2016/2017

Práctica 4: Elementos básicos

CREANDO INTERFACES DE USUARIOS

MARÍA GORETTI SUÁREZ RIVERO

Contenido

Introducción

En este informe se verá cómo se han desarrollado distintas tareas durante la cuarta práctica de la asignatura que nos han permitido observar distintas cosas:

- Poder ver desde un punto de vista práctico lo visto en teoría sobre las proyecciones ortográficas y la perspectiva, para modificar los puntos de vista desde los que vemos el objeto y las diferencias entre usar las distintas proyecciones.
- El factor *alpha* que nos permite trabajar con la transparencia cuando usamos RGBA. Podremos ver el efecto de usar transparencia junto con los colores en las figuras.
- Cómo trabajar con vértices y triángulos mediante cintas de triángulos a las que vamos restando y sumando vértices para ver los efectos que se producen. Trabajaremos también con el relleno y el *anti-aliasing* mientras hacemos uso de polígonos en OpenGL.
- Cómo trabajar con figuras 3D haciendo uso de la perspectiva y del z-buffer, el cual nos permite trabajar con un buffer de profundidad.
- Por último, se verá cómo pintar haciendo uso de arrays de vértices.

Desarrollo

Para esta práctica se han desarrollado un total de tres tareas.

Tarea 1

En esta primera tarea hemos tratado la perspectiva elemental, o sea, una visión básica de perspectiva, para empezar a entrar en contacto con ella, y también el Z-buffer.

La práctica consiste básicamente en dibujar dos cuadrados: uno rojo cercano al usuario, y otro azul que estará más lejos. Lo que se pretende ver es el efecto que causa tener un tipo de proyección u otra, ya que cuando usemos una proyección ortográfica, ambos se superpondran, y veremos únicamente el rojo. Sin embargo, cuando usemos la perspectiva, ambos aparecerán con distinto tamaño, apareciendo el azul dentro del rojo. Esto es debido a que el azul, al estar más lejos, se verá más pequeño. Realmente en tamaño no es más pequeño, es el efecto de la profundidad.

Para realizar esta tarea, creamos un proyecto en Visual Studio con la misma configuración establecida en las prácticas anteriores, incluyendo correctamente las librerías de OpenGL necesarias. Su directorio de includes se estableció como “C:/OpenGL/include”, y su directorio de librerías como “C:/OpenGL/lib”. Además, en su sección de Entrada dentro de las preferencias del Vinculador se añadieron las librerías “freeglut.lib” y “glew32.lib”.

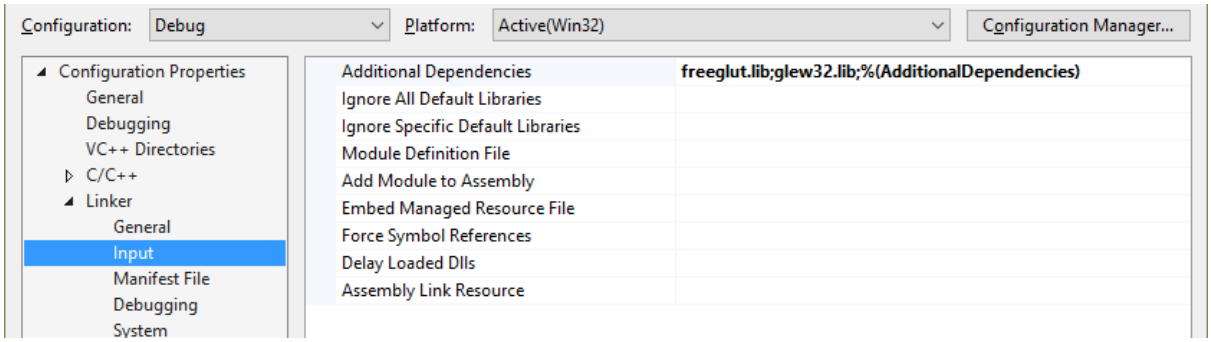

Figura 1: Configuración del proyecto de Visual Studio

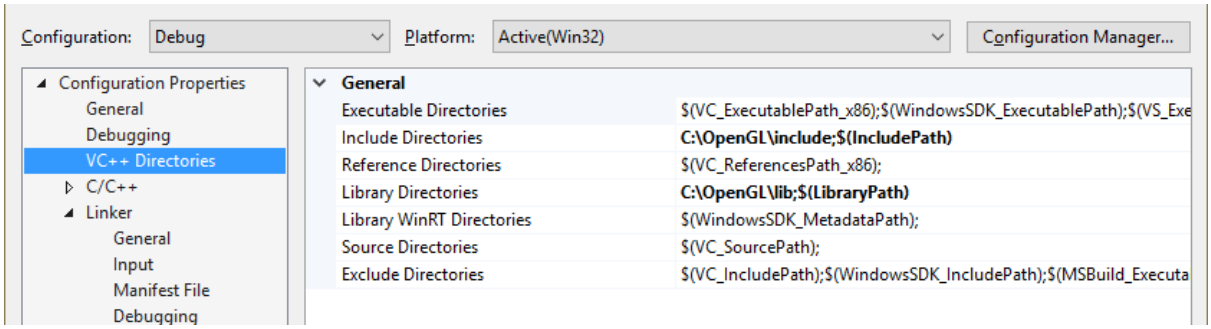

Figura 2: Configuración de archivos include y lib del proyecto Visual Studio

En el fichero .cpp creado para esta tarea, puede verse cómo se incluyen las librerías *glew.h* y *freeglut.h*, así como *stdio.h*, tal y como se ha hecho en proyectos anteriores y necesarias para poder utilizar las funciones de OpenGL que necesitamos. Tras ello, creamos las variables globales necesarias que se emplean en el resto de la tarea. En este caso, establecimos unas variables llamadas *gl_ancho* y *gl_alto*, ambas a 2, que definirían el tamaño de las figuras a usar. De la misma manera, definimos *gl_cerca* y *gl_lejos*, siendo sus valores 2 y 4 respectivamente. Estas variables representan los valores de profundidad, o sea, los valores sobre el eje Z sobre los que se definirían las figuras, tal y como se puede ver en *Figura 3*.

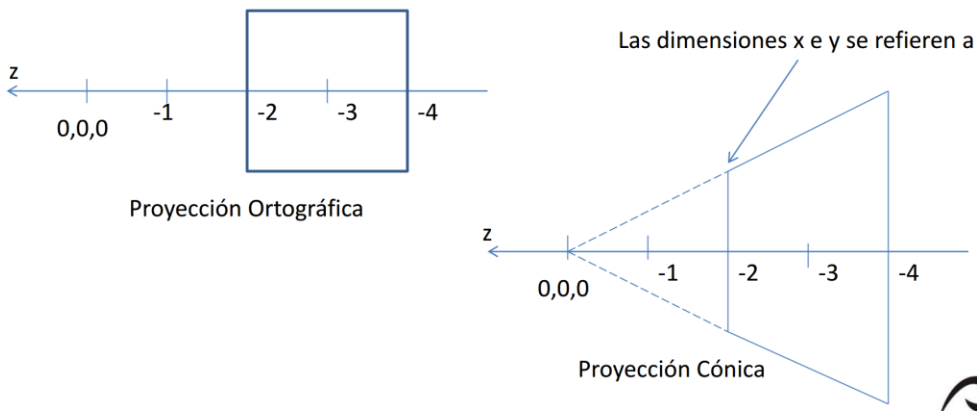

Figura 3: Explicación gráfica de los valores de las variables globales.

A parte, se definieron unas variables llamadas *w_ancho* y *w_alto* que determinan el tamaño de la ventana de la aplicación.

En el código, podemos ver como se define la función *InitGlew()*. Esta función ha sido cogida de las prácticas anteriores sin modificación alguna, y se encarga de inicializar glew, para poder usar transparentemente todas las extensiones de OpenGL.

Bajo ella, puede verse la definición de la función *Init*(), que en su interior, inicializa el entorno de trabajo. Primero comprueba que la relación de aspecto esté correcta, comprobando las proporciones de tamaño entre las variables globales previamente definidas. Tras ello, pone el fondo de la ventana a negro con *glClearColor(0.0, 0.0, 0.0, 0.0)* y habilita el test de buffer de profundidad para poder usarlo en esta práctica con *glEnable(GL_DEPTH_TEST)*. Definiremos que usaremos el modelo de proyección mediante *glMatrixMode(GL_PROJECTION)* y cargaremos la matriz identidad desde la que partirán todas las operaciones matriciales que se realicen. Finalmente, definiremos las dos proyecciones: la ortogonal y la de perspectiva. Cuando ejecutemos, ejecutaremos activando una de ambas, para ver los efectos que causan. Para definirlas, usamos los mismos parámetros en ambas, salvo que para definir la proyección ortogonal usamos las función *glOrtho()* y para elegir la proyección de perspectiva usamos *glFrustum()*.

Ahora, definimos la función de *Display()*, en la cual lo primero que hacemos es limpiar tanto el buffer de color como el de profundidad. Luego, pasamos a dibujar los dos cuadrados alrededor del punto central, de -3. Definimos como color del primero el rojo, y dibujamos los vértices sobre el punto -2.5 del eje Z. Luego, definimos el azul como color del segundo y dibujamos los vértices sobre el punto -3.5 del eje Z. Una vez dibujados los dos cuadrados, actualizamos el framebuffer con glFlush().

La función de *reshape* que viene a continuación, es la misma que la usada en las anteriores prácticas. Para saber cómo se va a redimensionar, hay que partir de la situación inicial, y es que la ventana se creará con unas dimensiones físicas de 500x500 y proyectará unas dimensiones lógicas de 2x2. Por tanto, para mantener la correcta proporción cuando las dimensiones físicas cambien tanto su ancho como su alto, se deberán implicar unas dimensiones lógicas dX y dY que se calculan de la siguiente manera:

- Si ancho > alto: dY=2 y dX=2*ancho/alto
- Si alto > ancho: dX=2 y dY = 2*alto/ancho

El cambio viene a la hora de definir el espacio ortogonal o de perspectiva que se utilizará en esta práctica. A diferencia de cómo era en las anteriores prácticas, en este caso se llamará a la función de la misma manera y con los mismos parámetros como se hizo en *Init()*. Por tanto, cuando queramos usar la proyección ortogonal, se usará *glOrtho(-dx / 2.0, dx / 2.0, -dy / 2.0, dy / 2.0, gl_cerca, gl_lejos)* , mientras que cuando queramos usar perspectiva, se usará *glFrustum(-dx / 2.0, dx / 2.0, -dy / 2.0, dy / 2.0, gl_cerca, gl_lejos)*. Luego, volveremos a dibujar las figuras para que esté correctamente redimensionadas.

Para finalizar, configuramos la función principal *main()* de manera parecida a como se ha hecho en las anteriores prácticas. Inicializamos glut, establecemos el tamaño y posición de la ventana, y definimos el modo de visualización, que en este caso incluirá GLUT_DEPTH para habilitar el z-buffer y que pueda ser usado. Tras eso, creamos la ventana, llamamos a las funciones de inicialización y establecemos las funciones *Display()* y *ReshapeSize()* como las funciones de *Display* y de *Reshape* que tendrá la aplicación. Finalmente, entramos en bucle con glutMainLoop().

Cuando ejecutamos la tarea con la proyección ortogonal, podemos ver como solo se dibuja el cuadrado rojo, como en *Figura 4*.

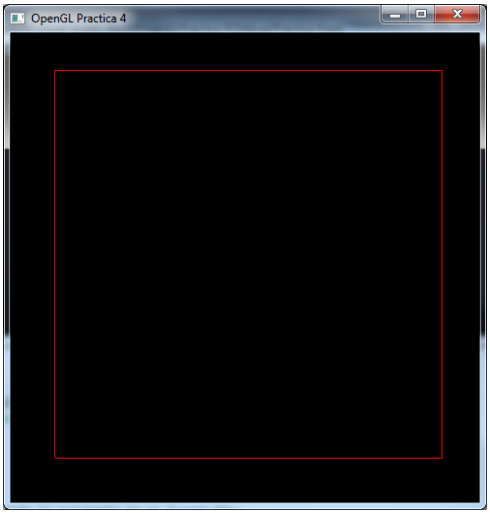

Figura 4: Proyección ortogonal

Sin embargo, cuando ejecutamos habilitando la perspectiva, nos es posible ver el cuadro azul que hemos dibujado con mayor lejanía, como puede verse en *Figura 5*.

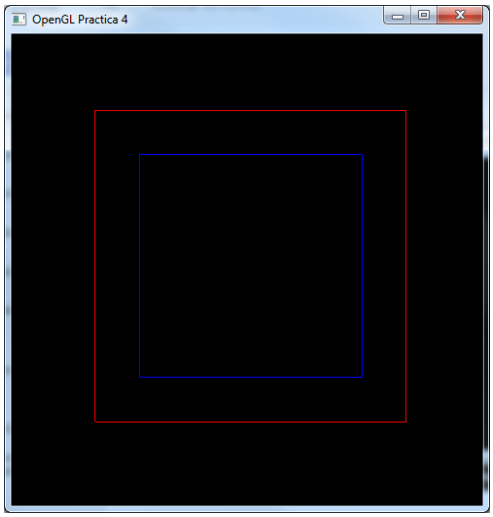

Figura 5: Con perspectiva

Es el Z-buffer el que nos permite poder obtener un resultado como el de *Figura 5,* almacenando con un cierto nivel de precisión la información de la profundidad de los pixel de los fragmentos. Cuando se intenta dibujar un pixel que en profundidad está más alejado, se rechaza esta posibilidad. En caso de que el pixel esté más cercano se produce un reemplazo.

Tarea 2

En esta segunda tarea se trabaja principalmente con la transparencia de los colores,, es decir, se nos presenta el factor alfa. Este factor influirá en cuán opaco o transparente es un color, permitiéndonos

visualizar otro color que pueda estar tras él en base a la transparencia del primero. Con la transparencia, lo que se produce es una mezcla de colores:

$$C_{final} = \alpha C_{fuente} + (1-\alpha) C_{fondo}$$

Figura 6: Fórmula de mezcla de colores

Para visualizar un ejemplo de como funciona esto, realizaremos un proyecto en Visual Studio que, utilizando OpenGL, dibuje dos triángulos en parte superpuestos. Ambos triángulos tendrán distinta profundidad, y el más cercano será parcialmente transparente, siendo casi opaco a la izquierda y progresivamente más transparente hacia la derecha.

Con la intención de facilitar la realización de la tarea, copiaremos el proyecto de Visual Studio anteriormente creado en la tarea 1, ya que posee la configuración necesaria para esta tarea (la configuración de OpenGL en Visual Studio es idéntica). Una vez hecho esto, renombramos el fichero .cpp y realizamos los cambios respectivos a esta práctica.

La importación de las librerías *glew.h*, *freeglut.h* y *stdio.h* se mantendrá, ya que siguen siendo necesarias, así como las variables globales que ya estaban definidas y el método *InitGlew()*. Las diferencias con respecto a la anterior tarea empiezan a aparecer en la función de inicialización *Init()*.

En el método de inicialización *Init()* seguimos comprobando primeramente si la relación de aspecto está correctamente definida, así como estableciendo el negro como color de fondo y habilitando el test de profundidad para poder trabajar con la profundidad. OpenGL no tiene una interfaz directa para trabajar con la transparencia, por lo que para crear este efecto lo que hacemos es habilitar la mezcla de colores y definir cómo será la función de mezcla nosotros mismos. Para esto, habilitamos la mezcla con *glEnable(GL_BLEND)* y definimos la función con *glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)*. Después de que la mezcla esté activada de esta manera, el color de entrada se mezclará con el color que esté almacenado en el *framebuffer*. La función definida en *glBlenFunc()* controla cómo sucede esta mezcla, y típicamente lo hace como lo define la fórmula mostrada en *Figura 6*. Tras especificar el uso de transparencia, especificamos el modo proyección y cargamos la matriz identidad como ya estaba en el código, y luego escogemos la proyección ortogonal, dejando la llamada a *glOrtho()* especificada, es decir, *glOrtho(-gl_ancho / 2.0, gl_ancho / 2.0, -gl_alto / 2.0, gl_alto / 2.0, gl_cerca, gl_lejos)*, trabajando de esta manera con la proyección ortogonal.

El método de *Display()* si presentará más modificaciones para realizar esta tarea con respecto al que teníamos de la tarea previa. Empezamos de la misma manera, borrando todo lo que pueda haber en el *framebuffer* de color y de profundidad. Tras eso, dibujamos primero el triángulo más alejado, que será rojo y opaco, por lo que definimos primero su color rojo y luego lo dibujamos, estableciendo que será un triángulo con *glBegin(GL_TRIANGLES)* y definiendo dónde está cada uno de sus vértices usando para cada uno *glVertex3f()*. Luego, dibujamos el triángulo cercano azul que será parcialmente transparente. Para ello, antes del primer vértice, definimos el color con 4 variables, las 3 RGB y por último, el factor alfa que definirá la transparencia. En el caso del primer vértice, el más cercano al centro y el cual será el más transparente, se definirá así su color: *glColor4f(0.0, 0.0, 1.0, 0.4)*. Cuanto más cerca esté de valer 0 el último valor, el que representa al factor alfa, más transparente será el color. Así, cuanto más cerca esté de valer 1, más opaco será. Tras definir el primer vértice, y antes de definir los otros dos siguientes, se establece el nuevo el color azul, pero esta vez con un factor de 0.9. De esta manera, desde el vértice

central hasta los otros dos, el color irá pasando de más transparente a menos, reduciéndose la transparencia.

Después de haber dibujado los dos triángulos, actualizamos el *framebuffer*.

Bajo la función de Display(), encontramos la función de reshape, la cual se mantendrá como en la primera tarea, salvo que nos quedaremos concretamente con el modo ortogonal utilizando glOrtho() tras cargar la matriz identidad dentro de la función. Finalmente, tendremos la función main, que realizará las mismas funciones que en la anterior tarea.

Podemos ver en Figura 7 el resultado de la ejecución del código de esta tarea, y cómo se dibujan los dos triángulos, presentando el azul, el más cercano, una cierta transparencia que nos permite ver a través de él el color rojo del triángulo más lejano.

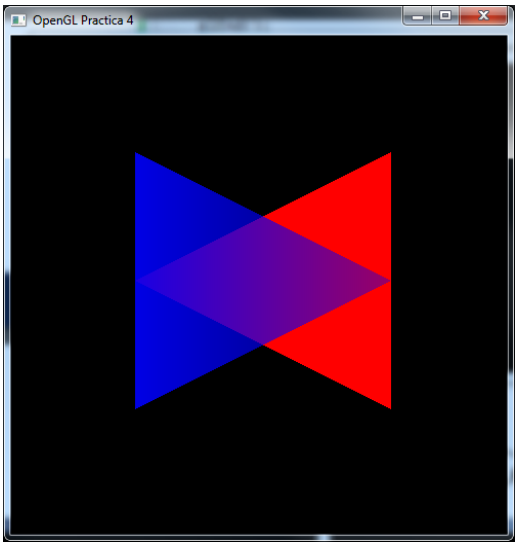

Figura 7: Resultado de la ejecución de la tarea 2

Tarea 3

En esta tercera tarea lo que hemos hecho ha sido dibujar una figura a base de crear una cinta de triángulos, dibujando progresivamente nuevos vértices, empezando por un único primer triángulo, en una cinta hasta obtener dicha figura completa.

Para esta tarea, y como se hizo en la segunda, copiamos el proyecto de la segunda tarea ya existente, con el fin de no tener que realizar de nuevo toda la configuración de OpenGL en el proyecto de Visual Studio y de aprovechar las funciones ya definidas que pueden ser de utilidad. En este caso, todas las funciones definidas en la tarea dos serán utilizadas tal y como están para la tarea 3, salvo la de display, que será la única que presente cambios.

En la nueva función Display() comenzamos borrando, como siempre, todo lo existente en el framebuffer con *glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)*. Hecho esto, definimos que la cinta de triángulos que procederemos a dibujar tendrá como color el rojo con *glColor3f(1.0, 0.0, 0.0)*. Es aquí cuando empezamos a dibujar la cinta. Comenzamos definiendo que será una cinta de triángulos con glBegin(GL_TRIANGLE_STRIP) y comenzamos a definir los vértices, cada uno con glVertex2f(). Una vez

escritos todos los vértices, se concluirá el dibujo de la cinta con glEnd() y se actualizará el framebuffer con glFlush().

Teniendo hecho el resto de funciones de la anterior tarea, definiendo siempre la proyección ortogonal, ejecutamos y vemos como, si todos los vertices están correctamente señalados, se forma una figura con la cinta de triángulos. Esta figura puede verse en *Figura 8*.

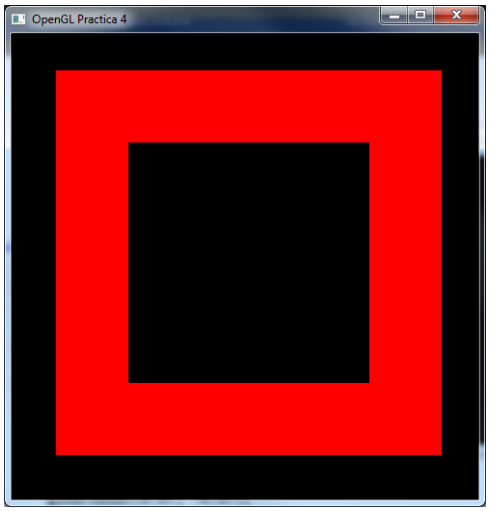

Figura 8: Visualización de la cinta de triángulos completa

Si quitamos vértices y, por tanto, quitamos triángulos que componen la cinta, vemos como la figura empieza a carecer de ciertas partes. Si, por ejemplo, quitamos los dos últimos vértices, veremos que el resultado es el obtenido en *Figura 9*. En cambio, si no solo quitamos los dos últimos sino también los dos primeros, veremos que el resultado es el de *Figura 10*. Cuantos menos triángulos, menor será la cinta.

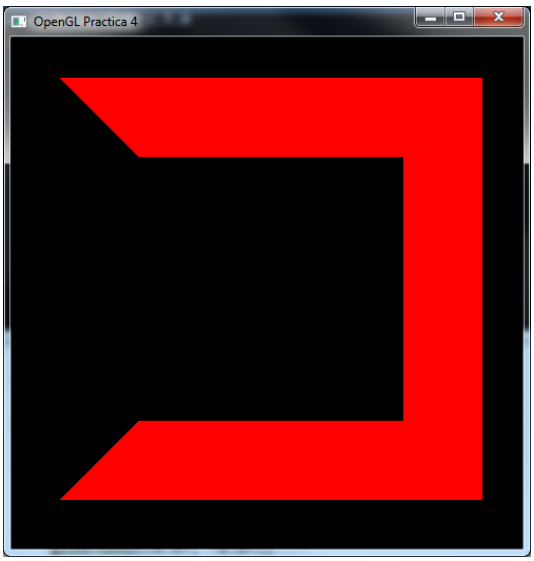

Figura 9: Cinta sin los dos últimos vértices definidos

```
void Display() {

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // borra

    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_TRIANGLE_STRIP);
        glVertex2f(-0.5, 0.5);
        glVertex2f(-0.8, 0.8);
        glVertex2f(0.5, 0.5);
        glVertex2f(0.8, 0.8);
        glVertex2f(0.5, -0.5);
        glVertex2f(0.8, -0.8);
        glVertex2f(-0.5, -0.5);
        glVertex2f(-0.8, -0.8);
        //glVertex2f(-0.5, 0.5);
        //glVertex2f(-0.8, 0.8);
    glEnd();

    glFlush();

}
```

Código 1: Función de visualización

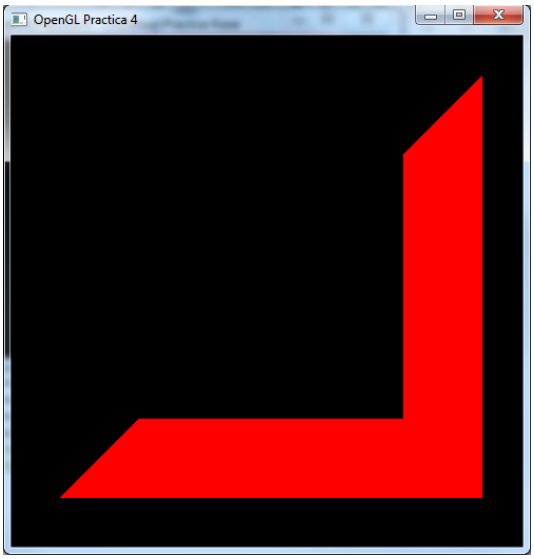

Figura 10: Cinta sin los dos primeros y últimos vértices definidos

```
void Display() {

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // borra todo lo existente en el framebuffer

    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_TRIANGLE_STRIP);
        //glVertex2f(-0.5, 0.5);
        //glVertex2f(-0.8, 0.8);
        glVertex2f(0.5, 0.5);
        glVertex2f(0.8, 0.8);
        glVertex2f(0.5, -0.5);
        glVertex2f(0.8, -0.8);
        glVertex2f(-0.5, -0.5);
        glVertex2f(-0.8, -0.8);
        //glVertex2f(-0.5, 0.5);
        //glVertex2f(-0.8, 0.8);
    glEnd();

    glFlush();

}
```

Código 2: Función de visualización modificada

Tarea 4

En esta cuarta tarea partiremos del trabajo realizado en la tercera tarea, realizando ligeras modificaciones en el proyecto para poder así controlar la forma de especificar el tipo de información de contorno/relleno. En esta tarea podremos definir si queremos o no anti-aliasing en los puntos y las líneas, y si en los polígonos que se forman queremos visualizar solo sus puntos, sus líneas, o el relleno.

Para realizarla, copiamos el proyecto de la tarea anterior, ya que disponía de todo lo necesario para realizar esta tarea, y con el fin de cumplimentar los objetivos de esta tarea, modificamos la función *Display()*.

Comenzamos la función limpiando el *framebuffer* de color y de profundidad al igual que anteriormente y definimos el color rojo para la figura que dibujaremos.

Tras esto, establecemos las funciones necesarias para que, en caso de activarlas, se pueda establecer el *anti-aliasing* de puntos (*glEnable(GL_POINT_SMOOTH)*), con un tamaño de 10 para estos (*glPointSize(10.0f)*). Luego, establecemos también las funciones necesarias para permitir, en caso de desearlo, el *anti-aliasing* de líneas, utilizando el *glEnable* como en los puntos pero en este caso con el parámetro GL_LINE_SMOOTH. Se estableció el tamaño de líneas a 1 con *glLineWidth(1.0f)*. Hecho esto, controlamos cómo será la interpretación de los polígonos para la rasterización con *glPolygonMode(face, mode)*. Como primer parámetro usaremos siempre GL_FRONT_AND_BACK, para que siempre los modos de rasterización se apliquen tanto a los polígonos que estén tanto al frente como por atrás. En nuestra tarea, tendremos tres llamadas con tres modos distintos, que al descomentar nos permitirán visualizar los polígonos de una manera u otra:

- *glPolygonMode(GL_FRONT_AND_BACK, GL_POINT)*: Nos permitirá visualizar únicamente los puntos.
- *glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)*: Nos permitirá solo visualizar las líneas.
- *glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)*: Visualizamos solo el relleno.

Hecho esto, definimos la cinta de triángulos tal y como hicimos en la anterior tarea, habilitando todos los vértices, y actualizando el framebuffer tras ello.

La primera ejecución se realizó habilidando solo lo relativo al anti-aliasing en los puntos y *glPolygonMode(GL_FRONT_AND_BACK, GL_POINT)*, visualizando así únicamente los puntos como puede verse en *Figura 11*.

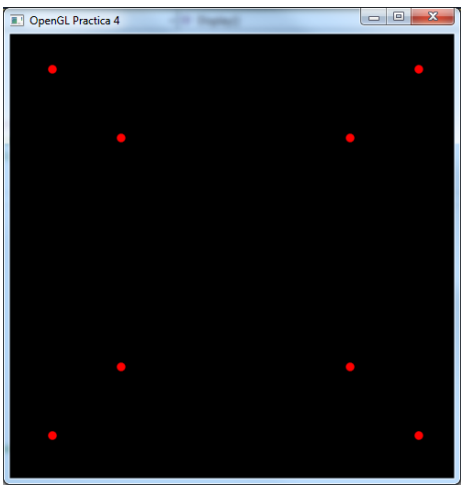

Figura 11: Visualización de los puntos con anti-aliasing

En una ejecución, se dejó lo anterior habilitado y además se habilitó el anti-aliasing en las líneas, y se habilitó la visualización de estas descomentando *glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)* y comentando el anterior habilitado para los puntos. El resultado puede verse en *Figura 12*.

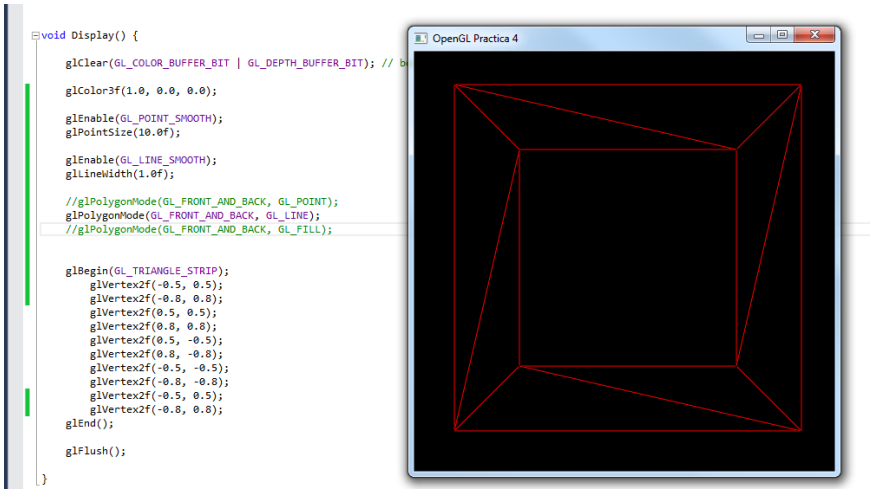

Figura 12: Visualización de líneas con anti-aliasing

Por último, se realizó una tercera ejecución en la que se comentaron los dos primeros glPolygonMode() y se dejó activo el correspondiente a la visualización del relleno: *glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)*. El resultado obtenido es el que puede verse en *Figura 13*.

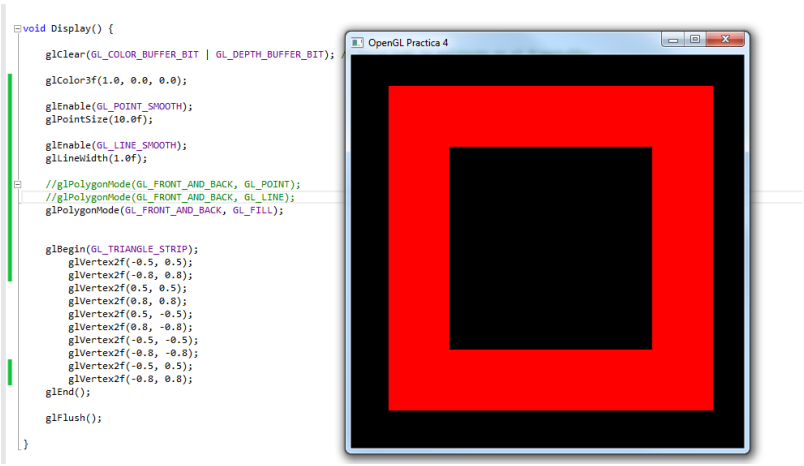

Figura 13: Visualización con relleno

Tarea 5

En esta quinta tarea se realizará el dibujo de una figura en 3D con la inclusión de perspectiva y del Z-buffer, ambas cosas usadas en tareas previas, y necesarias para eliminar la visualización de partes ocultas. La figura en 3D a dibujar será una compuesta por cuatro caras de un cubo, cada una con su

respectivo color que las distinga de las demás, y serán dibujadas mediante combinaciones de triángulos (dos triángulos por cuadrado), usando cintas como practicamos en las dos tareas anteriores.

Para llevar a cabo la tarea, hemos copiado el proyecto de la tarea anterior, debido a que necesitábamos la misma configuración para esta tarea, y hemos procedido a modificar el fichero .cpp interno para que funcione de acuerdo con las necesidades de esta tarea. A parte de conservar la importación de las librerias (*stdio.h*, *glew.h* y *freeglut.h*), hemos conservado también las variables globales definidas con los siguientes valores:

```
// Espacio para las variables globales de la ventana
float gl_ancho = 2.0, gl_alto= 2.0, gl_cerca=2.0, gl_lejos=4.0;
int w_ancho=500, w_alto=500;
```

Código 3: Variables globales

Estas variables seguirán siendo utilizadas para definir el espacio de trabajo (*gl_ancho, gl_alto, gl_cerca* y *gl_lejos*), así como el tamaño de la ventana a utilizar (*w_ancho*, *w_alto*).

La definición de la función *InitGlew()* se mantiene intacta, y en la función *Init()* usada para la inicialización, aparte de comprobar la relación de aspecto, poner el fondo negro y habilitar el test de profundidad, así como establecer el modo de proyección y cargar la matriz identidad, se establece que se incluya el caso de perspectiva, habilitando la función *glFrustum(-gl_ancho/2.0, gl_ancho/2.0, -gl_alto/2.0, gl_alto/2.0, gl_cerca,gl_lejos)*.

Tras definir estas funciones, se define la función *Display()*. En esta función, como siempre, comenzamos limpiando los *buffers* de color y profundidad con *glClear()*. Después de esto, comenzamos a dibujar cada cara de la figura 3D. La primera cara que se dibuja es la roja, estableciendo este color con *glColor3f()* y dibujándola como una cinta de triángulos, estableciendo el *glBegin(GL_TRIANGLE_STRIP)* y, tras él, los 4 vértices del cuadrado, cada uno con *glVertex3f()*. Cuando hemos acabado, llamamos a *glEnd()*. Realizamos las operaciones con cada una de las cuatro caras, cambiando al color correspondiente en cada caso, así como los vértices según corresponda, pero manteniendo el mismo patrón de dibujo. Una vez finalizado el dibujo de todas las caras, actualizamos el *framebuffer* con *glFlush()*.

La función de *reshape* empleada sigue la misma estructura que se ha empleado hasta ahora. Sin embargo, en vez de definir un espacio de trabajo ortogonal, para este caso se definirá uno de perspectiva, llamando a la función *glFrustum(-dx / 2.0, dx / 2.0, -dy / 2.0, dy / 2.0, gl_cerca, gl_lejos)* tras cargar la matriz identidad.

Finalmente, la función *main()* se mantendrá como hasta ahora, inicializando glut, la ventana, los modos de visualización y llamando a los métodos de inicialización definidos, así como estableciendo cuáles serán las funciones de *display* y de *reshape* (serán las nuestras, las que hemos definido).

Una vez ejecutado, podemos ver en *Figura 14* el resultado, las cuatro caras del cubo, cada una con su color, y con profundidad y perspectiva. Si la perspectiva no estuviese bien definida o el test de profundidad no se activase, se producirían superposiciones entre los cuadrados, anulándose parte del efecto 3D.

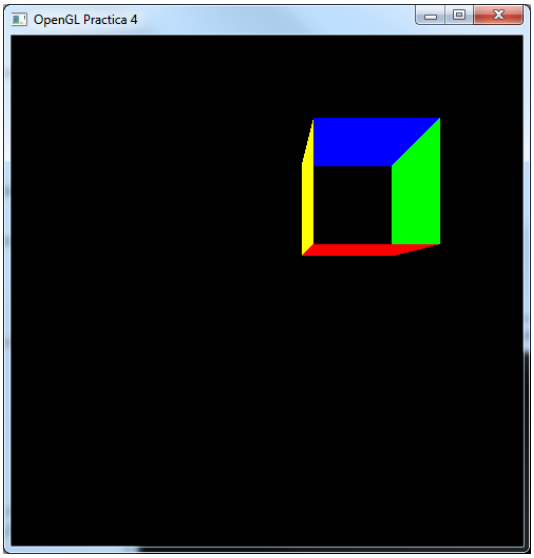

Figura 14: Visualización de figura 3D

P4_T5.cpp

(Ámbito global) Init()

```
void Init() {
    if (gl_ancho / w_ancho != gl_alto / w_alto){
        fprintf(stderr, "La relación de aspecto no es correcta\n");
    }

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glEnable(GL_DEPTH_TEST);
    //glEnable(GL_DEPTH_TEST);
    //glEnable(GL_BLEND);
    //glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    //glOrtho(-gl_ancho / 2.0, gl_ancho / 2.0, -gl_alto / 2.0, gl_alto / 2.0, gl_cerca, gl_lejos); // espacio de trabajo
    glFrustum(-gl_ancho/2.0, gl_ancho/2.0, -gl_alto/2.0, gl_alto/2.0, gl_cerca,gl_lejos);
}

void Display() {

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // borra todo lo existente en el framebuffer

    glColor3f(1.0, 0.0, 0.0); //cara roja
    glBegin(GL_TRIANGLE_STRIP);
        glVertex3f(0.2, 0.2, -2.5);
        glVertex3f(0.2, 0.2, -3.5);
        glVertex3f(0.8, 0.2, -2.5);
        glVertex3f(0.8, 0.2, -3.5);
    glEnd();

    glColor3f(0.0, 1.0, 0.0); //cara verde
    glBegin(GL_TRIANGLE_STRIP);
    glVertex3f(0.8, 0.2, -2.5);
    glVertex3f(0.8, 0.2, -3.5);
    glVertex3f(0.8, 0.8, -2.5);
    glVertex3f(0.8, 0.8, -3.5);
    glEnd();
```

Código 4: Función de inicialización y función de visualización

```
P4_T5.cpp
(Ámbito global)                                   Display()
void Display() {

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // borra todo lo existente en el framebuffer

    glColor3f(1.0, 0.0, 0.0); //cara roja
    glBegin(GL_TRIANGLE_STRIP);
        glVertex3f(0.2, 0.2, -2.5);
        glVertex3f(0.2, 0.2, -3.5);
        glVertex3f(0.8, 0.2, -2.5);
        glVertex3f(0.8, 0.2, -3.5);
    glEnd();

    glColor3f(0.0, 1.0, 0.0); //cara verde
    glBegin(GL_TRIANGLE_STRIP);
    glVertex3f(0.8, 0.2, -2.5);
    glVertex3f(0.8, 0.2, -3.5);
    glVertex3f(0.8, 0.8, -2.5);
    glVertex3f(0.8, 0.8, -3.5);
    glEnd();

    glColor3f(0.0, 0.0, 1.0); //cara azul
    glBegin(GL_TRIANGLE_STRIP);
    glVertex3f(0.2, 0.8, -2.5);
    glVertex3f(0.2, 0.8, -3.5);
    glVertex3f(0.8, 0.8, -2.5);
    glVertex3f(0.8, 0.8, -3.5);
    glEnd();

    glColor3f(1.0, 1.0, 0.0); //cara amarilla
    glBegin(GL_TRIANGLE_STRIP);
    glVertex3f(0.2, 0.2, -2.5);
    glVertex3f(0.2, 0.2, -3.5);
    glVertex3f(0.2, 0.8, -2.5);
    glVertex3f(0.2, 0.8, -3.5);
    glEnd();

    glFlush();

}
```

Código 5: Función de visualización

Tarea 6

En esta sexta y última tarea aprovecharemos lo realizado en la tarea 3 referente a las cintas de triángulos para introducir dos cosas: la codificación en arrays y la visualización colectiva. Para esta tarea, utilizaremos un array de 8 * 6, donde 8 son los vértices a dibujar, y cada uno tendrá 3 coordenadas que definen dónde están y, además, 3 coordenadas que definirán el color que posee (3 + 3 = 6). Incluiremos la definición del color en este caso para generar una representación unificada de ambos, color y coordenadas de vértices.

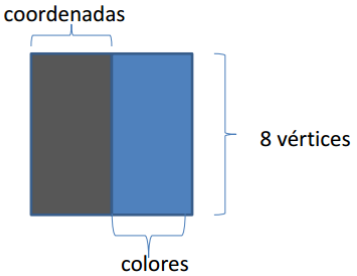

Figura 15: Definición de la matriz de puntos

Como esta tarea será una modificación de la tercera, empezaremos copiando el proyecto de la tercera, ya que poseerá la configuración necesaria así como métodos a utilizar ya definidos en el fichero .cpp. Hecho esto, en la zona de variables globales, bajo las utilizadas para establecer el espacio de trabajo, estableceremos el vector de puntos que vamos a utilizar, haciendo un vector de 8*6 de tamaño con la codificación de todos los vértices con sus coordenadas y colores. Bajo este, definimos otro vector de 10 posiciones, que tendrá la codificación de los índices del orden en el que deben ser dibujados en un TRIANGLE_STRIP.

```
5
6     float gl_ancho = 2.0, gl_alto = 2.0, gl_cerca = 0.0, gl_lejos = 4.0;
7     int w_ancho = 500, w_alto = 500;
8
9    float puntos[8 * 6] = {
10        -0.5, 0.5, 0.0, 1.0, 0.0, 0.0,
11        -0.8, 0.8, 0.0, 1.0, 0.0, 0.0,
12        0.5, 0.5, 0.0, 1.0, 0.0, 0.0,
13        0.8,0.8,0.0,1.0,0.0,0.0,
14        0.5, -0.5, 0.0, 1.0, 0.0, 0.0,
15        0.8, -0.8, 0.0, 1.0, 0.0, 0.0,
16        -0.5, -0.5, 0.0, 1.0, 0.0, 0.0,
17        -0.8, -0.8, 0.0, 1.0, 0.0, 0.0
18    };
19
20    int indices[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 0, 1 };
21
```

Código 6: Matriz de puntos y vector de índices

La única función que necesita ser redefinida con respecto al proyecto creado en la tercera tarea es la función *Display()*. Para esta tarea, crearemos dos funciones *Display()*, una que hará uso de *glDrawElements()* y otra que hará uso de *glDrawArrays()*. La primera función comienza limpiando los *buffers* de color y de profundidad. Tras esto, permitiremos que OpenGL tenga acceso al espacio de direcciones, con las funciones *glEnableClientState(GL_VERTEX_ARRAY)* y *glEnableClientState(GL_COLOR_ARRAY)*. Con esto, permitiremos que tanto los arrays de vértices como los de colores estén habilitados para escritura y uso durante el renderizado cuando, posteriormente, *glDrawElements()* sea llamado. Después de esto, definimos donde se encuentran los vectores de vértices y colores, declarando primero la variable *stride*, que define el salto entre el comienzo de datos de un vértice y otro (6 posiciones de diferencia, 3 primeras de coordenada del vértice y 3 siguientes de definición de su color), y declarando luego el número de datos por vértice, tipo de dato que posee, salto y comienzo con *glVertexPointer(3, GL_FLOAT, stride, puntos)* y declarando lo mismo con respecto a los colores con *glColorPointer(3, GL_FLOAT, stride, &(puntos[3]))*. Tras la definición de todos estos datos, se dibuja con la llamada *glDrawElements(GL_TRIANGLE_STRIP, 10, GL_UNSIGNED_INT, indices)*, y se actualiza posteriormente el *framebuffer*.

En la segunda función de *display* que poseeremos, se comienza igualmente limpiando los buffers de color y profundidad, y se realizan las mismas funciones que en el anterior *display*: se permite de la misma manera que OpenGL tenga acceso al espacio de direcciones y se define donde se encontrarán los vectores de vértices y colores con las mismas funciones. Sin embargo, en vez de utilizar *glDrawElements()*, se usará *glDrawArrays(GL_TRIANGLE_STRIP, 0,10)*, la cual, a diferencia de la anterior, no considera la información de índices, sólo los arrays. En este caso, no es capaz de cerrar la figura, dado que requiere los primeros vértices. Una vez realizado el dibujo, se actualiza el *framebuffer*.

Si ejecutamos utilizando la primera función *Display()*, veremos que se dibuja correctamente la misma cinta de triángulos que se dibujó en la tercera tarea. Sin embargo, esta ha sido dibujada usando *glDrawElements()*. El resultado puede verse en *Figura 16*.

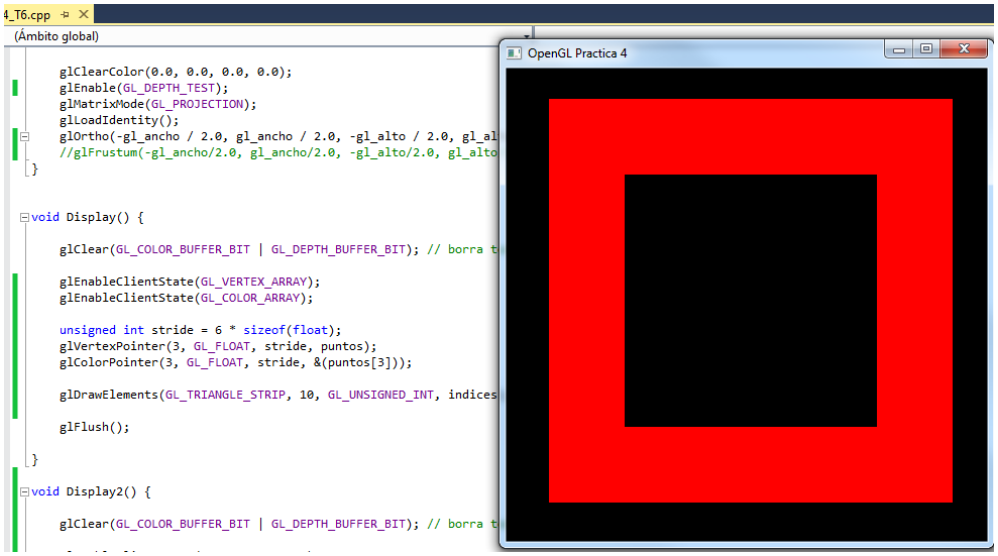

Figura 16: Dibujo de cinta de triángulos usando glDrawElements()

Si utilizamos la segunda definición de *Display()*, veremos, como se comentó antes, que carecemos de un lado de la cinta al hacer falta definir los primeros vértices, no pudiéndose completar la figura y obteniendo la imagen que vemos en *Figura 17*.

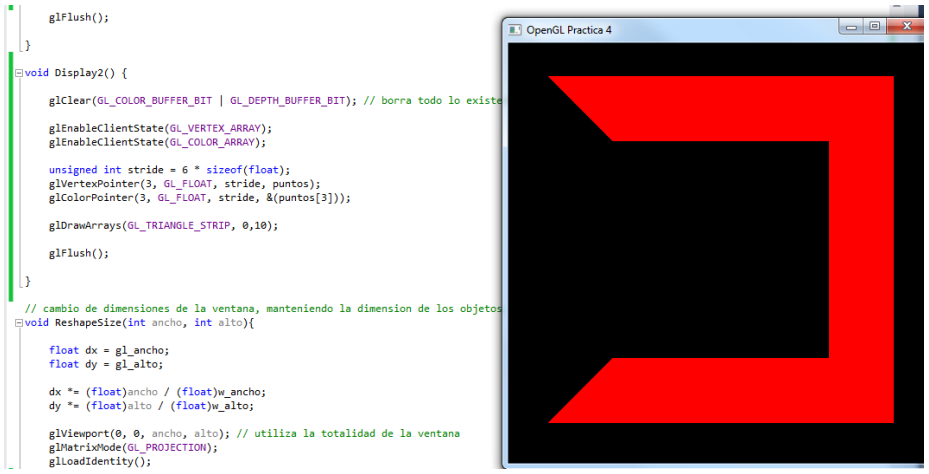

Figura 17: Dibujo de cinta de triángulos usando glDrawArrays()

Conclusión

En esta práctica, tal y como se ha podido ver durante el desarrollo de las tres tareas, hemos podido ver numerosas cosas, desde cómo funcionan las proyecciones ortográficas y la perspectiva, pasando por el manejo del Z-buffer para trabajar con profundidad y la importancia del factor alpha, hasta cómo trabajar con vértices y triángulos, así como con el relleno y el anti-aliasing en polígonos en OpenGL. Además, hemos reconstruido figuras 3D ayudándonos de lo aprendido sobre la perspectiva y el Z-buffer.

Por último, hemos hecho uso de la codificación en arrays y de la visualización colectiva, para poder ver cómo trabajar con operaciones colectivas en OpenGL.

Bibliografía

- Documentación de la práctica (Moodle)