de la fuente. Podemos definir los siguientes parámetros:

· GL_SPOT_CUTOFF para definir un "**cono**" de luz. Podemos especificar el *ángulo de abertura* del cono con un valor de 0.0 a 90.0 grados. Si optamos por el valor 180.0º estaremos desactivando esta opción.
· GL_SPOT_DIRECTION para restringir la dirección de la luz emitida. Es un vector de tres valores reales RGB. Por defecto la dirección es la de las Z's negativas.
· GL_SPOT_EXPONENT que regula la pérdida de intensidad de la luz a medida que nos alejamos del centro del cono. Valores entre 0.0 y 128.0.
Para más claridad veamos la figura:



8. Texturización

En este capítulo veremos un aspecto básico si queremos que una escena contenga un mínimo de realismo: La texturización. Por texturización entendemos en el proceso de asignar una imagen (bitmap) a un polígono, de manera que en lugar de ver este de un color plano, o un gradiente de colores, veremos la imagen proyectada en él.

8.1. Cargando texturas en memoria

El proceso de cargar la textura en memoria, no es propio de OpenGL, lo tendremos que hacer nosotros mismos. No obstante, hay que tener en cuenta unas ciertas limitaciones que la librería nos impone. Primeramente, las dimensiones de todas las texturas que carguemos tienen que ser potencias de 2, como por ejemplo 64x64, 128x64, etc. También hemos de tener en cuenta que si estamos dibujando en RGB, sin color indexado, o bien cargamos texturas en formato RGB o las convertimos a RGB. Es decir, si cargamos una imagen GIF, que tiene color indexado, correrá de nuestra cuenta pasarla a RGB.

Sea cuál sea el método que escojamos, al final tendremos un puntero a un segmento de memoria que contiene la imagen:

```
unsigned char *textura;
```

Es importante también guardar las propiedades de la textura, en concreto sus dimensiones de ancho y alto, así como su profundidad en bits. Si estamos trabajando en RGB, la profundidad será 24bits.

8.2. Pasando las texturas a OpenGL

Ahora ya tenemos la textura en la memoria RAM. No obstante, OpenGL no puede trabajar directamente con esta memoria, ha de usar su propia memoria para guardar las texturas. El se encargará de guardarlas en su espacio de memoria RAM o, directamente, pasarlas a la tarjeta aceleradora.

Una vez le pasemos la textura a OpenGL, éste nos devolverá un identificador que tendremos de guardar. Cada textura tendrá un identificador propio, que tendremos que usar después cuando dibujemos.

Veamos el proceso de obtención de este identificador. Creemos una variable para almacenarlo:

```
GLuint idTextura;
```

A continuación llamaremos a la función glGenTextures(...), a la cual le pasamos el numero de texturas que queremos generar, y un array de identificadores donde los queremos almacenar. En este caso, solo queremos una textura, y por lo tanto no hace falta pasarle un array, sino un puntero a una variable de tipo GLuint.

```
glGenTextures(1, &idTextura);
```

Con esto OpenGL mirará cuantas texturas tiene ya almacenadas, y en función de esto pondrá en idTextura el valor del identificador. Seguidamente, usaremos la función glBindTexture(...) para asignar el valor de idTextura, a una textura de destino. Es como si activáramos la textura asignada a idTextura, y todas las propiedades que modifiquemos a partir de entonces, serán modificaciones de esa textura solamente, y no de las demás, del mismo modo que activamos una luz y definimos sus propiedades.

```
glBindTexture(GL_TEXTURE_2D, idTextura);
```

Ahora falta el paso más importante, que es pasarle la textura a OpenGL. Para ello haremos uso de la función glTexImage2D(...)

```
GlTexImage2D(GL_TEXTURE_2D, 0, 3, anchoTextura, altoTextura, 0,
GL_RGB, GL_UNSIGNED_BYTE, textura);
```

Pero veamos todos los parámetros, viendo los parámetros de esta función uno por uno:

```
void glTexImage2D(
GLenum tipoTextura,
GLint nivelMipMap,
GLint formatoInterno,
GLsizei ancho,
GLsizei alto,
GLint borde,
GLenum formato,
GLenum tipo,
const GLvoid *pixels
);
```

· **tipoTextura:** El tipo de textura que estamos tratando. Tiene que ser GL_TEXTURE_2D.

· **NivelMipMap:** El nivel de MipMapping que deseemos. De momento pondremos '0', más adelante veremos que significa.

· **FormatoInterno:** El numero de componentes en la textura. Es decir, si estamos trabajando en formato RGB, el numero de componentes será 3.

· **Ancho, alto:** El ancho y alto de la textura. Han de ser potencias de 2.

· **Borde:** La anchura del borde. Puede ser 0.

· **Formato**: El formato en que esta almacenada la textura en memoria. Nosotros usaremos GL_RGB.

· **Tipo:** El tipo de variables en los que tenemos almacenada la textura. Si la hemos almacenado en un unsigned char, usaremos GL_UNSIGNED_BYTE.

· **Pixels:** El puntero a la región de memoria donde esté almacenada la imagen.

Una observación. Teníamos en memoria RAM una textura cargada desde un archivo. Esta textura, se la hemos pasado a OpenGL, que se la ha guardado en su propia memoria. Por tanto, ahora tenemos dos copias en memoria de la misma textura, solo que una ya no es necesaria, la nuestra. Por tanto, es recomendable eliminar nuestras texturas de memoria una vez se las hemos pasado a OpenGL.

```
free(textura);
```

8.3. Parámetros de las texturas

A cada textura le podemos asignar unas ciertas características. El primero de ellos es el filtro de visualización. Una textura es un bitmap, formado por un conjunto de píxels, dispuestos de manera regular. Si la textura es de 64 x 64 píxel, i la mostramos completa en una ventana de resolución 1024x768, OpenGL escalará estos pixels, de manera que cada píxel de la textura (de ahora en adelante **téxel**) ocupará 16x12 píxeles en la pantalla.

1024 pixels ancho / 64 texels ancho = 16;
768 pixels alto / 64 texels alto = 12;

Eso quiere decir que lo que veremos serán "cuadrados" de 16x12, representando cada uno un texel de la textura. Visualmente queda muy poco realista ver una textura 'pixelizada', de manera que le aplicamos filtros para evitarlo. El más común es el 'filtro lineal', que se basa en hacer una interpolación en cada píxel en pantalla que dibuja. A pesar de que pueda parecer costoso a nivel computacional, esto se hace por hardware y no afecta en absoluto al rendimiento. Veamos como lo podemos implementar:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Con esto estamos parametrizando dos filtros. Uno para cuando la textura se representa más grande de lo que es en realidad (el ejemplo que hemos comentado) y otro para cuando la textura es mas pequeña: GL_TEXTURE_MAG_FILTER y GL_TEXTURE_MIN_FILTER, respectivamente. En los dos le decimos que haga un filtro lineal. Tambien podriamos decirle que no aplicara ningún filtro (GL_NEAREST), como podemos ver en la siguiente imagen:

En la siguiente imagen vemos la misma escena, con filtrado lineal:

Otro aspecto a parametrizar de la textura, es la manera como esta se va a repetir. Generalmente al texturizar una escena, el modelador aplica una textura pequeña que se repite a lo largo de una superficie, de manera que, sacrificando realismo, usamos menos memoria. Podemos indicar a OpenGL que prepare la textura para ser dibujada a manera de 'tile' o para ser dibujada solo una vez. Por ejemplo, un cartel en una pared. En la pared habría una textura pequeña de ladrillo, que se repetiría n veces a lo largo de toda la superficie de la pared. En cambio, el cartel solo se dibuja una sola vez. La textura de ladrillo tendríamos que preparar para que se repitiese, y la del cartel no.

Realmente lo que hace OpenGL realmente es, al dibujar la textura y aplicar el filtro lineal, es, en los bordes, interpolar con los texels del borde opuesto, para dar un aspecto mas realista. Si queremos activar esta opción, haríamos:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

Si no nos interesa este efecto:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
```

Por GL_TEXTURE_WRAP_S y GL_TEXTURE_WRAP_T nos referimos al filtro para las filas y las columnas, respectivamente.

8.4. Renderizando con texturas

Ahora que ya tenemos las texturas cargadas y ajustadas a nuestro gusto, veamos ahora cómo podemos dibujar polígonos con texturas aplicadas. Supongamos que queremos dibujar un simple cuadrado, con la textura que hemos cargado anteriormente. Si lo dibujamos sin textura seria:

```
glBegin (GL_QUADS);
      glVertex3i (-100, -100, 5);
      glVertex3i (-100, 100, 5);
      glVertex3i (100, 100, 5);
      glVertex3i (100, -100, 5);
glEnd ();
```

Veamos ahora como lo hariamos aplicando una textura:

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

glBegin (GL_QUADS);
      glTexCoord2f(0.0f, 0.0f);
      glVertex3i (-100, -100, 5);

      glTexCoord2f(1.0f, 0.0f);
      glVertex3i (-100, 100, 5);

      glTexCoord2f(1.0f, 1.0f);
      glVertex3i (100, 100, 5);

      glTexCoord2f(0.0f, 1.0f);
      glVertex3i (100, -100, 5);
glEnd ();

glDisable(GL_TEXTURE_2D);
```

Analizemos el código paso por paso:

```
glEnable(GL_TEXTURE_2D);
```

La función glEnable de OpenGL nos permite activar y desactivar propiedades que se aplican a la hora de renderizar. En este caso, le estamos diciendo que active la texturización.

```
glBindTexture(GL_TEXTURE_2D, idTextura);
```

Como ya habiamos visto antes, la función glBindTexture se encarga de activar la textura que deseemos, referenciada por el identificador que habiamos guardado previamente en idTextura.

```
glBegin (GL_QUADS);
      glTexCoord2f(0.0f, 0.0f);
      glVertex3i (-100, -100, 5);
```

Aquí vemos algo que no estaba antes, nos referimos a glTexCoord2f(). Esta funcion nos introduce un concepto nuevo: las coordenadas de textura, y nos referiremos a ellas generalmente como ‘s’ y ‘t’, siendo ‘s’ el eje horizontal y ‘t’ el vertical. Se usan para referirnos a una posición de la textura. Generalmente se mueven en el intervalo [0,

1]. La coordenada (0, 0) se refiere a la esquina inferior izquierda, y la (1, 1) a la superior derecha. Por lo tanto, si nos fijamos en el código, a cada vértice le asignamos una coordenada de textura, **siempre en orden contrario a las agujas del reloj**, sino la textura se vería al revés.

Hemos comentado que las coordenadas de textura se mueven en el intérvalo [0, 1], pero OpenGL permite otros valores. Podríamos considerar que la textura se repite infinitamente en todas las direcciones, de manera que la coordenada (1, 0) seria el mismo texel que (2, 0), (3, 0), etc. No obstante, si dibujamos un cuadrado y le asignamos a un extremo (2, 2) en lugar de (1, 1), dibujara la textura repetida 2 veces en cada dirección, en total, 4 veces. Podemos jugar y asignar valores de (-1, -1) a (1, 1), etc.

Una vez hayamos dibujado la geometría que queramos, es importante desactivar la texturización, si no lo hacemos, si mas adelante queremos dibujar algún objeto sin texturizar, OpenGL intentará hacerlo, aunque no le pasemos coordenadas de textura, produciendo efectos extraños. Así pues:

```
glDisable(GL_TEXTURE_2D);
```

8.5. Colores, luces y texturas

Hemos estado hablando de texturizar una superficie, pero, que pasa con el color? Realmente podemos seguir usándolos, es mas, OpenGL texturizará y coloreará a la vez. Podemos activar el color rojo, y todas las texturas que se dibujen a partir de entonces, saldrán tintadas de ese mismo color. Del mismo modo con las luces. No hay ningún problema en combinar colores, luces y texturas a la vez. Si no queremos iluminación alguna, basta con no activar las iluminación, y si no queremos que las texturas salgan tintadas de algún color, basta con definir el color activo como el blanco.

9. Buffers de OpenGL

En este capítulo veremos la utilidad de los Buffers disponibles en OpenGL, su utilidad y modo de hacerlos servir. En concreto veremos el buffer de color (el buffer que usamos para dibujar) el de profundidad o zbuffer, el stencil buffer y el buffer de acumulación.

Cada buffer tiene propiedades específicas que van más allá que el simple doble buffer para animación y un buffer de ocultación para eliminar caras ocultas. En OpenGL un buffer es esencialmente una matriz bidimensional de valores que se corresponden con un píxel en una ventana o en una imagen fuera de pantalla. Cada buffer tiene el mismo número de filas y columnas que el área cliente actual de la ventana, pero mantiene un tipo y rango de valores distinto.

9.1. El buffer de color

El buffer de color contiene información de color de los píxels. Cada píxel puede contener un índice de color o valores rojo/verde/azul/alfa que describen la apariencia de cada píxel. Los píxels RGBA se visualizan directamente en pantalla utilizando el color más próximo disponible. La apariencia de los pixels de color con índice viene determinada por el valor asociado a este índice en una tabla de color RGB.

9.1.2. Doble Buffer

El doble buffer proporciona un buffer de color adicional fuera de la pantalla que se usa a menudo para animación. Con el doble buffer podemos dibujar una escena fuera de la pantalla e intercambiarla rápidamente con la que está en pantalla, eliminando el molesto parpadeo.

El doble buffer sólo afecta al buffer de color y no proporciona un segundo buffer de profundidad, acumulación o estarcido. Si elegimos un formato de píxel con doble buffer, OpenGL selecciona el buffer oculto para dibujar. Podemos cambiar esto usando la función glDrawBuffer para especificar uno de los siguientes valores:

Buffer	**Descripción**
GL_FRONT	Dibujamos sólo en el buffer de color frontal (visible).
GL_BACK	Dibujamos sólo en el buffer de color trasero (oculto).
GL_FRONT_AND_BACK	Dibujamos en ambos buffers de color.

9.1.3. Intercambio de buffers

OpenGl soporta doble buffer, pero no hay ninguna función para intercambiar los buffers frontal y oculto. Afortunadamente, cada sistema de ventanas con OpenGL soporta una función para hacerlo. Bajo windows, esta llamada es:

```
SwapBuffers(hdc);
```

Donde hdc es el contexto de dispositivo de la ventana en la que estamos dibujando. Si estamos usando GLUT, éste ya se encargará de hacerlo por nosotros automáticamente, no tendremos que preocuparnos.

9.2. El buffer de profundidad

El buffer de profundidad o z-buffer mantiene valores de distancia para cada píxel. Cada valor representa la distancia al píxel desde la posición del observador, y se escala para quedar dentro del volumen de trabajo actual. Este buffer se utiliza normalmente para ejecutar la eliminación de caras ocultas, pero también puede usarse para otros efectos especiales, como realizar un corte en los objetos para ver la superficie interior.

9.2.1. Comparaciones de profundidad

Cuando dibujamos en una ventana OpenGL, la posición Z de cada píxel se compara con un valor del buffer de profundidad. Si el resultado de la comparación es true, se almacena el píxel en el buffer de profundidad junto con su profundidad. OpenGL establece ocho funciones de comparación:

Nombre	**Función**
GL_NEVER	Siempre false.
GL_LESS	True si Z de referencia<Z de profundidad.
GL_EQUAL	True si Z de referencia=Z de profundidad.
GL_LEQUAL	True si Z de referencia<=Z de profundidad.
GL_GREATER	True si Z de referencia>Z de profundidad.
GL_NOTEQUAL	True si Z de referencia!=Z de profundidad.
GL_GEQUAL	True si Z de referencia=>Z de profundidad.
GL_ALWAYS	Siempre true.

La función por defecto es GL_LESS. Para cambiarla, llamamos a:

```
glDepthFunc(función);
```

Usando la función GL_LESS, los píxels de un polígono se dibujan si su valor de profundidad es menor que el valor de profundidad situado en el buffer.

9.2.2. Valores de profundidad

Cuando utilizamos las comparaciones de profundidad GL_EQUAL y GL_NOTEQUAL, algunas veces es necesario alterar el rango de valores de profundidad utilizado, para reducir el número de valores posibles (manteniendo este número en el mínimo). Usaremos glDepthRange como sigue:

```
glDepthRange(cerca, lejos);
```

Los parámetros cerca y lejos son valores de coma flotante entre 0.0 y 1.0, que son los valores por defecto. Normalmente, cerca tiene un valor inferior a lejos, pero podemos invertir ésto para obtener efectos especiales (o utilizar las funciones GL_GREATER y GL_GEQUAL). Al reducir el rango de valores almacenados en el buffer de profundidad, no modificamos el volumen de trabajo, pero reducirá la precisión del buffer de profundidad y puede llevar a errores en la eliminación de caras ocultas.

Algunas comparaciones de profundidad necesitan un valor inicial de profundidad distinto. Por defecto, el buffer de profundidad se limpia con 1.0 con la función glClear. Para especificar un valor diferente usaremos la función glClearDepth:

```
glClearDepth(profundidad);
```

El parámetro profundidad es un valor de coma flotante entre 0.0 y 1.0, a menos que hayamos definido un rango menor con glDepthRange. En general, usaremos un valor de 0.0 para las comparaciones GL_GREATER y GL_GEQUAL, y 1.0 para las comparaciones GL_LESS y GL_LEQUAL.

9.3. El Stencil Buffer

El buffer de estarcido proporciona muchas funciones para restringir el dibujo en pantalla y tiene muchas aplicaciones que el buffer de profundidad no puede realizar. Al nivel más simple, el buffer de estarcido puede usarse para cerrar ciertas áreas de la pantalla. Quizá la aplicación más interesante del buffer de estarcido sea el sombreado. Dependiendo de nuestro hardware gráfico, podemos generar sombras por hardware y por software de múltiples fuentes de luz, haciendo que nuestras escenas sean mucho más realistas.

Para activar el buffer, haremos una llamada a:

```
glEnable(GL_STENCIL_TEST);
```

Sin esta llamada, todas las operaciones del buffer de estarcido están desactivadas.

9.3.1. Funciones del Stencil Buffer

Hay cuatro funciones de estarcido en OpenGL:

```
void glClearStencil( GLint s);
void glStencilFunc( GLenum func, GLint ref, GLuint mascara);
void glStencilMask( GLuint mascara);
void glStencilOp( GLenum fallo, GLenum zfallo, GLenum zpass);
```

La primera función es similar a glClearColor, glClearDepth y glClearIndex; proporciona el valor inicial almacenado en el buffer de estarcido cuando se llama a glClear(GL_STENCIL_BIT). Por defecto, se almacena un valor de estarcido 0 en el

buffer de estarcido. A diferencia de los buffers de color y profundidad, no siempre borramos el buffer de estarcido cada vez que actualizamos la escena.

9.3.2. Dibujando con el Stencil Buffer

Una vez activado el buffer de estarcido, aún necesitamos definir cómo opera. Por defecto, no hace nada, permitiendo que las operaciones de dibujo se ejecuten en cualquier parte de la pantalla hasta que se actualice el buffer de estarcido. Las funciones glStencilFunc y glStencilOp gestionan esta interacción.

La función glStencilFunc define una función de comparación, valor de referencia y máscara para todas las operaciones del buffer de estarcido. La siguiente tabla recoge las funciones válidas:

Nombre	Función
GL_NEVER	La verificación de estarcido siempre falla (no se dibuja).
GL_LESS	Pasa si el valor de referencia es menor que el valor de estarcido.
GL_EQUAL	Pasa si el valor de referencia es igual que el valor de estarcido.
GL_LEQUAL	Pasa si el valor de referencia es menor o igual que el valor de estarcido.
GL_GREATER	Pasa si el valor de referencia es mayor que el valor de estarcido.
GL_NOTEQUAL	Pasa si el valor de referencia no es igual que el valor de estarcido.
GL_GEQUAL	Pasa si el valor de referencia es mayor o igual que el valor de estarcido.
GL_ALWAYS	Por defecto. La verificación de estarcido siempre pasa (siempre se realiza una operación de dibujo).

Emparejada con la función de estarcido, está la *operación* de estarcido, definida con glStencilOp. Las operaciones de estarcido válidas están recogidas en la siguiente tabla:

Operación	Descripción
GL_KEEP	Mantiene los contenidos actuales del buffer de estarcido.
GL_ZERO	Establece el valor cero en el buffer de estarcido.
GL_REPLACE	Establece el valor de la función de referencia en el buffer de estarcido.
GL_INCR	Incrementa el valor actual del buffer de estarcido.
GL_DECR	Decrementa el valor actual del buffer de estarcido.
GL_INVERT	Invierte el orden binario del valor actual del buffer de estarcido.

Normalmente se utiliza una máscara para perfilar el área en la que tiene lugar el dibujo. Aquí tenemos un ejemplo de cómo dibujar una máscara en el buffer de estarcido:

```
glStencilFunc(GL_ALWAYS, 1, 1);
```

```
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
```

Entonces podemos lanzar comandos de dibujo que almacenen el valor 1 en el buffer de estarcido. Para dibujar usando la máscara del buffer de estarcido, primero hemos de llevar a cabo lo siguiente:

```
glStencilFunc(GL_EQUAL, 1, 1);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
```

Dado que tiene efecto con todas las funciones de dibujo de OpenGL, incluido glBitMap, podemos usar el buffer de estarcido para crear efectos especiales de agujero en animaciones.

9.4. El buffer de acumulación

El buffer de acumulación proporciona soporte para muchos efectos especiales como difuminado dinámico y profundidad de campo. También soporta antiescalonado a pantalla completa, aunque hay otros métodos mejor diseñados para esta tarea.

El buffer de acumulación es considerablemente menos complejo que que los otros buffers comentados hasta aquí. Sólo tiene una función, glAccum, que gestiona todas las acciones del buffer de acumulación. La siguiente tabla recoge dichas acciones:

Operación	**Descripción**
GL_ACCUM	Añade valores de color escalados al buffer de acumulación.
GL_LOAD	Abre valores de color escalados en el buffer de acumulación, sustituyendo los que hubiera antes.
GL_ADD	Añade un color constante a los valores del buffer de acumulación.
GL_MULT	Multiplica los valores de color en el buffer de acumulación por un color constante (efectos de filtro).
GL_RETURN	Copia el buffer de acumulación en el buffer de color principal.

La forma habitual en la que empleamos el buffer de acumulación es para generar en él múltiples imágenes y mostrar la escena final compuesta con:

```
glAccum(GL_RETURN, 1.0);
```

10. Efectos especiales

Haciendo uso de algunas funciones de OpenGL podremos conseguir efectos visuales que dotarán de más espectacularidad y realismo a nuestras escenas. Veamos unos cuantos de ellos.

10.1. Niebla

Tenemos funciones nativas de OpenGL que nos ofrecen la posibilidad de hacer el efecto de niebla en nuestras aplicaciones. Esta se basa en 'colorear' los objetos de un cierto color que especifiquemos nosotros, en funcion de la distancia que estos estén de la cámara.

OpenGl soporta tres tipos de niebla:

· GL_LINEAR, para indicios de profundidad,
· GL_EXP, para niebla espesa o nubes,
· GL_EXP2, para neblina y humo.

Seleccionamos el modo de niebla usando glFogi:

```
glFogi(GL_FOG_MODE, GL_LINEAR);
glFogi(GL_FOG_MODE, GL_EXP);
glFogi(GL_FOG_MODE, GL_EXP2);
```

Una vez que hemos elegido el tipo de niebla, debemos elegir un color para la niebla que se mezclará con nuestra escena usando las funciones glFogfv o glFogfi:

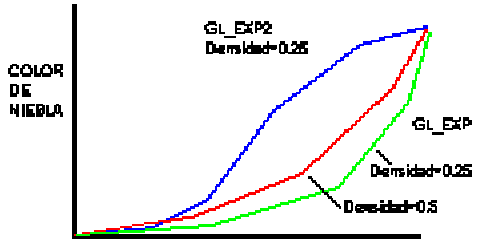
```
GLfloat color_de_niebla[4]={r,g,b,a};
glFogfv(GL_FOG_COLOR, color_de_niebla);

GLint color_de_niebla[4]={r,g,b,a};
glFogfi(GL_FOG_COLOR, color_de_niebla);
```

Para los indicios de profundidad, generalmente elegiremos el color de fondo para la niebla. Esto hará que los indicios de profundidad parezcan "correctos" al ojo, esto es, los objetos más lejanos parecerán desvanecerse con el fondo. En algunas aplicaciones, podemos querer dar un color brillante a la niebla, como amarillo, de manera que los efectos resalten más sobre el fondo. El siguiente ejemplo dibuja dos teteras usando indicios de profundidad, con el color negro (de fondo), para la niebla.

Para los otros tipos de niebla, probablemente elegiremos el color blanco o cualquier otro color luminoso. Además del color de niebla, los tipos GL_EXP y GL_EXP2 tienen un parámetro de densidad adicional:

```
glFogf(GL_FOG_DENSITY, densidad);
```

El parámetro densidad puede ser cualquier número mayor que 0.0, pero normalmente se mantiene por debajo de 0.1. La siguiente figura muestra cómo afecta la densidad de la niebla a la cantidad de color empleado.



La *distancia de niebla* es la componente Z transformada de todas las llamadas a glVertex. Esta componente Z está comprendida en el rango de 0.0 a 1.0 y es el mismo número que está almacenado en el buffer de profundidad. La distancia de niebla y la densidad determinan en qué cantidad se mezcla el color de niebla.

Por defecto, la niebla se aplica a todas las distancias entre 0.0 y 1.0. Los parámetros GL_FOG_START y GL_FOG_END restringen el rango de valores de profundidad usados en los cálculos de niebla. Esto se utiliza normalmente para definir con precisión la densidad de la niebla cuando el área inmediatamente cercana al observador no está cubierta.

10.2. Transparencias

Antes de empezar a hablar de cómo implementar transparencias en OpenGL, hablaremos un poco de la funcion de blending (mezclar). La mezcla de OpenGL proporciona un control a nivel de píxel del almacenamiento de valores RGBA en el buffer de color. Las operaciones de mezcla no pueden emplearse en el modo indexado de color y están desactivadas en las ventanas de color indexado. Para activar la mezcla en ventanas RGBA, primero debemos invocar a glEnable(GL_BLEND). Tras ésto, llamamos a glBlendFunc con dos argumentos: las funciones origen y destino para la

mezcla de color, que se especifican en las siguientes tablas. Por defecto, estos argumentos son GL_ONE y GL_ZERO, respectivamente, lo que equivale a glDisable(GL_BLEND).

Función origen	Descripción
GL_ZERO	Color fuente=0,0,0,0.
GL_ONE	Usos <?>Color fuente.
GL_DST_COLOR	El color de origen se multiplica por el color del píxel de destino.
GL_ONE_MINUS_DST_COLOR	El color de origen se multiplica por (1,1,1,1; color de destino).
GL_SRC_ALPHA	El color de origen se multiplica por el valor alfa de de origen.
GL_ONE_MINUS_SRC_ALPHA	El color de origen se multiplica por (1, valor alfa de origen).
GL_DST_ALPHA	El color de origen se multiplica por el valor alfa de destino. Microsoft OpenGL no lo soporta.
GL_ONE_MINUS_DST_ALPHA	El color de origen se multiplica por (1, valor alfa de destino). Microsoft OpenGL no lo soporta.
GL_SRC_ALPHA_SATURATE	El color de origen se multiplica por el mínimo de los valores alfa de origen y (1, valor de destino). Microsoft OpenGL no lo soporta.

Función destino	Descripción
GL_ZERO	Color destino=0,0,0,0.
GL_ONE	Usos <?>Color destino.
GL_DST_COLOR	El color de destino se multiplica por el color del píxel de origen.
GL_ONE_MINUS_DST_COLOR	El color de destino se multiplica por (1,1,1,1; color de origen).
GL_SRC_ALPHA	El color de destino se multiplica por el valor alfa de de origen.
GL_ONE_MINUS_SRC_ALPHA	El color de destino se multiplica por (1, valor alfa de origen).
GL_DST_ALPHA	El color de destino se multiplica por el valor alfa de destino. Microsoft OpenGL no lo soporta.
GL_ONE_MINUS_DST_ALPHA	El color de destino se multiplica por (1, valor alfa de

	destino). Microsoft OpenGL no lo soporta.
GL_SRC_ALPHA_SATURATE	El color de destino se multiplica por el mínimo de los valores alfa de origen y (1, valor de destino). Microsoft OpenGL no lo soporta.

La transpariencia es quizás el uso más típico de las mezclas, empleada a menudo en ventanas, botellas y otros objetos 3D a través de los cuales podemos ver. Estas son las funciones de mezcla para estas aplicaciones:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, Gl_ONE_MINUS_SRC_ALPHA);
```

Esta combinación toma el color de origen y lo escala basándose en la componente alfa, para sumarle luego el color de destino escalado en 1 menos el valor alfa. Dicho de otra manera, esta función de mezcla toma una fracción del color de dibujo actual y cubre con ella el píxel que hay en pantalla. La componente alfa del color puede valer de 0 (totalmente transparente) a 1 (totalmente opaco), como sigue:

```
Rd = Rs * As + Rd * (1 - As)
Gd = Gs * As + Gd * (1 - As)
Bd = Bs * As + Bd * (1 - As)
```

Dado que sólo se emplea la componente alfa del color de origen, no necesitamos una tarjeta gráfica que soporte planos de color alfa en el buffer de color. Algo que tenemos que recordar con la transpariencia por mezcla alfa es que la verificación normal de profundidad puede interferir con el efecto que tratamos de conseguir. Para asegurarnos de que las líneas y polígonos transparentes se dibujan apropiadamente, debemos dibujarlos siempre de atrás a adelante. Aquí tenemos un ejemplo del uso de transpariencias que dibuja una tetera opaca que se ve a través de otra transparente.

10.3. Antialiasing

El antialiasing consiste en ‘suavizar’ la imagen final. De este modo no notariamos tanto los bordes de los polígonos, las texturas aparecerian aún mas suavizadas, etc. Tenemos varias maneras de implementar antialiasing. La primera de

todas, independiente de OpenGL, es activar el antialiasing por hardware que algunas targetas aceleradoras gráficas incorporan. El método que implementan básicamente consistiria en renderizar la escena al doble o cuadruple de resolución, para luego reescalarla a la resolución original. Obviamente es un sistema costoso, pero los resultados son buenos. Por otro lado, podemos usar OpenGL para hacerlo.

La primera manera es mas simple, y consiste en aplicar blending al dibujar toda la geometria:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_LINE_SMOOTH);
glEnable(GL_POINT_SMOOTH);
glEnable(GL_POLYGON_SMOOTH);
```

Otra opción seria usar el buffer de acumulación. La estrategia básica es agitar la imagen medio píxel en todas las direcciones, para difuminar los bordes de una imagen pero no las áreas sólidas. Con sólo acumular cuatro de estas escenas "agitadas" conseguiremos imágenes considerablemente más suaves.

Realizar antiescalonado con el buffer de acumulación tiene un coste de velocidad, sin embargo. Si queremos realizar una animación en tiempo real con antiescalonado, deberemos buscar un hardware gráfico que soporte *multimuestra* para efectual el antiescalonado. El buffer de acumulación es demasiado lento para el trabajo interactivo. Si estamos generando diapositivas o animaciones paso a paso, el buffer de acumulación nos proporcionará un antiescalonado y unos efectos de profundidad de campo que simplemente no son posibles con la multimuestra.

10.4. Motion Blur

Percibimos difuminado dinámico o ‘motion blur’ cuando un objeto se mueve más rápido de lo que nuestros ojos pueden seguir. En esencia, la imagen cambia mientras el cerebro procesa la imagen, pero nunca se pierde el foco del objetivo en movimiento. En una cámara, la película está expuesta a la luz que entra por el grupo de lentes durante un tiempo finito. Dependiendo de la cámara y del fotógrafo, podemos ver un pequeño difuminado alrededor de los bordes del objeto, o puede cruzar toda la imagen. Cuando simulamos el difuminado dinámico con gráficos de ordenador, es importante recordar que la posición actual (o final) del objeto que estamos difuminando, debe aparecer más nítida (o enfocada) que el resto de la imagen. La manera más sencilla de hacer ésto es utilizar un gran factor de escalado de color cuando acumulamos el cuadro actual de manera que la mayoría de los valores de color del cuadro final resalten sobre el resto. Una implementación típica sería:

```
/*Dibuja el cuadro actual*/
draw_frame(0);

/*Almacena el 50% del cuadro actual en el buffer de acumulación*/
glAccum(GL_LOAD, 0.5);

/*Dibuja los 10 últimos cuadros y acumula el 5% de cada uno de ellos*/
for (i=1; i>=10; i++)
      {
      draw_frame(i);
```

```
      glAccum(GL_ACCUM, 0.05);
      };
/*Muestra la escena final*/
glAccum(GL_RETURN, 1.0);
```

Fijémonos en que no tenemos que usar glClear para inicializar los contenidos del buffer de acumulación, como hacemos con los buffers de color, profundidad y estarcido. En su lugar, la mayoría de las veces utilizaremos glAccum(GL_LOAD, s) en el primer cuadro de la escena.

11. Optimizaciones del render

Existen múltiples técnicas para optimizar el pintado de polígonos en OpenGL. Desde tener en cuenta el orden en que se llaman las funciones en OGL (poner mas glBegins de lo necesario) hasta usar estructuras más complejas tales como las Display Lists o los Vertex Arrays. De las primeras no hablaremos aquí, ya que las segundas estructuras de las que hemos hablado, los vertex arrays, dan un rendimiento muy superior.

11.1 Vertex Arrays

Hay dos cosas que debemos intentar hacer siempre en una aplicación gráfica:

· Enviar el mínimo posible de información por el bus.
· Enviar el mayor numero de información posible.

Lo primero lo podemos hacer optimizando nuestro programa y utilizando triangle strips, por ejemplo. Lo segundo lo podemos conseguir con los vertex arrays.

11.1.1.Como trabajar con Vertex Arrays

Para utilizar Vertex Arrays en nuestro programa, debemos organizar nuestros vértices, coordenadas de textura y color de los vértices en tres arrays distintos (que a partir de ahora llamaremos VAVertex, para el de vértices, VACoordTex, para el de coordenadas de textura, i VAColor, para el de colores).

Estos arrays han de estar organizados de tal manera que su información concuerde. Es decir, que el primer bloque de información del array de coordenadas de textura y del array de colores deben llevar la información correspondiente al primer vértice que se declare en el array de vértices, y así sucesivamente.

11.1.2. Dibujando con Vertex Arrays

Una vez tengamos los arrays hechos debemos usar la siguiente estructura para el pintado.

```
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3,GL_FLOAT, 0, VAVertex);

glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glTexCoordPointer(2,GL_FLOAT, 0, VACoordTex);

glEnableClientState(GL_COLOR_ARRAY);
glColorPointer(3,GL_FLOAT, 0, VAColor);

glDrawArrays(GL_TRIANGLES, 0, 3*numFaces);
```

En la primera línea de cada bloque indicamos cual será el array que especificaremos a continuación. Estos pueden ser GL_VERTEX_ARRAY, para el array

de vértices, GL_TEXTURE_COORD_ARRAY, para el array de coordenadas de textura o GL_COLOR_ARRAY, para el array de colores.

En la segunda línea especificamos el array en si. El primer parámetro indica el número de posiciones que tendrá cada bloque de información. El segundo parámetro indica el tipo del array. El tercero indica la separación entre bloques de información de este tipo en el array. El cuarto indica la dirección de memoria del array en si.

Una vez especificados los tres arrays podemos decir a OGL que pinte. Esto lo hacemos con la última instruccion en la que indicamos:

· Que es lo que queremos dibujar
· Desde que posicion de los arrays vamos a empezar a dibujar
· Hasta que posición de los arrays vamos a dibujar.

11.2. Optimizaciones independientes de OpenGL

Hasta ahora hemos hablado de cómo renderizar escenas usando todas las primitivas de OpenGL. Hemos comentado el concepto de clipping, es decir, dada una cámara y su ángulo de obertura, todo lo que la cámara que no ve se descarta, y no se dibuja. No obstante, este proceso implica una serie de cálculos. Si tenemos una escena muy compleja al nivel de geometría, a cada fotograma estamos procesándola toda, para luego dibujar solo una pequeña parte. Esto realmente es muy poco óptimo, estamos obligando a OpenGL a comprobar si todos y cada uno de los triángulos de la escena están dentro de la pirámide de visualización.

Comentaremos entonces algunas técnicas para organizar nuestra geometría para, mediante ciertos algoritmos, dibujar prácticamente sólo la geometría que la cámara ve. No será nuestro objetivo profundizar en ellas, sino sólo comentar su funcionamiento.

11.2.1.Quad-Trees

Ésta técnica consiste en dividir toda nuestra geometría en cubos regulares. Nuestra geometría quedaría dispuesta en una matriz casillas iguales. Si un triángulo quedara en medio de dos casillas, lo que haríamos seria partir ese triángulo.

Una vez dividida nuestra geometría, dada la posición de la cámara y su ángulo de apertura, sabríamos que casillas tendríamos que dibujar y, por tanto, que geometría hay dentro de cada casilla.

Una variación de los Quad Trees serian los Octrees, basados en la misma idea, pero en lugar de formar una matriz bidimensional, formaríamos una matriz tridimensional.

11.2.2. BSP Trees

Si con la técnica de los Quad Trees dividíamos la geometría en casillas regulares, el algoritmo de compilación del BSP nos dividiría la geometría en casillas irregulares y, además, las almacenaría en forma de árbol. De este modo, comprobar las

zonas que están dentro del volumen de visualización de la cámara tendría un coste logarítmico.

11.2.3. Portales

Esta técnica es una de las más simples, y a la vez, una de las más efectivas. Solo es viable si nuestro escenario se puede dividir en habitaciones. A cada habitación asignaríamos portales, que podría ser un rectángulo posicionado en cada una de las puertas, ventanas, etc. que comunican una habitación con otra. De este modo, al renderizar, si sabemos que estamos en una determinada habitación, y renderizamos uno de los portales, sabemos que tendremos que dibujar la habitación con la cuál comunica.

Podemos ir mas allá, y proyectar nuestra pirámide de visión a través del portal, y dibujar solo la geometría de la otra habitación que esté dentro de la pirámide proyectada. No obstante, este proceso no vale la pena, ya que este simple cálculo ya es mas costoso que dibujar la otra habitación entera.

11.2.4. PVS

Una técnica de PVS (Potentially Visible Surfaces), independientemente si usamos una de las anteriores técnicas, consiste en discriminar aquellas superficies que, estando dentro de nuestro volumen de visión, quedan ocultadas por otras superficies mas cercanas y, por tanto, no tienen que dibujarse.

11.2.5. Ordenación de las texturas

Normalmente en una escena suelen haber muchas texturas diferentes. Si dibujamos toda la geometría desordenadamente, es posible que cada pocos triángulos tengamos que cambiar de textura. El proceso de cambio de textura tiene un cierto coste, por eso es recomendable organizar nuestra geometría, haciendo grupos de superficies que tengan la misma textura, de manera que hagamos los mínimos cambios posibles. Asimismo, podemos hacer lo mismo con los materiales en el caso que hagamos uso de ellos.