Programación 3D

Práctica 1: Rotating triangles

En esta primera práctica, a partir de la plantilla realizada en clase, vamos a dibujar con OpenGL una serie de triángulos en pantalla. Implementaremos clases para la gestión de los vértices, buffers y shaders, y utilizaremos GLM para realizar las transformaciones necesarias para dibujar los objetos.

Clase Vertex

Los datos de un vértice de cualquier geometría que dibujemos con el motor van a estar contenidos en objetos de esta clase. Todas sus propiedades van a ser públicas (con lo que podemos implementarla como un `struct`), y únicamente necesitamos por el momento guardar la **posición** del vértice (por ejemplo, en un dato de tipo `glm::vec3`).

Clase Shader

Vamos a crear una clase `Shader`, que realmente maneja un programa de GPU de OpenGL (con su vertex y su fragment shader). Permite además escribir los valores de las variables uniformes del shader. Cuenta con los siguientes métodos:

- ```
  // Devuelve el identificador de OpenGL del programa
  uint32_t getId() const;
  ```

- ```
  // Obtiene el mensaje de error generado al compilar o enlazar
  const char* getError() const;
  ```

- ```
  // Activa el uso de este programa
  void use() const;
  ```

- ```
  // Activa la escritura de las variables attribute,
  // y especifica su formato
  void setupAttribs() const;
  ```

Cuando se enlaza un buffer y se activa un shader, se debe indicar en qué variables `attribute` del shader se va a escribir, y cómo están configurados estos datos en el buffer de vértices enlazado. Para cada variable `attribute` cuya localización no sea -1 (es decir, que exista en el shader), activamos su escritura en el shader con `glEnableVertexAttribArray`, e indicamos su configuración con `glVertexAttribPointer`.

```
// Obtiene la localización de una variable uniform
int getLocation(const char* name) const;

// Da valor a una variable uniform
void setInt(int loc, int val);
void setFloat(int loc, float val);
void setVec3(int loc, const glm::vec3& vec);
void setVec4(int loc, const glm::vec4& vec);
void setMatrix(int loc, const glm::mat4& matrix);
```

Añadiremos las variables miembro necesarias, un constructor que reciba el código del vertex y el fragment shader, y un destructor si es necesario. En el constructor, se debe obtener y almacenar la localización de todas las variables `attribute` del shader.

Los métodos `setInt`, `setFloat`, etc, deben comprobar que la localización en que se va a escribir no sea -1 antes de hacerlo.

A la hora de crear un shader, es más cómodo escribir su código en ficheros y cargar estos ficheros en un string antes de pasarlo al constructor de `Shader`. Podemos implementar una función que realice la carga del fichero usando STL:

```
std::string readString(const std::string& filename) {
	std::ifstream istream(filename.c_str(), std::ios_base::binary);
	std::stringstream sstream;
	sstream << istream.rdbuf();
	return sstream.str();
}
```

Clase Buffer

La información de la geometría a pintar va almacenada en buffer de VRAM (uno de vértices y otro de índices). Vamos a crear una clase `Buffer` que contendrá un paquete con los identificadores de ambos buffers de OpenGL:

- El buffer de vértices contiene todos los objetos `Vertex` con los vértices del objeto.
- El buffer de índices contiene todos los índices (de tipo `uint16_t`) para pintar la geometría en modo `GL_TRIANGLES`.

En el constructor, debemos pasarle los arrays de vértices e índices, a partir de los cuales generará los buffers de OpenGL. Se debe implementar además el siguiente método:

- `void draw(const Shader& shader) const;`

Éste enlazará los dos buffers de OpenGL contenidos en el objeto, llamará al método `setupAttribs` del shader para que se configuren las variables `attribute` del shader (asumimos que el shader está siendo usado), y dibujará la geometría de los buffers en modo `GL_TRIANGLES`.

Implementar las variables miembro que consideremos necesarias, además de constructor y destructor (si lo necesita).

Programa principal

Utilizaremos la plantilla realizada en clase que crea una ventana con GLFW y pone en marcha el bucle principal. Antes del bucle principal (pero después de crear la ventana y activar el contexto de OpenGL), realizaremos las siguientes tareas:

Inicialización del motor

Podemos crear una función `init` que realice las siguientes tareas:

- **Inicialización de las extensiones de OpenGL**. La cabecera por defecto de OpenGL que traen los compiladores de Windows utilizan una versión muy antigua de la API. Para acceder a la última funcionalidad, existen librerías como **GLEW**. Debemos incluir **"glew.h"** y llamar a `glewInit` (que debe devolver `GLEW_OK`) antes de utilizar OpenGL.
- **Inicialización de estados**. Debemos activar los siguientes estados de OpenGL:
  - `GL_DEPTH_TEST`
  - `GL_SCISSOR_TEST`

La función puede devolver un booleano indicando si GLEW se inicializó correctamente.

Creación de los shaders

Crearemos un objeto `Shader` con un vertex shader y un fragment shader. Activaremos su uso. El vertex shader tendrá la matriz MVP y la posición del vértice, y realizará la transformación estándar. El fragment shader pintará todos los fragmentos de blanco.

Creación del buffer del triángulo

Crearemos un objeto `Buffer` con los datos de vértices e índices de un triángulo.

Bucle principal

Debemos crear una matriz de proyección con perspectiva, y una matriz vista que coloque la cámara en 0, 0, 6 mirando hacia el origen de la escena.

Vamos a pintar un total de 9 triángulos (todos utilizando el mismo buffer), ordenados en tres filas (en las posiciones Z 0, -3, -6 respectivamente), con tres triángulos en cada fila (en las posiciones X -3, 0 y 3 respectivamente). Antes del pintado, borraremos los buffers de color y profundidad.

Cada uno de los triángulos rotará sobre su eje Y a una velocidad de 32 grados por segundo. La matriz MVP debe ser calculada y pasada al shader antes del pintado de cada objeto.

La imagen muestra una solución correcta a la práctica: