Ingeniería Informática
Informática Gráfica (Curso 2006/2007)

Dibujar en 3D líneas, círculos y polígonos.

Autores:

Javier Holguera Blanco
Carlos Muñoz Martín
Antonio Lillo Sanz

ÍNDICE DE CONTENIDOS

ÍNDICE DE IMÁGENES

1. INTRODUCCIÓN

OpenGL provee al programador de un interfaz de acceso al hardware gráfico. Es una librería poderosa y de renderización a bajo nivel, disponible en la mayoría de plataformas, con un amplio soporte hardware. Está diseñada para su uso en cualquier aplicación gráfica, desde juegos a modelado o CAD. Muchos juegos, como por ejemplo Doom 3 de ID Software, utilizan OpenGL como su motor principal de renderizado de gráficos. En la siguiente figura se puede apreciar un esquema más elaborado de lo que es OpenGL y su arquitectura.

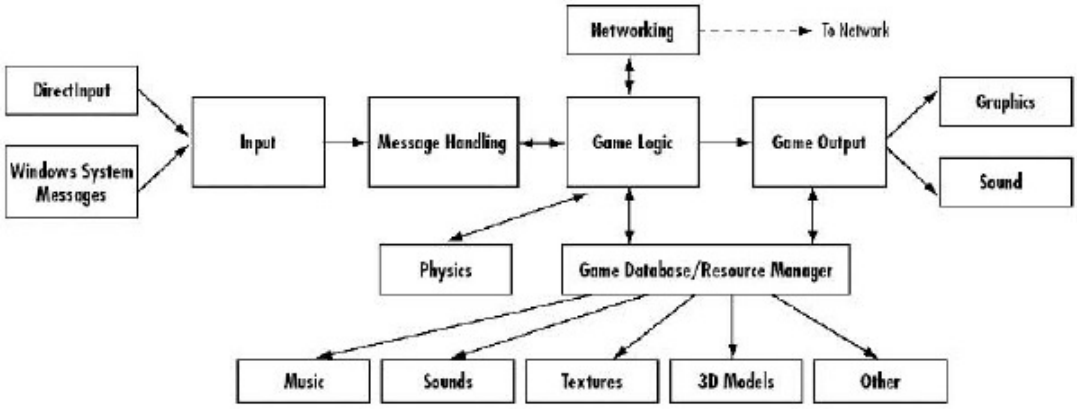


Ilustración 1: Arquitectura OpenGL

A la hora de representar gráficamente elementos mediante OpenGL será necesario hacer uso de las primitivas, las funciones básicas de OpenGL. Combinando estas primitivas podemos llegar a la construcción de figuras complejas. Es vital un conocimiento amplio de estas funciones para la realización de proyectos de mayor envergadura.

Sin embargo, en entornos profesionales como puede ser el desarrollo de videojuegos no se utilizan directamente estas primitivas. En su lugar, se prefiere hacer uso de herramientas gráficas que aceleran el proceso de implementación usando como primitivas figuras más complejas.

2. SISTEMAS DE COORDENADAS

En OpenGL, cuando se crea una ventana sobre la que dibujar, se debe especificar el sistema de coordenadas que se desea utilizar y cómo se mapean las coordenadas especificadas a los píxeles de la pantalla física.

Uno de los sistemas de coordenadas más habituales es el sistema cartesiano, donde se define un origen del sistema en el punto x=0, y=0. La coordenada x será una medida de la posición en sentido horizontal mientras que la coordenada y será una medida en el sentido vertical. Sin embargo, en OpenGL antes de empezar a dibujar sobre una ventana, es necesario establecer la traslación entre este sistema y los píxeles que conforman dicha ventana en la pantalla física. Esto se hace definiendo la región del espacio cartesiano ocupada por dicha ventana, lo que se denomina *Clipping area*. Se corresponderá con el mínimo y máximo valor para las coordenadas x e y. La Ilustración 2 muestra dos *clipping areas*.

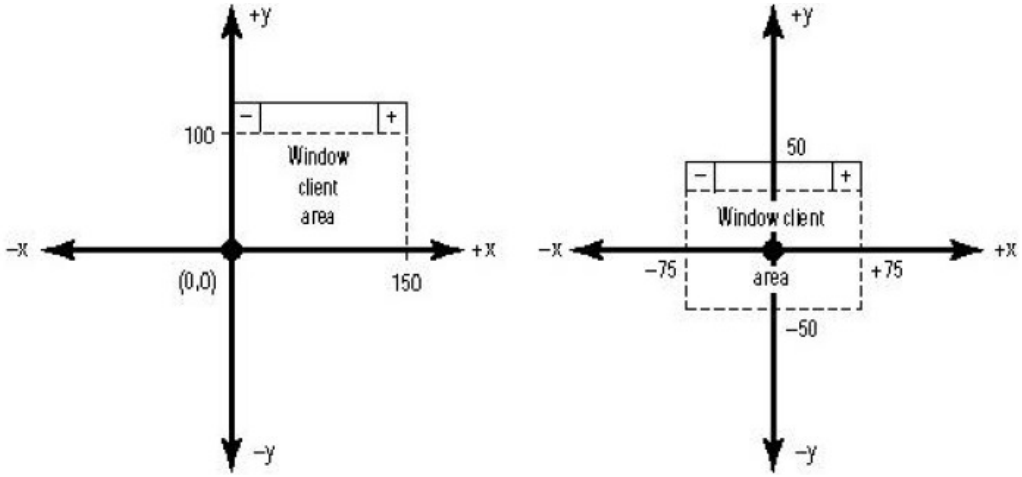


Ilustración 2: Clipping areas

Tanto en 2D como en 3D, cuando se dibuja un objeto realmente se estará componiendo de pequeñas formas conocidas como primitivas. Las primitivas son superficies en dos dimensiones como puntos, líneas y polígonos que se juntan en espacios de 3D para crear figuras de 3D. Por ejemplo, el cubo que se observa en la Ilustración 3 se compone de 6 cuadrados de 2 dimensiones, cada uno colocado en una cara diferente. Cada esquina de un cuadrado (o de cualquier primitiva) se conoce como *vertex*. Estos vértices se especifica que ocupen una determinada coordenada en espacio 2D o 3D.

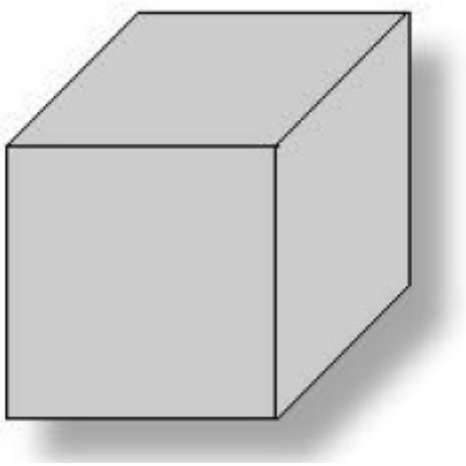


Ilustración 3: Vértices (*vertex*) en un cubo

Por último, para lograr un sistema de coordenadas en tres dimensiones es necesario extender el actual de dos dimensiones con una tercera y añadir una componente de profundidad. El eje z es perpendicular a los ejes x e y. La Ilustración 4 muestra un cubo que serviría de lienzo sobre el que dibujar las diferentes funciones de OpenGL.

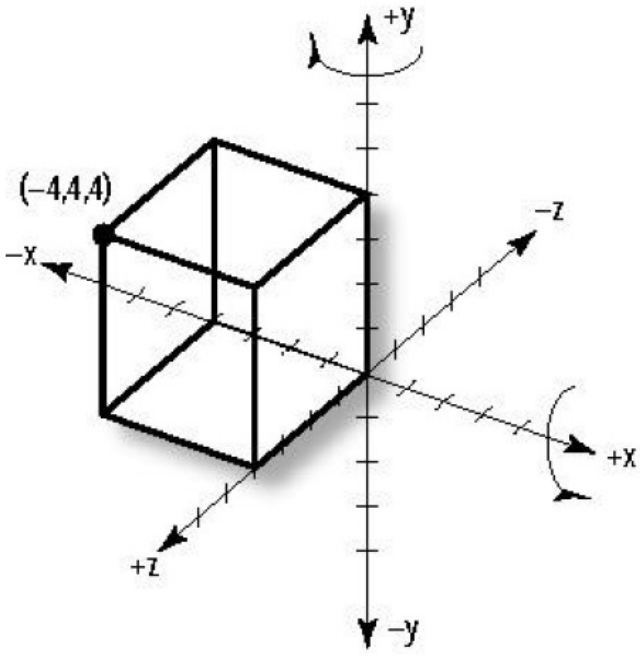


Ilustración 4: Cubo a modo de lienzo sobre el que dibujar

Para establecer este volumen se usa `GLOrtho(- x, +x, –y, +y, - z, +z)` donde (–n, +n) representa los valores mínimo y máximo visualizables en cada dimensión. Para que estos valores puedan utilizarse hay que llamar anteriormente a la función `glLoadIdentity()` para reiniciar el sistema de coordenadas a la unidad antes de que se ejecute cualquier manipulación de matrices. En principio el centro de coordenadas se localiza en el centro de la vista.

Una vez que se tiene definido el sistema de coordenadas y su posición, es necesario definir la vista de la ventana, es decir, la zona que servirá como área de dibujo. Suponiendo que `xIni` e `yIni` correspondieran a la esquina superior izquierda de la ventana, y ancho y alto las correspondientes variables para dichas magnitudes, ésta sería la función a utilizar:

```
glViewport(xIni, yIni, ancho, alto)
```

Por el contrario, si se quiere que la vista tenga la mitad de ancho y alto de la ventana y que esté centrada, la llamada sería:

```
glViewport(anchoVentana/4,altoVentana/4,anchoVentana/2,altoVentana/2).
```

Por último, una de las prácticas más habituales será la definición del volumen de visualización, para lo que la siguiente función será de gran utilidad. Esta función será invocada tanto al crear una ventana como al cambiar sus dimensiones:

```
void ChangeSize(Glsizei w, Glsizei h)
{
      Glfloat nRange = 100.0f;
      //previene una división por 0
      if(h == 0)
            h = 1;
      //Ajusta la vista a las dimensiones de la ventana
      glViewport(0, 0, w, h);
      //Reinicia la pila de la matriz de proyección
      glMatrixMode(GL_PROYECTION);
      glLoadIdentity();
      //Establece el volumen de trabajo
      if(w <= h)
            glOrtho(-nRange, nRange, -nRange*h/w, nRange*h/w, -
nRange, nRange);
      else
            glOrtho(-nRange*w/h, nRange*w/h, -nRange, nRange, -
nRange, nRange);
      //Reinicia la pila de la matriz del modelador
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
}
```

3. PUNTOS

Cuando se aprende a dibujar cualquier tipo de gráfico en cualquier computadora, lo normal es empezar con los píxeles. Un píxel es el elemento más pequeño en el monitor de un ordenador. En sistemas con color, este píxel puede tener cualquiera de los colores disponibles. Este sería el nivel más simple de gráficos de ordenador: dibujar un punto en cualquier lugar de la pantalla y darle un color específico. A partir de este concepto simple, es posible producir en el lenguaje de computadores favorito líneas, polígonos, círculos y otras figuras y gráficos.

De igual modo, en OpenGL los puntos son los elementos esenciales utilizados para componer diferentes figuras. Un punto estará representado por un conjunto de números en punto flotante *vertex* y del que se hablará en profundidad a continuación.

3.1. Un punto en 3D: el vertex

Para especificar un punto dibujado en la paleta 3D, se utiliza la función OpenGL glVertex, sin ninguna duda la función más utilizada de todas las que ofrece la API de OpenGL. Es el mínimo común denominador de todas las primitivas de OpenGL: un simple punto en el espacio. La función glVertex toma como argumentos de 2 a 4 parámetros de tipo numérico, de bytes a dobles.

El siguiente código establece un punto en un sistema de coordenadas a 20 unidades en el eje x, 20 en el eje y y 0 en el eje z:

```
glVertex3f(20.0f, 20.0f, 0.0f);
```

El siguiente paso es dibujar algo y lo más sencillo son puntos. El código que se puede ver a continuación dibujará 2 puntos en la pantalla, indicando que se va a utilizar puntos como primitiva.

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

El argumento `GL_POINTS` indica a OpenGL que los siguientes vértices deben ser interpretados y dibujados como puntos. En este caso se especifican dos vértices que van a ser dibujados como dos puntos. Es importante señalar que se puede realizar múltiples llamadas a primitivas del mismo tipo entre llamadas a `glBegin` y `glEnd`.

3.2. Modificar el tamaño del punto

OpenGL permite un gran control sobre cómo son dibujadas la primitivas y los puntos no son una excepción. Hay tres aspectos que se pueden modificar: el tamaño de los puntos, si son o no anti-aliasados y si la distancia tiene un efecto sobre el tamaño y la transparencia del punto.

Para cambiar el tamaño del punto, es necesario utilizar la siguiente función:

```
void glPointSize(GLfloat tamaño);
```

El resultado de esta llamada es un cuadrado de de lado igual al argumento pasado centrado en el *vertex* cuyas coordenadas se haya especificado. El tamaño por defecto es de 1.0. Si el antialiasado del punto está desactivado (lo que es el comportamiento por defecto) el tamaño del punto será redondeado al entero más cercano (con un tamaño mínimo de 1) indicando las dimensiones del píxel del punto. Si se desea, se puede utilizar la función `glGet()` con el argumento `GL_POINT_SIZE` para averiguar el tamaño seleccionado actualmente. Esto se puede ver en el siguiente ejemplo:

```
// Recuperar el tamaño actual del punto
GLfloat antiguoTamaño;
glGetFloatv(GL_POINT_SIZE, &antiguoTamaño);

// Si el tamaño del punto es pequeño, se agranda (6.0), de lo
contrario se mantiene
If(antiguoTamaño < 1.0)
      glPointSize(6.0);
else
      glPointSize(1.0);
```

La implementación Microsoft de OpenGL admite tamaños de puntos de 0.5 a 10.0, aunque otras implementaciones permiten tamaños mucho mayores. Para comprobar el rango de tamaños que soporta nuestra implementación podemos usar la llamada `glGetFloatv(GL_POINT_SIZE_RANGE, dimension)`, donde dimensión es una matriz de 2 elementos de tipo `GLfloat` en el que se almacenará el valor mínimo y máximo para el tamaño de los puntos.

Además, para definir el tamaño mínimo en el que se puede aumentar o disminuir el grosor de un punto, tenemos la granularidad, un valor real que se puede obtener mediante la llamada `glGetFloatv(GL_POINT_SIZE_GRANULARITY, &salto)`, donde salto será el entero en el que se almacenará la granularidad. En la implementación Microsoft de OpenGL el incremento mínimo es de 0.125.

3.3. Efecto de distancia

Normalmente, los puntos ocupan siempre la misma cantidad de espacio en la pantalla, independientemente de la distancia a la que se encuentra el punto de vista del que mira. Para algunas aplicaciones con puntos, como por ejemplo los sistemas de partículas, es conveniente ver los puntos más pequeños a medida que se aleja el punto de vista. Esto se puede conseguir mediante el uso de las funciones `glPointParameter()`:

```
void glPointParameteriARB (enum pname, type param);
void glPointParameterivARB (enum pname, type param);
void glPointParameterfARB (enum pname, type param);
void glPointParameterfvARB (enum pname, type param);
```

Los valores para los parámetros `pname` y `param` pueden ser: `GL_POINT_SIZE_MIN`, `GL_POINT_SIZE_MAX`, `GL_POINT_FADE_THRESHOLD` y `GL_POINT_DISTANCE_ATTENUATION`.

3.4. Ejemplos

En el primer ejemplo vamos a hacer uso de la primitiva glVertex2f() para dibujar puntos de dos dimensiones sobre la pantalla. Para ello, vamos a hacer uso de un poco de matemáticas para dibujar una circunferencia compuesta de puntos disjuntos. El código que se muestra a continuación lograría dicho efecto:

```
glBegin(GL_POINTS);
for(ANG = 0.0f; ANG < 2 * GL_PI; ANG += paso)
{
      x = radio * fsin(ANG);
      y = radio * fcos(ANG);
      glVertex2f(x,y);
}
glEnd();
```

El resultado de la programación de este fragmento de código sería algo como lo que se muestra en la Ilustración 5.

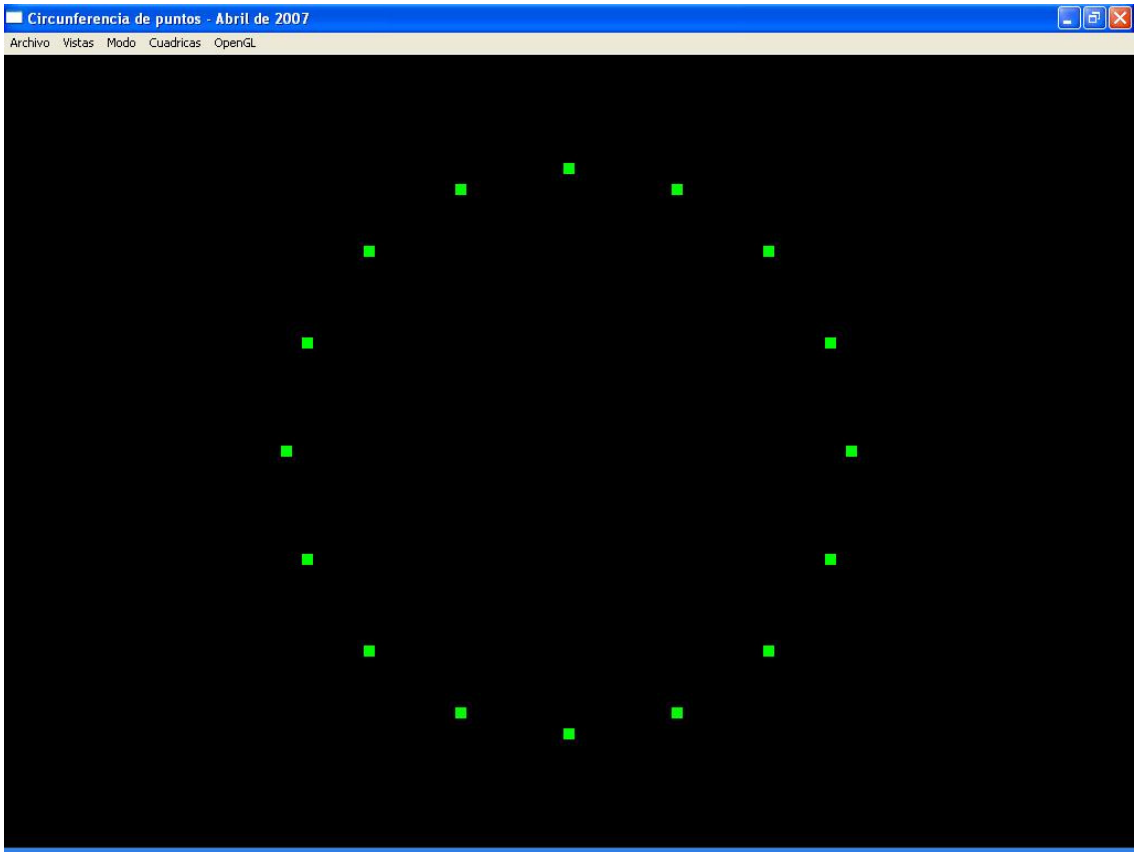


Ilustración 5: Resultado de la ejecución del primer ejemplo

En el próximo ejemplo se va a generar un conjunto de puntos en 2 dimensiones que van a estar colocados en el mismo plano. Para poder demostrar esta afirmación, se han hecho dos capturas del resultado de la aplicación del código que muestran los puntos de forma frontal y desde un lateral de la imagen. El código para lograr este efecto es:

```
void CALLBACK RenderScene(void)
{
	// Limpiamos la ventana con un color de fondo.
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	// Guardamos el estado de la matriz y hacemos la rotación.
	glPushMatrix();
	glRotatef(xRot, 1.0f, 0.0f, 0.0f);
	glRotatef(yRot, 0.0f, 1.0f, 0.0f);
	/**********ZONA DE DIBUJO***********/
	//Dibujamos 50 puntos aleatorios en la pantalla.
	//Los puntos están todos en el mismo plano (z=0)
	glBegin(GL_POINTS);
	for(int i=0;i<50;i++)
		glVertex2f(puntos[i][0],puntos[i][1]);
	glEnd();
	// Restore transformations
	glPopMatrix();
	// Flush drawing commands
	glFlush();
}
```

La matriz de puntos utilizada en esta función se habrá generado previamente en el evento `WM_CREATE` de la ventana. Es importante destacar cómo gracias a la función `glVertex2f` se generan cada uno de los 50 puntos. Todos ellos están en el mismo plano como se muestra en las siguientes capturas. La primera es de los puntos vistos frontalmente:

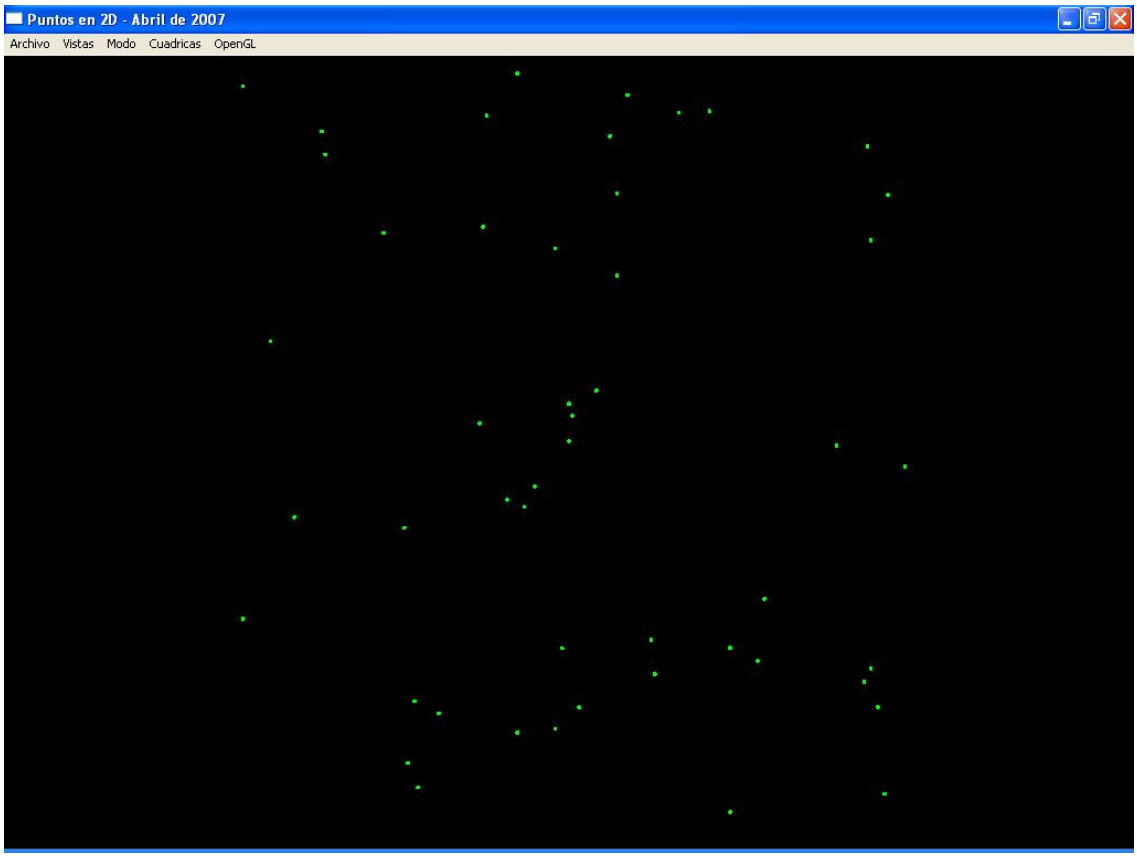


Ilustración 6: Resultado de la ejecución del segundo ejemplo (Frente)

La captura de la Ilustración 7 muestra los mismos puntos vistos desde otro de punto de vista, tras realizar una traslación en los ejes de dibujado.

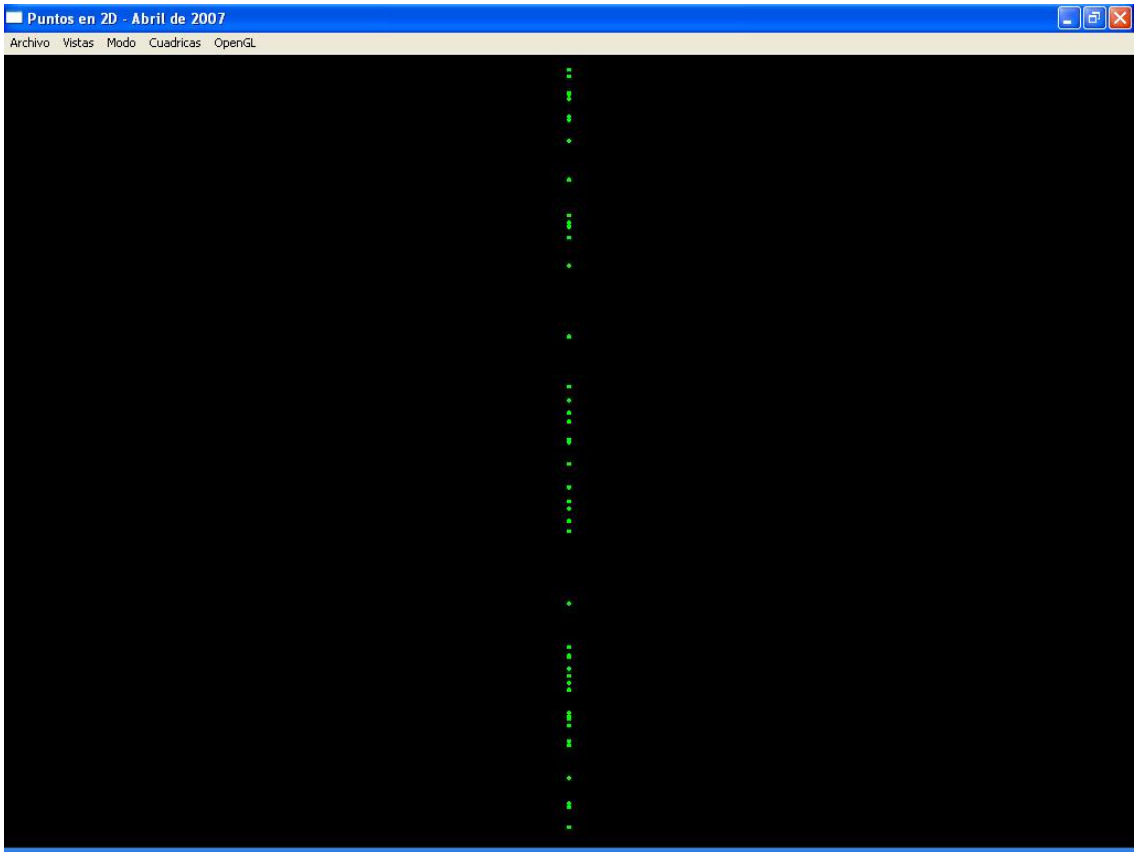


Ilustración 7: Resultado de la ejecución del segundo ejemplo (Lateral)

4. Dibujando líneas en 3D

La primitiva GL_POINTS que hemos venido utilizando hasta ahora es bastante directa, en tanto en cuanto para cada vértice especificado, dibuja un punto. El siguiente paso lógico es especificar dos vértices y dibujar una línea recta entre ellos.

Ésta operación es llevada a cabo por la primitiva GL_LINES. La siguiente sección de código dibuja una línea entre los puntos (0,0,0) y (20,20,20)

```
glBegin(GL_LINES);
      glVertex3f(0.0f, 0.0f, 0.0f);
      glVertex3f(20.0f, 20.0f, 20.0f);
glEnd();
```

Nótese que una línea es definida únicamente por dos vértices, pero la primitiva puede contener tantas parejas de vértices como se desee, tomándose de dos en dos de manera consecutiva. En el caso de que se especifique un mayor impar de vértices, el último será ignorado.

A continuación se muestra una sección de código más compleja que muestra una serie de líneas a modo de radios de una rueda

```
//Llamar una única vez para todos los puntos
glBegin(GL_LINES);
//Todas las líneas se definen en el plano xy
z = 0.0f;
for(angulo = 0.0f; angulo <= GL_PI*3.0f; angulo += 0.5f)
{
      //Puntos en la mitad superior de la circunferencia
      x = 40.0f*sin(angulo);
      y = 40.0f*cos(angulo);
      glVertex3f(x, y, z);
      //Puntos en la mitad inferior de la circunferencia
      x = 40.0f*sin(angulo+3.1415f);
      y = 40.0f*cos(angulo+3.1415f);
      glVertex3f(x, y, z);
}
glEnd();
```

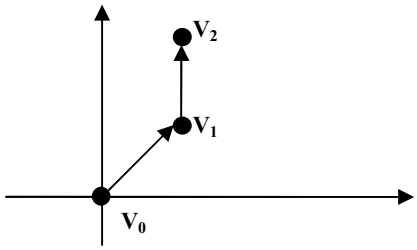
4.1. Series de líneas y lazos

Las dos siguientes primitivas de OpenGL se basan el `GL_LINES` para permitir especificar una lista de vértices a través de los cuales de dibuja una única línea continua.

De este modo, cuando se emplea la primitiva `GL_LINE_STR`IP, se especifica que se desea dibujar una línea de un vértice al siguiente pero de manera continua, de tal manera que un vértice sea inicio y fin de un segmento (a excepción del primero –sólo inicio– y del último –solo fin).

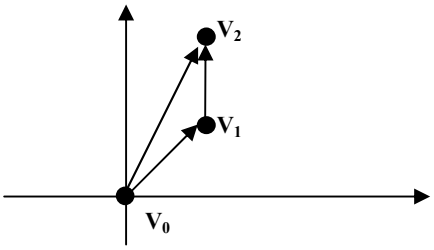
El siguiente segmento de código dibuja dos líneas en el plano xy definidas por tres vértices. Se muestra el resultado junto al código.

```
glBegin(GL_LINE_STRIP);
      glVertex3f(0.0f, 0.0f, 0.0f);   // V0
      glVertex3f(40.0f, 40.0f, 0.0f); // V1
      glVertex3f(40.0f, 90.0f, 0.0f); // V2
glEnd();
```



La última primitiva para el trazado de líneas es `GL_LINE_LOOP`. Esta primitiva se comporta exactamente igual que `GL_LINE_STRIP` con la salvedad de que se dibuja un línea adicional entre el primer y el último vértice especificado. Ésta es, por lo tanto, una manera sencilla de dibujar una figura de líneas cerrada. El siguiente segmento de código muestra el mismo ejemplo que para la primitiva `GL_LINE_STRIP`, pero empleando esta nueva primitiva.

```
glBegin(GL_LINE_LOOP);
      glVertex3f(0.0f, 0.0f, 0.0f);   // V0
      glVertex3f(40.0f, 40.0f, 0.0f); // V1
      glVertex3f(40.0f, 90.0f, 0.0f); // V2
glEnd();
```



4.2. Aproximación de curvas con líneas rectas

Obviamente, cualquier figura, y en particular, una curva, puede construirse a través de un conjunto de puntos correctamente especificados empelando la primitiva `POINTS` previamente tratada. Sin embargo, para figuras complejas, puede resultar excesivamente lento por requerir cientos o miles de puntos de tal manera que no se note la separación entre ellos.

Una manera más sencilla y válida de aproximar una curva es utilizar la primitiva `GL_LINE_STRIP` para conectar los puntos. A medida que los puntos se definen más próximos unos de otros, una curva más fina y visualmente más perfecta aparece, sin necesidad de especificar todos los puntos.

El siguiente segmento de código muestra cómo dibujar una espiral empleando esta técnica.

```
//Llamar una única vez para todos los puntos
glBegin(GL_LINE_STRIP);
z = -50.0f;
for(angulo=0.0f; angulo<=(2.0f*GL_PI)*3.0f; angulo+=0.1f)
{
      x=50.0f*sin(angulo);
      y=50.0f*cos(angulo);
      //Especificar el punto y mover el valor de z
      //ligeramente hacia arriba
      glVertex3f(x, y, z);
      z+=0.5f;
}
glEnd();
```

El resultado es el mostrado en la Ilustración 8:

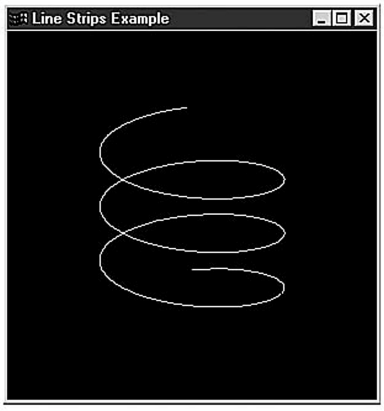


Ilustración 8: Resultado de ejemplo de dibujado de espiral

4.3. Establecer el ancho de una línea

Del mismo modo que se pueden establecer diferentes tamaños para los puntos, también es posible especificar varios anchos de línea cuando éstas son dibujadas. Esto es llevado a cabo con la función `glLineWidth`:

```
void glLineWidth(GLfloat width);
```

Esta función toma un único parámetro que especifica el ancho aproximado, en píxeles, de la línea dibujada. Exactamente igual que para el tamaño de los puntos, no todos los anchos son soportados, y se deberá comprobar si el ancho que se desea especificar es válido. El siguiente código muestra el rango de anchos válidos y el menor intervalo entre ellos:

```
GLfloat tamanos[2]; // Almacena el rango soportado de ancho de
línea
GLfloat intervalo; // Almacena el incremento soportado para
ancho de línea

// Obtiene el rango de ancho y el incremento soportados
glGetFloatv(GL_LINE_WIDTH_RANGE, sizes);
glGetFloatv(GL_LINE_WIDTH_GRANULARITY, &step);
```

En este ejemplo, la variable `tamanos` contendrá dos elementos que harán referencia al menor y el mayor valor válido que la función `glLineWidth` puede tomar. Por otro lado, en la variable `intervalo` se almacenará el intervalo menor de tamaño permitido entre valores de ancho de línea.

El siguiente ejemplo muestra cómo dibujar líneas de diferentes anchos:

```
// Llamada para redibujar una escena
void RenderScene(void)
{
      GLfloat y;
      GLfloat fTam[2];
      GLfloat fTamAct;
      …
      //Almacenar el valor menor y mayor para ancho
      glGetFloatv(GL_LINE_WIDTH_RANGE,fTam);
      fTamAct=fTam[0];

      // Subir 20 unidades en el eje Y cada línea
      for(y=-90.0f; y<90.0f; y+=20.0f)
      {
            // Establecer el ancho de línea
            glLineWidth(fTamAct);
            // Dibujar la línea
            glBegin(GL_LINES);
                  glVertex2f(-80.0f, y);
                  glVertex2f(80.0f, y);
            glEnd();
```

```
                // Se incrementa el ancho
                fTamAct += 1.0f;
        }
        …
}
```

El resultado obtenido con el ejemplo anterior se puede observar en la Ilustración 9.

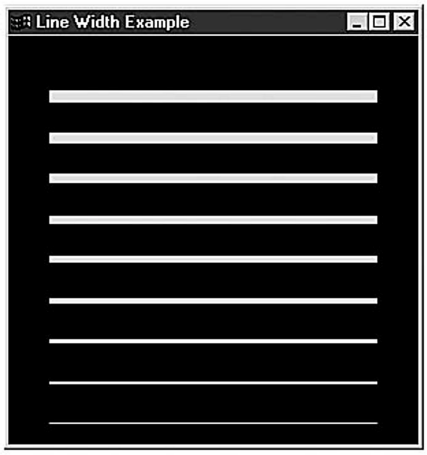


Ilustración 9: Líneas con distintos grosores

Nótese que en esta ocasión se ha empleado la función `glVertex2f` para indicar la posición de las coordenadas de las líneas. Esto es así dado que se está dibujando únicamente en el plano xy, con un valor de z de 0.

4.4. Líneas punteadas

Además de permitir camibar el ancho de las lineas, también es posible dibujar líneas con un patron punteado o rayado (*stippling*). Para utilizar esta técnica, es necesario activar la característica “punteado” con una llamada al método:

```
glEnable(GL_LINE_STIPPLE);
```

Una vez activado, la función `glLineStipple` establece el patrón que se empleará en el dibujado de la línea:

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

El parámetro patrón es un valor de 16 bits que especifica un patrón a utilizar cuando se dibujen las líneas. Cada uno de los bits representa una sección del segmento de la línea que está activo (se dibuja) o inactivo (no se dibuja). Por defecto, cada uno de los bits se corresponde con un único píxel, pero el parámetro factor se puede emplear como un multiplicador para incrementar el ancho del patrón.

Por ejemplo, si el valor del factor se establece a 5, cada bit en el patrón representará cinco píxeles en la línea que estarán bien activos o inactivos.

La figura ilustra un ejemplo de lo que se trata de explicar:

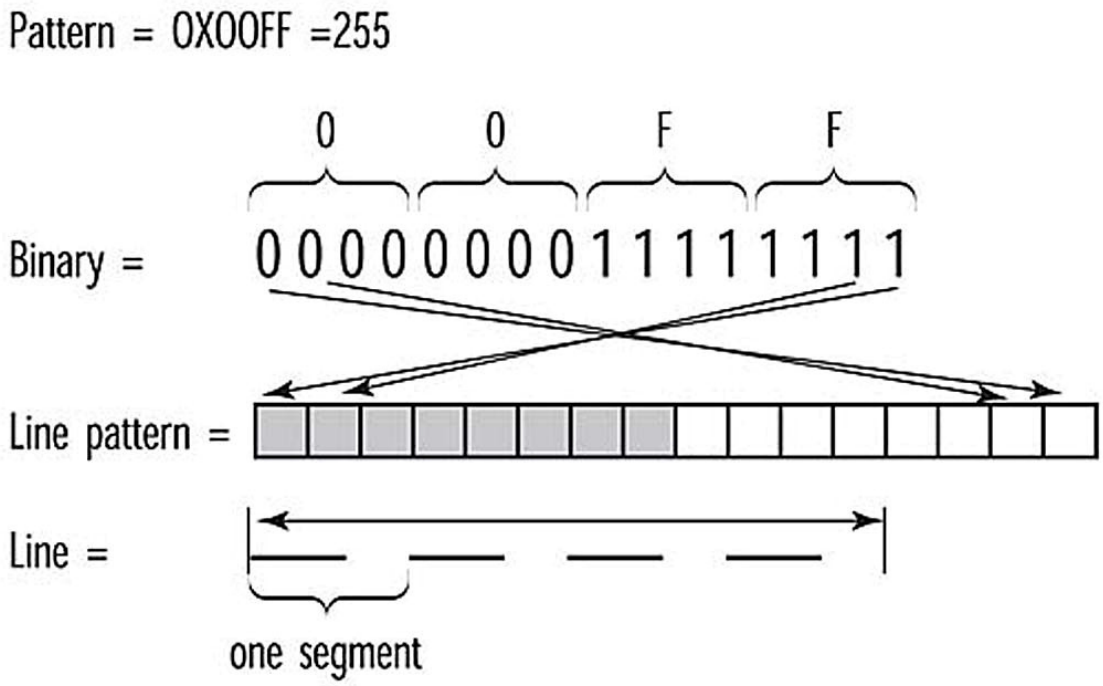


Ilustración 10: Patrón de bits para el pintado de líneas

El siguiente fragmento de código muestra un ejemplo de utilización de patrones de puntos, empleándose siempre el mismo patrón y variando el factor de multiplicación de dicho patrón:

```
// Llamada para redibujar una escena
void RenderScene(void)
{
      GLfloat y;
      GLint factor=1;
      GLushort patron = 0x5555;
      …

      // Activar el punteado
      glEnable(GL_LINE_STIPPLE);

      // Subir en el eje y 20 unidades cada vez
      for(y=-90.0f; y<90.0f; y+=20.0f)
      {
```

```
                // Establecer el patrón
                glLineSipple(factor, patron);

                // Dibujar la línea
                glBegin(GL_LINES);
                      glVertex2f(-80.0f, y);
                      glVertex2f(80.0f, y);
                glEnd();

                factor++;
        }
        …
}
```

La salida del programa resultante es la siguiente:

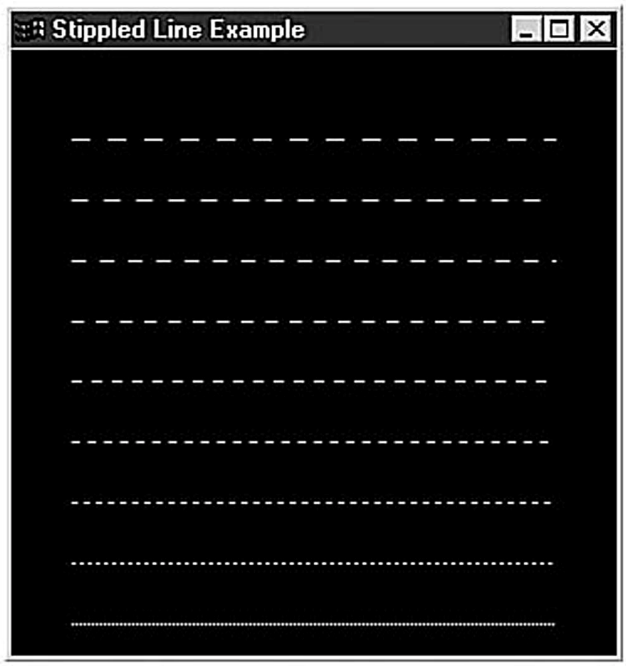


Ilustración 11: Resultado de ejemplo de dibujado de líneas punteadas

5. Polígonos

En este apartado se describirá ampliamente el uso de polígonos en OpenGL. Para ello este apartado estará dividido en dos partes: una primera dónde se presentan los aspectos teóricos y una segunda dedicada a la parte práctica mediante la visualización de ejemplos.

5.1. Teoría

En esta sección se tratarán los aspectos teóricos del manejo de polígonos en OpenGL comenzando con una definición para posteriormente definir la validez de un polígono. A continuación se presentarán aspectos más técnicos como son el encaramiento, los ajustes de color, los modos poligonales y la eliminación de las caras ocultas.

5.1.1. Definición

Los polígonos son las áreas comprendidas en bucles de líneas cerrados, donde los segmentos de líneas están especificados por los vértices de sus extremos. Normalmente los polígonos se dibujan con los píxeles de su interior rellenos, pero también pueden dibujarse sólo sus aristas o los puntos de los vértices.

5.1.2. Polígonos válidos

En general, los polígonos pueden ser complicados, así que OpenGL pone algunas restricciones en lo que se considera un polígono primitivo. Primero, las aristas de los polígonos en OpenGL no se pueden cortar, es decir, OpenGL sólo admite lo que matemáticamente se denomina polígonos simples; aunque, de todas formas, cualquier polígono complejo puede construirse mediante polígonos simples. En segundo lugar, los polígonos OpenGL deben ser convexos, lo cual significa que no pueden tener entrantes (también éstos pueden conseguirse con la composición de polígonos cóncavos). De forma más precisa, un polígono es convexo si la unión de cualesquiera dos vértices del polígono es una recta que está dentro de los límites del mismo. Por último, los polígonos con agujeros no se pueden describir, ya que no son convexos, y además no pueden ser dibujados con un solo bucle cerrado.

Como ya se ha comentado, estos tipos de polígonos complejos pueden crearse con la unión de polígonos simples, de esta forma existen funciones en

la biblioteca GLU que permiten la definición de polígonos complejos, cóncavos y con agujeros. Estas funciones se denominan rutinas de teselación.

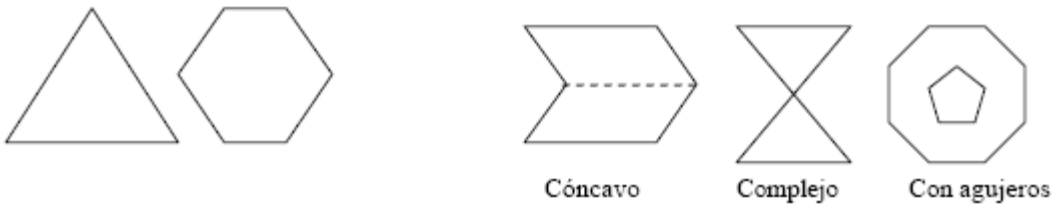


Ilustración 12: Polígonos válidos (izquierda) y no válidos (derecha)

Nota: En el ejemplo del polígono cóncavo la línea discontinua muestra una posible descomposición en polígonos convexos.

Ya que los vértices en OpenGL son siempre tridimensionales, los puntos que forman los límites de un polígono particular no tienen porqué estar en el mismo plano del espacio. Si los vértices del polígono no están en el mismo plano, después de varias rotaciones en el espacio, cambios en el punto de vista y en la proyección en pantalla, puede que los puntos no sigan formando un polígono convexo simple. Este problema puede evitarse utilizando triángulos, ya que sus tres puntos siempre están en el mismo plano.

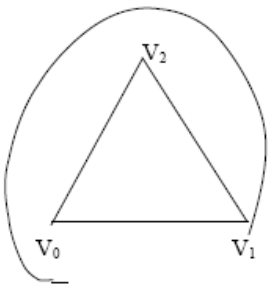
5.1.3. Encaramiento

Una de las características más importantes de cualquier polígono es el encaramiento.

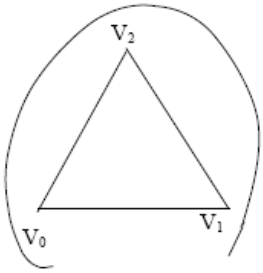
Aunque en principio podría considerarse una característica sin demasiada importancia, nos va a determinar cuál va a ser la cara frontal del polígono y cuál la trasera, lo cual será útil para las funciones de ocultación.

La cara frontal viene determinada por el orden en que se definen los vértices del polígono. Por defecto, si éstos se definen en el sentido contrario a las agujas del reloj, el polígono estará encarado frontalmente, en el caso contrario el encare será posterior. Aclaremos esto con un par de ejemplos:

- Sentido horario negativo:
  De esta forma conseguimos un encare frontal.
  Para ello definimos los vértices en el orden V0, V1, V2.



- Sentido horario positivo:
  De esta forma conseguimos un encare posterior.
  Para ello definimos los vértices en el orden V0, V2, V1.



El comportamiento por defecto puede cambiarse con la función glFrontFace(). Se le pasa como argumento una macro, que puede ser GL_CW, para determinar que el encare frontal se consiga definiendo los vértices en el sentido horario, o GL_CCW que tiene el mismo comportamiento que por defecto.

5.1.4. Ajuste de color en los polígonos

A la hora de realizar figuras más o menos complejas utilizando triángulos (o polígonos en general, aunque ya hemos mencionado que suelen usarse triángulos) puede interesarnos la utilización de distintos colores. Los colores por defecto se especifican para cada vértice, no para cada polígono. Esto nos lleva a que, si cambiamos el color entre cada especificación de vértices, el cambio será válido para el siguiente vértice y obtendremos así un efecto de degradado entre los vértices que tengan asignados distintos colores.

El comportamiento para el ajuste de colores puede cambiarse utilizando la función `glShadeModel()`. Si el argumento de dicha función es `GL_SMOOTH`, obtendremos el comportamiento por defecto, es decir, los colores se asignan a cada vértice y OpenGL tratará de interpolar los colores entre los especificados para cada vértice obteniendo el efecto de difuminación. Con la macro `GL_FLAT`, OpenGL rellenará los polígonos con el color sólido que estaba activo cuando se especificó el último vértice del polígono, es decir, obtendremos polígonos rellenos con un solo color.

5.1.5. Modos poligonales

Los polígonos pueden representarse de tres formas distintas. La función que cambia la representación es `glPoligonMode()`. Tiene dos argumentos, el primero para decirle las caras de las que queremos cambiar el tipo de representación (se especifica con las macros `GL_FRONT`, `GL_BAC`K o `GL_FRONT_AND_BACK`); y el segundo para el tipo de representación en sí. Este segundo parámetro puede tomar tres valores:

- `GL_POINT`: Muestra tan sólo los vértices del polígono.
- `GL_LINE`: Muestra las aristas del polígono.
- `GL_FILL`: Muestra los polígonos enteros, rellenos.

5.1.6. Eliminación de caras ocultas

La prueba de profundidad consiste en no dibujar polígonos que espacialmente están detrás de otras formas, luego no se van a ver en el total.

OpenGL entiende por estar detrás tener un valor de z menor que otro.

Hay que tener en cuenta que mediante esta prueba de profundidad no mostramos estas partes ocultas, pero aún así OpenGL sigue calculándolas, luego el rendimiento de la aplicación no aumentará e incluso se degradará.

Para habilitar el test de profundidad tenemos que llamar a `glEnable()` con la macro `GL_DEPTH_TEST` como parámetro. Para deshabilitarlo llamamos con la misma macro a `glDisable()`. Por defecto esta opción está deshabilitada.

Además, hay que limpiar el buffer de profundidad cada vez que se genera la escena. El buffer de profundidad es análogo al buffer de color en que contiene información sobre la distancia entre los píxeles y el observador. Este se usa para determinar si cualquier píxel está oculto por píxeles más cercanos al observador.

Para eso hacemos la siguiente llamada:

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

Podemos ver que la prueba de profundidad es prácticamente un prerrequisito para la creación de objetos sólidos a partir de polígonos.

Aparte del test de profundidad, y para solventar el problema que hemos comentado del decremento de rendimiento, está la estrategia de no calcular aquello que en última instancia no se va a mostrar.

Esto se consigue en OpenGL con la función `glEnable()` / `glDisable()`, con la macro `GL_CULL_FACE` como parámetro.

Esto nos elimina los polígonos que se ven por su cara trasera en todo momento. Este comportamiento puede dar resultados no deseados para figuras abiertas (como un cono sin base), ya que elimina el polígono entero, aunque éste o parte de éste sí que sea visible.

Combinando este comportamiento con la función `glFrontFace()` que se vio anteriormente podemos cambiar el efecto del ocultamiento.

5.2. Práctica

En esta segunda parte se describe la parte práctica del tratamiento de polígonos en OpenGL comenzando con la presentación de los triángulos, para continuar con los cuadriláteros y finalizar con los polígonos (más de 4 lados).

5.2.1. Triángulos

El triángulo es el polígono más simple, ya que tiene el mínimo número de vértices para definir una superficie cerrada. Como se comentó en el apartado teórico, todos los vértices de un triángulo se encuentran en el mismo plano, evitando así los problemas ya mencionados.

El triángulo es la primitiva más recomendada para programar en OpenGL, ya que cualquier otra forma poligonal puede descomponerse en triángulos. También podemos aproximar cualquier forma tridimensional cerrada utilizando triángulos de forma similar a la aproximación de curvas por medio de rectas que vimos en el apartado correspondiente.

Además, la mayoría del hardware de aceleración 3D está altamente optimizado para el dibujo de triángulos, de hecho varios bancos de prueba 3D están medidos en triángulos.

Para el dibujo de triángulos utilizaremos, como siempre, `glBegin()` y `glEnd()`. En este caso la macro del argumento para `glBegin` será `GL_TRIANGLES`. En el interior especificaremos un número de vértices que deberá ser múltiplo de tres. OpenGL cogerá los vértices de tres en tres para dibujar tantos triángulos como le sea posible. Si el número de puntos especificados no fuera múltiplo de tres, los últimos vértices definidos serían ignorados.

Por ejemplo, si usamos los vértices V_1, V_2, V_3, V_4, V_5, V_6 y V_7 tendríamos el resultado de la Ilustración 13.

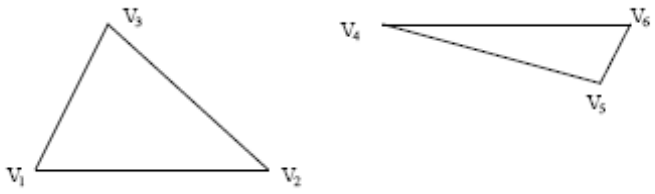


Ilustración 13: Resultado de dibujado de triángulos

Vemos que el vértice V7 ha sido ignorado.

Para muchas aplicaciones se necesitan conjuntos de triángulos conectados. La primitiva `GL_TRIANGLE_STRIP` nos permite crear una tira de triángulos con aristas unidas.

Los tres primeros vértices que se le especifiquen se utilizan para crear un triángulo. A partir del cuarto, toma el nuevo vértice y los dos últimos del triángulo anterior para dibujar otro triángulo que compartirá una arista con el anterior. OpenGL adapta el orden de los vértices de los triángulos de la tira para que todos tengan el mismo encaramiento que el primero.

Con los mismos puntos que antes pero con la primitiva `GL_TRIANGLE_STRIP` obtendremos el resultado mostrado en la Ilustración 14.

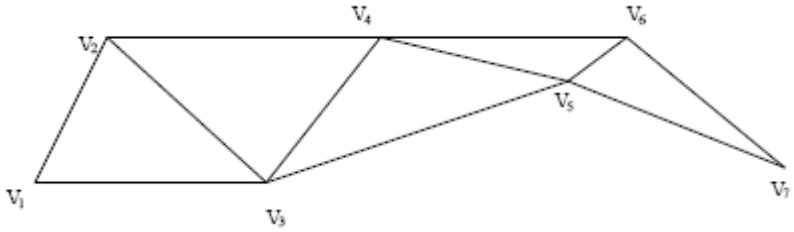


Ilustración 14: Resultado de aplicación de la primitiva GL_TRIANGLE_STRIP

La utilización de tiras de triángulos en lugar de especificar los triángulos separadamente tiene dos claras ventajas. Primero, tras especificar los tres primeros vértices para el triángulo inicial, sólo se necesita especificar un único punto para cada triángulo adicional. Esto ahorra mucho tiempo y espacio de datos cuando se dibujan muchos triángulos.

La segunda ventaja es que así podemos componer objetos o superficies utilizando triángulos lo cual, como hemos visto reporta beneficios.

Además de tiras de triángulos, se puede usar la primitiva `GL_TRIANGLE_FAN` para producir un grupo de triángulos conectados que se sitúan en torno a un punto central. El primer vértice forma el origen de giro, es decir, el punto central alrededor del cual se agruparán los triángulos. Después de que se usen los tres primeros vértices para dibujar el triángulo inicial, todos los vértices subsiguientes se emplean con el origen y el vértice que lo precede inmediatamente para formar el siguiente triángulo.

Así, si especificamos los vértices V_1, V_2, V_3, V_4 y V_5 tendríamos la figura de la Ilustración 15.

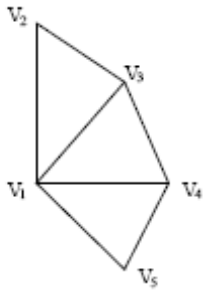


Ilustración 15: Resultado de aplicación de la primitiva GL_TRIANGLE_FAN

Vemos que todos los triángulos tienen como punto común V_1 y que toman éste y el último vértice del anterior triángulo para definir su primera arista.

5.2.1.1. Primer ejemplo: Triángulos aleatorios

Éste es un ejemplo muy completo que nos muestra todas las posibilidades de dibujo de triángulos.

El ejemplo crea cuatro triángulos con vértices aleatorios en sus tres coordenadas, mediante un código ya familiar:

```
glBegin(GL_TRIANGLES);
for(int i=0;i<13;i++)
{
   glColor3f(color[i][0], color[i][1], color[i][2]);
   glVertex3f(puntos[i][0],puntos[i][1],puntos[i][2]);
}
glEnd();
```

Los vectores color y puntos son calculados anteriormente con la función aleatoria `rand()`:

```
//Calculamos las coordenadas de los vértices aleatoriamente.
srand( (unsigned)time( NULL ));
for(i=0;i<13;i++)
{
   float x=(float)(rand()%100);
   float y=(float)(rand()%100);
   float z=(float)(rand()%100);
   int xs=(rand()%2);
   int ys=(rand()%2);
   int zs=(rand()%2);
   if(xs) x=-x;
   if(ys) y=-y;
   if(zs) z=-z;
   puntos[i][0]=x;
   puntos[i][1]=y;
   puntos[i][2]=z;
   color[i][0]=float((rand()%100))/100;
   color[i][1]=float((rand()%100))/100;
   color[i][2]=float((rand()%100))/100;
}
```

Como se puede observar, en ambos códigos contamos con 13 vértices, así podemos ver cómo OpenGL con la macro `GL_TRIANGLES` nos ignorará el último vértice por no ser suficiente para crear un triángulo completo.

La ejecución por defecto de este ejemplo produce la salida que se puede observar en la Ilustración 16.

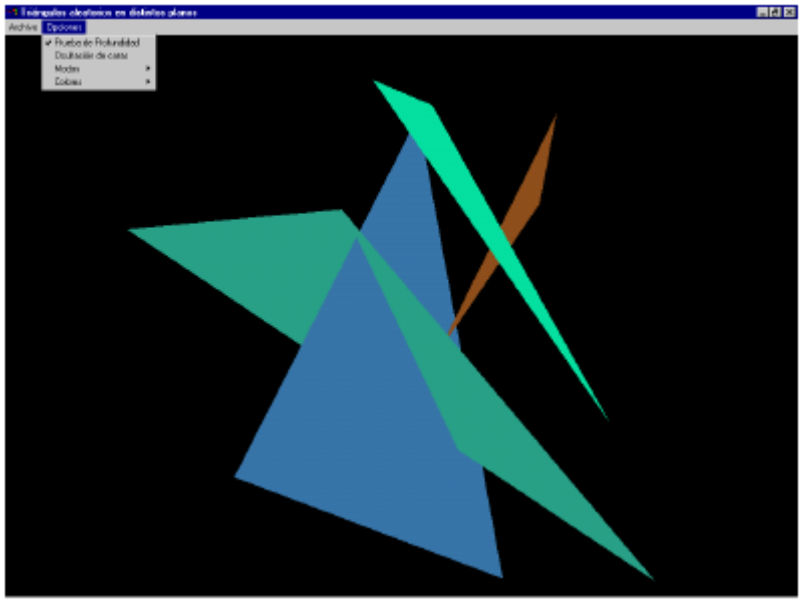


Ilustración 16. Resultado de la ejecución

Podemos ver los cuatro triángulos correspondientes a los 12 vértices y cómo estos están representados mediante la prueba de profundidad (que hemos activado por defecto), de modo que se cortan en algunos puntos. Sin la prueba de profundidad los triángulos quedan superpuestos como se puede ver en la Ilustración 17.

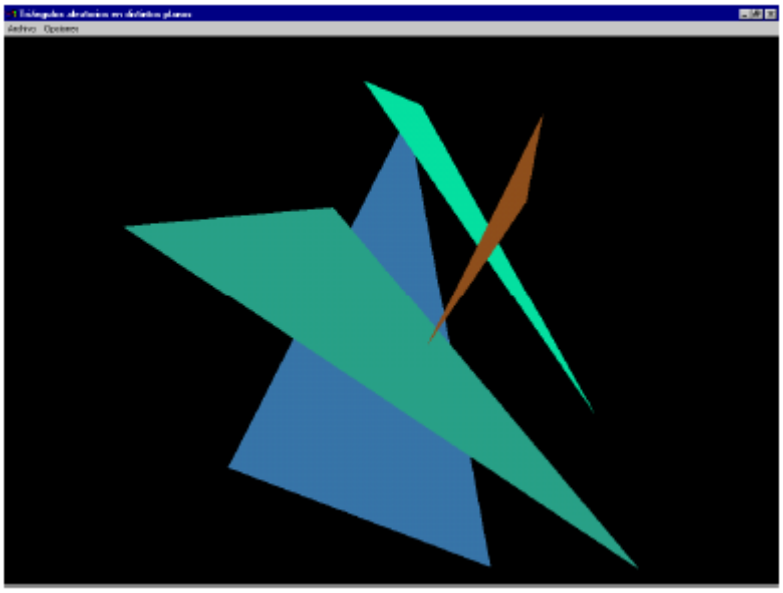


Ilustración 17: Resultado de la ejecución

La segunda opción que podemos activar es la “Ocultación de Caras”, que nos dejará sin dibujar aquellos triángulos que estén encarados por su parte trasera. No vamos a mostrar ninguna imagen porque lo veremos más claramente en ejemplos posteriores, aquí simplemente tendrá el efecto de dibujar menos triángulos.

Observemos ahora el resultado de cambiar el modo de dibujo para que aparezcan dibujadas sólo las aristas de los triángulos (Ilustración 18).

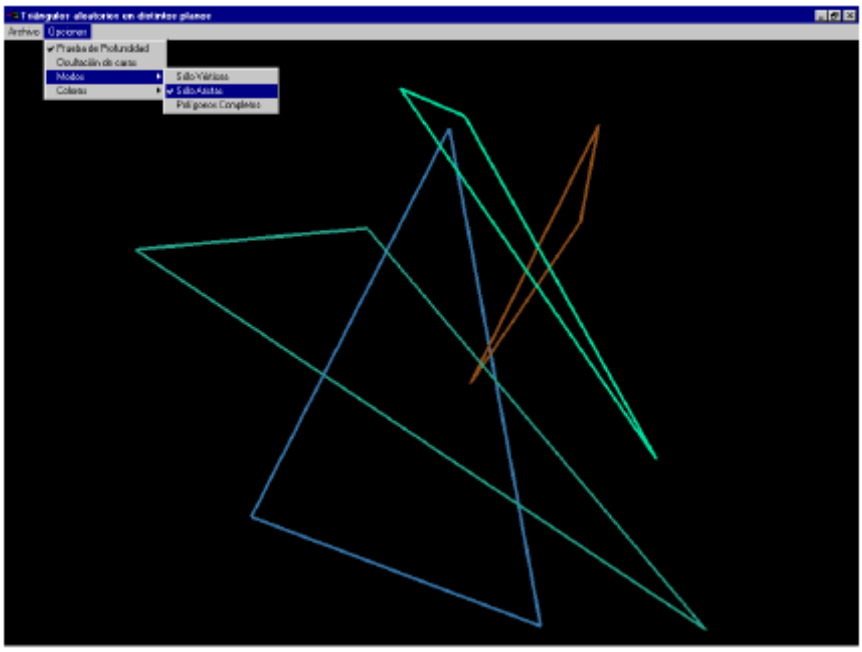


Ilustración 18: Resultado de la ejecución

Esto lo hemos conseguido gracias al siguiente fragmento de código:

```
switch(modos)
{
   case 0:
      glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
      break;
   case 1:
      glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
      break;
   case 2:
      glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
      break;
}
```

La variable `modos` se modifica en las opciones de menú correspondientes.

Ahora dibujando sólo los vértices resulta lo siguiente:

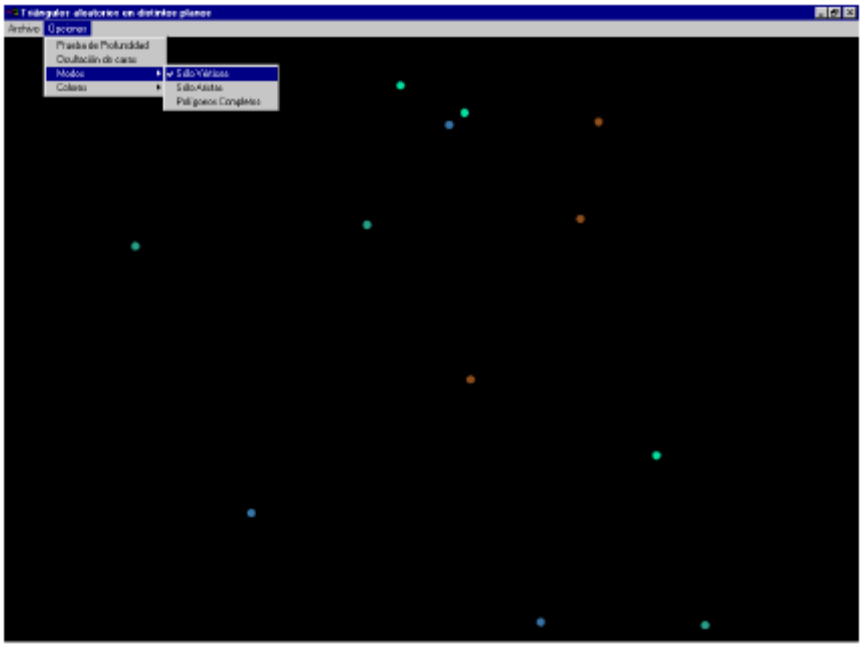


Ilustración 19: Resultado de la ejecución (dibujado de vértices)

Puntos, líneas y polígonos pueden suavizarse en sus formas evitando el *aliasing* mediante el botón de función F3, gracias al siguiente código:

```
if(wParam == VK_F3)
{
   if(!suave)
   {
      glEnable(GL_POLYGON_SMOOTH);
      glEnable(GL_LINE_SMOOTH);
      glEnable(GL_POINT_SMOOTH);
      glEnable(GL_BLEND);
      glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
      glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
      glHint(GL_POLYGON_SMOOTH_HINT, GL_FASTEST);
   }
   else
   {
      glDisable(GL_POINT_SMOOTH);
      glDisable(GL_LINE_SMOOTH);
      glDisable(GL_POLYGON_SMOOTH);
      glDisable(GL_BLEND);
   }
   suave=!suave;
}
```

Para el caso de polígonos, hemos usado la opción `GL_FASTEST` en vez de `GL_NICEST` porque el *antialiasing* de triángulos degrada bastante el rendimiento.

Hemos habilitado una opción que nos permite cambiar los colores, convirtiéndolos en un degradado a partir de cada vértice en vez de un color plano para cada triángulo.

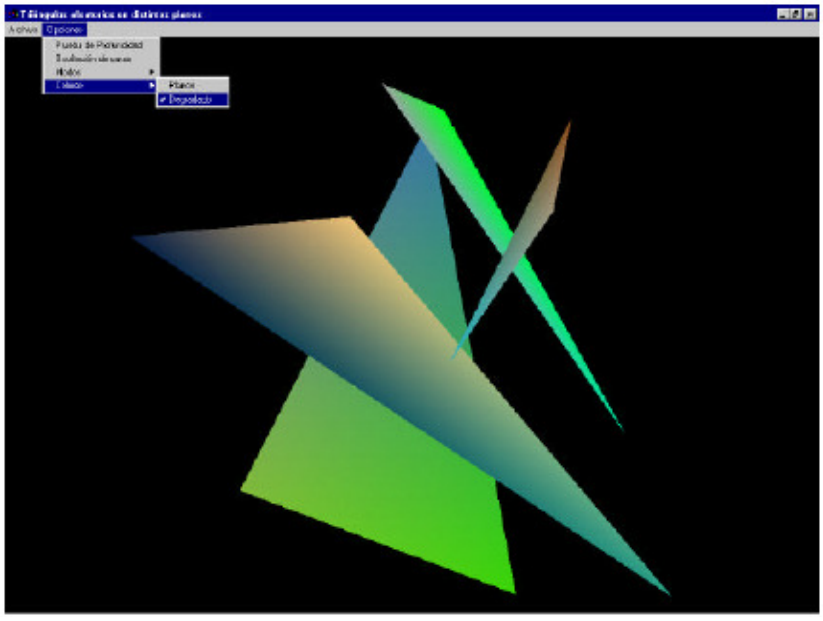


Ilustración 20: Triángulos con degradado

El código que regula el tipo de colores y los distintos tests se presenta a continuación (todas las variables implicadas son modificadas en las correspondientes opciones de menú):

```
if(bCull)
   glEnable(GL_CULL_FACE);
else
   glDisable(GL_CULL_FACE);
if(bDepth)
   glEnable(GL_DEPTH_TEST);
else
   glDisable(GL_DEPTH_TEST);
if(colores)
   glShadeModel(GL_SMOOTH);
else
   glShadeModel(GL_FLAT);
```

Por último tenemos las subopciones de tipo para usar los vértices no sólo como triángulos separados sino también como tiras de triángulos o abanicos de triángulos.

Para que sean fácilmente visualizables, los puntos elegidos están precalculados y son estáticos. En la Ilustración 21 se muestra un ejemplo de tira de triángulos:

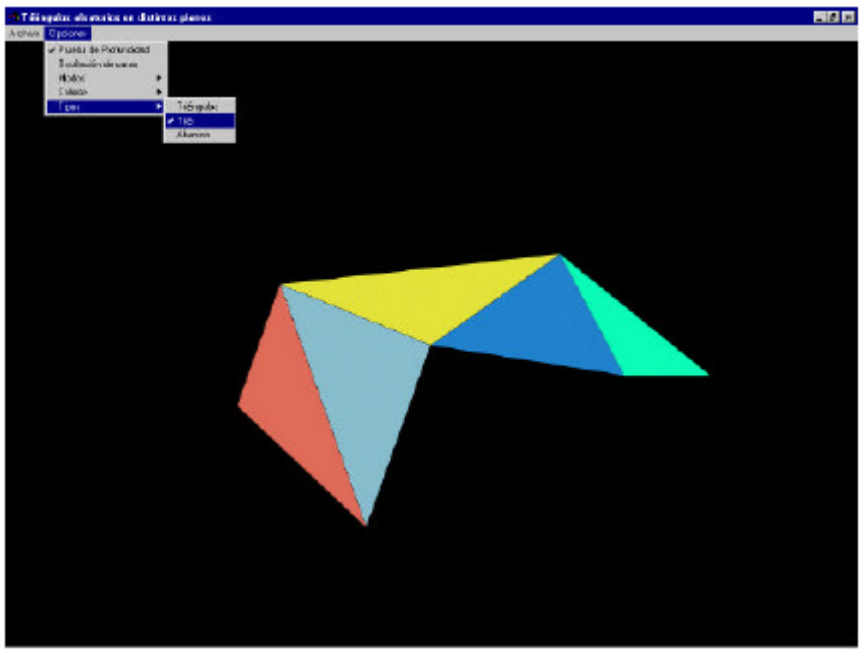


Ilustración 21: Tira de triángulos

Y en la Ilustración 22 se tiene un abanico que forma una especie de pirámide sin base:

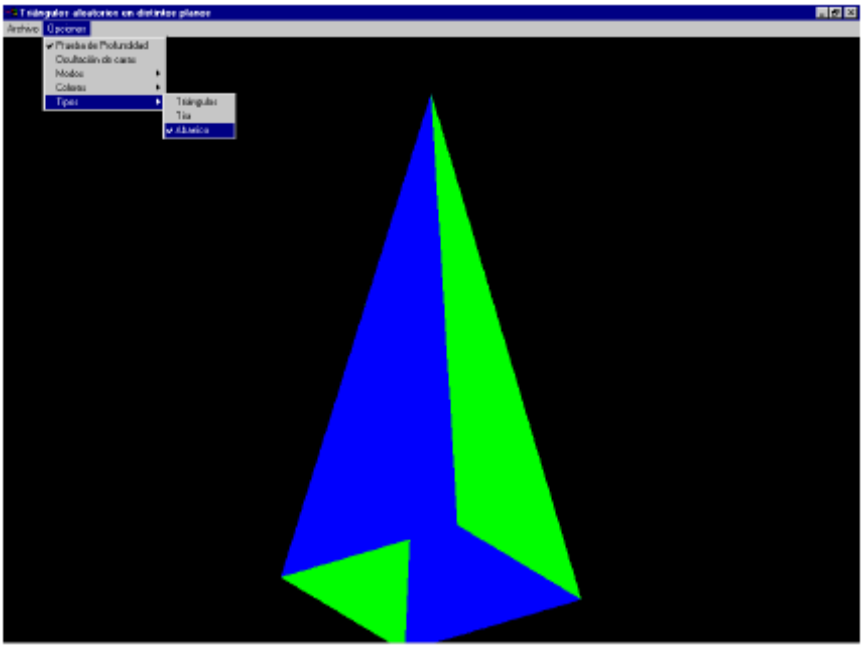


Ilustración 22: Abanico de triángulos

Vemos que los dos triángulos que quedan por detrás son visibles. Si activáramos la ocultación de caras, no se vería ninguna de las dos dándonos una figura un poco extraña, que si cerráramos (con un cuadrado u otro abanico) la base no se notaría.

5.2.1.2. Segundo ejemplo: Cono de colores

Este programa crea una pirámide con 6 caras dibujadas en colores alternados (azul y verde). La ejecución simple de este programa da resultado de la:

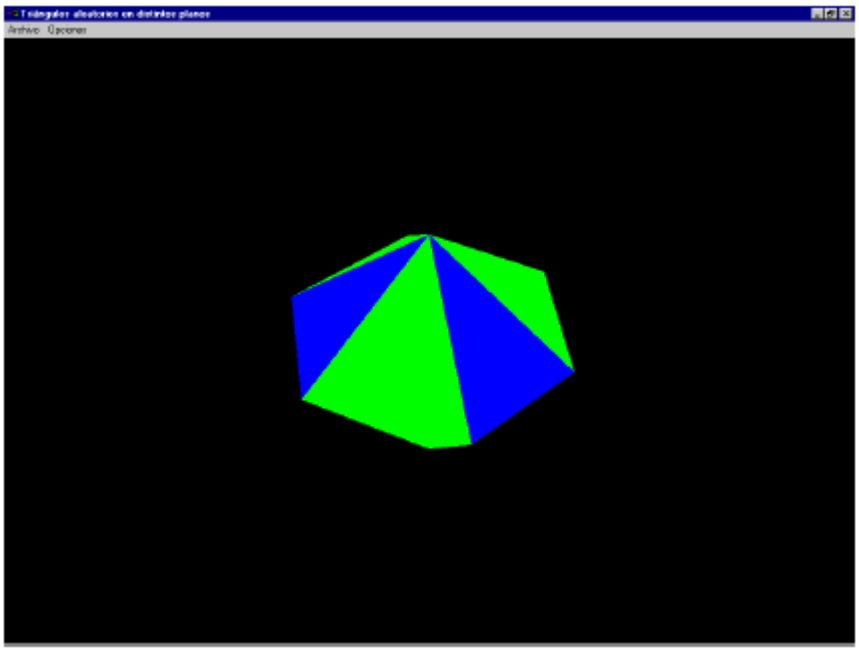


Ilustración 23: Resultado de la ejecución del programa

Se observa que la cara verde frontal tiene una séptima cara pequeña adyacente del mismo color. Esto se debe a que la pirámide realmente va a ser la primera aproximación hacia un cono con base obviamente circular.

Iremos aproximando ese cono de forma similar a como se hizo en el ejemplo tercero de líneas, pero esta vez con triángulos, para que quede demostrada la gran utilidad de este tipo de polígonos.

Éste es el código que dibuja y aproxima el cono:

```
glBegin(GL_TRIANGLE_FAN);
float x,x1;
float y,y1;
bool alt=true;
glColor3f(0.0f, 1.0f, 0.0f);
glVertex3f(.0f,.0f, 50.0f);
for(float ang=0.0f; ang<(2.0f*GL_PI); ang+=precision)
{
   if(alt) glColor3f(.0f,0.f,1.0f);
   else glColor3f(.0f,1.0f,.0f);
   alt=!alt;
   x=(float)50.0f*sin(ang);
   y=(float)50.0f*cos(ang);
   glVertex2f(x, y);
```

```
      if(ang==0.0)
      {
         x1=x; y1=y;
      }
   }
   //Ponemos otra vez el primer punto de la base para cerrar el
   //abanico con una cara irregular.
   if(alt) glColor3f(.0f,0.f,1.0f);
   else glColor3f(.0f,1.0f,.0f);
   glVertex2f(x1,y1);
   glEnd();
```

Como en el tercer ejemplo de líneas, precisión es una variable que se va ajustando con los botones de función F4 y F5.

El programa soporta las mismas opciones que el primer ejemplo de triángulos, pudiendo ver los vértices o aristas de la pirámide, cambiar el modo de color, realizar el *antialiasing*, etc.

Veamos ahora algunos de los pasos de la secuencia que nos da el cono mediante el aumento del número de caras (Ilustración 24).

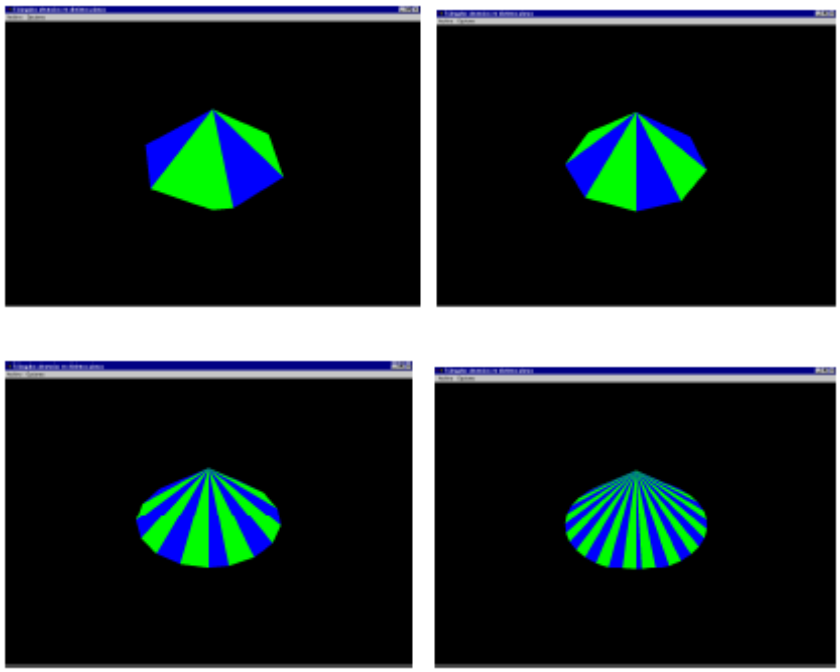


Ilustración 24: Efecto del aumento del número de caras

Ahora veamos el problema ya comentado de la ocultación de caras con figuras abiertas.

La Ilustración 25 es la imagen sin activar la ocultación de caras. Se dibujan todos los triángulos de la parte posterior, enteros.

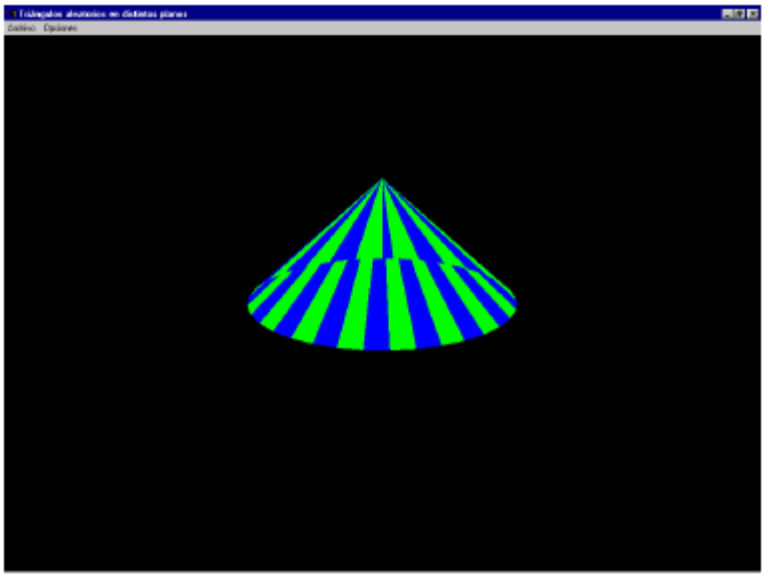


Ilustración 25: Cono sin activación de la ocultación de caras

Ahora activando la ocultación, no se dibujará ninguno encarado traseramente, no sólo lo que tapan los delanteros, sino todo el polígono. El efecto visual es peor, pero se gana rendimiento. Con figuras cerradas el efecto visual no se nota (Ilustración 26).

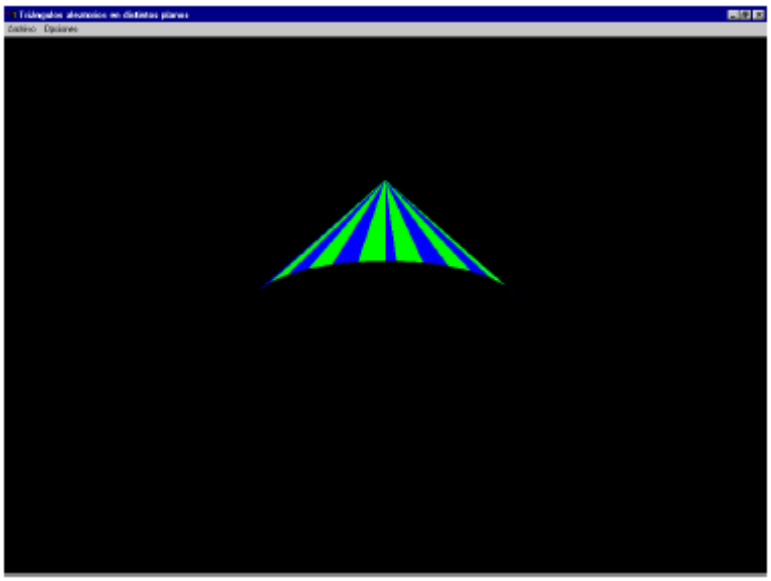


Ilustración 26: Cono con activación de la ocultación de caras

5.2.2. Cuadriláteros:

Una vez vistos los triángulos, el resto de polígonos no tiene ningún secreto. Tan sólo existen macros especiales para cuadriláteros además de para triángulos.

Hay una macro `GL_QUADS` y otra `GL_QUAD_STRIP`, con comportamiento análogo a las dos macros de triángulos:

Si tenemos nueve vértices $V_1,\ldots,V_9$:

La Ilustración 27 es el resultado con `GL_QUADS`, V_9 se ignora por no ser suficiente para crear un nuevo cuadrilátero.

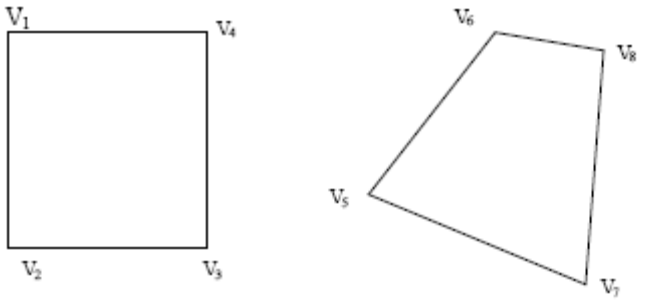


Ilustración 27: Aplicación de la primitiva GL_QUADS

Ahora vemos la tira que crea `GL_QUAD_STRIP`, que necesita cuatro vértices para el primer cuadrilátero y a partir de ahí pares de puntos que definan la nueva arista a unir. Así, V_9 es ignorada, pero si tuviéramos un V_{10} sería suficiente para definir otro cuadrilátero (Ilustración 28).

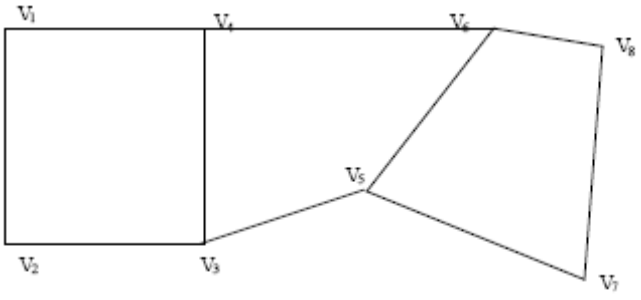


Ilustración 28 Aplicación de la primitiva GL_QUAD_STRIP

5.2.2.1. Primer ejemplo: Cuadriláteros aleatorios:

Generamos tres cuadriláteros aleatorios. Al ser las coordenadas de los cuatro vértices aleatorias, nos pueden quedar cuadriláteros complejos con las puntas en distintos planos. Para arreglar esto, hemos dejado una de las

coordenadas de las cuatro esquinas de cada cuadrilátero constante, aunque aleatorias de un cuadrilátero al siguiente.

Lo que no hemos contemplado es limitar las coordenadas a aquéllas que nos den polígonos simples convexos, así que algunos de los cuadriláteros que aparezcan en pantalla no serán demasiado correctos para OpenGL.

Por último, indicar que este ejemplo cuenta con todas las opciones en cuanto a color, aristas, etc. con las que contaban los ejemplos de triángulos.

La Ilustración 29 es un ejemplo de ejecución:

Ilustración 29: Ejemplo de dibujado de cuadriláteros

El cuadrilátero del fondo no es simple como puede intuirse, pero todos son planos.

El código correspondiente no tiene ninguna dificultad, salvo el hecho de dejar en el mismo plano los puntos de cada cuadrilátero:

```
glBegin(GL_QUADS);
for(j=0;j<3;j++)
{
   plano=puntos[j][0];
   for(k=0;k<4;k++,i++)
   {
      glColor3f(color[i][0], color[i][1], color[i][2]);
      switch(planos[j])
      {
         case 0:
            glVertex3f(plano, puntos[i][1], puntos[i][2]);
            break;
         case 1:
            glVertex3f(puntos[i][0], plano, puntos[i][2]);
            break;
         case 2:
```

```
                glVertex3f(puntos[i][0],puntos[i][1], plano);
                break;
            }
        }
    }
```

La opción de tiras nos genera una figura estática para su mejor visualización (Ilustración 30).

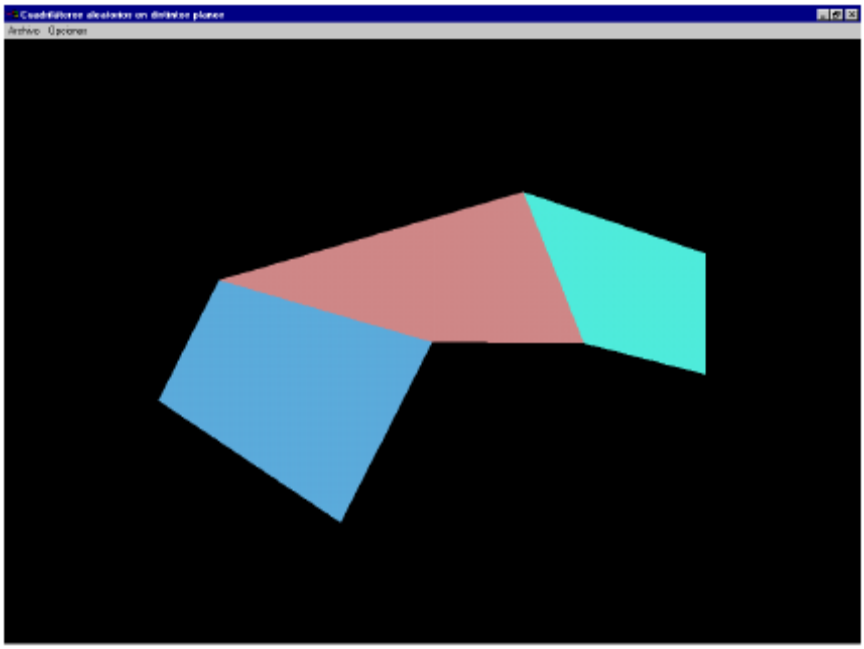


Ilustración 30: Cuadriláteros en tira

5.2.3. Poligonos

La última macro de OpenGL es `GL_POLYGON`, que puede emplearse para polígonos con cualquier número de vértices.

Aquí vemos un polígono de cinco vértices, si declaráramos entre `glBegin()` y `glEnd()` ese número de vértices. En este caso, al no tener un número concreto de vértices, OpenGL aprovechará todos los vértices que se declaren:

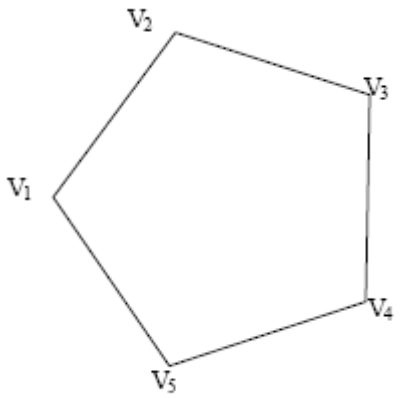


Ilustración 31: Utilización de la primitiva GL_POLIGON

Aparte de las 10 macros que ofrece OpenGL para usar con `glBegin/End`, existe una forma bastante utilizada de dibujar rectángulos, mediante la función `glRect()`, que lleva como parámetros las coordenadas x e y del punto más arriba a la izquierda y el de más abajo a la derecha. Aunque es una estrategia más limitada que `GL_QUADS` (no puede declarar coordenada z o dibujar cuadriláteros no paralelogramos), es mucho más cómoda.

5.2.3.1. Patrones:

Hay dos métodos para aplicar un patrón a polígonos sólidos. El método artesano es el mapeado de texturas, en el que se mapea un mapa de bits sobre la superficie de un polígono, como se cubre en posteriores temas.

Otra forma es especificar un patrón similarmente a como se hacía para las líneas. Un patrón poligonal no es más que un panel de bits monocromo de 32x32, usado para rellenar con patrones.

Para habilitar el patronaje de polígonos en OpenGL se llama a:

```
glEnable(GL_POLYGON_STIPPLE)
```

y posteriormente se invoca a `glPolygonStipple(pBitmap)`, en donde `pBitmap` es un puntero a un área de datos que contiene el patrón.

Este patrón es similar al que se usaba en el patronaje de líneas, exceptuando que el buffer es lo suficientemente grande como para soportar un patrón de 32x32 bits. También los bits se leen con la prioridad MSB (*Most Significant Bit*, o Bit Más Significativo), que es la contraria a la de los patrones de líneas.

Los patrones no rotan al girar las figuras en las que están inscritos, ya que el patrón sólo se utiliza para rellenar el polígono simplemente en pantalla.

5.2.3.2. Ejemplo de utilización de patrones

Como los polígonos tienen un tratamiento igual que los triángulos y los cuadriláteros, sólo vamos a mostrar ahora un ejemplo del uso de patrones. El patrón utilizado tiene el dibujo de una hoguera, y vamos a pintarlo sobre un octógono regular, obteniéndose el resultado visible en la Ilustración 33.

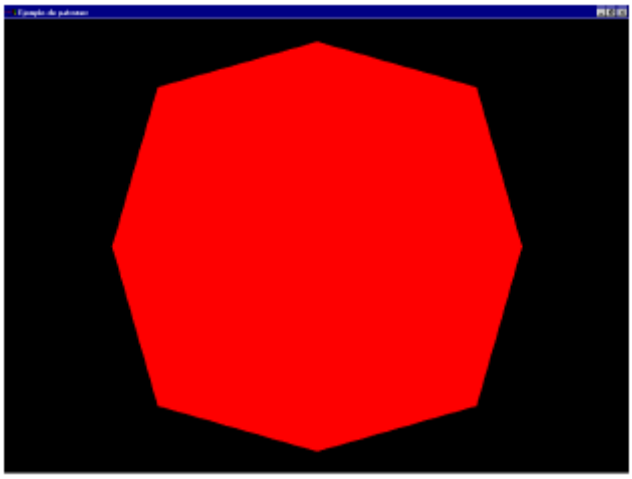


Ilustración 32: Polígono original

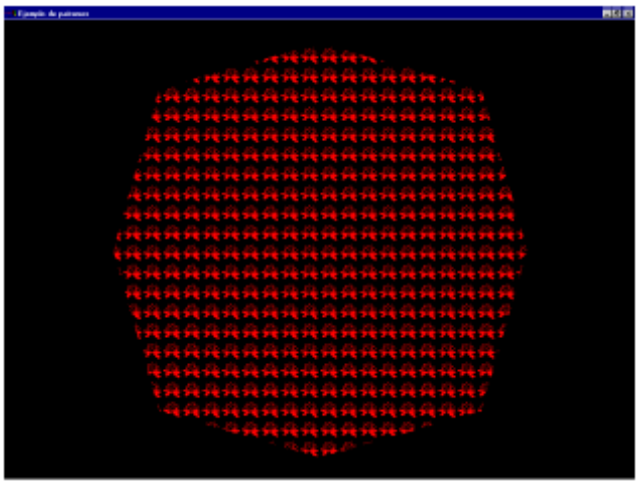


Ilustración 33: Resultado de aplicación de un patrón a un polígono

El código que permite usar el patrón de bits es el siguiente:

```
if(patron)
{
   glEnable(GL_POLYGON_STIPPLE);
   glPolygonStipple(fire);
}
else
   glDisable(GL_POLYGON_STIPPLE);
```

`patron` es una variable que cambia de valor lógico al pulsar el botón F4.

El patrón se expresa mediante una cadena de hexadecimales. En este caso la cadena es la siguiente:

```
//Un mapa de bits que dibuja una hoguera
GLubyte fire[] = { 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0xc0,
0x00, 0x00, 0x01, 0xf0,
0x00, 0x00, 0x07, 0xf0,
0x0f, 0x00, 0x1f, 0xe0,
0x1f, 0x80, 0x1f, 0xc0,
0x0f, 0xc0, 0x3f, 0x80,
0x07, 0xe0, 0x7e, 0x00,
0x03, 0xf0, 0xff, 0x80,
0x03, 0xf5, 0xff, 0xe0,
0x07, 0xfd, 0xff, 0xf8,
0x1f, 0xfc, 0xff, 0xe8,
0xff, 0xe3, 0xbf, 0x70,
0xde, 0x80, 0xb7, 0x00,
0x71, 0x10, 0x4a, 0x80,
```

```
0x03, 0x10, 0x4e, 0x40,
0x02, 0x88, 0x8c, 0x20,
0x05, 0x05, 0x04, 0x40,
0x02, 0x82, 0x14, 0x40,
0x02, 0x40, 0x10, 0x80,
0x02, 0x64, 0x1a, 0x80,
0x00, 0x92, 0x29, 0x00,
0x00, 0xb0, 0x48, 0x00,
0x00, 0xc8, 0x90, 0x00,
0x00, 0x85, 0x10, 0x00,
0x00, 0x03, 0x00, 0x00,
0x00, 0x00, 0x10, 0x00 };
```