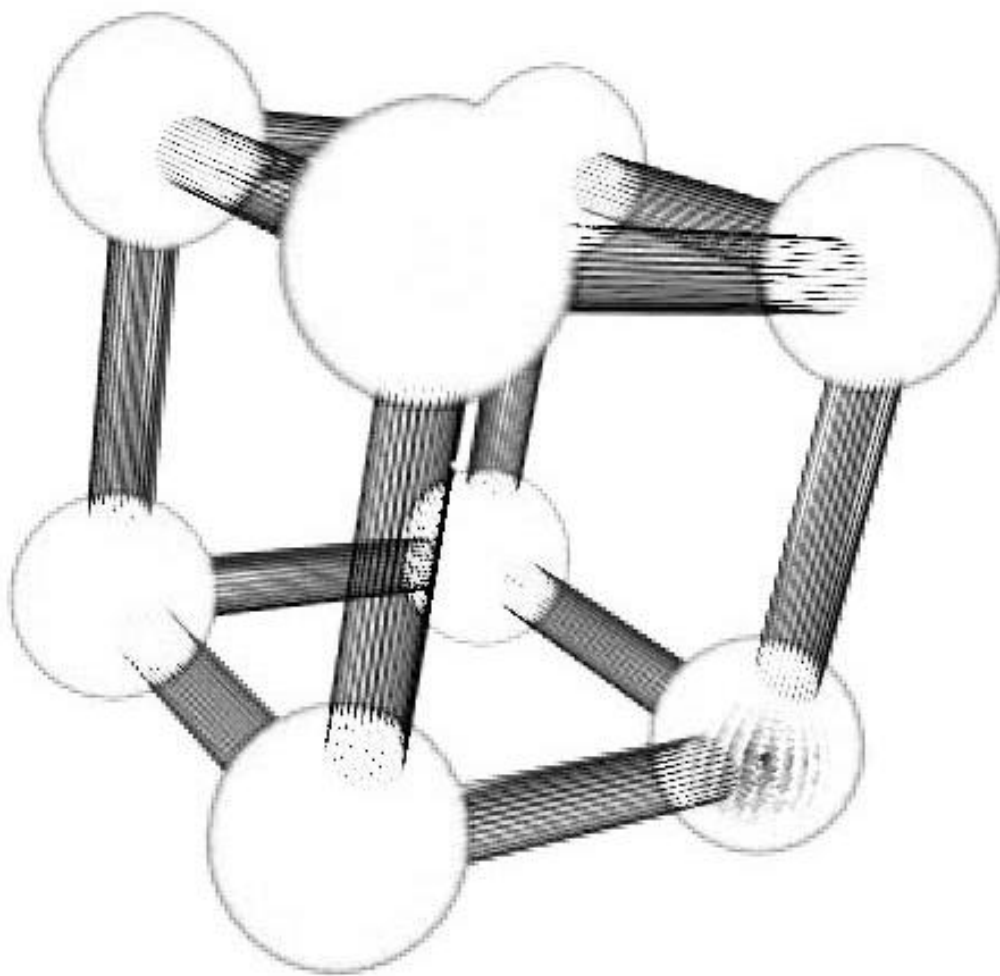


Una aproximación a OpenGL



Alberto Jaspe Villanueva
Julián Dorado de la Calle

1	<i>Introducción.....</i>	4
1.1	¿Qué es OpenGL?	4
1.2	OpenGL como una máquina de estados	4
1.3	El Pipeline de renderizado de OpenGL	5
1.4	Escribir código basado en OpenGL	6
1.4.1	Sintaxis	6
1.4.2	Animación.....	7
1.4.3	Librerías relacionadas con OpenGL	8
1.5	Pretensiones de estos apuntes	9
2	<i>El “Hello World” de OpenGL.....</i>	10
2.1	Requisitos del sistema	10
2.1.1	Hardware.....	10
2.1.2	Software	10
2.2	La OpenGL Utility Toolkit (GLUT)	10
2.3	“Hello World”	11
2.3.1	Código.....	11
2.3.2	Análisis del código	12
3	<i>Dibujando en 3D.....</i>	18
3.1	Definición de un lienzo en 3D	18
3.2	El punto en 3D: el vértice	19
3.3	Las primitivas.....	20
3.3.1	Dibujo de puntos (GL_POINTS)	20
3.3.1.1	Ajuste del tamaño del punto	21
3.3.2	Dibujo de líneas (GL_LINES)	22
3.3.3	Dibujo de polígonos.....	23
3.3.3.1	Triángulos (GL_TRIANGLES)	24
3.3.3.2	Cuadrados (GL_QUADS)	26
3.4	Construcción de objetos sólidos mediante polígonos.....	26
3.4.1	Color de relleno	26
3.4.2	Modelo de sombreado.....	27
3.4.3	Eliminación de las caras ocultas	29
4	<i>Moviéndonos por nuestro espacio 3D: transformaciones de coordenadas</i>	32
4.1	Coordenadas oculares	32

4.2	Transformaciones	33
4.2.1	El modelador.....	33
4.2.1.1	Transformaciones del observador	33
4.2.1.2	Transformaciones del modelo	34
4.2.1.3	Dualidad del modelador.....	34
4.2.2	Transformaciones de la proyección	35
4.2.3	Transformaciones de la vista	35
4.3	Matrices	35
4.3.1	El canal de transformaciones	35
4.3.2	La matriz del modelador.....	36
4.3.2.1	Translación	36
4.3.2.2	Rotación.....	37
4.3.2.3	Escalado	37
4.3.2.4	La matriz identidad	38
4.3.2.5	Las pilas de matrices.....	39
4.3.3	La matriz de proyección	40
4.3.3.1	Proyecciones ortográficas	40
4.3.3.2	Proyecciones perspectivas	41
4.4	Ejemplo: una escena simple	44
4.4.1	Código.....	44
4.4.2	Análisis del código	49

1 Introducción

1.1 ¿Qué es OpenGL?

OpenGL (ogl en adelante) es la interfaz software de hardware gráfico. Es un motor 3D cuyas rutinas están integradas en tarjetas gráficas 3D. Ogl posee todas las características necesarias para la representación mediante computadoras de escenas 3D modeladas con polígonos, desde el pintado más básico de triángulos, hasta el mapeado de texturas, iluminación o NURBS.

La compañía que desarrolla esta librería es Silicon Graphics Inc (SGI), en pro de hacer un estándar en la representación 3D gratuito y con código abierto (open source). Esta basado en sus propios OS y lenguajes IRIS, de forma que es perfectamente portable a otros lenguajes. Entre ellos C, C++, etc y las librerías dinámicas permiten usarlo sin problema en Visual Basic, Visual Fortran, Java, etc.

Ogl soporta hardware 3D, y es altamente recomendable poseer este tipo de hardware grafico. Si no se tiene disposición de el, las rutinas de representación correrán por soft, en vez de hard, decrementando en gran medida su velocidad.

1.2 OpenGL como una máquina de estados

Ogl es una maquina de estados. Cuando se activan o configuran varios estados de la maquina, sus efectos perduraran hasta que sean desactivados. Por ejemplo, si el color para pintar polígonos se pone a blanco, todos los polígonos se pintaran de este color hasta cambiar el estado de esa variable. Existen otros estados que funcionan como booleanos (on o off, 0 o 1). Estos se activa mediante las funciones glEnable y glDisable. Veremos ejemplos prácticos en los próximos capítulos.

Todos los estados tienen un valor por defecto, y también alguna función con la que conseguir su valor actual. Estas pueden ser mas generales, del tipo glGetDoublev() o glIsEnabled(), que devuelven un flotante y un valor booleano, respectivamente, en función de sus parámetros; o mas especificas, como glGetLight() o glGetError(), que devolverían una luz o un código de error.

1.3 El Pipeline de renderizado de OpenGL

La mayor parte de las implementaciones de ogl siguen un mismo orden en sus operaciones, una serie de plataformas de proceso, que en su conjunto crean lo que se suele llamar el “OpenGL Rendering Pipeline”.

El siguiente diagrama (Ilustración 1.1) describe el funcionamiento del pipeline:

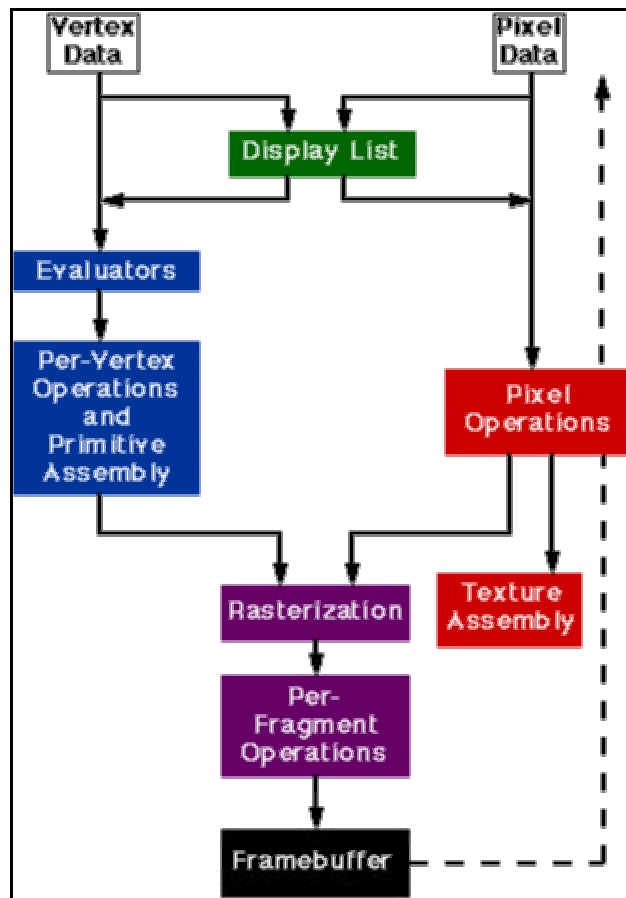


Ilustración 1.1

En este diagrama se puede apreciar el orden de operaciones que sigue el pipeline para renderizar. Por un lado tenemos el “vertex data”, que describe los objetos de nuestra

escena, y por el otro, el “píxel data”, que describe las propiedades de la escena que se aplican sobre la imagen tal y como se representa en el buffer. Ambas se pueden guardar en una “display list”, que es un conjunto de operaciones que se guardan para ser ejecutadas en cualquier momento.

Sobre el “vertex data” se pueden aplicar “evaluators”, para describir curvas o superficies parametrizadas mediante puntos de control. Luego se aplicaran las “per-vertex operations”, que convierten los vértices en primitivas. Aquí es donde se aplican las transformaciones geométricas como rotaciones, translaciones, etc., por cada vértice. En la sección de “primitive assembly”, se hace clipping de lo que queda fuera del plano de proyección, entre otros.

Por la parte de “píxel data”, tenemos las “píxel operations”. Aquí los píxeles son desempaquetados desde algún array del sistema (como el framebuffer) y tratados (escalados, etc.). Luego, si estamos tratando con texturas, se preparan en la sección “texture assembly”.

Ambos caminos convergen en la “Rasterization”, donde son convertidos en fragmentos. Cada fragmento será un píxel del framebuffer. Aquí es donde se tiene en cuenta el modelo de sombreado, la anchura de las líneas, o el antialiasing.

En la última etapa, las “per-fragment operations”, es donde se preparan los texels (elementos de texturas) para ser aplicados a cada píxel, la fog (niebla), el z-buffering, el blending, etc. Todas estas operaciones desembocan en el framebuffer, donde obtenemos el render final.

1.4 Escribir código basado en OpenGL

1.4.1 Sintaxis

Todas las funciones de ogl comienzan con el prefijo “gl” y las constantes con “GL_”. Como ejemplos, la función `glClearColor()` y la constante `GL_COLOR_BUFFER_BIT`.

En muchas de las funciones, aparece un sufijo compuesto por dos caracteres, una cifra y una letra, como por ejemplo `glColor3f()` o `glVertex3i()`. La cifra simboliza el número de parámetros que se le deben pasar a la función, y la letra el tipo de estos parámetros.

En ogl existen 8 tipos distintos de datos, de una forma muy parecida a los tipos de datos de C o C++. Además, ogl viene con sus propias definiciones de estos datos (typedef en C). Los tipos de datos:

Sufijo	Tipo de dato	Corresponde en C al tipo...	Definición en ogl del tipo
b	Entero 8-bits	signed char	GLbyte
s	Entero 16-bits	short	GLshort
i	Entero 32-bits	int o long	GLint, GLsizei
f	Punto flotante 32-bits	float	GLfloat, GLclampf
d	Punto flotante 64-bits	double	GLdouble, GLclampd
ub	Entero sin signo 8-bits	unsigned char	GLubyte, GLboolean
us	Entero sin signo 16-bits	unsigned short	GLushort
ui	Entero sin signo 32-bits	unsigned int	GLuint, GLenum, GLbitfield

1.4.2 Animación

Es muy importante dentro de los gráficos en computación la capacidad de conseguir movimiento a partir de una secuencia de fotogramas o “frames”. La animación es fundamental para un simulador de vuelo, una aplicación de mecánica industrial o un juego.

En el momento en que el ojo humano percibe mas de 24 frames en un segundo, el cerebro lo interpreta como movimiento real. En este momento, cualquier proyector convencional es capaz de alcanzar frecuencias de 60 frames/s o mas. Evidentemente, 60 fps (frames por segundo) será mas suave (y, por tanto, mas real) que 30 fps.

La clave para que la animación funcione es que cada frame esté completo (haya terminado de renderizar) cuando sea mostrado por pantalla. Supongamos que nuestro algoritmo (en pseudocódigo) para la animación es el siguiente:

```
abre_ventana();
for (i=0; i< ultimo_frame; i++) {
borra_ventana();
dibuj_a_frame(i);
espera_hasta_la_1/24_parte_de_segundo();
}
```

Si el borrado de pantalla y el dibujado del frame tardan más de 1/24 de segundo, antes de que se pueda dibujar el siguiente, el borrado ya se habrá producido. Esto causaría un parpadeo en la pantalla puesto que no hay una sincronización en los tiempos.

Para solucionar este problema, ogl nos permite usar la clásica técnica del doble-buffering: las imágenes renderizadas se van colocando en el primer buffer, y cuando termina el renderizado, se vuelca al segundo buffer, que es el que se dibuja en pantalla. Así nunca veremos una imagen cortada, solventando el problema del parpadeo. El algoritmo quedaría:

```
abre_ventana();
for (i=0; i< ultimo_frame; i++) {
borra_ventana();
dibuj_a_frame(i);
swap_buffers();
}
```

La cantidad de fps siempre quedará limitada por el refresco de nuestro monitor. Aunque OpenGL sea capaz de renderizar 100 frames en un segundo, si el periférico utilizado (monitor, cañón) solo nos alcanza los 60 hz., es decir, las 60 imágenes por segundo, aproximadamente 40 frames renderizados se perderán.

1.4.3 Librerías relacionadas con OpenGL

OpenGL contiene un conjunto de poderosos pero primitivos comandos, a muy bajo nivel. Además la apertura de una ventana en el sistema grafico que utilicemos (win32, X11, etc.) donde pintar no entra en el ámbito de OpenGL. Por eso las siguientes librerías son muy utilizadas en la programación de aplicaciones de ogl:

OpenGL Utility Library (GLU): contiene bastantes rutinas que usan ogl a bajo nivel para realizar tareas como transformaciones de matrices para tener una orientación específica, subdivisión de polígonos, etc.

GLX y WGL: GLX da soporte para maquinas que utilicen X Windows System, para inicializar una ventana, etc. WGL seria el equivalente para sistemas Microsoft. OpenGL Utility Toolkit (GLUT): es un sistema de ventanas, escrito por Mark Kilgard, que seria independiente del sistema usado, dándonos funciones tipo `abrir_ventana()`.

1.5 Pretensiones de estos apuntes

Este “tutorial” de ogl pretende enseñar al lector a “aprender OpenGL”. Ogl es un API muy extenso, con gran cantidad de funciones, estados, etc. Aquí se intentaran asentar las bases de la programación con el API de ogl, la dinámica con la que se debe empezar a escribir código para una aplicación grafica que utilice ogl.

Los ejemplos que se muestran durante esta guía están escritos utilizando C, por ser el lenguaje mas utilizado actualmente en este tipo de aplicaciones, además de ser el código “nativo” de OpenGL. También se utilizan las librerías GLU y GLUT, que serán explicadas mas adelante.

Se presuponen una serie de conocimientos matemáticos, como son el tratamiento con matrices, o la manera de hallar los vectores normales a una superficie.

Se pretende que esta guía junto con una buena referencia (como el RedBook, citado en la bibliografía) sea suficiente para comenzar con el mundo 3D.

Estos apuntes contendrán los siguientes capítulos:

El **primero**, esta introducción.

El **segundo**, un ejemplo practico para empezar a analizar la dinámica con la que se va a trabajar.

El **tercero**, aprenderemos a dibujar en tres dimensiones.

El **cuarto**, nos empezaremos a mover por nuestro entorno 3D gracias a las transformaciones matriciales.

El **quinto**, donde aprenderemos a usar los colores, materiales, iluminación y sombreado.

2 El “Hello World” de OpenGL

2.1 Requisitos del sistema

2.1.1 Hardware

En realidad, el API de ogl está pensado para trabajar bajo el respaldo de un hardware capaz de realizar las operaciones necesarias para el renderizado, pero si no se dispone de ese hardware, estas operaciones se calcularan por medio de un software contra la CPU del sistema. Así que los requerimientos hardware son escasos, aunque cuanto mayor sea las capacidades de la maquina, mayor será el rendimiento de las aplicaciones ogl.

2.1.2 Software

Para estos apuntes, supondremos la utilización de un sistema Unix, con X Windows System. Concretamente, sistema linux con X11. La librería Mesa3D, es un clon de OpenGL gratuito, y que nos vale perfectamente al ser compatible al 100% con OpenGL. Se puede bajar de <http://mesa3d.sourceforge.net>.

Lo segundo que necesitaremos será la librería GLUT, que podemos bajar directamente de SGI. La dirección es <http://www.sgi.com/software/opengl/glut.html>.

Con esto debería ser suficiente para empezar a desarrollar aplicaciones ogl sin problema.

2.2 La OpenGL Utility Toolkit (GLUT)

Como ya se ha mencionado, la librería glut esta diseñada para no tener preocupaciones con respecto al sistema de ventanas, incluyendo funciones del tipo abre_ventana(), que nos ocultan la complejidad de librerías a mas bajo nivel como la GLX. Pero además GLUT nos ofrece toda una dinámica de programación de aplicaciones ogl, gracias a la definición de funciones callback. Una función callback será llamada cada vez que se produzca un evento, como la pulsación de una tecla, el reescalado de la ventana o el mismo idle. Cuando utilizamos esta librería, le damos el control del flujo del programa

a glut, de forma que ejecutará código de diferentes funciones dependiendo del estado actual del programa (idle, el ratón cambia de posición, etc.).

2.3 “Hello World”

2.3.1 Código

Lo que podría considerarse la aplicación mas sencilla utilizando OpenGL se presenta en esta sección. El siguiente código abre una ventana y dibuja dentro un polígono (triángulo en este caso) blanco.

```
#include <GL/glut.h>

void reshape(int width, int height)
{
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1, 1, -1, 1, -1, 1);
    glMatrixMode(GL_MODELVIEW);
}

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1, 1, 1);
    glLoadIdentity();
    glBegin(GL_TRIANGLES);
        glVertex3f(-1, -1, 0);
        glVertex3f(1, -1, 0);
        glVertex3f(0, 1, 0);
    glEnd();
    glFlush();
}
```

```

}

void init()
{
    glClearColor(0, 0, 0, 0);
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition(50, 50);
    glutInitWindowSize(500, 500);
    glutCreateWindow("Hello OpenGL");
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}

```

Para compilar este código:

```
gcc -lglut -lGLU -lGL hello.c -o hello
```

2.3.2 Análisis del código

La estructura de un clásico programa sobre GLUT es la que se aprecia en el hello.c. Analicemos la función main():

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

Esta función es la que inicializa la GLUT, y negocia con el sistema de ventanas para abrir una. Los parámetros deben ser los mismos argc y argv sin modificar de la main(). Glut entiende una serie de parámetros que pueden ser pasados por línea de comandos.

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

Define el modo en el que debe dibujar en la ventana. Los parámetros, como gran parte de las funciones que iremos viendo, se definen con flags o máscaras de bits. En este caso en concreto, GLUT_SINGLE indica que se debe usar un solo buffer y GLUT_RGB el tipo de modelo de color con el que se dibujará.

```
glutInitWindowPosition(50, 50);
```

Posición x e y de la esquina superior izquierda de la nueva ventana, con respecto al escritorio en el que se trabaje.

```
glutInitWindowSize(500, 500);
```

El ancho y alto de la nueva ventana.

```
glutCreateWindow("Hello OpenGL");
```

Esta función es la que propiamente crea la ventana, y el parámetro es el nombre de la misma.

```
init();
```

En esta función que hemos definido nosotros, activamos y definimos una serie de estados de ogl, antes de pasar el control del programa a la GLUT.

```
glutDisplayFunc(display);
```

Aquí se define el primer callback. La función pasada como parámetro será llamada cada vez que GLUT determine oportuno que la ventana debe ser redibujada, como al maximizarse, poner otras ventanas por encima o sacarlas, etc.

```
glutReshapeFunc(reshape);
```

Aquí definimos otro callback, en este caso para saber que hace cuando la ventana es explícitamente reescalada. Esta acción afecta en principio directamente al render, puesto que se esta cambiando el tamaño del plano de proyección. Por eso en esta función (en este caso reshape) se suele corregir esto de alguna forma. Lo veremos mejor mas adelante.

```
glutMainLoop();
```

Esta función cede el control del flujo del programa a la GLUT, que a partir de estos "eventos", ira llamando a las funciones que han sido pasadas como callbacks.

GLUT tiene muchas mas posibilidades, por supuesto. Nos ofrece funciones para el manejo del ratón y teclado, y gran facilidad para la creación de menús, que permite vincular con un evento como el clic del ratón.

Contenido de los callbacks

Aunque no sea uno de ellos, empezaremos analizando la función `init()`. Como se ha comentado, ogl puede ser vista como una maquina de estados. Por lo tanto antes de empezar a hacer nada, habrá que configurar alguno de estos estados. En `init()` podemos apreciar:

```
glClearColor(0, 0, 0, 0);
```

Con esto se define el color con el que se borrara el buffer al hacer un `glClear()`. Los 3 primeros parámetros son las componentes R, G y B, siguiendo un rango de `[0..1]`. La última es el valor alpha, del que hablaremos mas adelante.

Veamos ahora la función `reshape()`. Esta función, al ser pasada a `glutReshapeFunc`, será llamada cada vez que se reescale a ventana. La función siempre debe ser definida con el siguiente esqueleto:

```
void reshape(int, int) { ... }
```

El primer parámetro será el ancho y el segundo el alto, después del reescalado. Con estos dos valores trabajara la función cuando, en tiempo de ejecución, el usuario reescale la ventana.

Analicemos ahora el contenido de nuestra función:

```
glViewport(0, 0, width, height);
```

Esta función define la porción de ventana donde puede dibujar ogl. Los parámetros son x e y, esquina superior izquierda del "cuadro" donde puede dibujar (con referencia la

ventana), y ancho y alto. En este caso coje el width y height, que son los parámetros de reshape(), es decir, los datos que acaba de recibir por culpa del reescalado de la ventana.

```
glMatrixMode(GL_PROJECTION);
```

Especifica la matriz actual. En ogl las operaciones de rotación, translación, escalado, etc. se realizan a través de matrices de transformación. Dependiendo de lo que estemos tratando, hay tres tipos de matriz (que son los tres posibles flags que puede llevar de parámetro la función): matriz de proyección (GL_PROJECTION), matriz de modelo (GL_MODELVIEW) y matriz de textura (GL_TEXTURE). Con esta función indicamos a cual de estas tres se deben afectar las operaciones. Concretamente, GL_PROJECTION afecta a las vistas o perspectivas o proyecciones. Todo esto se verá en el capítulo 4.

```
glLoadIdentity();
```

Con esto cargamos en el "tipo" de matriz actual la matriz identidad (es como resetear la matriz).

```
glOrtho(-1, 1, -1, 1, -1, 1);
```

glOrtho() define una perspectiva ortonormal. Esto quiere decir que lo que veremos será una proyección en uno de los planos definidos por los ejes. Es como plasmar los objetos en un plano, y luego observar el plano. Los parámetros son para delimitar la zona de trabajo, y son x_minima, x_maxima, y_minima, y_maxima, z_minima, z_maxima. Con estos seis puntos, definimos una caja que será lo que se proyecte. Esta función se trata más a fondo en el capítulo 4.

```
glMatrixMode(GL_MODELVIEW);
```

Se vuelve a este tipo de matrices, que afecta a las primitivas geométricas.

Ya solo nos queda la función display() por ver. Como se ha dicho, al ser pasada a glutDisplayFunc(), será llamada cada vez que haya que redibujar la ventana. La función debe ser definida con el siguiente esqueleto:

```
void display(void) { ... }
```

Analicemos ahora el contenido de la función en nuestro ejemplo:

```
glClear(GL_COLOR_BUFFER_BIT);
```

Borra un buffer, o una combinación de varios, definidos por flags. En este caso, el buffer de los colores lo borra (en realidad, cada componente R G y B tienen un buffer distinto, pero aquí los trata como el mismo). Para borrarlos utiliza el color que ha sido previamente definido en `init()` mediante `glClearColor()`, en este caso, el (0,0,0,0) es decir, negro. La composición de colores se verá en el capítulo 5.

```
glColor3f(1, 1, 1);
```

Selecciona el color actual con el que dibujar. Parámetros R G y B, rango [0..1], así que estamos ante el color blanco.

```
glLoadIdentity();
```

Carga la matriz identidad.

```
glBegin(GL_TRIANGLES);  
glVertex3f(-1, -1, 0);  
glVertex3f(1, -1, 0);  
glVertex3f(0, 1, 0);  
glEnd();
```

Analizamos toda esta parte entera, por formar una estructura. `glBegin()` comienza una secuencia de vértices con los que se construirán primitivas. El tipo de primitivas viene dado por el parámetro de `glBegin()`, en este caso `GL_TRIANGLES`. Al haber tres vértices dentro de la estructura, esta definiendo un triángulo. `glEnd()` simplemente cierra la estructura. Los posibles parámetros de `glBegin`, y la manera de construir primitivas se verán mas adelante, en próximo capítulo.

```
glFlush();
```

Dependiendo del hardware, controladores, etc., ogl guarda los comandos como peticiones en pila, para optimizar el rendimiento. El comando `glFlush` causa la ejecución de cualquier comando en espera.

El resultado final se puede ver en la ilustración 2.1.

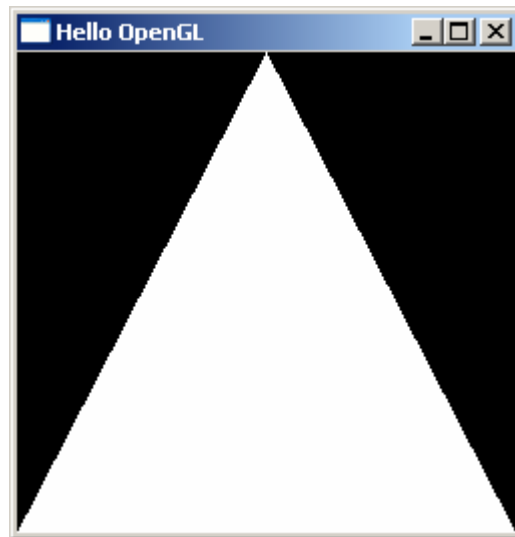


Ilustración 2.1

3 Dibujando en 3D

El dibujo 3D en OpenGL se basa en la composición de pequeños elementos, con los que vamos construyendo la escena deseada. Estos elementos se llaman primitivas. Todas las primitivas de ogl son objetos de una o dos dimensiones, abarcando desde puntos simples a líneas o polígonos complejos. Las primitivas se componen de vértices, que no son más que puntos 3D. En este capítulo se pretende presentar las herramientas necesarias para dibujar objetos en 3D a partir de éstas formas más sencillas. Para ello necesitamos deshacernos de la mentalidad en 2D de la computación gráfica clásica y definir el nuevo espacio de trabajo, ya en 3D.

3.1 Definición de un lienzo en 3D

La Ilustración 3.1 muestra un eje de coordenadas inmerso en un volumen de visualización sencillo, que utilizaremos para definir y explicar el espacio en el que vamos a trabajar. Este volumen se correspondería con una perspectiva ortonormal, como la que hemos definido en el capítulo anterior haciendo una llamada a `glOrtho()`. Como puede observarse en la figura, para el punto de vista el eje de las x sería horizontal, y crecería de izquierda a derecha; el eje y, vertical, y creciendo de abajo hacia arriba; y, por último, el eje z, que sería el de profundidad, y que crecería hacia nuestras espaldas, es decir, cuanto más lejos de la cámara esté el punto, menor será su coordenada z. En el siguiente capítulo se abordará este tema con mayor profundidad, cuando definamos las transformaciones sobre el espacio.

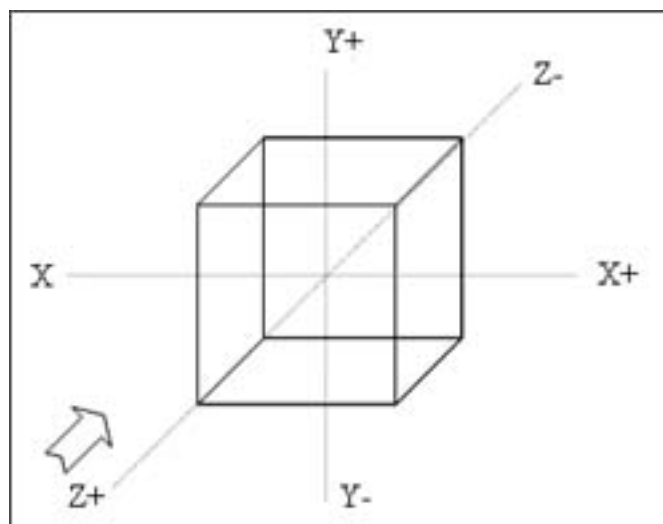


Ilustración 3.1

3.2 El punto en 3D: el vértice

Los vértices (puntos 3D) son el denominador común en cualquiera de las primitivas de OpenGL. Con ellos se definen puntos, líneas y polígonos. La función que define vértices es `glVertex`, y puede tomar de dos a cuatro parámetros de cualquier tipo numérico. Por ejemplo, la siguiente línea de código define un vértice en el punto (10, 5, 3).

```
glVertex3f(10.0f, 5.0f, 3.0f);
```

Este punto se muestra en la Ilustración 3.2. Aquí hemos decidido representar las coordenadas como valores en coma flotante, y con tres argumentos, x, y y z, respectivamente.

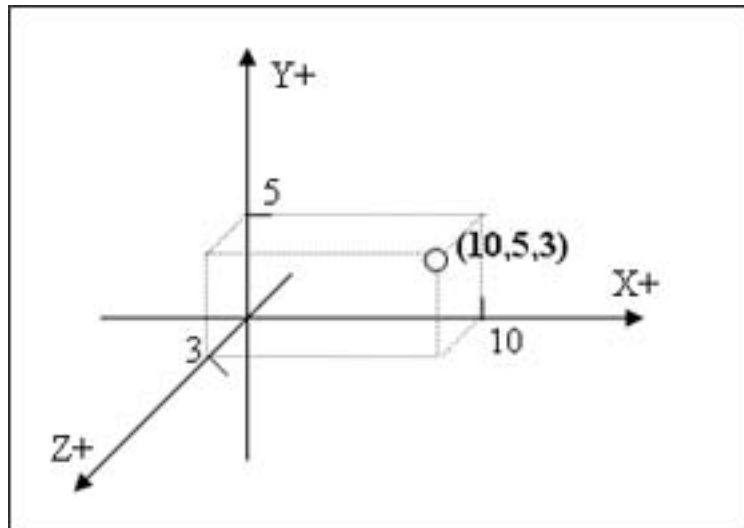


Ilustración 3.2

Ahora debemos aprender a darle sentido a estos vértices, que pueden ser tanto la esquina de un cubo como el extremo de una línea.

3.3 Las primitivas

Una primitiva es simplemente la interpretación de un conjunto de vértices, dibujados de una manera específica en pantalla. Hay diez primitivas distintas en ogl, pero en estos apuntes explicaremos solamente las más importantes: puntos (GL_POINTS), líneas (GL_LINES), triángulos (GL_TRIANGLES) y cuadrados (GL_QUADS). Comentaremos también las primitivas GL_LINES_STRIP, GL_TRIANGLE_STRIP y GL_QUAD_STRIP, utilizadas para definir “tiras” de líneas, triángulos y de cuadrados respectivamente.

Para crear primitivas en ogl se utilizan las funciones glBegin y glEnd. La sintaxis de estas funciones sigue el siguiente modelo:

```
glBegin(<tipo de primitiva>;  
    glVertex(...);  
    glVertex(...);  
    ...  
    glVertex(...);  
glEnd();
```

Puede observarse que glBegin y glEnd actúan como llaves (“{” y “}”) de las primitivas, por eso es común añadirle tabulados a las glVertex contenidos entre ellas. El parámetro de glBegin <tipo de primitiva> es del tipo GLenum (definido por OpenGL), y será el flag con el nombre de la primitiva (GL_POINTS, GL_QUADS, etc.).

3.3.1 Dibujo de puntos (GL_POINTS)

Es la más simple de las primitivas de ogl: los puntos. Para comenzar, veamos el siguiente código:

```
glBegin(GL_POINTS);  
    glVertex3f(0.0f, 0.0f, 0.0f);  
    glVertex3f(10.0f, 10.0f, 10.0f);  
glEnd();
```

El parámetro pasado a glBegin es GL_POINTS, con lo cual interpreta los vértices contenidos en el bloque glBegin-glEnd como puntos. Aquí se dibujarán dos puntos, en

(0, 0, 0) y en (10, 10, 10). Como podemos ver, podemos listar múltiples primitivas entre llamadas mientras que sean para el mismo tipo de primitiva. El siguiente código dibujara exactamente lo mismo, pero invertirá mucho mas tiempo en hacerlo, decelerando considerablemente la velocidad de nuestra aplicación:

```
glBegin(GL_POINTS);  
    glVertex3f(0.0f, 0.0f, 0.0f);  
glEnd();  
  
glBegin(GL_POINTS);  
    glVertex3f(10.0f, 10.0f, 10.0f);  
glEnd();
```

3.3.1.1 Ajuste del tamaño del punto

Cuando dibujamos un punto, el tamaño por defecto es de un pixel. Podemos cambiar este ancho con la función `glPointSize`, que lleva como parámetro un flotante con el tamaño aproximado en pixels del punto dibujado. Sin embargo, no está permitido cualquier tamaño. Para conocer el rango de tamaños soportados y el paso (incremento) de éstos, podemos usar la función `glGetFloatv`, que devuelve el valor de alguna medida o variable interna de OpenGL, llamadas “variables de estado”, que pueda depender directamente de la máquina o de la implementación de ogl. Así, el siguiente código conseguiría estos valores:

```
GLfloat rango[2];  
GLfloat incremento;  
  
glGetFloatv(GL_POINT_SIZE_RANGE, rango);  
glGetFloatv(GL_POINT_SIZE_GRANULARITY, &incremento);
```

Por ejemplo la implementación OpenGL de MicroSoft permite tamaños de punto de 0.5 a 10.0, con un incremento o paso de 0.125. Especificar un tamaño de rango incorrecto no causará un error, si no que la maquina OpenGL usará el tamaño correcto mas aproximado al definido por el usuario.

3.3.2 Dibujo de líneas (GL_LINES)

En el dibujo de puntos, la sintaxis era muy cómoda: cada vértice es un punto. En las líneas los vértices se cuentan por parejas, denotando punto inicial y punto final de la línea. Si especificamos un número impar de vértices, el último de ellos se ignora.

El siguiente código dibuja una serie de líneas radiales:

```
GLfloat angulo;  
int i;  
glBegin(GL_LINES);  
for (i=0; i<360; i+=3)  
{  
    angulo = (GLfloat)i*3.14159f/180.0f; // grados a radianes  
    glVertex3f(0.0f, 0.0f, 0.0f);  
    glVertex3f(cos(angulo), sin(angulo), 0.0f);  
}  
glEnd();
```

Este ejemplo dibuja 120 líneas en el mismo plano (ya que en los puntos que las definen $z = 0.0f$), con el mismo punto inicial (0,0,0) y puntos finales describiendo una circunferencia. El resultado sería el de la Ilustración 3.3.

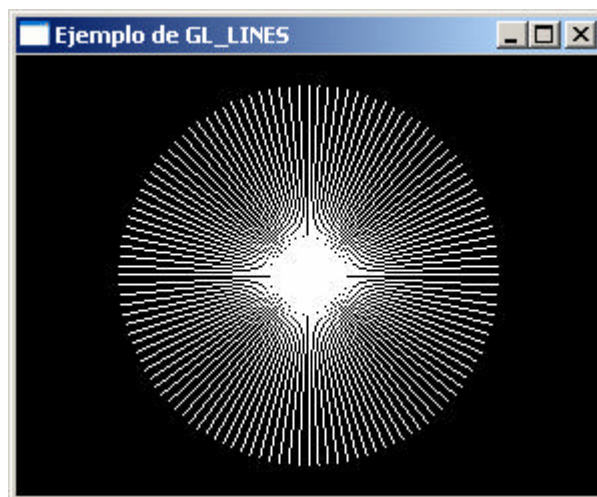


Ilustración 3.3

Si en vez de GL_LINES utilizásemos GL_LINE_STRIP, ogl ya no trataría los vértices en parejas, si no que el primer vértice y el segundo definirían una línea, y el final de ésta definiría otra línea con el siguiente vértice, y así sucesivamente, definiendo una segmento continuo. Veamos el siguiente código como ejemplo:

```
glBegin(GL_LINE_STRIP);  
    glVertex3f(0.0f, 0.0f, 0.0f); // V0  
    glVertex3f(2.0f, 1.0f, 0.0f); // V1  
    glVertex3f(2.0f, 2.0f, 0.0f); // V2  
glEnd();
```

Este código construiría las líneas como se ve en la Ilustración 3.4.

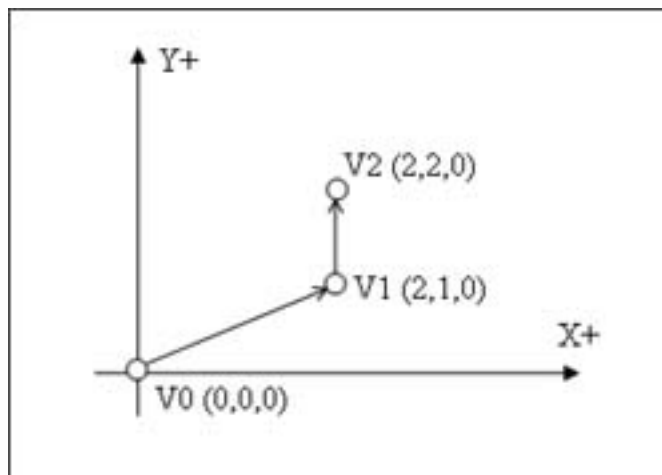


Ilustración 3.4

Mencionaremos por último la primitiva GL_LINE_LOOP, que funciona igual que GL_LINE_STRIP, pero además une el primer vértice con el último, creando siempre una cuerda cerrada.

3.3.3 Dibujo de polígonos

En la creación de objetos sólidos, el uso de puntos y líneas es insuficiente. Se necesitan primitivas que sean superficies cerradas, rellenas de uno o varios colores, que, en conjunto, modelen el objeto deseado. En el campo de la representación 3D de los gráficos en computación, se suelen utilizar polígonos (que a menudo son triángulos) para dar forma a objetos “semisólidos” (ya que en realidad son superficies, están huecos por dentro). Ahora veremos la manera de hacer esto mediante las primitivas GL_TRIANGLES y GL_QUADS.

3.3.3.1 Triángulos (GL_TRIANGLES)

El polígono más simple, es el triángulo, con sólo tres lados. En esta primitiva, los vértices van de tres en tres. El siguiente código dibuja dos triángulos, como se muestra en la Ilustración 3.5:

```
glBegin(GL_TRIANGLES);  
    glVertex3f(0.0f, 0.0f, 0.0f); // V0  
    glVertex3f(1.0f, 1.0f, 0.0f); // V1  
    glVertex3f(2.0f, 0.0f, 0.0f); // V2  
  
    glVertex3f(-1.0f, 0.0f, 0.0f); // V3  
    glVertex3f(-3.0f, 2.0f, 0.0f); // V4  
    glVertex3f(-2.0f, 0.0f, 0.0f); // V5  
glEnd();
```

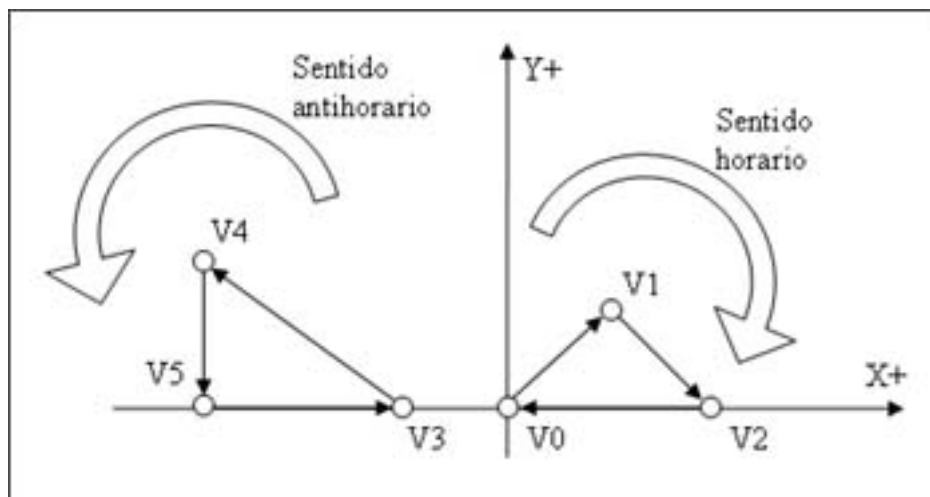


Ilustración 3.5

Hagamos hincapié en el orden en que especificamos los vértices. En el primer triángulo (el de la derecha), se sigue la política de “sentido horario”, y en el segundo, “sentido antihorario”. Cuando un polígono cualquiera, tiene sentido horario (los vértices avanzan en el mismo sentido que las agujas del reloj), se dice que es positivo; en caso contrario, se dice que es negativo. OpenGL considera que, por defecto, los polígonos que tienen sentido negativo tienen un “encare frontal”. Esto significa que el triángulo de la izquierda nos muestra su cara frontal, y el de la derecha su cara trasera. Veremos más adelante que es sumamente importante mantener una consistencia en el sentido de los triángulos al construir objetos sólidos.

Si necesitamos invertir el comportamiento por defecto de ogl, basta con una llamada a la función `glFrontFace()`. Ésta acepta como parámetro `GL_CW` (considera los polígonos positivos con encare frontal) ó `GL_CCW` (considera los polígonos negativos con encare frontal).

Como con la primitiva de líneas, con triángulos también existe `GL_TRIANGLE_STRIP`, y funciona como se puede observar en la Ilustración 3.6, que sigue el siguiente pseudo-código:

```
glBegin(GL_TRIANGLE_STRIP);
    glVertex(v0);
    glVertex(v1);
    ...
glEnd();
```

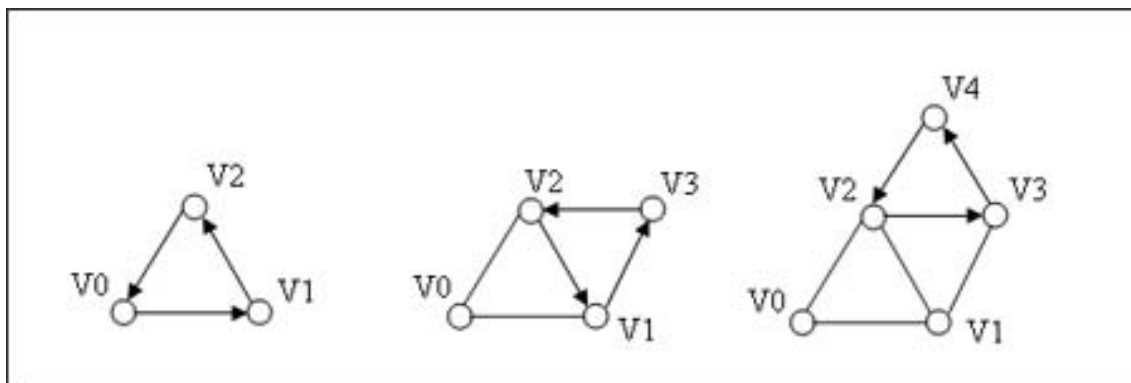


Ilustración 3.6

Por último comentar que la manera de componer objetos y dibujarlos más eficiente es mediante triángulos, ya que sólo necesitan tres vértices.

3.3.3.2 Cuadrados (GL_QUADS)

Esta primitiva funciona exactamente igual que GL_TRIANGLES, pero dibujando cuadrados. También tiene la variación de GL_QUAD_STRIP, para dibujar “tiras” de cuadrados.

3.4 Construcción de objetos sólidos mediante polígonos

Componer un objeto sólido a partir de polígonos implica algo más que ensamblar vértices en un espacio coordenado 3D. Veremos ahora una serie de puntos a tener en cuenta para construir objetos, aunque dando sólo una breve acercamiento ya que cada uno de estos puntos se verán más a fondo en los siguientes capítulos.

3.4.1 Color de relleno

Para elegir el color de los polígonos, basta con hacer una llamada a glColor entre la definición de cada polígono. Por ejemplo, modifiquemos el código que dibujaba dos triángulos:

```
glBegin(GL_TRIANGLES);  
    glColor3f(1.0f, 0.0f, 0.0f);  
    glVertex3f(0.0f, 0.0f, 0.0f);  
    glVertex3f(2.0f, 0.0f, 0.0f);  
    glVertex3f(1.0f, 1.0f, 0.0f);  
  
    glColor3f(0.0f, 1.0f, 0.0f);  
    glVertex3f(-1.0f, 0.0f, 0.0f);  
    glVertex3f(-3.0f, 2.0f, 0.0f);  
    glVertex3f(-2.0f, 0.0f, 0.0f);  
glEnd();
```

Esta modificación provocará que el primer triángulo se pinte en rojo y el segundo en verde. La función `glColor` define el color de relleno actual y lleva como parámetros los valores de las componentes RGB del color deseado, y, opcionalmente, un cuarto parámetro con el valor alpha. De esto se hablará mas adelante, aunque se adelanta que estos parámetros son flotantes que se mueven en el rango [0.0-1.0], y con ello se pueden componer todos los colores de el modo de video usado en ese instante.

3.4.2 Modelo de sombreado

Es el método que utiliza OpenGL para rellenar de color los polígonos. Se especifica con la función `glShadeModel`. Si el parámetro es `GL_FLAT`, ogl rellenará los polígonos con el color activo en el momento que se definió el último parámetro; si es `GL_SMOOTH`, ogl rellenará el polígono interpolando los colores activos en la definición de cada vértice.

Este código es un ejemplo de `GL_FLAT`:

```
glShadeModel(GL_FLAT);
glBegin(GL_TRIANGLE);
    glColor3f(1.0f, 0.0f, 0.0f); // activamos el color rojo
    glVertex3f(-1.0f, 0.0f, 0.0f);
    glColor3f(0.0f, 1.0f, 0.0f); // activamos el color verde
    glVertex3f(1.0f, 0.0f, 0.0f);
    glColor3f(1.0f, 0.0f, 0.0f); // activamos el color azul
    glVertex3f(0.0f, 0.0f, 1.0f);
glEnd();
```

La salida sería la Ilustración 3.7

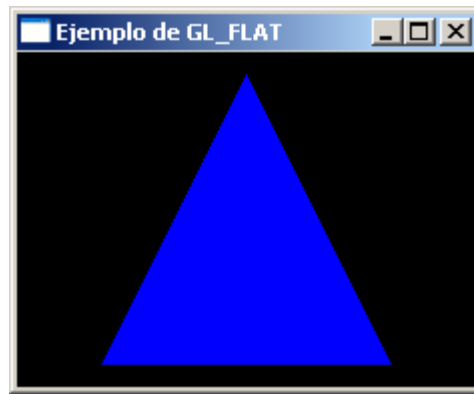


Ilustración 3.7

El triángulo se rellena con el color azul, puesto que el modelo de sombreado es GL_FLAT y el color activo en la definición del último vértice es el azul. Sin embargo, este mismo código, cambiando la primera línea:

```
glShadeModel (GL_SMOOTH);  
glBegin (GL_TRIANGLE);  
    glColor3f(1.0f, 0.0f, 0.0f); // activamos el color rojo  
    glVertex3f(-1.0f, 0.0f, 0.0f);  
    glColor3f(0.0f, 1.0f, 0.0f); // activamos el color verde  
    glVertex3f(1.0f, 0.0f, 0.0f);  
    glColor3f(1.0f, 0.0f, 0.0f); // activamos el color azul  
    glVertex3f(0.0f, 0.0f, 1.0f);  
glEnd();
```

produciría una salida similar a la de la Ilustración 3.8, donde se aprecia claramente la interpolación de colores.

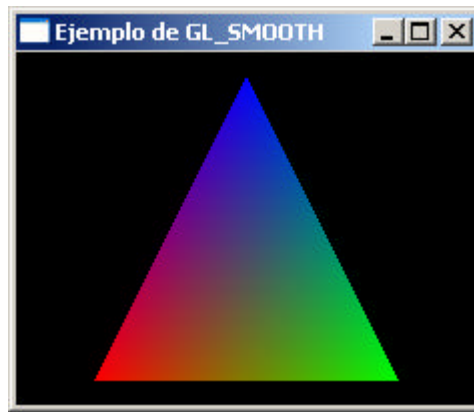


Ilustración 3.8

3.4.3 Eliminación de las caras ocultas

Cuando tenemos un objeto sólido, o quizá varios objetos, algunos de ellos estarán más próximos a nosotros que otros. Si un objeto está por detrás de otro, evidentemente, sólo veremos el de delante, que tapa al de atrás. OpenGL, por defecto, no tiene en cuenta esta posible situación, de forma que simplemente va pintando en pantalla los puntos, líneas y polígonos siguiendo el orden en el que se especifican en el código. Veamos un ejemplo:

```
glColor3f(1.0f, 1.0f, 1.0f); // activamos el color blanco
glBegin(GL_TRIANGLES);
    glVertex3f(-1.0f, -1.0f, -1.0f);
    glVertex3f(1.0f, -1.0f, -1.0f);
    glVertex3f(0.0f, 1.0f, -1.0f);
glEnd();

glPointSize(6.0f); // tamaño del pixel = 6, para que se vea bien
glColor3f(1.0f, 0.0f, 0.0f); // activamos el color rojo
glBegin(GL_POINTS);
    glVertex3f(0.0f, 0.0f, -2.0f);
    glVertex3f(2.0f, 1.0f, -2.0f);
glEnd();
```

Aquí estamos pintando un triángulo de frente a nosotros, en el plano $z = -1$. Luego pintamos dos puntos, el primero en $(0,0,-2)$ y el segundo en $(2,1,-2)$. Ambos están en el

plano $z = -2$, es decir, más lejos de nuestro punto de vista que el triángulo. El primer punto nos lo debería tapar el triángulo, pero como por defecto OpenGL no comprueba qué es lo que está por delante y por detrás, lo que hace es pintar primero el triángulo y después los puntos, quedando un resultado como el de la Ilustración 3.9.

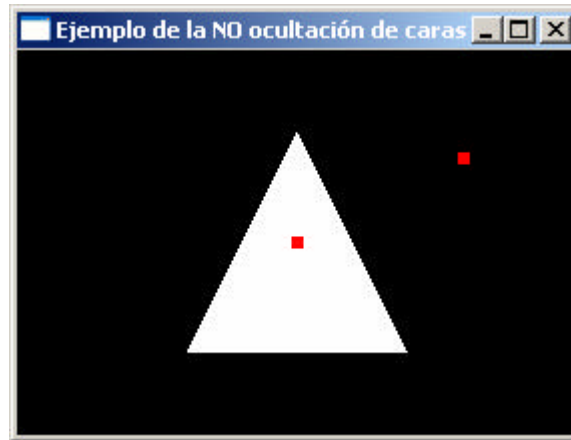


Ilustración 3.9

Para solucionar esto, introduciremos aquí uno de los buffers que OpenGL pone a nuestra disposición, el “depth buffer” (buffer de profundidad, también conocido como “z-buffer”). En él se almacenan “las zetas” o distancias desde el punto de vista a cada píxel de los objetos de la escena, y, a la hora de pintarlos por pantalla, hace una comprobación de que no haya ninguna primitiva que esté por delante tapando a lo que vamos a pintar en ese lugar.

Para activar esta característica, llamada “depth test”, solo tenemos que hacer una modificación de su variable en la máquina de estados de OpenGL, usando glEnable, de la siguiente forma:

```
glEnable(GL_DEPTH_BUFFER);
```

Para desactivarlo, usaremos la función glDisable con el mismo parámetro. Añadiendo esta última línea al principio del código del anterior ejemplo, obtendremos el efecto de ocultación deseado, como se puede ver en la Ilustración 3.10.

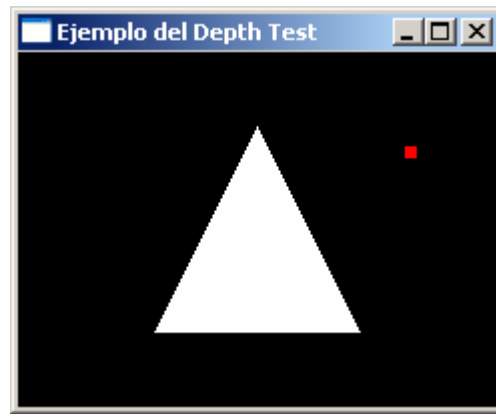


Ilustración 3.10

El uso del “test de profundidad” conlleva a que en la función que renderiza la escena, de la misma forma que hacíamos al principio un `glClear` con la variable `GL_COLOR_BUFFER_BIT`, para que a cada frame nos limpiase la pantalla antes de dibujar el siguiente, debemos indicarle también que borre el depth buffer, por si algún objeto se ha movido y ha cambiado la situación de la escena, quedando la siguiente línea:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Podemos utilizar el operador binario “or” ya que las definiciones de ogl terminadas en “_BIT” son flags, pudiendose combinar varias en una misma función.

Gracias a la eliminación de las caras ocultas, ganamos gran realismo en la escena, y al tiempo ahorramos el procesado de todos aquellos polígonos que no se ven, ganando también en velocidad.

4 Moviéndonos por nuestro espacio 3D: transformaciones de coordenadas

En este capítulo aprenderemos a mover nuestro punto de vista sobre la escena y los objetos que la componen, sobre nuestro sistema de coordenadas. Como veremos, las herramientas que OpenGL nos aporta para hacer esto están basadas en matrices de transformación, que, aplicadas sobre los sistemas de coordenadas con un orden específico, construirán la escena deseada.

En las dos primeras secciones (coordenadas oculares y transformaciones) se intentará dar una idea de cómo trabaja ogl con el espacio 3D, a alto nivel. En las siguientes, se hablará más del código necesario para conseguir los resultados deseados. Se termina con un ejemplo en el que se aplica todo lo aprendido.

4.1 Coordenadas oculares

Las coordenadas oculares se sitúan en el punto de vista del observador, sin importar las transformaciones que tengan lugar. Por tanto, estas coordenadas representan un sistema virtual de coordenadas fijo usado como marco de referencia común. En la ilustración 4.1 se pueden apreciar dos perspectivas de este sistema de coordenadas.

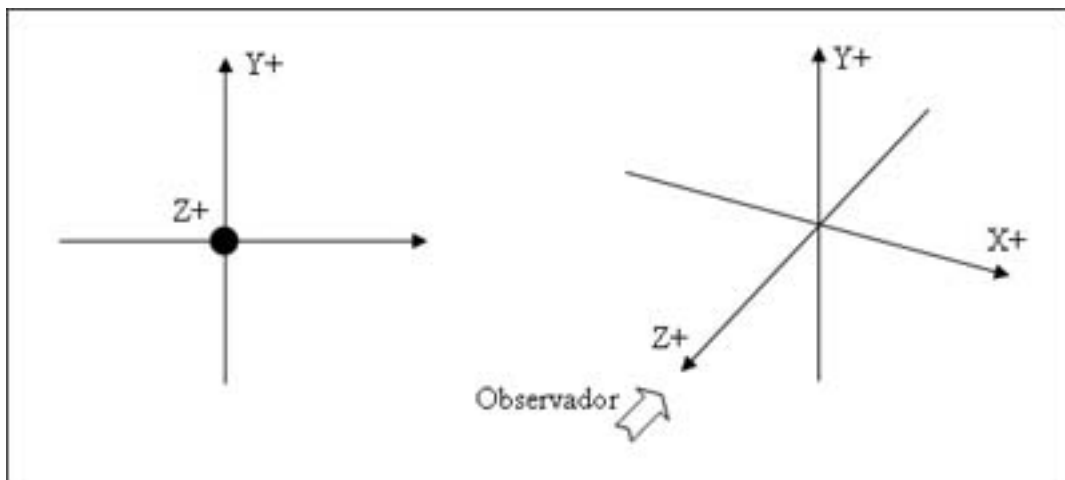


Ilustración 4.1

Cuando dibujamos en 3D con ogl, utilizamos el sistema de coordenadas cartesiano. En ausencia de cualquier transformación, el sistema en uso será idéntico al sistema de coordenadas oculares.

4.2 Transformaciones

Son las transformaciones las que hacen posible la proyección de coordenadas 3D sobre superficies 2D. También son las encargadas de mover, rotar y escalar objetos. En realidad, estas transformaciones no se aplican a los modelos en sí, si no al sistema de coordenadas, de forma que si queremos rotar un objeto, no lo rotamos a el, sino al eje sobre el que se sitúa. Las transformaciones 3D que OpenGL efectúa se pueden apreciar en la siguiente tabla:

Del observador	Especifica la localización de la cámara.
Del modelo	Mueve los objetos por la escena.
Del modelador	Describe la dualidad de las transformaciones del observador y del modelado.
De la proyección	Define y dimensiona el volumen de visualización.
De la vista	Escala la salida final a la ventana.

4.2.1 El modelador

En esta sección se recogen las transformaciones del observador y del modelado puesto que, como veremos en el apartado 4.2.1.3, constituyen al fin y al cabo la misma transformación.

4.2.1.1 Transformaciones del observador

La transformación del observador es la primera que se aplica a nuestra escena, y se usa para determinar el punto más ventajoso de la escena. Por defecto, el punto de vista está en el origen (0,0,0) mirando en dirección negativa del eje z. La transformación del observador nos permite colocar y apuntar la cámara donde y hacia donde queramos. Todas las transformaciones posteriores tienen lugar basadas en el nuevo sistema de coordenadas modificado.

4.2.1.2 Transformaciones del modelo

Estas transformaciones se usan para situar, rotar y escalar los objetos de la escena. La apariencia final de nuestros objetos depende en gran medida del orden con el que se hayan aplicado las transformaciones. Por ejemplo, en la ilustración 4.2 podemos ver la diferencia entre aplicar primero una rotación y luego una translación, y hacer esto mismo invirtiendo el orden.

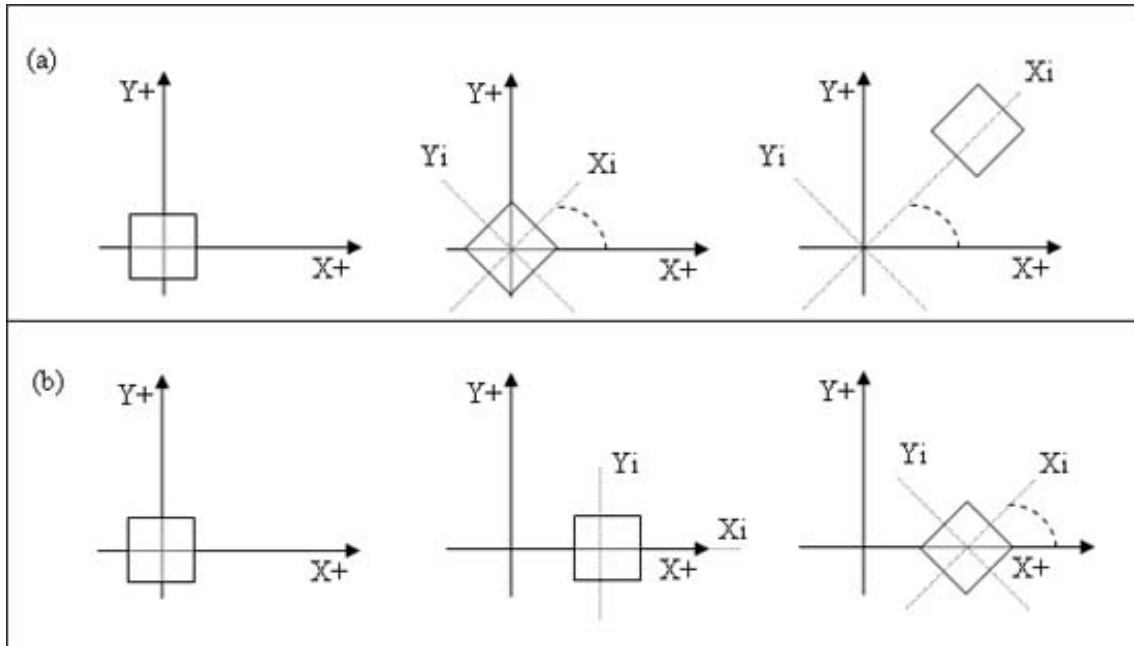


Ilustración 4.2

En el caso (a), primero se aplica una rotación, rotando así el sistema de coordenadas propio del objeto. Luego, al aplicar la translación sobre el eje x , como éste está rotado, la figura ya no se desplaza sobre el eje x del sistema de coordenadas oculares, sino sobre el suyo propio. En el segundo caso, (b), primero se hace la translación sobre el eje x , así que la figura se mueve hacia nuestra derecha, también en el sistema de coordenadas oculares, ya que ambos sistemas coinciden. Luego se hace la rotación, pero el objeto gira sobre sí mismo, ya que aún está centrado en su propio sistema de coordenadas.

4.2.1.3 Dualidad del modelador

Las transformaciones del observador y del modelo son, en realidad, la misma en términos de sus efectos internos así como en la apariencia final de la escena. Están separadas únicamente por comodidad para el programador. En realidad, no hay ninguna diferencia en mover un objeto hacia atrás o mover el sistema de coordenadas hacia delante.

4.2.2 Transformaciones de la proyección

La transformación de proyección se aplica a la orientación final del modelador. Esta proyección define el volumen de visualización y establece los planos de trabajo. A efectos prácticos, esta transformación especifica cómo se traslada una escena finalizada a la imagen final de la pantalla.

Los dos tipos de proyección más utilizados son la ortográfica y la perspectiva, que veremos más adelante.

4.2.3 Transformaciones de la vista

En el momento en que se ha terminado todo el proceso de transformaciones, solo queda un último paso: proyectar lo que hemos dibujado en 3D al 2D de la pantalla, en la ventana en la que estamos trabajando. Esta es la denominada transformación de la vista.

4.3 Matrices

Las matemáticas que hay tras estas transformaciones se simplifican gracias a las matrices. Cada una de las transformaciones de las que acabamos de hablar puede conseguirse multiplicando una matriz que contenga los vértices por una matriz que describa la transformación. Por tanto todas las transformaciones ejecutables con ogl pueden describirse como la multiplicación de dos o más matrices.

4.3.1 El canal de transformaciones

Para poder llevar a cabo todas las transformaciones de las que acabamos de hablar, deben modificarse dos matrices: la matriz del Modelador y la matriz de Proyección. OpenGL proporciona muchas funciones de alto nivel que hacen muy sencillo la construcción de matrices para transformaciones. Éstas se aplican sobre la matriz que

este activa en ese instante. Para activar una de las dos matrices utilizamos la función `glMatrixMode`. Hay dos parámetros posibles:

```
glMatrixMode(GL_PROJECTION);
```

activa la matriz de proyección, y

```
glMatrixMode(GL_MODELVIEW);
```

activa la del modelador. Es necesario especificar con que matriz se trabaja, para poder aplicar las transformaciones necesarias en función de lo que deseemos hacer.

El camino que va desde los datos “en bruto” de los vértices hasta la coordenadas en pantalla sigue el siguiente camino: Primero, nuestro vértice se convierte en una matriz 1x4 en la que los tres primeros valores son las coordenadas x,y,z. El cuarto número (llamado parámetro w) es un factor de escala, que no vamos a usar de momento, usando el valor 1.0. Entonces se multiplica el vértice por la matriz del modelador, para obtener las coordenadas oculares. Éstas se multiplican por la matriz de proyección para conseguir las coordenadas de trabajo. Con esto eliminamos todos los datos que estén fuera del volumen de proyección. Estas coordenadas de trabajo se dividen por el parámetro w del vértice, para hacer el escalado relativo del que hablamos antes. Finalmente, las coordenadas resultantes se mapean en un plano 2D mediante la transformación de la vista.

4.3.2 La matriz del modelador

La matriz del modelador es una matriz 4x4 que representa el sistema de coordenadas transformado que estamos usando para colocar y orientar nuestros objetos. Si multiplicamos la matriz de nuestro vértice (de tamaño 1x4) por ésta obtenemos otra matriz 1x4 con los vértices transformados sobre ese sistema de coordenadas.

OpenGL nos proporciona funciones de alto nivel para conseguir matrices de translación, rotación y escalado, y además la multiplican por la matriz activa en ese instante, de manera que nosotros no tenemos que preocuparnos por ello en absoluto.

4.3.2.1 Translación

Imaginemos que queremos dibujar un cubo con la función de la librería GLUT `glutSolidCube`, que lleva como parámetro el lado del cubo. Si escribimos el siguiente código

```
glutSolidCube(5);
```

obtendremos un cubo centrado en el origen (0,0,0) y con el lado de la arista 5. Ahora pensemos que queremos moverlo 10 unidades hacia la derecha (es decir, 10 unidades en el sentido positivo del eje de las x). Para ello tendríamos que construir una matriz de transformación y multiplicarla por la matriz del modelador. Ogl nos ofrece la función `glTranslate`, que crea la matriz de transformación y la multiplica por la matriz que esté activa en ese instante (en este caso debería ser la del modelador, `GL_MODELVIEW`). Entonces el código quedaría de la siguiente manera:

```
glTranslatef(5.0f, 0.0f, 0.0f);  
glutSolidCube(5);
```

La “f” añadida a la función indica que usaremos flotantes. Los parámetros de `glTranslate` son las unidades a desplazar en el eje x, y y z, respectivamente. Pueden ser valores negativos, para trasladar en el sentido contrario.

4.3.2.2 Rotación

Para rotar, tenemos también una función de alto nivel que construye la matriz de transformación y la multiplica por la matriz activa, `glRotate`. Lleva como parámetros el ángulo a rotar (en grados, sentido horario), y después x, y y z del vector sobre el cual queremos rotar el objeto. Una rotación simple, sobre el eje y, de 10° sería:

```
glRotatef(10, 0.0f, 1.0f, 0.0f);
```

4.3.2.3 Escalado

Una transformación de escala incrementa el tamaño de nuestro objeto expandiendo todos los vértices a lo largo de los tres ejes por los factores especificados. La función `glScale` lleva como parámetros la escala en x, y y z, respectivamente. El valor 1.0f es la referencia de la escala, de tal forma que la siguiente línea:

```
glScalef(1.0f, 1.0f, 1.0f);
```

no modificaría el objeto en absoluto. Un valor de $2.0f$ sería el doble, y $0.5f$ sería la mitad. Por ejemplo, para ensanchar un objeto a lo largo de su eje z , de tal forma que quedase cuatro veces más “alargado” en este eje, sería:

```
glScalef(1.0f, 1.0f, 4.0f);
```

4.3.2.4 La matriz identidad

El “problema” del uso de estas funciones surge cuando tenemos más de un objeto en la escena. Estas funciones tienen efectos acumulativos. Es decir, si queremos preparar una escena como la de la ilustración 4.3,

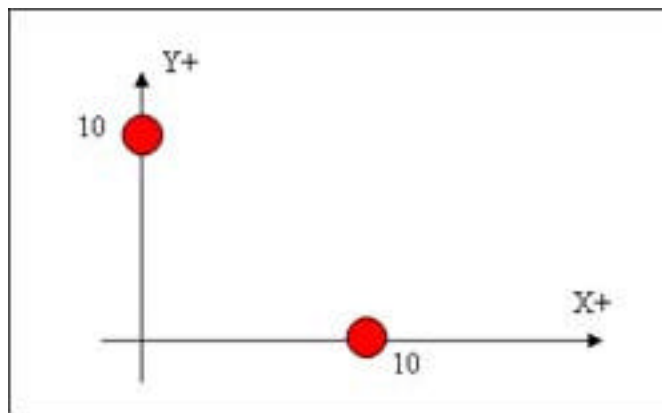


Ilustración 4.3

con una esfera (de radio 3) centrada en $(0,10,0)$ y otra centrada en $(10,0,0)$, escribiríamos el siguiente código, que es incorrecto:

```
glTranslatef(0.0f, 10.0f, 0.0f);  
glutSolidSphere(3.0f);  
glTranslatef(10.0f, 0.0f, 0.0f);  
glutSolidSphere(3.0f);
```

En este código, dibujamos primero una esfera en $(0,10,0)$ como queríamos. Pero después, estamos multiplicando la matriz del modelador que teníamos (que ya estaba transformada para dibujar la primera esfera) por otra matriz de transformación que nos

desplaza 10 unidades hacia la derecha. Por ello la segunda matriz se dibujaría, como se puede ver en la ilustración 4.4, en (10,10,0), y no en (10,0,0), como pretendíamos.

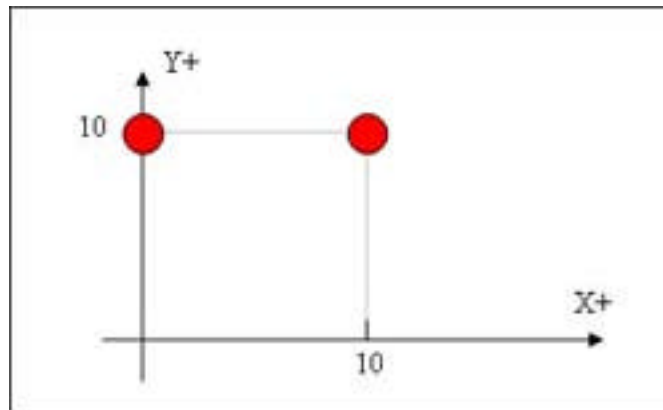


Ilustración 4.4

Para solventar este problema debemos reiniciar la matriz del modelador a un estado mas conocido, en este caso, centrada en el origen de nuestro sistema de coordenadas oculares. Para ello se carga en la matriz del modelador la matriz identidad (una matriz 4x4 llena de ceros excepto en la diagonal, que contiene unos). Esto se consigue gracias a la función `glLoadIdentity()`, que no lleva parámetros. Simplemente carga la matriz identidad en la matriz activa en ese instante. El código correcto para el ejemplo anterior quedaría de la siguiente manera:

```
glTranslatef(0.0f, 10.0f, 0.0f);  
glutSolidSphere(3.0f);  
glLoadIdentity();  
glTranslatef(10.0f, 0.0f, 0.0f);  
glutSolidSphere(3.0f);
```

4.3.2.5 Las pilas de matrices

No siempre es deseable reiniciar la matriz del modelador con la identidad antes de colocar cada objeto. A menudo queremos almacenar el estado actual de transformación y entonces recuperarlo después de haber colocado varios objetos. Para ello, `ogl` mantiene una pila de matrices para el modelador (`GL_MODELVIEW`) y otra para la proyección (`GL_PROJECTION`). La profundidad varía dependiendo de la plataforma y puede obtenerse mediante las siguientes líneas:

```
glGet(GL_MAX_MODELVIEW_DEPTH);  
glGet(GL_MAX_PROJECTION_DEPTH);
```

Por ejemplo, para la implementación de MicroSoft de ogl, éstos valores son de 32 para GL_MODELVIEW y 2 para GL_PROJECTION.

Para meter una matriz en su pila correspondiente se usa la función glPushMatrix (sin parámetros), y para sacarla glPopMatrix (sin parámetros tampoco).

4.3.3 La matriz de proyección

La matriz de proyección especifica el tamaño y la forma de nuestro volumen de visualización. El volumen de visualización es aquel volumen cuyo contenido es el que representaremos en pantalla. Éste está delimitado por una serie de planos de trabajo, que lo delimitan. De estos planos, los más importantes son los planos de corte, que son los que nos acotan el volumen de visualización por delante y por detrás. En el plano más cercano a la cámara (znear), es donde se proyecta la escena para luego pasarla a la pantalla. Todo lo que está más adelante del plano de corte más alejado de la cámara (zfar) no se representa.

Veremos los distintos volúmenes de visualización de las dos proyecciones más usadas: ortográficas y perspectivas.

Cuando cargamos la identidad en la matriz de proyección, la diagonal de unos especifica que los planos de trabajo se extienden desde el origen hasta los unos positivos en todas las direcciones. Como antes, veremos ahora que existen funciones de alto nivel que nos facilitan todo el proceso.

4.3.3.1 Proyecciones ortográficas

Una proyección ortográfica es cuadrada en todas sus caras. Esto produce una proyección paralela, útil para aplicaciones de tipo CAD o dibujos arquitectónicos, o también para tomar medidas, ya que las dimensiones de lo que representan no se ven alteradas por la proyección.

Una aproximación menos técnica pero más comprensible de esta proyección es imaginar que tenemos un objeto fabricado con un material deformable, y lo aplastamos literalmente como una pared. Obtendríamos el mismo objeto, pero plano, liso. Pues eso es lo que veríamos por pantalla.

Para definir la matriz de proyección ortográfica y multiplicarla por la matriz activa (que debería ser en ese momento la de proyección, `GL_PROJECTION`), utilizamos la función `glOrtho`, que se define de la siguiente forma

```
glOrtho(límiteIzquierdo, límiteDerecho, límiteAbajo, límiteArriba,  
znear, zfar)
```

siendo todos flotantes. Con esto simplemente acotamos lo que será nuestro volumen de visualización (un cubo).

Por ejemplo, la ilustración 4.5 es un render de un coche con proyección ortográfica, visto desde delante.



Ilustración 4.5

El código utilizado para esta proyección ha sido

```
glOrtho(-0.5f, 0.5f, -0.5f, 0.5f, 0.01f, 20.0f);
```

En la siguiente sección, se podrá apreciar la diferencia usando una proyección perspectiva.

4.3.3.2 Proyecciones perspectivas

Una proyección en perspectiva ejecuta una división en perspectiva para reducir y estirar los objetos más alejados del observador. Es importante saber que las medidas de la

proyección no tienen por qué coincidir con las del objeto real, ya que han sido deformadas.

El volumen de visualización creado por una perspectiva se llama frustum. Un frustum es una sección piramidal, vista desde la parte afilada hasta la base (ilustración 4.6).

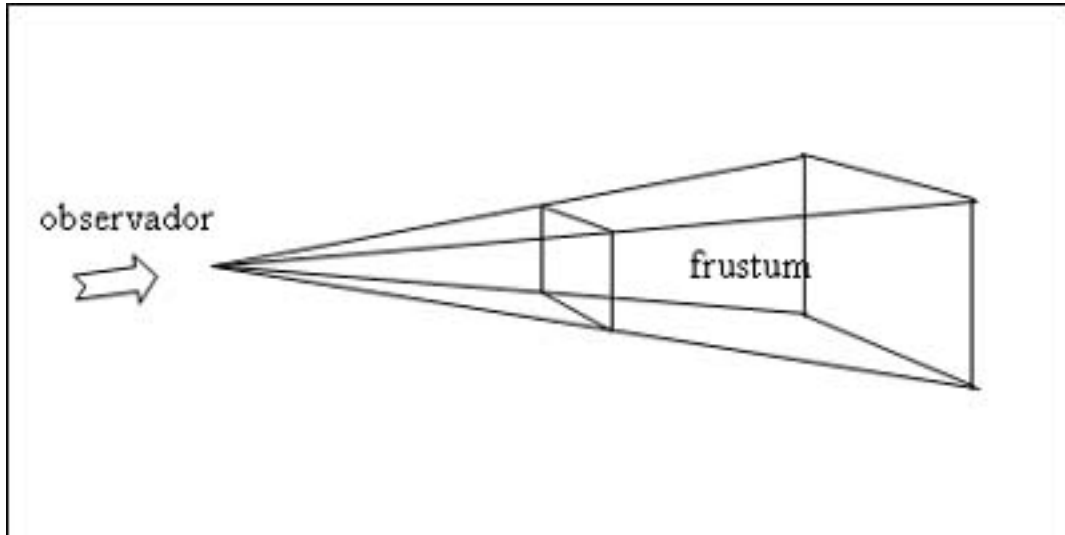


Ilustración 4.6

Podemos definir esta proyección utilizando la función `glFrustum`. Pero existe otra función de la librería GLU llamada `gluPerspective` que hace el proceso más sencillo. Se define de la siguiente forma:

```
Void gluPerspective(angulo, aspecto, znear, zfar);
```

Los parámetros de `gluPerspective` son flotantes, y definen las características mostradas en la ilustración 4.7: el ángulo para el campo de visión en sentido vertical, el aspecto es la relación entre la altura(h) y la anchura(w) y las distancias `znear` y `zfar` de los planos que acotan el frustum.

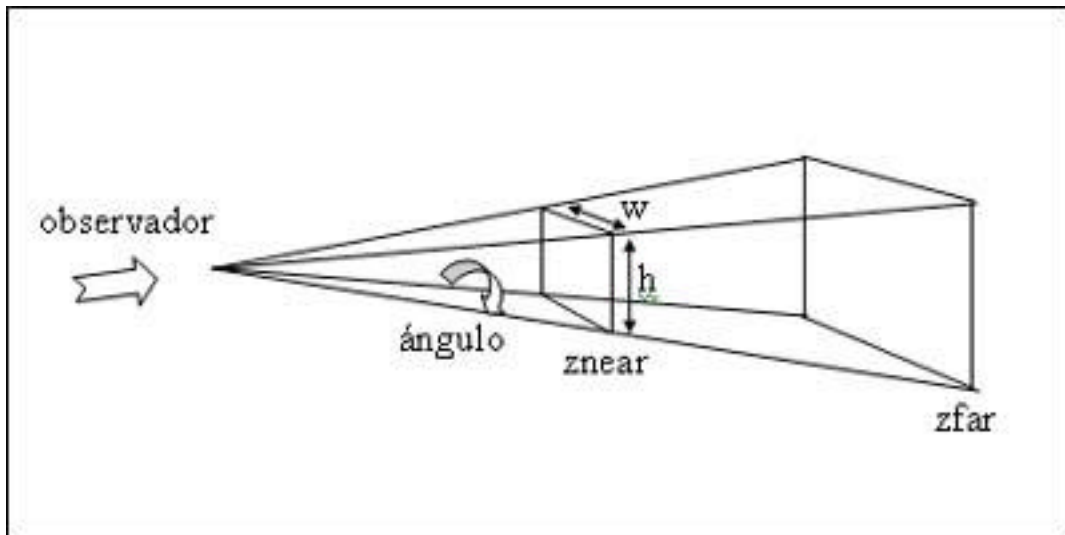


Ilustración 4.7

La ilustración 4.8 muestra la escena del coche de la sección anterior, esta vez con una proyección en perspectiva:

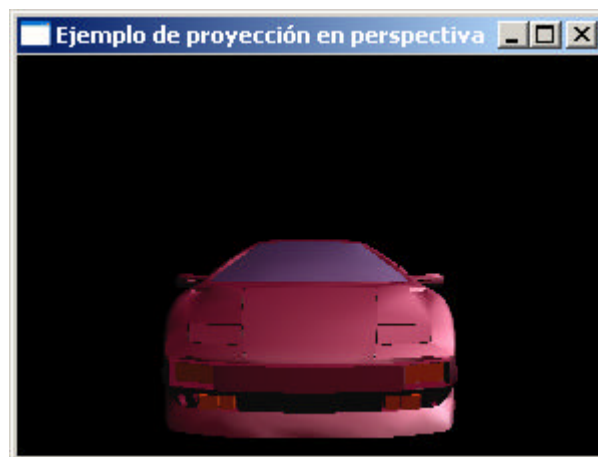


Ilustración 4.8

El código utilizado para definir la proyección ha sido

```
gluPerspective(45.0f, (GLfloat)(width/height), 0.01f, 100.0f);
```

Usamos 45° de ángulo, la relación entre el ancho y alto de la pantalla (width y height son el ancho y alto actual de la ventana), y las distancias a los planos de corte znear y zfar son 0.01 y 100 respectivamente.

4.4 Ejemplo: una escena simple

4.4.1 Código

El siguiente código es un programa que usa OpenGL mediante la librería GLUT. Primero se lista el código completo, y luego se comenta línea por línea. Con él aplicaremos lo aprendido en el anterior capítulo y el presente, además de nuevas funciones de la librería GLUT.

La escena consiste en un cubo de colores girando sobre si misma, y una esfera blanca de alambre (mas conocido como “wired”, consiste en no dibujar las caras del objeto, si no solamente las líneas que unen sus vértices, dando la sensación de ser una figura de alambre) girando alrededor del cubo. Podemos utilizar las teclas ‘o’ y ‘p’ para cambiar el tipo de proyección, ortográfica y perspectiva, respectivamente. Con la tecla ‘esc’ se abandona el programa.

```
#include <GL/glut.h>

GLfloat anguloCuboX = 0.0f;
GLfloat anguloCuboY = 0.0f;
GLfloat anguloEsfera = 0.0f;

GLint ancho, alto;

int hazPerspectiva = 0;

void reshape(int width, int height)
{
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
```

```

    if(hazPerspectiva)
        gluPerspective(60.0f, (GLfloat)width/(GLfloat)height, 1.0f,
20.0f);
    else
        glOrtho(-4, 4, -4, 4, 1, 10);

    glMatrixMode(GL_MODELVIEW);

    ancho = width;
    alto = height;
}

void drawCube(void)
{
    glColor3f(1.0f, 0.0f, 0.0f);
    glBegin(GL_QUADS); //cara frontal
    glVertex3f(-1.0f, -1.0f, 1.0f);
    glVertex3f( 1.0f, -1.0f, 1.0f);
    glVertex3f( 1.0f,  1.0f, 1.0f);
    glVertex3f(-1.0f,  1.0f, 1.0f);
    glEnd();

    glColor3f(0.0f, 1.0f, 0.0f);
    glBegin(GL_QUADS); //cara trasera
    glVertex3f( 1.0f, -1.0f, -1.0f);
    glVertex3f(-1.0f, -1.0f, -1.0f);
    glVertex3f(-1.0f,  1.0f, -1.0f);
    glVertex3f( 1.0f,  1.0f, -1.0f);
    glEnd();

    glColor3f(0.0f, 0.0f, 1.0f);
    glBegin(GL_QUADS); //cara lateral izq
    glVertex3f(-1.0f, -1.0f, -1.0f);
    glVertex3f(-1.0f, -1.0f,  1.0f);
    glVertex3f(-1.0f,  1.0f,  1.0f);
    glVertex3f(-1.0f,  1.0f, -1.0f);
    glEnd();
}

```

```

    glColor3f(1.0f, 1.0f, 0.0f);
    glBegin(GL_QUADS); //cara lateral dcha
    glVertex3f( 1.0f, -1.0f,  1.0f);
    glVertex3f( 1.0f, -1.0f, -1.0f);
    glVertex3f( 1.0f,  1.0f, -1.0f);
    glVertex3f( 1.0f,  1.0f,  1.0f);
    glEnd();

    glColor3f(0.0f, 1.0f, 1.0f);
    glBegin(GL_QUADS); //cara arriba
    glVertex3f(-1.0f,  1.0f,  1.0f);
    glVertex3f( 1.0f,  1.0f,  1.0f);
    glVertex3f( 1.0f,  1.0f, -1.0f);
    glVertex3f(-1.0f,  1.0f, -1.0f);
    glEnd();

    glColor3f(1.0f, 0.0f, 1.0f);
    glBegin(GL_QUADS); //cara abajo
    glVertex3f( 1.0f, -1.0f, -1.0f);
    glVertex3f( 1.0f, -1.0f,  1.0f);
    glVertex3f(-1.0f, -1.0f,  1.0f);
    glVertex3f(-1.0f, -1.0f, -1.0f);
    glEnd();
}

void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glLoadIdentity();

    glTranslatef(0.0f, 0.0f, -5.0f);

    glRotatef(anguloCuboX, 1.0f, 0.0f, 0.0f);
    glRotatef(anguloCuboY, 0.0f, 1.0f, 0.0f);

```

```

    drawCube();

    glLoadIdentity();

    glTranslatef(0.0f, 0.0f, -5.0f);
    glRotatef(anguloEsfera, 0.0f, 1.0f, 0.0f);
    glTranslatef(3.0f, 0.0f, 0.0f);

    glColor3f(1.0f, 1.0f, 1.0f);
    glutWireSphere(0.5f, 8, 8);

    glFlush();
    glutSwapBuffers();

    anguloCuboX+=0.1f;
    anguloCuboY+=0.1f;
    anguloEsfera+=0.2f;
}

void init()
{
    glClearColor(0,0,0,0);
    glEnable(GL_DEPTH_TEST);
    ancho = 400;
    alto = 400;
}

void idle()
{
    display();
}

void keyboard(unsigned char key, int x, int y)
{
    switch(key)
    {

```

```

    case 'p':
    case 'P':
        hazPerspectiva=1;
        reshape(ancho, alto);
        break;

    case 'o':
    case 'O':
        hazPerspectiva=0;
        reshape(ancho, alto);
        break;

    case 27:    // escape
        exit(0);
        break;

}
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowPosition(100, 100);
    glutInitWindowSize(ancho, alto);
    glutCreateWindow("Cubo 1");
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutIdleFunc(idle);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}

```

4.4.2 Análisis del código

Pasamos ahora a comentar el código. Se reproduce el código entero, pero iremos haciendo pausas en las funciones que no se hayan explicado en el capítulo 2.

Empezaremos por la función main():

```
int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
```

En esta ocasión, utilizamos GLUT_DOUBLE en vez de GLUT_SIMPLE. Esto hace posible la utilización de la técnica de “double buffer”, con la utilizamos dos buffers para ir sacando frames por pantalla, en vez de uno. Con esto conseguimos una mayor fluidez en escenas animadas.

```
    glutInitWindowPosition(100, 100);
    glutInitWindowSize(ancho, alto);
    glutCreateWindow("Cubo 1");
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutIdleFunc(idle);
```

Aquí añadimos una función callback nueva, el del idle. Esta función es llamada cuando la ventana esta en idle, es decir, no se está realizando ninguna acción sobre ella. Normalmente se usa la misma función que la de disuado, como en este ejemplo (la función idle() que hemos definido simplemente llama a la función redraw()).

```
    glutKeyboardFunc(keyboard);
```

Otro callback nuevo, usado para capturar y manejar el teclado cuando nuestra ventana está activa. La definición de esta función ha de ser de la forma

```
void teclado(unsigned char tecla, int x, int y)
```

donde “tecla” es el código ASCII de la tecla pulsada en cada momento y “x” e “y” las coordenadas del ratón en ese instante.

```
glutMainLoop();  
return 0;  
}
```

Ahora veremos la función `drawCube`, que utilizamos para crear un cubo. El cubo, como veremos ahora, está centrado en el origen y el lado es igual a 2.

```
void drawCube(void)  
{  
    glColor3f(1.0f, 0.0f, 0.0f);  
    glBegin(GL_QUADS); //cara frontal (C0)  
    glVertex3f(-1.0f, -1.0f, 1.0f);  
    glVertex3f( 1.0f, -1.0f, 1.0f);  
    glVertex3f( 1.0f,  1.0f, 1.0f);  
    glVertex3f(-1.0f,  1.0f, 1.0f);  
    glEnd();
```

Con éstas líneas creamos un polígono cuadrado, ensamblando cuatro vértices. Además hemos utilizado previamente `glColor` para asignarle el color rojo. Es importante observar que utilizamos el sentido antihorario, para que la normal del polígono vaya hacia fuera (recordemos que, por defecto, el sentido antihorario es el que define la normal hacia fuera). Este cuadrado conformará la cara del cubo frontal, la más cercana a nosotros.

```
    glColor3f(0.0f, 1.0f, 0.0f);  
    glBegin(GL_QUADS); //cara trasera (C1)  
    glVertex3f( 1.0f, -1.0f, -1.0f);  
    glVertex3f(-1.0f, -1.0f, -1.0f);  
    glVertex3f(-1.0f,  1.0f, -1.0f);  
    glVertex3f( 1.0f,  1.0f, -1.0f);  
    glEnd();
```

Definimos aquí la cara trasera, la más alejada de nosotros, de color verde.

```
    glColor3f(0.0f, 0.0f, 1.0f);
```

```
glBegin(GL_QUADS); //cara lateral izq (C2)
glVertex3f(-1.0f, -1.0f, -1.0f);
glVertex3f(-1.0f, -1.0f, 1.0f);
glVertex3f(-1.0f, 1.0f, 1.0f);
glVertex3f(-1.0f, 1.0f, -1.0f);
glEnd();
```

Cara lateral izquierda, de color azul.

```
glColor3f(1.0f, 1.0f, 0.0f);
glBegin(GL_QUADS); //cara lateral dcha (C3)
glVertex3f(1.0f, -1.0f, 1.0f);
glVertex3f(1.0f, -1.0f, -1.0f);
glVertex3f(1.0f, 1.0f, -1.0f);
glVertex3f(1.0f, 1.0f, 1.0f);
glEnd();
```

Cara lateral derecha, de color amarillo, por ser mezcla de rojo y verde.

```
glColor3f(0.0f, 1.0f, 1.0f);
glBegin(GL_QUADS); //cara arriba (C4)
glVertex3f(-1.0f, 1.0f, 1.0f);
glVertex3f(1.0f, 1.0f, 1.0f);
glVertex3f(1.0f, 1.0f, -1.0f);
glVertex3f(-1.0f, 1.0f, -1.0f);
glEnd();
```

Ésta será la cara de arriba, de color azul claro.

```
glColor3f(1.0f, 0.0f, 1.0f);
glBegin(GL_QUADS); //cara abajo (C5)
glVertex3f(1.0f, -1.0f, -1.0f);
glVertex3f(1.0f, -1.0f, 1.0f);
glVertex3f(-1.0f, -1.0f, 1.0f);
glVertex3f(-1.0f, -1.0f, -1.0f);
glEnd();
```

```
}
```

Y, finalmente, la cara de abajo, de color violeta.

El cubo que hemos definido, se puede ver de una forma más grafica en la ilustración 4.9. Sobre los colores hablaremos en el próximo capítulo, ya que, a primera vista puede resultar poco coherente su conformación.

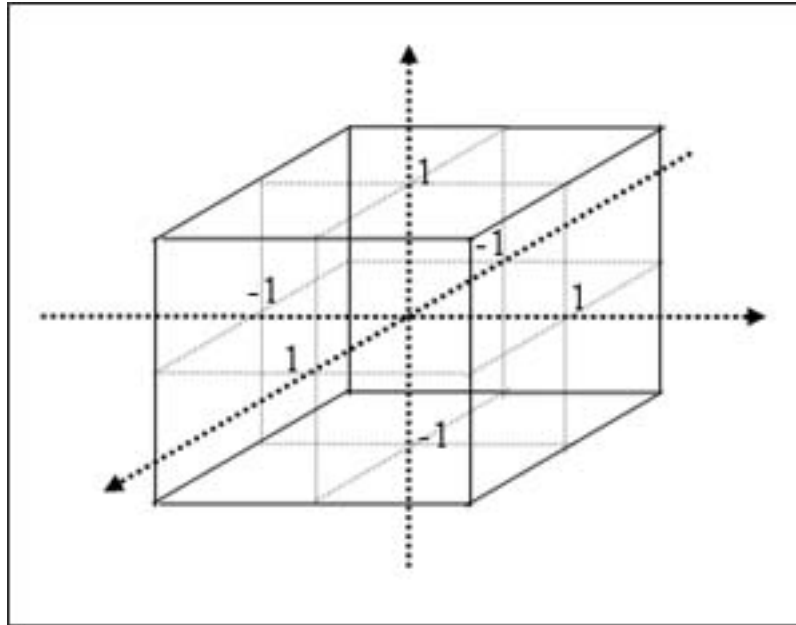


Ilustración 4.9

Vamos ahora con el contenido de los callbacks:

Primero vemos el `init()`, que como hemos dicho, no es un callback, simplemente activa los estados iniciales de opgl que queramos. En este caso, activamos un nuevo estado:

```
glEnable(GL_DEPTH_TEST);
```

para que haga el test de profundidad, utilizando el z-buffer. Además asignamos el ancho y alto de la ventana.

El callback `reshape`, llamado cuando la ventana se redimensiona:

```
void reshape(int width, int height)
{
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
```

Hacemos que la matriz activa sea la de proyección, puesto que es aquí donde definiremos el tipo de proyección a usar.

```
glLoadIdentity();
```

Cargamos en la matriz de proyección la identidad, para resetearla y poder trabajar sobre ella.

```
if(hazPerspectiva)
    gluPerspective(60.0f, (GLfloat)width/(GLfloat)height, 1.0f,
20.0f);
else
    glOrtho(-4, 4, -4, 4, 1, 10);
```

La variable “hazPerspectiva”, definida como un entero, hace las funciones de un booleano. Si su valor es cero, hace una proyección ortonormal. Para ello usamos la función glOrtho, definiendo los límites de los planos de trabajo. Si la variable esta a uno, hacemos una perspectiva con gluPerspective.

```
glMatrixMode(GL_MODELVIEW);
```

Aquí reactivamos la matriz del modelador, que es con la que trabajamos habitualmente.

```
    ancho = width;
    alto = height;
}
```

Por último, actualizamos las variables “ancho” y “alto”, con los valores actuales de la ventana.

Veamos ahora el callback del teclado:

```
void keyboard(unsigned char key, int x, int y)
```

```
{
    switch(key)
    {
        case 'p':
        case 'P':
            hazPerspectiva=1;
            reshape(ancho, alto);
            break;
    }
}
```

Si pulsamos la tecla p (en minúscula o mayúscula) activamos la variable “hazPerspectiva”. Para que se efectúe el cambio, llamamos a la función reshape, de una manera un tanto “artificial”, ya que el propósito de esta función es ajustar los parámetros de la proyección ante un cambio de la dimensión de la ventana. En este caso, la ventana no se ha redimensionado, y llamamos manualmente a la función con los valores actuales, para realizar el cambio de proyección.

```
case 'o':
case 'O':
    hazPerspectiva=0;
    reshape(ancho, alto);
    break;
```

Si la tecla pulsada es la ‘o’, usamos, igual que antes, la proyección ortográfica.

```
case 27: // escape
    exit(0);
    break;
}
}
```

En caso de que la tecla sea “ESC” (su código ASCII es el 27), salimos del programa.

El último callback a analizar es el que dibuja la escena:

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Aquí limpiamos el frame buffer, que es el buffer donde dibujamos, y el z-buffer, utilizado para el test de profundidad.

```
glLoadIdentity();
```

Reseteamos la matriz del modelador.

```
glTranslatef(0.0f, 0.0f, -5.0f);  
  
glRotatef(anguloCuboX, 1.0f, 0.0f, 0.0f);  
glRotatef(anguloCuboY, 0.0f, 1.0f, 0.0f);
```

Con estas tres líneas cargamos en la matriz del modelador una transformación de la siguiente forma: primero trasladamos lo que vayamos a dibujar (que será el cubo) cinco unidades hacia atrás, para ponerlo delante de nosotros. Luego lo giramos sobre el eje x y el eje y el número de grados que marquen las variables “anguloCuboX” y “anguloCuboY” respectivamente. Estas variables se van incrementando en cada frame, como veremos pocas líneas mas abajo.

```
drawCube();
```

Llamamos a la función que hemos creado para dibujar un cubo. Al hacerlo, se ejecutarán todas las funciones que crean las caras, centrando el cubo en el origen. Pero inmediatamente se ven modificadas las posiciones de los vértices por la matriz de transformación cargada en el modelador, dejando así el cubo donde nos interesa.

```
glLoadIdentity();
```

Reseteamos la matriz del modelador, ya que el cubo ya está situado en donde queremos, y necesitamos una nueva matriz de transformación para poner la esfera.

```
glTranslatef(0.0f, 0.0f, -5.0f);  
glRotatef(anguloEsfera, 0.0f, 1.0f, 0.0f);  
glTranslatef(3.0f, 0.0f, 0.0f);
```

Con estas nuevas tres líneas, creamos la matriz de transformación para la esfera. Primero la trasladamos 5 unidades hacia atrás, de forma que queda centrada en el mismo sitio que el cubo. Ahora rotamos el sistema de coordenadas tantos grados como

contenga la variable “anguloEsfera”, sobre el eje y. Ahora que tenemos rotado el sistema de coordenadas de la esfera, solo hay que desplazar la esfera en el eje x, de forma que según incrementemos “anguloEsfera”, ésta vaya describiendo una circunferencia, de radio el número de unidades que la desplazamos (en este caso 3).

```
glColor3f(1.0f, 1.0f, 1.0f);  
glutWireSphere(0.5f, 8, 8);
```

Activamos el color que deseemos para la esfera, blanco en este caso, y la dibujamos. Para ello, la librería GLUT proporciona una serie de funciones de alto nivel con objetos comunes, como cubos, cilindros, esferas, etc. Además GLUT permite dibujarlas como “Solid” o como “Wire”, es decir, como un objeto sólido, conformado por polígonos, o como un objeto “de alambre”. Los parámetros de glutWireSphere (y también los de glutSolidSphere) son el radio, el número de líneas de longitud y el número de líneas de latitud.

Para dibujar el cubo podíamos haber utilizado la función glutSolidCube, que lleva como parámetro la longitud del lado, pero se ha preferido crear una función que lo haga, con fines didácticos.

```
glFlush();
```

Hacemos que todos los comandos de ogl que estén en espera se ejecuten.

```
glutSwapBuffers();
```

Al utilizar la técnica de doble buffer, tenemos que llamar siempre a esta función al final del pintado de cada frame, para que vuelque de un buffer a otro el frame correspondiente.

```
anguloCuboX+=0.1f;  
anguloCuboY+=0.1f;  
anguloEsfera+=0.2f;  
}
```

Por último, incrementamos todos los ángulos que estamos usando.

En la ilustración 4.10 vemos un frame de la salida del programa.

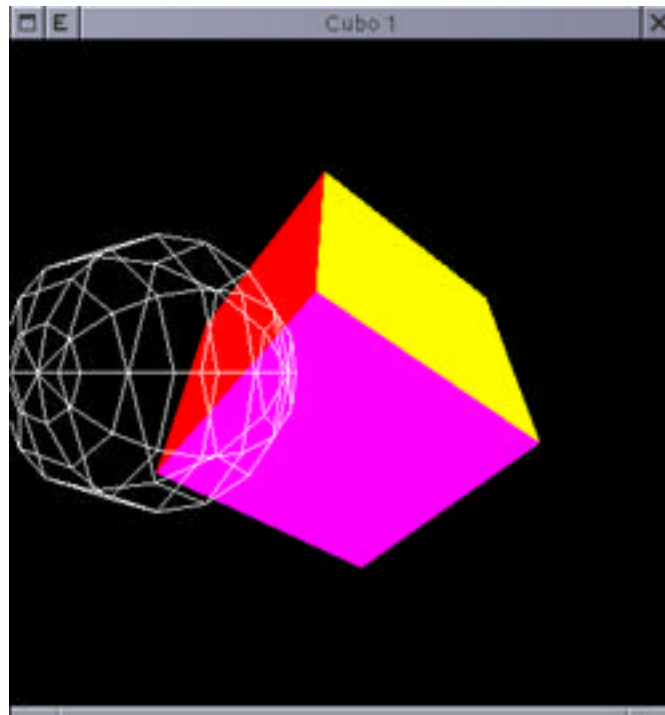


Ilustración 4.10