2016/2017

Práctica 1: Primeros pasos en OpenGL

CREANDO INTERFACES DE USUARIO

MARÍA GORETTI SUÁREZ RIVERO

Contenido

Introducción

En este informe se verá cómo se han desarrollado distintas tareas durante la primera práctica de la asignatura que nos han permitido aprender más sobre OpenGL y su uso y funcionamiento. Veremos cómo MATLAB hace uso de OpenGL con un pequeño ejemplo, para luego usar ficheros PLY con otros modelos que poder cargar y visualizar en la misma herramienta. Finalmente, acabaremos la práctica con una tarea de programación de OpenGL, usando sus librerías, utilizando C, mostrando en una pequeña ventana dos triángulos coloreados generados tras definir sus vértices

Desarrollo

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

Tarea 1

En esta primera tarea, tras comprobar que toda la instalación del software necesario estaba correctamente realizada para poder realizar las distintas tareas, procedimos a ejecutar en MATLAB ***teapotdemo***, instrucción que nos generaba una demo con el famoso modelo de la tetera en 3D para poder jugar con ella. Esto es gracias a que MATLAB usa de manera implícita OpenGL para mostrar este tipo de modelos.

La instrucción generaba una ventana donde, aparte de mostrarnos el modelo de la tetera, nos daba una barra de tareas sobre este y un menú a su derecha, para poder probar distintas configuraciones.

Lo primero que se hizo fue activar la última opción de la izquierda de la barra de tareas, que nos permitía realizar una operación de rotación de cámara alrededor del objeto, pudiendo así observarlo por distintos ángulos. Siguiendo en la barra de tareas, probamos también con el segundo botón, el de *Orbit Scene Light*, para así cambiar de posición el foco de luz que había sobre el objeto. Además, con el penúltimo botón, utilizamos también el zoom.

Tras jugar con las opciones de la barra de herramientas, empezamos a modificar los campos del menú lateral derecho. El campo *lighting* proporcionaba distintos tipos de iluminación sobre el objeto. Podemos ver como cambia la iluminación en el objeto en *Figura 1* y *Figura 2*, ya que en la primera la iluminación elegida es de tipo *flat*, y en la segunda es de tipo *phong*.

El campo *colormap* nos indica que tipo de colorido tendrá el objeto, y vemos también cómo se produce el cambio en las dos primeras figuras, pasando de utilizar un *colormap autumn* a un *colormap HSV*.

El campo *material* nos permite elegir el material del cual obtendremos sus propiedades y se simulará que el objeto 3D las posee. En *Figura 3* vemos como cambia si en vez de utilizar el material por defecto, utilizamos metal.

El campo *style* nos permite cambiar el estilo del modelo. Por ejemplo, con *wireframe* vemos los elementos del dibujo, los cuadriláteros que definen el modelo de alambre del objeto. También podríamos especificar que fuesen sólo los puntos que forman el objeto los que se mostrasen. Ambos estilos pueden verse en *Figura 4* y *Figura 5* respectivamente.

Tras todo esto, solo nos quedó cambiar la especificación de si era o no opaco el objeto, marcando o no la casilla de *transparent*, especificar o no si queríamos ver los vértices marcando o no la propiedad *edges* y cambiar la posición de la tapa de la tetera y la resolución de la tetera en sí. Esto puede verse en *Figura 6*.

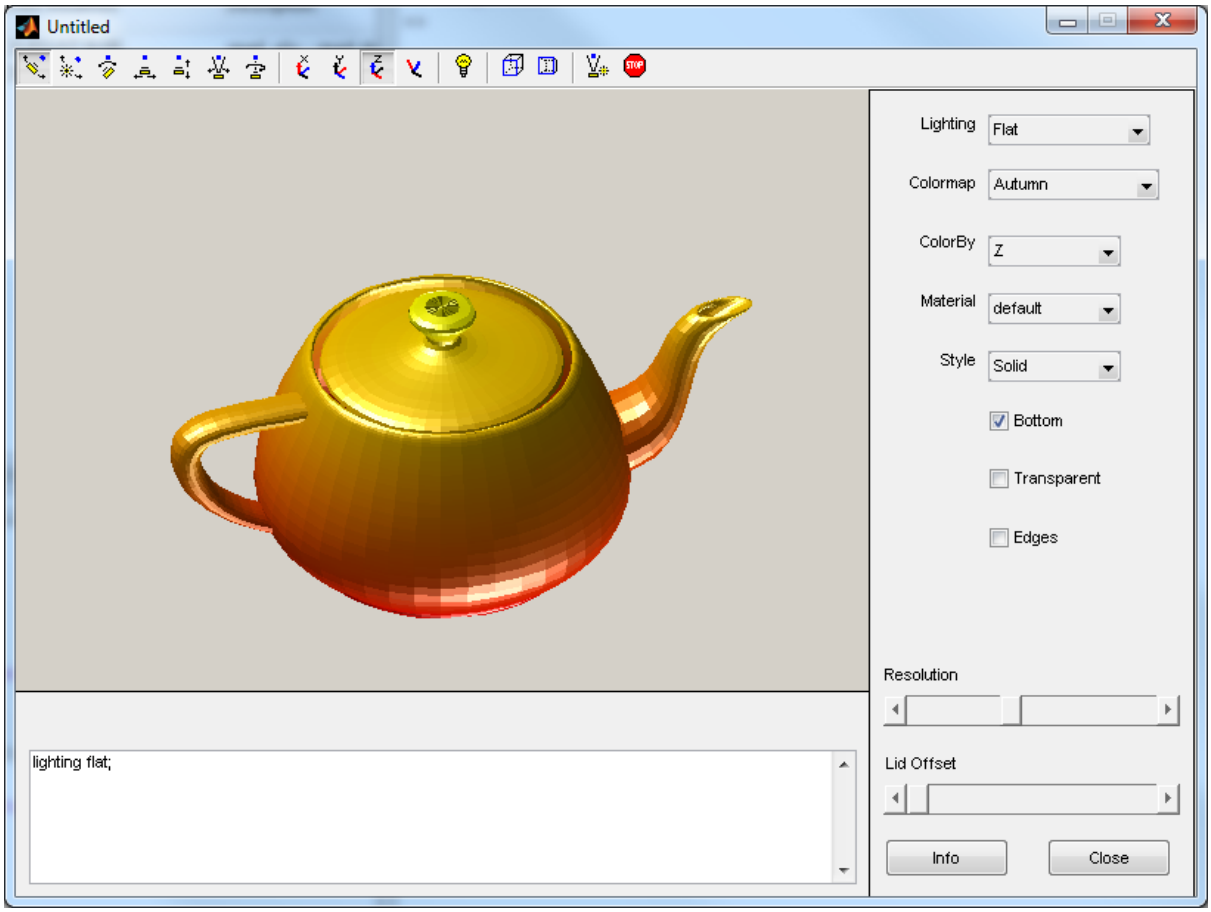

Figura 1: Modelo 3D con iluminación cambiada a *flat*.

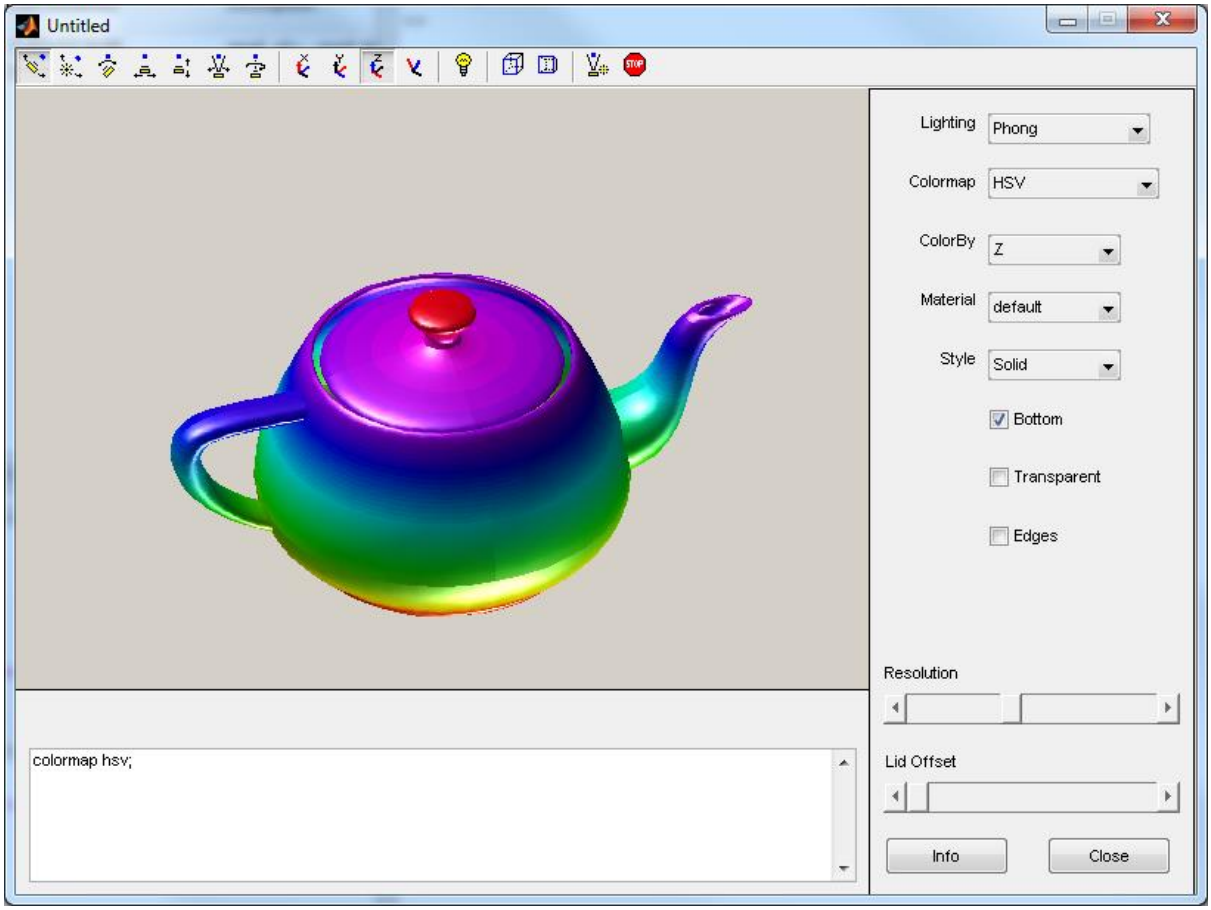

Figura 2: Modelo 3D con iluminación *phong* y distinto *colormap*

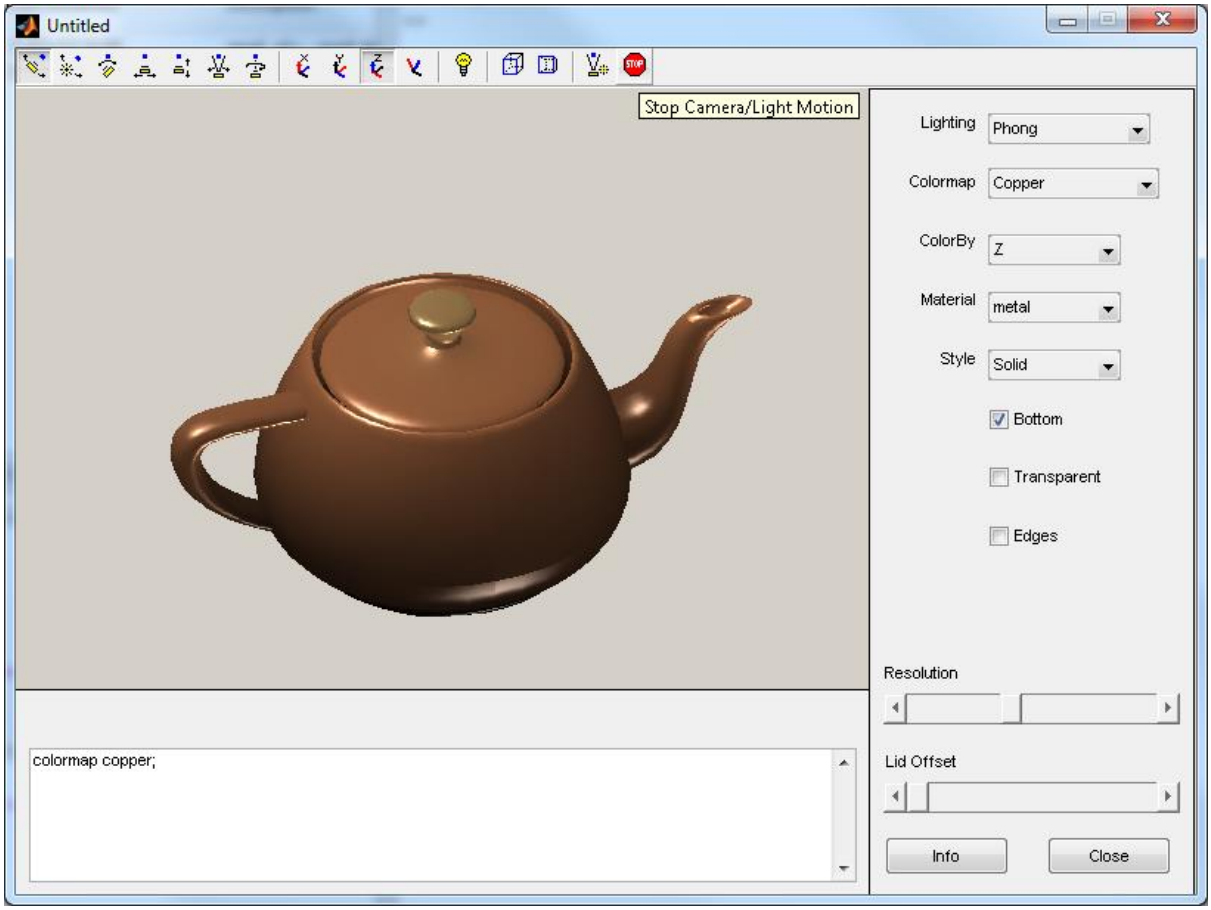

Figura 3: Modelo 3D con iluminación *phong*, *colormap copper* y que usa metal como material

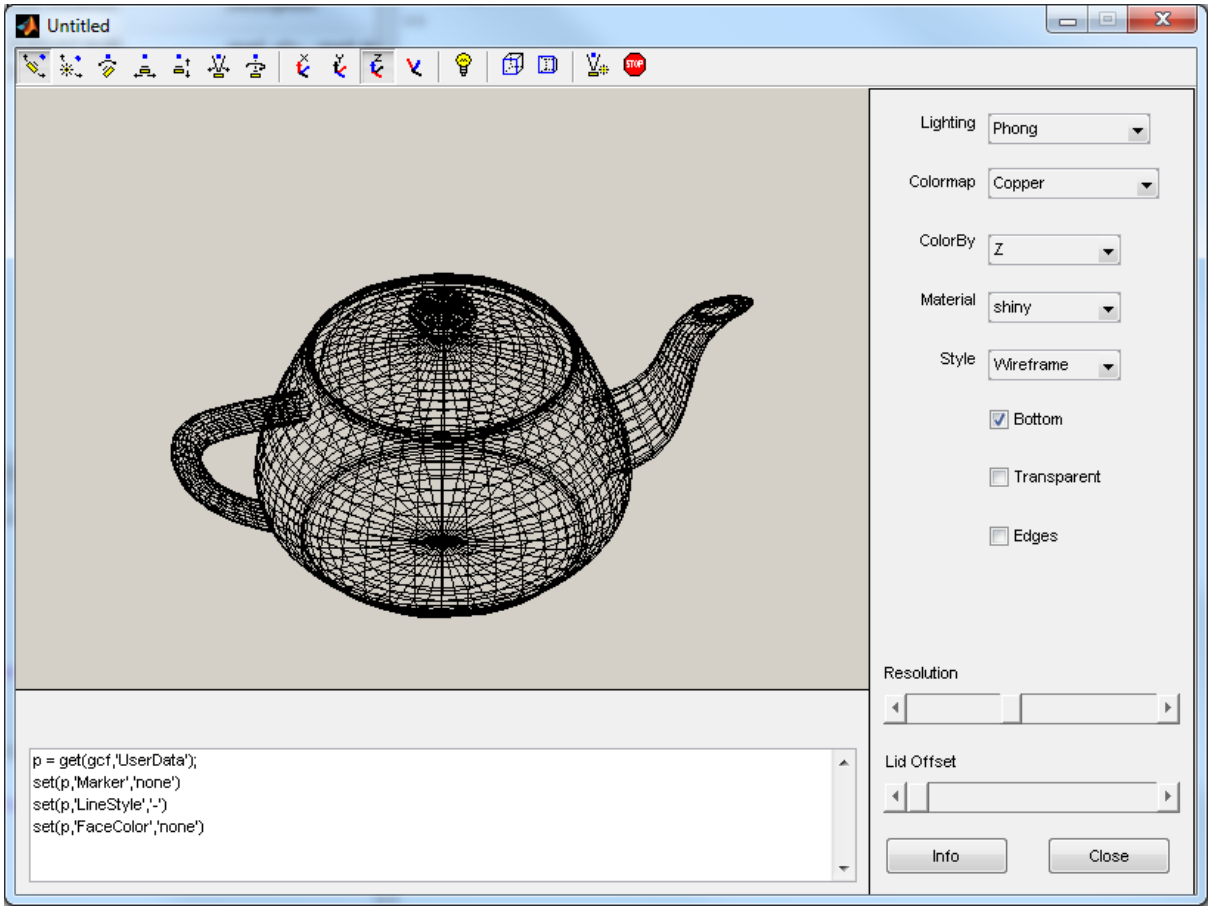

Figura 4: Modelo 3D con estilo *wireframe*

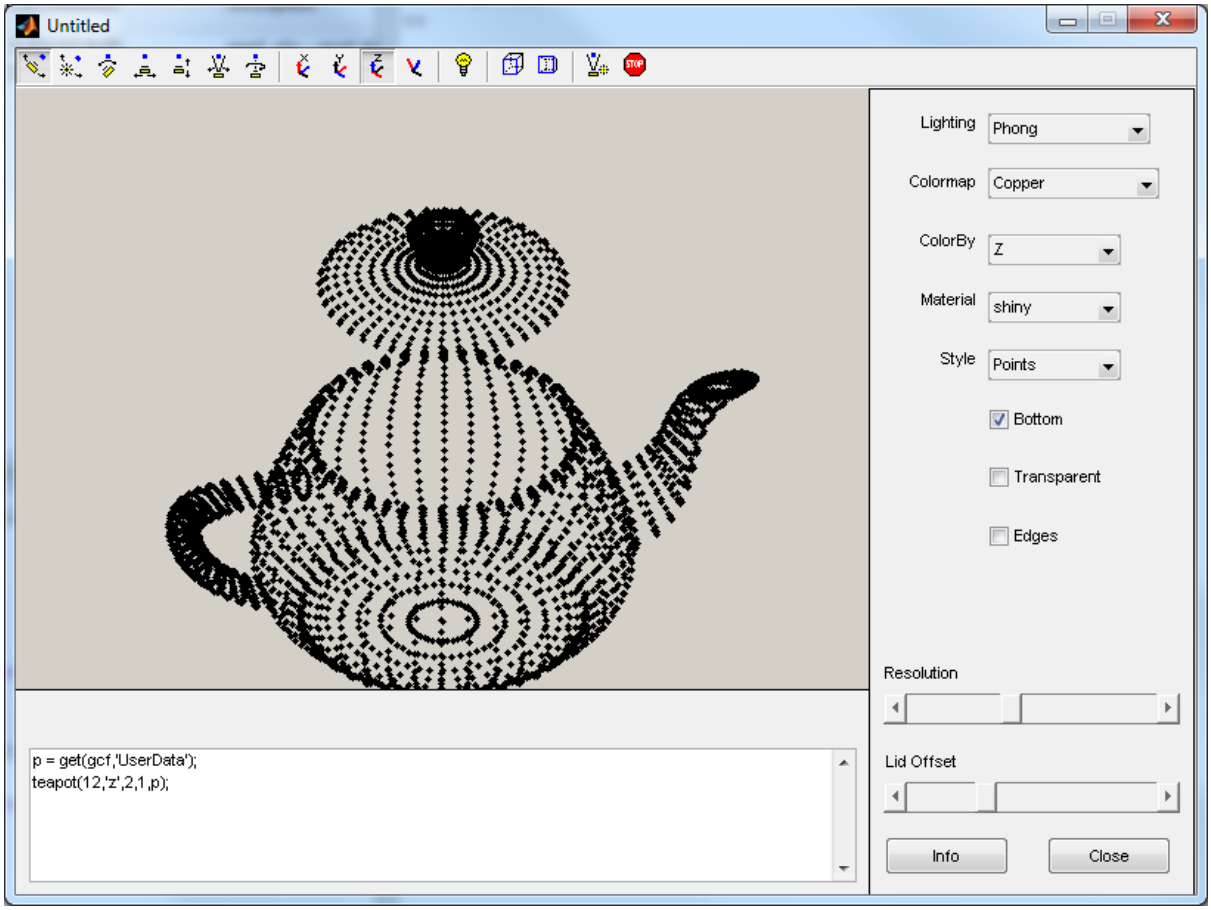

Figura 5: Modelo 3D con estilo de puntos y la tapa levantada

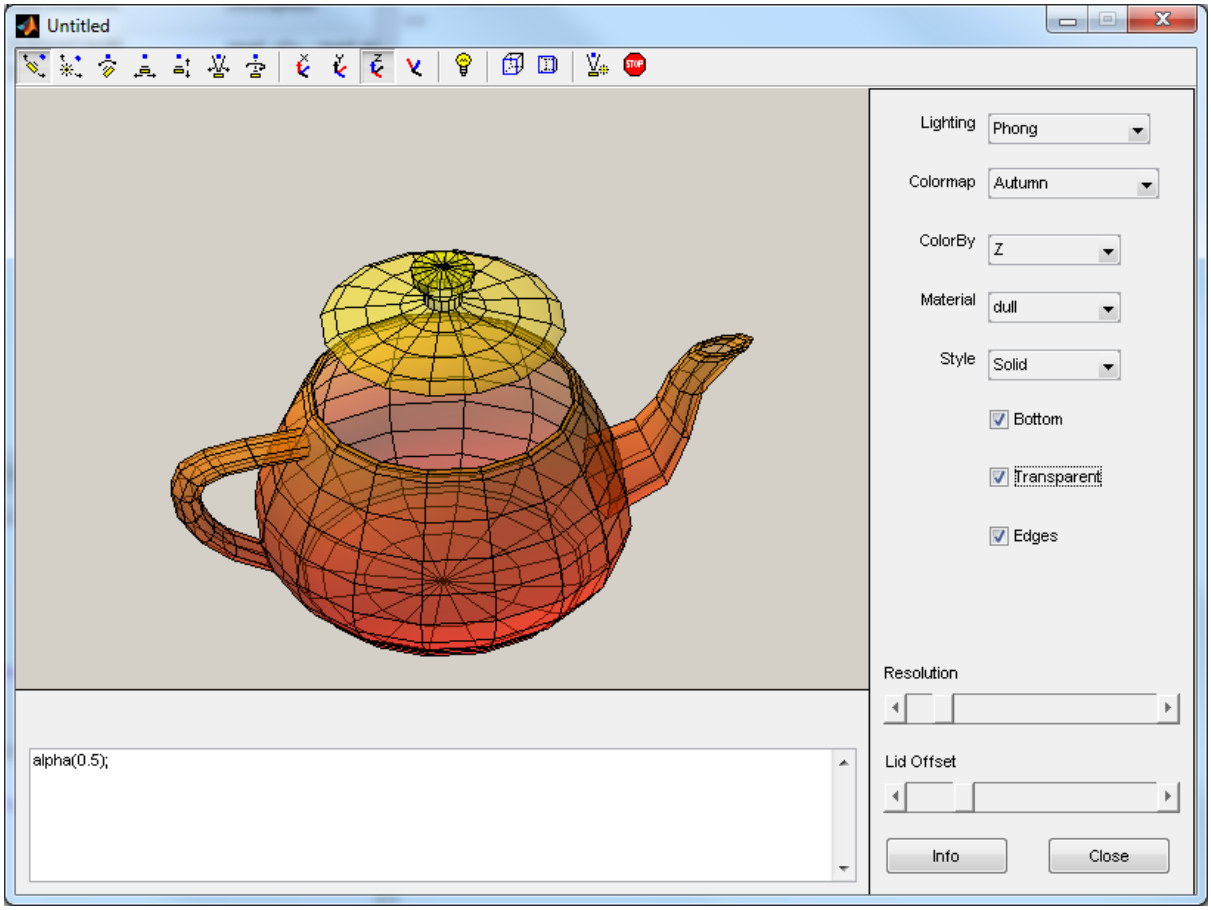

Figura 6: Modelo 3D con transparencia, muestra de vértices y tapa levantada

Tarea 2

En esta segunda tarea se nos explicó más sobre la triangulación de los objetos, y como la representación de objetos usando triángulos /cuadriláteros 3D es la base para el modelado gráfico y para la simulación mediante procedimientos numéricos. Para contener el modelo de distintos objetos existen varios tipos de ficheros, y el formato PLY es uno de los más utilizados y existen herramientas de lectura de este tipo de fichero para la mayoría de los lenguajes. En los ficheros PLY se codifica dos tipos de información: las coordenadas de los vértices y la secuencia o concatenación de puntos que determina elemento cuadrado o triangular, denominada faceta. Por tanto, aparte de más información, en un fichero PLY podemos encontrar vértices y facetas.

En esta tarea utilizamos un *toolbox* para MATLAB para poder leer este tipo de ficheros, consistente en un fichero llamado *read_ply.m*. Teniendo este fichero en nuestro espacio de trabajo, podíamos usar la instrucción:

[vertex, facet] = read_ply(filename)

Esta nos devuelve:

- *Vertex* → Una matriz Mx3 con M puntos y 3 columnas, donde cada columna representa una de las coordenadas (x,y,z).
- *Facet* → Una matriz con los elementos, de dimensión Nx3 o Nx4, donde cada fila contiene los índices de los puntos del elemento.

Y se le suministra un fichero del tipo PLY que será de donde extraiga la información.

Tras descargar un modelo 3D de una araña de la web http://people.sc.fsu.edu/~jburkardt/data/ply/ply.html , se representó en MATLAB como puede verse en *Figura 7*.

```
      4646        4538        4539
      4665        4646        4539
      4666        4665        4539
      4666        4539        4542
      4666        4542        4543
      4667        4666        4543
      4545        4667        4543
      4668        4545        4546
      4669        4668        4546
      4669        4546        4549
      4669        4549        4550
      4670        4669        4550
      4552        4670        4550

 size(vertex)

s =

      4670           3

 size(facet)

s =

      9286           3

 trisurf(facet,vertex(:,1),vertex(:,2),vertex(:,3));
 cameratoolbar;axis equal;
 title('Araña, María Goretti Suárez Rivero para CIU');
```

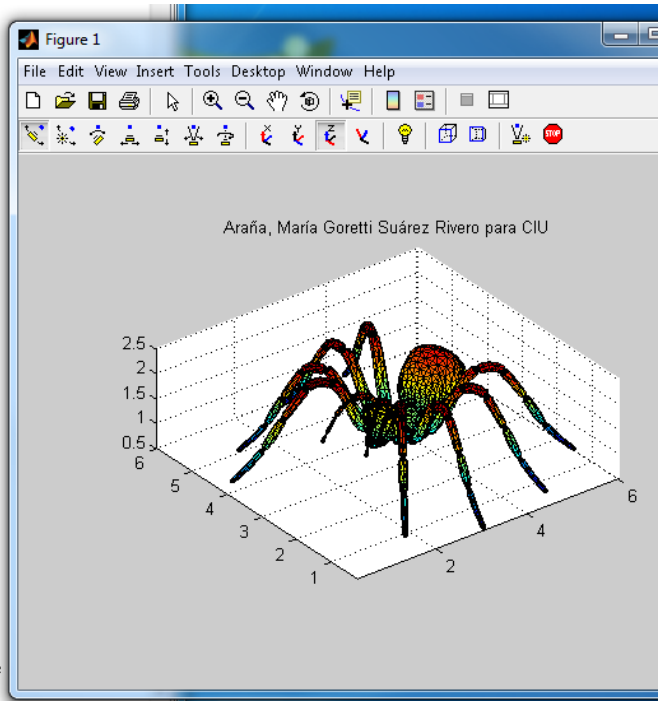

Figura 7: Representación de una araña 3D en MATLAB a partir de un fichero PLY

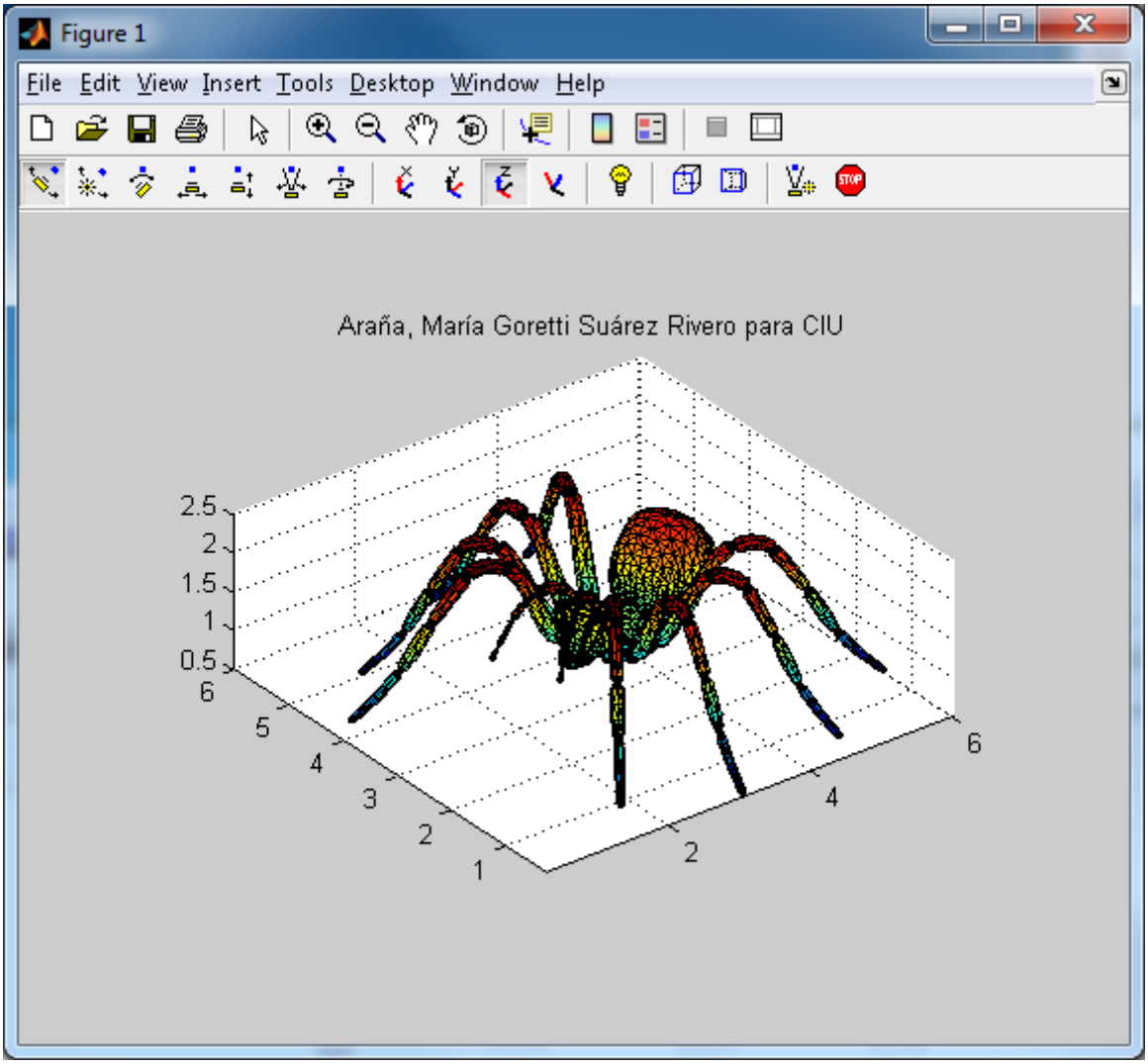

Figura 8: Mejor vista de la araña 3D

Tarea 3

En esta última tarea programamos con C en OpenGL, creando un nuevo proyecto de Visual Studio y configurándolo para que utilice las librerías OpenGL y *freeglut.lib* como se muestra en *Figura 9* y *Figura 10*.

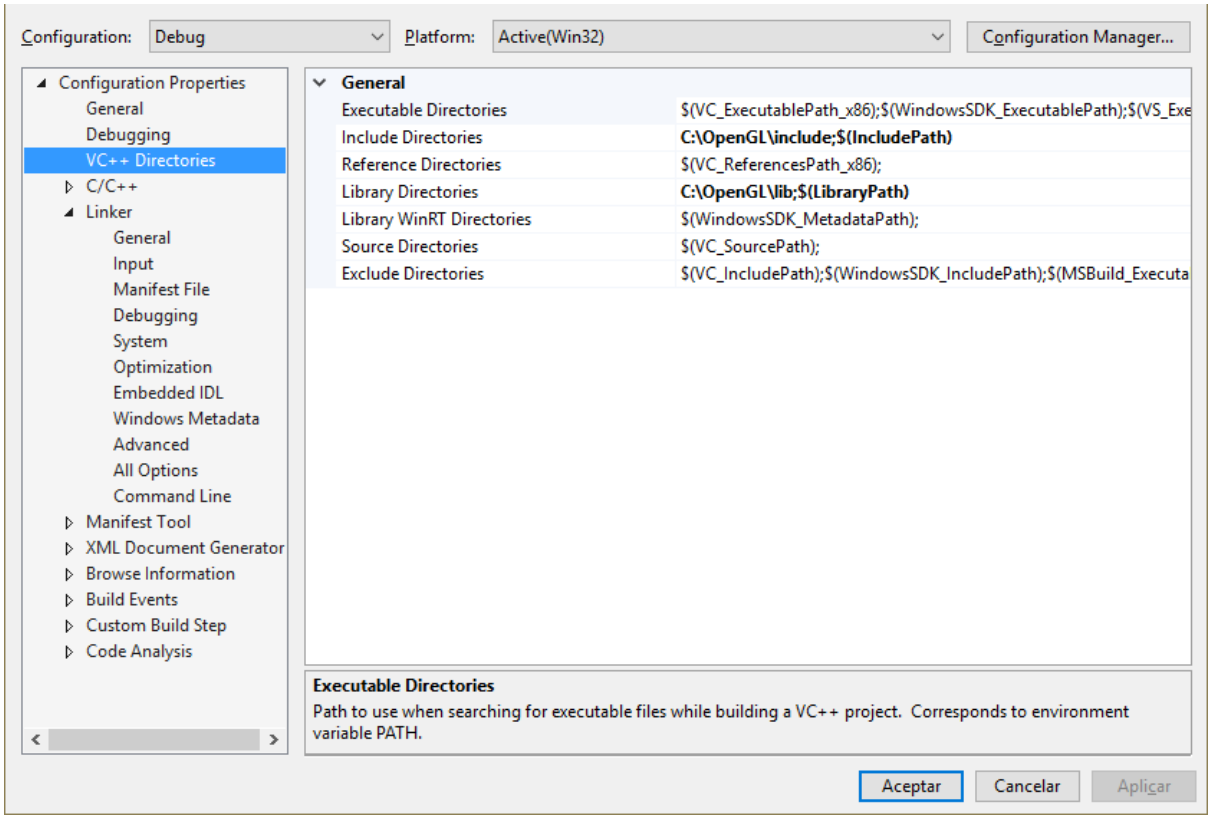

Figura 9: Configuración de las librerías OpenGL en Visual Studio

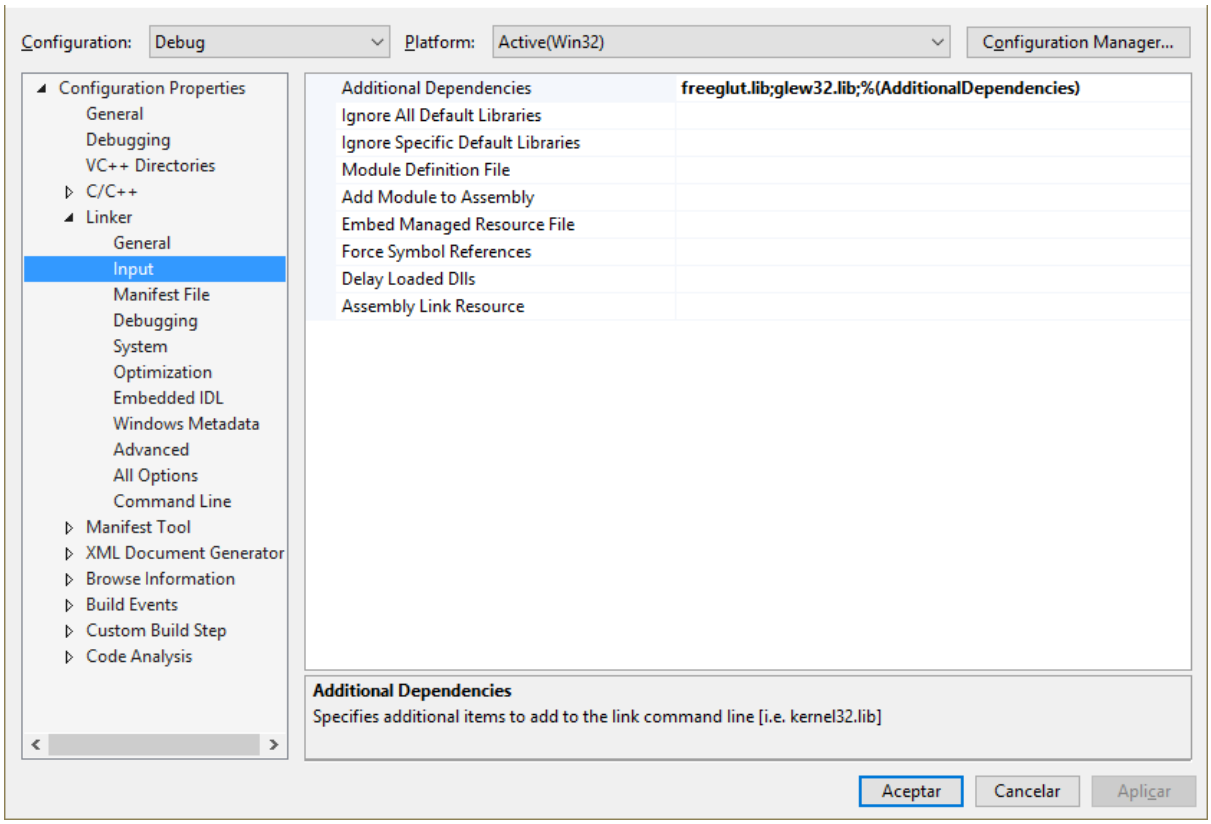

Figura 10: Configuración de freeglut.lib

El fichero *.cpp* que contendría el programa empezaría incluyendo la típica cabecera *stdio.h* y, además, la cabecera de *freeglut.h*, lo cual nos permitiría trabajar con el *toolkit* de *glut*.

Tras esto, declaramos el primer método, el método *Init()*, donde se estableció un fondo negro para la ventana y un espacio de trabajo con unas coordenadas lógicas de las dimensiones que podemos observar en *Figura 11*. Al cambiar el tamaño de la ventana, se conservarán las dimensiones, pero la relación de aspecto cambiará. No es físicamente cuadrado en la ventana final.

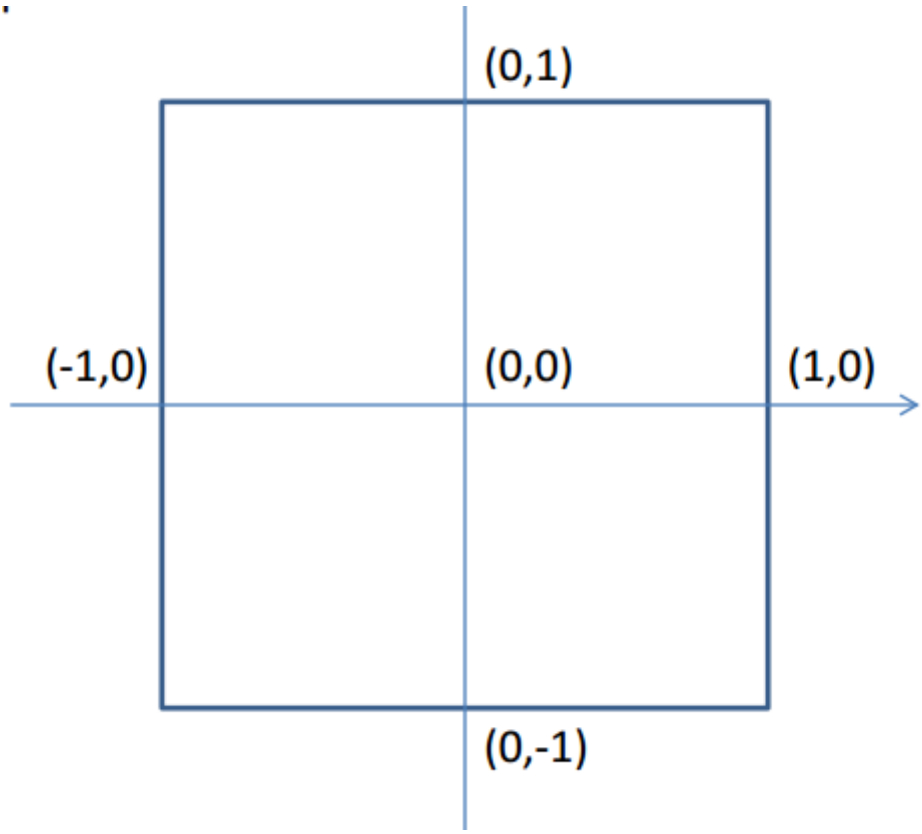

Figura 11: Coordenadas del espacio de trabajo a utilizar

Una vez definido, pasamos a definir el método *Display()*, considerado el gestor de evento de la aplicación. Este método se ejecutará cada vez que se necesite repintar la figura, y siempre se iniciara con una limpieza del buffer con *glClear()* y se acabará actualizando el buffer con *glFlush()*.

Tras limpiar el buffer, definimos lo que se haría cada vez que se llamase a este método, definiendo con *glBegin(GL_TRIANGLES)* que las figuras a dibujar serían triángulos, y comenzando a especificar los vértices con *glVertex2f(...)*, pasándole sus coordenadas en el

espacio, y sus respectivos colores con *glColor3f(...)*, poniendo en esta última las coordenadas RGB que queremos adjudicar al punto que viene a continuación de esta instrucción. Cuando la definición de los vértices de los triángulos estuvo terminada, se especificó con *glEnd()*.

Por último, se especificó la función *main()* que se encargaría de inicializar el entorno, crear la ventana con cierto tamaño y posición, especificar que se utilizaría un buffer simple pues no íbamos a usar animación en esta tarea, llamar a *Init(()* para inicializar el espacio de trabajo y, por último, especificar el uso de la función *Display()* para cada vez que se quisiese repintar la figura.

El resultado puede verse en *Figura 12* y *Figura 13*. Como vemos, aunque solo definimos la propiedad RGB de los vértices concretos, los distintos puntos del interior del triángulo también adoptan un color, obtenido por la interpolación que realizan las librerías de OpenGL.

En *Figura 14* observamos cómo se modifica el contenido si disminuimos el tamaño de la ventana.

Main.cpp

(Ámbito global)

```
#include <stdio.h>
#include <GL\freeglut.h>
//María Goretti Suárez Rivero para CIU
void Init() {
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 0.0f);
}

void Display(){

    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_TRIANGLES);
        glColor3f(1.0f, 0.0f, 0.0f);
        glVertex2f(0.0f, 0.0f);
        glColor3f(0.0f, 1.0f, 0.0f);
        glVertex2f(1.0f, 0.0f);
        glColor3f(0.0f, 0.0f, 1.0f);
        glVertex2f(0.0f, 1.0f);

        glColor3f(1.0f, 0.0f, 0.0f);
        glVertex2f(0.0f, 0.0f);
        glVertex2f(-1.0f, 0.0f);
        glVertex2f(0.0f, -1.0f);
    glEnd();
    glFlush();

}

int main(int argc, char *argv[]) {
    glutInit(&argc, argv);
    glutInitWindowPosition(100, 100);
    glutInitWindowSize(300, 200);
    glutInitDisplayMode(GLUT_RGBA | GLUT_SINGLE);
    glutCreateWindow("OpenGL Practica 1_1");
    Init();
    glutDisplayFunc(Display);

    glutMainLoop();

    return 0;
}
```

Figura 12: Código final

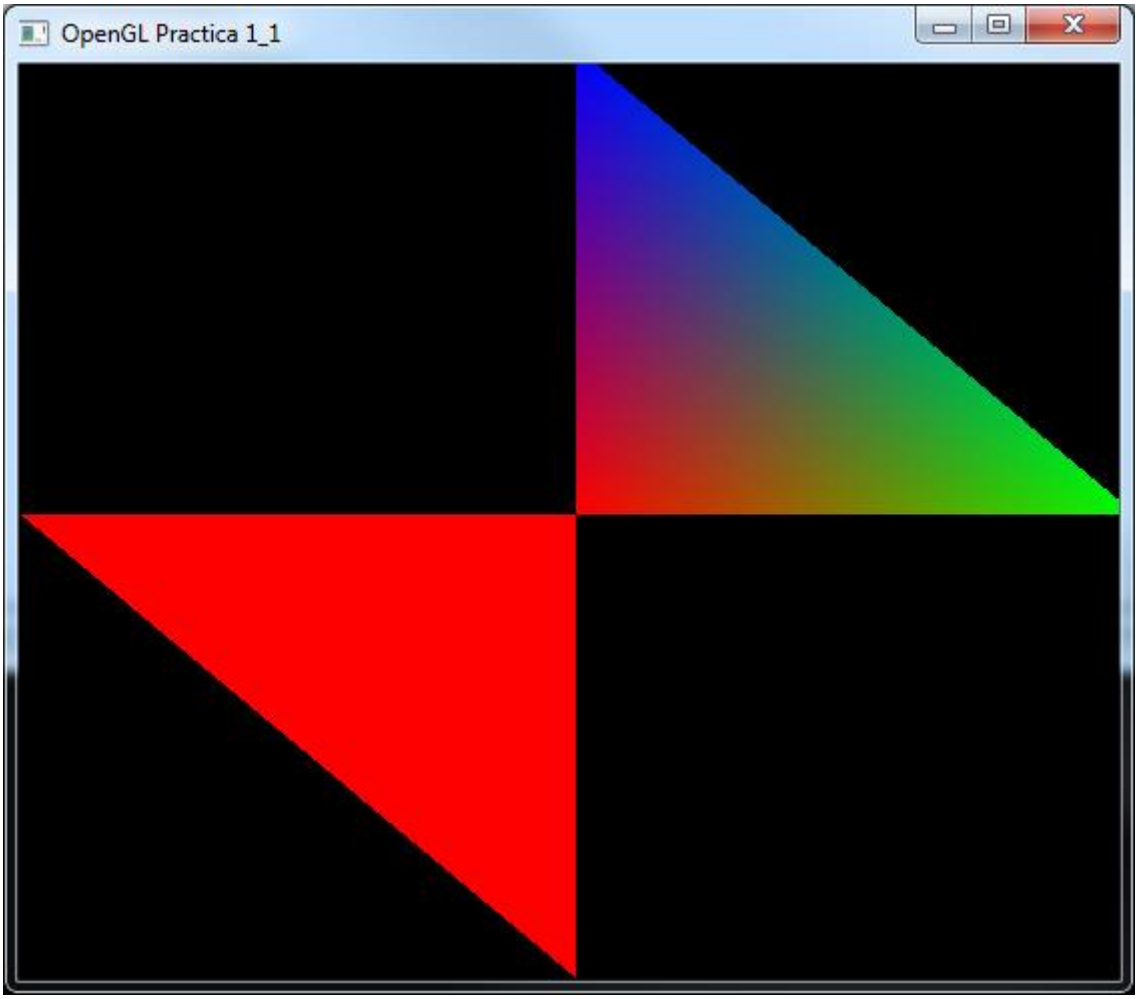

Figura 13: Resultado de la ejecución base

Figura 14: Disminución del tamaño de la ventana durante la ejecución

En *Figura 15* y *Figura 16* vemos como de base cambiamos el código para que la ventana se inicialice con un tamaño de 500x500, y el resultado con unos colores distintos, habiendo cambiado las distintas instrucciones *glColor3f(...)* como se muestra en *Figura 17*.

```
int main(int argc, char *argv[]) {
    glutInit(&argc, argv);
    glutInitWindowPosition(50, 50);
    glutInitWindowSize(500, 500);
    glutInitDisplayMode(GLUT_RGBA | GLUT_SINGLE);
    glutCreateWindow("OpenGL Practica 1_1");
    Init();
    glutDisplayFunc(Display);

    glutMainLoop();

    return 0;
}
```

Figura 15: Código cambiando en el main para distinto tamaño de ventana

```
glBegin(GL_TRIANGLES);
    glColor3f(1.0f, 1.0f, 0.0f);
    glVertex2f(0.0f, 0.0f);
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex2f(1.0f, 0.0f);
    glColor3f(1.0f, 0.0f, 1.0f);
    glVertex2f(0.0f, 1.0f);

    glColor3f(1.0f, 1.0f, 0.0f);
    glVertex2f(0.0f, 0.0f);
    glVertex2f(-1.0f, 0.0f);
    glVertex2f(0.0f, -1.0f);
glEnd();
```

Figura 16: Código para el cambio de color

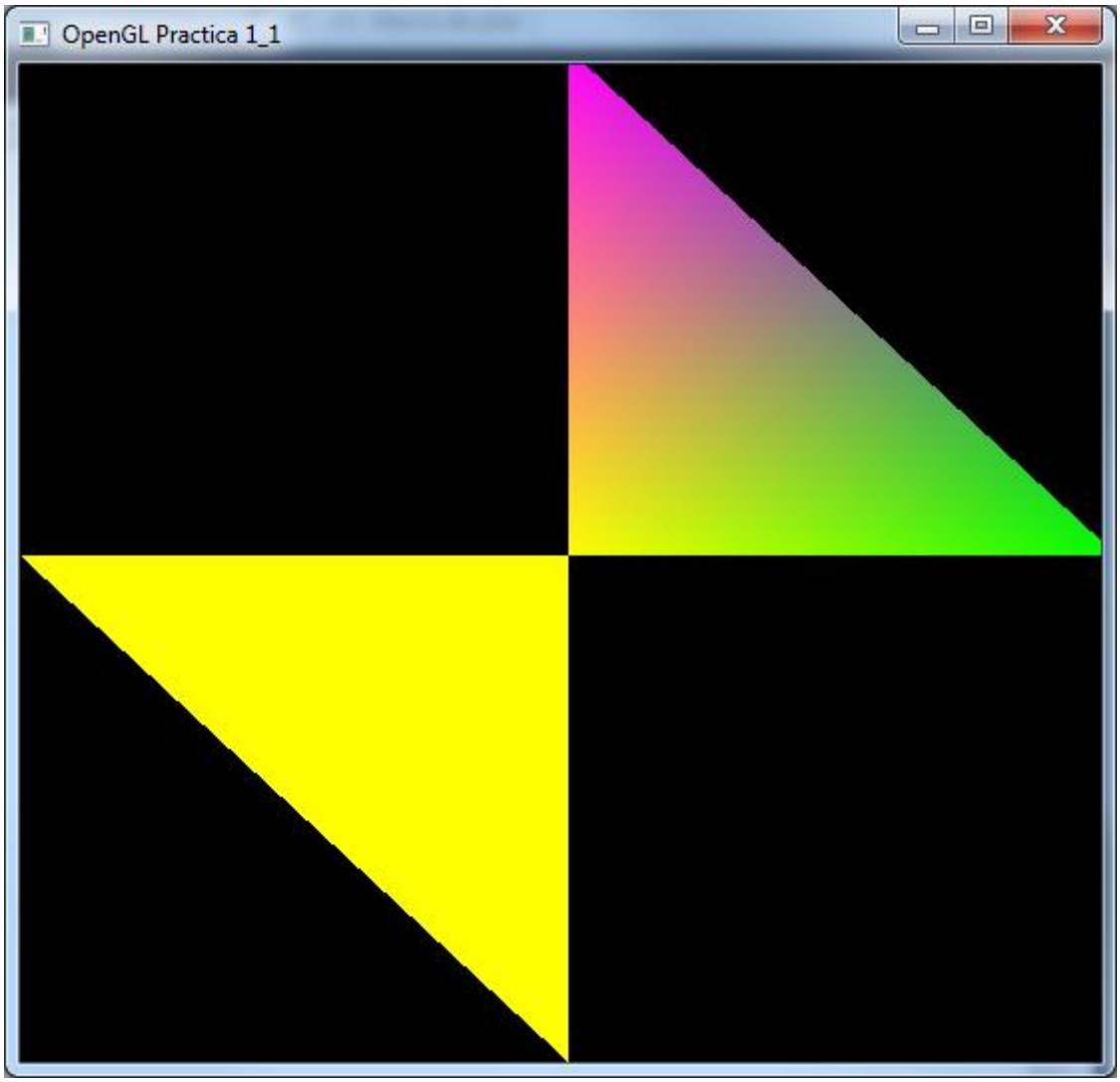

Figura 17: Resultado de la ejecución con los cambios

La última prueba realizada fue cambiar el espacio de trabajo en el *Init()*, cambiando las coordenadas que se le pasan a la función *glOrtho(...)*, la que se encarga de definirlo. El resultado puede verse en *Figura 18*.

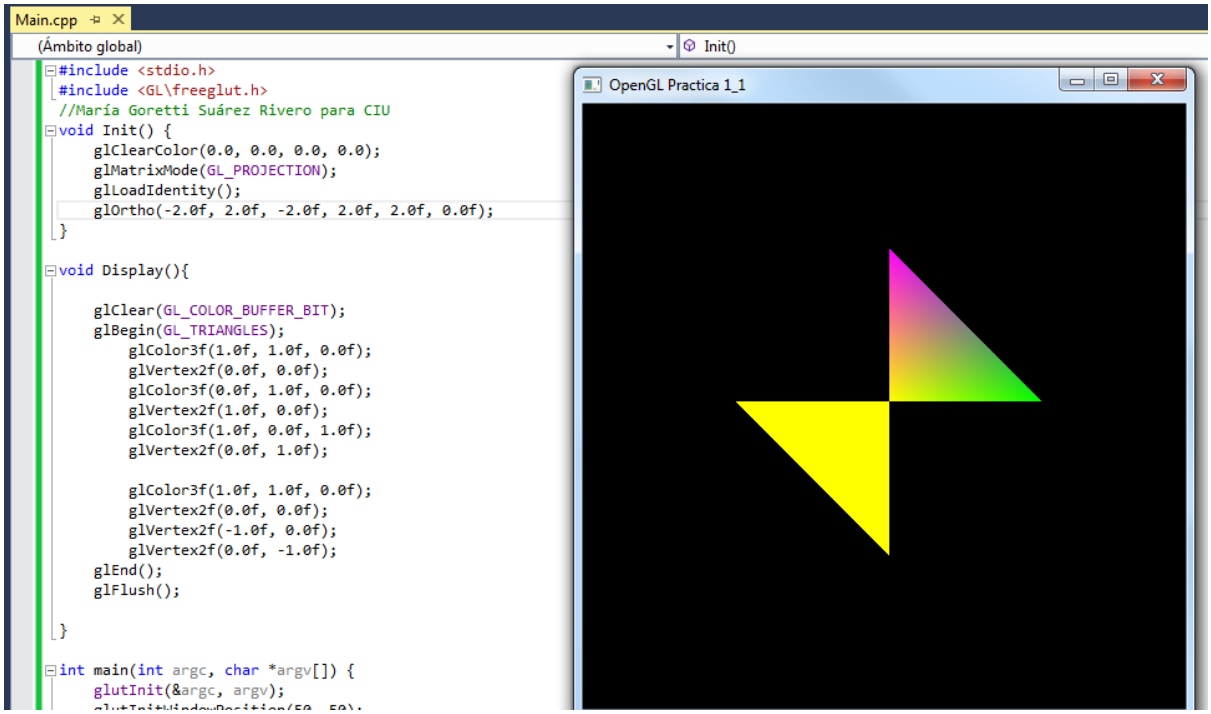

Figura 18: Cambio en las coordenadas del espacio de trabajo

Conclusión

En esta práctica, tal y como se ha podido ver durante el desarrollo de las tres tareas, hemos podido ver como MATLAB usa OpenGL internamente para la representación de modelos 3D y cómo podemos visualizarlos en esta herramienta. Para ello usamos una demo como ejemplo y cargamos ficheros PLY con información sobre modelos. Por último, aprendimos a programar con OpenGL directamente en C, representando dos triángulos y cambiando sus distintas propiedades.

Bibliografía

- Documentación de la práctica (Moodle)