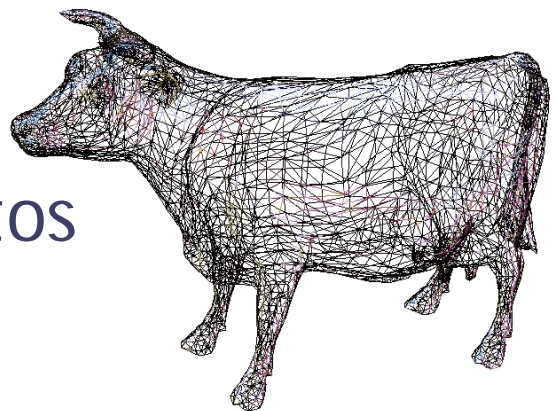


02/09/2016

Geometría y atributos

Seminario ISGI (S2)



R. Vivó

Geometría y atributos

Seminario ISGI (S2)

Este seminario explica cuáles son los elementos de dibujo de que disponemos, qué atributos más comunes tienen y cómo podemos cambiarlos. También se presenta cómo podemos agrupar las órdenes de dibujo en listas para hacer más eficiente la ejecución y cómo configurar el modelo de polígonos basado en listas de vértices.

Objetos 3D predefinidos

La librería GLUT dispone de varios objetos 3D preconstruidos. Todos ellos se dibujan en modo inmediato, es decir, se lanzan todas las órdenes OpenGL de dibujo sin ningún preprocesado previo. Se presentan en dos formatos: formato alámbrico y formato sólido. En el último, se han generado también las normales para su correcta iluminación. Sólo el objeto *teapot* genera, además, coordenadas de textura.

Los objetos se generan centrados en el origen de coordenadas y, en algunos casos, es posible indicar su tamaño o grado de detalle mediante parámetros de entrada a la función.

Los objetos preconstruidos son:

Esfera	<code>glut[Wire Solid]Sphere(radio,rodajas,secciones)</code>
Cubo	<code>glut[Wire Solid]Cube(tamaño)</code>
Toro	<code>glut[Wire Solid]Torus(radint,radext,facet,secciones)</code>
Icosaedro	<code>glut[Wire Solid]Icosahedron()</code>
Octaedro	<code>glut[Wire Solid]Octahedron()</code>
Tetraedro	<code>glut[Wire Solid]Tetrahedron()</code>
Dodecaedro	<code>glut[Wire Solid]Dodecahedron()</code>
Cono	<code>glut[Wire Solid]Cone(radbase,altura,rodajas,secciones)</code>
Tetera	<code>glut[Wire Solid]Teapot(tamaño)</code>



Figura 1. Tetera GLUT

En la sección de código que sigue se muestra como dibujar una tetera alámbrica. Se indica sólo la función `display()`.

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glutWireTeapot(0.5);
    glFlush();
}
```

Para ver con detalle las llamadas a las funciones de dibujo debe consultarse las fuentes [1], [2] y [3].

Ejercicio S2E01: Dibujar una tetera en alámbrico y sólido.

Puntos, Líneas y Polígonos

Lo primero que hay que entender es que el “papel” donde se dibuja es una cuadrícula de rectángulos de color elementales que llamamos píxeles. Las primitivas gráficas de dibujo como por ejemplo los segmentos de recta o líneas son, matemáticamente hablando, de grosor nulo. Lo mismo sucede con los puntos que no tienen dimensión. Sin embargo al dibujarlos es necesario hacer un proceso de conversión

desde el mundo abstracto de las matemáticas al mundo de números enteros de la pantalla de un ordenador. A este proceso se le llama *rasterización* y consiste en hacer un muestreo de la primitiva definida en un sistema de coordenadas R^3 para adaptarla, lo mejor posible, a un sistema N^2 . Por ejemplo, si definimos dos puntos muy juntos es posible que se dibujen en el mismo pixel. Lo mismo puede suceder con dos líneas cuyos extremos respectivos casi coinciden.

Lo segundo y no menos importante es que el sistema de coordenadas que usamos para situar los puntos, las líneas y los polígonos es el sistema de coordenadas del **mundo real**. Este sistema de coordenadas es independiente de los pixeles, por tanto nunca debemos pensar en pixeles sino en coordenadas reales. La adaptación de un sistema de coordenadas a otro se realiza mediante transformaciones entre sistemas de las que se ocupa OpenGL, habiendo fijado nosotros cuáles son el tamaño y la posición de los espacios que deben corresponderse.

Puntos

El conjunto de coordenadas en el espacio 3D que definen la posición de un punto lo denominamos **vértice**. El vértice consta de tres coordenadas x, y y z del mundo real. Si sólo damos la x y la y , entonces se considera la z como 0. Para especificar un vértice en OpenGL usaremos habitualmente:

```
glVertex3f(float x, float y, float z)
```

Hay variantes en la llamada sobre el tipo y número de parámetros pudiendo darse incluso en forma de vector.

Líneas

En realidad usamos segmentos de recta que aquí llamamos líneas. Una línea se define por dos vértices que son su origen y final. OpenGL se encarga de transformar las coordenadas de los vértices 3D en coordenadas del dispositivo (pixeles) y calcula los pixeles que mejor representan el trazado de la línea. Cualquier curva acaba representándose como una polilínea, es decir, una serie de líneas conectadas.

Polígonos

Un polígono se define por una colección de vértices ordenados. El orden es importante pues indica cómo han de conectarse esos vértices para delimitar la superficie del polígono. Visto de otra forma, un polígono es la superficie delimitada por la polilínea cerrada que se obtiene al unir los vértices en el orden dado. Obviamente, el último vértice se une al primero para cerrar la polilínea y es por ello que no hace falta repetir el primer vértice.

OpenGL impone fuertes restricciones al tipo de polígonos que podemos usar. Hemos de seguir estas directivas:

1. El polígono debe ser conexo
2. El polígono debe ser plano
3. El polígono debe ser convexo

La primera restricción hace referencia a que el polígono debe ser de una pieza, es decir, dados dos puntos cualesquiera del polígono debe poderse trazar una curva que los una sin salirse del polígono. En realidad esta restricción está incluida en la tercera pero así remarcamos la imposibilidad de polígonos compuestos por varias piezas.

La segunda restricción se refiere a que todos los vértices que definen el polígono deben estar en el mismo plano, es decir, la superficie del polígono debe ser plana.

La tercera restricción se refiere a que dados dos puntos interiores al polígono deben poderse unir mediante una recta sin salirse del polígono. Por ejemplo, son polígonos no válidos aquellos cuyas aristas se cortan, tienen algún agujero o tienen hendiduras. La figura 2 muestra algunos de estos casos.

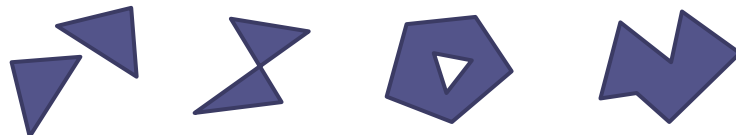


Figura 2. Polígonos no válidos

El uso de polígonos no válidos conduce a resultados inesperados.

Es muy habitual usar triángulos cuando se trabaja con polígonos pues son siempre planos y convexos.

Primitivas

Una vez definido cómo se especifican vértices es necesario saber indicar a OpenGL lo que se quiere dibujar. El método general es hacer llamadas a la función `glVertex()` dentro de un bloque de dibujo delimitado por las instrucciones `glBegin()` y `glEnd()`. La función de inicio del dibujo admite un parámetro -constante- que indica el tipo de primitiva que queremos dibujar. Así, por ejemplo, si queremos que los vértices representen puntos debemos pasar como parámetro `GL_POINTS`. A continuación se muestra en la figura 3 los tipos de primitivas según el parámetro usado.

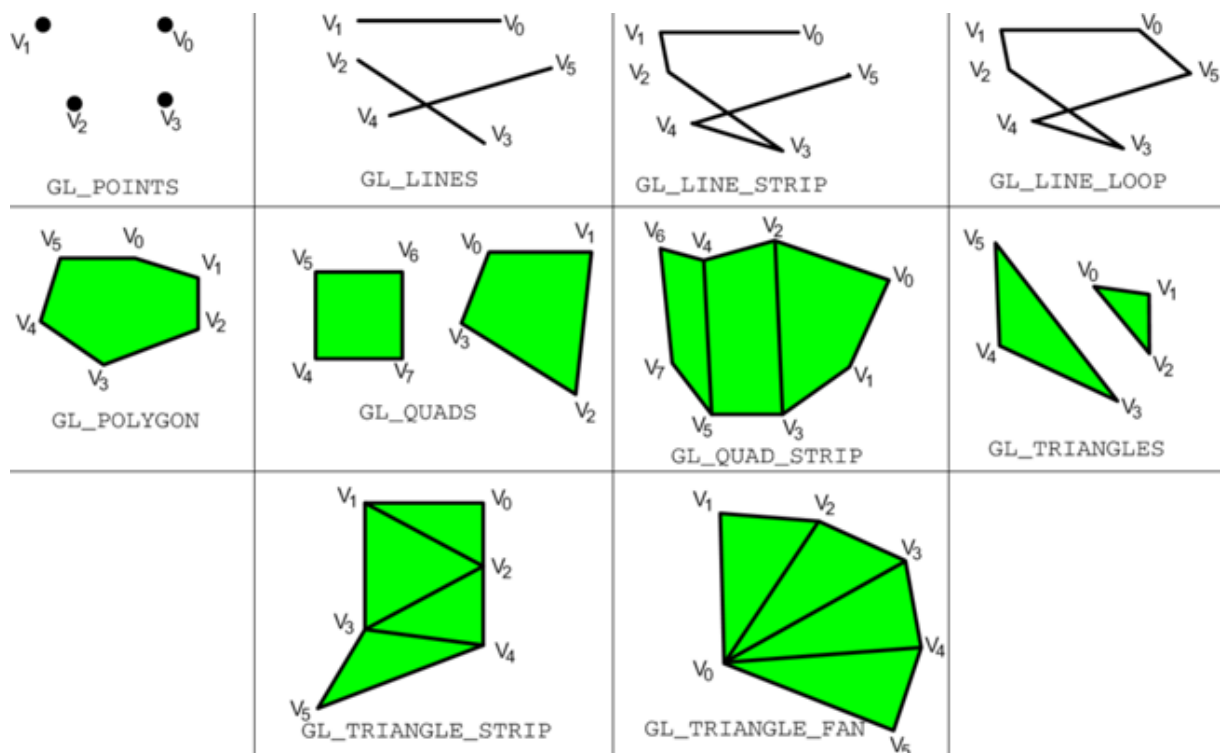


Figura 3. Tipos de primitivas en OpenGL

El código que sigue muestra cómo dibujar un polígono relleno en color blanco (figura 4). El código se inserta en la función `display()`, tal como se muestra. Se observa que la función `glEnd()` no lleva parámetros.

Se ha usado la indentación de las instrucciones `glVertex()` para aclarar que están dentro del bloque de dibujo. Es importante tener en cuenta que **no se pueden anidar** bloques de dibujo.

```
glClear(GL_COLOR_BUFFER_BIT);
glBegin(GL_POLYGON);
    glVertex2f(-1.0,-1.0);
    glVertex2f(0.0,-1.0);
    glVertex2f(1.0,0.0);
    glVertex2f(0.0,1.0);
    glVertex2f(-1.0,1.0);
glEnd();
glFlush();
```

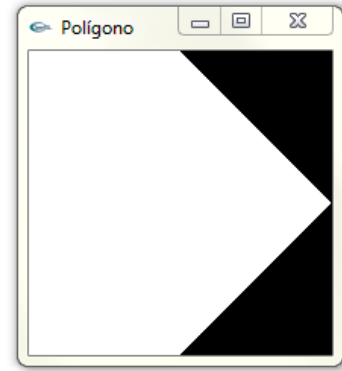


Figura 4. Polígono resultante

Ejercicio S2E02: Dibujar un pentágono con `GL_POLYGON`, `GL_TRIANGLE_STRIP` y `GL_TRIANGLE_FAN`

Atributos del vértice

El vértice es, en 3D, la unidad básica con la que dibujamos, de igual manera que en la pantalla 2D la unidad básica es el píxel. Los atributos o características que podemos asociar a un vértice además de su posición son principalmente:

1. El color **RGB**
2. El vector que llamamos vector **normal**
3. Las coordenadas en el espacio de la **textura**

Los dos últimos atributos los dejaremos para seminarios más avanzados. Ahora sólo nos ocuparemos del color aunque se aplican las mismas reglas para cualquier atributo.

El color de un vértice es el que usará para dibujarlo en pantalla si decidimos que se represente como un punto. Si forma parte de una entidad de dimensión mayor –línea o polígono- el color del vértice se usará en la interpolación automática que lleva a cabo OpenGL para colorear la primitiva correspondiente. Así por ejemplo, si asignamos tres colores diferentes a los vértices de un triángulo sólido, éste se dibujará coloreando los píxeles de su interior a un color mezcla de los tres según esté el píxel más o menos próximo a cada vértice.

La función que fija el color es `glColor()` que admite variedad en el tipo y número de parámetros. La más común es `glColor3f(float rojo, float verde, float azul)`. Los parámetros *rojo*, *verde* y *azul* de la llamada deben ser números reales dentro del intervalo [0,1]. El color se calcula sopesando la cantidad de cada color primario como si fuera una mezcla de pinturas. Así, por ejemplo, el color blanco corresponde al (1.0,1.0,1.0), el amarillo al (1.0,1.0,0.0) y el negro al (0.0,0.0,0.0).

Es muy importante tener en cuenta que OpenGL funciona como una máquina de estados, es decir, una vez fijado el color todos los vértices tendrán ese color hasta que se vuelva a cambiar con otra llamada a `glColor()`. La función puede usarse dentro o fuera del bloque de dibujo aunque si se quiere dotar a cada vértice de un color diferente es obvio que debe preceder a `glVertex()` dentro del bloque de dibujo.

Ejercicio S2E03: Dibujar el pentágono anterior con vértices rojo, verde, azul, amarillo y cyan.

Atributos de la primitiva

Por defecto los puntos se dibujan como un píxel, las líneas continuas de un píxel de grueso y los polígonos rellenos según el color de sus vértices. A continuación vemos como cambiar este comportamiento.

Puntos

Para cambiar el tamaño de un punto se usa:

```
glPointSize(GLfloat tamaño_en_pixels)
```

donde el parámetro indica el número de pixeles que tendrá el lado del cuadrado que representa al punto. Números fraccionarios sólo son útiles cuando se habilita el antialiasing.

Líneas

Para cambiar el ancho de las líneas se usa:

```
glLineWidth(GLfloat grosor)
```

donde grosor indica el número de pixeles que tendrá de ancho la línea.

Para cambiar el patrón de dibujo (continuo, guiones, raya-punto, etc.) se usa:

```
glLineStipple(GLint factor, GLushort patron)
```

donde el parámetro `patron` indica una secuencia de bits que indican cómo dibujar la línea. En el patrón los 1's indican dibujo y los 0's su ausencia, empezando por el menos significativo. Por ejemplo, si el patrón es 0x3F07 (0011111100000111 en binario), la línea comienza por una raya de tres pixeles de largo, cinco pixeles en blanco, raya de seis y dos en blanco finales. Con el parámetro `factor` multiplicamos la longitud de cada sección del patrón por el valor dado.

Por defecto la capacidad de dibujar líneas según un patrón no está activada por lo que, si queremos hacer uso de ella, es necesario ordenar:

```
glEnable(GL_LINE_STIPPLE)
```

Esta es la manera que tiene OpenGL de activar (o desactivar con `glDisable`) ciertas capacidades o estados.

Polígonos

Los polígonos admiten mayor variedad de formas de dibujo. La principal es determinar si el polígono se quiere relleno (por defecto), sólo dibujando las aristas o sólo los vértices y, dado que en 3D tiene dos caras, a cuál nos estamos refiriendo. Las caras de un polígono en OpenGL se las nombra como `GL_FRONT` y `GL_BACK`. Cuando vemos que los vértices del polígono están ordenados en sentido contrario a las agujas del reloj respecto al orden en que fueron dados al construirlo, la cara es la frontal. Si sucede al revés, es la trasera.

La orden que varía la forma en la que dibuja un polígono es:

```
glPolygonMode(constante, modo)
```

donde `constante` puede ser `GL_FRONT`, `GL_BACK` o `GL_FRONT_AND_BACK` y `modo` puede ser `GL_POINT`, `GL_LINE` o `GL_FILL` (este último por defecto).

También se puede invertir la definición de cara delantera y trasera sin más que usar la orden:

```
glFrontFace(modos)
```

donde modo puede ser GL_CCW (el de defecto) o GL_CW (el inverso).

Ejercicio S2E04: Dibujar el pentágono anterior relleno de color rojo, con sus aristas de color azul y los vértices como puntos gruesos de color verde.

Grupos de atributos

En ocasiones es conveniente guardar los valores actuales de la máquina de estados de OpenGL para recuperarlos después de haberlos cambiado por otros. Así, si estamos dentro de una función que dibuja, pongamos por caso, un polígono de color rojo, primero guardaremos el color actual de dibujo, sea cual sea, segundo fijaremos el color de dibujo a rojo y dibujaremos y, por último, antes de abandonar la función restauraremos el color guardado.

Para guardar el estado de un conjunto de atributos se usa la orden:

```
glPushAttrib(mascara)
```

donde mascara es la unión de conjuntos de atributos a guardar. Por ejemplo si la máscara es GL_POINT_BIT|GL_LINE_BIT|GL_POLYGON_BIT se salvarán los atributos sobre puntos, líneas y polígonos vistos antes. Otro conjunto interesante es el GL_CURRENT_BIT que se refiere a los valores actuales del color, las normales, etc., que afectan a los vértices.

Para restaurar los valores simplemente se ordena:

```
glPopAttrib()
```

Esta función restaura los valores guardados con el último glPushAttrib(), pues en realidad se usa una pila de almacén.

Listas de dibujo

Algunas órdenes de OpenGL pueden almacenarse en una lista para ser ejecutadas luego. Cuando esto sucede hablamos de “**modo retenido**” de dibujo. Si por el contrario las órdenes son ejecutadas al instante hablamos de “**modo inmediato**” de dibujo. Ambos modos de trabajo pueden mezclarse en una aplicación.

OpenGL proporciona una manera de almacenar órdenes: las listas de dibujo (*Display Lists*). La principal ventaja de usar listas de órdenes está en que pueden usarse múltiples veces para hacer lo mismo, por ejemplo dibujar la misma geometría o fijar el mismo valor a los atributos. El uso de listas de dibujo se entiende mejor cuando se usan transformaciones -traslaciones, escalados, etc.- pues permiten dibujar la misma geometría en distintas posiciones y tamaños.

Para el uso de listas se siguen tres pasos:

1. Obtener un identificador de la lista. Se consigue con la orden:

```
GLuint glGenLists(numero)
```

que devuelve un entero como identificador de la lista si numero es 1, o del comienzo de identificadores consecutivos si numero es mayor que 1.

2. Construir la lista. Para ello se sigue la siguiente secuencia de órdenes:

```
glNewList(identificador, GL_COMPILE);
  lista de órdenes
glEndList();
```

donde indicamos la lista de órdenes que quedaran almacenadas -y compiladas- en la lista de tal identificador.

3. Ejecutar las órdenes de la lista. Para ello usaremos:

```
glCallList(identificador)
```

que lanzará las órdenes almacenadas en la lista.

Hay que tener en cuenta que las listas almacenan valores estáticos que no pueden ser modificados una vez creada, es decir, las variables tomaran el valor que tenían en el momento de construcción de la lista aunque después se modifiquen. También es interesante observar que dentro de la lista de órdenes pueden haber `glCallList()` lo que permite anidar listas en su construcción para formar una jerarquía de listas.

Ejercicio S2E05: Construir el pentágono anterior como una lista de dibujo al principio de la aplicación.

Listas de vértices

Entre los diferentes modelos de representar una malla de polígonos el más común es el de lista de vértices. En este modelo se describen las coordenadas de los vértices en una tabla por un lado y, en otra tabla, la forma de conectarlos para formar polígonos. La ventaja de este modelo es que no se necesita repetir las coordenadas de un vértice compartido por varios polígonos.

OpenGL proporciona un método para crear listas -arrays- de información asociada a los vértices. Así, pueden ponerse en un *array* todas las coordenadas de los vértices, las normales en otro, los colores en otro, etc. Al final hay que dar un vector con los índices ordenados de los anteriores de manera que siguiendo ese orden se construyan bien los polígonos.

En el tetraedro de la figura 4, los *arrays* de coordenadas el vector de índices (sentido antihorario) son:

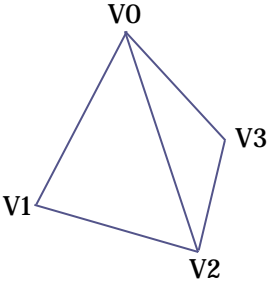
Coordenadas	Índices (triángulos)	
[0]: x0,y0,z0	[0]: 0	
[1]: x1,y1,z1	[1]: 1	
[2]: x2,y2,z2	[2]: 2	
[3]: x3,y3,z3	[3]: 0	
	[4]: 2	
	[5]: 3	
	[6]: 0	
	[7]: 3	
	[8]: 1	
	[9]: 1	
	[10]: 3	
	[11]: 2	

Figura 4. Tetraedro

De la misma manera se podrían dar en otro *array* los colores por vértice por ejemplo.

En OpenGL esta misma organización se construye en tres pasos:

1. Habilitar el uso de *arrays* y el espacio para ello. Usaremos la orden:


```
glEnableClientState(nombre_array)
```

donde `nombre_array` puede ser `GL_VERTEX_ARRAY` para coordenadas, `GL_COLOR_ARRAY` para colores, `GL_NORMAL_ARRAY` para normales, entre otros. Para deshabilitar el `array` y dejar de usarlo se manda la orden contraria:

```
glDisableClientState(nombre_array)
```

2. Llenar los `arrays` con datos. Para ello primero definimos un vector del tipo y la dimensión requerida para cada grupo de datos y lo llenamos. Por ejemplo, si nos referimos a las coordenadas de los vértices que son de tipo `GLfloat` y hay tres por vértice necesitaremos un vector de 12 reales. Después pasaremos el puntero al vector como entrada al `array` de vértices:

```
GLfloat coordenadas[12]={x0,y0,z0,x1,y1,z1,x2,y2,z2,x3,y3,z3};  
glVertexPointer(3, GL_FLOAT, 0, coordenadas);
```

donde el primer parámetro indica las coordenadas por vértice, el segundo su tipo, el tercero la separación entre dos vértices consecutivos (*byte offset*) y el cuarto el puntero al vector de coordenadas.

Para formar la ordenación según el vector de índices declaramos:

```
GLuint indices[12]={0,1,2,0,2,3,0,3,1,1,3,2};
```

Después usaremos este vector para decir cómo se han de conectar los vértices.

3. Ejecutar las órdenes de dibujo. Ahora hay que indicar qué tipo de primitiva vamos a construir con la geometría y cómo es la ordenación. La mejor opción es usar:

```
glDrawElements(GL_TRIANGLES, 12, GL_UNSIGNED_INT, indices)
```

donde el primer parámetro es el tipo de primitiva a construir, el segundo el número total de índices que vamos a usar (llamadas a `glVertex`), el tercero el tipo de los índices y el cuarto es el vector de índices. **No hay que usar `glDrawElements` dentro de un `glBegin()/glEnd()`.**

Cuando ya no se necesite mantener el puntero al `array` de vértices se debe deshabilitar como se dice más arriba.

Ejercicio S2E06: Construir un octaedro con los vértices en (1,0,0), (0,1,0), (0,0,1), (-1,0,0), (0,-1,0), (0,0,-1) donde cada vértice sea de un color diferente. Usar Listas de Vértices.

Referencias

[1] «GLUT de Nate Robins.» [En línea]. Available: <http://user.xmission.com/~nate/glut.html>.

[2] J. Lluch, «Video de polimedia sobre GLUT.» [En línea]. Available: <https://polimedia.upv.es/visor/?id=26a97a02-a9d7-a549-b733-339ffd024609>.

[3] D. S. e. al., "Appendix D," in *OpenGL Programming Guide, Sixth Edition, Version 2.1*, Addison Wesley, 2008.

Ejercicio S2E01: Dibujar una tetera en alámbrico y sólido.

```

/*****
ISGI::Tetera GLUT
Roberto Vivo', 2013 (v1.0)

Dibujo basico de una tetera GLUT en OpenGL

Dependencias:
+GLUT
*****/
#define PROYECTO "ISGI::S2E01::Tetera GLUT"

#include <iostream>                                     // Biblioteca de entrada salida
#include <gl\freeglut.h>                                // Biblioteca grafica

void display()
// Funcion de atencion al dibujo
{
    glClear(GL_COLOR_BUFFER_BIT);                      // Borra la pantalla
    //glutWireTeapot(0.5);                             // Dibujo de una tetera en alambrico
    glutSolidTeapot(0.5);                             // Dibujo de una tetera solida
    glFlush();                                          // Finaliza el dibujo
}

void reshape(GLint w, GLint h)
// Funcion de atencion al redimensionamiento
{
}

void main(int argc, char** argv)
// Programa principal
{
    glutInit(&argc, argv);                             // Inicializacion de GLUT
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);          // Alta de buffers a usar
    glutInitWindowSize(400,400);                       // Tamanyo inicial de la ventana
    glutCreateWindow(PROYECTO);                         // Creacion de la ventana con su titulo
    std::cout << PROYECTO << " running" << std::endl; // Mensaje por consola
    glutDisplayFunc(display);                          // Alta de la funcion de atencion a display
    glutReshapeFunc(reshape);                          // Alta de la funcion de atencion a reshape
    glutMainLoop();                                    // Puesta en marcha del programa
}

```

Ejercicio S2E02: Dibujar un pentágono con GL_POLYGON, GL_TRIANGLE_STRIP y GL_TRIANGLE_FAN

```

/*****
ISGI::Pentagono
Roberto Vivo', 2013 (v1.0)

Dibujo de un pentagono de diferentes maneras

Dependencias:
+GLUT
*****/
#define PROYECTO "ISGI::S2E02::Pentagono"

#include <iostream> // Biblioteca de entrada salida
#include <cmath> // Biblioteca matemática de C
#include <gl\freeglut.h> // Biblioteca grafica

void display()
// Funcion de atencion al dibujo
{
    glClear(GL_COLOR_BUFFER_BIT); // Borra la pantalla

    // Dibujo como POLYGON
    glBegin(GL_POLYGON);
        glVertex3f(0.5*cos(0*2*3.1415926/5), 0.5*sin(0*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(1*2*3.1415926/5), 0.5*sin(1*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(2*2*3.1415926/5), 0.5*sin(2*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(3*2*3.1415926/5), 0.5*sin(3*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(4*2*3.1415926/5), 0.5*sin(4*2*3.1415926/5), 0.0);
    glEnd();

    // Dibujo como TRIANGLE_STRIP
    /*
    glBegin(GL_TRIANGLE_STRIP);
        glVertex3f(0.5*cos(0*2*3.1415926/5), 0.5*sin(0*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(1*2*3.1415926/5), 0.5*sin(1*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(4*2*3.1415926/5), 0.5*sin(4*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(2*2*3.1415926/5), 0.5*sin(2*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(3*2*3.1415926/5), 0.5*sin(3*2*3.1415926/5), 0.0);
    glEnd();
    */

    // Dibujo como TRIANGLE_FAN
    /*
    glBegin(GL_TRIANGLE_FAN);
        glVertex3f(0.5*cos(0*2*3.1415926/5), 0.5*sin(0*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(1*2*3.1415926/5), 0.5*sin(1*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(2*2*3.1415926/5), 0.5*sin(2*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(3*2*3.1415926/5), 0.5*sin(3*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(4*2*3.1415926/5), 0.5*sin(4*2*3.1415926/5), 0.0);
    glEnd();
    */

    glFlush(); // Finaliza el dibujo
}

```

Ejercicio S2E03: Dibujar el pentágono anterior con vértices rojo, verde, azul, amarillo y cyan.

```
/******
ISGI::Pentagono coloreado
Roberto Vivo', 2013 (v1.0)

Dibujo de un pentagono con vertices coloreados

Dependencias:
+GLUT
*****/
#define PROYECTO "ISGI::S2E03::Pentagono Colores"

#include <iostream> // Biblioteca de entrada salida
#include <cmath> // Biblioteca matemática de C
#include <gl\freeglut.h> // Biblioteca grafica

void display()
// Funcion de atencion al dibujo
{
    glClear(GL_COLOR_BUFFER_BIT); // Borra la pantalla

    // Dibujo como POLYGON con los vertices coloreados
    glBegin(GL_POLYGON);
        glColor3f(1.0,0.0,0.0);
        glVertex3f(0.5*cos(0*2*3.1415926/5), 0.5*sin(0*2*3.1415926/5), 0.0);
        glColor3f(0.0,1.0,0.0);
        glVertex3f(0.5*cos(1*2*3.1415926/5), 0.5*sin(1*2*3.1415926/5), 0.0);
        glColor3f(0.0,0.0,1.0);
        glVertex3f(0.5*cos(2*2*3.1415926/5), 0.5*sin(2*2*3.1415926/5), 0.0);
        glColor3f(1.0,1.0,0.0);
        glVertex3f(0.5*cos(3*2*3.1415926/5), 0.5*sin(3*2*3.1415926/5), 0.0);
        glColor3f(0.0,1.0,1.0);
        glVertex3f(0.5*cos(4*2*3.1415926/5), 0.5*sin(4*2*3.1415926/5), 0.0);
    glEnd();

    glFlush(); // Finaliza el dibujo
}
```

Ejercicio S2E04: Dibujar el pentágono anterior relleno de color rojo, con sus aristas de color azul y los vértices como puntos gruesos de color verde.

```

/*****
ISGI::Pentagono con contorno
Roberto Vivo', 2013 (v1.0)

Dibujo de un pentagono con vertices y aristas diferenciados

Dependencias:
+GLUT
*****/
#define PROYECTO "ISGI::S2E04::Pentagono Contorno"

#include <iostream> // Biblioteca de entrada salida
#include <cmath> // Biblioteca matemática de C
#include <gl\freeglut.h> // Biblioteca grafica

void display()
// Funcion de atencion al dibujo
{
    glClearColor(GL_COLOR_BUFFER_BIT); // Borra la pantalla

    // Dibujo como POLYGON relleno
    glColor3f(1.0,0.0,0.0);
    glBegin(GL_POLYGON);
        glVertex3f(0.5*cos(0*2*3.1415926/5), 0.5*sin(0*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(1*2*3.1415926/5), 0.5*sin(1*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(2*2*3.1415926/5), 0.5*sin(2*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(3*2*3.1415926/5), 0.5*sin(3*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(4*2*3.1415926/5), 0.5*sin(4*2*3.1415926/5), 0.0);
    glEnd();

    // Dibujo como POLYGON solo aristas
    glPolygonMode(GL_FRONT, GL_LINE);
    glLineWidth(2.0);
    glColor3f(0.0,0.0,1.0);
    glBegin(GL_POLYGON);
        glVertex3f(0.5*cos(0*2*3.1415926/5), 0.5*sin(0*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(1*2*3.1415926/5), 0.5*sin(1*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(2*2*3.1415926/5), 0.5*sin(2*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(3*2*3.1415926/5), 0.5*sin(3*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(4*2*3.1415926/5), 0.5*sin(4*2*3.1415926/5), 0.0);
    glEnd();

    // Dibujo como POLYGON solo vertices
    glPolygonMode(GL_FRONT, GL_POINT);
    glPointSize(10.0);
    glColor3f(0.0,1.0,0.0);
    glBegin(GL_POLYGON);
        glVertex3f(0.5*cos(0*2*3.1415926/5), 0.5*sin(0*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(1*2*3.1415926/5), 0.5*sin(1*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(2*2*3.1415926/5), 0.5*sin(2*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(3*2*3.1415926/5), 0.5*sin(3*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(4*2*3.1415926/5), 0.5*sin(4*2*3.1415926/5), 0.0);
    glEnd();

    glFlush(); // Finaliza el dibujo
}

```

Ejercicio S2E05: Construir el pentágono anterior como una lista de dibujo al principio de la aplicación.

```

/*****
ISGI::Pentagono como display list
Roberto Vivo', 2013 (v1.0)

Dibujo de un pentagono con vertices y aristas diferenciados
como display list

Dependencias:
+GLUT
*****/
#define PROYECTO "ISGI::S2E05::Pentagono Display List"

#include <iostream> // Biblioteca de entrada salida
#include <cmath> // Biblioteca matemática de C
#include <gl\freeglut.h> // Biblioteca grafica

static GLuint pentagono; // Identificador del objeto

void init()
// Funcion de inicializacion propia
{
    pentagono= glGenLists(1); // Obtiene el identificador de la lista

    glNewList(pentagono, GL_COMPILE); // Abre la lista

    // Dibuja el pentagono en la lista
    glBegin(GL_POLYGON);
        glVertex3f(0.5*cos(0*2*3.1415926/5), 0.5*sin(0*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(1*2*3.1415926/5), 0.5*sin(1*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(2*2*3.1415926/5), 0.5*sin(2*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(3*2*3.1415926/5), 0.5*sin(3*2*3.1415926/5), 0.0);
        glVertex3f(0.5*cos(4*2*3.1415926/5), 0.5*sin(4*2*3.1415926/5), 0.0);
    glEnd();

    glEndList(); // Cierra la lista
}

void display()
// Funcion de atencion al dibujo
{
    glClear(GL_COLOR_BUFFER_BIT); // Borra la pantalla

    // Dibujo como POLYGON relleno
    glColor3f(1.0,0.0,0.0);
    glCallList(pentagono);

    // Dibujo como POLYGON solo aristas
    glPolygonMode(GL_FRONT, GL_LINE);
    glLineWidth(2.0);
    glColor3f(0.0,0.0,1.0);
    glCallList(pentagono);

    // Dibujo como POLYGON solo vertices
    glPolygonMode(GL_FRONT, GL_POINT);
    glPointSize(10.0);
    glColor3f(0.0,1.0,0.0);
    glCallList(pentagono);

    glFlush(); // Finaliza el dibujo
}

void reshape(GLint w, GLint h)
// Funcion de atencion al redimensionamiento
{
}

void main(int argc, char** argv)
// Programa principal
{
    glutInit(&argc, argv); // Inicializacion de GLUT

```

```
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);    // Alta de buffers a usar
glutInitWindowSize(400,400);                 // Tamanyo inicial de la ventana
glutCreateWindow(PROYECTO);                   // Creacion de la ventana con su titulo
init();                                       // Inicializacion propia. IMPORTANTE SITUAR AQUI
std::cout << PROYECTO << " running" << std::endl; // Mensaje por consola
glutDisplayFunc(display);                     // Alta de la funcion de atencion a display
glutReshapeFunc(reshape);                     // Alta de la funcion de atencion a reshape
glutMainLoop();                              // Puesta en marcha del programa
}
```

Ejercicio S2E06: Construir un octaedro con los vértices en $(1,0,0), (0,1,0), (0,0,1), (-1,0,0), (0,-1,0), (0,0,-1)$ donde cada vértice sea de un color diferente. Usar Listas de Vértices.

```

/*****
ISGI::Octaedro como Vertex Array
Roberto Vivo', 2013 (v1.0)

Dibujo de un octaedro con vertices coloreados usando
arrays de vertices, colores e indices

Dependencias:
+GLUT
*****/
#define PROYECTO "ISGI::S2E06::Octaedro Vertex Array"

#include <iostream>                // Biblioteca de entrada salida
#include <cmath>                   // Biblioteca matemática de C
#include <gl\glut.h>               // Biblioteca grafica

// Vector de coordenadas 6x3 de los vertices del octaedro
static const GLfloat vertices[18]={1,0,0, 0,1,0, 0,0,1, -1,0,0, 0,-1,0, 0,0,-1};

// Vector de colores 6x3 de los vertices del octaedro
static const GLfloat colores[18]={1,0,0, 0,1,0, 0,0,1, 1,1,0, 0,1,1, 1,0,1};

// Vector de indices de los triángulos (antihorario) que forman el octaedro 8x3
static const GLuint indices[24]={1,2,0, 1,0,5, 1,5,3, 1,3,2, 4,2,0, 4,0,5, 4,5,3, 4,3,2};

void init()
// Funcion de inicializacion propia
{
    glEnableClientState(GL_VERTEX_ARRAY);        // Activa el uso del array de vertices
    glVertexPointer(3,GL_FLOAT,0,vertices);      // Carga el array de vertices
}

void display()
// Funcion de atencion al dibujo
{
    glClear(GL_COLOR_BUFFER_BIT);                // Borra la pantalla

    glEnableClientState(GL_COLOR_ARRAY);         // Activa el uso del array de colores
    glColorPointer(3,GL_FLOAT,0,colores);        // Carga el array de colores
    glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);    // Las caras del poligono rellenas

    // Dibujo del octaedro como 8 triangulos coloreados
    glDrawElements(GL_TRIANGLES,24,GL_UNSIGNED_INT,indices);

    glDisableClientState(GL_COLOR_ARRAY);        // Desactiva el array de colores
    glColor3f(1,1,1);                            // Fija el color a blanco
    glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);    // Las caras del poligono en alambrico

    // Dibujo del octaedro como 8 triangulos en alambrico
    glDrawElements(GL_TRIANGLES,24,GL_UNSIGNED_INT,indices);

    glFlush();                                    // Finaliza el dibujo
}

```