Utilización de APIs Tridimensionales

Práctica 3: 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 3D 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 vTextCoords”.

Clase Camera

Ésta será una subclase heredera 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;

projectionType_e type; //enumerador para elegir entre cámaras de tipo ortogonal o perspectiva

Métodos a implementar

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

La matriz proyección se calculará con un ángulo de apertura de 45º, plano near en 0.01f unidades y plano far en 1000.0f unidades.

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 usando teclas de teclado (sin ratón, se deja libertad para poder implementarlo). Se aconseja crear unos atributos "velocidad" y "dirección" para facilitar los movimientos.

Cambios en clase World

Añadir un atributo "std::list<Camera*> cameras", y getter/setters/removers para ella. Se pueden añadir tantas cámaras como se considere y borrarlas cuando se crea necesario, aunque en la mayoría de las prácticas usaremos sólo la primera.

Añadir un atributo "int activeCamera" que indicará cuál es la cámara activa en un momento dado. Añadir getter/setters para esta variable, no debe ser menor que cero ni mayor que el número de cámaras en la lista de cámaras.

Modificar el método "update" para que invoque el método "step" de la lista de cámaras y actualicen sus matrices model/view. Por defecto

Cambios (1) en GLSLMaterial y programa main()

En este momento podemos probar las cámaras, antes de implementar más clases. Para ello, modificar el método “prepare” de GLSLMaterial para que acceda a la matriz vista y proyección de la cámara que esté activa. Deberá crear una matriz MVP que copiará al shader de vértices.

El programa “main” añadirá una cámara de tipo “CameraKeyboard” al mundo. Ahora tendremos un triángulo rotando, pero podremos movernos con las teclas de teclado. En caso de no funcionar, se debe repasar lo siguiente:

- Método “step” de CameraKeyboard debe detectar las teclas (añadir couts para facilitar la depuración).
- Main añade una cámara al mundo.
- Por defecto, la cámara seleccionada es la “0”
- En método “System::mainloop” se debe llamar a método “update” del mundo. Este método a su vez debe recorrer las cámaras, llamar a su método “step”, y actualizar sus matrices view y projection
- En método “GLSLMaterial::prepare()” se debe acceder a las matrices modelo, vssta y proyección, multiplicarlas, y se suben al shader.

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 y atributos (además de los añadidos habituales que considere el alumno: variables miembro, constructores, destructor…):

Atributos:

uint32_t textID:

Identificador único de la textura. Se debe inicializar al llamar al constructor, usando un contador que devuelva un integer no repetido

glm::ivec2 size;

Tamaño en píxeles a lo largo de los ejes X/Y

bool cubemap = false;

Variable booleana para indicar si es una textura cúbica (de momento, false)

std::vector<unsigned char> texBytes;

Vector que almacena los bytes consecutivos “en bruto”de la textura cargada de fichero

std::string fileName;

Nombre del fichero cargado

bool bilinear = false;

Variable que indica si se debe aplicar un filtro bilinear

bool repeat = true;

Variable que indica si las coordenadas de textura mayores que 1.0 o menores que 0.0 deben ser cíclicas a lo largo de los ejes X/Y

Métodos:

Getter/setters para todos los atributos anteriores

Texture():

Constructor por defecto (sin parámetros). Asigna un identificador único a textId

Texture(std::string fileName):

Constructor que recibe un nombre de fichero, asignará un identificador único a textID y llamara al método de carga de datos de fichero

void load(std::string filename):

Método que, dado un nombre de fichero, lo abrirá usando la librería stbi y cargará los datos del fichero en el vector “texBytes”

virtual void update():

Método virtual cuya implementación dependerá del backend usado (Vulkan o GL4). Cargará los datos de “texBytes” en GPU y creará las estructuras necesarias para poder usarlo en una impelmentación de RenderProgram.

Clase GLTexture

Clase que implementa una Textura usando opengl . El método update debe cargar los datos de “texBytes” en GPU usando los métodos vistos de OpenGL. En función de las variables “repeat” y “bilinear” seteará las propiedades seleccionadas para tener filtros bilineares y repetición en coordenadas X/Y.

Añadir un atributo local glTextId que almacenará el identificador de textura devuelto por Opengl. Añadir get/setters para ese atributo.

Clases RenderProgram y GLSLProgram:

Añadir los siguientes métodos virtuales a la interfaz RenderProgram:

- **setColorTextEnable**/**Disable**(): Inicializa la variable uniform que indicará al shader de fragmentos si debe de utilizar texturas al colorear. Para facilitar la implementación, el nombre de la variable en el shader se conoce a priori (la misma usada en clase).
- **bindColorTextureSampler** (int binding, Texture* text): Selecciona el punto de enlace (binding) para la textura pasada por parámetros. Igualmente, se usará el mismo nombre de variable sampler2D usada en clase y GLSLProgram la conoce.

Implementar adecuadamente esos métodos en la clase GLSLProgram

Cambios clase Material

Añadir un atributo de tipo puntero a “Texture” llamado “colorText”. Añadir getter/setters para acceder a él. Modificar el método “prepare()” para poder hacer uso de las texturas en el RenderProgram seleccionado, llamando a los métodos “**setColorTextEnable” y “bindColorTextureSampler”** adecuadamente. Tener en cuenta que un objeto no debe tener obligatoriamente una textura (comprobar si la textura es nula, o implementar algún otro método para poder consultarlo). También debe hacer uso de las matrices cámara y proyección para subirlas al shader, puede acceder a ellas a partir de System::world.

También añadir una llamada a “setVertexAttrib” para inicializar el atributo de coordenadas de textura del vértice.

Hasta ahora no se han usado colores, los programas de fragmentos dibujaban el color blanco por defecto. Para facilitar el acceso a colores de malla, mover el atributo “glm::vec4 colorRGBA” de la clase Mesh a la clase Material, junto con sus getter y setters. Será la clase material la que decida el color de las mallas a partir de ahora, y los copiará al shader en el método “prepare”.

OJO: Se deben inicializar SIEMPRE todas las variables del shader. Se debe llamar adecuadamente a los métodos “setColorText/bindTexture...()” de RenderProgram

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 cambiar el atributo “Mesh3D* mesh” por un vector de mallas “std::vector<Mesh3D*> meshes” y adaptar los métodos de acceso a mallas con los siguientes métodos:

Mesh3D* getMesh(int pos):
    Devuelve la malla en la posición “pos”
std::vector<Mesh3D*>& getMeshes():
    Devuelve una referencia a la lista de mallas

void addMesh(Mesh3D* m);

Añade una malla a la lista de mallas.

Cambios en clases Render (GL1 y GL4)

Adaptar el método “drawObject” y “setupObject” para poder dibujar cada malla del objeto. Se deben crear tantos bufferObjects como mallas tenga, y luego dibujarlas de una en una con los métodos vistos en clase.

Añadir los métodos de activación de buffer de profundidad y limpieza de buffer adecuados:

glEnable(GL_DEPTH_TEST)

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

Código de los shaders

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

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 8 vértices con forma de cubo, y sus propios materiales:

- 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 “top.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 1, 1, 3, que esté enfocada al origen de coordenadas (apuntando al cubo).

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