GRÁFICOS

APUNTES DE OPENGL

CURSO 1998-99

Rafael Molina Carmona
Juan Antonio Puchol García

ALGUNAS CUESTIONES PREVIAS

¿QUÉ ES OPENGL?

OpenGL es una **librería de modelado y gráficos 3D**. Fue desarrollada inicialmente por Silicon Graphics, aunque pronto pasó a convertirse en un estándar, en especial cuando lo adoptó Microsoft para su sistema operativo Windows. Sus principales características son:

- Es fácilmente portable y muy rápida.
- Es un sistema procedural y no descriptivo, es decir, el programador no describe una escena sino los objetos de la escena y los pasos necesarios para configurar la escena final.
- Actúa en modo inmediato, es decir, los objetos son dibujados conforme van creándose.
- Incluye:
    - Primitivas gráficas: puntos, líneas, polígonos.
    - Iluminación y sombreado.
    - Texturas.
    - Animaciones.
    - Otros efectos especiales.
- OpenGL trata con contextos de visualización o de *rendering*, asociados a un contexto de dispositivo que, a su vez, se encuentra asociado a un ventana.

¿QUÉ ES MESA?

Mesa es un **clónico gratuito de OpenGL**. Se trata de una librería cuyo API es básicamente igual que el de OpenGL. Incorpora toda la funcionalidad de OpenGL con la única excepción de algunas rutinas muy específicas, que se refieren a:

- NURBS recortadas.
- *Antialiasing* para polígonos.

Nosotros utilizaremos Mesa en este curso. En lo sucesivo nos referiremos indistintamente a Mesa o a OpenGL, ya que no existen diferencias sustanciales.

ESTRUCTURA BÁSICA DE OPENGL

- OpenGL es un API, no un lenguaje de programación. Necesitaremos, por lo tanto, un lenguaje (generalmente C o C++) en el que escribir el programa que realizará llamadas a funciones de la librería, utilizando para ello la sintaxis de C.
- OpenGL es independiente del sistema de ventanas utilizado y del sistema operativo, es decir, no incorpora rutinas para el manejo de ventanas. Este manejo debe realizarse a través del API del entorno de ventanas elegido.
- Se necesita, por lo tanto, un conjunto limitado de rutinas que pongan el contacto al sistema operativo y al entorno de ventanas con OpenGL. Este conjunto de rutinas es diferente para cada sistema operativo y no pertenece a OpenGL sino a una librería auxiliar. De esta manera, el núcleo de OpenGL permanece inalterado para todos los sistemas operativos.

- OpenGL incorpora la funcionalidad básica para realizar la visualización (*rendering*). Existe, no obstante, un conjunto de aspectos que no son estrictamente de visualización y que se encuentran en una librería estándar adicional llamada GLU (*OpenGL Utility Library*).
- Existen algunas herramientas o *toolkits* que facilitan la labor de "pegar" OpenGL con un entorno de ventanas. Un ejemplo es GLUT (*OpenGL Utility Toolkit*), un completo conjunto de herramientas que facilitan enormente el trabajo con OpenGL y MS-Windows o X Windows.
- En resumen, para trabajar con OpenGL necesitamos al menos las siguientes componentes:
    - Un lenguaje de programación con su compilador, que pueda realizar llamadas a funciones en formato C: C, C++, VisualBasic, Java...
    - Librería OpenGL: Contiene las funciones propias de OpenGL, dedicadas fundamentalmente a las tareas de *rendering*. Las funciones incorporan el prefijo `gl-`.
    - Librería de utilidades GLU: Contiene algunas funciones, con el prefijo `glu-`, que realizan las siguientes tareas:
        - Transformación de coordenadas.
        - Poligonalización de objetos.
        - Manipulación de imágenes para aplicarlas como texturas.
        - *Rendering* de figuras canónicas: esferas, cilindros y discos.
        - Curvas y superficies NURBS.
        - Informe de errores.
    - Librería para "pegar" OpenGL y el sistema de ventanas. Esta librería depende del sistema elegido. Entre los más habituales encontramos:
        - WGL: Es la librería para la familia de sistemas MS-Windows. El prefijo utilizado en sus funciones es `wgl-`.
        - GLX: Es la librería para X Windows. Utiliza el prefijo `glx-` en sus funciones.
    - Alternativamente al uso de las librerías anteriores (WGL, GLX...), es posible encontrar un amplio abanico de librerías que encapsulan el complicado trabajo con el API del entorno de ventanas y facilitan la creación y gestión de ventanas. Entre ellas destaca GLUT, en versiones tanto para MS Windows como para X Windows. Sus funciones llevan el prefijo `glut-`.

Nosotros vamos a trabajar con GLUT para X Windows, haciendo uso, además, de las librerías OpenGL y GLU, utilizando C como lenguaje de programación.

1. GLUT

INTRODUCCIÓN

Como paso previo al estudio de OpenGL, vamos a ver los fundamentos de la programación con GLUT. Puesto que el estudio de GLUT no es el objetivo de este curso, veremos solamente los conceptos fundamentales para poder trabajar con esta librería de herramientas.

GLUT es un API con formato ANSI C, pensado para escribir programas OpenGL independientemente del sistema de ventanas utilizado. Existen versiones de GLUT para los principales sistemas de ventanas, incluyendo MS Windows y X Windows. La interfaz de las funciones de GLUT siempre es la misma, lo que facilita al máximo la portabilidad de programas entre diferentes entornos.

GLUT es una librería pequeña y fácil de aprender y utilizar. No cubre, no obstante, todas las facetas de la programación con APIs de ventanas, por lo que para la programación de aplicaciones complejas se hace imprescindible la programación directa de los sistemas de ventanas, utilizando las librerías para "pegarlos" con OpenGL (WGL, GLX, ...). Para el propósito de este curso, el uso de GLUT es suficiente.

El API de GLUT es una máquina de estados. Esto quiere decir que el sistema tiene un conjunto de variables de estado que durante la ejecución del programa fijan las características que debe tener la aplicación en cada momento. Estas variables tienen unos valores iniciales que han sido escogidos para adaptarse a la mayoría de las aplicaciones, aunque pueden ser modificados por el programador para conseguir el efecto deseado. Una misma función de GLUT puede tener efectos diferentes dependiendo de los valores de las variables de estado. La propia librería OpenGL se comporta, así mismo, como una máquina de estados, tal y como veremos más adelante.

Las funciones de GLUT son simples, con pocos parámetros y evitan el uso de punteros. Dependiendo de la funcionalidad podemos distinguir funciones para:

- Inicialización
- Control de ventanas
- Control de menús
- Procesamiento de eventos
- Registro de funciones *callback*
- Obtención del estado
- Control de *overlays*
- Control del mapa de colores
- Visualización de fuentes de texto
- Dibujo de formas geométricas

Los últimos cuatro grupos de funciones no los veremos, por quedar fuera del alcance de este curso. Puede consultarse una completa referencia de estos temas en [Kilg96].

A continuación se comentan algunas de las particularidades y convenciones de GLUT:

- En una ventana, el origen de coordenadas (0,0) se encuentra en la esquina superior izquierda. Esto es inconsistente con OpenGL, cuyo origen está en la esquina inferior izquierda, pero no supone ningún problema como veremos más adelante.
- Los identificadores enteros en GLUT (utilizados para ventanas, menús, etc.) empiezan en 1 y no en 0.
- Las definiciones de las funciones de GLUT se encuentran en el archivo de cabeceras `glut.h`.

INICIALIZACIÓN

Las rutinas que empiezan con el prefijo `glutInit-` se utilizan para **inicializar el estado** de GLUT. Algunas de estas rutinas son:

- `void glutInit (int *argcp, char **argv)`
  `argcp`: puntero al parámetro `argc` de la función `main` del programa.
  `argv`: parámetro `argv` de la función `main`.

  `glutInit` inicializa la librería GLUT, pudiéndosele pasar algún parámetro por línea de comandos a través de `argcp` y `argv`.

  Esta rutina debe ser llamada una única vez al principio del programa. Ninguna otra función que no sea de inicialización puede llamarse antes. Si utilizamos alguna función de inicialización (con el prefijo `glutInit-`) delante de ésta función, estamos fijando el estado por defecto de la aplicación al inicializarse.

- `void glutInitWindowSize (int ancho, int alto)`
  `ancho`: anchura de la ventana en pixels
  `alto`: altura de la ventana en pixels

  Esta rutina sirve para indicar el tamaño inicial de las ventanas que se creen.

- `void glutInitWindowPosition (int x, int y)`
  `x`: posición en x de la ventana en pixels
  `y`: posición en y de la ventana en pixels

  Con esta función fijamos la posición inicial de las ventanas que se creen.

- `void glutInitDisplayMode (unsigned int modo)`
  `modo`: modo de *display*. Es una composición mediante conectores "`|`" de algunos de los valores siguientes:

`GLUT_RGBA`:	Selecciona una ventana en modo RGBA. Es el valor por defecto si no se indican ni `GLUT_RGBA` ni `GLUT_INDEX`.
`GLUT_RGB`:	Lo mismo que `GLUT_RGBA`.
`GLUT_INDEX`:	Seleciona una ventana en modo de índice de colores. Se impone sobre `GLUT_RGBA`.
`GLUT_SINGLE`:	Selecciona una ventana en modo *buffer* simple. Es el valor por defecto.
`GLUT_DOUBLE`:	Selecciona una ventana en modo *buffer* doble. Se impone sobre `GLUT_SINGLE`.

GLUT_ACCUM:	Selecciona una ventana con un *buffer* acumulativo.
GLUT_ALPHA:	Selecciona una ventana con una componente alfa del *buffer* de color.
GLUT_DEPTH:	Selecciona una ventana con un *buffer* de profundidad.
GLUT_STENCIL:	Selecciona una ventana con un *buffer* de estarcido.
GLUT_MULTISAMPLE:	Selecciona una ventana con soporte multimuestra.
GLUT_STEREO :	Selecciona una ventana estéreo.
GLUT_LUMINANCE:	Selecciona una ventana con un modelo de color en tonos de gris.

Esta función fija el modo de *display* inicial con que se crearán las ventanas. Los dos valores que suelen darse en la mayoría de las aplicaciones son GLUT_RGBA, para fijar un modelo de color RGB con componente alfa, y GLUT_DOUBLE para seleccionar una ventana con doble buffer. Cuando tratemos OpenGL, veremos estos conceptos con detenimiento.

A modo de ejemplo mostramos a continuación el esqueleto básico de un programa en GLUT con las inicializaciones pertinentes.

EJEMPLO 1

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

void main (int argc, char **argv)
{
      // Fijar tamaño y posición inicial de las ventanas

      glutInitWindowSize (640, 480);
      glutInitWindowPosition (0, 0);

      // Seleccionar modo de display: RGBA y doble buffer

      glutInitDisplayMode (GLUT_RGBA | GLUT_DOUBLE);

      // Inicializar la librería GLUT

      glutInit (&argc, argv);

      // Código del programa
      // ...
}
```

CONTROL DE VENTANAS

Una vez inicializada la librería, llega el momento de crear ventanas en nuestra aplicación. Cuando se crea una ventana, la librería devuelve un entero que es el identificador de ventana.

GLUT hace uso del concepto de **ventana activa**, es decir, en cada momento de la ejecución del programa sólo una ventana de las creadas es la activa. La mayoría de las funciones que afectan a las ventanas no tienen ningún parámetro que especifique la ventana a la que afectan, de manera que es la ventana activa la que recibe esa acción. Sólo unas pocas funciones (crear subventana, activar ventana y destruir ventana) hacen uso del identificador de ventana para conocer sobre qué ventana deben actuar. El resto de funciones (forzar el redibujado, intercambiar los *buffers*, fijar la posición y el tamaño,

etc.) actúan sobre la ventana activa. La ventana activa es la última que haya sido creada o, en su caso, la última que haya sido activada.

GLUT proporciona dos tipos de ventanas: **ventanas principales** y **subventanas**. En ambos casos se devuelve un identificador de la ventana, que es único para toda la aplicación.

Algunas de las principales funciones para el control de ventanas se detallan a continuación.

- `int glutCreateWindow (char *nombre)`
  `nombre`: cadena de caracteres con el nombre de la ventana.
  Devuelve un entero que es el identificador único de la ventana.

  Esta función crea una ventana principal. Cuando se crea una ventana, la ventana activa pasa a ser esta nueva ventana. Toda ventana lleva implícito un contexto de visualización de OpenGL. Este contexto es el que permite que la ventana creada con GLUT (o en su caso, con cualquier entorno de ventanas) se ponga en contacto con OpenGL. El concepto de contexto de visualización será ampliamente estudiado cuando veamos OpenGL.

  Las ventanas creadas con esta función no son visualizadas todavía. La visualización la llevará a cabo una función de respuesta al evento de redibujado de la ventana. También trataremos este concepto más adelante. El tamaño de la ventana, su posición y su modo de *display*, viene dados por las especificaciones iniciales hechas mediante las funciones `glutInit-` vistas en el apartado anterior

- `int glutCreateSubWindow (int idVentanaPadre, int x, int y, int ancho, int alto)`
  `idVentanaPadre`: identificador de la ventana padre de la subventana
  `x, y`: posición (x,y) en pixels de la subventana, relativa a la ventana padre
  `ancho, alto`: anchura y altura de la subventana en pixels
  Devuelve un identificador único para la subventana

  Esta rutina crea una subventana, hija de la ventana identificada por `idVentanaPadre`. La ventana activa de la aplicación pasa a ser la nueva subventana.

  Como en el caso de las ventanas principales, las subventanas llevan asociado un contexto de visualización, y su modo de *display* viene dado por las especificaciones iniciales (no así su tamaño y su posición, que en este caso son dados explícitamente).

- `void glutSetWindow (int idVentana)`
  `idVentana`: identificador de la ventana

  Fija la ventana identificada por `idVentana` como ventana activa.

- `int glutGetWindow (void)`
  Devuelve un identificador de ventana

Devuelve el identificador de la ventana activa

- `void glutDestroyWindow (int idVentana)`
  `idVentana`: identificador de la ventana

  Destruye la ventana identificada por `idVentana`.

- `void glutSwapBuffers (void)`

  Intercambia los *buffers* de la ventana actual. Se utiliza en el modo de doble buffer. Esta importante característica de OpenGL (y de GLUT por extensión) hace que se reduzcan al mínimo los parpadeos durante el redibujado, especialmente si hay animaciones. La ventana tiene dos *buffers*: uno de ellos visible y el otro invisible. El dibujo se realiza en el buffer oculto y, al intercambiarlos, se dibuja todo de una vez, eliminando así el parpadeo. Este tema lo trataremos con detalle cuando veamos OpenGL.

- `void glutPositionWindow (int x, int y)`
  `x, y`: posición (x,y) en pixels de la ventana

  Solicita el cambio de la posición de la ventana actual en la pantalla. Si la ventana actual es principal, (x,y) se toma con referencia al origen de la pantalla. Si es una subventana, la distancia se toma con referencia al origen de la ventana padre.

- `void glutReshapeWindow (int ancho, int alto)`
  `ancho`: anchura de la ventana en pixels
  `alto`: altura de la ventana el pixels

  Solicita el cambio del tamaño de la ventana actual, dándole al ancho y el alto especificados.

Otras funciones referentes a las ventanas se muestran a continuación junto con una breve descripción.

- `void glutPostRedisplay (void)`: Solicita el redibujado de la ventana actual.
- `void glutFullScreen (void)`: Solicita que la ventana actual sea maximizada. Sólo funciona para ventanas principales.
- `void glutPopWindow (void)`: Adelanta la ventana actual una posición en la pila de ventanas.
- `void glutPushWindow (void)`: Retrasa la ventana actual una posición en la pila de ventanas.
- `void glutShowWindow (void)`: Muestra la ventana actual.
- `void glutHideWindow (void)`: Oculta la ventana actual.
- `void glutIconifyWindow (void)`: Solicita la minimización de la ventana actual y su conversión en icono.
- `void glutSetWindowTitle (char *nombre)`: Cambia el nombre de la ventana actual.
- `void glutSetIconTitle (char *nombre)`: Da nombre a la ventana actual cuando está minimizada.

- `void glutSetCursor (int cursor)`: Cambia el puntero del ratón cuando sobrevuela la ventana actual. El parámetro es un valor de una enumeración. Esa enumeración puede consultarse en el manual de programación de GLUT [Kilg96].

Obsérvese que en la mayoría de los casos no se realizan acciones, sino que se solicita realizarlas. Realmente, estas funciones desencadenan eventos que son atendidos cuando les llega su turno en la cola de eventos y no inmediatamente.

A continuación presentamos una ampliación del ejemplo 1, introduciendo la creación y el manejo básico de las ventanas.

EJEMPLO 2

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

void main (int argc, char **argv)
{
      // Identificadores para ventana principal y su subventana
      int idPrincipal, idSubVent;

      // Fijar tamaño y posición inicial de las ventanas
      glutInitWindowSize (640, 480);
      glutInitWindowPosition (0, 0);

      // Seleccionar modo de display: RGBA y doble buffer
      glutInitDisplayMode (GLUT_RGBA | GLUT_DOUBLE);

      // Inicializar la librería GLUT
      glutInit (&argc, argv);

      // Creación de las ventanas
      idPrincipal = glutCreateWindow ("Ventana principal");
      idSubVent = glutCreateSubWindow (idPrincipal,5,5,100,100);

      // Poner la ventana principal como actual y maximizarla
      glutSetWindow (idPrincipal);
      glutFullScreen ();

      // Poner subventana como actual y asociarle cursor con
      // forma de flecha apuntando hacia arriba y derecha
      glutSetWindow (idSubVent);
      glutSetCursor (GLUT_CURSOR_RIGHT_ARROW);

      // Código del programa
      // ...

      // Destruir la subventana
      glutDestroyWindow (idSubVent);
}
```

CONTROL DE MENÚS

GLUT ofrece un manejo muy básico de los menús. Sólo incorpora **menús flotantes desplegables**, que aparecen al pulsar uno de los tres botones del ratón en la posición donde se encuentra el cursor. No es posible crear, destruir, añadir, eliminar o cambiar ítems de un menú mientras está en uso, es decir, desplegado.

Los menús GLUT vienen identificados por un valor entero que asigna el sistema cuando se crea el menú, de una manera similar a lo que sucede con las ventanas. Para responder a las acciones del menú es necesario escribir una función de respuesta (*callback*) que las realice. A esta función se le pasa como parámetro un entero que identifica el ítem del menú que ha sido seleccionado, y que la función debe utilizar para decidir la acción a llevar a cabo.

En el caso de los menús, también se aplica el concepto de **menú activo**, de manera que las funciones sucesivas a las que se llame, actúan sobre el menú activo, tal y como ocurre con las ventanas.

- `int glutCreateMenu (void (*funcion) (int valor))`
  `funcion`: Función *callback* a la que se llama cuando se selecciona un ítem del menú. El valor que se le pasa a la función identifica al ítem del menú.
  Devuelve un identificador único para el menú.

  Esta función crea un menú flotante asociado a la ventana actual y devuelve un identificador único para el mismo. Este menú pasa a ser el menú activo.

- `void glutSetMenu (int idMenu)`
  `idMenu`: identificador del menú.

  Esta función pone el menú identificado por `idMenu` como menú activo.

- `int glutGetMenu (void)`
  Devuelve un identificador de menú.

  Devuelve el identificador del menú activo.

- `void glutDestroyMenu (int idMenu)`
  `idMenu`: identificador del menú.

  Destruye el menú identificado por `idMenu`.

- `void glutAddMenuEntry (char *nombre, int valor)`
  `nombre`: cadena de caracteres que aparece en la opción del menú
  `valor`: identificador para el ítem del menú. Este valor es el que se pasa a la función de respuesta al menú.

  Añade una entrada al final del menú activo, y la cadena que se visualiza es el nombre. Cuando el usuario selecciona esa opción, se llama a la función de respuesta con el valor como parámetro.

- `void glutAddSubMenu (char *nombre, int idMenu)`
  `nombre`: cadena de caracteres que aparece en la opción del menú principal.
  `idMenu`: identificador del submenú asociado al ítem.

Crea un ítem del menú y le asocia un submenú que se despliega al ser seleccionada la entrada del menú principal. El submenú debe haber sido creado previamente como cualquier otro menú.

- `void glutAttachMenu (int boton)`
  `boton`: botón del ratón al que asociamos el menú.

Asocia el menú activo a un botón del ratón en la ventana activa, de manera que el menú se despliega cuando se pulsa el botón del ratón. El botón puede valer `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON` o `GLUT_RIGHT_BUTTON`, para asociarlo al botón izquierdo, central o derecho respectivamente.

Otras funciones relacionadas con el manejo de los menús, son las siguientes:

- `void glutChangeToMenuEntry (int entrada, char *nombre, int valor)`: Cambia la entrada (el parámetro entrada indica el índice de la misma, empezando en 1) por otra diferente.
- `void glutChangeToSubMenu (int entrada, char *nombre, int valor)`: Cambia un submenú por otro.
- `void glutRemoveMenuItem (int entrada)`: Elimina una entrada del menú.
- `void glutDetachMenu (int boton)`: Elimina la asociación entre el menú activo y el botón del ratón.

EJEMPLO 3

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

void main (int argc, char **argv)
{
      // Identificador de la ventana
      int idVentana;

      // Fijar tamaño y posición inicial de las ventanas
      glutInitWindowSize (640, 480);
      glutInitWindowPosition (0, 0);

      // Seleccionar modo de display: RGBA y doble buffer
      glutInitDisplayMode (GLUT_RGBA | GLUT_DOUBLE);

      // Inicializar la librería GLUT
      glutInit (&argc, argv);

      // Creación de la ventana
      idVentana = glutCreateWindow ("Ejemplo 3");

      // Identificadores de los menús
      int menuColor, menuPrincipal;

      // Crear submenú y asociarle su función de respuesta
      menuColor = glutCreateMenu (RespuestaMenuColor);
      glutAddMenuEntry ("Rojo", 1);
      glutAddMenuEntry ("Verde", 2);
      glutAddMenuEntry ("Azul", 3);

      // Crear menú principal y asociarle su función de respuesta
```

```
        menuPrincipal = glutCreateMenu (RespuestaMenuPrincipal);
        glutAddMenuEntry (“Luz”, 1);
        glutAddMenuEntry (“Material”, 2);
        glutAddSubMenu (“Color”, menuColor);

        // Asociar el menú al botón derecho del ratón
        glutAttachMenu (GLUT_RIGHT_BUTTON);

        // Código del programa
        // ...
}
```

PROCESAMIENTO DE EVENTOS

GLUT incorpora su propio **bucle de proceso de eventos**. Lo único que es necesario hacer es llamar a ese bucle con la función `glutMainLoop`. Esta función implementa un bucle infinito que se encarga de consultar la cola de eventos y de responder a cada uno de ellos ejecutando una función. La función que se ejecuta cuando se produce un evento debe ser escrita por el programador y registrada como una función de respuesta (*callback*). De esta manera, GLUT sabe a qué función llamar para responder a un determinado evento.

- `void glutMainLoop (void)`

  Esta rutina debe llamarse una vez en el programa. Puesto que implementa un bucle infinito (realmente el bucle termina cuando se produce un evento de cerrar aplicación), la función no termina nunca. Por ello es necesario registrar previamente las funciones *callback* y que, además, la llamada a esta función sea la última de `main`.

REGISTRO DE FUNCIONES DE RESPUESTA

El **registro de las funciones de respuesta** (*callback*) a eventos debe realizarse antes de llamar a la función `glutMainLoop`. Para realizar el registro se utilizan un conjunto de funciones con sintaxis común: `void glut[Evento]Func (tipo (*funcion) (parámetros))`, que indican que para responder al evento `Evento` debe utilizarse la función `funcion`.

Existen tres tipos de eventos: de ventana, de menús y globales. Los primeros están asociados a las ventanas y tratan cuestiones como el redibujado, el redimensionamiento, cambios en la visibilidad y el foco de la ventana. Los eventos de menú responden a la selección de las opciones de un menú y ya han sido tratados en el apartado anterior. Por último, los eventos globales no están asociados a ninguna ventana ni menú y se refieren, fundamentalmente, al control de temporizadores y acciones *idle*.

El orden en que se registran las funciones de respuesta a los eventos es indefinido. Sin embargo, al registrar las funciones de respuesta asociadas a eventos de ventana o de menú, se ha de tener en cuenta que éstas se asocian a la ventana activa o al menú activo en cada caso. Por otro lado, los eventos no se propagan a las ventanas padre.

FUNCIONES DE RESPUESTA A EVENTOS DE VENTANA

- `void glutDisplayFunc (void (*funcion) (void))`
  `funcion`: función de respuesta al evento

  Registra la función que responde al evento de redibujado de la ventana activa. GLUT no proporciona una función de respuesta por defecto para el redibujado, por lo que es obligatorio escribir una función de este tipo para cada ventana creada. Si se crea una ventana y no se registra la función de respuesta al redibujado, se produce un error.

- `void glutReshapeFunc (void (*funcion) (int ancho, int alto))`
  `funcion`: función de respuesta al evento

  Registra la función de respuesta al evento de redimensionamiento de la ventana activa. La función de respuesta debe tener como parámetros la anchura y la altura de la ventana tras el redimensionamiento. GLUT dispone, en este caso, de una función de respuesta por defecto, que se utiliza cuando no se registra ninguna función para este evento. El redimensionamiento de la ventana principal no genera ningún evento de redimensionamiento de sus subventanas, por lo que es necesario realizarlo explícitamente.

- `void glutKeyboardFunc (void (*funcion) (unsigned char tecla, int x, int y))`
  `funcion`: función de respuesta al evento.

  Se utiliza para registrar la función que responde a eventos del teclado sobre la ventana activa. La función de respuesta debe tener como parámetros la tecla que se ha pulsado (su carácter ASCII) y la posición (x,y) del puntero del ratón en ese momento, relativa a la ventana. No es posible detectar las teclas de modificadores directamente (CTRL, ALT, ...). Debemos utilizar para ello la función `glutGetModifiers`. Si no se registra ninguna función, los eventos de teclado son ignorados.

- `void glutMouseFunc (void (*funcion) (int boton, int estado, int x, int y))`
  `funcion`: función de respuesta al evento.

  Registra para la ventana activa la función de respuesta a eventos del ratón. Los eventos de ratón se producen tanto cuando se pulsa como cuando se suelta un botón del ratón. Los parámetros de la función de respuesta deben ser el botón (`GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON` o `GLUT_RIGHT_BUTTON`), el estado del botón (`GLUT_UP` o `GLUT_DOWN`) y la posición (`x,y`) del puntero relativa a la ventana. Cuando existe un menú asociado a un botón del ratón, el evento de pulsar ese botón es ignorado, prevaleciendo el menú sobre otras funcionalidades. Como en el caso del teclado, es posible detectar el uso de modificadores con la función `glutGetModifiers`. Si no registra una función de respuesta a eventos del ratón, éstos son ignorados.

- `void glutMotionFunc (void (*funcion) (int x, int y))`
  `funcion`: función de respuesta al evento.

Registra para la ventana activa la función de respuesta a movimientos del ratón cuando se mantiene pulsado algún botón del mismo. Los parámetros (x,y) indican la posición del puntero en coordenadas relativas a la ventana.

- `void glutPassiveMotionFunc (void (*funcion) (int x, int y))`
  `funcion`: función de respuesta al evento.

Registra para la ventana activa la función de respuesta a movimientos del ratón cuando no se mantiene pulsado ningún botón del mismo. Los parámetros (x,y) indican la posición del puntero en coordenadas relativas a la ventana.

Otras funciones de registro asociadas a ventanas son las siguientes:

- `void glutVisibilityFunc (void (*funcion) (int estado))`: Registra para la ventana activa la función de respuesta al evento de cambios en la visibilidad de la ventana. El parámetro `estado` puede ser `GLUT_NOT_VISIBLE` o `GLUT_VISIBLE`.

- `void glutEntryFunc (void (*funcion) (int estado))`: Registra para la ventana activa la función de respuesta al evento de entrada y salida del ratón en el área de la ventana. El parámetro `estado` puede ser `GLUT_LEFT` o `GLUT_ENTERED`.

- `void glutSpecialFunc (void (*funcion) (int tecla, int x, int y))`: Registra para la ventana activa el evento de pulsar una tecla especial. El parámetro `tecla` puede ser `GLUT_KEY_Fn` para teclas de función (n=1, 2, ..., 12), `GLUT_KEY_RIGHT`, `GLUT_KEY_LEFT`, `GLUT_KEY_UP`, `GLUT_KEY_DOWN`, `GLUT_KEY_PAGE_UP`, `GLUT_KEY_PAGE_DOWN`, `GLUT_KEY_HOME`, `GLUT_KEY_END` o `GLUT_KEY_INSERT`.

FUNCIONES DE RESPUESTA A EVENTOS GLOBALES

- `void glutMenuStatusFunc (void (*funcion) (int estado, int x, int y))`
  `funcion`: función de respuesta al evento

Registra la función que responde al evento que se produce cuando se despliega o se pliega un menú. El parámetro `estado` puede valer `GLUT_MENU_IN_USE` cuando se despliega un menú, o `GLUT_MENU_NOT_IN_USE` cuando se pliega. Los valores (x,y) son la posición del puntero del ratón en cada caso.

- `void glutIdleFunc (void (*funcion) (void))`
  `funcion`: función de respuesta al evento

Registra la función de respuesta al evento *idle*. Este evento se produce cada vez que el sistema no tiene ningún otro evento que atender. En OpenGL se suele utilizar para hacer animaciones. La función que da respuesta al evento debe ser lo más pequeña posible para evitar mermas en la capacidad de interacción de la aplicación.

- `void glutTimerFunc (unsigned int miliseg, void (*funcion) (int valor), int valor)`
  `miliseg`: Número de milisegundos del temporizador

`funcion`: Función de respuesta al evento
`valor`: Valor que debe utilizarse para llamar a la función de respuesta

Registra tanto un temporizador como la función de respuesta al mismo. Debemos indicar el tiempo en milisegundos, un valor de identificación del temporizador y la función que responde al mismo, cuyo único parámetro debe ser el identificador del temporizador.

OBTENCIÓN DEL ESTADO

GLUT incorpora un conjunto de funciones que devuelven las diferentes variables de estado.

- `int glutGet (GLenum estado)`
  `estado`: variable concreta de estado para la que deseamos obtener información.
  Devuelve un entero indicando el valor de la variable referida.

  Esta función devuelve el valor de una variable de estado. El parámetro `estado` hace referencia a la variable de la que deseamos conocer su valor. Este parámetro es un valor de una enumeración, entre cuyos valores encontramos `GLUT_WINDOW_X`, `GLUT_WINDOW_Y`, para obtener la posición de la ventana activa, `GLUT_WINDOW_WIDTH`, `GLUT_WINDOW_HIGHT`, para obtener el tamaño de la ventana activa, `GLUT_WINDOW_PARENT`, para obtener el identificador de la ventana padre de la activa, etc. La lista completa de valores podemos encontrarla en [Kilg96].

- `int glutDeviceGet (GLenum info)`
  `info`: dispositivo sobre el que obtener información
  Devuelve información sobre el dispositivo indicado

  Obtener información sobre los dispositivos del sistema. El parámetro `info` indica el dispositivo: `GLUT_HAS_KEYBOARD`, `GLUT_HAS_MOUSE`, `GLUT_NUM_MOUSE_BUTTONS`, etc. En los dos primeros casos, la función devuelve 0 si no existe el dispositivo y otro valor si existe. En el tercer caso, devuelve el número de botones del ratón. Existen valores de la enumeración para otros dispositivos. La referencia puede encontrarse completa en [Kilg96].

- `int glutGetModifiers (void)`
  Devuelve un valor que indica el modificador que se ha pulsado.

  Devuelve la tecla modificador que ha sido pulsada: `GLUT_ACTIVE_SHIFT`, `GLUT_ACTIVE_CTRL` o `GLUT_ACTIVE_ALT`.

La librería GLUT permite realizar otras muchas acciones, referentes al manejo del color, a la visualización de texto, al control de las capas y al diseño de algunas primitivas geométricas. Puesto que OpenGL realiza también el manejo de estos elementos y dado que vamos a utilizar GLUT como herramienta de apoyo pero no como objetivo del curso en sí mismo, obviaremos esta parte. No obstante, una buena referencia puede encontrarse en [Kilg96].

2. OPENGL

SINTAXIS DE LAS FUNCIONES

Todos los nombres de funciones de OpenGL siguen una convención en cuanto a sintaxis, que es:

<Prefijo> <Raiz> <Nº argumentos (opcional)> <Tipos argumentos (opcional)>

- El prefijo indica la librería de la que procede. Puede ser `glut-` (para las funciones de la utilidad GLUT, que ya hemos visto), `gl-` (para las funciones de OpenGL), `aux-` (para las funciones de la librería auxiliar, que no veremos) o `glu-` (para las funciones de la librería de utilidades, que tampoco veremos).
- La raíz indica qué realiza la función.
- El número de argumentos y los tipos de estos, como vemos son opcionales. Los posibles tipos los veremos en el próximo apartado.

Por ejemplo, la función `glColor3f` es una función de OpenGL, referente al color, con tres argumentos de tipo `float`.

TIPOS DE DATOS DE OPENGL

OpenGL incorpora tipos de datos propios para facilitar la portabilidad. Los tipos de datos, junto con su correspondencia en C y el sufijo que se suele utilizar habitualmente para nombrar las variables son:

Tipo de OpenGL	Espacio en memoria	Tipo de C	Sufijo
`Glbyte`	entero de 8 bits	signed char	b
`Glshort`	entero de 16 bits	short	s
`GLint, Glsizei`	entero de 32 bits	long	l
`Glfloat, Glclampf`	flotante de 32 bits	float	f
`Gldouble, Glclampd`	flotante de 64 bits	double	d
`Glubyte, Glboolean`	entero sin signo de 8 bits	unsigned char	ub
`Glushort`	entero sin signo de 16 bits	unsigned short	us
`Gluint, Glenum, Glbitfield`	entero sin signo de 32 bits	unsigned long	ui

EL CONTEXTO DE VISUALIZACIÓN (RENDER CONTEXT)

En los sistemas operativos de interfaz gráfica, cada ventana lleva asociado un **contexto de dispositivo** o *device context* que almacena todas las inicializaciones referentes a la forma de dibujar en la ventana. De igual manera, cuando trabajamos con OpenGL, cada ventana en la que se produce la visualización de objetos tiene asociado un **contexto de visualización** o *render context*. El contexto de visualización lleva asociadas todas las inicializaciones referentes al entorno, desde modos de dibujo hasta comandos. Funciona de una manera similar a como lo hace el contexto de dispositivo.

Cuando trabajamos con GLUT, la creación y manejo del contexto de visualización es transparente. Cuando creamos una ventana, la librería se encarga de asociarle un contexto de visualización adecuado. Cuando se cierra, es la librería la que destruye ese contexto. Por lo tanto, no debemos preocuparnos del control de este elemento, aunque es recomendable conocer su existencia, especialmente si desechamos GLUT en favor de otro entorno de ventanas que lo requiera.

En apartados posteriores veremos algunas funciones que nos permiten alterar el estado del contexto de visualización (`glEnable` y `glDisable`), aunque no hagamos referencia a él directamente. Estas funciones permitirán alterar las opciones de visualización, por ejemplo, ocultación de caras, incorporación de un Z-*buffer*, etc.

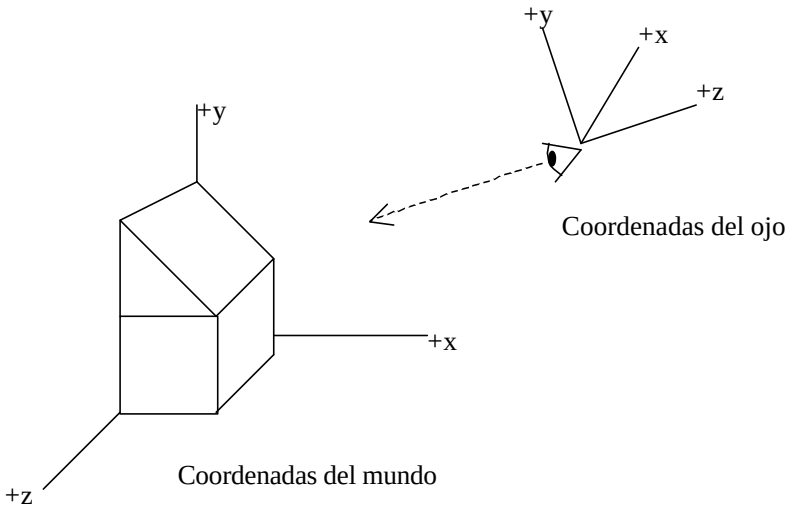
SISTEMAS DE COORDENADAS

Una vez creado el contexto de visualización (cuestión ésta que se reduce a crear la ventana cuando trabajamos con GLUT), el siguiente paso es crear los sistemas de coordenadas adecuados para que la visualización de los objetos se realice tal y como deseamos. En los siguientes apartados repasamos los diferentes conceptos y presentamos como definirlos en OpenGL.

TRANSFORMACIÓN DE PUNTOS 3D A PIXELS

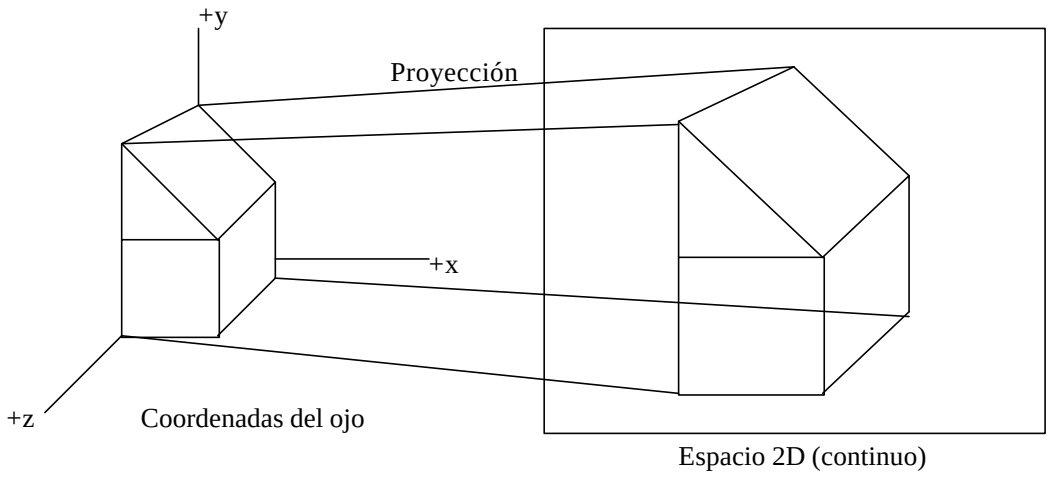
El **mundo real** es la región del espacio cartesiano 3D donde se encuentran los objetos que queremos visualizar. Este espacio, se mide, como es evidente en unidades del mundo real, es decir, en coordenadas continuas y en algún tipo de medida de longitud (metros, milímetros, ...). Estas son las que llamamos **coordenadas del mundo** (*World Coordinates*). Para que los objetos definidos en coordenadas del mundo puedan ser visualizados sobre una pantalla en dos dimensiones, deben sufrir diferentes transformaciones.

Los objetos del mundo real son vistos desde un punto del espacio, denominado **punto de vista**. El punto de vista es el punto del espacio en el que está situado el observador. Este punto define un nuevo sistemas de coordenadas, denominadas **coordenadas del ojo** (*Eye Coordenates*) pues es donde se encuentra el ojo que mira al mundo real. La primera transformación que deben sufrir los puntos del objeto consiste en ponerlos en función del sistema de coordenadas del ojo. Esta transformación se llama **transformación de vista** (*Viewing Transformation*).



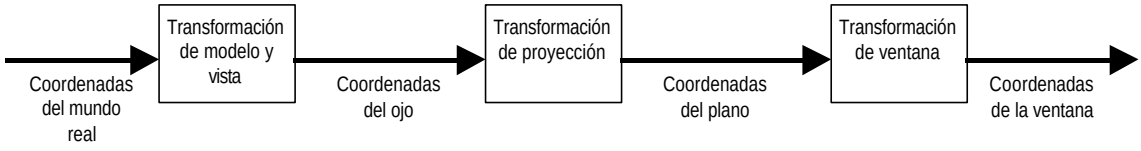
A los objetos situados en el mundo real se le pueden aplicar tres operaciones básicas: **traslación**, **rotación** y **escalado**. Estas operaciones permiten situar el objeto en la posición elegida (traslación), darle la orientación adecuada (rotación) y alterar sus dimensiones (escalado). La aplicación de un conjunto de estas operaciones constituye la segunda transformación: **transformación de modelo** (*Modeling Transformation*). Las transformaciones de vista y de modelo son duales, es decir, aplicar las operaciones elementales vistas es equivalente a alterar la posición del punto de vista y viceversa. Por esta razón utilizamos una única transformación para representar este proceso dual: **transformación de modelo y vista** (*Modelview Transformation*).

La pantalla del ordenador es un plano finito, por lo que no puede representar todo el espacio del mundo real. Debemos definir qué porción del espacio vamos a visualizar. Esta región se denomina **volumen de recorte** y está limitado tanto en alto y en ancho como en profundidad. Además, los objetos que hemos de visualizar se encuentran definidos en las coordenadas tridimensionales (x,y,z). Para poder visualizarlos en una pantalla (espacio con dos dimensiones) es necesario transformar estos puntos tridimensionales en puntos de un espacio 2D (x,y), utilizando algún tipo de proyección (ortográfica o perspectiva). El proceso de recorte y el paso de los puntos en coordenadas del ojo a puntos en **coordenadas del plano** (*Clip Coordinates*) configuran la segunda de las transformaciones necesarias, denominada **transformación de proyección** (*Projection Transformation*). La utilización de un volumen de recorte permite eliminar de antemano los objetos que se encuentran fuera del mismo y acelerar, de esta manera, el proceso de visualización. La forma de ese volumen de recorte determina el tipo de proyección como veremos más adelante. En la siguiente figura representamos gráficamente este proceso.



El último paso es convertir las coordenadas 2D continuas en las **coordenadas de la ventana**, que son bidimensionales y discretas. Este paso se denomina **transformación de ventana** (*Viewport Transformation*).

De forma resumida, los pasos que debemos dar para convertir las coordenadas tridimensionales de un objeto en el mundo real a coordenadas bidimensionales discretas de la ventana (en pixels) son:



MATRICES DE TRANSFORMACIÓN

Existen, como hemos comentado, tres operaciones básicas que pueden aplicarse a vértices de un objeto: **traslación**, **rotación** y **escalado**. Para aplicar una de estas operaciones es suficiente con multiplicar el punto en coordenadas homogéneas por una matriz (también en coordenadas homogéneas). Todas las transformaciones de coordenadas vistas en el apartado anterior pueden representarse mediante composición (multiplicación) de las matrices de las operaciones básicas. Estas matrices son:

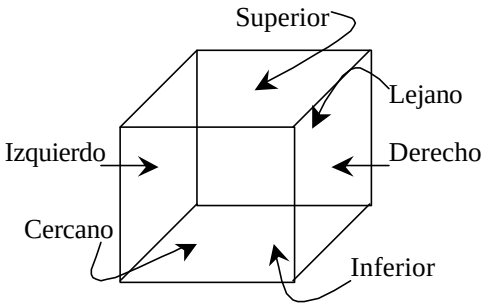
$$T=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\t_x&t_y&t_z&1\end{bmatrix}\quad S=\begin{bmatrix}s_x&0&0&0\\0&s_y&0&0\\0&0&s_z&0\\0&0&0&1\end{bmatrix}\quad R_x(a)=\begin{bmatrix}1&0&0&0\\0&\cos a&\text{sen}\,a&0\\0&-\text{sen}\,a&\cos a&0\\0&0&0&1\end{bmatrix}\quad R_y(a)=\begin{bmatrix}\cos a&0&-\text{sen}\,a&0\\0&1&0&0\\\text{sen}\,a&0&\cos a&0\\0&0&0&1\end{bmatrix}\quad R_z(a)=\begin{bmatrix}\cos a&\text{sen}\,a&0&0\\-\text{sen}\,a&\cos a&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$$

Figura 3.4.1 Matrices de traslación (T), escalado(S) y rotación (R_x, R_y y R_z)

PROYECCIONES

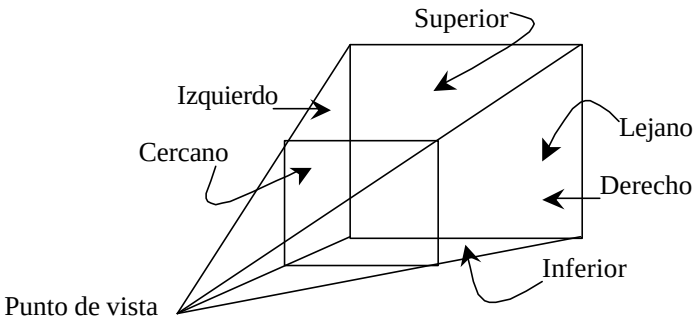
Como ya hemos comentado, las **proyecciones** realizan el paso de 3D a 2D. Las dos proyecciones principales que vamos a ver son la **proyección ortográfica o paralela** y la **proyección perspectiva**.

En la proyección ortográfica o paralela el punto de vista se supone que se encuentra en el infinito y, por lo tanto el volumen de recorte es un prisma. La proyección se realiza trazando visuales paralelas (líneas que unen el punto de vista con el punto a proyectar). Para esta proyección debemos especificar los planos cercano, lejano, derecho, izquierdo, superior e inferior del volumen de recorte.



Volumen de recorte

En la proyección perspectiva el punto de vista se encuentra en algún punto del espacio, por lo tanto el volumen de recorte es un tronco de pirámide. La proyección se realiza trazando visuales que convergen en el punto de vista. Debemos especificar también los planos cercano, lejano, derecho, izquierdo, superior e inferior del volumen de recorte.



Volumen de recorte

Las proyecciones, como transformaciones que son, pueden realizarse utilizando la multiplicación de matrices.

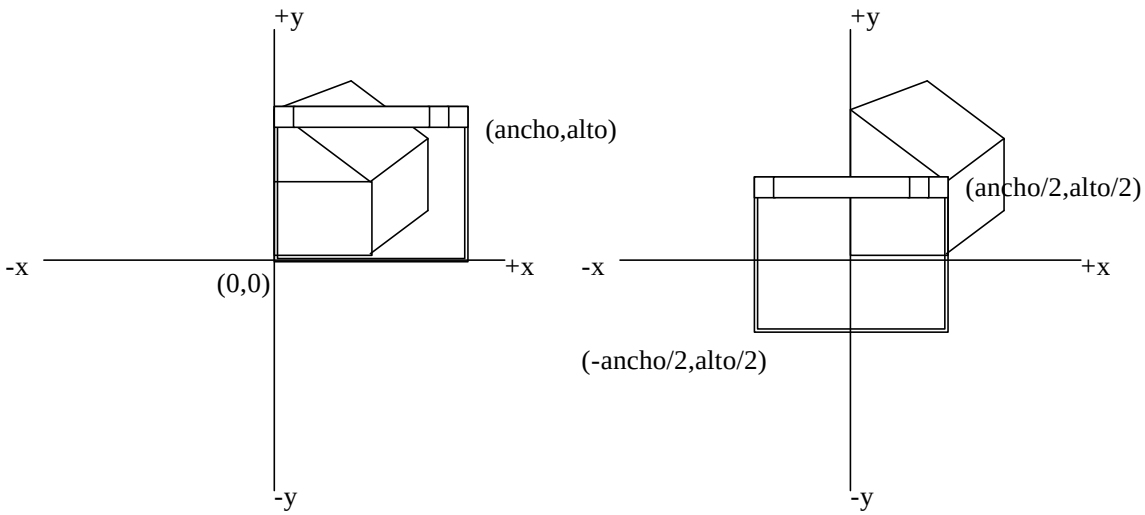
TRANSFORMACIONES EN OPENGL

OpenGL incorpora todas estas transformaciones que hemos visto. En la mayoría de los casos no es necesario el manejo directo de las matrices, puesto que OpenGL tiene predefinidas las matrices más habituales.

OpenGL dispone de dos matrices, una para la transformación de proyección y otra para la de modelo y vista, de manera que a cada vértice se le aplican las matrices de proyección y de modelo y vista que se hayan definido. La forma de definir las matrices es irles multiplicando matrices de operaciones elementales. Hemos de tener en cuenta que la multiplicación de matrices no es conmutativa, es decir, la aplicación de las transformaciones tiene que definirse en el orden correcto.

La primera transformación que debemos definir en OpenGL es la de la ventana (*Viewport Transformation*). Para esta transformación no existe una matriz. Simplemente es necesario definir el área de la ventana especificando las coordenadas de la esquina inferior izquierda de la ventana y el ancho y el alto de la misma. Hemos de tener en cuenta que estas coordenadas son discretas, puesto que representan el tamaño y la posición en pixels de la ventana.

Dos casos típicos son los de áreas de ventana situadas en el 1er cuadrante y centradas en el origen. Podemos verlas en la siguiente figura.



La función que utilizaremos es `glViewport`:

- `void glViewport (Glint x, Glint y, GLsizei ancho, GLsizei alto)`
  `x, y`: Número de pixels desde el origen de la ventana hasta el origen de coordenadas del viewport.
  `ancho, alto`: Número de pixels en ancho y en alto de la ventana.

Las otras dos transformaciones (la de proyección y la de modelo y vista) disponen de una matriz cada una a la que ir multiplicando las distintas matrices elementales que hemos de aplicar. El primer paso es seleccionar la matriz. Esto se

realiza con la función `glMatrixMode`, y su efecto es poner la **matriz seleccionada como activa**, de manera que las matrices que se definan se multiplican por esa matriz activa.

- `void glMatrixMode (GLenum modo)`
  `modo`: indica la matriz que se pone como activa. Puede valer `GL_PROJECTION` (matriz de la transformación de proyección), `GL_MODELVIEW` (matriz de la transformación de modelo y vista) o `GL_TEXTURE` (matriz de la transformación de textura, que veremos más adelante).

Inicialmente es conveniente asegurarse de que la matriz esté "vacía". Para ello lo que hacemos es inicializarla con la matriz identidad, que no realiza ninguna operación, utilizando la función `glLoadIdentity`:

- `void glLoadIdentity (void)`
  Carga en la matriz activa la matriz identidad.

También es posible inicializar la matriz activa con matrices diferentes de la identidad con la función `glLoadMatrix`:

- `void glLoadMatrixd (const GLdouble *matriz)`
- `void glLoadMatrixf (const GLfloat *matriz)`
  `matriz`: matriz de `GLdouble` o `GLfloat` que hay cargar en la matriz activa. Esta matriz ha debido definirse anteriormente.

Ahora la matriz activa se encuentra preparada para irle multiplicando las matrices que deseemos. Para ello podemos definir la matriz y multiplicarla utilizando la función `glMultMatrix`.

- `void glMultMatrixd (const GLdouble *matriz)`
- `void glMultMatrixf (const GLfloat *matriz)`
  `matriz`: matriz de `GLdouble` o `GLfloat` que se multiplica por la matriz activa. Esta matriz ha debido definirse anteriormente.

En el caso de las matrices predefinidas es posible, sin embargo, utilizar funciones que facilitan el proceso y, además, aceleran la visualización. Para el caso de la transformación de proyección es posible multiplicar la matriz activa (lógicamente debe ser la de la transformación de proyección) por las matrices de proyección paralela y perspectiva utilizando las siguientes funciones:

- `void glOrtho (GLdouble izquierda, GLdouble derecha, GLdouble superior, GLdouble inferior, GLdouble cercano, GLdouble lejano)`
  `izquierda, derecha, superior, inferior, cercano, lejano`: posición de los planos en coordenadas continuas del volumen de recorte para una proyección paralela.

- `void glFrustum (GLdouble izquierda, GLdouble derecha, GLdouble superior, GLdouble inferior, GLdouble cercano, GLdouble lejano)`
  `izquierda, derecha, superior, inferior, cercano, lejano`: posición de los planos en coordenadas continuas del volumen de recorte para una proyección perspectiva. El punto de vista se supone situado en el origen (0,0,0).

Para las operaciones elementales también existen matrices predefinidas que se multiplican por la matriz activa mediante las siguientes funciones:

- `void glTranslated (GLdouble x, GLdouble y, GLdouble z)`
- `void glTranslatef (GLfloat x, GLfloat y, GLfloat z)`
  `x, y, z`: cantidad en que se traslada el punto en los tres ejes.

- `void glScaled (GLdouble x, GLdouble y, GLdouble z)`
- `void glScalef (GLfloat x, GLfloat y, GLfloat z)`
  `x, y, z`: cantidad en que se escala el punto en los tres ejes.

- `void glRotated (GLdouble angulo, GLdouble x, GLdouble y, GLdouble z)`
- `void glRotatef (GLfloat angulo, GLfloat x, GLfloat y, GLfloat z)`
  `angulo`: angulo en que se rota el punto. Se define en sentido antihorario.
  `x, y, z`: Define un vector que pasa por el origen y (x,y,z), alrededor del cual se rota.

Cuando se trabaja con OpenGL debemos definir en primer lugar la transformación de ventana (el *viewport*), a continuación la transformación de proyección y, por último, la de modelo y vista. Es en esta matriz donde se almacenan las sucesivas matrices que utilizamos para rotar, escalar o trasladar objetos y, por lo tanto, la que debe quedar activa tras las inicializaciones de las anteriores.

Muy a menudo es necesario volver a estados anteriores de las matrices. Por ejemplo, supongamos que diseñamos un objeto situado en el origen de coordenadas y deseamos dibujar varias copias del mismo en diferentes posiciones del espacio. A ese objeto le aplicaríamos en primer lugar una serie de operaciones de traslación, rotación y escalado hasta colocarlo en la posición adecuada. Para dibujar la segunda copia debemos devolver de nuevo el objeto al origen. Tendríamos en ese caso que aplicar las inversas de las matrices en orden inverso. Para evitar esta tarea, OpenGL incorpora **dos pilas de matrices**, una para la transformación de proyección y otra para la de modelo y vista. Podemos utilizar esas pilas para guardar una matriz que nos interese conservar. En otro punto del programa podemos recuperar esa matriz sin más que desapilarla. Las funciones asociadas a las pilas son:

- `void glPushMatrix (void)`
  Apila la matriz actual en la pila actual.

- `void glPopMatrix (void)`
  Desapila la matriz de la cima de la pila actual, que pasa a ser la matriz actual.

PRIMITIVAS DE DISEÑO

El diseño en OpenGL se realiza utilizando tres primitivas básicas: **puntos**, **líneas** y **polígonos**. Todas ellas se basan en un elemento fundamental: los **vértices**. La función de creación de vértices es `glVertex`. Esta función tiene alrededor de 25 versiones en función del número y tipo de los argumentos. Algunas de las versiones más utilizadas son:

- `void glVertex2f (GLfloat x, GLfloat y)`
- `void glVertex3f (GLfloat x, GLfloat y, GLfloat z)`

- `void glVertex4f (GLfloat x, GLfloat y, GLfloat z, GLfloat w)`
  `x, y, z`: coordenadas de la posición del vértice. Si `z` no se especifica, vale 0.
  `w`: cuarta coordenada cuando se utilizan coordenadas homogéneas. Si no se especifica, vale 1.

La definición de primitivas comienza siempre con la función `glBegin` y termina con `glEnd`. Entre estas dos funciones debemos definir los vértices que forman la primitiva.

- `void glBegin (GLenum primitiva)`
  primitiva: tipo de primitiva que iniciamos. Puede valer: `GL_POINTS` (puntos), `GL_LINES` (líneas), `GL_LINE_STRIP` (polilíneas), `GL_LINE_LOOP` (polilíneas cerradas), `GL_TRIANGLES` (triángulos), `GL_TRIANGLE_STRIP` (triángulos encadenados), `GL_TRIANGLE_FAN` (triángulos alrededor de un punto), `GL_QUADS` (cuadriláteros), `GL_QUAD_STRIP` (cuadriláteros encadenados), `GL_POLYGON` (polígono convexo con cualquier número de vértices).

  Esta función marca el inicio de un conjunto de vértices que definen una o varias primitivas.

- `void glEnd (void)`
  Marca el final de la última primitiva que se haya iniciado con `glBegin`.

Entre las sentencias `glBegin` y `glEnd` puede aparecer código adicional para el cálculo de los vértices y para fijar algunos atributos de las primitivas. Sólo las siguientes llamadas a funciones de OpenGL se permiten dentro del bloque `glBegin-glEnd`:

- `glVertex`
- `glColor`
- `glIndex`
- `glNormal`
- `glEvalCoord`
- `glCallList`
- `glCallLists`
- `glTextCoord`
- `glEdgeFlag`
- `glMaterial`

Iremos viendo estas funciones conforme avancemos en el estudio de las primitivas.

PUNTOS

Los **puntos** son las primitivas más sencillas.. En este caso pueden definirse varios puntos dentro del un solo bloque `glBegin-glEnd`. Los dos ejemplos siguientes producen el mismo efecto, aunque el segundo es más compacto y más rápido:

EJEMPLO

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

glBegin (GL_POINTS);
      glVertex3f (50.0, 50.0, 50.0);
glEnd ();
```

EJEMPLO

```
glBegin (GL_POINTS);
      glVertex3f (0.0, 0.0, 0.0);
      glVertex3f (50.0, 50.0, 50.0);
glEnd ();
```

Uno de los atributos de los puntos que podemos alterar es su tamaño. Matemáticamente, un punto no tiene dimensión. Sin embargo, para poder ser representado en una pantalla debemos darle dimensión. Si no se indica, la dimensión por defecto es 1. La siguiente función permite cambiar el tamaño del punto y tiene que ser llamada fuera del bloque `glBegin-glEnd`.

- `void glPointSize (GLfloat tamaño)`
  `tamaño`: diámetro en pixels del punto.

EJEMPLO

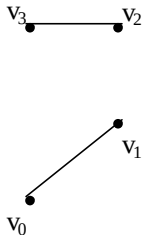
```
int i, j;

for (i=1; i<5; i++)
{
      glPointSize (i);

      glBegin (GL_POINTS);
      for (j=0; j<100; j+=10)
                  glVertex3f (j, j, j);
      glEnd ();
}
```

LÍNEAS Y POLILÍNEAS

La primitiva para diseño de **líneas** (GL_LINES) permite trazar segmentos de línea independientes entre dos puntos sucesivos. El número de puntos debe ser, por lo tanto par (si es impar, se ignora el último). Cuando se trata de **polilíneas** (GL_LINE_STRIP), los segmentos de líneas se encuentran unidos unos con otros. En el caso de **polilíneas cerradas** (GL_LINE_LOOP), se traza un segmento entre los puntos inicial y final para cerrarla. A continuación vemos las diferencias entre los tres tipos de líneas:

EJEMPLO

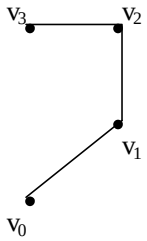
```
glBegin (GL_LINES);
```



```
        glVertex3f (0, 0, 0);
        glVertex3f (50, 50, 0);
        glVertex3f (50, 100, 0);
        glVertex3f (0, 100, 0);
    glEnd ();
```

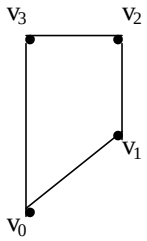
EJEMPLO

```
    glBegin (GL_LINE_STRIP);
        glVertex3f (0, 0, 0);
        glVertex3f (50, 50, 0);
        glVertex3f (50, 100, 0);
        glVertex3f (0, 100, 0);
    glEnd ();
```



EJEMPLO

```
    glBegin (GL_LINE_LOOP);
        glVertex3f (0, 0, 0);
        glVertex3f (50, 50, 0);
        glVertex3f (50, 100, 0);
        glVertex3f (0, 100, 0);
    glEnd ();
```



Existe una función que nos permite alterar el grosor de la línea. Como en el caso de los puntos, esta función no puede ser llamada desde dentro del bloque `glBegin-glEnd`.

- `void glLineWidth (GLfloat grosor)`
  `grosor`: grosor en pixels de la línea.

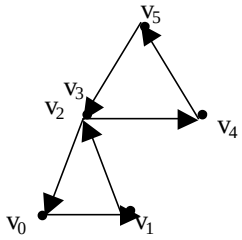
POLÍGONOS

Además de la función general de definición de **polígonos** (GL_POLYGON), también existen funciones para los casos particulares de **triángulos** y **cuadriláteros**. En todos los casos, sin embargo, hay que tener en cuenta que el orden de definición de los vértices es fundamental, puesto que OpenGL utiliza la orientación de los mismos para determinar las caras delantera y trasera. Así, la cara de delante es aquella en la que los vértices se encuentran orientados en sentido antihorario. La de detrás es la que tiene los vértices en sentido horario.

La función de creación de **triángulos** (GL_TRIANGLES) se utiliza para definir triángulos independientes. Estos triángulos han de definirse de tal manera que la cara de delante quede en sentido antihorario. En el caso de los **triángulos encadenados** (GL_TRIANGLE_STRIP), cada nuevo punto sirve para incorporar un nuevo triángulo al conjunto. No es necesario, en este caso tener en cuenta el sentido, pues OpenGL se encarga de mantener en todos los triángulos el sentido antihorario. Cuando los **triángulos** giran **alrededor de un vértice** (GL_TRIANGLE_FAN), el vértice inicial sirve como punto central del conjunto. También en este caso OpenGL mantiene el sentido de los triángulos, aunque en este caso se colocan en sentido horario. Los ejemplos siguientes ilustran las diferencias. Hemos de tener en cuenta que los triángulos independientes generan caras diferentes, mientras que los encadenados forman una única cara. Esto tendrá consecuencias a la hora de realizar el sombreado.

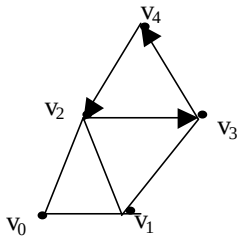
EJEMPLO

```
glBegin (GL_TRIANGLES);
      glVertex3f (0, 0, 0);
      glVertex3f (50, 0, 0);
      glVertex3f (25, 50, 0);
      glVertex3f (25, 50, 0);
      glVertex3f (75, 50, 0);
      glVertex3f (50, 100, 0);
glEnd ();
```



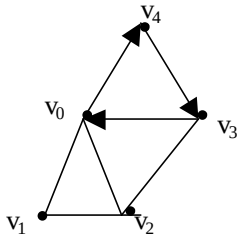
EJEMPLO

```
glBegin (GL_TRIANGLE_STRIP);
      glVertex3f (0, 0, 0);
      glVertex3f (50, 0, 0);
      glVertex3f (25, 50, 0);
      glVertex3f (75, 50, 0);
      glVertex3f (50, 100, 0);
glEnd ();
```



EJEMPLO

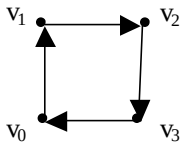
```
glBegin (GL_TRIANGLE_FAN);
      glVertex3f (25, 50, 0);
      glVertex3f (0, 0, 0);
      glVertex3f (50, 0, 0);
      glVertex3f (75, 50, 0);
      glVertex3f (50, 100, 0);
glEnd ();
```



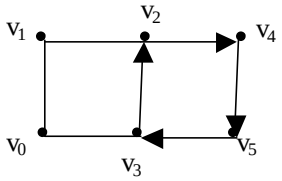
El caso de los **cuadriláteros** es muy similar al de los triángulos. Se pueden definir cuadriláteros **independientes** (GL_QUADS) o **encadenados** (GL_QUAD_STRIP). En ambos casos debe mantenerse orientación horaria, aunque en el caso de los encadenados esto lo realiza OpenGL automáticamente.

EJEMPLO

```
glBegin (GL_QUADS);
      glVertex3f (0, 0, 0);
      glVertex3f (0, 50, 0);
      glVertex3f (50, 50, 0);
      glVertex3f (50, 0, 0);
glEnd ();
```




```
glBegin (GL_QUADS);
      glVertex3f (0, 0, 0);
      glVertex3f (0, 50, 0);
      glVertex3f (50, 50, 0);
      glVertex3f (50, 0, 0);
      glVertex3f (100, 50, 0);
      glVertex3f (100, 0, 0);
glEnd ();
```

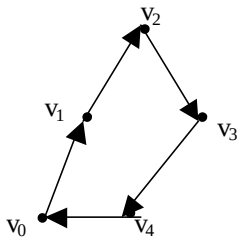


OpenGL permite generar **polígonos con cualquier número de vértices**. La ordenación de éstos es horaria. Todos los polígonos definidos en OpenGL, independientemente del número de lados que tengan, deben cumplir tres condiciones:

deben ser **planares** (todos sus vértices tienen que encontrarse en un mismo plano), sus **aristas no pueden cortarse entre sí**, y deben ser **convexos**. Estas condiciones hacen que los triángulos sean los polígonos más utilizados, ya que cumplen todas las condiciones y, además, cualquier polígono puede descomponerse en un conjunto de triángulos. A continuación vemos como definir un polígono de cinco lados.

EJEMPLO

```
glBegin (GL_POLYGON);
      glVertex3f (0, 0, 0);
      glVertex3f (25, 50, 0);
      glVertex3f (50, 100, 0);
      glVertex3f (75, 50, 0);
      glVertex3f (50, 0, 0);
glEnd ();
```



VISUALIZACIÓN DE POLÍGONOS

Hasta el momento hemos visto la forma de definir polígonos, pero ¿cómo se representan en pantalla? ¿aparecen sólo las aristas o los polígonos son sólidos?. En este apartado responderemos a estas preguntas.

DEFINICIÓN DE LAS CARAS INTERIORES Y EXTERIORES

Como hemos comentado, la **ordenación de los vértices** en el polígono marca la interioridad o exterioridad de la cara que define. Por defecto, cuando estamos en la cara exterior del objeto, los vértices aparecen en sentido antihorario; cuando estamos en el interior, en sentido horario. Sin embargo, esto puede alterarse utilizando la función `glFrontFace`.

- `void glFrontFace (GLenum modo)`
  `modo`: indica qué sentido deben tener los vértices cuando se miran desde el exterior. Puede valer `GL_CW` (sentido horario) o `GL_CCW` (sentido antihorario).

MODOS DE VISUALIZACIÓN

Los polígonos pueden visualizarse de tres **modos**: en **modo sólido** (rellenos con el color actual), en **modo alámbrico** (sólo se visualizan las aristas) o en **modo puntos** (sólo se visualizan los vértices). Es más, podemos visualizar la parte exterior y la parte interior de los polígonos de dos modos diferentes. La función que cambia el modo de visualización es `glPolygonMode`. Una vez que hemos cambiado el modo de visualización, todos los polígonos que se definan a continuación se visualizarán en ese modo, hasta que se cambie de nuevo.

- `void glPolygonMode (GLenum cara, GLenum modo)`

`cara`: indica la cara a la que aplicamos el modo de visualización. Puede valer `GL_FRONT` (cara exterior), `GL_BACK` (cara interior) o `GL_FRONT_AND_BACK` (ambas caras).
`modo`: indica el modo de visualización que se aplica. Puede valer `GL_FILL` (modo sólido; es el modo por defecto), `GL_LINE` (modo alámbrico) o `GL_POINT` (modo puntos).

En ocasiones puede interesarnos que ciertas aristas del objeto no se visualicen en modo alámbrico. Por ejemplo, supongamos un polígono no convexo que hemos dividido en triángulos. Las aristas que hemos añadido para formar los triángulos no deben visualizarse, puesto que no forman parte del polígono. Disponemos de una función, `glEdgeFlag`, que nos permite ocultar ciertas aristas y vértices en modo alámbrico o en modo puntos (no afecta cuando visualizamos en modo sólido). Esta función se utiliza en la definición del polígono, es decir, dentro del bloque `glBegin-glEnd` e indica que se altera la forma de visualizar las aristas que unen los vértices que se definen a continuación. Afecta a todos los puntos siguientes hasta que se cambie de nuevo.

- `void glEdgeFlag (GLboolean flag)`
- `void glEdgeFlagv (const GLboolean *flag)`
  `flag`: vale `true` cuando la arista forma parte del polígono, es decir, cuando debe visualizarse. Vale `false` cuando no debe visualizarse.

OCULTACIÓN DE CARAS (Z-BUFFER Y CULLING)

Cuando visualizamos un objeto, aquellos polígonos que se encuentran detrás de otros, desde el punto de vista del observador, no deben dibujarse. El **algoritmo de eliminación de caras ocultas** más conocido es el del *Z-buffer*. OpenGL mantiene un *Z-buffer* (o ***buffer* de profundidad**) para realizar la ocultación de caras, cuyo manejo es muy sencillo. Consiste en una matriz en la que se almacena la profundidad (el valor de la componente z) para cada pixel. De esta manera es posible conocer qué elementos se encuentran delante de otros.

El algoritmo del *Z-buffer* es bastante costoso. Para incrementar la velocidad de proceso, OpenGL permite **eliminar** previamente las **caras interiores** de los objetos (*culling*). Obviamente, este proceso es válido para objetos cerrados, en los que las caras interiores nunca están visibles. En el caso de objetos abiertos, utilizar *culling* puede producir efectos no deseados. El uso de la eliminación de caras interiores debe hacerse con precaución, puesto que la interioridad o exterioridad de las caras depende de su orientación y del valor que hayamos introducido con `glFrontFace`.

Para activar el *Z-buffer* y el proceso de *culling* se utilizan las funciones `glEnable` y `glDisable`. Como veremos más adelante, estas funciones se utilizan para activar y desactivar otras muchas opciones.

- `void glEnable (GLenum valor)`
- `void glDisable (GLenum valor)`
  `valor`: es el elemento que se activa o desactiva. Puede valer, entre otras cosas, `GL_DEPTH_TEST` para activar / desactivar el *Z-buffer* o `GL_CULL_FACE` para activar / desactivar el algoritmo de *culling*.

En cuanto al Z-*buffer*, debe ser inicializado cada vez que se visualiza la escena, mediante la función `glClear`. Esta función se utiliza para inicializar otros *buffers*, tal y como veremos en apartados posteriores.

- `void glClear (GLuint bits)`
  `bits`: indica el elemento que se inicializa. Para inicializar el Z-*buffer* debemos utilizar el valor `GL_DEPTH_BUFFER_BIT`.

COLOR

En este apartado veremos los diferentes modelos de color que incorpora OpenGL, como dar color a los objetos y al fondo y los diferentes modos de colorear los polígonos.

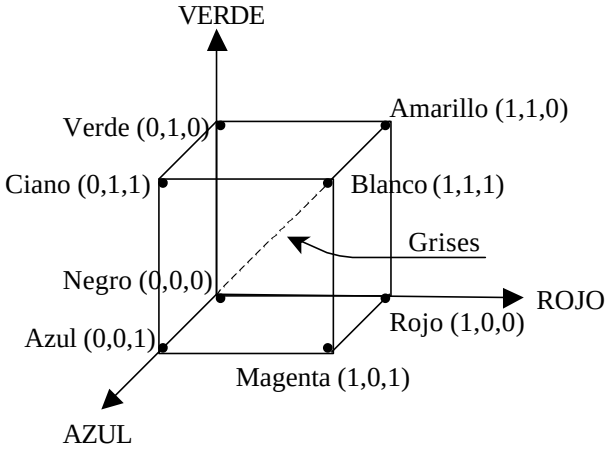
MODELOS DE COLOR

OpenGL incorpora dos **modelos de color**: **modo RGBA** y **modo índices de color**. En el primer caso el color se define a partir de tres valores de color (correspondientes al rojo, al verde y al azul) y, opcionalmente, un valor de alfa, que indica la translucidez del color. En el modo de índices de color, los colores se encuentran ordenados y se accede a ellos a través de un único valor que se corresponde con un índice asociado al color.

El modo RGBA es el más versátil y el más utilizado. El modo de índices de color se utiliza en casos muy específicos. En este texto nos centraremos en el uso del modo RGBA.

EL CUBO DEL COLOR

Los colores en modo RGB (de momento obviaremos la cuarta componente alfa) se pueden representar utilizando un **cubo** de lado 1, donde cada eje representa uno de los tres colores. Los demás vértices representan los colores amarillo, magenta, ciano, blanco y negro. Cada punto en el interior del cubo representa un color, y sus coordenadas indican la cantidad de rojo, verde y azul que lo componen. Sobre la línea que une dos colores (dos puntos) se encuentra toda la graduación existente entre los dos. Así entre el blanco y el negro, se representa toda la gama de grises.



Para seleccionar un color debemos utilizar la función `glColor`, cuyos parámetros son los valores RGB del color. Una vez seleccionado un color, todo lo que se dibuje se coloreará con ese color, hasta que cambiemos a otro. La función `glColor` tiene varias versiones, según se utilice la componente alfa de translucidez, y según el tipo de los argumentos. Con respecto al tipo, podemos utilizar dos notaciones: con tipos reales, los parámetros toman valores entre 0 (ausencia del color) hasta 1 (saturación total del color), tal y como hemos visto en la representación con el cubo. Con tipos enteros, los valores se toman entre 0 (ausencia del color) hasta 255 (saturación total del color), al estilo de cómo lo hacen otros sistemas. A continuación presentamos algunas de las versiones más utilizadas de la función `glColor`.

- `void glColor3f (GLfloat red, GLfloat green, GLfloat blue)`
- `void glColor4f (GLfloat red, GLfloat green, GLfloat blue, GLfloat alfa)`
- `void glColor3i (GLint red, GLint green, GLint blue)`
- `void glColor4i (GLint red, GLint green, GLint blue, GLint alfa)`

  `red, green, blue`: valores para el rojo, verde y azul. En las versiones de tipo real toman valores en el intervalo [0, 1]; en las de tipo entero en [0, 255].

  `alfa`: valor para la componente de translucidez. En las versiones de tipo real toma valores en el intervalo [0, 1]; en las de tipo entero en [0, 255].

La cantidad de colores que nuestro ordenador es capaz de representar depende de la tarjeta de vídeo de que dispongamos. Así, si la tarjeta tiene una **profundidad de color** (la profundidad es el número de bits que utilizamos para representar el color de cada pixel) de 4 bits, puede representar 16 colores; si la profundidad es de 8 bits, el número de colores disponibles es de 256; y si es de 24 bits, puede representar más de 16 millones de colores. Si al utilizar la función `glColor` seleccionamos un color del que no dispone nuestro sistema, OpenGL se encarga de seleccionar en su lugar el color más cercano al elegido.

COLOREAR EL FONDO

Al igual que el buffer de profundidad o *Z-buffer*, OpenGL mantiene un *buffer* de color que indica el color en que debe dibujarse cada pixel. Para borrar el área de dibujo y colorearla a un color, basta con inicializar ese buffer al color deseado. Para ello utilizamos las funciones `glClearColor` y `glClear` (esta última función es la misma que permite inicializar el *buffer* de profundidad).

- `void glClearColor (GLclampf red, GLclampf green, GLclampf blue, GLclampf alfa)`

  `red, gree, blue, alfa`: son las cuatro componentes del color. El tipo `GLclampf` es un tipo real, por lo tanto estos argumentos toman valores en el intervalo [0, 1].

  Esta función indica el color con que debe inicializarse el buffer de color.

- `void glClear (GLuint bits)`

  `bits`: indica el elemento que se inicializa. Para inicializar el *buffer* de color debemos utilizar el valor `GL_COLOR_BUFFER_BIT`.

Puesto que muy habitualmente tanto el *buffer* de color como el de profundidad deben ser inicializados cada vez que se redibuja la escena, puede llamarse una sola vez a la función `glClear` utilizando un conector de tipo OR:

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

COLOREAR LAS PRIMITIVAS

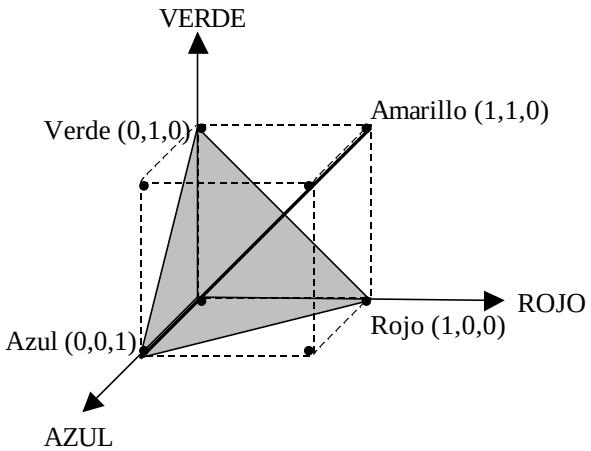
La función `glColor` que ya hemos visto sirve para seleccionar el color en que se dibujan todos los elementos que se diseñen a continuación. Cuando definimos una primitiva de diseño, el color se asigna a cada uno de sus vértices y no a la primitiva completa. Esto implica que puede haber primitivas como líneas o polígonos en las que cada vértice se dibuje de un color. Entonces ¿cómo se colorea el interior de la primitiva?.

En el caso de los puntos no hay problema. Puesto que el vértice tiene asignado un color, el punto se dibujará con ese color. El caso de las líneas y los vértices es más complicado. El coloreado del interior depende, en estos casos, del **modelo de sombreado** (*shading model*) elegido: plano o suavizado. El modelo de sombreado se elige con la función `glShadeModel` que afecta a todas las primitivas que se definan posteriormente, hasta que sea alterado.

- `void glShadeModel (GLenum modo)`
  `modo`: modelo de sombreado elegido. Puede valer `GL_FLAT` (modelo plano) o `GL_SMOOTH` (modelo suavizado, que es el valor por defecto).

En el **modelo plano** (GL_FLAT) el interior de la línea o del polígono se colorea completamente a un único color. Este color es el correspondiente al último vértice de la primitiva, en el caso de las líneas, o al primer vértice de la primitiva, en el caso de los polígonos.

En el **modelo suavizado** (GL_SMOOTH) el interior de las primitivas se colorea haciendo una interpolación o suavizado entre los colores asociados a los vértices. Así, si una línea tiene un vértice de color amarillo y otro de color azul, la parte interior se colorea en toda la gama de colores que van del amarillo al azul. Lo mismo ocurre en el caso de los polígonos, aunque se juega en este caso con tres, cuatro o más colores. El degradado puede representarse sobre el cubo de color mediante la recta que une dos colores, en el caso de las rectas, o el plano que une tres colores, en el caso de los triángulos, etc.



Evidentemente, el modelo de sombreado elegido sólo afecta cuando los vértices de las primitivas tienen colores distintos. Si los colores fueran iguales, la representación equivale al modelo plano.

ILUMINACIÓN

La iluminación es el elemento que más contribuye a la apariencia real de los gráficos por ordenador. La luz aumenta el efecto de tridimensionalidad de los objetos y la forma en que el objeto refleja esa luz nos da una idea del material del que está hecho.

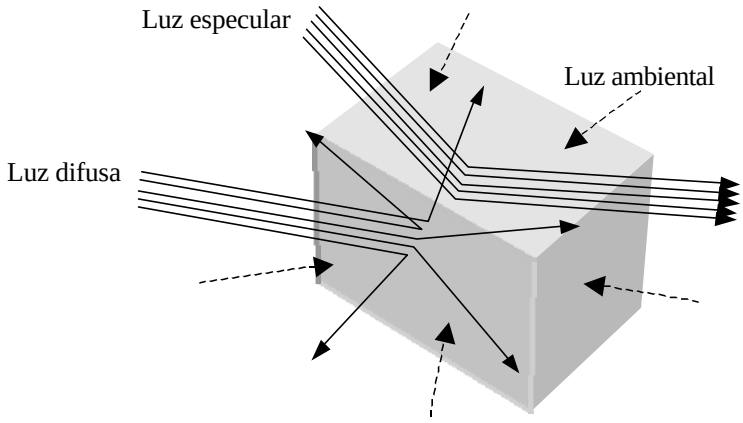
TIPOS DE ILUMINACIÓN

Los objetos que no emiten su propia luz, reciben tres **tipos de luz** diferentes: **luz ambiental**, **luz difusa** y **luz especular**.

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

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

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



Cada foco de luz que incluyamos en la escena tiene tres componentes: luz ambiental, luz difusa y luz especular. La proporción de cada componente determina el tipo de luz. Por ejemplo, un láser esta casi totalmente formado por luz especular. Las componentes difusa y ambiental son mínimas y se deben a la presencia de polvo en el ambiente. Una bombilla, sin embargo, tiene una parte importante de luz ambiental y difusa, y una parte mínima de luz especular.

Cuando se define un foco de luz, no sólo es necesario definir las tres componentes de la luz, sino también el color de cada componente. Así, la definición de un láser rojo puede ser la siguiente:

Componente	Rojo	Verde	Azul
Ambiental	0.05	0.0	0.0
Difusa	0.10	0.0	0.0
Especular	0.99	0.0	0.0

MATERIALES

El aspecto final de los objetos diseñados no depende únicamente de la luz que reciban, sino también del **material** del que estén hechos. El material tiene dos atributos que determinan de manera muy importante la visualización: el color y la forma en que se refleja la luz.

Cuando un objeto es de color azul, realmente lo que ocurre en que absorbe todos los colores del espectro de la luz excepto el azul, que es reflejado. Como consecuencia de ello, el color final con el que veremos el objeto depende del color del objeto y del color de la luz. Por ejemplo, el objeto azul iluminado por una luz roja, se verá negro, puesto que toda la luz será absorbida.

El material, además del color, posee propiedades de **reflectancia** de la luz, que marcan cómo se refleja cada componente de la luz. Así, un material que tenga alta reflectancia para las componentes ambiental y difusa de la luz, pero baja para la especular, se iluminará principalmente de manera ambiental y difusa, por muy especular que sea la luz con el que lo iluminemos. Las propiedades de reflectancia de los objetos se fijan también utilizando el modelo RGB. Un objeto azul, con alta reflectancia para la luz ambiental y difusa, y baja para la especular, podría definirse de la siguiente manera:

Reflectancia	Rojo	Verde	Azul
Ambiental	0.00	0.0	0.80
Difusa	0.00	0.0	0.70
Especular	0.00	0.0	0.10

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

EJEMPLO

Luz

Componente	Rojo	Verde	Azul
Ambiental	0.7	0.6	0.1
Difusa	0.5	0.4	0.1
Especular	1.0	0.3	0.0

Material

Reflectancia	Rojo	Verde	Azul
Ambiental	0.4	0.1	0.80
Difusa	1.0	0.2	0.70
Especular	0.9	0.1	0.10

Valor final

Reflectancia	Rojo	Verde	Azul
Ambiental	0.28	0.06	0.08
Difusa*	0.3	0.05	0.04
Especular*	0.6	0.02	0.0
Valor final	1	0.13	0.12

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

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

ACTIVAR Y DESACTIVAR LA ILUMINACIÓN

El primer paso es activar la iluminación. Para activar/desactivar el uso de las luces utilizamos dos funciones que ya hemos visto anteriormente: `glEnable` y `glDisable`. Como dijimos, estas funciones se utilizan para activar y desactivar otras muchas opciones.

- `void glEnable (GLenum valor)`
- `void glDisable (GLenum valor)`
  `valor`: es el elemento que se activa o desactiva. Para activar/desactivar la iluminación utilizamos el valor `GL_LIGHTING`.

DEFINIR LA LUZ AMBIENTAL Y OTRAS CUESTIONES

Una vez activada la iluminación, debemos utilizar la función `glLightModel` para seleccionar tres aspectos de la iluminación. Esta función tiene varias versiones. Sólo veremos dos, que son las que utilizaremos.

- `void glLightModelfv (GLenum elemento, const GLfloat *parametro)`
- `void glLightModeli (GLenum elemento, Glint parametro)`
  `elemento`: indica el elemento o aspecto que estamos definiendo. Puede valer `GL_LIGHT_MODEL_AMBIENT`, `GL_LIGHT_MODEL_TWO_SIDE` o `GL_LIGHT_MODEL_LOCAL_VIEWER`.
  `parametros`: parámetros necesarios según el aspecto que definamos.

Los tres aspectos que se definen utilizando esta función son:

- Fijar la luz ambiente: cuando utilizamos el valor `GL_LIGHT_MODEL_AMBIENT` estamos fijando la luz ambiental de la escena. En este caso, usaremos la primera versión de la función. Los parámetros que necesita son las componentes de la luz

ambiental, es decir, un vector de cuatro componentes con los valores de RGBA (la componente alfa tiene una incidencia compleja que no veremos de momento. Siempre le pondremos valor 1). Por ejemplo, para poner una luz ambiental blanca muy brillante escribiríamos el siguiente código.

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

glEnable (GL_LIGHTING);
glLightModelfv (GL_LIGHT_MODEL_AMBIENT, luzAmbiente);
```

Si no se especifica una luz ambiental, el valor por defecto es (0.2, 0.2, 0.2, 1.0).

- Fijar las caras de los polígonos que se iluminan: utilizamos el valor `GL_LIGHT_MODEL_TWO_SIDE` para indicar si las dos caras de los polígonos son iluminadas. Por defecto, se ilumina la cara exterior del polígono. En este caso utilizaremos la segunda versión de la función. Si el parámetro vale 0, sólo se ilumina la cara exterior. Para iluminar ambas caras, basta con darle cualquier otro valor.

- Modificar los cálculos de los ángulos de reflexión especular: el valor que debemos emplear es `GL_LIGHT_MODEL_LOCAL_VIEWER`, y sirve para fijar la visual que se utiliza para calcular los ángulos de reflexión especular. La versión de la función que usaremos es la segunda, valiendo 0 el parámetro si tomamos una visual paralela al eje z y en dirección -z (el punto de vista se considera situado en el infinito), u otro valor para una visual que parta del origen del sistema de coordenadas del ojo. La primera opción es menos realista pero más rápida; la segunda da resultados más reales pero es menos eficiente.

DEFINIR PUNTOS DE LUZ

Lo visto en el apartado anterior nos permite colocar una luz ambiental general que ilumina a todo el objeto por igual. Sin embargo, en la realidad las luces se encuentran en una posición del espacio y brillan en una dirección. Veremos como colocar luces en una posición. Será necesario, en estos casos, definir no sólo las componentes de la luz, sino también su posición y la dirección en la que emiten luz. OpenGL permite colocar hasta `GL_MAX_LIGHTS` luces diferentes en la escena, cada una con la posición que ocupa y la dirección en la que emite la luz (La constante `GL_MAX_LIGHTS` vale 8 en Linux). Se nombran como `GL_LIGHT0`, `GL_LIGHT1`, etc. Además de la posición y dirección, debemos especificar de qué tipo de luz se trata (difusa, ambiental o especular), su color y otros aspectos más complejos que veremos detenidamente.

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

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

  `luz`: Indica cuál de las luces estamos modificando. Pude valer `GL_LIGHT0`, `GL_LIGHT1`,... `GL_LIGHT7`.

  `valor`: Especifica qué parámetro de la luz estamos fijando. Puede valer `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, `GL_POSITION`, `GL_SPOT_DIRECTION`, `GL_SPOT_EXPONENT`, `GL_SPOT_CUTOFF`, `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION` o `GL_QUADRATIC_ATTENUATION`.

`parametro`: vector de cuatro componentes o valor real, cuyo significado depende del argumento anterior.

Veamos con detenimiento el significado y el uso de la función anterior, para cada posible valor del segundo de los argumentos:

- `GL_AMBIENT`: sirve para fijar la componente ambiental de la luz que estamos definiendo. Para fijar esta característica utilizamos la primera de las versiones de la función. El tercer parámetro es un vector de cuatro componentes que indica la composición RGBA de la luz.

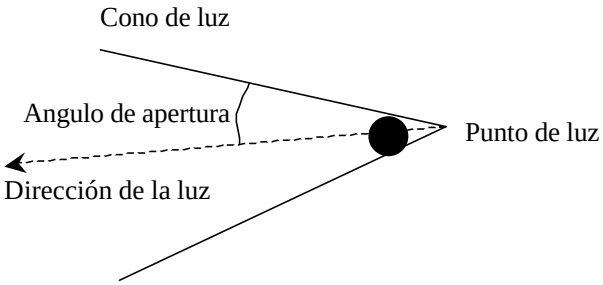
- `GL_DIFFUSE`: la utilizamos para fijar la componente difusa de la luz. Como en el caso anterior utilizamos la primera versión y el tercer parámetro es la composición RGBA de la componente difusa.

- `GL_SPECULAR`: con este valor fijamos la componente especular del foco de luz. Se utiliza, como en los dos casos anteriores, la primera versión y sirve para especificar la composición RGBA de la luz.

- `GL_POSITION`: utilizando este valor fijamos la posición del punto de luz. La versión adecuada es la primera. El tercer parámetro es un vector de cuatro componentes: las tres primeras indican la posición (x,y,z) del punto de luz, y la cuarta componente puede valer 1.0, para indicar que la luz se encuentra en la posición indicada, u otro valor para indicar que se encuentra en el infinito y que sus rayos son paralelos al vector indicado por (x,y,z).

- `GL_SPOT_DIRECTION`: se emplea para definir luces direccionales. Utiliza la primera versión, y el tercer parámetro es un vector de tres componentes que indica la dirección de los rayos de luz. Este vector no necesita ser normalizado y se define en coordenadas del ojo. Si no se indica, la luz brilla en todas direcciones.

- `GL_SPOT_CUTOFF`: Las luces direccionales brillan formando un cono de luz. Esta opción nos sirve para fijar el ángulo de apertura del cono de luz. La versión adecuada en este caso es la segunda, y el último argumento indica el ángulo desde el punto central del cono (marcado por la dirección de la luz) y el lado del cono. Este ángulo debe valer entre 0.0 y 90.0. Por defecto, el ángulo es 180.0º, es decir, si no se indica, la luz se emite en todas direcciones.



- `GL_SPOT_EXPONENT`: Dentro del cono de luz, los puntos más cercanos al centro se encuentran más iluminados. El exponente indica la concentración de luz alrededor del punto central del cono. Se utiliza la segunda versión de la función y el tercer parámetro indica el exponente. El valor por defecto de este exponente es 0,

indicando que todo el cono de luz se ilumina con la misma intensidad. Cuanto mayor es el exponente, más se concentra la luz en el centro del cono.

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

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

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

Una vez fijados todos los parámetros de la luz, sólo falta activarla mediante la función glEnable que ya conocemos:

- `void glEnable (GLenum valor)`
- `void glDisable (GLenum valor)`

`valor`: es el elemento que se activa o desactiva. Para activar/desactivar un punto de luz utilizamos uno de los valores `GL_LIGHT0, GL_LIGHT1, ...`

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

EJEMPLO

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

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

// Activar iluminación
glEnable (GL_LIGHTING);

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

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

DEFINIR LOS MATERIALES

El otro aspecto fundamental que afecta a la visualización es el material del que está hecho el objeto. Definir el material supone fijar las reflectancias para cada tipo de luz, tal y como hemos comentado anteriormente. La forma de definir las características (reflectancias) del material, es utilizando la función `glMaterial`. Esta función afecta a todos los polígonos que se definan a continuación, hasta que sea alterada. Veremos sólo dos versiones de las cuatro disponibles:

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

  `cara`: cara del polígono a la que asignamos el material. Puede valer `GL_FRONT` (cara exterior), `GL_BACK` (cara interior) o `GL_FRONT_AND_BACK` (ambas caras).

  `valor`: característica que estamos fijando. Puede valer `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, `GL_EMISSION`, `GL_SHININESS`, `GL_AMBIENT_AND_DIFFUSE` o `GL_COLOR_INDEXES` (se utiliza cuando estamos usando el modelo de índices de color. No lo vamos a ver).

  `parametro`: valor real o vector de reales que indican el valor correspondiente a la característica elegida.

Veamos a continuación las diferentes características que es posible definir con esta función:

- `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR` y `GL_AMBIENT_AND_DIFFUSE`: Con esta opción se fija las reflectancias correspondientes a la luz ambiental, difusa y especular, además de poder fijar las reflectancias ambiental y difusa en una única llamada si son iguales. La versión utilizada es la segunda y el vector de reales será el valor RGBA de la reflectancia.

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

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

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

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

- `void glEnable (GLenum valor)`

- `void glDisable (GLenum valor)`
  `valor`: es el elemento que se activa o desactiva. Para activar/desactivar el *tracking* de color utilizamos el valor `GL_COLOR_MATERIAL`.

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

- `void glColorMaterial (GLenum cara, GLenum modo)`
  `cara`: especifica a qué cara afecta el *tracking* de color. Puede valer `GL_FRONT`, `GL_BACK` o `GL_FRONT_AND_BACK`.
  `modo`: indica qué característica va a depender del color. Puede valer `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, `GL_EMISSION` o `GL_AMBIENT_AND_DIFFUSE`.

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

EJEMPLO

```
// Activar tracking de color
glEnable (GL_COLOR_MATERIAL);

// Fijar la reflectancia ambiental y difusa en la cara exterior
// como dependiente del color
glColorMaterial (GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
...

// Al alterar el color, estamos alterando también la
// reflectancia ambiental y difusa del material
glColor 3f (0.2, 0.5, 0.8);
...

// Desactivar el tracking de color
glDisable (GL_COLOR_MATERIAL);
```

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

DEFINIR LAS NORMALES

Cuando se ilumina un polígono con una luz direccional, los rayos son reflejados con un ángulo, que depende del ángulo de incidencia de la luz, y de la **normal** a la superficie (además del material, como ya hemos comentado). Sin embargo, los polígonos se definen a partir de sus vértices y, de hecho, los cálculos de la iluminación se realizan sobre los vértices, que no tienen normal. ¿Qué normal utilizamos para calcular el ángulo de reflexión?. Esto es lo que vamos a tratar a continuación.

La normal en el vértice debe ser suministrada por el programador. En principio, esta normal debe ser la normal del polígono que define. Es muy sencillo calcular las normales para un polígono a partir de tres puntos utilizando el producto vectorial de dos vectores. Aunque no es necesario que esta normal esté normalizada, es conveniente

hacerlo. Si no lo hacemos, debemos activar la normalización automática que incorpora OpenGL, mediante la función `glEnable`.

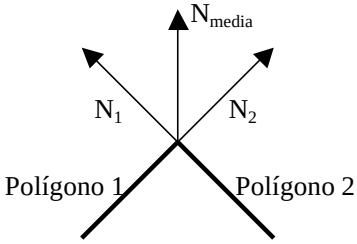
- `void glEnable (GLenum valor)`
- `void glDisable (GLenum valor)`
  `valor`: es el elemento que se activa o desactiva. Para activar/desactivar la normalización de las normales utilizamos el valor `GL_NORMALIZE`.

Sin embargo, es mucho más eficiente normalizarlas nosotros previamente, para evitar que OpenGL deba hacerlo cada vez.

Para definir la normal a un vértice utilizamos la función `glNormal`. Una vez definida, todos los vértices que se creen a continuación tendrán esa normal, hasta que se cambie por otra. Esta función tiene 10 versiones, aunque sólo veremos una.

- `glNormal3f (GLfloat nx, GLfloat ny, GLfloat nz)`
  `nx, xy, nz`: coordenadas (x,y,z) de la normal

Cuando queremos obtener una superficie suave, eliminando el facetado entre polígonos es conveniente utilizar como normal en cada punto la media de las normales de los polígonos adyacentes.



A continuación presentamos dos funciones para el cálculo y la normalización de las normales.

```
// Constantes para manejar los vectores:
// la componente 0 es la X, la 1 la Y, y la 2 la Z

#define X 0;
#define Y 1;
#define Z 2;

void normalizar (float vector[3])
{
      float modulo;

      modulo = sqrt (vector[X]*vector[X] +
                     vector[Y]*vector[Y] +
                     vector[Z]*vector[Z]);

      if (modulo == 0.0)
            modulo = 1.0;

      vector[X] /= modulo;
      vector[Y] /= modulo;
      vector[Z] /= modulo;
}
```

```
void calcularNormales (float puntos[3][3], float normal[3])
{
      float v[3], w[3];

      // Calcular dos vectores a partir de los tres puntos

      v[X] = puntos[0][X] - puntos[1][X];
      v[Y] = puntos[0][Y] - puntos[1][Y];
      v[Z] = puntos[0][Z] - puntos[1][Z];

      w[X] = puntos[1][X] - puntos[2][X];
      w[Y] = puntos[1][Y] - puntos[2][Y];
      w[Z] = puntos[1][Z] - puntos[2][Z];

      // Calcular la normal utilizando el producto vectorial

      normal[X] = v[Y]*w[Z] - v[Z]*w[Y];
      normal[Y] = v[Z]*w[X] - v[X]*w[Z];
      normal[Z] = v[X]*w[Y] - v[Y]*w[X];

      // Normalizar la normal

      normalizar (normal);
}
```

LISTAS DE VISUALIZACIÓN (DISPLAY LISTS)

Una **lista de visualización** o *display list* es un conjunto de comandos que se almacenan para ser ejecutados posteriormente. La ejecución de los comandos se realiza en el orden en que se encuentran cuando se crea.

La mayoría de los comandos de OpenGL se pueden ejecutar de dos maneras:

- **Modo inmediato**: Los comandos se ejecutan conforme se encuentran en el programa. Hasta ahora todos los ejemplos vistos se ejecutan de modo inmediato.
- **Modo diferido**: Los comandos se almacenan en una lista de visualización y son ejecutados en otro punto del programa

Los modos de ejecución no son excluyentes, es decir, en un mismo programa pueden aparecen comandos que se ejecuten en modo inmediato y en modo diferido.

La utilización de listas de visualización incluye dos etapas: **creación** de la lista y **ejecución** de la misma. Durante la creación (también llamada etapa de compilación de la lista, aunque se realice durante la ejecución del programa), se definen las operaciones que forman parte de la lista y se realizan los cálculos, cuyos resultados se almacenan junto con las llamadas a las funciones de OpenGL. Durante la ejecución se realizan las llamadas a las funciones almacenadas.

CREAR UNA LISTA DE VISUALIZACIÓN

Para crear una lista de visualización utilizamos las funciones `glNewList` y `glEndList`. La primera de ellas marca el inicio del conjunto de funciones que se almacenan en la lista, y la segunda el final del mismo.

- `void glNewList (GLuint lista, GLenum modo)`

`lista`: es un entero que identifica a la lista. Debe ser único.
`modo`: indica el modo de la lista. Puede valer `GL_COMPILE` (sólo compila la lista) o `GL_COMPILE_AND_EXECUTE` (la compila y la ejecuta a la vez).
Esta función indica el inicio de una nueva lista de visualización.

- `void glEndList (void)`
  Indica el final de la lista de visualización .

EJEMPLO

```
...
glNewList (1, GL_COMPILE)
      glColor3f (1.0, 0.0, 0.0);
      glBegin (GL_POLYGON)
            glVertex3f (50.0, 100.0, 50.0);
            glVertex3f (100.0, 200.0, 10.0);
            glVertex3f (10.0, 200.0, 50.0);
      glEnd ();
      glTranslatef (3.0, 0.0, 0.0);
glEndList ();
```

En una lista de visualización pueden aparecer funciones de OpenGL y otras que no lo son, aunque sólo se almacenan los comandos de OpenGL. Todas las demás instrucciones que aparezcan se calculan al crear (o compilar) la lista de visualización y se almacenan sus resultados junto con los comandos de OpenGL. Esto permite que las operaciones se realicen en tiempo de compilación de la lista, lo que acelera el proceso de visualización cuando se ejecuta. En general, el uso de listas de visualización resulta más rápido que la ejecución en modo inmediato y, por lo tanto, resulta conveniente utilizarlas. Además, facilitan el encapsulamiento y la reutilización de objetos. Creando una lista y ejecutándola donde necesitemos, estamos realizando un encapsulamiento de las instrucciones que la componen. La ventaja principal de realizar este encapsulamiento con una lista de visualización en vez de con una función, es que las operaciones que incluye se realizan al compilar la lista.

Puesto que en una lista de visualización los cálculos correspondientes a funciones que no son de OpenGL se realizan durante la compilación, no es posible alterar las listas una vez creadas. Sí podremos, no obstante, crear nuevas listas con los mismos identificadores, que sustituirán a las listas previas.

En el siguiente ejemplo dibujamos un círculo en modo inmediato y utilizando una lista de visualización. El cálculo de los vértices es un proceso computacionalmente costoso debido a las funciones trigonométricas que aparecen. La versión con lista de visualización resulta mucho más rápida al ejecutarla, puesto que los cálculos se realizan en tiempo de compilación.

EJEMPLO

```
// Versión en modo inmediato

dibujarCirculo ()
{
      GLint i;
      GLfloat coseno, seno;
      glBegin (GL_POLYGON);
            for (i=0; i<100; i++)
```

```
            {
                  coseno = cos (i*2*PI/100.0);
                  seno = sin (i*2*PI/100.0);
                  glVertex2f (coseno, seno);
            }
      glEnd ();
}

// Versión con lista de visualización

#define LISTA_CIRCULO 1

dibujarCirculo ()
{
      GLint i;
      GLfloat coseno, seno;

      glNewList (LISTA_CIRCULO, GL_COMPILE);
      glBegin (GL_POLYGON);
            for (i=0; i<100; i++)
            {
                  coseno = cos (i*2*PI/100.0);
                  seno = sin (i*2*PI/100.0);
                  glVertex2f (coseno, seno);
            }
      glEnd ();
      glEndList ();
}
```

EJECUTAR UNA LISTA DE VISUALIZACIÓN

Una vez creada la lista de visualización, podemos ejecutarla en cualquier lugar del programa, y tantas veces como queramos. La función que la ejecuta es `glCallList`.

- `void glCallList (GLuint lista)`
  `lista`: identificador de la lista.

En el ejemplo siguiente, suponemos definida la función `dibujarCirculo` del apartado anterior.

EJEMPLO

```
main ()
{
      ...
      // Crear la lista de visualización
      dibujarCirculo ();
      ...
      // Ejecutar la lista de visualización
      glCallList (LISTA_CIRCULO);
      ...
      glCallList (LISTA_CIRCULO);
      ...
}
```

CÓMO Y CUÁNDO UTILIZAR LAS LISTAS DE VISUALIZACIÓN

Vamos a ver algunos consejos sobre la utilización de listas de visualización, así como algunas otras cuestiones adicionales sobre este tema.

El incremento en la velocidad de proceso que se produce al utilizar listas de visualización, depende de la implementación concreta de OpenGL y del hardware sobre el que esté instalado. En cualquier caso, como mínimo el uso de listas de visualización es tan rápido como el modo inmediato. En general, este incremento de velocidad se nota más cuando la lista incluye las siguientes funciones:

- Operaciones con matrices.
- Visualización de bitmaps e imágenes.
- Iluminación y propiedades de los materiales.
- Texturas.
- Cálculos intensivos.
- Lectura de los datos desde disco.

Por otro lado, hemos de tener en cuenta al utilizar listas de visualización, que algunos comandos son sensibles al contexto: por ejemplo, el cálculo de los vértices de un objeto depende de las matrices que se hayan definido anteriormente. Además, los cambios que se produzcan en el contexto dentro de la lista, repercutirán a continuación en los demás comandos, estén dentro de una lista o se ejecuten en modo inmediato. En el ejemplo siguiente, la instrucción `glTranslatef`, hace que cada vez que se ejecuta la lista, se acumule una traslación en la matriz de modelo y vista y que, por lo tanto, los objetos que se dibujen a continuación, incluso la propia lista si se ejecuta de nuevo, sufran esta traslación. De igual manera, el cambio de color afecta a todos los objetos que se ejecuten a continuación.

EJEMPLO

```
void creaTriangulo ()
{
      glNewList (1, GL_COMPILE);
            glColor3f (1, 0, 0);
            glBegin (GL_TRIANGLES);
                  glVertex2f (0, 0);
                  glVertex2f (1, 0);
                  glVertex2f (0, 1);
            glEnd ();
            glTraslatef (1.5, 0, 0);
      glEndList ();
}
```

Para conseguir que no se altere el contexto, debemos utilizar las funciones `glPushMatrix`, `glPopMatrix`, `glPushAttrib` y `glPopAttrib`.

EJEMPLO

```
void crearTriangulo ()
{
      glNewList (1, GL_COMPILE);
            glPushMatrix ();
            glPushAttrib (GL_CURRENT_BIT);
            glColor3f (1, 0, 0);
            glBegin (GL_TRIANGLES);
```

```
                    glVertex2f (0, 0);
                    glVertex2f (1, 0);
                    glVertex2f (0, 1);
              glEnd ();
              glTraslatef (1.5, 0, 0);
              glPopAttrib ();
              glPopMatrix ();
        glEndList ();
}
```

Por otro lado, cabe destacar que no todas las funciones de OpenGL pueden guardarse en una lista de visualización. En concreto, esto ocurre con las funciones que pasan algún parámetro por referencia, o que devuelven un valor. Estas funciones son: `glDeleteList`, `glFeedbackBuffer`, `glFinish`, `glFlush`, `glGenLists`, `glGet`, `glIsEnabled`, `glIsList`, `glPixelStore`, `glReadPixels`, `glRenderMode` y `glSelectBuffer`. Si se incluye alguna de estas funciones, se ejecutará durante la creación (compilación) de la lista, pero no se almacenará en la misma.

Es posible crear listas jerárquicas, es decir, listas que ejecutan otras listas. Las listas jerárquicas son muy útiles cuando estamos definiendo objetos formados por varias componentes, especialmente si algunas de ellas se repiten varias veces. Para ejecutar una lista dentro de otra no es necesario siquiera que aquella esté creada cuando se crea la segunda. Para evitar recursiones infinitas existe un límite en el nivel de anidamiento que es de 64. En el ejemplo siguiente empleamos una lista de visualización para crear una bicicleta, suponiendo que creamos otras listas para definir el manillar, el cuadro y las ruedas.

EJEMPLO

```
#define MANILLAR 1
#define CUADRO 2
#define RUEDA 3
#define BICICLETA 4

glNewList (MANILLAR, GL_COMPILE);
...
glEndList ();

glNewList (CUADRO, GL_COMPILE);
...
glEndList ();

glNewList (RUEDA, GL_COMPILE);
...
glEndList ();

glNewList (BICICLETA, GL_COMPILE);
      glCallList (MANILLAR);
      glCallList (CUADRO);
      glTranslatef (1.0, 0.0, 0.0);
      glCallList (RUEDA);
      glTranslatef (3.0, 0.0, 0.0);
      glCallList (RUEDA);
glEndList ();
```

MANEJAR LAS LISTAS Y SUS ÍNDICES

Los identificadores de las listas de visualización deben ser enteros positivos únicos, lo que implica que debemos tener un especial cuidado en no duplicar índices. La forma más útil, suele ser definir constantes con números correlativos y nombres que faciliten la lectura del programa, tal y como hemos visto en ejemplos anteriores.

Cuando se manejan muchas listas de visualización puede ser útil recurrir a determinadas funciones de OpenGL que proporcionan índices de listas de visualización no utilizados:

- `GLuint glGenLists (GLsizei rango)`
  `rango`: número de índices que queremos obtener
  Esta función nos proporciona tantos índices para listas de visualización no usados como indiquemos con el parámetro `rango`. Devuelve el primer índice vacío y éste junto con los demás, que son correlativos, se marcan como ocupados.

- `GLboolean glIsList (GLuint indice)`
  `indice`: índice de la lista
  Indica si el índice está usado

- `void glDeleteLists (GLuint indice, GLsizei rango)`
  `indice`: índice inicial
  `rango`: número de índices que queremos borrar
  Esta función borra un conjunto de listas de visualización correlativas, definidas a partir de un índice inicial y un rango.

EJEMPLO

```
GLuint indiceLista;

indiceLista = glGenLists (1);
if (glIsList (indiceLista))
{
      glNewList (indiceLista, GL_COMPILE);
            ...
      glEndList();
}
...
glDeleteLists (indiceLista, 1);
```

OpenGL proporciona también la posibilidad de ejecutar varias listas de visualización con una única llamada. Para ello es necesario guardar los índices en un array (que puede ser de cualquier tipo) y utilizar las siguientes funciones:

- `void glCallLists (GLsize n, GLenum tipo, const GLvoid *listas)`
  `n`: número de elementos del array de índices de listas.
  `tipo`: tipo de datos del array. Puede ser `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT`, `GL_FLOAT`, `GL_2_BYTES`, `GL_3_BYTES` o `GL_4_BYTES`.
  `listas`: array de índices de las listas.
  Esta función ejecuta un conjunto de listas de visualización cuyos índices se encuentran en el array `listas`.

- `void glListBase (GLuint base)`
  `base`: valor del offset.
  Fija un valor como offset que debe añadirse a los índices del array que se ejecuta con glCallLists.

La ejecución de varias listas con `glCallLists` se utiliza, por ejemplo, para escribir texto. En el ejemplo siguiente tenemos una función que crea una lista de visualización para cada letra del alfabeto. Una segunda función escribe una cadena de caracteres utilizando llamadas a las listas de visualización correspondientes.

EJEMPLO

```
GLuint CrearFuente ()
{
      GLuint fuente;

      fuente = glGenLists (26);

      /* Crear lista para la letra A */
      glNewList (fuente, GL_COMPILE);
            ...
      glEndList();

      /* Crear lista para la letra B */
      glNewList (fuente+1, GL_COMPILE);
            ...
      glEndList();

      ...
      return (fuente);
}

void EscribirCadena (GLuint fuente, char *cadena)
{
      if (fuente == 0 || cadena == NULL)
            return;

      /* Se pone como base el primer índice de la fuente y le
      restamos 65. Así, para la A (código ASCII 65), se mostrará
      la primera lista, para la B la segunda ... */

      glListBase (fuente-65);
      glCallLists (strlen (cadena), GL_UNSIGNED_BYTE, cadena);
}

main ()
{
      ...
      GLuint fuente;

      fuente = CrearFuente ();
      EscribirCadena (fuente, "OPENGL");
      ...
}
```

MAPAS DE BITS Y DE PIXELS

OpenGL es una librería que trabaja con gráficos vectoriales, como hemos visto hasta el momento. Sin embargo también es posible presentar en pantalla mapas de bits y de pixels, para utilizarlos, por ejemplo, como fondo. Un **mapa de bits** o *bitmap* es una imagen en dos colores, que se utiliza para dibujar rápidamente algunos elementos como caracteres, símbolos, iconos o como máscaras. Un **mapa de pixels** o *pixmap* (mal llamados en ocasiones bitmaps) es una imagen en color, que se utiliza, fundamentalmente, para incorporar un fondo a la escena, o como textura para los polígonos.

MAPAS DE BITS

Para dibujar mapas de bits, OpenGL dispone de la función `glBitmap`. La posición en que se dibuja el bitmap viene marcada por la **posición actual en la pantalla**. Dicha posición puede alterarse utilizando la función `glRasterPos`.

- `void glRasterPos3f (GLfloat x, GLfloat y, GLfloat z)`
  `x, y, z`: coordenadas de la posición.
  Esta función coloca la posición actual en la pantalla en (x,y,z). Obsérvese que estas coordenadas son del mundo real (valores continuos), por lo que la posición en la ventana se calculará transformando las coordenadas (x,y,z) a coordenadas de pantalla, utilizando las transformaciones de modelo-vista y perspectiva definidas en ese momento. Esta función tiene otras versiones con 2, 3 y 4 parámetros, de tipo entero y real.

- `void glBitmap (GLsizei ancho, GLsizei alto, GLfloat xorig, GLfloat yorig, GLfloat xincr, GLfloat yincr, const GLubyte *bitmap)`
  `ancho, alto`: medidas del bitmap
  `xorig, yorig`: coordenadas del origen del bitmap
  `xincr, yincr`: incremento de la posición actual en la pantalla
  `bitmap`: matriz de ceros y unos que forma el bitmap
  Esta función dibuja en pantalla el bitmap, colocando su origen en la posición actual de la pantalla y produciendo un incremento en dicha posición tras dibujar dicho bitmap.

Cuando se dibuja un bitmap con esta función, el primer color (0) es trasparente, mientras que el segundo (1) adquiere el color y los atributos del material activos en ese momento.

La matriz que contiene el bitmap debe tener un número de columnas múltiplo de 8. Sin embargo, el ancho del bitmap puede ser menor. En ese caso, los bits que sobran no se dibujan. Además, las filas de la matriz deben definirse en orden inverso a como deben dibujarse. Las columnas, sin embargo, se definen en el mismo orden.

Como hemos comentado, el bitmap se dibuja en la posición actual, que podemos fijar utilizando `glRasterPos`. Esta posición puede, sin embargo, estar fuera del espacio de la ventana. En ese caso la posición es no válida y el mapa de bits no se dibuja. Podemos conocer la posición actual en la ventana y saber si es válida o no con las funciones `glGetFloatv` y `glGetBooleanv`:

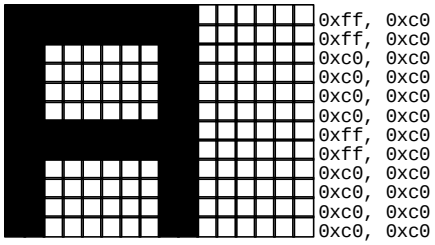
- `void glGetFloatv (GLenum valor, GLfloat *parametros)`
  `valor`: indica qué valor estamos obteniendo. En este caso debemos utilizar `GL_CURRENT_RASTER_POSITION`.
  `parametros`: vector con cuatro reales (x,y,z,w) que indican la posición.

- `void glGetBooleanv (GLenum valor, GLboolean *parametros)`
  `valor`: indica qué valor estamos obteniendo. En este caso debemos utilizar `GL_CURRENT_RASTER_POSITION_VALID`.
  `parametros`: puntero a un valor lógico que indica si la posición actual es válida.

La función `glBitmap`, avanza la posición actual en una determinada cantidad en x y en y, aunque no se altera el valor de validez de la posición actual (incluso en el caso de que esta deje de ser válida). Este comportamiento se aprovecha para el dibujo de caracteres en pantalla: cuando se dibuja una letra, se avanza hacia la derecha para poder dibujar la siguiente. Cuando se llega al final de una línea, debemos retroceder al principio, y bajar a la siguiente línea. En el siguiente ejemplo retomamos la función `CrearFuente` vista en el apartado anterior para completarla con el dibujo de bitmaps. Obsérvese que la matriz para la letra A se crea en orden inverso a como se dibuja

EJEMPLO

```
GLuint CrearFuente ()
{
      GLuint fuente;

      GLubyte letraA[24] = {
      0xc0, 0xc0, 0xc0, 0xc0,
      0xc0, 0xc0, 0xc0, 0xc0,
      0xff, 0xc0, 0xff, 0xc0,
      0xc0, 0xc0, 0xc0, 0xc0,
      0xc0, 0xc0, 0xc0, 0xc0,
      0xff, 0xc0, 0xff, 0xc0};

      GLubyte letraB[24] = {
      ...};

      ...
      fuente = glGenLists (26);

      /* Crear lista para la letra A: tiene 10x12 pixels (aunque
         la matriz tiene 16x12. El origen está en 0,0 y cuando se
         dibuja se avanzan 12 para escribir la siguiente letra */

      glNewList (fuente, GL_COMPILE);
            glBitmap (10, 12, 0.0, 0.0, 12.0, 0.0, letraA);
      glEndList();

      /* Crear lista para la letra B */
      glNewList (fuente+1, GL_COMPILE);
            ...
      glEndList();

      ...
      return (fuente);
}
```



MAPAS DE PIXELS

Para el manejo de mapas de pixels (pixmaps o imágenes en color), OpenGL proporciona tres funciones fundamentales: `glDrawPixels` para dibujar en uno de los buffers un pixmap, `glReadPixels` para leer un array rectangular de pixels de uno de

los buffers y guardarlo en memoria, y `glCopyPixels` para copiar un array rectangular de pixels de un área a otra de alguno de los buffers.

- `void glDrawPixels (GLsizei ancho, GLsizei alto, GLenum formato, GLenum tipo, const GLvoid *pixels)`
  `ancho, alto`: dimensiones de la imagen en pixels.
  `formato`: indica en qué formato se encuentra el pixmap: `GL_COLOR_INDEX`, `GL_LUMINANCE`, `GL_LUMINANCE_ALPHA`, `GL_RGB`, `GL_RGBA`, `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_STENCIL_INDEX` o `GL_DEPTH_COMPONENT`.
  `tipo`: tipo de datos del array de pixels: `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_BITMAP`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT` o `GL_FLOAT`.
  `pixels`: array de pixels.
  Esta función dibuja en el buffer pixmap guardado en un array de pixels. Este pixmap puede tener diversos formatos (RGBA, escala de grises, valores del buffer de profundidad, etc) y el tipo de sus datos puede ser uno de los especificados en el parámetro `tipo`. El array se dibuja en la posición actual en la pantalla, que puede definirse utilizando la función `glRasterPos`. Si esta posición no es válida, el array no se copia.

- `void glReadPixels (GLint x, GLint y, GLsizei ancho, GLsizei alto, GLenum formato, GLenum tipo, GLvoid *pixels)`
  `x, y`: posición en pixels de la esquina inferior izquierda del rectángulo de pixels que se van a leer.
  `ancho, alto`: dimensiones del rectángulo en pixels.
  `formato`: indica en qué formato se encuentra el pixmap (ver función anterior).
  `tipo`: tipo de datos del array de pixels (ver función anterior).
  `pixels`: puntero al array de pixels.
  Con esta función podemos leer y guardar en memoria un array de pixels de alguno de los buffers. El parámetro `formato` indica el buffer desde el que copiamos, y el tipo de pixels, mientras que el `tipo` indica el tipo de los datos del array.

- `void glCopyPixels (GLint x, GLint y, GLsizei ancho, GLsizei alto, GLenum tipo)`
  `x, y`: posición en pixels de la esquina inferior izquierda del rectángulo de pixels.
  `ancho, alto`: dimensiones del rectángulo en pixels.
  `tipo`: tipo del buffer del que se lee y donde se copia el array de pixels. Puede valer `GL_COLOR`, `GL_STENCIL` o `GL_DEPTH`.
  Esta función copia un área del buffer indicado por el parámetro tipo, en otra área. Los parámetros `x,y` indican la esquina inferior izquierda del rectángulo a leer, mientras que la posición actual en la ventana marca la posición donde va a copiarse. Esta posición puede alterarse utilizando la función `glRasterPos`. Si no es válida, el array no se copia. Esta función podría sustituirse por el uso combinado de `glReadPixels` y `glDrawPixels`.

Sobre las imágenes también es posible realizar operaciones de escalado para ampliarlas o reducirlas. Por defecto, cada pixel de la imagen ocupa un pixel en la pantalla. La función `glPixelZoom` sirve para alterar esta escala de la imagen.

- `void glPixelZoom (GLfloat zoomx, GLfloat zoomy)`
  `zoomx, zoomy`: factor de escala para las imágenes que se dibujen a continuación. Por defecto estos valores valen 1.0. Si, por ejemplo, `zoomx` y `zoomy` valen ambos

2.0, cada pixel de la imagen ocupará 2x2=4 pixels. Se permiten valores no enteros e, incluso, negativos.

También es posible convertir pixels de un formato a otro. Para ello es necesario un conocimiento de los modos de almacenamiento y de transferencia de los pixels que escapa al propósito de este texto. Pueden consultarse estos temas en [Neid93].

TEXTURAS

Una textura es una imagen que se asigna a alguna primitiva, de manera que esta primitiva aparece rellena con la imagen de la textura en lugar de con un color sólido como ocurría en los casos vistos hasta el momento. Aunque podemos asociar texturas a cualquier tipo de primitiva, generalmente su utilidad radica en asociarlas a polígonos. Las texturas sólo funcionan en el modelo de color RGB.

Las texturas permiten añadir realismo a las imágenes, sin necesidad de ampliar el número de polígonos de la escena. Además, la textura que apliquemos sobre un polígono sufre las mismas transformaciones que éste, incluyendo los efectos de la perspectiva, la iluminación, etc.

Los pasos que se siguen para incorporar texturas a las primitivas son los siguientes:

1. Definir la textura.
2. Indicar el modo de aplicación de la textura a cada pixel.
3. Activar la aplicación de texturas.
4. Dibujar la escena, suministrando las coordenadas geométricas y de textura.

DEFINICIÓN DE TEXTURAS

Las texturas pueden ser unidimensionales o bidimiensionales. Una textura unidimensional tiene ancho pero no alto, es decir, son imágenes con un solo pixel de altura. Se utilizan para dar textura a los objetos en forma de bandas, de manera que sólo interesa la variación de la imagen en una dirección.

Una textura bidimensional es una imagen con ancho y alto. Utilizamos las funciones `glTexImage1D` y `glTexImage2D`, para definir texturas unidimensionales y bidimensionales respectivamente. El significado de sus parámetros es el mismo en ambos casos y por eso sólo los veremos una vez. Aunque, en general, nos centraremos en las texturas bidimensionales, todo lo que veamos será aplicable a las de una única dimensión.

- `void glTexImage1D (GLenum valor, GLint nivel, GLint componentes, GLsizei ancho, GLint borde, GLenum formato, GLenum tipo, const GLvoid *pixels)`

- `void glTexImage2D (GLenum valor, GLint nivel, GLint componentes, GLsizei ancho, GLsizei alto, GLint borde, GLenum formato, GLenum tipo, const GLvoid *pixels)`

  `valor`: debe ser GL_TEXTURE_1D o GL_TEXTURE_2D.

  `nivel`: nivel de detalle. Se utiliza para texturas múltiples como veremos más adelante. Para texturas simples debe valer 0.

componentes: número de componentes del color. Puede valer entre 1 y 4.
ancho, alto: dimensiones de la imagen. Ambas dimensiones deben valer 2^n*2b, donde n es un número entero y b el grosor del borde. Las texturas unidimensionales sólo tienen ancho.
borde: grosor del borde. Puede valer 0, 1 o 2.
formato: indica en qué formato se encuentra la imagen: GL_COLOR_INDEX, GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB, GL_RGBA, GL_RED, GL_GREEN, GL_BLUE o GL_ALPHA.
tipo: tipo de datos del array de pixels: GL_BYTE, GL_UNSIGNED_BYTE, GL_BITMAP, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT o GL_FLOAT.
pixels: array de pixels que forman la imagen.

Veremos a continuación con detalle el significado de estos parámetros y como afectan a la imagen. El primer parámetro, valor, se contempla para versiones posteriores de OpenGL. En la actual, sólo existen dos posibles valores y deben ser GL_TEXTURE_1D para texturas unidimensionales o GL_TEXTURE_2D para texturas con dos dimensiones.

El parámetro nivel se utiliza para indicar el número de niveles de resolución de la textura. Cuando la textura es simple debe valer 0. Pero su razón de ser es el uso de texturas múltiples o *mipmaps*. Cuando un objeto se encuentra cercano al observador, su textura puede apreciarse con mucho más detalle del que se ve cuando el objeto se aleja. OpenGL realiza operaciones para adaptar las texturas de los objetos al tamaño que tienen en pantalla. Este proceso puede producir, sin embargo, efectos no deseados: los objetos demasiado cercanos pierden precisión, mientras que para los lejanos no se consigue una textura adecuada. En estos casos pueden utilizarse texturas múltiples o *mipmaps*: se trata de una familia de texturas de diferentes tamaños, de tal manera que OpenGL decide cuál de ellas utilizar dependiendo del tamaño que tenga la primitiva en pantalla. Estas texturas deben tener todas un tamaño que sea potencia de 2, desde la textura más grande utilizada hasta la de tamaño 1x1. Así, si la mayor tiene tamaño 64x16, debemos definir las demás con tamaños 32x8, 16x4, 8x2, 4x1, 2x1 y 1x1. OpenGL elegirá en cada momento el tamaño que mejor se adapte a cada primitiva. En el caso de que no encuentre una de tamaño adecuado, elegirá la más cercana o aplicará algún tipo de interpolación entre tamaños consecutivos, como veremos posteriormente. Para definir texturas múltiples debemos realizar llamadas consecutivas a glTexImage2D con los diferentes valores de ancho y alto de la imagen, en orden decreciente de tamaño. El parámetro nivel será 0 para la textura de mayor tamaño, incrementándose a cada paso conforme disminuye el tamaño de las texturas.

El argumento componentes indica qué componentes del modelo RGBA son seleccionadas para realizar las operaciones de modulación y *blending*, que veremos posteriormente. Puede valer 1 (se selecciona la componente R), 2 (se seleccionan R y G), 3 (R, G y B) o 4 (R, G, B y A).

Los parámetros ancho, alto y borde son las dimensiones de la imagen y del borde. El borde debe valer 0, 1 o 2, mientras que las dimensiones de la imagen deben valer 2^n+2b, donde n es un número entero y b el grosor del borde. El tamaño de la textura se encuentra limitado y depende de la implementación concreta de OpenGL, aunque en la mayoría de sistemas este límite es de 64x64 pixels (66x66 con los bordes). El uso de bordes se justifica cuando el tamaño de la textura no es suficiente para rellenar toda una primitiva. En estos casos hay que recurrir a utilizar un mosaico de texturas. Hay que tener especial cuidado en adaptar adecuadamente el dibujo de las

texturas para que no se noten los bordes (por ejemplo, si se trata una textura de madera, las vetas deben disponerse de tal manera que al colocar dos texturas consecutivamente no se note la discontinuidad). En este proceso también interviene el uso de filtros (que veremos más adelante), que se utilizan para realizar un suavizado entre las dos texturas adyacentes. Para que este suavizado sea correcto, debemos definir un borde para cada textura cuyos pixels deben ser iguales a los pixels adyacentes de la textura.

Los parámetros `formato` y `tipo` tienen el mismo significado que en la función `glDrawPixels` que vimos en un apartado anterior. Por último, `pixels` contiene el array de pixels que forma la imagen.

MODOS DE APLICACIÓN DE LA TEXTURA

La forma en que la textura se aplica a la primitiva depende de varios factores, entre ellos, del filtro elegido, de si se aplica un mosaico o no de la textura y de la operación de aplicación: pegado, modulación o *blending*. En este apartado veremos como afectan todos estos factores y qué funciones utilizar para manejarlos.

Las imágenes utilizadas para las texturas son cuadradas o rectangulares, mientras que las primitivas sobre las que se aplican no tienen porqué serlo. Además, el tamaño de éstas no coincidirá en general con el de aquellas. Por lo tanto, rara vez un pixel de la textura (también llamado texel) se corresponde con un pixel de la escena. En estos casos hay que recurrir a algún filtro que obtenga el valor adecuado para cada pixel de la escena a partir de los texels de la textura. Cuando utilizamos texturas múltiples, es necesario, además, tener en cuenta más de una textura. La función `glTexParameter`, permite, entre otras cosas, fijar los filtros para cada ocasión. Tiene varias versiones, aunque la que se utiliza para definir el filtro es la siguiente:

- `void glTexParameteri (GLenum tipo, GLenum valor, GLint parametro)`
  `tipo`: indica si la textura es unidimensional o bidimensional. Puede valer `GL_TEXTURE_1D` o `GL_TEXTURE_2D`.
  `valor`: indica el parámetro que se está fijando. Puede valer `GL_TEXTURE_MIN_FILTER` o `GL_TEXTURE_MAG_FILTER`.
  `parametro`: es el tipo de filtro. Para `GL_TEXTURE_MAG_FILTER` puede valer `GL_NEAREST` o `GL_LINEAR`. Para `GL_TEXTURE_MIN_FILTER` puede valer `GL_NEAREST`, `GL_LINEAR`, `GL_NEAREST_MIPMAP_NEAREST`, `GL_NEAREST_MIPMAP_LINEAR`, `GL_LINEAR_MIPMAP_NEAREST` o `GL_LINEAR_MIPMAP_LINEAR`.

Cuando hay que aplicar un filtro, por no haber correspondencia uno a uno entre pixels de la escena y texels, pueden darse dos casos: que un pixel se corresponda con varios texels (minificación) o que se corresponda con una parte de un texel (magnificación). Es posible fijar filtros diferentes para cada una de estas dos operaciones, utilizando `glTexParameter` con el argumento `valor` a `GL_TEXTURE_MIN_FILTER` o a `GL_TEXTURE_MAG_FILTER`, respectivamente. Los posibles filtros que pueden definirse son los siguientes:

- `GL_NEAREST`: elige como valor para un pixel el texel con coordenadas más cercanas al centro del pixel. Se utiliza tanto para magnificar como para minificar. Puede producir efectos de *aliasing*.

- GL_LINEAR: elige como valor para un pixel la media del array 2x2 de texels más cercano al centro del pixel. También se utiliza para magnificar y minificar, y produce efectos más suaves que GL_NEAREST.
- GL_NEAREST_MIPMAP_NEAREST: este y los siguientes, se utilizan sólo para minificar cuando se han definido texturas múltiples. En este caso, elige la textura más cercana en tamaño y se comporta como GL_NEAREST.
- GL_LINEAR_MIPMAP_NEAREST: Para texturas múltiples, elige la textura más cercana y actúa como el filtro GL_LINEAR.
- GL_NEAREST_MIPMAP_LINEAR: Para texturas múltiples, elige las dos texturas más cercanas y las interpola, y aplica el filtro GL_NEAREST.
- GL_LINEAR_MIPMAP_LINEAR: Para texturas múltiples, elige las dos texturas más cercanas y las interpola, y aplica el filtro GL_LINEAR.

El último de los filtros es el que obtiene unos mejores resultados, aunque es el más costoso computacionalmente.

Otro aspecto fundamental a la hora de manejar las texturas es conocer cómo se "pegan" las texturas a las primitivas y si sufren alguna operación adicional. Existen tres formas de aplicar las texturas: **pegarlas** (o *decalling*), cuyo efecto es colocar la textura sobre la primitiva y usar simplemente sus colores para rellenarla, **modularlas** (o *modulate*), que permite modular o matizar el color de la primitiva con los colores de la textura, y ***blending***, para difuminar el color de la primitiva con los colores de la textura. La función que fija el modo de operación es glTexEnv, que tiene varias versiones, entre ellas:

- `void glTexEnvi (GLenum tipo, GLenum valor, GLint parametros)`
- `void glTexEnvfv (GLenum tipo, GLenum valor, GLfloat *parametros)`

  `tipo`: debe valer GL_TEXTURE_ENV.

  `valor`: parámetro que estamos definiendo. Puede valer GL_TEXTURE_ENV_MODE o GL_TEXTURE_ENV_COLOR.

  `parametro`: si `valor` es GL_TEXTURE_ENV_MODE, hemos de utilizar la primera versión de la función, y `parámetros` puede valer GL_DECAL (pegar), GL_MODULATE (modular) o GL_BLEND (blending). Si `valor` es GL_TEXTURE_ENV_COLOR, la versión adecuada es la segunda y `parámetros` es un conjunto de valores RGBA.

Los modos de operación se comportan de la siguiente manera:

- Modo GL_DECAL: la textura se aplica directamente sobre la primitiva. Se utiliza para aplicar texturas opacas.
- Modo GL_MODULATE: el color de la primitiva es modulado por la textura. Se utiliza para poder aplicar luces sobre las texturas.
- Modo GL_BLEND: el color de la primitiva es difuminado por la textura. La textura debe estar formada por texels de sólo una o dos componentes (R o RG). Se utiliza para simular transparencias. La operación de blending depende de un color que se especifica con esta misma función con el parámetro GL_TEXTURE_ENV_COLOR. Las opciones de blending las veremos en un apartado posterior.

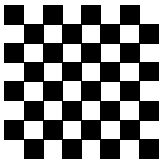
La última cuestión que afecta a los modos de aplicación de las texturas es la posibilidad de expandirlas o repetirlas. Cuando la textura es menor que la primitiva sobre la que debe colocarse, existen dos posibilidades: repetirla en forma de mosaico o expandirla más allá del final de la textura, utilizando su propio borde. La textura puede repetirse en un sentido y expandirse en otro, o realizar la misma operación en

ambas direcciones. La función que utilizaremos es `glTexParameter`, en la versión que vemos a continuación:

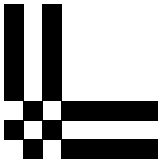
- `void glTexParameteri (GLenum tipo, GLenum valor, GLint parametro)`
  `tipo`: indica si la textura es unidimensional o bidimensional. Puede valer `GL_TEXTURE_1D` o `GL_TEXTURE_2D`.
  `valor`: indica el sentido al que afecta el modo de repetición. Puede valer `GL_TEXTURE_WRAP_S` (coordenada S) o `GL_TEXTURE_WRAP_T` (coordenada T). Las coordenadas de textura las trataremos en el siguiente apartado.
  `parametro`: es el tipo de repetición. Puede valer `GL_REPEAT` (repetición en modo mosaico) o `GL_CLAMP` (repetición en modo expandir).

La repetición `GL_REPEAT` se realiza colocando la textura en modo de mosaico. Los texels del borde se calculan como la media con los del mosaico adyacente. En el modo `GL_CLAMP` los pixels del borde se expanden en toda la primitiva. Veamos un ejemplo ilustrativo de estos modos:

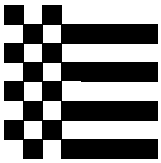
Textura original



Operación `GL_REPEAT` en ambos sentidos



Operación `GL_CLAMP` en ambos sentidos



Operación `GL_REPEAT` en un sentido y `GL_CLAMP` en otro

ACTIVACIÓN DE LAS TEXTURAS

Una vez que las texturas han sido creadas y se ha determinado el modo de aplicación de las mismas, deben ser activadas para que se apliquen a las primitivas. Las funciones de activación y desactivación son `glEnable` y `glDisable`:

- `void glEnable (GLenum valor)`
- `void glDisable (GLenum valor)`
  `valor`: es el elemento que se activa o desactiva. En este caso debe valer `GL_TEXTURE_1D` o `GL_TEXTURE_2D`.

CONSIDERACIONES FINALES

OpenGL incluye muchas otras funcionalidades que no hemos incluido aquí. Entre ellas cabe destacar:

- Fuentes de texto.
- Primitivas para crear cuádricas: esferas, cilindros y discos.
- Efectos visuales como la niebla.
- Curvas y supeficies de tipo NURB.
- Triangulación de polígonos.

Todos estos temas avanzados no pueden abarcarse en un curso como este. Puede encontrarse amplia información en la bibliografía que se aporta al final, especialmente en [Neid93] y [Wrig96].

3. BIBLIOGRAFÍA

LIBROS Y TEXTOS DE INTERÉS

[Kilg96] Mark J. Kilgard.*The OpenGL Utility Toolkit (GLUT). Programming Interface*. 1996. Se puede obtener en la web de GLUT: reality.sgi.com/mjk/#glut

[Neid93] Jackie Neider, Tom Davis, Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, 1993.

[Wrig96] Richard S. Wright JR., Michael Sweet. *OpenGL Superbible*. Waite Group Press, 1996.

DIRECCIONES DE INTERÉS

- Página de OpenGL: www.opengl.org
- Página de Mesa: www.mesa3d.org
- Página de Silicon Graphics: www.sgi.com
- Página de GLUT: reality.sgi.com/mjk/#glut