UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Introducción al desarrollo de aplicaciones inmersivas con OpenGL moderno

Apellidos, nombre	**Agustí i Melchor, Manuel** (magusti@disca.upv.es)
Departamento	**Departamento de Informática de Sistemas y Computadores (DISCA)**
Centro	Universitat Politècnica de València

1 Resumen de las ideas clave

En diferentes tipos de aplicaciones se utilizan interfaces gráficas que ofrecen un contenido visual tridimensional para proporcionar al usuario la información con el nivel de atracción y de detalle que se espera le resulte interesante e impresionante. Sucede así desde los desarrollos para **videojuegos** a los recientes usos de tecnologías multimedia inmersivas que han dado pie a aplicaciones de realidad virtual (RV) o aumentada (RA).

En este contexto, el uso de OpenGL como especificación para el uso de gráficos (principalmente en 3D) es un estándar desde finales de 1990, que se ha ido actualizando conforme se sucedían los cambios en el hardware gráfico de los computadores y especialmente con la introducción de las GPUs [1]. Debido a estos avances, hoy es posible disponer de aplicaciones interactivas complejas con gráficos 3D de alta resolución, en tiempo real y a frecuencias superiores a los treinta cuadros por segundo (o *frames per second*, fps), llegando habitualmente a valores superiores a los 60 fps.

Para conseguir estos valores, el modo de proceder de OpenGL ha evolucionado y se puede encontrar (entre la cantidad de ejemplos disponibles en libros y en la red, muchas veces repetidos), la coletilla de "OpenGL clásico" para diferenciar el cambio de proceder que se inicia [2] en la versión 2.0 con la inclusión del soporte de GPUs y el lenguaje GLSL[1] y que termina en la versión OpenGL 3.2 dando lugar al que se denomina "**OpenGL moderno**".

En este artículo se va a presentar un ejemplo básico de aplicación sobre OpenGL en modo moderno que sea posible ejecutar en el computador de escritorio, y que se podría portar a videoconsolas (desde la 3DS a las más actuales Switch, Xbox o PlayStation) que disponen de una **GPU** o muchos otros ejemplos que se pueden encontrar sobre otras plataformas, como el computador de escritorio.

El trabajo mostrado se ha realizado sobre el sistema operativo Linux, pero los resultados son portables a otros sistemas operativos.

2 Objetivos

Una vez que el alumno haya leído el documento y explorado la aplicación de ejemplo que se describe, será capaz de:

- Desplegar la aplicación de ejemplo en su propio computador.
- Describir los elementos básicos de una aplicación basada en OpenGL, funcionando sobre la premisa de ejecutarse con un subsistema de hardware gráfico basado en GPU o uno que lo emule por software.
- Proponer un ejemplo completo de código para poder experimentar con las bases de OpenGL moderno.

[1] Puede leer sobre qué es GLSL en <https://www.khronos.org/opengl/wiki/Core_Language_(GLSL)>.

3 Introducción

OpenGL ha evolucionado, véase Figura 1, desde el modo clásico de OpenGL que utiliza una secuencia fija de operaciones (*fixed pipeline* o tubería fija) para pintar en pantalla, hasta que, en la versión OpenGL 2.0 (en 2003), se inicia un cambio con la introducción de la tubería **programable** (*programmable pipeline*, *[5]*) en la que aparece el concepto de sombreadores (*shaders*) que tienen acceso directo a la GPU. Este modo de desarrollo, de lo que se ha dado en denominar ***OpenGL moderno***, utiliza *shaders* para controlar cómo se pintan los gráficos. Para establecer las dependencias que puede necesitar una aplicación sobre OpenGL, como parte de su inicialización se puede establecer el perfil de la misma: *legacy* para indicar la compatibilidad con el modo clásico y *core* para indicar la ruptura que estable el nuevo modo de operación.

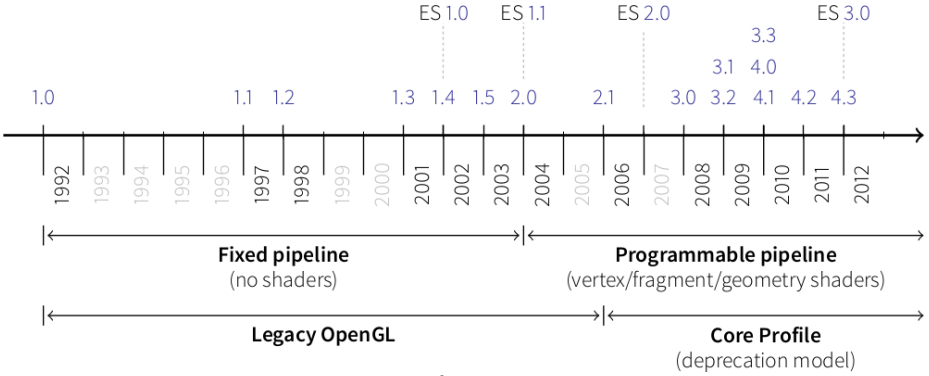
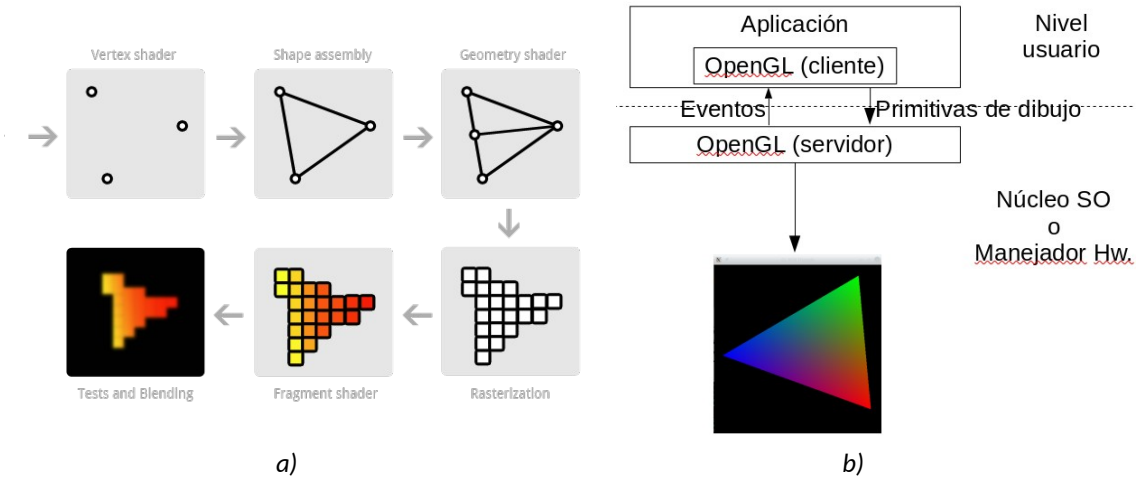


Figura 1: Evolución de OpenGL: asociación entre pipeline y perfiles (imagen de <https://vispy.org/getting_started/modern-gl.html>).

Como muestra la Figura 2a (*[5]*, *[10]*), el uso de OpenGL en una aplicación consiste en invocar las primitivas gráficas que componen su API y que se traducen en un conjunto de vértices (es la parte del programa que se ejecuta en la CPU) que describen una escena 3D; a partir de aquí hay una serie de etapas hasta conseguir el renderizado final en la pantalla que se ejecuta en la tarjeta gráfica. Esta es, casi, un pequeño computador especializado en el procesado de las operaciones relativas a cálculos para generar los gráficos, en el sentido de que tiene un procesador (la GPU) con un juego de instrucciones diferente del procesador central, puede tener su propia memoria y funciona a una velocidad de reloj diferente. Para escribir estos programas se utiliza el lenguaje GLSL (*GL Shader Language*). Los *shaders* o sombreadores, son pequeños programas que se ejecutan en las unidades de procesado paralelo de las GPU y están escritos en el lenguaje GLSL.

Las operaciones de renderizado de OpenGL se reparten así entre dos niveles: cliente y servidor (Figura 2b). El cliente de OpenGL es la parte que se ejecuta en la CPU, se encarga de inicializar un contexto gráfico compatible con OpenGL; está trabajando a nivel de usuario utilizando el API de OpenGL para lanzar las primitivas gráficas que describen los objetos 3D a pintar en la ventana y esperará recibir eventos, si se han registrado o declarado, para poder modificar su flujo de operaciones a raíz de las acciones del usuario. El servidor de OpenGL puede estar a nivel del manejador (o *driver*) de la tarjeta gráfica ofreciendo toda la aceleración que esta ofrece; o a nivel de núcleo del Sistema Operativo (SO), donde proporcionar una implementación totalmente software de OpenGL.



a) *b)*

Figura 2: Flujo de información en OpenGL: (a) La tubería programable (imagen de [10]) y (b) eventos a nivel de la aplicación.

3.1 Instalación básica de OpenGL

Dado que OpenGL solo es una especificación[2], no existe un producto único (una única librería o SDK) que haya que instalar, se puede encontrar ampliamente comentado en el sitio Web de Khronos[3] para las diferentes plataformas:

- En Linux tenemos Mesa, que ofrece tanto implementación software como aceleración gráfica.
- En macOS, Apple incluye su propia implementación de OpenGL y ha acabado desarrollando todo un nivel que se denomina Metal.
- En MS Windows, está basada en el uso de los manejadores para los que los fabricantes de tarjetas gráficas (como NVIDIA o AMD) hacen sus propias implementaciones encaminadas a aprovechar al máximo la funcionalidad de su hardware y también compite con ellos con su propio nivel gráfico a nivel de SO (DirectX).

En cada caso, actualizar el componente que implementa OpenGL es suficiente para el usuario de aplicaciones, pero no para el desarrollador. En nuestro caso, sobre Linux[4], se puede utilizar la orden

```
$ sudo apt-get install build-essential libglu1-mesa-dev freeglut3-dev \
mesa-common-dev
```

[2] Véase en <https://www.khronos.org/opengl/wiki/FAQ#Is_OpenGL_Open_Source?>.

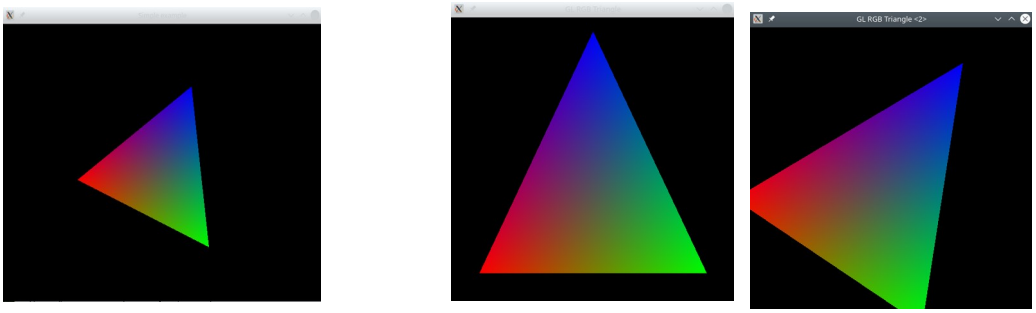
[3] En el apartado "Getting Started" <https://www.khronos.org/opengl/wiki/Getting_Started> y en el de FAQ <https://www.khronos.org/opengl/wiki/FAQ>,

[4] Inspirado en "How to Install OpenGL on Ubuntu Linux"<http://www.codebind.com/linux-tutorials/install-opengl-ubuntu-linux/> y "Cómo instalar Mesa (OpenGL) en Linux Mint" <https://es.wikihow.com/instalar-Mesa-(OpenGL)-en-Linux-Mint>.

4 Desarrollo

Ahora que ya se dispone de una instalación de OpenGL[5], se va a mostrar una aplicación que toma como base, el código de [11][6] (véase Figura 3a) que proporciona una aplicación sencilla en cuanto a la escena: un triángulo con una gradación de color en su interior y que gira en pantalla, realizado con GLFW 3, para la creación del contexto para OpenGL en una ventana y la gestión de los eventos. La parte importante es que el renderizado de la escena se realiza con las técnicas del OpenGL moderno. Se puede compilar con:

```
$ gcc triangle_animat_modern_glfw.c -o triangle_animat_modern_glfw -lglfw -lGL -lm
```



a) *b)*

Figura 3: Capturas de los ejemplos de partida: (a) Ejemplo con OpenGL moderno + GLFW ([11]) y (b) ejemplos con OpenGL clásico + GLUT ([6] y [7]).

Tomando este código de base, se va a reescribir con GLUT, en lugar de con GLFW, para el interfaz. Este cambio a GLUT permitirá comparar con un ejemplo de OpenGL clásico para que el lector pueda contrastar cómo es el mismo ejemplo utilizando las técnicas de OpenGL moderno y clásico. El ejemplo de OpenGL clásico está realizado a partir de una mezcla entre un ejemplo del libro rojo [6] (referencia obligada al hablar de OpenGL), que pinta un triángulo estático en la ventana y otro de [7] (una de las referencias más interesantes que están disponibles en la red), que describe cómo animar un objeto sencillo haciéndolo rotar sobre su centro de gravedad continuamente. Se pueden ver capturas de ambos en la Figura 3b (izquierda y derecha, respectivamente). Como no se puede ver el movimiento en este soporte, la diferencia no es mucha, ¿verdad? ;-). Se pueden compilar con una orden similar a la anterior:

```
$ gcc simple.c -o simple -lglut -lGL -lm
$ gcc triangle_animat_clasic_glut.c -o triangle_animat_clasic_glut -lglut -lGL -lm
```

4.1 Ejemplo de OpenGL moderno con GLUT

El resultado se muestra en la Figura 3b, donde se ve un triángulo cuyo interior muestra la gradación de colores entre los tres primarios situados en sus vértices y cuya posición cambia de forma continua al recibir un clic del ratón en cualquier posición de la ventana.

---

[5] Se puede comprobar su disponibilidad con la orden *glxinfo*.

[6] Ah, se necesita el fichero *linmath.h*, que es un fichero que recoje definiciones de tipos y funciones matemáticas de uso habitual en aplicaciones gráficas, como es el caso de OpenGL.h. Está disponible en <https://github.com/datenwolf/linmath.h>.

El código que lleva a cabo esta aplicación se puede encontrar al completo en Github [8] y del que se muestra en los listados 1, 2 y 3 su contenido principal. En el resto de este punto se va a examinar el contenido de los listados para presentar las instrucciones de OpenGL que aparecen, junto a los comentarios encontrados en los ficheros de código tomados de referencia.

El Listado 1 muestra, entre las líneas 1 a la 16, las cabeceras necesarias para una aplicación OpenGL, junto con la de *glut* que implementará el cliente de OpenGL (a través de *freeglut*, que permitirá recrear el situar el control de eventos en el bucle principal) y la *linmath* (para operaciones algebraicas). Con estas dos últimas se recreará el modo de proceder de la versión que sirve de base a este trabajo ([11]) y, también por ello se ha mantenido la estructura de datos que guarda el conjunto de vértices y colores en la estructura de las líneas 9 a la 16.

Entre las líneas 17 a la 33 se puede ver uno de los pilares de este OpenGL moderno: el uso de *shaders*. Aquí vemos la definición de los dos que se utilizan en este ejemplo. Es una práctica habitual incluirlos como cadenas de caracteres, aunque recuerde que son programas y hay que compilarlos, enlazarlos y ejecutarlos; se verá un poco más tarde. Sobre el código de los mismos hay que hacer notar dos cosas.

- La primera, la versión[7], es la más "básica", ya que la complejidad del ejemplo también lo es.
- El primer *shader* calcula el color para un punto de los objectos en la escena 3D. El segundo calcula el color del polígono que definen los vértices.

Además, entre las líneas 35 y 51, se recogen las funciones para tratar los eventos de teclado (*key_callback*) y la función auxiliar (*spinDisplay*) que gestiona la evolución de la animación de giro del triángulo.

El Listado 2 muestra la mayor parte de la funcionalidad de este ejemplo. Desde la línea 57 a la 63 se crea la ventana de OpenGL con el uso de las funciones de GLUT (contexto de OpenGL, tamaño en píxeles, título de la ventana y asignación de la función que gestiona los eventos de teclado). Se podrían haber registrado otras funciones para la gestión de los eventos[8] de dibujado, redimensionado de la ventana, uso del ratón… Pero se ha preferido no hacerlo para tener una comparativa más directa con el contenido del código de partida [11].

A partir de la línea 65 del Listado 2 se puede ver el código característico de las actividades propias de OpenGL moderno. Siguiendo la secuencia descrita en la Figura 2, primero se cargan los vértices (línea 65 a la 67). Después se cargan los *shaders* (líneas 69 a la 79), la compilación y el enlazado de los cuales podría devolver un error, pero se han omitido estas comprobaciones por brevedad de la exposición. Ahora (líneas 81 a la 83), se puede preguntar por las variables declaradas en estos shaders para poder intercambiar valores entre estas variables (recordemos que son programas que estarán ejecutándose en la GPU) y las del programa principal (que se ejecuta en la CPU). Con esos valores obtenidos se ejecutan otras primitivas de OpenGL entre las líneas 85 a la 90.

---

[7] Se puede ver más sobre las versiones de los shaders y OpenGL en la URL <https://en.wikipedia.org/wiki/OpenGL_Shading_Language>.

[8] Puede ver más sobre las *callback* disponibles en la URL <https://freeglut.sourceforge.net/docs/api.php#WindowCallback>.

```
1.  #include <GL/gl.h>
2.  #include <GL/glut.h>
3.  #include <GL/freeglut_ext.h>   // glutMainLoopEvent
4.  #include <stdlib.h>
5.  #include <stdio.h>
6.
7.  #include "linmath.h"
8.
9.  static const struct {
10.     float x, y;
11.     float r, g, b;
12. } vertices[3] = {
13.     { -0.6f, -0.4f, 1.f, 0.f, 0.f },
14.     {  0.6f, -0.4f, 0.f, 1.f, 0.f },
15.     {   0.f,  0.6f, 0.f, 0.f, 1.f }
16. };
17. static const char* vertex_shader_text =
18. "#version 110\n"
19. "uniform mat4 MVP;\n" //MVP = model * view * projection
    matrices
20. "attribute vec3 vCol;\n"
21. "attribute vec2 vPos;\n"
22. "varying vec3 color;\n"
23. "void main(){\n"
24. "    gl_Position = MVP * vec4(vPos, 0.0, 1.0);\n"
25. "    color = vCol;\n"
26. "}\n";
27.
28. static const char* fragment_shader_text =
29. "#version 110\n"
30. "varying vec3 color;\n"
31. "void main(){\n"
32. "    gl_FragColor = vec4(color, 1.0);\n"
33. "}\n";
34.
35. #define KEY_ESCAPE 27
36.  void key_callback( unsigned char key, int x, int y ) {
37.   switch ( key )  {
38.     case KEY_ESCAPE: // tecla ESC
39.       glutDestroyWindow ( glutGetWindow() );
40.       printf("Acabant per ESC\n");
41.       exit (0);
42.       break;
43.   }
44.   glutPostRedisplay();
45.  }
46.
47. static GLfloat spin = 0.0;
48. void spinDisplay(void) {
49.    spin = spin + 0.0001; // 0.000001;
50.    if (spin > 360.0)   spin = 0.0; //spin - 360.0;
51. }
    ..
```

Listado 1: Ejemplo de aplicación bajo el modelo de OpenGL clásico: first-first-triangle_animat_modern_glut.c.

```
   ...
52. int main( int argc, char** argv ) {
53.     int window;
54.     GLuint vertex_buffer, vertex_shader, fragment_shader, program;
55.     GLint mvp_location, vpos_location, vcol_location;
56.
57.     glutInit( &argc, argv );                          // Initialize GLUT
58.     glutInitDisplayMode( GLUT_DOUBLE  );
59.     glutInitWindowSize( 640,480 );
60.     window = glutCreateWindow(
61.      "GL RGB Triangle + Simple example (OGL modern) + MainLoopEvent");
62.     if ( !window ) exit(EXIT_FAILURE);
63.     glutKeyboardFunc( key_callback );
64.
65.     glGenBuffers(1, &vertex_buffer);
66.     glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
67.     glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
68.                  GL_STATIC_DRAW);
69.     vertex_shader = glCreateShader(GL_VERTEX_SHADER);
70.     glShaderSource(vertex_shader, 1, &vertex_shader_text, NULL);
71.     glCompileShader(vertex_shader);
72.     fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
73.     glShaderSource(fragment_shader, 1, &fragment_shader_text, NULL);
74.     glCompileShader(fragment_shader);
75.
76.     program = glCreateProgram();
77.     glAttachShader(program, vertex_shader);
78.     glAttachShader(program, fragment_shader);
79.     glLinkProgram(program);
80.
81.     mvp_location = glGetUniformLocation(program, "MVP");
82.     vpos_location = glGetAttribLocation(program, "vPos");
83.     vcol_location = glGetAttribLocation(program, "vCol");
84.
85.     glEnableVertexAttribArray(vpos_location);
86.     glVertexAttribPointer(vpos_location, 2, GL_FLOAT, GL_FALSE,
87.                           sizeof(vertices[0]), (void*) 0);
88.     glEnableVertexAttribArray(vcol_location);
89.     glVertexAttribPointer(vcol_location, 3, GL_FLOAT, GL_FALSE,
90.                       sizeof(vertices[0]), (void*) (sizeof(float) * 2));
   ...
```

Listado 2: Ejemplo de aplicación bajo el modelo de OpenGL clásico: first-first-triangle_animat_modern_glut .c (continuación, 2).

El Listado 3 recoge la segunda parte del programa principal, es el encargado de actualizar los valores para que se vea el avance del programa. Es por ello que todas las operaciones están dentro de un bucle sin fin (el *while*) entre las líneas 91 a la 114. Dentro de las cuales podemos distinguir:

- De la 96 a la 99, consulta la geometría de la ventana, para actualizar los valores que se modifican al redimensionar el usuario la ventana. Observe que se realizan siempre, lo que no siempre es necesario, pero es por mantener la secuencia del ejemplo de partida.

- De la 101 a la 105 se utilizan las operaciones algebraicas definidas en *linmath.h* para actualizar los valores que se ejecutarán en los *shaders*. De la línea 107 a la 109, se están ejecutando los *shaders* y actualizando la matriz MVP (compuesta de las tres que definen la escena *Model*, *View*, *Projection*).
- Ahora ya solo falta pedir que se realice el dibujado en pantalla, (línea 111 *glDrawArrays)*, que se intercambien los valores del doble buffer[9] (línea 112, *glutSwapBuffers*) que se utiliza para actualizar los valores de los píxeles (evitando el parpadeo) y que se refresque la cola de eventos (115. con *glutMainLoopEvent*).

```
    ...
91.     while( 1 ) {
92.       float ratio;
93.       int width, height;
94.       mat4x4 m, p, mvp;
95.
96.       width = glutGet( GLUT_WINDOW_WIDTH );
97.       height= glutGet( GLUT_WINDOW_HEIGHT );
98.       ratio = (float)width / (float)height;
99.       glViewport(0, 0, width, height); //--> reshape
100.
101.      mat4x4_identity(m);
102.      spinDisplay( );
103.      mat4x4_rotate_Z(m, m, (float) spin );
104.      mat4x4_ortho(p, -ratio, ratio, -1.f, 1.f, 1.f, -1.f);
105.      mat4x4_mul(mvp, p, m);
106.
107.      glClear(GL_COLOR_BUFFER_BIT);
108.      glUseProgram(program);
109.      glUniformMatrix4fv(mvp_location, 1, GL_FALSE, (const GLfloat*)
    mvp);
110.
111.      glDrawArrays(GL_TRIANGLES, 0, 3);
112.      glutSwapBuffers();
113.      glutMainLoopEvent();
114.     }
115.     glutDestroyWindow( window );
116.     exit(EXIT_SUCCESS);
117.}
```

Listado 3: Ejemplo de aplicación bajo el modelo de OpenGL clásico: first-first-triangle_animat_modern_glut .c (cont. y 3).

Y ya solo falta por saber cómo compilar la aplicación, por ejemplo en Linux, con C/C++ (*gcc* o *g++*), hay que enlazar la aplicación con OpenGL (a través de la implementación de Mesa) y, si se utiliza el API de GLUT para la gestión de eventos, con una orden como:

```
$ gcc ficheroFuente.c -o ficheroEjecutable -lGL -lglut
```

[9] is reversed each frame.

5 Conclusión y cierre

A lo largo de este objeto de aprendizaje se ha explorado la forma de trabajo de aplicaciones bajo OpenGL funcionando en modo moderno. Se ha revisado un ejemplo de código que permite el dibujado y la interacción con el usuario y se ha mostrado cómo implementar la parte del cliente de OpenGL con GLFW y GLUT (véase la Figura 4).

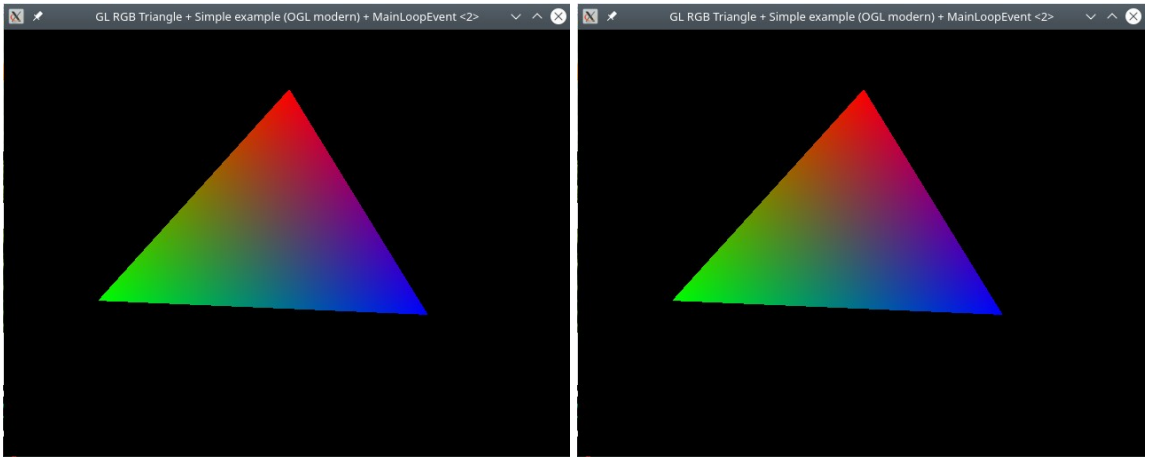


Figura 4: Dos instantes de la ejecución de first-triangle_animat_modern_glut.

Sería interesante ver esta misma realización desde el punto de vista del OpenGL "clásico", para la que se han propuesto las referencias oportunas. Así es posible comparar las dos grandes versiones de OpenGL. Para comprobar qué realmente has aprendido deberías descargar el código y comprobar tú mismo lo que se dice en este artículo y, como no, hacer tus propias pruebas ¡¡ÁNIMO!!

6 Bibliografía

[1] Kirk, D. B., Hwu, W. W. (2013). Chapter 2 - History of GPU Computing, Programming Massively Parallel Processors (Second Edition). Morgan Kaufmann. ISBN 9780124159921, Disponible en la URL <https://doi.org/10.1016/B978-0-12-415992-1.00002-X>.

[2] History of OpenGL. (2022). OpenGL Wiki, Disponible en la URL <http://www.khronos.org/opengl/wiki_opengl/index.php?title=History_of_OpenGL&oldid=14895 >.

[3] Woo, M., Neider, J. y Davis, T. (1997). The OpenGL Programming Guide. Disponible en la URL <http://www.glprogramming.com/red/chapter01.html>.

[4]. Eck, D. J. (2021). Introduction to Computer Graphics. Disponible en la URL < https://math.hws.edu/graphicsbook/source/glut/>.

[5] Chris X Edwards. Fixed vs. Programmable Pipeline or Older vs. Newer. Programming OpenGL On Linux <https://xed.ch/help/opengl.html#py>

[6] OpenGL Programming Guide. The Official Guide to Learning OpenGL, Version 1.1. Sitio web del libro de Woo, M., Neider, J., Davis T. y el OpenGL Architecture Review Board. (1997). Disponible en < http://www.glprogramming.com/red/chapter01.html>.

[7] Eck, D. J. (2021). Introduction to Computer Graphics. Version 1.3, Disponible en la URL <https://math.hws.edu/graphicsbook/>.

[8] Agustí-Melchor, M. (2023). OpenGL modern. Repositorio en Github del código de ejemplo. Disponible en la URL <https://github.com/magusti/OpenGL_examples/OpenGL_modern/>.

[9] Segal, M., Akeley, K. y Frazier, C.. (1994). The OpenGL Graphics System: A Specification (Version 1.0). Disponible en <https://khronos.org/registry/OpenGL/specs/gl/glspec10.pdf>.

[10] Overvoorde, A. (2019). Modern OpenGL Guide test. Disponible en la URL <https://open.gl/>.

[11] GLFW. Getting started. Disponible en la URL <https://www.glfw.org/docs/3.3/quick_guide.html>.