Computación Gráfica - Código base para TPS

El ejemplo *base* muestra cómo cargar y visualizar un modelo utilizando *OpenGL* y una biblioteca de clases y funciones que provee la cátedra. Para poder desarrollar los próximos trabajos prácticos, alcanza con que el alumno analice y entienda el código de `main.cpp`. No es necesario profundizar en demasiados detalles del funcionamiento interno de las mencionadas clases y funciones, sino solo saber para qué sirven y algunos nociones básicas de *OpenGL*. Este documento describe lo necesario para analizar el funcionamiento del ejemplo base.

1. Compilación

Pero lo primero lo primero, que es que el estudiante pueda compilar el proyecto. El código utiliza las bibliotecas *OpenGL*, *glm*, *GLFW*, *ImGUI*, *glad* y *stb_image*. Las últimas 3 (*ImGUI*, *glad* y *stb_image*) ya están incluidas en el directorio `common/third`; las otras tres (*OpenGL*, *glm* y *GLFW*), deberán instalarse por separado.

Si utiliza *GNU/Linux*, deberá instalarlas por su cuenta. Usualmente es simple hacerlo utilizando el gestor de paquetes de su distribución. Recuerde que al buscar los paquetes de las bibliotecas debe instalar las versiones de desarrollo (en la mayoría de las distribuciones se distinguen porque agregan *-dev* o *-devel* al nombre)[1]

Si utiliza Windows, el proceso de instalación y configuración dependerá del IDE y del compilador utilizado.

Para quienes utilicen *ZinajI*, los prácticos incluirán los archivos de proyecto con las configuraciones necesarias para compilarlos. Además, para quienes utilicen *ZinjaI* en *Windows*, se puede descargar del aula virtual un archivo de complemento con las bibliotecas necesarias [2]

Quienes quieran utilizar otro IDE y configurar su propio proyecto pueden preguntar a los docentes en clases o utilizar los foros de consultas para resolver los posibles errores de compilación.

1.1. Sobre el error 0x10007 de GLFW

Si puede compilar exitosamente el proyecto, pero al ejecutarlo se encuentra con este error:

```
ERROR: GLFW code 0x10007: WGL: Failed to create OpenGL context
```

el problema es que su sistema parece no soportar la versión necesaria de *OpenGL*. Esto podría pasar si el sistema o el hardware son demasiado antiguos (más de 10 años), o más frecuentemente si el driver no está correctamente instalado o actualizado.

Si está intentando ejecutar el programa en una máquina virtual (como *VirtualBox*), la limitación puede estar en la propia máquina virtual. Si el sistema virtualizado es Windows, puede solucionarlo descargando el archivo *opengl32.dll* desde https://fdossena.com/?p=mesa/index.frag[3].

[1] Puede aprovechar para instalar también *FreeGLUT*, que si bien no es necesaria para ejecutar este código, sí lo será para algunos demos y programas de ejemplo que se entregarán como material de estudio adicional en otras unidades.

[2] Para instalar un complemento en ZinaI puede arrastrar el archivo .zcp a la ventana de ZinjaI, o usar la opción ”Instalar complementos...” del menú ”Herramientas”

[3] Si utiliza *ZinjaI* en *Windows*, por defecto el toolchain utilizado para compilar será de 32bits (aunque se instale en un sistema operativo de 64). Por esto, al descargar el archivo *opengl32.dll* (el *dll* siempre tiene este nombre, aún en su versión de 64bits), seleccione la versión de 32bits.

2. ¿Qué hace cada biblioteca?

2.1. OpenGL

Simplificando un poco, *OpenGL* sirve para enviar información a la GPU y lograr dibujar cosas en una ventana. Sin embargo, es importante resaltar que solo tiene funciones para dibujar en una ventan (técnicamente deberíamos decir en un *contexto OpenGL*), pero no tiene funciones para crear esa ventana, ni para gestionar sus eventos (mouse, teclado, etc). Más aún, solo permite *dibujar* mediante primitivas, pero no tiene controles/widgets de alto nivel (cuadros de texto, botones, menúes, etc). Por eso se utiliza en combinación con otras bibliotecas.

2.1.1. OpenGL y el *estado global*

Al utilizar *OpenGL* es importante mencionar que funciona como una máquina de estado, con un estado global (técnicamente uno por cada *contexto*) compartido por todas sus funciones. Por ejemplo, para *borrar* la pantalla (esto sería, pintar todo de un *color de fondo*) se utiliza una función que borra (`glClear(...)`) y se le indica que lo que queremos borrar es el *buffer de color* (pasándole `GL_COLOR_BUFFER`) pero no se le dice con qué color de fondo rellenar el buffer. Ese color está definido en el *estado* (puede pensarlo como una configuración global), y puede cambiarse con otra función específica para ello (`glClearColor(...)`). Esto evita que se deba repetir información que no cambia (si el fondo va siempre del mismo color, no hay que decirlo en cada *frame*), o que las funciones deban tener decenas de argumentos. En general las funciones de OpenGL son simples (reciben pocos argumentos) porque toman muchas cosas de ese *estado*.

2.1.2. Shaders y GLSL

Cuando se utilizan las versiones modernas de *OpenGL* (lo que se denomina perfil *core*), la responsabilidad de definir las etapas del *pipeline* de procesamiento de vértices y de procesamiento de fragmentos recaen totalmente sobre el programador [4]. El programador debe proveerle a la GPU el código de los programas que ejecutará en estas etapas. A estos programas para las etapas del pipeline que ejecuta la GPU se los denomina *shaders*, y tienen un lenguaje de programación propio. Junto con *OpenGL*, el lenguaje de *shaders* que se utililza es *GLSL*, y guarda ciertas similaridades con C y C++, por lo que aprendiendo unos pocos conceptos básicos podrá entenderlo sin mayores problemas.

2.2. glm

La biblioteca *glm* provee clases para representar vectores y matrices en varias dimensiones (2, 3 o 4), y también provee funciones y sobrecargas de operadores para las operaciones algebraicas más habituales. Los programas en 3D hacen un uso intensivo de estos elementos. La biblioteca *glm* los modela imitando el comportamiento del lenguaje *glsl* (los tipos tienen los mismos nombres, las funciones libres también, y la forma en que los datos se organizan en memoria en cada clase también es la misma para poder enviar las instancias directamente a la GPU).

Además, la biblioteca incluye funciones específicas para el uso de vectores y matrices para renderizar; muchas de las cuales estaban disponibles dentro de *OpenGL* (o de una biblioteca auxiliar *GLU*) para las primeras versiones, y fueron eliminadas luego. Por ejemplo, cuando analicemos las matrices y transformaciones que se usan para

[4] En el perfil de *compatibilidad* las etapas de procesamiento de vértices y de procesamiento de fragmentos tienen una funcionalidad fija predefinida, y el programador solo debe hacer ciertas configuraciones sobre las mismas. El cambio de perfil de *compatibilidad* a *core* se da en la versión 3.2 de OpenGL.

manipular la cámara y la posición de los objetos, veremos que hay funciones dentro de *glm* para armar esas matrices en particular.

2.3. GLFW

Esta biblioteca será la encargada de crear una ventana útil para *OpenGL* y de detectar los eventos sobre la misma (pulsaciones de teclas, clicks y movimientos del ratón, cambios de tamaño de la ventana, etc)[5].

2.3.1. Callbacks

GLFW (y también *GLUT*) utiliza el concepto de *callback* para programar el comportamiento de sus ventanas. Un *callback* no es más que una función con cierto prototipo específico que la biblioteca invocará en caso de ocurrir un evento. Por ejemplo, si queremos en nuestra aplicación hacer algo cuando el usuario hace click, programamos eso en una función y le avisamos a *GLFW* que queremos que invoque a esa función cuando detecte un click. A esto se le dice *registrar el callback del mouse*. Por esto, cuando se crea una ventana, usualmente a continuación encontrará las llamadas a las funciones que registran los callbacks (que le pasan a GLFW los punteros a las funciones que queremos que use).

En la mayoría de los prácticos la ventana mostrará un modelo 3D que podremos manipular de forma muy básica con el ratón. Para esto, estarán programadas las acciones necesarias en dos funciones que oficiarán de *callbacks* del ratón: una para el evento del click (usualmente llamado *mouse*) y otra para el evento del moviemiento del puntero (usualmente llamado *motion*). Como los TPs comparten ese comportamiento, esto se ha implementado una vez en los archivos `common/utils/Callbacks.{cpp,hpp}`, y casi todos los tps las utilizarán simplemente invocando a la función `setCommonCallbacks` para que los registre luego de crear la ventana. A continuación, cada TP podrá registrar *callbacks* adicionales (por ej, el del teclado).

2.4. ImGUI

Como se mencionó antes, no hay en *OpenGL* ni en *GLFW* facilidades para poner controles más abstractos como cuadros de texto, selectores de colores, barras de desplazamiento, botones, menúes, etc en la aplicación. Una solución a esto sería embeber OpenGL en alguna biblioteca de ventanas más completa que *GLFW* (como por ejemplo *QT* o *wxWidgets*), pero eso agregaría cierta complejidad propia de esas bibliotecas que no queremos tener en estos TPs (que son muy simples, y generalmente buscan que el alumno se enfonquen solo en la parte relacionada a OpenGL). Una solución intermedia es la biblioteca *ImGUI*, y será utilizada en los prácticos para colocar sobre el dibujo del modelo una pequeña ventana flotante con algunas opciones de configuración.

2.5. Otras

No es necesario en este punto entrar en detalle sobre las demás bibllliotecas utilizadas, pero podemos decir que *GLAD* colabora con la inicialización de *OpenGL* (en particular con lidiar con el driver y determinar qué funcionalidades ”avanzadas” de OpenGL soporta), y que *stb_image* simplifica la lectura archivos de imagen en formatos comprimidos (como *png* o *jpeg*).

[5] En códigos más viejos se utilizaba en lugar de *GLFW*, para esta misma función, la biblioteca *GLUT* (o más habitualmente su alternativa *FreeGLUT*). Todos los TPs donde el alumno deba programar utilizarán *GLFW*, pero algunos programas de ejemplo que el alumno tendrá a disposición para complementar el estudio de algunos temas han sido desarrollados con *FreeGLUT*.

El resto de las clases y cabeceras a las que hacen mención los códigos se encuentran en el directorio `common/utils` y han sido desarrolladas por la cátedra.

3. Las clases en `common/utils`

En el directorio `common/utils` hay clases y funciones que simplifican ciertas tareas tediosas y repetitivas que casi todo TP necesita, o que implementan ciertos algoritmos complejos para que el alumno no deba enfrentarse a ciertos problemas antes de tiempo.

3.1. La ventana

`Window` es una clase para simplificar la creación/destrucción una ventana *GLFW* (incluyendo la incialización/cierre de *OpenGL*, *GLFW* y *GLAD*, y opcionalmente lo necesario para usar *ImGUI* en esa ventana). Con crear una instancia de esta clase, ya se está creando una ventana con una configuración básica suficiente para comenzar a utilizarla. Usualemente, luego de crear la ventana se registran los *callbacks*. Como se mencionó antes, la mayoría de los TPs utilizarán una función de `common/utis/Callbacks.hpp` que define *callbacks* para la manipulación básica de la vista 3D, y luego uno o más *callbacks* específicos del TP (como por ejemplo el de teclado, que varía de un TP a otro).

Luego de creada la ventana, ya se dispone de un *contexto* OpenGL válido, por lo cual se pueden empezar a utilizar las funciones de OpenGL. Lo habitual es a continuación configurar el estado inicial, y pasar a cargar los modelos.

Finalmente, lo habitual es que la función `main` termine con el bucle principal de la aplicación (o también llamado *bucle de eventos*), bucle que se repetirá hasta que la aplicación se cierre. En cada iteración del bucle se actualiza el contenido de la ventana y se da pie a la biblioteca *GLFW* para que detecte los eventos e invoque a los callbacks que corresponda (con `glfwPollEvents()`).

Algunos TPs utilizan la clase `FrameTimer` (también definida en `common/utils`), para medir o controlar el tiempo entre un frame y otro (es decir, la velocidad a la que itera el bucle principal).

3.2. El modelo

Para mostrar un modelo en pantalla es necesario:

1. **Cargarlo de algún archivo (o varios) en disco.** Hay muchos formatos para guardar modelos y escenas. En los TPs utilizaremos el formato *Wavefront obj* (archivos *.obj* y *.mtl*). La clase `ObjMesh` es la encargada de leer y procesar esos archivos.
2. **Almacenar esa información en un formato útil para los fines de nuestro programa.** La clase `Geometry` define un formato simple y útil para nuestros TPs. No es exactamente la misma información que se lee del *.obj*, por eso en general luego de leerla se covierte a una instancia de `Geometry` con la función `toGeometry`. La conversión, además ”reorganizar” la inforamción, también puede implicar completar información faltante (por ejemplo, una malla en un archivo *.obj* podría no tener información de normales por nodo, y en ese caso esta se calcula al convertir de `ObjMesh` a `Geometry`).
3. **Enviar esa información a la GPU.** La clase `Geometry` tendrá en sus atributos los datos del modelo en un formato que la GPU puede aceptar. Para ver ese modelo en pantalla se debe enviar esa información a la GPU. Esto es, usar funciones de *OpenGL* para crear *buffers* de memoria de video, y copiar la información de RAM a esos *buffers*. Las funciones de *OpenGL* retornan un *ID* por cada buffer creado, que se usa luego para referenciarlo. La clase `GeometryRenderer` tiene funciones para enviar una geometría (a partir de una `Geometry`)

a la GPU, creando los *buffers* necesarios y guardando los correspondientes *IDs* para luego poder actualizarlos o liberarlos cuando ya no se utilicen.

La clase `Model` encapsula las 3 etapas e incluye además una textura (mediante una instancia de `Texture`) si es neceseria. Las funciones `Model::load` y `Model::loadSingle` leen los archivos *.obj* usando una instancia de `ObjMesh`, los convierten a instancias de `Geometry` y luego a `GeometryRenderer`, para guardar esto último en una o varias instancia de `Model`.

Al leer un objeto desde un *.obj*, se lee también su material. El material se representa con la clase `Material`, y consiste en un conjunto de valores para configurar la ecuación del modelo de iluminación (lo analizaremos en detalle más adelante en la unidad de color e iluminación), y puede incluir una textura (imagen para *pegar* sobre el objeto en lugar de pintarlo de un único color). La textura se debe leer de un archivo de imagen (por ej *.jpeg* o *.png*) para cargar en RAM descomprimida, enviar a la GPU, y gestionar esa memoria de la GPU mediante su *ID*. Todas estas operaciones están encapsuladas dentro de la clase `Texture`.

Finalmente, cabe mencionar que en una archivo *.obj* puede haber una o varias *partes* de un modelo. Esto es porque cada *parte* tiene un único material (color o textura). Entonces, si un objeto tiene partes de diferentes materiales, en el *.obj* se representan como mallas diferentes. La clase `Model` representa en realidad a una sola parte. Si el objeto tiene varias partes, la función `Model::load` retorna un vector de instancias de `Model`, cada una conteniendo las primitivas y el material (y textura eventualmente) de una parte.

El programa cliente generalmente utilizará a la clase `Model` para leer y mostrar un modelo, y no requerirá interactuar con `ObjMesh`, `Geometry` o `GeometryRenderer`. Solo se hará referencia a las últimas dos cuando el TP deba modificar al modelo (por ejemplo, mover los vértices de las primitivas).

3.3. Los Shaders

Para usar un shader, hay que avisarle al driver de video que queremos crear uno (nos dará un *ID* para referirnos al mismo), enviarle el código fuente, pedirle que lo compile, y verificar que no se hayan generado errores. Para completar un pipelina válido, se deben compilar y enlazar por los menos dos shaders, el *vertex shader* (archivo *.vert*) y el *fragment shader* (archivo *.frag*). Se debe pensar cada uno como si fueran dos partes de un mismo programa. La salida del primero define la entrada del segundo. Todo el proceso de cargarlos, compilarlos, enlazarlos, reportar los errores, o activarlos para renderizar, está ya implementado en la clase `Shader`. Esta clase recibe en su constructor las rutas a los fuentes (*.vert* y *.frag*) y los compila (el programa se detiene si hay errores). Al momento de usarlos, puede ser necesario que el programa cliente defina el valor de ciertas variables de entrada de los shaders. A estas variables se las denomina *uniforms*, y la clase `Shader` tiene métodos para cambiar estos valores.

Usualmente un shader *parece* un programa C o C++ (hay una función `main` y eventualmente puede haber antes otras funciones auxiliares) que hace uso de un conjunto de pseudo-variables *globales* para la entrada y salida. Cada variable *global* tiene un calificativo en su definición que dice para qué se usa. Puede ser:

- `in`: es un valor que cambia con cada vértice (en el caso del *vertex shader*) o con cada fragmento (en el caso del *fragment shader*). El programa principal le envía a la gpu un vector de datos (por ejemplo de posiciones de vértices), y el *vertex shader* se ejecuta una vez por cada uno de ellos, y en cada vez, la variable *in* contiene a ese elemento. Además de las posiciones de los vértices (que no pueden faltar) puede haber otra información de entrada (como las normales de cada vértice, o las coordenadas de textura). Análogamente, el *fragment shader* se ejecuta una vez por cada fragmento, y las variables `in` pueden tener un valor diferente para cada uno de ellos. Las variables in del *fragment shader* tienen que coincidir con las `out` del *vertex shader*.

- `out`: son los datos de salida del shader. En el caso del *vertex shader*, es información que va asociada al vértice, y que luego se transfiere a los fragmentos (por eso el `out` del *vertex shader* termina siendo el `in` del *fragment shader*). En el caso del *fragment shader*, el `out` es el color final con el que se pintará el pixel correspondiente al fragmento.
- `uniform`: las variable uniform son datos de entrada para el programa (puede ser en cualquiera de los *shaders*) que no varían por cada vértice o fragmento, sino que tienen el mismo valor para todos ellos. Si una geometría tiene por ejemplo 100 vértices, por cada `in` tenemos que enviar 100 datos al *vertex shader*, pero un valor `uniform` se envía una sola vez y se usa ese mismo para las 100 ejecuciones.

Dentro de las funciones del shader, se suelen utilizar los tipos de datos vec3, vec4, mat3, mat4 para representar vectores (en el sentido de matemática, no de programación) y matrices de 3 y 4 dimensiones. Se pueden pensar como clases con varios de sus operadores sobrecargados para realizar las operaciones matemáticas habituales. Y además se dispone de una biblioteca de funciones libres que los complementa (pero no es necesario usar `#include`[6] en *GLSL*).

4. Más información

El directorio *docs* contiene documentación más detallada sobre el conjunto de clases desarrolladas por la cátedra. No es necesario que el alumno analice esa documentación para resolver los trabajos prácticos; pero está allí por si tiene curiosidad o necesita hacer algo diferente (por ejemplo, para su proyecto final). Sin embargo, dado que la biblioteca se ha desarrollado recientemente y todavía está cambiando mucho, esta documentación podría no estar suficientemente completa o actualizada. En caso de requerir mayor ayuda consulte a los docentes de la cátedra.

Para el resto de las bibliotecas, puede encontrar ayuda en las fuentes habituales (por ej, encontrar las referencias oficiales *googleando*, o preguntas particulares sobre el uso en sitios como *stackoverflow*).

[6] La directiva `#include` no existe en GLSL básico. Sin embargo, la verá utilizada en algunos shader de los TPs. Es la clase *Shader* la que hace la inclusión (busca el archivo de cabecera y reemplaza la linea del `#include` por el contenido del archivo) antes de enviar el código al driver de video.