CENTRO UNIVERSITARIO
DE TECNOLOGÍA Y ARTE DIGITAL

Programación 3D

Práctica 3: Carga de texturas

Con lo visto en clase, vamos a añadir al motor soporte para carga de texturas. Las texturas se representarán mediante la clase `Texture`. Aunque por el momento la textura y el shader a utilizar son las únicas propiedades visuales de la geometría a pintar, iremos añadiendo otras, con lo que vamos a crear una clase `Material` que contenga todas ellas. Ahora una malla va a contener, además de una serie de buffers, un material para cada buffer. Para realizar esta práctica, se proporcionan la siguientes texturas:

	
data/front.png	data/top.png

Clase Vertex

Se debe modificar para que contenta información de las coordenadas de textura.

Clase Texture

Una textura podrá ser cargada desde un fichero de disco (utilizaremos STB Image para realizar la carga del archivo de imagen). Necesita los siguientes métodos (además de los añadidos habituales que consideremos: variables miembro, constructores, destructor…):

```
static std::shared_ptr<Texture>     load(const char* filename);
```

```
uint32_t                        getId() const;
const glm::ivec2&               getSize() const;
void                            bind() const;
```

El método `load` debe cargar el archivo de imagen y generar una textura con los píxeles obtenidos. Devolverá el puntero a la textura, o `nullptr` si no se pudo cargar el archivo de imagen. Debe utilizar filtrado trilineal. Hay que tener en cuenta que los buffers que devuelve STB Image empiezan en la fila superior de la textura, mientras que OpenGL espera que la primera fila que se le pase sea la inferior.

El método `getId` devuelve el identificador de OpenGL de la textura.

Clase Material

Un material recoge las propiedades visuales que se aplican a la geometría a renderizar. Por el momento, serán un shader (si es `nullptr`, se utilizará `State::defaultShader`) y una textura (si es `nullptr`, se pintará la geometría de color blanco). Necesita los siguientes métodos:

```
Material(const std::shared_ptr<Texture>& tex = nullptr,
            const std::shared_ptr<Shader>& shader = nullptr);
const std::shared_ptr<Shader>& getShader() const;
std::shared_ptr<Shader>& getShader();
void setShader(const std::shared_ptr<Shader>& shader);
const std::shared_ptr<Texture>& getTexture() const;
void setTexture(const std::shared_ptr<Texture>& tex);
void prepare();
```

El método `getShader` debe devolver el shader utilizado por el material o, si éste es `nullptr`, devolverá `State::defaultShader`.

El método `prepare` se encargará de preparar OpenGL para pintar con las propiedades establecidas por este material. Este proceso consta de los siguientes pasos:

1. Activamos el shader devuelto por `getShader`.
2. Escribimos las variables uniformes necesarias. La matriz MVP se puede obtener a partir de las matrices de la clase State. Además, debemos crear variables uniformes nuevas para indicar si se debe utilizar textura (en caso contrario el objeto se pinta de blanco), y el sampler de la textura.
3. Si el material tiene textura, la enlazamos.

Clase Mesh

Vamos a hacer algunas modificaciones a esta clase. Anteriormente, almacenábamos una lista de buffers y sus respectivos shaders. Ahora no almacenamos los shaders directamente, sino un material por cada buffer. Modificaremos por tanto el método `addBuffer` para que quede de la forma siguiente:

```
void addBuffer(const std::shared_ptr<Buffer>& buffer, const Material& material);
```

Además, vamos a permitir obtener el material de un buffer concreto mediante el siguiente método (que implementaremos en versión constante y mutable):

```
const Material& getMaterial(size_t index) const;
Material& getMaterial(size_t index);
```

Por último, vamos a modificar el método `draw`. Algunas de las cosas que hacíamos aquí (activar el shader, escribir la matriz MVP) son realizadas ahora por el método `Material::prepare`, así que actualizaremos la implementación para que utilice este método.

Shaders

En el método `Shader::setupAttribs`, debemos configurar las coordenadas de textura del vértice. Además, se deben actualizar el vertex y el fragment shader para que se dibuje con textura (en el caso de que se deba hacer así).

Programa principal

Realizaremos un ejercicio para probar que la funcionalidad del motor ha sido correctamente implementada. Vamos a pintar en el origen de la escena un modelo que rotará continuamente sobre su eje vertical.

Dicho modelo utilizará una malla que crearemos proceduralmente. Será un cubo de un punto de tamaño, que utilizará dos buffers:

- Caras superior e inferior, textura **"data/top.png"**.
- Resto de caras, textura **"data/front.png"**.

Crearemos una cámara, en las coordenadas 0, 1, 3, con una rotación sobre X de -20 grados.

El resultado de implementar esta práctica quedará de la siguiente manera: