Utilización de APIs Tridimensionales

Práctica 2: Cámara, mundo y 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`. 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

Estructura vertex_t

Se debe modificar para que contenta información de las coordenadas de textura. Se debe añadir un registro “glm::vec2 vText”.

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):

Atributos:

glm::mat4 view;

glm::mat4 projection;

glm::vec3 up;

glm::vec3 lookAt;

Métodos a implementar

Camera(projection type,glm::vec3 position, glm::vec3 up, glm::vec3 lookAt);
glm::mat4 getProjection()
glm::mat4 getView()
void computeProjectionMatrix():
void computeViewMatrix();

Métodos Virtuales puros:

void step(float timestep);

Clase CameraKeyboard

Esta clase implementará una cámara que permitirá moverla usando eventos de teclado. El método step consultará el estado del teclado en la clase "InputManager" a través de la clase estática "System". Se debe poder mover en 3 dimensiones y girar el punto de mira respecto de la posición de la cámara (se deja libertad para poder implementarlo).

Cambios en clase System

Añadir un atributo estático "Camera* cam", y getter/setters para ella.

Clase Interfaz Texture

Esta interfaz servirá para establecer los métodos básicos de carga e inicialización de texturas. 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…):

load(std::string filename): Método virtual que , dado un nombre de fichero, lee los datos de la imagen, guarda sus atributos y la carga en GPU.

void getId(): Método virtual que devuelve un identificador de textura (puede ser el mismo identificador de OpenGL)

glm::ivec2 getSize():Método para acceder a las propiedades “alto ancho” usando un vector de 2 integers

void bind(): Método virtual para señalar que en los siguientes pasos de render se usará esta textura

Clase GLTexture

Clase que implementa una Textura usando opengl y STB Image. El método load debe cargar el archivo de imagen y generar una textura con los píxeles obtenidos cargados en OpenGL. Debe utilizar filtrado bilineal. 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, y bind llama a glBindTexture con los parámetros apropiados.

Cambios clase Material

Añadir un atributo de tipo puntero a “Texture”. Añadir getter/setters para acceder a él.

Cambios clase GLSLMaterial

Modificar el método “prepare()” para poder hacer uso de las texturas.

Cambios clase FactoryEngine

Añadir un método estático “Texture* getNewTexture()”. En caso de haber seleccionado el backend gráfico GL1 ó GL4, devolverá una implementación de GLTexture.

Cambios en clase Object

Vamos a añadir la opción de poder usar varias mallas dentro de nuestro motor. Para ello, se pide ambiar el atributo “Mesh3D* mesh” por un vector de mallas “std::vector<Mesh3D*> meshes”. Adaptar el método setMesh() para poder añadir las mallas de una en una en el nuevo vector.

Adaptar el método “getMesh()” para que ahora se pueda acceder al vector de mallas (renombrarlo a getMeshes).

Cambios en clases Render (GL1 y GL4)

Adaptar el método “drawObject” para poder dibujar cada malla del objeto (primero averiguar cuantas mallas tiene, y luego dibujarlas de una en una con los métodos que teníamos antes).

Código de los shaders

Actualizar el código de los shaders con los cambios vistos en clase para poder usar texturas.

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.

Para ello, crear una clase llamada “CubeTex”, que implemente una clase de tipo Object3D. Dicho modelo utilizará una malla que crearemos proceduralmente en el constructor (añadiremos los vértices a mano). Será un cubo de una unidad de tamaño (1x1x1), que utilizará dos materiales:

Su constructor creará dos mallas de 6 vértices con forma de cubo, y sus propios mat:

- Una malla tendrá las caras verticales del cubo, usando la textura “front.png”

- La segunda malla tendrá las tapas superior/inferior del cubo, usando la textura “front.png”.

Cada cara del cubo estará formada por dos triángulos, se debe intentar reaprovechar los “materiales” necesarios (texturas/shaders). Al estar centrado en el origen de coordenadas, se deben calcular las posiciones de cada uno de los vértices, sus coordenadas de textura, y su array

de índices para dibujar. Se aconseja empezar con una sola cara (dos triángulos), comprobar que se dibuje bien con la textura, e ir variando sus posiciones para conseguir el resto de caras.

Crearemos una cámara de tipo CameraKeyboard, en las coordenadas 0, 1, 3, que esté enfocada al origen de coordenadas (apuntando al cubo).

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