Programación 3D

Práctica 2: El mundo y sus entidades

En la segunda práctica, vamos a añadir algunas clases al motor de forma que la transformación de los elementos no deba realizarse mediante manipulaciones de matrices, sino de forma más intuitiva (con vectores para la posición y escala, y un vector o cuaternión para la rotación). Igualmente, la cámara se manejará mediante una clase en lugar de sus matrices de proyección y vista, y todos los elementos se contendrán en una clase World que representa nuestro mundo.
Reescribiremos la práctica anterior para utilizar las nuevas clases.

Clase State

Cuando los objetos trabajan (especialmente durante el renderizado), van a establecer valores que necesitarán ser consultados por otros objetos. Por ejemplo, la cámara va a calcular unas matrices vista y proyección, y al dibujar un modelo combinaremos estos con la transformación del elemento para obtener la matriz MVP. Por ello, vamos a implementar un sistema muy sencillo para que los objetos puedan acceder a los datos generales del estado del motor: una clase State con variables **miembro públicas estáticas** (de forma que cualquier objeto pueda acceder a estos datos). Por el momento, tendrá las siguientes variables (aunque irá creciendo según progresemos en la asignatura):

```
static std::shared_ptr<Shader>      defaultShader;
static glm::mat4                    projectionMatrix;
static glm::mat4                    viewMatrix;
static glm::mat4                    modelMatrix;
```

En el caso de defaultShader, vamos a guardar un shader por defecto que se utilizará en todos los objetos que pintemos (a no ser que un objeto específico indique que quiere utilizar un shader diferente). Este shader será el que empleamos en la práctica anterior (aunque irá creciendo en las
prácticas siguientes), y deberíamos crearlo en la función init que inicializa el motor (se implementó en la primera práctica).

Clase Mesh

En la práctica anterior, habíamos implementado una clase Buffer con las listas de vértices de índices para dibujar una determinada geometría. Objetos más complejos van a necesitar varios buffers para pintarse (por ejemplo, cuando metamos soporte de texturas, la geometría que se pinta con cada textura debe ir en un buffer separado). Por ello, en nuestro motor representaremos una malla 3D mediante la clase Mesh, que puede contener uno o varios buffers según necesite. Tendrá los siguientes métodos (además de las variables miembro que necesitemos):

- `void addBuffer(const std::shared_ptr<Buffer>& buffer, const std::shared_ptr<Shader>& shader = nullptr);`

Con el método addBuffer, añadimos un nuevo buffer a la malla. Cada buffer puede pintarse con un shader diferente, con lo que pasamos un puntero al mismo al añadir el buffer (opcional, si no se pasa un shader se utilizará el shader por defecto para el pintado). Debemos almacenar todos los buffers de la malla con sus respectivos shaders.

- `size_t getNumBuffers() const;`
- `const std::shared_ptr<Buffer>& getBuffer(size_t index) const;`
- `std::shared_ptr<Buffer>& getBuffer(size_t index);`
- `void draw();`

En el método draw, pintaremos cada buffer, activando previamente su shader (en el caso de que tenga; si no, activaremos State::defaultShader), y escribiendo la matriz MVP en el mismo. La matriz MVP debemos calcularla a partir de las variables estáticas projectionMatrix, viewMatrix y modelMatrix de la clase State.

Clase Entity

Todo elemento presente en el mundo será una entidad. Dispone de los siguientes métodos, y le añadiremos las variables miembro necesarias:

- `Entity();`
- `virtual ~Entity() {}`
- `const glm::vec3& getPosition() const;`
- `void setPosition(const glm::vec3& pos);`
- `const glm::vec3& getRotation() const; // puede ser quat`
- `const void setRotation(const glm::vec3& rot); // puede ser quat`
- `const glm::vec3& getScale() const;`

- `void setScale(const glm::vec3& scale);`
- `void move(const glm::vec3& vec); // ver en las diapositivas`
- `virtual void update(float deltaTime) {}`
- `virtual void draw() {}`

Como vemos, la funcionalidad base de una entidad es manipular su posición, rotación y escala. Tiene además métodos update y draw que permitirán al mundo actualizar y dibujar la entidad (cada subclase redefinirá estos métodos si es necesario).

Clase Model

Esta subclase de Entity se utilizará para dibujar mallas en el mundo. Tendrá los siguientes métodos:

- `Model(const std::shared_ptr<Mesh>& mesh);`
- `virtual void draw() override;`

La malla a pintar se indica en la construcción, y no puede ser modificada después (aunque podríamos añadir un método set si deseamos esta funcionalidad).

En el método draw, estableceremos la matriz State::modelMatrix a partir de la posición, rotación y escala del modelo, y a continuación llamaremos al método draw de la malla.

Clase Camera

Ésta será una subclase de Entity, que definirá los siguientes métodos (crear las variables miembro correspondientes, y constructor y destructor si es necesario):

- `const glm::mat4& getProjection() const;`
- `void setProjection(const glm::mat4& proj);`
- `const glm::ivec4& getViewport() const;`
- `void setViewport(const glm::ivec4& vp);`
- `const glm::vec3& getClearColor() const;`
- `void setClearColor(const glm::vec3& color);`
- `void prepare();`

Vamos a explicar el método prepare. Éste lo utilizará la clase World (que veremos en un momento) para preparar el proceso de pintado. Se realizará esta llamada antes de dibujar la entidades del mundo. Se debe hacer lo siguiente:

1. Pasar a State::projectionMatrix la matriz que se estableció con setProjection.
2. Calcular la matriz State::viewMatrix con la posición y rotación de la cámara.
3. Establecemos el viewport con las funciones glViewport y glScissor, utilizando los datos establecidos con setViewport.
4. Establecemos como color para limpiar el fondo el establecido por la función setClearColor, y borramos los buffers de color y profundidad.

NOTA: Para calcular la matriz modelo de un elemento de la escena, efectuamos las siguientes transformaciones: Escalado, Rotación, Traslación -que se realizan en orden inverso en OpenGL-. Para calcular la matriz vista, en cambio, hay que tener en cuenta las siguientes consideraciones:

- No aplicamos escala, con lo que las transformaciones a aplicar son Rotación, Traslación.
- Hacer Rotación, Traslación de la cámara es en realidad hacer Traslación, Rotación de los vértices a pintar, con el vector de traslación opuesto al de la cámara, y rotando en el sentido contrario.

Clase World

Esta clase contiene información sobre el mundo. Básicamente, contiene todas las entidades, y métodos para actualizarlas y dibujarlas. Necesitamos los siguientes métodos:

- `void addEntity(const std::shared_ptr<Entity>& entity);`

El método addEntity debe añadir la entidad a una lista. Además, mediante std::dynamic_pointer_cast, comprobaremos si la entidad es una cámara, en cuyo caso la añadiremos también a una lista de cámaras, de forma que podamos acceder inmediatamente a todas las cámaras presentes en el mundo.

- `void removeEntity(const std::shared_ptr<Entity>& entity);`

El método removeEntity hace la operación inversa.`size_t getNumEntities() const;`

- `const std::shared_ptr<Entity>& getEntity(size_t index) const;`
- `std::shared_ptr<Entity>& getEntity(size_t index);`
- `void update(float deltaTime);`

El método update debe actualizar todas las entidades (llamando al método update de cada una de ellas).

- `void draw();`

El método draw debe iterar por la lista de cámaras. Para cada una, llamará a su método prepare y recorrerá la lista de entidades, llamando al método draw de cada una.

Programa principal

Vamos a reescribir la práctica anterior, de forma que utilicemos todas las nuevas clases.

Así que el resultado debería seguir siendo el siguiente: