

# Utilización de APIs Tridimensionales

## Práctica 2a: Rotating triangles (OpenGL 1.0)

En esta práctica, a partir de la plantilla realizada en clase, vamos a dibujar con OpenGL 1.0 una serie de triángulos en pantalla. Implementaremos clases para la gestión de los vértices, objetos, entidades y utilizaremos GLM para realizar las transformaciones necesarias para dibujar los objetos.

Las clases que se muestran a continuación son “sugerencias”, se deja al alumno libertad de implementación. En cualquier caso, se usará un esquema similar al mostrado para las próximas prácticas, por lo que será responsabilidad del alumno poder adaptar su código a lo pedido.

### API básica a implementar

Esta práctica está orientada a crear una API de programación 3d que tenga facilidades de adaptación a distintas arquitecturas y librerías gráficas. Para ello se crearán una serie de clases, interfaces y herencias que faciliten su uso una vez implementado. Se aconseja crear un subdirectorio “API3D” donde se almacenarán los archivos que implementen las clases. A continuación se describen las clases que se implementarán:

#### Estructura “vertex\_t”

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::vec4`).

Único registro: **posición** (tipo `vec4`)

#### Clase Mesh3D

Esta clase almacenará los datos necesarios de una malla 3D, que será dibujada en pantalla por nuestro motor de render. Por el momento contendrá los siguientes atributos/métodos:

##### Atributos:

- **meshID**: Identificador **único** de la malla, asignado en el constructor del objeto (PISTA: Se pueden usar variables de clase estática, actualizadas cada vez que se crea un nuevo objeto Mesh3D)
- **vec4 colorRGBA** : Color básico de esta malla, incluyendo opciones de transparencia
- **vector<struct vertex\_t>\* vVertList**: Lista de vértices que forman la malla. Se agrupan de 3 en 3, representando los triángulos

##### Métodos:

- **Mesh3D()**: Constructor por defecto de la clase. Inicializa los atributos a valores por defecto

- **getMeshID()**: Devuelve el identificador de esta malla
- **addVertex(struct vertex\_t)**: Método para añadir vértices a la malla
- **vector<struct vertex\_t>\* getVertList()**: Método para acceder a la lista de vértices almacenada

## Clase Interfaz “Entity”:

Los objetos que dispondremos en nuestra API pueden contener diversas propiedades. Pueden ser cámaras, objetos 3D con su mallas y texturas, o incluso luces con color e intensidades. Para agrupar las características comunes, definimos una clase básica llamada “Entity”. Esta clase representará un objeto “básico” con el mínimo de atributos comunes para objetos que interactúen con nuestro escenario:

Atributos:

- Posición
- Rotación
- Escalado
- Matriz Modelo

Métodos:

- **Getter/Setters** para los atributos
- **void computeModelMatrix()** : Método que actualiza la matriz Modelo de esta entidad, calculándola a partir de sus atributos posición/rotación/escalado
- **virtual void step(double deltaTime)** : Método **abstracto puro** que representará la “IA” de un objeto. Se ejecuta una vez por cada frame dibujado. Las clases que implementen Entity deben dar una implementación de este método. Al ejecutarlo, se actualizarán los atributos “posición”, “rotación” y “escalado” (junto con cualquier otro atributo que pueda tener esa nueva clase) siguiendo las reglas que haya dictado el programador (ej, moverse 0.1 unidades a la izquierda). Recibe por parámetros el tiempo que ha transcurrido entre distintos pasos de renderizado.

## Clase Interfaz “Object”:

Esta clase representará objetos que pueden ser dibujados en pantalla y heredarán de la interfaz de Entity. A su vez, es una interfaz que no tiene ninguna implementación nueva, sólo ofrece las cabeceras de los métodos que deben implementar los distintos tipos de objetos con mallas con los que trabajaremos. Se deben añadir los siguientes atributos y métodos virtuales

Atributos nuevos:

- **Mesh3D\* mesh**: Puntero a una malla que almacenará la geometría de este objeto
- **Tipo**: atributo para guardar el tipo de objeto que se implementará. Por el momento no se usa (sólo tenemos un tipo de objeto).

Métodos nuevos:

- **Get/Set** para el atributo “mesh”
- **loadDataFromFile(std::string file)**: Método abstracto puro que sirve para cargar los datos de este objeto desde fichero. Las clases que implementen esta interfaz deberán dar una implementación a esta función.

## Clase “Object3D”:

Esta interfaz hereda de Object. Sirve simplemente para tener una división entre objetos 3D y objetos 2D que implementaremos más adelante en el curso. Por ahora sólo tendrá una implementación vacía del método “**loadDataFromFile()**” (lo implementaremos en la práctica 4).

El método “**step**” sigue siendo virtual, se implementará en las clases herederas de Object3D.

## Clase Interfaz “Render”

Esta interfaz se usará para dar las funcionalidades básicas para implementar diferentes backends de renderizado. Ofrecerá las siguientes funciones :

### Atributos:

- **Width,Height:** Almacenan la información de “ancho” y “alto” del framebuffer/pantalla que se usará

### Métodos virtuales:

- **init():** Método que inicializa las librerías que se usarán para implementar un sistema de render
- **setupObject(Object\* obj):** Método que prepara las estructuras internas para dibujar un objeto.
- **removeObject(Object\* obj):** Método análogo al anterior para liberar las estructuras internas de la librería para este objeto
- **drawObjects(std::vector<Object\*>\* objs):** Método que dibuja en pantalla una lista de objetos pasada por parámetros. Estos objetos deben haberse inicializado antes usando el método “setupObject”
- **Get/Set** para las variables de ancho y alto.
- **isClosed():** Método para averiguar si se ha cerrado la “sesión” del render (en nuestro caso, si se ha cerrado la ventana)

## Clase GL1Render

Clase que implementa la interfaz “Render” para dibujado de mallas 3D usando OpenGL 1.0 y glfw. Tendrá los siguientes atributos, e implementará las funciones virtuales de “Render”:

### Atributos:

- **GLFWwindow\* window:** Puntero a una ventana de glfw

### Métodos:

- **Constructor** que reciba por parámetros ancho/alto de la ventana
- **init():** Inicializa glfw y estados de OpenGL

- **setup/removeObject()**: Por el momento, vacíos
- **drawObjects**(std::vector<Object\*>\* objs): Método que dibujará una lista de objetos usando OpenGL 1.0 (por ahora, son solo triángulos). Antes de dibujarlos, debe actualizar su matriz Modelo

## Clase virtual “InputManager”

Usaremos esta clase para definir un sistema de acceso a los eventos del usuario (principalmente, eventos de teclado y ratón). Tendrá los siguientes atributos y métodos:

### Atributos:

- **std::map<int, bool> keyState**: Mapa/lista de caracteres para poder consultar si una tecla está apretada o no.

### Métodos

- **init()**: Método virtual que inicializará el sistema de “input” para poder consultar el estado del teclado a partir del array keybEvent
- **bool isPressed**(char key): Retorna el estado de una tecla pasada por parámetros. Ej: una llamada como “isPressed(‘GLFW\_KEY\_A’)” devolverá true si está apretada la tecla “a”.

## Clase GLFWInputManager

Implementación de la clase “inputManager” usando los métodos de la librería glfw. El método “init” inicializará una función de tipo “keyCallback” en la librería que actualizará apropiadamente el array “keybEvent”. Por el momento no es necesario implementar estos métodos (no se usará hasta más adelante), pero se deja libertad al alumno para adelantar esta parte de nuestro motor gráfico.

## Clase estática FactoryEngine

El objetivo de esta clase es dar una “fábrica” de implementaciones para algunas de las clases “especiales” que pueden tener distintos backends. Por ahora se encargará de dar las implementaciones adecuadas de los backends para las clases de “Render” e “InputManager”. Por el momento sólo tenemos un backend para estas

clases, que serían GL1Render y GLFWInputManager. La clase tendrá los siguientes atributos y métodos:

Atributos estáticos:

- **selecteGraphicsBackend, selectedInputBackend**: Indica el tipo de backend seleccionado. Se deja libertad al alumno para implementar de forma adecuada estas variables. Por ahora, únicamente GL1.0

Métodos estáticos:

- **Get/Set** para los tipos backends
- **Render\* getNewRender()**: En función del tipo de backend, devolverá una nueva instancia del render seleccionado (por ahora, sólo GL1Render)
- **InputManager\* getNewInputManager()**: En función del tipo de backend, devolverá una nueva instancia del InputManager seleccionado (por ahora GLFWInputManager)
- **isClosed()**: Devolverá "true" en caso de haberse cerrado la ventana de glfw

## Clase World

Esta clase estará encargada de contener un escenario con objetos, cámaras, luces, etc... También deberá actualizar los movimientos de esos objetos llamando en el orden correcto a las funciones de "step" de los objetos que contenga. Por el momento, contendrá los siguientes atributos/fuciones:

Atributos privados:

- **std::list<Object\*> objects**; Lista de objetos que están en el escenario.

Métodos públicos:

- **World()** : Constructor de la clase, por ahora vacío
- **void addObject(Object\* obj)**: Método para añadir objetos a la lista de objetos de este mundo

- **void removeObject(Object\* obj):** Método para borrar un objeto. Se busca en la lista a través de su referencia/puntero y se borra
- **size\_t getNumObjects():** Método que devuelve el tamaño de la lista de objetos
- **Object\* getObject(size\_t index):** Método que devuelve el objeto que se encuentra en la posición "index" de la lista
- **std::list<Object\*> &getObjects():** Método que devuelve por referencia la lista completa de objetos
- **void update(float deltaTime):** Método que actualiza el estado de este mundo. Llama al método "step" de cada uno de sus objetos pasándoles la variable "deltaTime", que representa el tiempo transcurrido entre actualizaciones.

## Clase estática System

Nuestro motor gráfico usará esta clase para gobernar la ejecución de los métodos necesarios para dibujar objetos y escenarios 3D. **Será el núcleo del sistema**, uniendo todas las clases anteriores. Irá evolucionando en las siguientes entregas, por ahora tendrá los siguientes atributos/métodos estáticos:

### Atributos estáticos:

- **Render\* render:** Puntero a una clase "Render" que implemente esa interfaz
- **InputManager\* inputManager:** Puntero a una clase "InputManager" que la implemente
- **bool end:** Variable para indicar si se debe acabar la ejecución
- **World\* world:** Variable que apunta a un objeto de tipo "mundo" que representa el escenario activo en este momento.

### Métodos estáticos

- **initSystem():** Inicializa el sistema. Inicializará los atributos de esta clase. Seleccionará en "FactoryEngine" los backends apropiados para las clases que hemos implementado. Para inicializar las variables render e inputManager se usarán los métodos de "FactoryEngine". Se debe llamar a su vez a los métodos "init" de cada una de esas clases.
- **addObject(Object\* obj):** Añadirá un nuevo objeto a la lista "objects"
- **exit():** Método para acabar la ejecución del sistema (pone a "true" la variable end)

- **mainLoop()**: Método de bucle principal del sistema. Deberá realizar las siguientes operaciones:
  - LLamar a la función “setupObject” de render con cada objeto del mundo que esté activo
  - while(!end)
    - calcular “deltaTime”
    - actualizar el mundo y sus objetos (llamar al método update del mundo activo)
    - dibujar objetos (llamar a la función drawObjects de render, usando el vector de objetos actual)
- Getter/setters para las variables estáticas de clase

## Programa Principal (main)

Todas las clases anteriores conforman nuestra API de programación (el código que usarán otras personas para implementar sus aplicaciones). Ahora prepararemos un programa que, apoyándose en esas clases, implementará un escenario con 1 triángulo rotando. Para ello, se deberá implementar lo siguiente:

### Creación del objeto triángulo

Crearemos una clase “TrianguloRot” que heredará de Object3D, implementando su método “step”. En su constructor creará un objeto “Mesh3D” con los datos de vértices e índices de un triángulo (las mismas coordenadas usadas en el ejemplo de clase).

El método “step()” de la clase “TrianguloRot” actualizará su atributo de rotaciones, haciéndolo girar sobre el eje “Y” a una velocidad de 90° por segundo. Además, comprobará si se ha apretado la tecla “e” cada vez que se le invoque. Si es así, invocará la función “exit” de la clase System.

### Método “main”

El programa principal realizará las siguientes tareas:

- Inicializar el backend de la clase “FactoryEngine” para poder usar OpenGL 1.0 y el input manager por defecto.
- Inicializar la clase System
- Crear un objeto de tipo “TrianguloRot” en la posición <0,0,0>, y añadirlo al sistema
- Crear un objeto “World” y añadir el triángulo creado anteriormente
- Añadir el objeto World a System
- Ejecutar el mainLoop del sistema, hasta que el usuario presione la tecla “e”

