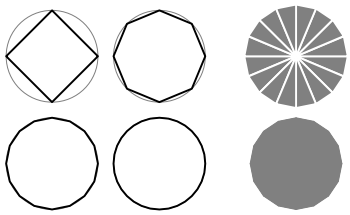
Introducción a OpenGL Moderno

En este tutorial vamos a aprender lo mínimo que se necesita para poner en marcha un programa 2D o 3D que ”dibuje” con OpenGL moderno. Se van a repasar algunos conceptos teóricos generales de cómo funcionan este tipo de programas, y luego se van a ir introduciendo de a poco todos los elementos necesarios para lograr una especie de ”Hola Mundo” gráfico, explicando el por qué de cada paso. Se parte de un programa completamente en blanco, y a lo largo del texto se irá presentando todo el código necesario. El programa final hará uso de las bibliotecas OpenGL, GLEW, GLM, y GLFW3; más adelante se describe el por qué de cada una.

1. Pipeline de una aplicación gráfica

En todo programa gráfico, parte del trabajo se lleva a cabo en la CPU, y parte en la GPU. La **GPU** es un componente de hardware especialmente diseñado para ejecutar con mucha velocidad y eficiencia las operaciones necesarias para generar un imagen a partir de ciertos datos y mostrarla en pantalla. Esos ”ciertos datos” son un descripción de lo que vamos a ver en la imagen. Obviamente la GPU no admite cualquier tipo de descripción, sino solo una muy particular; y entonces es tarea de la parte que corre en CPU decidir qué se debe mostrar y ”describirlo adecuadamente” para que la GPU lo pueda procesar luego. Al proceso que realiza la GPU, convertir esa descripción en una imagen, se lo denomina **rendering** o **renderizado**.

En este contexto, denominamos **primitivas** a las cosas que le podemos pedir a la GPU que ”dibuje”. Las primitivas posibles son muy pocas: puntos, segmentos de recta, y triángulos. Cualquier otra geometría más compleja, debe descomponerse en primitivas. Por ej, para dibujar una circunferencia, dado que no es una primitiva, debemos aproximarla con muchos segmentos, y enviar a la GPU la lista de segmentos. Cuantos más segmentos usemos, mejor será la aproximación, pero también más trabajo tendrán CPU y GPU. Entonces, siempre habrá que hacer compromisos y bucar un equilibrio entre cuan fielmente se aproximan los objetos y cuanto poder de cómputo podemos utilizar.

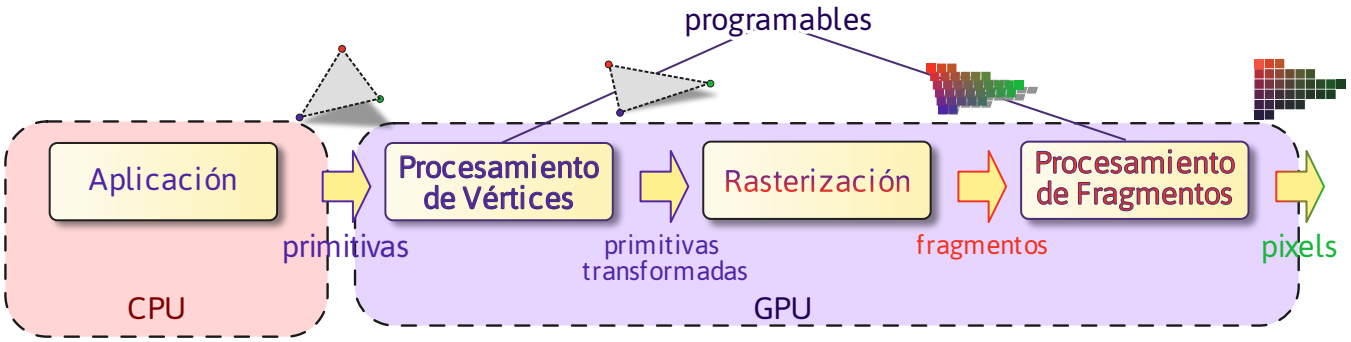


- *Derecha: distintas aproximaciones de una circunferencia (el contorno) utilizando segmentos. Izquierda: posible aproximación de un círculo (interior) utilizando triángulos.*

El procesamiento que realiza la GPU se divide en varias etapas que se ejecutan secuencialmente (una tras otra) para cada primitiva. Si bien, para una primitiva, el procesamiento es secuencial (no se puede pasar a la 2da etapa sin tener los resultados de la primera), la GPU logra un alto grado de paralelismo

dado que usualmente se le pide procesar un gran número de primitivas. Entonces, puede ejecutar muchas primeras etapas en paralelo, o ejecutar la primera etapa de un grupo a la par de la 2da etapa de otro, etc. Nos referimos a este modelo de procesamiento como **pipeline** (como en una linea de montaje fordiana) de renderizado.

Para modelar un pipeline básico, necesitamos describir al menos 3 etapas del pipeline en la GPU: procesamiento de vértices, rasterización, y procesamiento de fragmentos. Se pueden hacer descripciones más detalladas, o agregar más etapas, pero para un primer acercamiento a OpenGL, vamos a pensar solo en estas 3. De esas 3 etapas, la primera y la última son programables (y deben ser programadas, ya no es opcional hacerlo); la 2da etapa en cambio, es fija, siempre se realiza de la misma manera, y no podemos reprogramar ese proceso. Considerando todo esto, podemos modelar una aplicación gráfica de la siguiente manera:



Se desprende de aquí que para generar un programa gráfico tendremos que escribir al 3 programas, uno para la CPU (aplicación) y otros 2 para la GPU (procesamiento de vértices y de fragmentos). Por razones históricas, a los programas que corren en la GPU se los denomina **shaders**. Los shaders se escriben usualmente en un lenguaje propio, denominado GLSL, que no es C ni C++, pero que tiene una gran similitud en muchos aspectos.

La aplicación es entonces la que se encarga de decidir qué objetos hay que mostrar, y describir esos objetos mediante primitivas. Las primitivas se definen mediante uno, dos o tres vértices (puntos, segmentos o triángulos). Pero los vértices no son solo una posición, sino que pueden llevar mucha más información asociada; información que servirá, por ejemplo, para determinar cómo se debe pintar la primitiva. Entonces, lo que la aplicación le da a la GPU es una listas de primitivas formadas por vértices, donde cada **vértice** representa una posición en el espacio más otras propiedades (como el color) asociadas a esa posición.

En la primera etapa programable de la GPU, el procesamiento de vértices, se reciben esos vértices que envió la CPU, y se los transforma en otros. Esto es porque lo vértices estarán expresados en un sistema de coordenadas arbitrario (a gusto y conveniencia de la aplicación), pero para el renderizado se debe determinar en qué posición de la pantalla (0 de la imagen) caen. Por esto se transforman las cordenadas de ese sistema arbitrario al de la imagen. Esta transformación suele descomponerse en 2 o 3 pasos, porque depende de dónde se ubica el objeto que estamos renderizando en nuestro mundo virtual, de dónde se encuentre la cámara y cómo se oriente, y de otras propiedades de esa cámara que definen cómo ese espacio 3D que ve la cámara se ”aplasta” en una imagen 2D. [1]

[1] En la teoría de transformaciones se describe con mucho más detalle y formalismo este proceso; por el momento no es necesario

Una vez que los vértices están ubicados en el plano 2D de la imagen, la etapa de rasterización es la que determina para cada primitiva, qué ”pixeles” de la imagen cubre (cuáles habría que habría que pintar). Por ej, en un triángulo, tendremos los 3 vértices ubicados sobre la imagen, y al conectarlos el triágulo cubrirá un cierto conjunto de pixeles. Sin embargo, al resultado de esta etapa lo llamaremos **fragmento** en lugar de pixel. El fragmento es por un lado un ”candidato a pixel”, pero todavía no está decidido si efectivamente llegará a la imagen final (falta la etapa de procesamiento de fragmentos). Por ejemplo, podría ser que parte del triángulo, desde el punto de vista de la camara, se encuentre tapado por (detrás de) otro objeto, y entonces esos fragmentos no deberían pintarse en la imagen final. Pero además, al igual que los vértices, el fragmento lleva información adicional (no es solo la posición de un pixel) que podrá ser usada en la siguiente etapa. Para cada fragmento que genera un triángulo, se asignan sus propiedades combinando las que tenían los tres vértices de ese triángulo.

La última etapa en nuestro pipeline simplificado, el procesamiento de fragmentos, es la que determina cuales fragmentos finalmente se convertirán en píxeles (y cuales serán descartados) y de qué color será ese píxel (todo esto calculado de acuerdo a esas propiedades adicionales que lleva el fragmento).

El pixel finalmente podrá pintarse en la imagen reemplazando el contenido previo, o mezclarse con el contenido previo. En el 2do caso, habría una última etapa denominada ”blending”, pero por el momento para simplificar la estamos omitiendo.

Resumiendo, tendremos que escribir 3 programas:

1. La aplicación propiamente dicha, que corre en CPU y es la que genera las primitivas y sus propiedades para enviar a la GPU.
2. El **vertex-shader**, un programa que corren en la GPU, por cada vértice que envió la CPU, y puede modificarlo para luego pasárselo al rasterizador.
3. El **fragment-shader**, otro programa que también corre en la GPU, por cada fragmento generado por el rasterizador, y determina si debe descartarse o no, y en caso de que no, su color final.

2. Bibliotecas e implementación

OpenGL provee funciones para enviar datos, configuraciones y shaders a la GPU; pero nada más que eso. OpenGL por ejemplo no provee funciones para crear un ventana en la cual dibujar; sino que solo pude dibujar sobre una ventana ya creada por otra biblioteca. Es por esto que toda aplicación OpenGL hará uso de una o más bibliotecas adicionales para gestionar la creación de la ventana y la detección de eventos (teclas, clicks, redimensión, etc) sobre la misma. Todas estas operaciones varían de un sistema operativo a otro, por lo cual no es conveniente utilizar las bibliotecas propias de un sistema operativo, sino buscar algún wrapper multiplataforma que nos independice del mismo. Para aplicaciones sencillas con OpenGL, las más utilizadas son FreeGLUT y **GLFW3**. Estas proveen una funcionalidad mínima, muy limitada, a cambio de una gran simpleza de uso. Para aplicaciones más complejas, se podrían usar bibliotecas más completas (y complejas) como QT o wxWidgets. Para este tutorial, seleccionamos GLFW3.

profundizar para avanzar con el ejemplo.

La biblioteca que incluiremos para tener acceso a las funciones de OpenGL moderno es GLEW. GLEW nace originalmente para dar acceso a las extensiones de OpenGL. Pero ocurre habitualmente que las funciones que se van agregando a la API en cada versión, se presentan primero como extensiones en alguna versión previa.

Otra biblioteca que se suele utilizar con mucha frecuencia para estas aplicaciones es **GLM**. Esta biblioteca provee tipos de datos que son equivalentes a los tipos de datos que ofrece GLSL (el lenguaje de los shaders), y funciones y sobrecargas para realizar una gran cantidad de operaciones sobre los mismos. La mayoría de estas operaciones se corresponden con operaciones algebraicas sobre puntos, vectores y matrices; operaciones muy útiles y frecuentes en cualquier programa gráfico, especialmente si se trabaja en 3D.

TODO: *poner algo sobre cómo instalar y configurar estas bibliotecas no std en windows y linux*

Finalmente, es importante destacar por qué el título de este texto aclara que nos referimos a OpenGL *moderno*. Las primeras versiones de OpenGL difieren en muchos aspectos importantes de las actuales. OpenGL nace en los 80s y se formaliza a principios de los 90s. Por esto, las primeras versiones fueron pensadas para el hardware y las necesidades de ese momento (todavía ni existía el concepto de GPU!), muy diferentes a las actuales. Desde la versión 3.0 de OpenGL se empezaron a producir grandes cambios en la API para adaptarlo al hardware moderno que implicaron la ruptura de la compatibilidad hacia atrás. Hoy en día, al momento de inicializar OpenGL en una aplicación, debemos elegir entre el modo de trabajo viejo (denominado *compatibility*) y el nuevo (denominado *core*).

3. Concepto de máquina de estados

La API de OpenGL está diseñada alrededor del concepto de máquina de estados. Esto implica que la maquinaria de OpenGL en todo momento se encuentra en cierto estado, y ese estado se va a mantener hasta que alguna llamada a alguna función pida explícitamente modificarlo. Esto hace que muchas funciones usen como entrada, además de los argumentos de la misma, variables de ese estado. Esto es una entrada de alguna manera oculta o implícita para la función. Conceptualmente ya no es una función pura; pues puede suceder que dos invocaciones a la misma función con los mismos argumentos resulten de diferentes salidas, debido a que se ejecutan en momentos en que OpenGL está en diferentes estados.

Veamos un ejemplo: la función `glClear` sirve para *limpiar* un buffer. Por ejemplo, para *limpiar* la memoria de una imagen (buffer de color) antes de comenzar a dibujar una nueva. La función recibe un enum que le indica cual buffer se quiere limpiar. Pero, ¿qué significa *limpiar* un buffer? Significa poner todas sus posiciones en un mismo valor inicial. Por ejemplo, si se trata del buffer de color, podría ser poner todo en negro, o todo en blanco, o todo en verde, o cualquier color que el usuario quiera. Pero si la función solo recibe cual buffer, ¿cómo sabe qué color? Lo toma del estado, y hay una función `glClearColor` que sirve para modificar ese estado y decir qué color hay que usar cuando se limpia el buffer de color. Entonces, si un programa define un color con `glClearColor`, será ese el que se use para todos los `glClear(GL_COLOR_BUFFER_BIT)` que se ejecuten de allí en adelante, hasta finalizar el programa, o hasta que otra llamada a `glClearColor` vuelva a modificar ese estado. Esto evita que las funciones deban recibir tantos argumentos; pero a veces

complica la legibilidad del código porque no está explícito qué parte del estado es importante para cada función.

Otra forma de verlo, es pensar que la GPU es una maquina muy muy compleja, con muchas perillas, botones, ranuras, etc donde hay que definir configuraciones y cargar entradas *antes de que empiece a trabajar*. Ese proceso es complejo y tedioso, pero una vez finalizado, uno simplemente enciende la máquina y esta comienza a trabajar inmediatamente, ya que tiene todo lo que necesita predefinido en su estado. Llevado a, por ejemplo, un juego; la idea es que se cargue en GPU una sola vez, antes de empezar a jugar, todo lo necesario para renderizar, y que en una vez comenzada la partida, para renderizar cada frame simplemente tengamos que *encender* la maquinaria y nada más, que ya tenga todo lo que necesita listo para no perder tiempo.

Muchas de las bibliotecas auxiliares que se utililzan en combinación con OpenGL replican la misma filosofía.

4. Inicialización de una ventana para OpenGL

Ya mencionamos que GLFW3 se iba a encargar de crear la ventana y gestionar los eventos. Para ello debemos incializar glfw (con la función `glfw_init`), indicar la versión mínima de OpenGL que vamos a necesitar (con la función `glfwWindowHint`), y finalmente pedirle que cree una ventana (en este ejemplo de 800x600px) y active su contexto OpenGL (esto es algo así como seleccionar esa ventana, para que todas las próximas llamadas a OpenGL trabajen sobre esa ventana, ya que podría haber otras).

```
#include <GLFW/glfw3.h>

int main() {
    // inicializar glfw
    glfwInit();

    // definir el tipo de contexto opengl que vamos a necesitar
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR,3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR,2);
    glfwWindowHint(GLFW_OPENGL_PROFILE,GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT,GL_TRUE);

    // crear y activar la ventana
    GLFWwindow *window = glfwCreateWindow(800,600,"Ejemplo CG",nullptr,nullptr);
    glfwMakeContextCurrent(window);

    ...
}
```

Una vez inicializada la ventana, hay que definir los **callbacks** para los eventos e implementar el bucle de eventos principal. Un callback es una función a la que hay que invocar cuando sucede algo en particular. Crearemos una para que glfw invoque cuando alguien presione una tecla, y la usaremos para cerrar la ventana al presionar Escape:

```
void keyboard_cb(GLFWwindow* glfw_win, int key, int scancode, int action, int mods) {
    if (key == GLFW_KEY_ESCAPE)
        glfwSetWindowShouldClose(glfw_win,GL_TRUE);
}
```

Dado que esta función será invocada por glfw, es la biblioteca glfw la que impone el prototipo que debe cumplir la misma. Ahora, haremos la asociación entre el evento y la función; y luego el bucle principal:

```
#include <GLFW/glfw3.h>

int main() {
    /* ...inicializaciones varias... */

    // registrar callbacks
    glfwSetKeyCallback(window,keyboard_cb);

    // bucle de eventos
    while (not glfwWindowShouldClose(window)) {
        // to-do:renderizar
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glfwTerminate();
}
```

En lugar del comentario `to-do:renderizar` deberemos colocar luego el código que efectivamente renderize algo. Lo habitual es que durante el renderizado OpenGL no escriba los píxeles directamente en la pantalla (porque si el renderizado tarda se podría llegar a ver la imagen a mitad del proceso, sin terminar), sino que lo haga en otro buffer de memoria, y una vez que esté finalizada, le indique a la placa de video que ahora debe utilizar ese nuevo buffer para enviar al monitor. A esta técnica se la conoce como **double-buffering**, un buffer es el que se está enviando al monitor, otro el que se está usando para preparar la próxima imagen. Cuando está lista, la función `glfwSwapBuffers` es la que los ”intercambia” [2]

El bucle de eventos se repetirá constantemente. En cada iteración, si se genera un evento con un callback asociado, la llamada a `glfwPollEvents` lo detectará y ejecutará el callback. Este bucle se rompe cuando se pide cerrar la ventana, y en ese caso la aplicación finaliza.

En este punto, ya podemos ejecutar el program y ver simplemente una ventana en blanco, que se cierra al presionar la tecla Escape.

[2] no es literalmente un intercambio, ya que si bien el buffer en el que estabamos renderizando se va a mostrar en pantalla, no implica necesariamente que el buffer que estaba en pantalla sea el que usemos (sin cambios) para renderizar la siguiente imagen.

5. Envío de datos a la GPU

En las primeras versiones de OpenGL, los vértices (y sus propiedades) se enviaban uno por uno a la GPU en cada frame. En el hardware actual, capaz de procesar millones de vértices y primitivas por segundo, esto sería extremadamente ineficiente. Lo que se hace actualmente es juntar en un vector muchos vértices (o las propiedades de muchos vértices, o muchas primitivas, etc) y enviar toda esa información junta a la GPU en un solo paso, y una sola vez. Esto le permite al sistema optimizar la comunicación con la GPU, porque la GPU necesita tener en su propia memoria ram (usualmente más pequeña pero más rápida que la principal) la información necesaria para el procesamiento, para poder realizarlo eficientemente y en paralelo. Entonces, la aplicación arma el vector, y luego lo envía a la GPU. Al vector en la GPU se lo denomina **Vertex Buffer Object** (o simplemente **VBO**). Una aplicación puede generar varios VBOs (por ej, porque hay varios objetos complejos para renderizar), y luego en cada parte del renderizado decidir cuales usar y para qué; pero lo importante es, si la información no cambia, tratar de enviarla una sola vez a la GPU, y no por cada cuadro que se renderiza.

Los VBOs se agrupan de **Vertex Array Objects** (o **VAO**). Esto permite enviar a la GPU diferentes VBOs, agrupados en diferentes VAOs, y en cada etapa del rendering elegir cuales activar. Tener varios VAOs nos permite pasar de usar un cierto conjunto de VBOs, a usar inmediatemente otro conjunto de VBOs, con una sola llamada a la API. Por ejemplo, si en una escena hay dos objetos complejos, y cada uno requiere varios VBOs (por ej para las posiciones de los vértices, los colores, las normales, coordenadas de texturas, etc), podemos tener todo en memoria asociando los VBOs de un objeto a un VAO, y los del otro a otro, y alternar entre ambos conjuntos de VBOs simplemente cambiando el VAO activo. Sin embargo, para la mayoría de las aplicaciones de este curso, dado que serán aplicaciones simples, utilizaremos un único VAO.

Entonces, lo que haremos será crear un arreglo con 3 vértices para definir un triángulo. Luego, crearemos un VAO; dentro del VAO crearemos un VBO, y en el VBO pondremos la información de esos vértices:

```cpp
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <glm/glm.hpp>
#include <vector>

int main() {

    /* ...inicializar glfw y crear ventana... */

    glewInit();

    // datos de los vertices
    std::vector<glm::vec4> vPos = {
        { -0.5, -0.5, 0.0, 1.0 },
        { +0.5, -0.5, 0.0, 1.0 },
        {  0.0, +0.5, 0.0, 1.0 } };

    // crear el vao
    GLuint vao;
    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);

    // crear un vbo
    GLuint vbo_pos;
    glGenBuffers(1,&vbo_pos);
    // llenar el vbo con los datos de vPos
    glBindBuffer(GL_ARRAY_BUFFER,vbo_pos);
    glBufferData(GL_ARRAY_BUFFER, v.size()*sizeof(glm::vec4), v.data(), GL_STATIC_DRAW);

    /* ...definir eventos y bucle principal... */
}
```

Definimos los vértices en 4D mediante el tipo de dato `glm::vec4`, por lo que hay que incluir la biblioteca. La GPU internamente opera en 4 dimensiones, por eso suele ser conveniente definirlo así aunque solo utilicemos las dos primeras. El significado y la utilidad de la 4ta coordenada, denominada *w*, se estudiará en detalle más adelante en la teoría de espacios y transformaciones. Por el momento alcanza con decir que debe ser 1 para puntos, y 0 para vectores.

Las funciones para gestionar los VAOs y VBOs están en la biblioteca GLEW, por lo que habrá que incluirla e inicializarla. Es importante para GLEW el orden: debe incluirse antes que cualquier otra biblioteca relacionada a OpenGL, y debe inicializarce luego de que la ventana esté creada y su contexto activo.

Las funciones `glGenVertexArrays` y `glGenBuffers` crean VAOs y VBOs respectivamente. Se les puede pedir crear más de un buffer, por ello el primer argumento es la cantidad, y el segundo un puntero a uno o a un arreglo de enteros (de tipo GLuint) en los que colocar los ids de los buffers creados. Luego de la creación, nos referimos a los buffers a través de sus ids. La funciones `glBind...` definen algo como activo/actual (recordar el concepto de máquina de estados presentado antes). Por ejemplo, `glBindVertexArray` define cual es el vertex-array actual; y este será entonces el que se utilice de ahora en más para cualquier operación de OpenGL, hasta que decidamos cambiar a otro (nuevamente con `glBindVertexArray`).

Todo este código se debe ejecutar una vez, antes del bucle principal. Luego, el contenido del buffer se

mantendrá en la memoria de la GPU para utilizarlo todas las veces que sea necesario (para renderizarlo una y otra vez). Luego del bucle, al finalizar la aplicación, podemos liberar esos recursos:

```
int main() {

    /* ...incializiaciones... */

    /* ...bucle principal... */

    glDeleteBuffers(1,&vbo_pos);
    glDeleteVertexArrays(1,&vao);
    glfwTerminate();
}
```

6. Renderizado

En este punto, la GPU tiene la información de los vértices, pero aún no definimos qué debe hacer con ella. Habrá que indicarle que queremos que la use para dibujar un triángulo con `glDrawArrays`.

La arquitectura de las GPUs actuales está pensada para que en cada frame se reconstruya toda la imagen completa, en lugar de intentar actualizar solo lo que cambia desde el frame anterior. Por esto, casi siempre el primer paso del renderizado es *limpiar* todo.

```
#include <GLFW/glfw3.h>

int main() {

    /* ...inicializaciones varias... */
    glClearColor(1,1,1,1);

    // bucle de eventos
    while (not glfwWindowShouldClose(window)) {
        glClear(GL_COLOR_BUFFER_BIT);
        glDrawArrays(GL_TRIANGLES, 0, 3);
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    /* ...liberar recursos... */
}
```

El bucle comienza limpiando el buffer sobre el que vamos a dibujar (el buffer de color, más adelante veremos que hay otros buffers auxiliares para usar durante el renderizado), con `glClear`.

En una aplicación compleja, en lugar de una simple llamada a `glDrawArrays`, tendremos probablemente varias llamadas, cada una precedidad de comandos para seleccionar distintos buffers y distintos shaders (`glBind...`). En esta aplicación hemos definido un solo VAO y un solo VBO, y han quedado activos, por lo que no es necesario volver a seleccionarlos.

Las piezas claves que faltan para ver el triángulo en pantalla son los shaders. La llamada a `glDrawArrays` con 3 vértices hará que esos vértices pasen por el vertex-shader, luego se rasterice el triángulo que forman (se determine qué pixeles cubriría), y finalmente los fragmentos generados pasen por el fragment-shader. Pero todavía no hemos cargado ningún shader, por lo que aún no veremos nada en la ventana si ejecutamos el programa.

7. Compilación de Shaders

Cuando se programa en C o en C++, el programa final suele ser compilado por su desarrollador una vez por cada plataforma, y luego distribuye ese ejecutable generado a sus clientes. Un ejecutable compilado para Windows, normalmente correrá en cualquier sistema Windows; entonces el usuario solo necesita ese ejecutable y no el código fuente.

En el caso de los shaders, el modelo de trabajo no suele ser así. Cada combinación de sistema+GPU+driver puede requerir de un *ejecutable* diferente para un shader. Esto es así porque las arquitecturas de las GPUs varían mucho entre sí, no todas garantizan las mismas capacidades, y además la optimización para cada una de ellas puede ser muy diferente. La solución para esto es hacer que el driver de video sea el que compile el shader. Entonces, cada programa que distribuimos lleva el código fuente de sus shaders; y es el driver de video el que, al momento de ejecutar el programa, en cada sistema realiza la compilación para su GPU en particular.

El código fuente del shader puede estar en el programa como una cadena de texto constante (`const char fuente[]="..."`), puede cargarse desde un archivo de texto (por comodida, así lo haremos aquí), o hasta en casos complejos donde se requieren múltiples variantes puede ser generado automáticamente con operaciones de cadenas combinando fragmentos de código predefinidos. Cualquiera sea el origen, lo que espera la api de GLEW es una cadena de caracteres por cada shader.

Mínimamente se deben definir un vertex-shader y un fragment-shader (existen otros shaders para generar pipelines más complejos, pero son opcionales y no se requieren para este ejemplo).

El código para cargar y compilar un shader podría ser:

```
std::string read_file(const std::string &path) {
    std::string content;
    std::ifstream file(path,std::ios::binary|std::ios::ate);
    if (not file.is_open()) {
        std::cerr << "No se pudo leer el archivo " << path << std::endl;
        exit(1);
    }
    content.resize(file.tellg());
    file.seekg(0);
    file.read(&(content[0]),content.size());
    return content;
}

GLuint compile_shader(const std::string &path, GLenum type)  {
    // crear un shader en blanco
    GLuint handler = glCreateShader(type);
    // cargar y asignar el código fuente
    std::string src = read_file(path);
    const char *src_ptr = src.c_str();
    int src_len = src.size();
    glShaderSource(handler, 1, &src_ptr, &src_len);
    // compilar y verificar si hay errores
    glCompileShader(handler);
    GLint status;
    glGetShaderiv(handler, GL_COMPILE_STATUS, &status);
    if (status!=GL_TRUE) { // si no compilo, mostrar errores
        char buffer[4096];
        glGetShaderInfoLog(handler, 4096, NULL, buffer);
        std::cerr << "Errores compilando " << path << std::endl;
        std::cerr << buffer << std::endl;
        exit(1);
    }
    return handler;

}
```

La primera función simplemente carga todo el contenido del archivo en un string. La segunda crea el shader con `glCreateShader` (donde `type` indicará si es vertex o fragment), asigna el código fuente cargado (con `glShaderSource`, que recibe un arreglo de cstrings y arreglo de sus longitudes), lo compila (con `glCompileShader`), y si la compilación no es exitosa le pide los errores (con `glGetShaderInfoLog`) y los muestra en consola. En todo momento, para hacer referencia al shader creado se usa el id que retornó `glCreateShader`.

Con estas funciones podemos cargar y compilar tanto el vertex-shader como el fragment-shader. Pero para la GPU, estos dos shaders deben enlazarse y enviarse a la misma como un solo y único programa (cada programa se compone por un conjunto de shaders que definen un pipeline completo). La siguiente función crea el programa combinando un fragment-shader y un vertex-shader.

```
GLuint load_shaders(const std::string &vertex_path, const std::string &fragment_path) {
    // crear programa
    GLuint program_handler = glCreateProgram();
    // cargar y compilar cada shader
    GLuint vertex_handler = compile_shader(vertex_path,GL_VERTEX_SHADER);
    GLuint fragment_handler = compile_shader(fragment_path,GL_FRAGMENT_SHADER);
    // enlazar el programa final
    glAttachShader(program_handler,vertex_handler);
    glAttachShader(program_handler,fragment_handler);
    glLinkProgram(program_handler);
    return program_handler;
}
```

8. Programación de shaders

Ya tenemos entonces las funciones necesarias para que, dados los códigos fuentes de los shaders, se carguen, compilen, enlacen y estén listos para ejecutarse en la GPU. Falta especificar entonces sus códigos fuentes.

El vertex-shader inicial será el siguiente:

```
#version 150 core

in vec4 vertPosition;

void main() {
    gl_Position = vPosition;
}
```

La primer linea indica qué versión mínima del lenguaje GLSL se requiere para compilarlo (a medida que OpenGL evoluciona y se publican nuevas versiones, también lo hace a la par GLSL). Luego, se declaran las entradas (`uniform` e `in`) y salidas (`out`) del shader, de forma similar a variables globales. En este caso la única entrada es la posición del vértice, a la que llamaremos `vertPosition`. La salida es `gl_Position`, variable que sí o sí hay que asignar en el main. Esta salida es la única obligatoria y por ello no se declara, porque está siempre. Si quisiéramos agregar más propiedades a los vértices para disponer de ellas en el fragment-shader podríamos declarar más salidas (`out`). En este código la entrada se declaró con `in` y eso indica que es un dato que varía con cada vértice. Una entrada declarada con `uniform` tendrá el mismo valor para todos los vértices. Habrá un ejemplo con `uniform` y `out` más adelante.

El fragment-shader incial será el siguiente:

```
#version 150 core

out vec4 outColor;
```

```
void main() {
    outColor = vec4(1.0,0.0,0.0,1.0);
}
```

Aquí el valor de salida es el color del fragmento, al que llamamos `outColor`. En este ejemplo no se calcula nada a partir de las entradas; pero hay algunas implícitamente definidas (`gl_FragCoord` y `gl_FragDepth`) que refieren a la posición del fragmento en la imagen (y que por el momento no usaremos).

Notar que al asignar el color se usan `1.0` y `0.0` en lugar de `1` y `0`. En C o C++, serían indistinto, ya que el valor entero se promovería automáticamente a flotante. En GLSL no hay conversiones implícitas, por lo que el tipo debe coincidir exactamente, o convertirse explícitamente.

El color de salida aquí es rojo, los cuatro valores corresponden a RGBA: cuanto de rojo, cuanto de verde, cuanto de azul, cuando de alpha (alpha se usa habitualmente como opacidad).

9. Uso de shaders

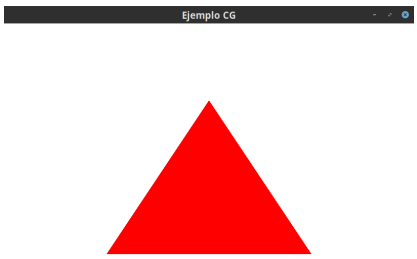
Resta entonces activar los shaders y conectar los datos que ya enviamos a la GPU con las entradas del vertex-shader. Suponiendo que guardamos los fuentes en los archivos ”shader.vert” y ”shader.frag” (*.vert* y *.frag* son extensiones habituales para vertex-shaders y fragment-shaders respectivamente), entonces podemos usarlos en el main de la siguiente manera:

```
int main() {
    /* ...inicializaciones varias... */

    // cargar y configurar los shaders
    GLuint shader_program = load_shaders("shader.vert","shader.frag");
    glUseProgram(shader_program);
    GLint pos = glGetAttribLocation(shader_program, "vertPosition");
    glVertexAttribPointer(pos, 4, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(pos);

    // bucle de eventos
    while (not glfwWindowShouldClose(window)) {
        glClear(GL_COLOR_BUFFER_BIT);
        glDrawArrays(GL_TRIANGLES, 0, 3);
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    /* ...liberar recursos... */
}
```

Con `glUseProgram` activamos un par (fragment+vertex) identificado por el id que nos retornó `glCreateProgram`. Recordemos que el program representa un cierto pipeline, y podemos tener varios compilados y enlazados e ir alternando cual usamos para cada conjunto de primitivas. En este caso, lo activamos una sola vez fuera del bucle porque, al ser el único, permanecerá siempre activo. El otro

paso necesario luego de activar el shader es conectar los VBOs con las entradas del vertex-shader. Recordemos que el vertex-shader puede tener varias entradas, y que podemos haber cargado a la GPU varios VBOs; entonces hay que asociar cuál se corresponde con cual (hasta podría ser que en un mismo VBO tengamos mezclados datos para varias entradas). Para eso son las llamadas a `glGetAttribLocation` (para identificar la entrada denominada `vertPosition` dentro del vertex-shader), a `glEnableVertexAttribArray` (para habilitar esa entrada), y a `glVertexAttribPointer` (para establecer la asociación y especificar qué tipo de información tiene el buffer).

En este punto, al ejecutar el programa ya deberíamos ver el triángulo rojo en el centro de la ventana:



10. Programación de shaders (II)

Para finalizar el ejemplo 2D, vamos a hacer que los vértices tengan colores propios (para el ilustrar el uso de otras propiedades en los vértices y ver cómo pasan al fragment-shader), y que el triángulo se mueva (para ilustrar el uso de `uniform`).

Para que cada vértice tenga un color propio, debemos hacer un arreglo con 3 colores y cargarlo a la GPU. Podría añadirse al VBO que ya tenemos (poniendo primero todas las posiciones y luego todos los colores, o también intercalandolos, ambas opciones son posibles); o también podemos usar para ello un VBO nuevo. Programamos la segunda alternativa:

```
int main() {

    /* ...inicializar glfw, crear ventana, inicializar glew... */

    // datos de los vertices
    std::vector<glm::vec4> vPos = {
        { -0.5, -0.5, 0.0, 1.0 },
        { +0.5, -0.5, 0.0, 1.0 },
        {  0.0, +0.5, 0.0, 1.0 } };
    std::vector<glm::vec4> vColor = {
        { 1.0, 0.0, 0.0, 1.0 },
        { 0,0, 1.0, 0.0, 1.0 },
        { 0.0, 0.0, 1.0, 1.0 } };

    // crear el vao
    GLuint vao;
    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);

    // crear 2 vbos
    GLuint vbos[2];
    glGenBuffers(2,vbos);
    glBindBuffer(GL_ARRAY_BUFFER,vbos[0]);
    glBufferData(GL_ARRAY_BUFFER, vPos.size()*sizeof(glm::vec4), vPos.data(), GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER,vbos[1]);
    glBufferData(GL_ARRAY_BUFFER, vColor.size()*sizeof(glm::vec4), vColor.data(), GL_STATIC_DRAW);

    // compilar y configurar shadersvPos
    GLuint shader_program = load_shaders("shader.vert","shader.frag");
    glUseProgram(shader_program);

    glBindBuffer(GL_ARRAY_BUFFER,vbos[0]);
    GLint loc_position = glGetAttribLocation(shader_program, "vertPosition");
    glVertexAttribPointer(loc_position, 4, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(loc_position);

    glBindBuffer(GL_ARRAY_BUFFER,vbos[1]);
    GLint loc_color = glGetAttribLocation(shader_program, "vertColor");
    glVertexAttribPointer(loc_color, 4, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(loc_color);

    /* ...bucle principal... */

    glDeleteBuffers(2,vbos);
    glDeleteVertexArrays(1,&vao);
    glfwTerminate();
}
```

Notar que al asociar una entradas del vertex-shader con un VBO, la función que lo hace no recibe el id del VBO, sino que utiliza el VBO activo, por lo que ahora que hay 2 VBOs, debemos activar en cada caso el que corresponda antes de usarlo (con `glBindBuffer`).

Los nuevos shaders, con este agregado, quedarían como sigue:

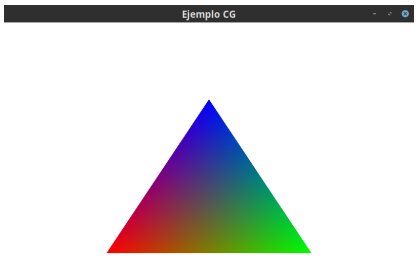
```
#version 150 core

in vec4 vertPosition;
in vec4 vertColor;

out vec4 color;

void main() {
    color = vertColor;
    gl_Position = vertPosition;
}
```

El vertex-shader tiene una entrada más (`vertColor`), y también una salida más (`color`). El color de entrada se traslada directamente al de salida. De esta forma cada vértice llega a la etapa de rasterización con un color asociado, y esto hará que cada fragmento que se genere tenga también un color asociado. Dicho color saldrá de la interpolación (promedio ponderado) de los colores de los vértices.

```
#version 150 core

in vec4 color;

out vec4 outColor;

void main() {
    outColor = color;
}
```

El fragment-shader ahora toma ese color (interpolado en la rasterización) como entrada y es el que pasa directamente a la salida.

Con estos cambios, deberíamos ver el triángulo con sus extremos de colores y un degradé en el interior:



Finalmente, vamos a hacer que el vertex-shader transforme los vértices, según una matriz[3] que recibirá como `uniform` (la misma para todos los vértices). Para generar una matriz que se corresponda con una transformación, usaremos la función `glm::rotate` de la biblioteca GLM (que recibe una matriz incial, en

[3] Es común en computación gráfica representar transformaciones como matrices de 4x4; los detalles se desarrollarán en la teoría de transformaciones.

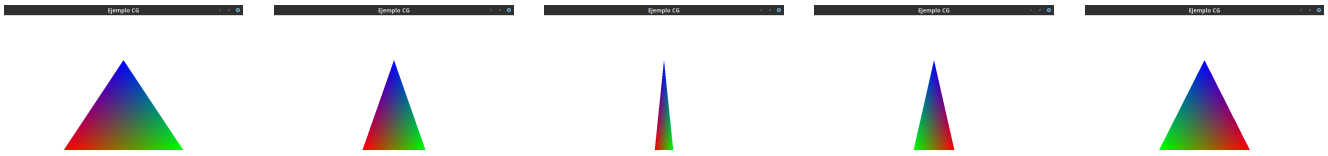
este caso la identidad, luego el ángulo a rotar en radianes, y finalmente el vector eje sobre el cual rotar). Para pasarle la matriz al shader, primero identificamos la entrada con `glGetUniformLocation`, y luego la asignamos con `glUniformMatrix4fv`:

```
int main() {

     /* ...incializiaciones... */

    GLint loc_matrix = glGetUniformLocation(shader_program, "rotMatrix");

    // main event-loop
    while (not glfwWindowShouldClose(window)) {

        glClear(GL_COLOR_BUFFER_BIT);

        glm::mat4 m = glm::rotate(glm::mat4(1.0), (float)glfwGetTime(), glm::vec3(0.0,1.0,0.0));
        glUniformMatrix4fv(loc_matrix,1,GL_FALSE,&m[0][0]);

        glDrawArrays(GL_TRIANGLES, 0, 3);

        glfwSwapBuffers(window);

        glfwPollEvents();
    }

    /* ...liberar recursos... */
}
```

El código del nuevo vertex shader quedaría como sigue:

```
#version 150 core

in vec4 vertPosition;
in vec4 vertColor;

uniform mat4 rotMatrix;

out vec4 color;

void main() {
    color = vertColor;
    gl_Position = rotMatrix*vertPosition;
}
```

Ahora, al ejecutar el programa, el triángulo debería girar:



11. Continuará

Podría hacer un 2do apunte con los shaders básicos, el vertex con las 3 matrices y el fragment con phong.