Generación de un sistema de *render*

PID_00249853

Jorge Arnal Montoya

Tiempo mínimo de dedicación recomendado: 2 horas

Universitat Oberta de Catalunya

Índice

Introducción

En este material, vamos a implementar nuestra arquitectura de motor de videojuegos. Para ello, definiremos todas las clases que contienen nuestro motor.

Aprenderemos cómo funciona internamente una aplicación en tiempo real y las tres funcionalidades principales de un videojuego.

A continuación, explicaremos cómo leer un fichero FBX y poder pintarlo en pantalla.

Entenderemos conceptos de bajo nivel, como VertexBuffer e IndexBuffer, ConstantBuffer o *shaders*.

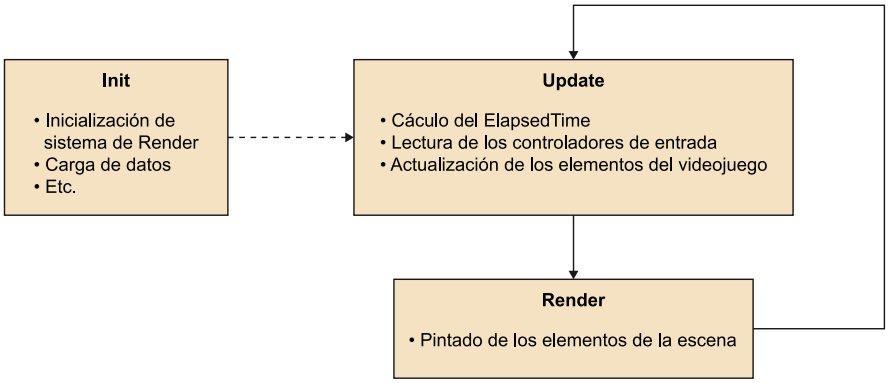
Todo el código que implementaremos se basará en la API gráfica DirectX 11.

1. Diagrama de una aplicación gráfica en tiempo real

En cualquier aplicación en tiempo real como un videojuego, siempre encontramos tres funcionalidades principales:

1) **Init**. La función Init se encargará de inicializar todo lo referente al motor de juego y la carga de elementos que necesitaremos para arrancar el mismo.
2) **Update**. La función Update actualizará todos los elementos de nuestro videojuego. En esta función, calcularemos el ElapsedTime, que contiene el tiempo transcurrido desde el *frame* anterior al actual.
3) **Render**. La función Render será la encargada de pintar todos los elementos de nuestro videojuego.

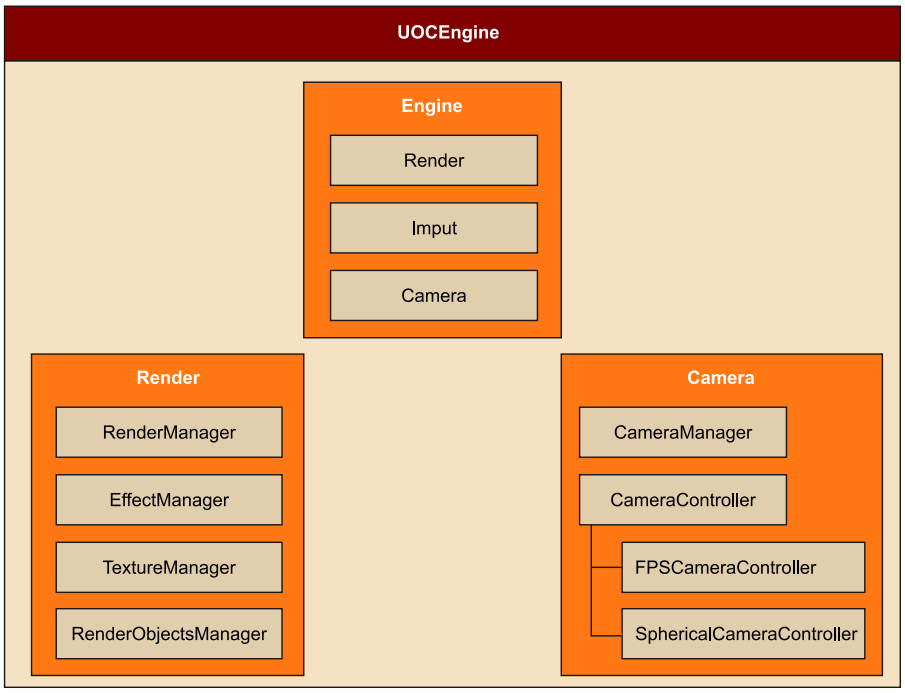
Podemos ver este sistema en el siguiente diagrama.



Aunque este diagrama está planteado como una aplicación de un solo hilo, se puede adaptar para usar arquitecturas multiproceso, en las que estos métodos se pueden dividir en diferentes procesos como múltiples tareas y ejecutarlas en paralelo.

2. Arquitectura del motor

Como hemos visto anteriormente, el diseño de la arquitectura del motor es clave para realizar cualquier pieza de software y, en especial, un motor de videojuego. A continuación, mostramos el diagrama de la arquitectura del motor que hemos construido para esta asignatura.



2.1. Engine

El módulo Engine será el responsable de controlar todos los componentes de nuestro motor. Este módulo tendrá como clase principal CUOCEngine, una clase que, partiendo de un patrón de tipo *singleton*[1] , nos dará acceso al resto de módulos de nuestro motor.

(1)El patrón de diseño *singleton* nos asegura que en todo nuestro código habrá una única instancia de un objeto. En nuestro caso, solo tendremos un motor.

Esta clase contendrá los métodos Get a las siguientes clases:

- CRenderManager
- CEffectManager
- CInputManager
- CCameraManager
- CDebugRender
- CFBXManager
- CRenderableObjectManager

- CTextureManager

2.2. RenderManager

El módulo RenderManager se implementará en la clase CRenderManager y encapsulará las diferentes funcionalidades de DirectX 11. En esta clase encontramos tres métodos destacados.

2.2.1. Init

Este método será el responsable de crear el *device* y el *device context* de DirectX 11, las dos principales estructuras utilizadas para cualquier tarea que haga referencia al sistema de *render*. Para ello, utilizamos el método D3D11CreateDeviceAndSwapChain.

A continuación, creamos el *depth stencil buffer* utilizando el método CreateDepthStencilView y lo establecemos con el método OMSetRenderTargets.

Por último, establecemos el *viewport* utilizando el método RSSetViewports.

2.2.2. BeginRenderDX

Este método se llamará a cada inicio de *frame* y limpiará tanto el *render target* como el *depth stencil buffer* utilizando los métodos ClearRenderTargetView y ClearDepthStencilView respectivamente.

2.2.3. EndRenderDX

Al igual que tenemos el método BeginRenderDX, que se llama al principio del *frame*, deberemos llamar al EndRenderDX para terminar el *frame*, que hará que el *frame* que hemos pintado sea visible por el usuario. Para ello, utilizaremos el método Present de DirectX.

2.3. CameraManager

La clase CCameraManager encapsulará la cámara y los controladores de cámara de nuestro motor. Esta clase derivará de la clase CXMLParser, que nos permite leer un fichero XML donde estableceremos las cámaras que hay en nuestro juego de forma externa.

Utilizamos un mapa de la librería STL, donde la clave del mapa será el nombre del controlador de la cámara y el valor del mapa será de tipo CCameraController. De esta manera, podremos acceder a un controlador de cámara pidiéndolo al CameraManager según el nombre del controlador.

2.3.1. Camera

La clase CCamera encapsulará la información para crear una cámara de tipo perspectiva en DirectX 11. Una cámara en 3D está definida por las matrices de View y Projection.

Para crear la matriz de View, necesitaremos el ojo de la cámara, el punto hacia donde mira y el vector *up* de la cámara. Utilizaremos el método XMMatrix-LookAtRH.

Para crear la matriz de Projection, necesitaremos el campo de visión (FOV), las distancias del plano cercano y lejano, y el *aspect ratio* de la cámara. Utilizaremos el método XMMatrixPerspectiveFovRH de DirectX para crear la matriz.

2.3.2. *Frustum*

El *frustum* de cámara nos define la figura geométrica que representa la forma de la cámara; un *frustum* es una pirámide recortada. Llamaremos al método Update en cada *frame* pasándole la matriz View*Projection para calcular los seis planos que forman el *frustum* de nuestra cámara.

2.3.3. CameraController

La clase CCameraController es una clase abstracta que nos va a permitir derivar de ella dónde los diferentes controladores deberán sobrescribir el método virtual puro SetCamera.

2.3.4. FPSCamera

La clase CFPSCamera es un controlador basado en los videojuegos de tipo *shooters* en primera persona. Este controlador establece la cámara con posición en los ojos del jugador y el LookAt se calcula sumándole la dirección calculada según los ángulos *yaw* y *pitch* a la posición de la cámara.

2.3.5. SphericalCamera

Si la clase CFPSCamera es un controlador en el que el punto de vista parte desde lo que ve un objeto, esta cámara nos permite tener siempre en nuestro punto de vista un objeto. En este caso, el punto donde miraremos será la posición del objeto, y la posición de la cámara se calculará restando la dirección, nuevamente calculada a través de los ángulos *yaw* y *pitch*, multiplicada por la distancia a la que queramos estar del objeto.

2.4. InputManager

La clase CInputManager será la responsable de leer los dispositivos de entrada; en este caso, tendremos dos *accessors*, para el teclado y el ratón.

2.4.1. MouseInput

La clase CMouseInput encapsulará el código para leer el dispositivo ratón.

En el constructor de la clase, se inicializan las estructuras de DirectX que permiten leer el dispositivo. Para ello, se utilizan los métodos DirectInput8Create y CreateDevice, que construyen el *device*. A continuación, se utiliza el método SetDataFormat, que establece el tipo de formato que queremos utilizar para leer el estado del ratón. Por último, utilizaremos la función SetCooperativeLevel para definir cómo se comporta el ratón con la ventana.

La clase CMouseInput contiene también nuestro método Update, que actualiza el estado del ratón utilizando la función GetDeviceState de DirectX.

2.4.2. CKeyboardInput

Igual que teníamos una clase para encapsular el dispositivo ratón, la clase CKeyboardInput encapsulará el dispositivo teclado.

En el constructor de la clase, se utilizan métodos con los mismos nombres que en el ratón: DirectInput8Create, CreateDevice, SetDataFormat y SetCooperativeLevel.

Y al igual que en el ratón, tenemos el método Update, con su función GetDeviceState de DirectX.

2.5. FBXManager

Con FBXManager empezamos a implementar nuestra primera clase que va a pintar algo en pantalla. Esta clase encapsula el código de la biblioteca FBXSDK[2] que necesitamos para poder leer un fichero de tipo FBX para convertirlo en una malla estática en nuestro motor.

(2)La biblioteca FBXSDK está creada por Autodesk y nos permitirá leer ficheros generados en formato FBX.

En esta clase encontramos dos funcionalidades principales:

- **Load.** Este método se encarga de cargar un fichero de tipo FBX y generar las mallas estáticas que hay dentro del fichero.
- **ImportNode.** Nos encontramos con un método al que se llama de forma recursiva y que va creando las mallas estáticas que va encontrando desde el nodo principal a todos los nodos hijos. Por cada malla estática que crea, construye un RenderableObject, que nos permitirá pintar esa malla está-

tica situándolo en la posición, la rotación y la escala que le toque según el fichero.

2.5.1. FBXStaticMesh

La clase CFBXStaticMesh contiene toda la información necesaria para poder pintar una malla estática importada de un fichero en formato FBX.

Como métodos importantes destacaremos los siguientes:

- **Generate.** Genera toda la información necesaria para pintar la malla estática a partir del nodo y la malla del fichero FBX. Para ello, recorre el vector de materiales del nodo para extraer las texturas de difuso del objeto. A continuación, extraeremos la información de vértices, normales y coordenadas de textura del nodo para generar un RenderableVertex por cada uno de los materiales del nodo. Una vez que tengamos toda la información de vértices de nuestro nodo, calcularemos la esfera y la caja contenedora[3] del nodo. Por último, recogeremos el *effect* que utilizaremos para poder pintar dicho nodo.

(3)Una esfera contenedora es la esfera mínima que contiene todos los vértices de la geometría. Una caja contenedora será la caja mínima que contiene todos los vértices alineada a los ejes.

- **Render.** Este método nos permitirá pintar el objeto. Para ello, recorreremos el vector de RenderableVertexs, activaremos las texturas de difuso para cada material asociado y llamaremos a su método DrawIndexed.

2.5.2. VertexBuffer-IndexBuffer

Desde las primeras versiones de API gráficas, hemos encontrado el concepto de VertexBuffer e IndexBuffer. Como su propio nombre indica, estas estructuras son un *array* de elementos que contienen o bien la información de vértices o bien la información de índices de un objeto.

Un vértice es una estructura no fija que puede contener información como su posición geométrica, su color, su normal o sus coordenadas de textura…

En cambio, un índice es un valor que indica la posición de un vértice dentro del VertexBuffer.

Para pintar una malla 3D, podemos hacerlo solo con un VertexBuffer o con la unión de un VertexBuffer y un IndexBuffer.

2.5.3. Tipos de primitivas

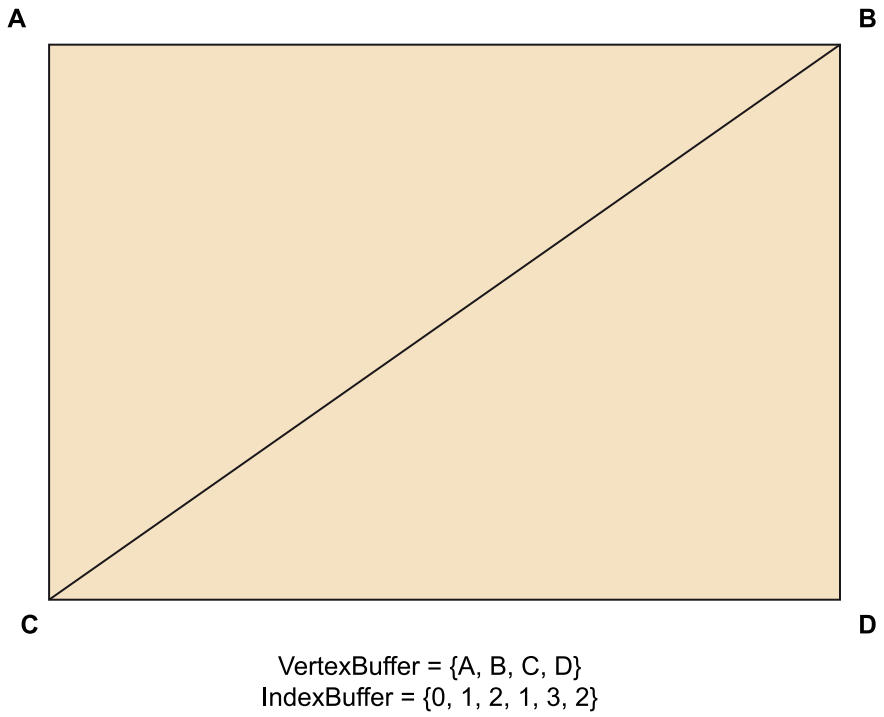
Los tipos de primitivas típicos que encontraremos en las API gráficas del mercado son los siguientes:

- Líneas. Nos permiten pintar un conjunto de líneas.

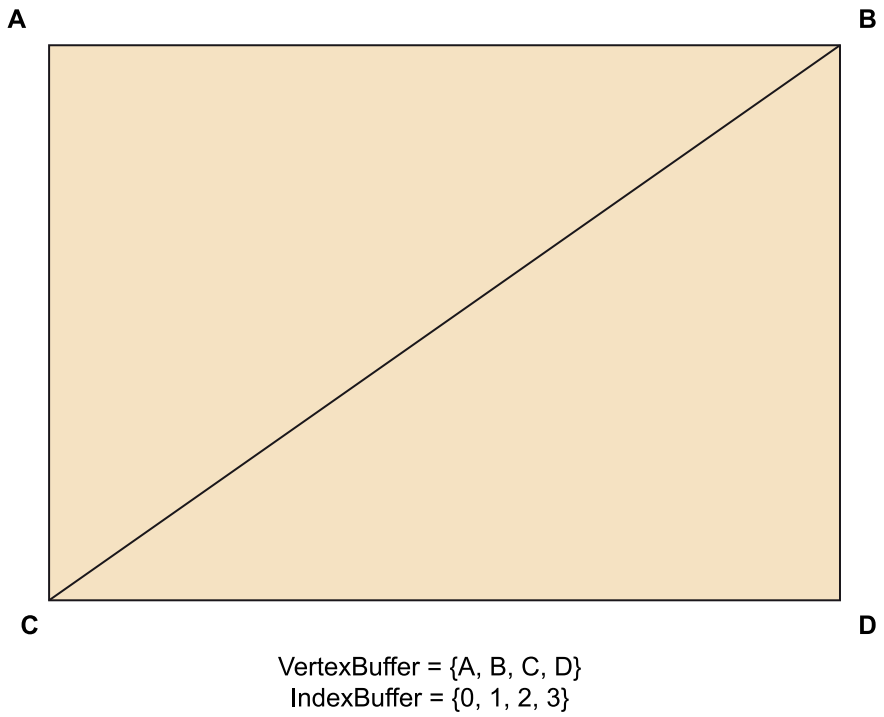
- Triángulos. Nos permiten pintar un conjunto de triángulos; son la primitiva más utilizada.

También explicaremos los dos tipos de topologías más utilizadas y extendidas en el *render*.

- Lista de triángulos. Es la topología más sencilla y la más utilizada; define que cada tres vértices forman un triángulo. Por ejemplo, para definir un *quad*,podemos crear cuatro vértices en el VertexBuffer y necesitaremos un IndexBuffer con seis índices; por cada tres índices, formaremos un triángulo.



- Triángulos estripificados. Esta topología consiste en que el primer triángulo se pinta a través de los tres primeros vértices, mientras que los siguientes triángulos utilizan los dos últimos vértices más el nuevo vértice para pintar un nuevo triángulo. En el caso anterior, para pintar el *quad* solo necesitaremos cuatro vértices para pintarlo con triángulos estripificados y cuatro índices.



VertexTypes

Previamente hemos hablado de los VertexBuffers, que contenían un *array* de tipo vértice. Esta estructura puede estar definida por la información más básica, como su posición geométrica como contener información más relevante.

Para crear estos tipos de vértices, hemos generado el fichero VertexTypes.h, en el que utilizando la potencia de preprocesado de C++, podemos crear todas las estructuras de vértice con una llamada a la macro CREATE_UOCD3D11_VERTEX. Esta macro recibe como parámetros el nombre de la estructura, si la estructura tiene posición geométrica de tres o cuatro componentes, si tiene normal, si tiene pesos e índices (lo que se utiliza para animación esqueletal), si tiene color, si tiene una pareja de coordenadas de texturas o dos. Por tanto, solo con una llamada a la macro nos crea todo el código de una estructura completa.

Hay que destacar, dentro de cada una de estas estructuras, la función CreateInputLayout, que generará un *layout* de la estructura que utilizaremos para decirle al Vertex Shader la estructura de nuestro vértice.

2.6. RenderableVertexs

Tal como hemos visto anteriormente, para pintar una malla de primitivas, podemos utilizar bien un VertexBuffer o bien un VertexBuffer con un IndexBuffer. La clase CRenderableVertexs es una clase de la que derivaremos que nos va a permitir pintar una malla de primitivas solo con un VertexBuffer o con

un VertexBuffer y un IndexBuffer. Como vemos en su declaración, esta clase contiene dos métodos: Draw y DrawIndexed. El primero utilizará un VertexBuffer para pintarse, y el segundo, VertexBuffer e IndexBuffer.

2.6.1. CTemplatedRenderableVertexs

A continuación, encontramos la clase templatizada CTemplatedRenderableVertexs, que deriva de la clase CRenderableVertexs. Esta clase nos permite utilizar cualquier tipo de vértice mediante el *template* de la clase.

En esta clase, podemos comprobar cómo el constructor construye el VertexBuffer según el tipo de vértice utilizando la función de DirectX CreateBuffer.

A continuación, encontramos el método Draw, que utiliza las siguientes funciones de DirectX:

- IASetVertexBuffers. Esta función establece el VertexBuffer creado previamente.

- IASetPrimitiveTopology. Esta función establece el tipo de primitiva que vamos a utilizar para pintar esta malla de triángulos.

- IASetInputLayout. Esta función le dice al Vertex Shader el formato del vértice.

- VSSetShader. Esta función establece el *shader* que utilizaremos como Vertex Shader.

- PSSetShader. Esta función establece el *shader* que utilizaremos como Pixel Shader.

- Draw. Por último, llamaremos a esta función para hacer el pintado de la malla.

2.6.2. CTemplatedRenderableIndexedVertexs

En este caso, encontramos una clase que, igual que la anterior, deriva de CRenderableVertexs. Sin embargo, esta clase nos incorpora, además del VertexBuffer, un IndexBuffer para poder pintar la malla de primitivas.

En su constructor encontramos que, además de crear el *buffer* para los vértices, hace un segundo *buffer* para los índices.

Esta clase sobrescribe el método DrawIndexed, y se diferencia respecto al método Draw en que incorpora la llamada IASetIndexBuffer, donde se establece el IndexBuffer que utilizaremos, y en vez de usar la llamada Draw, como en la clase anterior, emplearemos ahora la llamada a la función DrawIndexed.

2.7. TextureManager

Como hemos podido apreciar previamente, en el desarrollo de videojuegos es muy común utilizar clases llamadas *managers*. En este caso, nos encontramos una *manager* de texturas que nos permite gestionar todas las texturas de nuestro juego.

Una *manager* como en este caso suele consistir en un mapa de STL, donde la clave del mapa es un *string* y el valor que contiene es el tipo del objeto que guarda, en este caso una textura.

Como en este caso, en la *manager* tendremos un método Load o Get, donde recibiremos el nombre del recurso que queramos recibir, y la *manager* comprueba si ya existe dentro del mapa buscándolo por su nombre como clave. En caso de no existir, lo carga y lo introduce en el mapa; en caso de que existiese ese recurso, directamente devuelve un puntero al recurso. De esta forma, tenemos el recurso una sola vez en memoria; a este objeto se le suele llamar *objeto core*, y tenemos diferentes punteros a este objeto. A estos objetos apuntadores se les suele llamar *instancias*.

2.7.1. Texture

La clase CTexture es la clase que utilizamos para encapsular todas las estructuras necesarias para crear una textura en DirectX 11.

En esta clase destacaremos el método Load, que carga un fichero de textura utilizando la función de DirectX D3DX11CreateShaderResourceViewFromFile, y creará una *sampler* de textura con el método CreateSamplerState. Gracias a la *sampler*, podremos definir los tipos de filtro que se utilizarán para hacer el *mipmap*, el *minmap* y el *magmap*, y definiremos las *addresses*[4] cuando las UV de las texturas estén fuera del rango 0..1.

(4)La *address* de textura nos permitirá definir cómo se debe comportar al pintarse una primitiva con coordenadas de textura fuera del rango 0..1. Los ejemplos típicos son que la textura se repita o que la textura repita el píxel de los bordes.

2.8. CRenderableObjectManager

La clase CRenderableObjectManager[5] será la responsable de pintar todos los objetos de escena de nuestro juego. Para ello, contendrá un vector de STL, que, a su vez, incluirá todos los objetos CRenderableObject.

(5)La clase CRenderableObjectManager la podríamos llamar CLayerManager, ya que contiene un conjunto de elementos renderizables. Podemos organizar los elementos por capas de nuestro motor; esto nos puede ir bien porque hay elementos que se necesitan pintar en un orden concreto o de una manera determinada. Motores como Unity dividen sus objetos renderizables por capas.

Esta clase tiene tres métodos principales:

- AddRenderableObject, que añade un objeto renderizable al conjunto de objetos.
- Update, que actualiza todos los objetos renderizables según el ElapsedTime del juego.
- Render, que pinta todos los objetos renderizables de nuestro juego.

2.8.1. CRenderableObject

La clase CRenderableObject es una clase que nos va a permitir derivar de ella para poder pintar objetos en nuestro videojuego.

Dentro de esta clase, destacaremos que contiene información de posición, rotación y escala para poder crear la matriz de mundo del objeto. Para hacer esta matriz de mundo, utilizamos el método GetTransform. Debemos crear una matriz de traslación utilizando la función XMMatrixTranslation. A continuación, creamos la matriz de rotación usando las funciones XMMatrixRotationX, XMMatrixRotationY y XMMatrixRotationZ. Por último, creamos la función de escalado utilizando la función XMMatrixScaling. Ahora solo nos falta calcular la matriz de mundo, que se calcula multiplicando en este orden la matriz de escala por la matriz de rotación por la matriz de traslación.

Información de transformación del objeto

Como hemos podido ver, la información de transformación del objeto está incrustada dentro de la clase CRenderableObject. Esta información podríamos extraerla en una nueva clase CTransform, al igual que hacen otros motores como Unity.

En esta clase destacamos también los métodos Update y Render, que están implementados vacíos y nos permitirán actualizar o pintar el objeto en las clases que deriven de CRenderableObject.

2.8.2. MeshInstance

La clase MeshInstance es una clase que deriva de CRenderableObject y nos va a permitir pintar objetos estáticos de tipo FBX.

Esta clase contiene una instancia de un objeto de tipo CFBXStaticMesh y sobrescribe el método Render, donde establece la matriz de mundo del objeto utilizando la propiedad m_World de los parámetros de objeto de la clase CEffectManager. A continuación, fija los parámetros de objeto en los Constant-

Buffers de los *shaders* mediante el método SetObjectConstantBuffer de la clase CEffectManager, y llama al método Render de la clase CFBXStaticMesh para pintar el objeto.

2.8.3. CDebugRender

La clase CDebugRender nos va a permitir pintar diferentes primitivas básicas que nos ayudarán a depurar el juego en el ámbito de *render*.

Esta clase permite, mediante el método DrawAxis, pintar unos ejes para poder ver la posición, la orientación y la escala de un objeto. También se puede pintar una *grid* a través del método DrawGrid, un cubo con el método DrawCube y una esfera con DrawSphere. Todos estos métodos pintan los objetos mediante líneas.

2.9. CEffectManager

La clase CEffectManager será la responsable de gestionar todos los CEffects de nuestro motor, además de los ConstantBuffers de los *shaders*.

En esta clase destacaremos los métodos siguientes:

- SetSceneConstantBuffer. Este método establecerá los valores del ConstantBuffer de escena.
- SetObjectConstantBuffer. Este método establecerá los valores del ConstantBuffer de objeto.
- SetAnimatedModelConstantBuffer. Este método establecerá los valores del ConstantBuffer de los modelos animados.
- CreateConstantBuffer. Este método nos permitirá crear un ConstantBuffer utilizando el método de DirectX CreateBuffer.
- SetConstantBuffer. Este método nos va a permitir establecer los parámetros de constantes que tenemos en memoria RAM en los ConstantBuffers de VRAM. Para ello, utilizaremos las funciones UpdateSubresource, VSSetConstantBuffers y PSSetConstantBuffers.

2.9.1. Effect

La clase CEffect encapsulará las estructuras de DirectX 11 que nos permiten crear *shaders*; cada efecto contendrá un Vertex Shader y un Pixel Shader.

Al principio de la generación de tarjetas de vídeo 3D, se utilizaba la llamada *fixed pipeline*, donde definíamos cómo se debía transformar un vértice o iluminar un píxel. Poco después, las tarjetas gráficas se pudieron programar. Estos fragmentos de código que creamos y que se ejecutan en la GPU se denominan *shaders*.

En este curso diferenciaremos entre los denominados Vertex Shader y Pixel Shader. Un Vertex Shader es un fragmento de código que transforma el vértice y se ejecuta por cada uno de los vértices que le llegan a la tarjeta de vídeo al hacer el *render* de las primitivas. El Pixel Shader es un fragmento de código que se ejecuta para cada uno de los píxeles que se pintan en pantalla cuando estamos pintando una malla.

Shader

Un *shader* es un fragmento de código que se ejecuta en la GPU de la tarjeta de vídeo. Los lenguajes más utilizados para programarlo son HLSL y GLSL. Dependiendo del nivel al que se ejecuta, podemos tener *shaders* de tipo vértice, píxel...

Para la compilación de un *shader*, utilizamos la función de DirectX D3DX11CompileFromFile. Una vez que tenemos el fichero compilado, podemos crear un Vertex Shader usando la función CreateVertexShader o un Pixel Shader con la función CreatePixelShader.

2.9.2. EffectParameters

Como hemos comentado cuando hablábamos de la clase CEffectManager, vamos a tener ConstantBuffers en los *shaders*.

ConstantBuffers

Los ConstantBuffers son estructuras de datos que contienen los *shaders* para todo el *render*.

Es decir, en nuestro motor el código está implementado en C++, y todos los datos, como la información de las matrices de View, Projection o World, están en memoria RAM. Esta información se la debemos hacer llegar a los *shaders* para poder hacer los cálculos de transformación. Esto lo haremos a través de los ConstantBuffers. Estas estructuras serán réplicas de la información que tenemos en memoria RAM, pero en VRAM.

En nuestro motor, hemos definido tres ConstantBuffers llamados EffectParameters:

- CSceneConstantBufferParameters contiene la información de cámara, como su matriz de View, Projection o sus vectores *right* y *up*.
- CObjectConstantBufferParameters contiene la información por objeto, como son su matriz de World, un color y un parámetro de DebugRenderScale.
- CAnimatedModelConstantBufferParameters contiene el conjunto de matrices que definen los huesos de nuestro modelo animado.

3. Retos

En este material vamos a realizar los siguientes retos:

- Lectura de un modelo 3D desde un fichero utilizando la librería FBXSDK.
- Crear la estructura de datos de vértices e índices de la malla.
- Pintar las primitivas básicas utilizando los métodos de pintado del SDK.

Resumen

Como hemos podido apreciar, la creación de un motor de videojuegos, por sencillo que sea, es un producto de software altamente complicado y que requiere una gran cantidad de código.

En este material, hemos creado nuestra primera aproximación a la implementación de un motor de videojuegos, con el que somos capaces de pintar mallas estáticas además de crear diferentes controladores de cámara y poder movernos por la escena.

Hemos aprendido también conceptos de más bajo nivel, como los de VertexBuffer o IndexBuffer, qué es un ConstantBuffer o qué es un *shader,* tanto en relación con los vértices como con los píxeles.

Además de aprender y entender estos conceptos, hemos visto una implementación de toda esta arquitectura.

Bibliografía

Direct3D 11 Reference. https://msdn.microsoft.com/en-us/library/windows/desktop/ff476079(v=vs.85).aspx

FBX SDK. http://docs.autodesk.com/FBX/2014/ENU/FBX-SDK-Documentation/

Introduction to Buffers in Direct3D 11. https://msdn.microsoft.com/en-us/library/windows/desktop/ff476898(v=vs.85).aspx

Implementing Lighting Models with HLSL. http://www.gamasutra.com/view/feature/2866/implementing_lighting_models_with_.php