Integración de bibliotecas de terceros

PID_00249855

Jorge Arnal Montoya

Tiempo mínimo de dedicación recomendado: 1 hora

Universitat Oberta de Catalunya

Índice

Introducción 5

1. Arquitectura del motor 7

2. Física 8
- 2.1. NVIDIA Physx 9
- 2.2. PhysicsManager 9
- 2.3. MeshInstance 10
- 2.4. Player 10

3. *Scripting* 12
- 3.1. ScriptManager 12
- 3.2. Registro de funciones con Lua 12

4. Retos 14

Resumen 15

Bibliografía 17

Introducción

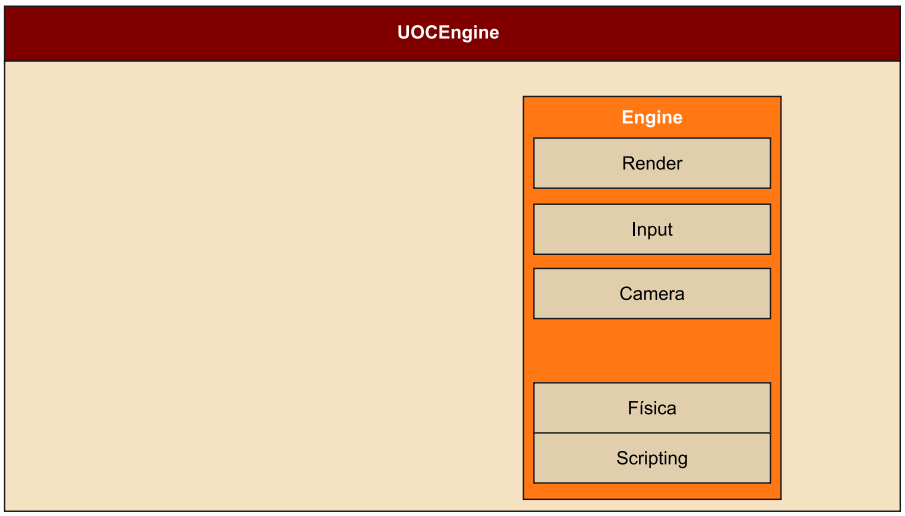
En este material, vamos a modificar nuestra arquitectura de motor de videojuegos para poder integrar dos nuevas bibliotecas de terceros.

La primera de ellas será una biblioteca de física como NVIDIA Physx, que nos va a permitir integrar modelos físicos dentro de nuestro motor, como cuerpos rígidos y controladores de personajes con los que poder interactuar en un videojuego.

A continuación, integraremos una biblioteca como Lua, que nos permitirá integrar una biblioteca de *scripting* para poder externalizar parte del código. Hay que destacar que los videojuegos separan la parte de tecnología de la parte de *gameplay*. El lenguaje típico para programar la parte de tecnología sería C++, mientras que el *gameplay* se suele implementar en un lenguaje de *scripting*, que nos permite modificar el código y comprobar su funcionamiento sin tener que volver a compilar y generar el ejecutable como debemos hacer en C++.

1. Arquitectura del motor

En este material, vamos a modificar la arquitectura del motor para integrar una biblioteca de física con la que podremos introducir colisiones. También integraremos una biblioteca de *scripting*, que nos permitirá externalizar el código de *gameplay* del código del motor de videojuego.



2. Física

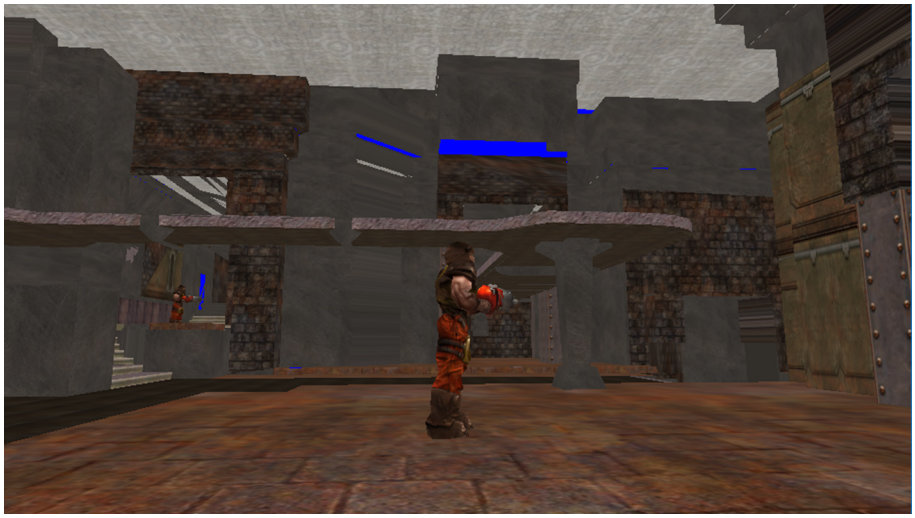
Desde los primeros videojuegos de la década de los ochenta del siglo pasado, la física y las colisiones de los elementos con los escenarios han sido importantes para crear la sensación de realismo en el jugador o la falsa creencia de entornos limitados a este.

Para la implementación de estas colisiones o física, se han utilizado funcionalidades típicas de física o modelos que representan elementos con los que se pueden calcular las colisiones de forma rápida.

En 2D podemos utilizar círculos, cuadrados o polígonos para representar los elementos de nuestro videojuego, con lo que se simula el escenario donde el jugador juega.

En 3D las primitivas básicas más usadas son cubos, esferas, cápsulas o, para modelos más complejos, mallas convexas o de triángulos.

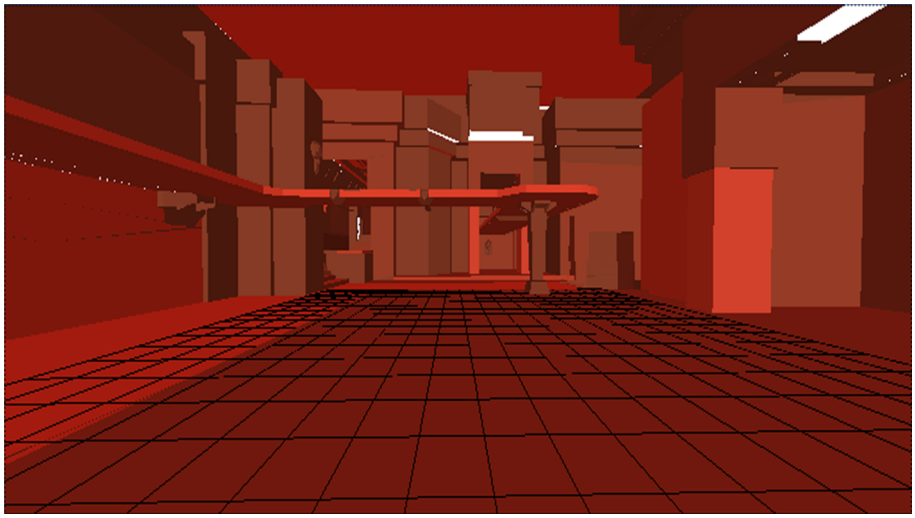
Gracias a estos elementos, podemos limitar el entorno de nuestro videojuego y conseguir, a través del sistema de renderizado, la falsa creencia en el jugador de que está jugando en un entorno real.



Ejemplo

Hoy en día, gracias a la física, se están creando diseños de niveles mucho más ricos. Videojuegos como Half Life 2 o Zelda Breath of Wild utilizaban elementos físicos para dar un valor añadido a los puzles del juego.

(1)Motores comerciales como Unity o Unreal utilizan la biblioteca NVIDIA Physx.



2.1. NVIDIA Physx

Como hemos podido comprobar en los materiales anteriores, muchas de las funcionalidades que se crean en un motor de videojuegos dependen de bibliotecas de terceros.

La creación de una biblioteca de física desde cero es altamente costosa y requiere, para construirla, muchas horas de trabajo. Por ello la mayoría de los videojuegos actuales que encontramos en el mercado[1] utilizan bibliotecas como NVIDIA Physx o Havok de AMD.

En nuestro motor, vamos a introducir la biblioteca NVIDIA PhysX, que nos permitirá crear mallas de triángulos con las que limitar nuestro escenario, así como un controlador de personaje o *character controller* con el que poder desplazarnos por este según las limitaciones fijadas.

2.2. PhysicsManager

Como hemos ido haciendo hasta ahora, vamos a crear una clase CPhysicsManager que encapsulará toda la funcionalidad de la biblioteca de física. En esta clase podemos destacar los siguientes métodos:

- Init. Este método será el responsable de inicializar la biblioteca de física. Para ello arrancará el motor utilizando las funciones de PhysX PxCreateFoundation y PxCreatePhysics. A continuación, si estamos en modo *debug* y permitimos el acceso al Visual Debugger que integra PhysX, crearemos la conexión con el *debugger* utilizando la función PxVisualDebuggerExt::createConnection. Una vez arrancado el motor de física, creamos la clase PxCooking, que nos permitirá «cocinar» las mallas de triángulos. Podremos pasarle mallas triangulares a Physx para que este genere la malla física del modelo. Después se inicializa la escena con el

método createScene y, por último, el gestor de controladores de física, con el método PxCreateControllerManager.

- CreateTriangleMesh. Este método recibe la información de vértices e índices de nuestra malla de triángulos y nos devuelve una malla de triángulos de física con la que podremos crear, a continuación, objetos rígidos con los que colisionar. Para construir esta malla de triángulos, utiliza los métodos cookTriangleMesh y createTriangleMesh.
- CreateRigidStatic. Este método recibe una malla de triángulos de física y una posición, y construye un objeto rígido estático. Para crear este objeto estático, utilizamos el método createRigidStatic. Una vez construido el objeto rígido, lo introducimos en la escena con el método addActor.
- CreateCharacterController. Gracias a este método, podremos crear un controlador de personaje. En los videojuegos en 3D, los controladores de personaje están representados por cápsulas. Para crear dichas cápsulas, establecemos la altura de la cápsula y su radio, la altura máxima de escalón (StepOffset) que es capaz de subir el personaje y el grado máximo de pendiente (SlopeLimit) que puede ascender. Para crear dicho controlador de personaje, utilizamos el método createController.
- TestRaycast. La última funcionalidad de Physx que hemos encapsulado es la de comprobar la colisión de un rayo. Un test típico en videojuegos es el de lanzar rayos contra el escenario y ver si colisionan; por ejemplo, para controlar el disparo de una bala o para comprobar si desde un punto A hay una línea directa hasta un punto B sin colisiones. Para hacer esta comprobación, utilizamos el método raycastSingle de la biblioteca de Physx.

2.3. MeshInstance

Una vez encapsulada la biblioteca de física en nuestro motor, debemos modificar nuestro código para que los elementos que queramos tengan cualidades físicas. A fin de realizar esta tarea, hemos de introducir cambios en la clase CMehsInstance para que nos cree un cuerpo rígido de los modelos 3D.

Para conseguir esta funcionalidad, hemos añadido en la clase CFBXStaticMesh un vector de PxTriangleMesh que se rellena con todas las submallas de nuestro modelo 3D al cargar el fichero FBX.

Una vez que tenemos las mallas de triángulos de física en nuestros modelos estáticos, cuando creamos una instancia de nuestras mallas estáticas, creamos también su cuerpo rígido en nuestra escena de física. Para ello simplemente utilizamos el método CreatePhysx de la clase CMeshInstance, donde se crea un cuerpo físico por cada una de las submallas que definen el modelo 3D.

2.4. Player

Como hemos comentado previamente, los personajes de juego utilizarán, por norma general, un controlador de tipo CharacterController, donde la figura geométrica que los representará será una cápsula. En nuestro código, hemos

creado una clase llamada CPlayer, que representa el personaje de juego y que encapsula el control del jugador. Para controlar la física del personaje, utiliza el método CreateCharacterController; para crear nuestro CharacterController en el constructor de la clase y en la funcionalidad de Update del personaje, se calcula el vector de movimiento del personaje según la gravedad y el *input* de teclado. Una vez calculado dicho vector, se utiliza el método Move para el movimiento del jugador. Ejecutada esta orden, es PhysX quien se encarga de mover el controlador y de hacer que colisione con los elementos físicos de la escena.

3. *Scripting*

Para terminar este material, vamos a introducir una biblioteca de *scripting* en nuestro motor. La elegida es Lua, una biblioteca creada en la Universidad Pontificia Católica de Río de Janeiro, en Brasil.

Videojuegos con *scripting*

Videojuegos como Maniac Mansion, Loom y aventuras gráficas de LucasArts ya utilizaban un lenguaje de *scripting* como era SCUMM (Script Creation Utility for Maniac Mansion).

Lua es uno de los lenguajes de *scripting* más utilizados en el mundo de los videojuegos. Debido a sus características de robustez, rapidez y al hecho de que es portable a múltiples plataformas, es un lenguaje muy extendido en la industria. Además de estas características, podemos destacar también que es gratuito incluso para productos comerciales.

El único inconveniente que podemos achacarle a Lua es su incapacidad para trabajar con objetos debido a que es una biblioteca desarrollada en C. De todas formas, este punto en contra puede ser subsanado con otras bibliotecas que añaden la funcionalidad de código orientado a objeto.

El funcionamiento de una biblioteca de *scripting* siempre será el mismo: registraremos funciones en C que, una vez registradas, pueden ser utilizadas desde el código Lua.

3.1. ScriptManager

Nuevamente, crearemos una clase que va a encapsular nuestra biblioteca de *scripting*. En este caso, se tratará de CScriptManager, donde remarcaremos los siguientes métodos:

- Initialize. Inicializa la biblioteca de Lua utilizando la función luaL_newstate.
- RunCode. Ejecuta el código escrito en Lua que le pasamos por parámetro de tipo *string*. Para ejecutar el *script*, emplea la función luaL_dostring.
- RunFile. Ejecuta el código escrito en un fichero Lua mediante la función luaL_dofile.

3.2. Registro de funciones con Lua

Destacaremos, por último, la forma en que debemos registrar una función en C para que pueda ser utilizada correctamente desde un *script* de Lua. Los parámetros de la función y lo que devuelve esta función siempre tienen que ser los mismos.

Una función llamada TestCFunction debe ser declarada de la siguiente forma.

```
1. int TestCFunction(lua_State *State);
```

Para poder registrarse en Lua, una función de C ha de devolver, como vemos, un entero, que le dirá a Lua el número de valores que le devuelve. Esta función será un valor de 0 o 1. Por otra parte, el parámetro de entrada será siempre lua_State; de lua_State podremos extraer los parámetros de la función a la que hemos llamado desde Lua.

Por ejemplo, si quisiéramos crear una función en Lua que nos devolviera la vida del *player*, utilizaríamos un código como el siguiente.

```
1.  //Este sería el cuerpo en C de la función si la creásemos para ser utilizada dentro de C
2.  //int GetPlayerLife();
3.  int GetPlayerLife(lua_State *State)
4.  {
5.  int l_PlayerLife=CPlayer::GetPlayer()->GetPlayerLife(); //Devolvemos la vida del player
6.  lua_pushinteger(State, l_PlayerLife);
7.  return 1;
8.  }
```

Con la función lua_pushinteger, devolvemos un valor a Lua, en este caso la vida del *player*.

Si quisiéramos crear una función para asignar un valor a la vida del *player*, utilizaríamos el siguiente código.

```
1.  //Este sería el cuerpo en C de la función si la creásemos para ser utilizada dentro de C
2.  //void SetPlayerLife(int Life);
3.  int SetPlayerLife(lua_State *State)
4.  {
5.  int l_PlayerLife=(int)lua_tointeger(State, 1);
6.  CPlayer::GetPlayer()->GetPlayerLife(); //Establecemos la vida del player
7.  return 0;
8.  }
```

4. Retos

En este material vamos a realizar los siguientes retos:

- Integrar una biblioteca de física dentro del motro de juego.
- Implementar un controlador de personaje como jugador.
- Implementar elementos físicos con los que interactuar.
- Integrar una biblioteca *scripting* dentro del motor de juego.

Resumen

En este material, hemos aprendido que en la industria del videojuego se utilizan bibliotecas de física y *scripting*.

Hemos empezado introduciendo conceptos básicos de física y cómo integrar una biblioteca como NVIDIA Physx dentro de nuestro motor de videojuegos. También hemos integrado cuerpos rígidos y controladores de personaje en nuestro motor.

A continuación, hemos conocido una biblioteca como Lua, que nos permite crear código mediante *script* para poder implementar el *gameplay* de nuestro juego de manera más rápida, ya que no necesita compilación y enlace del código.

Bibliografía

Documentación de la biblioteca Physx. http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Index.html

Collision detection for dummies. http://www.wildbunny.co.uk/blog/2011/04/20/collision-detection-for-dummies/

Advanced Collision Detection Techniques. http://www.gamasutra.com/view/feature/131598/advanced_collision_detection_.php

Character controller video sample. https://www.youtube.com/watch?v=LJp2NnKj1X4

Documentación de biblioteca Lua. http://www.lua.org