ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Trabajo fin de Carrera

Ingeniería Informática

Integración de CLIPS en Unreal Engine 4

Realizado por
Francisco Javier Carpio Franco

Dirigido por
Francisco Jesús Martín Mateos

Departamento
Ciencias de la Computación e Inteligencia Artficial

Sevilla, septiembre de 2015

Resumen

Este Proyecto de Fin de Carrera es, en esencia, una integración del conocido entorno para el desarrollo de sistemas expertos CLIPS en el motor de videojuegos Unreal Engine 4.

Esta integración tiene como objetivo ayudar al proceso de desarrollo de un videojuego, permitiendo que el programador del sistema experto ponga de manera rápida y sencilla un conjunto de herramientas fáciles de usar a los miembros de perfiles no técnicos del equipo, como un diseñador de niveles.

A lo largo de esta documentación repasaremos todo el proceso de integración, desde la compilación de las librerías de CLIPS hasta la exposición de las funciones al sistema de scripting visual empleado por Unreal Engine 4, detallando los problemas que se presentan y explicando el proceso seguido para afrontarlos. Además de esto, también se incluirá un último capítulo que servirá como ejemplo de uso de la herramienta.

Agradecimientos

Cuando uno ha hecho un recorrido tan largo, desde el inicio de una carrera como esta Ingeniería Informática hasta este punto final con el Proyecto de Fin de Carrera, le es difícil hablar de agradecimientos. Principalmente porque la lista de nombres de gente que te ha ayudado, enseñado y acompañado es enorme. Pero, lamentablemente, no tenemos espacio para dedicar unas palabras a todos y cada uno de ellos; así que quiero lanzar desde aquí un enorme y sincero agradecimiento a todos aquellos que habéis hecho que haya podido llegar hasta aquí.

No obstante, sería injusto no hacer una mención especial a la persona sin la cual no sé si hubiera podido lograr llevar a cabo esta hazaña, pero os puedo asegurar que no lo hubiera logrado manteniendo la cordura. Muchas gracias, Aly.

Índice general

Índice de cuadros

Índice de figuras

Índice de código

CAPÍTULO 1

Introducción

1.1– Motivación

Si me lo permiten, me gustaría comenzar con una afirmación que aunque ampliamente conocida - y repetida - es imprescindible a la hora de entender la motivación de este proyecto: la industria del videojuego es extremadamente joven. Es importante dejar claro la edad temprana de este sector porque, sin ella, el lector que sea ajeno a las vicisitudes de esta industria seguramente se sienta confundido al ver como los desarrollos se realizan bajo una planificación extremadamente optimista e ineficiente. Y eso en el caso en el que exista una planificación. Estamos ante una industria donde el tiempo que se concede a un proyecto es mínimo para que el videojuego en cuestión sea siempre el más actual y en el que, por tanto, las "chapuzas" para solucionar los problemas de la forma más directa (que no eficiente) son la norma. Sirva como muestra de esta afirmación como, aún hoy día, existen videojuegos en los que se opta por emplear un único modelo 3D que haga de escenario para un nivel, en lugar de emplear un editor modular que permita reutilizar el trabajo ya realizado, o arreglar errores sin tener que rehacer todo de nuevo.

Pongamos por ello un ejemplo: imaginemos un videojuego de puzzles. Imaginemos que el jugador toma el control de un personaje que se encuentra atrapado en una habitación completamente cerrada. En dicha habitación encontramos tres puertas diferentes a un lado y otros tantos interruptores en el extremo opuesto. Cada puerta puede abrirse con una combinación exacta de los interruptores que se encuentran al otro extremo de la sala. No es demasiado complicado imaginar una implementación para tal problema: tan sólo tendríamos que realizar una comprobación cada vez que un interruptor cambia de estado para ver si alguna de las puertas puede abrirse. Ahora supongamos que en lugar de tres puertas y tres interruptores tenemos un millar de cada. La cosa cambia, ¿verdad?

Este problema, es extremadamente común en los videojuegos. Cada cinemática, puerta, habilidad o cualquier otro elemento que se quiera introducir, depende directamente de ciertas variables que definen el estado del juego. Por supuesto, esta situación se puede abordar de infinitas maneras; sin embargo, suele hacerse con la más evidente y directa: poner puntos de control en los que se compruebe el estado de dichas variables y actuar en consecuencia. Algo que no es complicado en sí mismo de realizar, pero sí bastante engorroso; especialmente a efectos de controlar todas las posibles situaciones que puedan darse en un momento concreto del juego. Influenciadas, en gran medida, por lo que el jugador desee y decida hacer en cada ocasión.

Una solución mucho más eficiente – y cómoda – para el problema anterior sería hacer uso de un sistema experto que entienda el funcionamiento del juego en cuestión. Éste debe ser capaz de interpretar en cada momento la situación actual del jugador y las posibilidades que debe brindarle. En pocas palabras, cumpliría el rol de un director de juego.

Por esto mismo, la propuesta para este Proyecto de Fin de Carrera consiste en la integración de una herramienta ampliamente conocida para el desarrollo de sistemas expertos como es CLIPS, en uno de los motores más usados en la industria del videojuego actual: Unreal Engine 4. De esta manera, en las siguientes páginas se detallará una posible integración del entorno de CLIPS en Unreal Engine 4 para que pueda funcionar en un segundo plano, aplicando reglas para definir el estado actual del juego y pudiendo comunicarse con este para recibir información de los cambios que se realizan y enviar las conclusiones a las que se llegan.

1.2– Introducción a Unreal Engine 4

Llegados a esta parte, quizás el lector se pregunte por qué hablamos en este trabajo de integrar CLIPS en Unreal Engine 4 y no de algo como el uso en sí de sistemas expertos aplicados a videojuegos. Suena un caso demasiado concreto, ¿verdad? Bien, el motivo es sencillo: las exigencias de la industria actual obligan al uso de motores de juego comerciales en el desarrollo de videojuegos; en el caso de que no se desee invertir una gran cantidad de recursos y tiempo en el desarrollo de un motor propio. Estos motores se podrían entender como una *suite* de herramientas que cubren todos o casi todos los ámbitos necesarios para la producción de un videojuego y, por tanto, su nivel de complejidad es muy alto. Es por esto mismo que el mero hecho de integrar un nuevo sistema de manera eficiente en uno de estos motores ya supone un desafío en sí mismo, sin embargo ofrece un tremendo potencial de uso como herramienta adicional a las ya existentes en el propio motor.

Por otra parte está la elección del motor en sí. La oferta actual de motores de juego es muy amplia, tanto de motores comerciales como de

motores libres. ¿Por qué elegir Unreal Engine 4 entonces? Hay varios motivos. El primero y más reseñable es que, actualmente, es el motor *de facto* en las grandes producciones de videojuegos y pocas son las que se libran de emplear esta o alguna de sus versiones anteriores, lo que aumenta la valía de este trabajo en cuanto a sus aplicaciones potenciales.

Por otro lado, también se da otro factor muy importante que es el libre acceso al código fuente del motor. Este permite un amplio margen de maniobra a la hora de integrar nuevo módulos a los ya existentes tal y como veremos más adelante.

Por último, estaría el factor del precio. Pese a ser un motor comercial Unreal Engine 4 es gratuito para usos no comerciales del mismo y no impone ningún tipo de limitaciones, cosa que sí ocurre con otras alternativas en sus versiones gratuitas o libres. Todo esto y algunos otros detalles de menor importancia, que veremos en apartados posteriores del trabajo, son los que consideramos para afirmar que Unreal Engine 4 es el candidato perfecto a la hora de realizar este proyecto.

Pero bien, ¿qué es exactamente Unreal Engine 4? Como ya se ha mencionado, es un motor de videojuegos. O lo que es lo mismo, un variopinto conjunto de herramientas dispuestas para que todas las facetas del desarrollo de un videojuego se puedan realizar cómodamente. No obstante, aunque cómodas y relativamente sencillas de utilizar una vez conocido su manejo, la extensión de las mismas no es baladí, por lo que hacer introducir al lector a todas ellas escapa del marco de este proyecto. Sin embargo, sí se comentará brevemente la cadena de trabajo básica y que emplearemos en este proyecto. Por otra parte, también se presupondrá los conocimientos necesarios para manejar Visual Studio 2013 y su compilador de C++, ya que es el empleado por Unreal Engine 4 para trabajar y, una vez más, se escaparía del interés de este proyecto – y probablemente del lector – explicar todas sus entrañas.

Aclarado esto, podemos pasar a hablar de los elementos de Unreal Engine 4 con los que es importante familiarizarse para el trabajo que vamos a realizar.

Por un lado tenemos los *Actors* (actores), que son todos aquellos elementos que puedan disponerse en el espacio 3D con una posición, rotación y escala. Junto a ellos, tendríamos los propios *Levels* (niveles), también conocidos como *Scenes* (escenas), que serían los susodichos espacios 3D en los que disponemos actores.

Adicionalmente, Unreal Engine 4 dispone de un sistema de herencia de objetos y *scripting* visual denominado Blueprint. Con él podemos crear especializaciones de los actores y escenas antes mencionados, de forma que cubran todas nuestras necesidades; como por ejemplo un actor que sea un personaje capaz de recibir órdenes de entrada del usuario o un nivel que contenga cierta disposición concreta de actores para formar un escenario de un juego.

Para finalizar, también contamos con un sistema de componentes denominados *ActorComponents* (componentes del actor) que nos permite incluir de una manera muy rápida una lógica a un actor ya creado.

Estas dos últimas características serán de las que más uso hagamos, ya que nuestro objetivo es crear una integración de CLIPS que sea útil para un usuario no técnico (como un diseñador de niveles) y los componentes nos brindan esa posibilidad de una manera limpia y directa.

1.3– Instalación de los sistemas

Aquí no nos detendremos demasiado ya que el proceso es realmente sencillo e intuitivo. Todo lo que necesitaremos se encuentra disponible desde las páginas oficiales de Unreal Engine [1] y de CLIPS [2]. Por su parte, el proceso de instalación es trivial, consistiendo la instalación de Unreal Engine 4 en un asistente que nos permite elegir la versión que deseemos y la de CLIPS en tan sólo descomprimir los archivos descargados. Las versiones que emplearemos serán: la versión 4.8.3 para Unreal Engine y la versión 6.3 de CLIPS. Debido a los procesos de actualización de ambas herramientas, cualquier versión diferente a las mencionadas (incluidas versiones futuras, ya que la retrocompatibilidad de las APIs no está asegurada) podría no funcionar correctamente con los pasos que detallamos en este documento. Por lo anterior, se recomienda encarecidamente el uso de estas versiones si se desea realizar un seguimiento exhaustivo de lo aquí descrito.

1.4– Estructura del proyecto en Unreal Engine 4

En esta sección nos detendremos algo más que en la anterior en lo que respecta a Unreal Engine 4, aunque no demasiado. La estructura de un proyecto de Unreal Engine 4 puede adoptar bastantes formas dependiendo del contexto de trabajo (plataforma objetivo, plataforma desde la que se desarrolla, tipo de proyecto que se está realizando, etc.). Sin embargo, explicaremos los elementos de la estructura que vamos a seguir a continuación para que no quepa lugar a la pérdida. Así pues, en el directorio del proyecto podremos encontrar las siguientes carpetas:

- **Binaries**, como su propio nombre indica, en esta carpeta será donde el compilador almacene todos los archivos binarios generados. Por norma general no hay que gestionar el contenido de dicha carpeta, ya que lo generará y lo eliminará el compilador de manera automática.

- **Config**, su nombre también resulta bastante explícito acerca de su función. En esta carpeta se almacenan los archivos de configuración

[1] http://www.unrealengine.com/
[2] http://clipsrules.sourceforge.net/

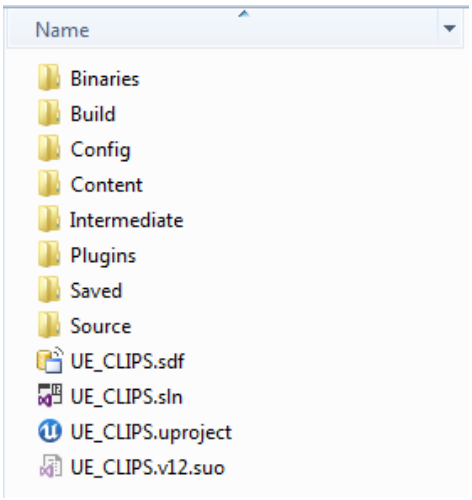

Figura 1.1: Estructura habitual de un proyecto de Unreal Engine 4

que Unreal Engine empleará. Es importante notar que estos archivos funcionan en cascada, por lo es importante conocer la existencia de los archivos equivalentes que hay en la raíz de la instalación del motor y en los documentos del usuario, ya que no pocas veces la solución a un problema que nos podamos encontrar reside en una regla que se está aplicando sin que nos percatemos de ellos.

- **Content**, aunque pueda no parecerlo, esta carpeta es la más ambigua en cuanto a la relación entre su nombre y su contenido. En ella se situarán todos los elementos que compongan el juego (modelos, texturas, niveles, actores, etc.) Sin embargo, no se almacenarán en un formato estándar correspondiente al tipo de archivo. Tampoco se almacenará la lógica o código del juego en ella, a pesar de que en el llamado "Content browser" del motor, sí aparezcan estos últimos elementos. Más allá de esto último, el uso de la carpeta es bastante directo ya que todo su contenido se gestionará desde la propia interfaz del motor salvo contadas excepciones.

- **Plugins**, esta es una carpeta que no nos aparecerá si creamos un proyecto nuevo de Unreal Engine 4; deberemos crearla manualmente. En esta carpeta es donde almacenaremos el código de integración y las librerías de CLIPS.

- **Source**, esta carpeta será la que contenga todo el código propio del proyecto. A su vez, el código viene dividido por defecto en dos carpetas: Public y Private, siendo su principal uso el de organizar el código que queramos dejar expuesto al compilar un *plugin* de

Unreal Engine y el que no. Más allá de esto último, la organización que se haga del código queda a elección del usuario. En nuestro caso emplearemos esta organización que trae por defecto.

Además de estas carpetas, también encontraremos otras como **Intermediate**, **Saved** o **Build** que serán generadas automáticamente por el motor para almacenar información relevante a la instancia que se está ejecutando del proyecto. En otras palabras, estas carpetas se generan automáticamente para un uso temporal y son prescindibles en última instancia, por lo tanto no les prestaremos mayor atención.

1.5– Estructura de la documentación

Para finalizar con esta introducción, vamos a hablar brevemente sobre cómo se organizará el resto de la documentación.

Los dos siguientes capítulos, corresponderán a la parte de integración de CLIPS en Unreal Engine. En estas secciones se comentarán los objetivos que se buscan conseguir, el proceso que hemos seguido para cumplirlos y un comentario del diseño que se ha empleado para la implementación. Además, también se comentarán los problemas que han surgido durante la realización y las soluciones encontradas y elegidas para cada uno de ellos.

Tras ello, pasaremos a una explicación de cómo poner en práctica todo el sistema que hemos implementado por medio de un proyecto de ejemplo. En este capítulo procederemos de manera similar a los anteriores, explicando los objetivos que queremos alcanzar y los procedimientos empleados para lograrlos; pero con una diferencia bastante notable: tomaremos un enfoque más práctico para mostrar el uso básico de todas las herramientas que se han creado y que sirva así de referencia para el lector que desee hacer uso de las mismas.

Sirviendo de cierre para esta documentación, encontraremos el obligado y necesario capítulo dedicado a las pruebas realizadas, los resultados obtenidos y las conclusiones. Como es costumbre, se repasarán los potenciales usos del proyecto, las posibles alternativas y los frentes que han quedado abiertos o que puedan ser motivo de futuras investigaciones.

Por último, antes de pasar manos a la obra, queda por hacer un último aviso. El código que se muestra en estos capítulos no debe ser tomado nada más que por lo que es: una guía para el lector con la que pueda seguir el hilo de los razonamientos y entender cómo se realiza la implementación.

Por ello se recomienda, especialmente a aquellos interesados en los detalles de la implementación, que consulten el código adjunto al proyecto. También es importante notar que si bien el código está escrito en C++, hace un uso extenso de la API de Unreal Engine 4; esto significa que el uso de macros definidas en el propio motor será práctica habitual, y tampoco será extraño encontrar operadores sobrecargados sirviendo para un

uso poco común o intuitivo. Es por esto que, una vez más, se recomienda al lector interesado en entender a fondo estos aspectos de la implementación que consulte la documentación del motor donde se desglosa el uso y significado de todos estos aspectos de la API.

CAPÍTULO 2

Integración básica de CLIPS

A lo largo de esta sección comentaremos cómo realizar una primera integración básica de CLIPS en Unreal Engine 4. Nuestro objetivo, de momento, será conseguir leer e interpretar código de CLIPS desde nuestro proyecto en Unreal Engine 4 y, además, poder comprobar que está funcionando correctamente por medio de una interfaz de salida.

Con esto en mente ya tenemos una meta clara. Ahora el problema es cómo llegar hasta ella, claro. No son pocas las cosas de las que queremos ocuparnos, así que mejor centrarnos en lo más básico, ¿cómo conseguimos hacer que Unreal Engine 4 tenga acceso a CLIPS?

2.1– Integración de las librerías en Unreal Engine

La integración de librerías de terceros en un programa es de lo más común. De hecho, a lo largo de todos los años de la carrera de Ingeniería Informática, la afirmación que uno puede llegar a oír en mayor cantidad de ocasiones es eso de "reutilizar el código"; y las librerías son el camino directo a esto mismo. Sin embargo tenemos que tener en cuenta que en la práctica, especialmente cuando trabajamos con una base de código tan extensa como es la de Unreal Engine 4, integrar una librería externa no se resume a emplear una directiva `#include` y esperar que todo funcione a las mil maravillas.

En el caso de las librerías dinámicas, la integración sí que resulta más sencilla *a priori*; tan sólo tendríamos que emplear un manejador para acceder a las funciones de la librería. Nada especialmente complicado, su funcionamiento es exactamente igual que en cualquier otro ámbito en el que se empleen librerías dinámicas. No obstante, la vida de un ingeniero nunca puede ser tan sencilla y el uso de librerías dinámicas externas provoca una buena cantidad de fallos de exportación del proyecto final en Unreal Engine 4. Además, también dificulta enormemente el trabajo

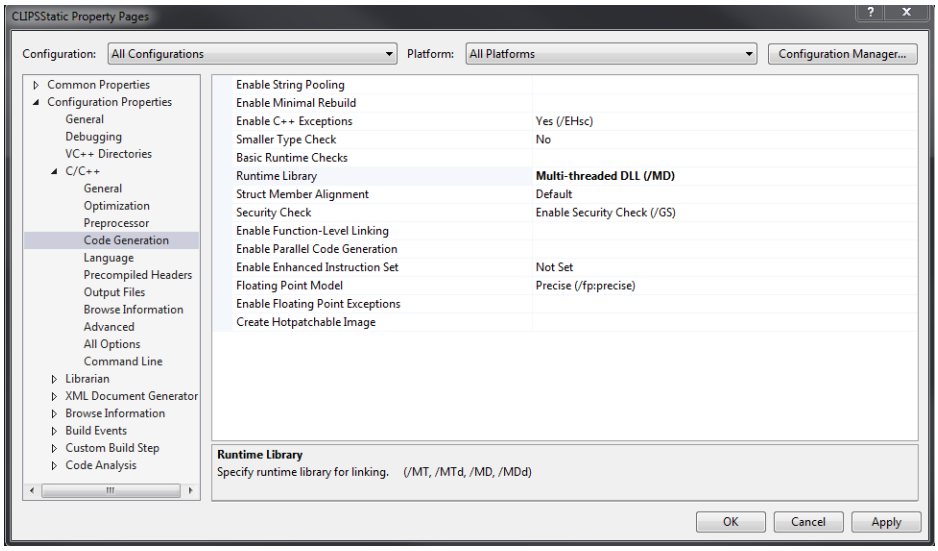

Figura 2.1: Configuración de la librería en Visual Studio 2013

en equipo; ya que, al compilarse automáticamente todo el código en el momento de abrir el proyecto, este se negaría a cargarse en el sistema de cualquier miembro que no tenga una versión completamente funcional de la librería y podría llegar a provocar cierta corrupción en los datos del proyecto. Algo realmente problemático si otros miembros del equipo no son de un perfil técnico, una situación especialmente habitual en equipos de desarrollo de videojuegos donde la faceta artística está tan marcada.

Veamos entonces el uso de librerías estáticas. Como es bien sabido, estas tienen ciertos inconvenientes como el aumento de tamaño del resultado. Aunque en nuestro caso estos inconvenientes no resultan tan problemáticos, principalmente porque partimos de la base de que nuestro proyecto ocupará de media entre 100 y 1000 veces más que la librería estática que queremos enlazar. Sin embargo, en esta ocasión es cuando nos toparemos con esa barrera antes mencionada: enlazar una librería estática en una base de código sumamente compleja necesita de ciertos pasos previos.

Por suerte para nosotros, como suele pasar con las herramientas populares, la documentación de Unreal Engine 4 no se limita a la proporcionada por Epic Games, sino que cuenta con un gran apoyo de la comunidad. Es por ello que operaciones que en condiciones normales podrían resultar confusas, como el enlazado de una librería estática, quedan explicadas a la perfección como puede verse en [Cha15]. No obstante, explicaremos brevemente el proceso.

2.1.1. Compilación de la librería

Como en cualquier enlazado de librería, nuestra primera parada será compilar la librería estática de CLIPS. El proceso es trivial, ya que si hemos descargado los paquetes de CLIPS correspondientes a la versión 6.3 de la web oficial, nos encontraremos con que ya tenemos algunas versiones compiladas de las librerías, con ejemplos y proyectos de Visual Studio preparados para recompilar dichas librerías según sea necesario. En nuestro caso tendremos que hacer uso de esa recompilación, ya que Unreal Engine 4 requiere de algunas características activas a la hora de compilar nuestra librería.

Necesitaremos hacer una compilación de la librería para una versión x86 de Windows y otra para una versión x64. Esto se debe a que así brindamos la posibilidad de exportar nuestro proyecto a ambas plataformas y, aunque nos vayamos a centrar en la versión x64 del proyecto (por resultar lo más común hoy día), el motor nos obliga a emplear una librería creada expresamente para dicha arquitectura si no queremos encontrarnos con problemas más adelante. El proceso de elección de plataforma en la que se compila la librería no será explicado como se avisó en el capítulo de introducción, ya que se da por hecho que el lector sabe hacer un manejo básico de Visual Studio 2013.

También nos hará falta compilar la librería de manera que esté enlazada a un archivo DLL. Puede sonar contradictorio teniendo en cuenta que hace unas cuantas líneas establecimos que una librería dinámica no era la mejor opción en nuestro caso, sin embargo, es de esta manera como el sistema de compilación de Unreal Engine 4 espera recibir una librería estática, por lo que debemos asegurarnos de que sea así como lo haga. Para ello la opción que tendremos que activar es la de generar el código como un archivo DLL multihilo, usando la opción `/MD` del compilador.

Realizados estos ajustes, tendremos nuestra librería compilada y lista para ser usada.

2.1.2. Script de compilación

Ahora que ya tenemos nuestra flamante librería estática dispuesta para ser integrada en Unreal Engine 4, deberemos facilitar al motor todos los archivos necesarios. Para ello haremos uso de la carpeta de Plugins, como ya se comentó en el capítulo de introducción, disponiendo los archivos en dos directorios separados: uno para las librerías que hemos compilado y otro para el código de integración en el que se basan estas librerías. Este paso no es obligatorio, ya que sólo deberemos proporcionar la ruta de los archivos al motor para que este realice la integración correctamente, pero siempre es útil disponer de una buena organización de archivos.

El siguiente paso consistirá en proporcionar los directorios en los que hemos colocado todos los archivos a nuestra solución de Visual Studio

2013, ya que necesitará las rutas en el momento de compilar. Tras ello podremos empezar a trabajar en el script de compilación de nuestro proyecto de Unreal Engine 4.

El script de compilación es un archivo que, al contrario que la mayoría del código que se escribe cuando se trabaja con Unreal Engine 4, está escrito haciendo uso de C#. Este archivo se encuentra en la carpeta Source de nuestro proyecto, siempre llevará por nombre `<NombreDelProyecto>.build.cs` y sirve todas las dependencias que deben cumplirse en el momento de compilar el proyecto para una determinada plataforma. De esta manera, pueden definirse diferentes módulos dependiendo del objetivo del proyecto que se realice. Por ejemplo: un videojuego 2D haría uso de módulos para dibujo de *sprites* y físicas 2D, mientras que un videojuego enfocado al uso de realidad virtual querrá hacer uso de los módulos de soporte para ésta.

En nuestro caso, usaremos este script para añadir a las librerías adicionales de compilación nuestra librería de CLIPS. Para ello, deberemos implementar una función en la que se tome como parámetros un objeto de tipo `TargetInfo`, que incluye toda la información sobre la plataforma objetivo de la compilación y asegurarnos de proporcionar la ruta correcta a las librerías para Windows x86 y x64 en cada caso. Una vez tengamos la ruta correspondiente tan sólo deberemos añadirlas haciendo uso del método `PublicAdditionalLibrearies.Add` que toma como único parámetro de entrada una cadena con la ruta de la librería en cuestión.

¡Ya está! Por fin tendremos la librería integrada en el motor y podremos hacer uso de ella en cualquier momento. Sin embargo, ahora debemos hacernos otra pregunta: ¿cómo hacemos uso de la librería? Y, sobre todo, ¿cómo podemos comunicar a nuestros actores en escena con ella?

2.2– Definición de ActorComponent

Hasta el momento, todo lo que se ha visto ha sido un asunto bastante generalista. Casi podría decirse que es un proceso habitual e independiente de la tecnología que se emplee. No obstante, a partir de este punto la cosa va a centrarse bastante más en la API y las herramientas que ofrece Unreal Engine 4. Empecemos, pues, por la última pregunta que hemos dejado planteada, ya que el uso de la librería dependerá principalmente de la API de CLIPS. ¿Cómo comunicamos pues esta API con los elementos en escena?

Como es de suponer, no existe una respuesta única a esta pregunta. Ahora bien, dada la estructura orientada a componentes que plantea el motor, podemos agregar a cualquier actor en escena un trozo de código en el que se hagan llamadas a la librería. Pero antes de lanzarnos de cabeza a escribir, debemos pensar en el uso que nosotros queremos darle a esta librería; esto es, que sea versátil y de fácil acceso para el usuario. Por esto mismo, la solución por la que hemos optado para resolver esta situación es,

precisamente, la creación de un componente denominado `CLIPSComponent` que se pueda agregar a cualquier actor desde la interfaz del motor y que, solo con ello, ya ofrezca toda la funcionalidad básica de CLIPS de forma visible para el usuario. Para ello, haremos uso de la clase `ActorComponent`.

Como ya se mencionó anteriormente, la clase `ActorComponent` ofrece las características necesarias que buscamos para poder trabajar de manera aséptica a lo que ocurre en escena o en el propio actor al que se agregue este componente. Sin embargo, también permite que el actor creado por el usuario al que se agregue este componente pueda comunicarse con él, por lo que nos valdremos de esta posibilidad para transmitir a CLIPS toda la información que necesite para ejecutarse correctamente.

```
1  class UE_CLIPS_API UCLIPSComponent : public UActorComponent
2  {
3  //...
4      UCLIPSComponent();
5
6      virtual void BeginPlay() override;
7
8      virtual void TickComponent( float DeltaTime, ELevelTick
           TickType, FActorComponentTickFunction* ThisTickFunction )
           override;
9
10     virtual void BeginDestroy();
11
12     void InitCLIPS();
13
14     UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "CLIPS")
15         bool loadFromFile = false;
16     UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "CLIPS")
17         FText buildCode;
18     UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "CLIPS")
19         FText fileLocation;
20     UFUNCTION(BlueprintCallable, Category = "CLIPS")
21         void CLIPSBuild(FString fs);
22     UFUNCTION(BlueprintCallable, Category = "CLIPS")
23         void CLIPSLoad(FString fs);
24     UFUNCTION(BlueprintCallable, Category = "CLIPS")
25         FString CLIPSOutput();
26     UFUNCTION(BlueprintCallable, Category = "CLIPS")
27         void CLIPSClear();
28 //...
29 };
```

Código 2.1: Extracto de la cabecera de CLIPSComponent

En el código 2.1 podemos observar como en la cabecera de la clase `CLIPSComponent` se hace uso de varias funciones heredadas (`BeginPlay`, `TickComponent` y `BeginDestroy`) que, como su propio nombre indican, sir-

ven para ejecutar un código al inicio del juego, de manera cíclica durante el juego o en el instante en el que este actor llama a su destructor. De estas tres, haremos un mayor uso de la primera, ya que nos permitirá inicializar el entorno de CLIPS en el momento que cargue la escena. Además de esto, incorporamos la función `InitCLIPS`, en la que incorporaremos la susodicha inicialización del entorno.

Acompañando a las definiciones anteriores también veremos diferentes funciones precedidas de la macro `UFUNCTION` que serán las que expongamos para el uso en Blueprint. Estas funciones se encargarán de proporcionarle a CLIPS un código de entrada en el caso de `CLIPSBuild` o una ruta a un archivo *.clp* en el de `CLIPSLoad`. También se incorporan dos funciones con las que leer un stream de salida de CLIPS o borrarlo, en estas dos funciones indagaremos más adelante.

Finalmente, también encontramos algunos parámetros expuestos a Blueprint y precedidos de la macro `UPROPERTY` que definen si la entrada para el entorno de CLIPS será la variable `buildCode` o `fileLocation`. Si nos centramos en la implementación de esta clase, únicamente deberemos tener en cuenta que en el archivo *.cpp* deberemos usar la directiva `#include 'clipscpp.h'` para tener acceso a las funciones de la librería de CLIPS. El funcionamiento de la librería se basa fundamentalmente en el uso del objeto `CLIPSCPPEnv`, que servirá de punto de entrada al entorno de CLIPS y que crearemos en nuestra función de inicialización. Desde este objeto podremos hacer uso de las funciones `Build` y `Load`, ambas recibiendo como único parámetro de entrada una cadena de texto y siendo su uso idéntico al que hemos definido para nuestro objeto `CLIPSComponent`, por lo que la implementación de estos métodos consistirá en delegar la tarea en el entorno de CLIPS.

```
1 class CLIPS::CLIPSCPPEnv;
2 class CustomFileRouter : public CLIPS::CLIPSCPPRouter
3 {
4 public:
5     virtual int Query(CLIPS::CLIPSCPPEnv *, const char *);
6     virtual int Print(CLIPS::CLIPSCPPEnv *, const char *, const
          char *);
7 };
```

Código 2.2: Extracto de la cabecera de una clase hija de CLIPSCPPRouter

Mención especial, como ya adelantábamos, recibe la implementación de los métodos dedicados a gestionar la salida de CLIPS. Para realizar esta tarea deberemos tener en cuenta que CLIPS hace uso de los denominados routers para dirigir la salida del entorno, como por ejemplo cuando realizamos una función printout. Por tanto, para acceder a este contenido deberemos implementar una clase que herede de `CLIPSCPPRouter` e implemente los métodos que se muestran en el código 2.2, indicando cúando y

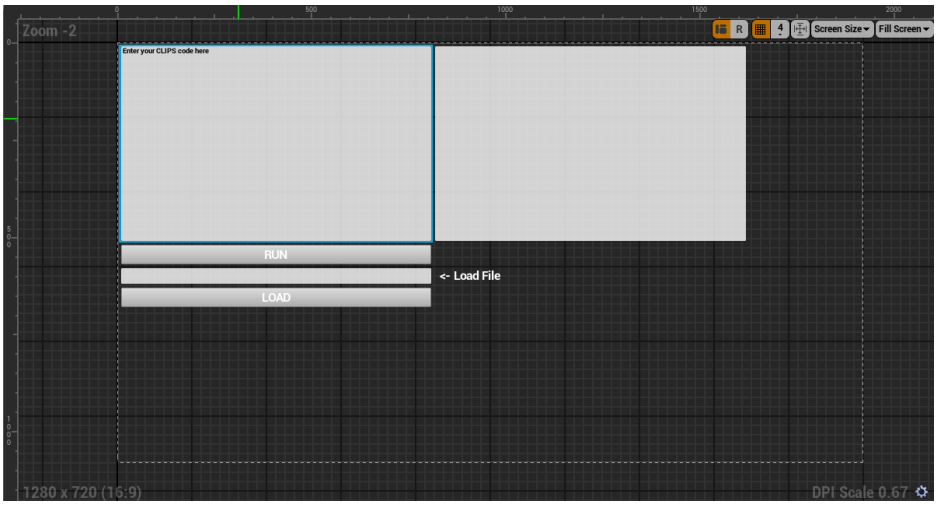

Figura 2.2: Disposición del UserWidget en el editor

cómo mostrar el texto en las funciones `Query` y `Print`. En nuestro caso, nuestra implementación consistirá en almacenar todo en una variable de tipo `std::stringstream` para poder realizar un volcado de su contenido según nos haga falta. Por último, para hacer que el entorno reconozca nuestro nuevo router, deberemos indicarle al entorno de CLIPS que lo agregue a la lista de routers con el método `AddRouter`. Indicando en este último el nombre que deberá indicarse en las expresiones de CLIPS para usarlo, su nivel de prioridad y el propio objeto que queremos emplear como router.

Tras todo esto, estaremos realmente cerca de cumplir nuestro objetivo para esta primera implementación básica: tenemos la librería, tenemos un componente que es capaz de comunicar escena y librería... tan solo nos falta comprobar que está todo funcionando correctamente. Para ello, crearemos una interfaz visual en la que podamos definir la dirección de un archivo *.clp* o un código de entrada y se nos muestre el resultado.

2.3– Creación de UserWidget

Aunque parezca sorprendente, no vamos a tocar código para crear esta interfaz. En lugar de ello, vamos a asegurarnos de que nuestra exposición de funciones a Blueprint funciona correctamente y haremos uso de nuestro componente `CLIPSComponent`. Es decir, que en esta parte vamos a ver el funcionamiento de nuestra integración desde la interfaz de Unreal Engine 4, exactamente igual que lo vería un usuario en estos momentos.

Para comenzar con la creación de esta interfaz, lo primero será crear un actor de tipo `UserWidget` (técnicamente no es un actor, ya que en lugar de

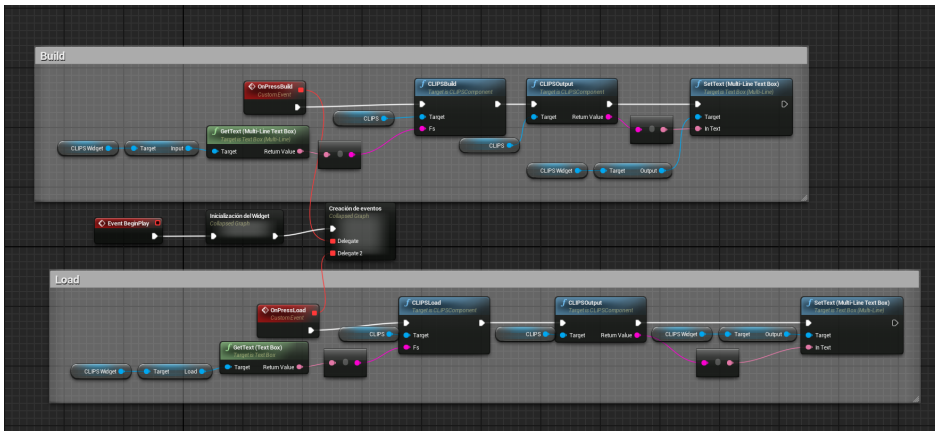

Figura 2.3: Extracto del Blueprint asociado al UserWidget

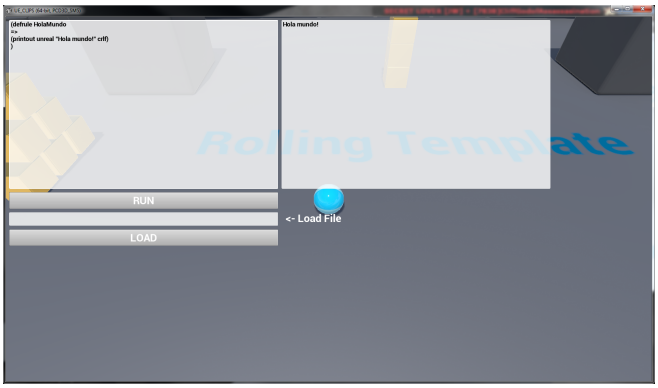

Figura 2.4: Resultado final de la integración básica

estar en una escena 3D se dispone directamente sobre un plano de pantalla 2D, pero a efectos prácticos el funcionamiento es idéntico por lo que esta distinción no es especialmente importante). En este *widget* dispondremos tres campos de texto multilínea y dos botones, tal y como se indica en la figura 2.2.

Tras ello, crearemos un segundo actor que contendrá nuestro componente `CLIPSComponent` y, además, se encargará de crear una instancia del *widget*, tomar de él los datos de entrada y enviar los de salida. El proceso se realizará usando el lenguaje de scripting visual y, como puede apreciarse en la figura 2.3, consistirá en añadir una función a cada botón para que se encarguen de tomar el parámetro de entrada y aplicar la función correspondiente en cada caso. El texto de salida se colocará haciendo una llamada regular cada ciertos segundos al método `CLIPSOutput`. En la figura 2.4 podemos observar el resultado final.

CAPÍTULO 3

Integración multihilo de CLIPS

En el capítulo anterior hemos logrado una interfaz visual en Unreal Engine 4 que es capaz de recibir código de CLIPS y devolvernos resultados. No obstante, si intentamos hacer el procesamiento de una base de conocimiento más compleja, nos encontraremos con un problema bastante evidente: CLIPS tardará bastante tiempo en obtener una conclusión y, mientras este proceso concluye, el juego queda completamente congelado.

Debemos tener en cuenta que un videojuego no sólo se trata de un sistema que interactúa en tiempo real con el usuario, como podría ser la clásica interfaz gráfica de un programa ofimático, sino que además necesita que el tiempo de respuesta sea el más corto posible para que la experiencia sea satisfactoria. Usualmente se apunta a conseguir entre 30 y 60 FPS[1], lo que nos deja en el mejor de los casos con apenas 33 milésimas de segundo para hacer todos los cálculos que necesitemos, obtener unos resultados, interpretarlos y mostrar las conclusiones en el nuevo fotograma que se dibuje en pantalla. Y esto sin contar con que debemos dejar tiempo para el renderizado de dicho fotograma, lo que usualmente consume la mayor parte de ese tiempo. Entonces, ¿cómo podemos hacer uso de CLIPS para situaciones en las que la respuesta no se encuentre en unas pocas milésimas de segundo? Muy fácil: haciendo que CLIPS sea un proceso independiente.

Lo que quizás no sea tan sencillo es llevarlo a la práctica, ya que antes deberemos realizar una serie de consideraciones y tener claro nuestro plan de acción. Así que vamos a revisar bien qué queremos hacer antes de ponernos manos a la obra.

[1] Fotogramas por segundo.

3.1– Diseño de la integración multihilo

Es importante notar que separar CLIPS en un proceso independiente nos brinda muchas posibilidades, sin embargo, también plantea nuevos problemas que antes no teníamos. Para empezar, necesitaremos determinar claramente cómo vamos a controlar este proceso, ya que, como es bien sabido, si no se mantiene un control estricto de un hilo de procesamiento paralelo este puede ocupar memoria y capacidad de procesado de manera desmesurada. Y, además de esto, deberemos establecer un sistema de comunicación con este proceso que nos permita tanto enviarle una información en su inicio como realizar consultas en cualquier momento y obtener así las conclusiones que necesitamos interpretar en nuestro juego.

Para poder cumplir estas exigencias, emplearemos un diseño basado en tres clases: una que servirá de gestor del proceso, otra que hará de cliente de este proceso y por último la clase que contendrá el proceso paralelo en sí mismo. Pasemos pues a verlas en detalle:

- `CLIPSComponent`, la recordaremos del capítulo anterior. Será la clase que haga de gestor del proceso, encargándose de iniciarlo y cerrarlo según sea necesario. Las modificaciones en la clase que ya habíamos programado serán bastante leves, ya que la mayor diferencia será que ahora delegará sus funciones expuestas al propio hilo de procesamiento y que se encargará de cerrar y destruir el hilo una vez se llame al destructor del propio componente. De esta manera nos aseguramos de que el hilo de CLIPS solo puede aparecer en una escena tanto como lo haga el actor que contiene al componente que la gestiona.

- `CLIPSClient`, este será un segundo tipo de componente que nos permitirá enviar y recibir información del proceso de CLIPS. Sus funciones están orientadas a exponer al usuario los usos más habituales que hará del sistema de manera sencilla (esto es, incluir, retirar y comprobar hechos en la base de conocimiento actual de CLIPS) y dejar que pueda realizar alguna consulta más compleja en caso de necesidad. Su funcionamiento se detallará más adelante.

- `CLIPSWorker`, esta clase será la verdadera protagonista de nuestro diseño. Es una clase que hereda de FRunnable, la clase que dispone la API de Unreal Engine 4 para generar hilos de procesamiento paralelo en el motor. En ella definiremos unas funciones similares a las que teníamos en nuestra versión anterior de `CLIPSComponent`, encargándose de inicializar el entorno, pasarle el código que debe cargar, gestionar los routers y obtener la salida de estos últimos. Además de, por supuesto, ofrecer una funcionalidad añadida para atender consultas de los clientes.

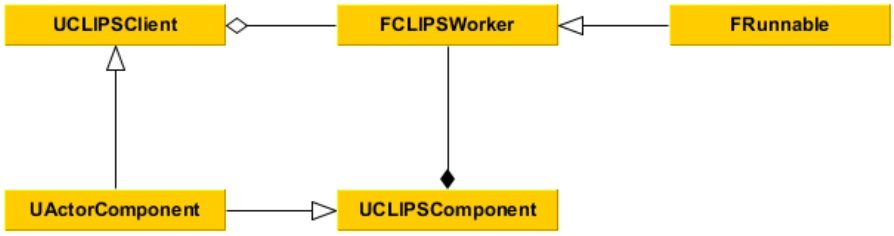

Figura 3.1: Diagrama UML para la integración multihilo

Como puede observarse, se trata de un diseño sin demasiadas complicaciones pero que cumplirá perfectamente con su cometido. Eso sí, ahora queda entender realmente el funcionamiento de estos dos nuevos elementos que hemos agregado a nuestra integración.

3.2– Definición de CLIPSWorker

Para entender cómo vamos a gestionar el funcionamiento de este hilo paralelo, debemos entender primero cómo gestiona Unreal Engine 4 estos procesos. Como se ha mencionado anteriormente, la opción más viable es el uso de la herencia de la clase `FRunnable`, que nos ofrece una interfaz sencilla con la que gestionar el inicio, la parada y el funcionamiento del hilo paralelo de manera similar a como lo haríamos con cualquier otro componente de Unreal Engine 4.

Es importante notar que el funcionamiento del hilo es cíclico, es decir, una vez que tengamos nuestro proceso en iniciado, se hará una llamada periódica a la función `Run` hasta que se cierre el proceso con alguna de sus funciones de parada. En nuestro caso esto nos será tremendamente útil ya que podremos desglosar el procesamiento que realice CLIPS en etapas. En otras palabras, cada vez que se ejecute la función `Run`, el entorno de CLIPS realice un número limitado de ejecuciones de reglas, en lugar de ejecutarse hasta que no encuentre más reglas que aplicar. Hacer esto nos brindará la oportunidad de intercalar consultas externas cada cierto número de reglas ejecutadas, de manera que los clientes del proceso reciben información en tiempo real de lo que está ocurriendo en el entorno de CLIPS.

Y son precisamente estas consultas la parte más interesante de esta integración. Porque claro, ¿cómo hacemos para gestionarlas? Ya hemos adelantado que, haciendo que CLIPS ejecute una serie de reglas (o una sola) y acto seguido atienda a consultas en cada iteración que realice, podemos lograr una atención en tiempo real para los clientes. Pero aún no hemos solucionado el problema que plantea tener una serie de consultas externas que nos llegan de forma totalmente asíncrona. Y lo que es más,

tampoco hemos abordado cómo accederá al proceso para realizar dichas peticiones.

```
class FCLIPSWorker : public FRunnable
{
//...
public:
    FCLIPSWorker();
    virtual ~FCLIPSWorker();

    virtual bool Init();
    virtual uint32 Run();
    virtual void Stop();

    void CLIPSBuild(FString fs);
    void CLIPSLoad(FString fs);
    void CLIPSRun(int inInt);
    void CLIPSReset();
    std::string GetOutput();
    int32 CLIPSEval(FString fs);
    void ClearOutput();

    static FCLIPSWorker* Runnable;
    static FCLIPSWorker* ClipsInit();
    static FCLIPSWorker* GetClipsWorker();
    static void Shutdown();
//...
private:
    void HandleEval();
    UPROPERTY()
    static TMap<int, FString> EvalMap;
    UPROPERTY()
    static TQueue<int, EQueueMode::Mpsc> EvalQueue;
//...
};
```

Código 3.1: Extracto de la cabecera de la clase CLIPSWorker

Vayamos por partes. En primer lugar solucionaremos el acceso de los clientes al proceso haciendo uso de una variante del conocido patrón de diseño *Singleton*, con el que nos aseguramos tener solo una instancia de CLIPS en memoria al mismo tiempo. Para ello, crearemos una función estática con la que gestionar el acceso al proceso. De manera que, si el proceso está inicializado, devolverá un puntero al objeto ya creado y, a diferencia de lo que ocurriría en un Singleton clásico, si el proceso no se ha creado (y por tanto este puntero es nulo) devolverá un valor nulo, avisando así al cliente de que no existe un entorno con el que interactuar en ese momento. De esta manera nos aseguramos que ningún cliente puede crear de manera accidental un nuevo hilo de procesamiento que se nos vaya

de las manos y, al mismo tiempo, tener un acceso global al mismo.

Ahora que ya tenemos una vía de acceso al proceso, debemos solucionar cómo se realizarán las peticiones de información por parte de los clientes. Para resolver esto, propondremos el uso de colas de consulta, a la que añadiremos cada petición que se nos realice. Un sistema que nos permitirá ir solucionando las consultas pendientes en el procesamiento cíclico que mencionábamos anteriormente pero que nos deja con un problema nuevo: ¿cómo recibe el cliente la respuesta a su consulta? No puede quedar esperando hasta que le toque "su turno" como es obvio, si hacemos esto perderíamos todas las ventajas de tener un procesado paralelo que no ralentiza la ejecución del programa principal; especialmente en situaciones en las que haya una cantidad gigantesca de consultas que saturen la cola. En lugar de una espera indefinida, lo que haremos será emplear eventos y delegados. Términos que, probablemente, parezcan más propios de lenguajes como C# que de C++, pero de los que sin embargo Unreal Engine 4 ofrece una implementación en su API de la que podemos valernos igualmente para dar una solución limpia a este problema.

Con estas herramientas en nuestro "arsenal", nuestra solución pasará por generar un número de identificación único que se devuelve al cliente cada vez que realiza una petición de consulta. Nuestro `CLIPSWorker` almacenará este número de identificación junto a la consulta a la que va asociada en un diccionario o mapa, además de hacerlo en la cola de consultas. De esta manera, cada vez que se ejecute una consulta podremos disparar un evento en el que se indicará el número de la consulta y el resultado obtenido; dejando como responsabilidad del cliente que ha realizado la consulta estar suscrito al evento y conocer de qué peticiones espera resultados.

Y, ya que hemos sacado la responsabilidad de los clientes a la luz, hablemos de ellos.

3.3– Definición de CLIPSClient

Como bien sabemos ya, esta clase será el componente que agregaremos a nuestros actores siempre que queramos que interactúen con `CLIPSWorker` que hemos definido. Para ello dispondrá principalmente de la función de petición de consulta, ya que será a través de estas consultas como modifique y obtenga información del sistema.

Para lograr esto, nos aseguraremos de que en su inicialización, el cliente intenta contactar con `CLIPSWorker` activo a través de su método estático `GetCLIPSWorker` como veíamos en la sección anterior. Una vez logrado dicho contacto, almacenará en un puntero la referencia a `CLIPSWorker` para mayor velocidad en las futuras interacciones. Eso sí, teniendo en cuenta que esta referencia puede volverse nula en cualquier momento si el proceso es apagado. Además, como puede darse el caso de que el cliente se

inicie antes que el propio proceso, nos aseguraremos siempre que vayamos a hacer uso del puntero de que, si este es nulo, intente nuevamente recuperar la referencia de `CLIPSWorker`, si no lo consigue notificará un error y bloqueará la función que estuviese realizando. Por último, es importante destacar que no solo almacenaremos una referencia al Worker, sino que además usaremos la función `EvalQueryCallback` para suscribirnos al evento definido en éste para notificar los resultados de una consulta. Esta suscripción la tendremos que eliminar cuando se llame al destructor del cliente, ya que de lo contrario nos encontraríamos con una violación de acceso por intentar acceder a un puntero no válido.

```
1  class UE_CLIPS_API UCLIPSClient : public UActorComponent
2  {
3  //...
4  public:
5      UCLIPSClient();
6
7      virtual void BeginPlay() override;
8      virtual void BeginDestroy() override;
9
10     virtual void TickComponent(float DeltaTime, ELevelTick TickType
           , FActorComponentTickFunction* ThisTickFunction) override;
11
12     UFUNCTION(BlueprintCallable, Category = "CLIPS")
13         void EvalQuery(const FString fs);
14     UFUNCTION(BlueprintCallable, Category = "CLIPS")
15         void CheckFact(const FString tipo, const FString id);
16     UFUNCTION(BlueprintCallable, Category = "CLIPS")
17         void AssertFact(const FString tipo, const FString id);
18     UFUNCTION(BlueprintCallable, Category = "CLIPS")
19         void RetractFact(const FString tipo, const FString id);
20     UPROPERTY(BlueprintAssignable, Category = "CLIPS")
21         FCLIPSClientEventResult OnQueryResult;
22 //...
23 private:
24     UFUNCTION()
25         void EvalQueryCallback(int32 queryID, FString result);
26     void LinkToWorker();
27     TArray<int32> querysInQueue;
28     FCLIPSWorker* CLIPSWorker;
29 //...
30 };
```

Código 3.2: Extracto de la cabecera de la clase CLIPSClient

Por último, tan solo debemos asegurarnos de almacenar una lista de todos los identificadores de consultas de las que el cliente está esperando resultados, de esa manera podremos filtrar aquella información que nos interesa en cada situación.

Además, como ya se hizo en el caso de `CLIPSComponent` se exponen diferentes funciones al usuario por medio de las macros Ufunction, además de un segundo evento llamado `FCLIPSClientEventResult` al que llamaremos tras filtrar la información en `EvalQueryCallback` para que el usuario pueda definir una lógica de actuación desde Blueprint en caso de que fuera necesario. No indagaremos más en los detalles de la implementación de esta clase por ser análoga a lo realizado en el capítulo anterior con `CLIPSComponent`.

3.4– Solucionando problemas de concurrencia

Antes del cerrar el capítulo, merece una mención especial y por separado los problemas de concurrencia que pueden surgir – y surgen – cuando se realiza un proceso como este en paralelo.

Deberemos tener muy en cuenta en la clase `CLIPSWorker` que tanto las colas de eventos como las propias funciones de apagado e inicio de la clase pueden ser llamadas desde múltiples partes del código de manera simultánea. Por esto mismo, es bastante común encontrarse con numerosas violaciones de acceso si no se toman las precauciones adecuadas. En nuestro caso hemos solventado estos errores haciendo uso de un cerrojo clásico para proteger las zonas de código sensibles; especialmente todas aquellas que incluyan la modificación de una variable.

No obstante, el uso de un cerrojo no bastará para todas las ocasiones. En los momentos de inicializar y cerrar el proceso puede darse el caso de que tengamos errores de concurrencia a pesar de haber empleado cerrojos, por ello deberemos aplicar una doble comprobación a los estados de las variables dentro del código protegido por el cerrojo, de manera que si encontramos que el proceso ya ha sido iniciado o cerrado (según cada caso) pararemos inmediatamente la ejecución de la función.

CAPÍTULO 4

Creación de un proyecto de ejemplo

Si todo ha ido bien, en estos momentos tendremos la integración terminada. Sí, ya hemos pasado la parte difícil. Ahora viene disfrutar de la recompensa de nuestro trabajo y comprobar si todo funciona como debe. Para ello vamos a hacer un pequeño proyecto de juego que haga un uso variado de nuestra implementación. No será ninguna obra digna de reconocimiento, pero sí una prueba perfectamente válida para comprobar la funcionalidad de CLIPS en Unreal Engine 4.

Nuestro juego será similar al primer ejemplo que poníamos en la introducción de este trabajo. Es decir, tendremos control de un personaje y un escenario compuesto de tres interruptores y, en vez de una puerta, vamos a tener un panel luminoso que se active cuando se pulsen todos los interruptores. Nada ostentoso como decíamos, pero merecerá la pena explicar un poco su proceso. Vamos a ello.

4.1– Estructuras de CLIPS

En primer lugar, tendremos que establecer la lógica que va a seguir nuestro juego por la parte que respecta a CLIPS, nuestro gran protagonista en este ejemplo.

Por un lado, haremos uso de dos templates. Con una representaremos lo que vamos a denominar condiciones del juego y que resumirían el estado actual del juego, en nuestro caso estas condiciones representarían el estado de los interruptores o si se ha activado o no el panel luminoso. Con la otra template definiremos los eventos del juego, es decir, todo aquello que puede ocurrir en el juego de lo que esperamos una notificación; en nuestro caso, si se han pulsado todos los interruptores o no. Por supuesto, esto podría considerarse una condición y a efectos de implementación no habría mayor diferencia; sin embargo, vamos a plantearlo de tal manera que la estructura fuera la usual en un juego de estas características.

Además, necesitaremos definir las reglas con las que trabajará CLIPS. No serán muchas, tan solo dos: una regla inicial con la que iniciaremos el sistema indicándole que aún no se ha activado el evento y otra con la que activaremos dicho evento al encontrar los tres interruptores activos. Con estas dos reglas podremos trabajar perfectamente.

Esto será todo por la parte de CLIPS. Por supuesto, en un caso real la estructura sería bastante más amplia, especialmente las reglas, pero para nuestro ejemplo esto será suficiente.

4.2– Creación de los actores

Bien, ahora pasemos a trabajar la parte de Unreal Engine 4. Necesitaremos crear, al menos, media docena de actores: nuestro personaje, un entorno sólido por el que desplazarse, los tres interruptores y nuestro panel luminoso. Por supuesto, para el caso que estamos tratando solo son interesantes los cuatro últimos, así que nos centraremos en ellos.

Los interruptores los crearemos de una manera genérica para luego poderlos instanciar, por lo que solo tendremos que definir el comportamiento general de uno de ellos. Dado que actuarán a modo de clientes, haciendo uso de `CLIPSClient`, deberemos agregarlo a su lista de componentes. Hecho esto, deberemos añadir la lógica necesaria para detectar cuando el interruptor esté activo o no y, actuando en consecuencia, haremos uso de nuestro componente para enviar esta información a CLIPS por medio del uso de `AssertFact` y `RetractFact`. Los nombres concretos de las condiciones que activaremos los dejaremos como una variable pública y editable, de manera que al colocar estos actores en escena podamos dar los nombres correspondientes a cada instancia. En la figura 4.1 se muestra un gráfico con el resultado final para este actor.

Para el panel luminoso tendremos que trabajar un poco más aunque, al igual que en el interruptor, también haremos uso del componente `CLIPSClient` y definiremos su lógica de manera general, dejando los detalles concretos para la instancia en escena. Sin embargo, a diferencia del interruptor, el panel no actuará sobre la base de conocimiento de CLIPS, sino que tendrá que comprobar cuando se dispara el evento que le activa. En esta clase de situaciones, debemos considerar que si realizamos una consulta continua al proceso de CLIPS bloquearemos el proceso principal del juego, dado que las respuestas del proceso paralelo son asíncronas. Para evitar esto, limitaremos la cantidad de consultas que realicemos al sistema; en nuestro caso realizaremos una consulta cada 0.2 segundos, un tiempo suficiente para que la reacción del panel resulte instantánea a ojos del jugador.

Además, tampoco queremos realizar una nueva consulta si todavía no hemos recibido la respuesta de la anterior, por lo que estructuraremos la lógica de manera que realicemos una primera consulta inicial al comienzo

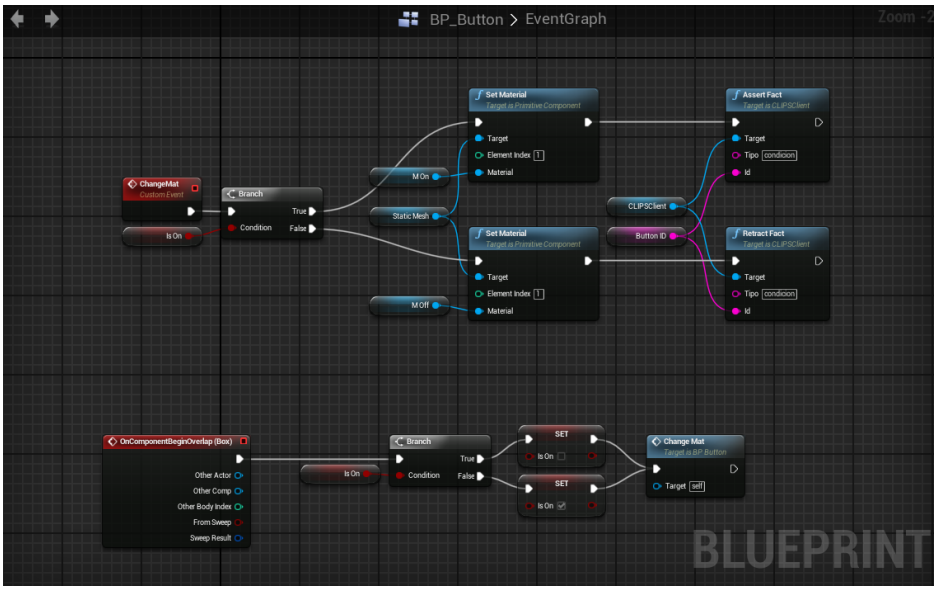

Figura 4.1: Extracto del Blueprint del actor interruptor

del juego y, una vez recibida la respuesta en el evento `OnQueryResult`, enviaremos una nueva consulta pasado 0.2 segundos en el caso de que ésta no sea positiva. Por supuesto, una vez recibida una respuesta positiva, activaremos el panel y dejaremos de realizar consultas. Cabe añadir que, de manera análoga al caso del interruptor, realizaremos la consulta con el método `CheckFact` y dejando el nombre del evento a consultar como una variable pública y editable. En la figura 4.2 se muestra un gráfico con el resultado final para este actor.

Finalmente, tan solo nos quedará colocar nuestro componente maestro, `CLIPSComponent`. Para ello, tenemos varias opciones: crear un actor para tal efecto que tan solo contenga este componente, usar algún elemento en escena para que lleve la responsabilidad (como el propio protagonista) o usar algunos métodos más avanzados como hacer que el componente sea parte del denominado `GameMode`; que sin ser un actor propiamente dicho es capaz de contener componentes al igual que uno. En nuestro caso, vamos a hacer uso de esto último por comodidad, pero el proceso sería análogo para incluirlo en cualquier otro actor. Una vez tengamos nuestro componente añadido lo único que tendremos que realizar será marcar la casilla para cargar desde un archivo e indicar la ruta al archivo *.clp* donde hayamos definido todo el código de CLIPS.

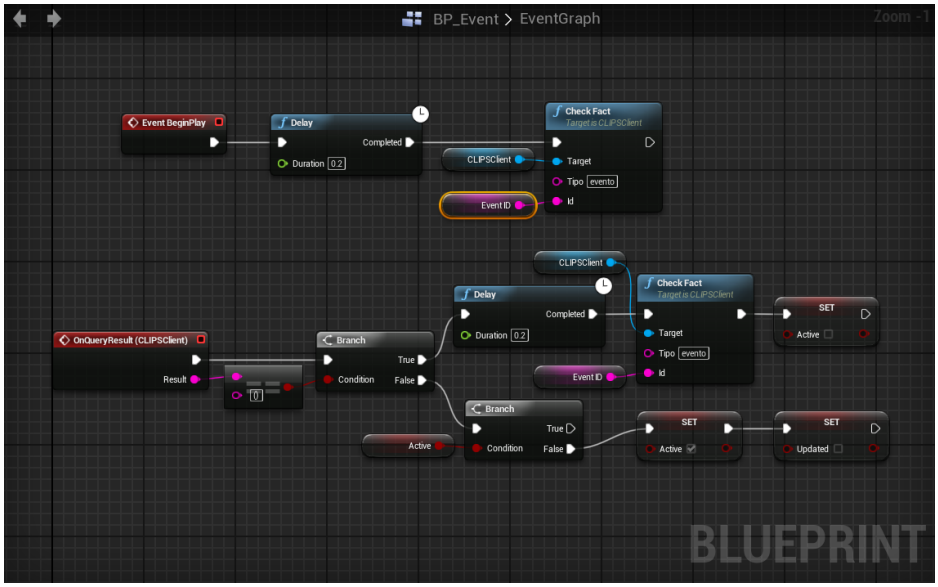

Figura 4.2: Extracto del Blueprint del actor panel

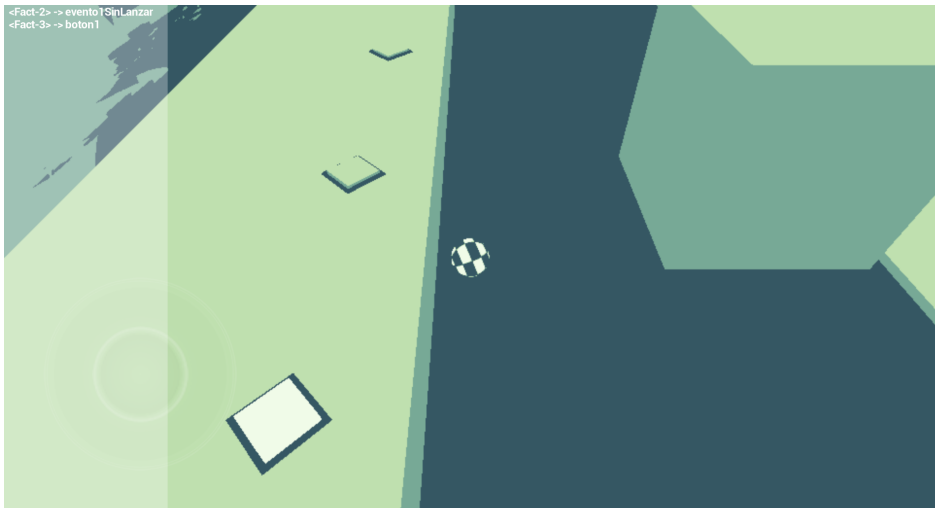

Figura 4.3: Resultado final del proyecto de ejemplo

4.3– Resultado final

Con todo esto ya tendremos nuestro proyecto casi terminado. Tan solo resta colocar todos estos actores en escena como buenamente queramos y darle los nombres apropiados a las variables que habíamos dejado por definir. Tras ello ya estaremos preparados para darle al botón de *play* y disfrutar de nuestro trabajo funcionando (con algo de suerte) a la perfección.

Por supuesto, también podremos exportar nuestro resultado a un ejecutable independiente del entorno del editor de Unreal Engine 4 siguiendo el procedimiento habitual de exportación; no obstante, es importante remarcar que la integración que aquí se ha descrito y que se incorpora en el código adjunto sólo es válida para plataformas Windows. Para otras plataformas el proceso sería análogo, aunque probablemente la integración multihilo y la compilación de librerías variarían en diferentes puntos.

CAPÍTULO 5

Resultados y conclusiones

5.1– Pruebas de rendimiento

Empecemos por el rendimiento. Es importante que veamos qué impacto tiene hacer uso de este sistema para solucionar el problema del seguimiento del estado de un juego frente al rendimiento "base" que haga uso de soluciones más habituales como variables booleanas. Esta importancia deriva, sobre todo, de esa necesidad que comentábamos más atrás de mantener, al menos, 30 FPS constantes durante el desarrollo de un videojuego para que la experiencia del usuario no se vea afectada. Por tanto, si el impacto es muy grande es probable que el uso de esta solución no sea el más adecuado para todas las situaciones.

Por suerte, en las pruebas que hemos podido realizar, el impacto es inapreciable. Como ejemplo hemos tomado el proyecto de juego explicado en el capítulo anterior y lo hemos implementado haciendo uso únicamente de variables booleanas; obteniendo en ambos casos un tiempo de procesado por fotograma que oscila entre los 7 y los 8 milisegundos. Por supuesto, este tiempo dependerá de la máquina [1] en la que esté funcionando el proyecto, sin embargo, el resultado representa que para usos sencillos de este sistema no sufriríamos ningún impacto en el rendimiento. Ahora bien, debemos considerar que esta prueba es un caso extremadamente simplificado, por lo que es probable que el impacto no se pueda apreciar. Escalemos un poco más la situación.

Pongamos, que ejecutamos un juego mientras hacemos funcionar el proceso de CLIPS en bucle de manera ininterrumpida y comparemos los resultados con los obtenidos en una situación idéntica pero sin el entorno de CLIPS funcionando de fondo. Haciendo esto deberíamos poder apreciar

[1] Como referencia, todas estas pruebas se han realizado en una máquina que usa una CPU AMD FX-8350@4.00GHz, una GPU AMD R9-280X@1.1GHz y 16 GB de RAM DDR3@1.9GHz

el impacto que supone tener un segundo hilo funcionando en paralelo con el hilo principal del juego. No obstante, como era de esperar, no apreciamos ninguna diferencia. La medida de tiempo exacta varía dependiendo del entorno donde hagamos la prueba, pero siempre resulta en el mismo tiempo de procesado por fotograma esté o no esté activo CLIPS. ¿Por qué ocurre esto? Bueno, debemos tener en cuenta que, a fin de cuentas, es un hilo paralelo el que está funcionando por lo que, mientras la máquina tenga potencia suficiente en sus núcleos para trabajar en paralelo, no obtendremos impacto alguno en el rendimiento. Por supuesto, la cosa cambiaría muchísimo si hablamos de procesadores que no admitan varios hilos de procesamiento. No abordaremos este caso por ser algo poco habitual hoy día y, en el caso de que nos encontrásemos ante tal situación, es probable que el propio motor fuera incapaz de funcionar dadas su exigencia, por lo que nuestra implementación de CLIPS sería el menor de los problemas en una situación como esta.

Así pues, ¿en qué situación podemos poner a nuestro sistema en un compromiso de rendimiento? Bien, la respuesta es clara: haciendo consultas. Dada su naturaleza asíncrona, las consultas son de lejos el mayor problema al que nos debemos enfrentar cuando hacemos uso de esta implementación. Para comprobarlo hemos realizado una serie de pruebas en las que escalamos el número de actores que se encuentran realizando consultas a CLIPS y hemos anotado como se ve afectado el tiempo medio de procesado de fotograma para cada caso. En la siguiente tabla se muestran los resultados:

Nº elementos (consultas)	**Tiempo (ms)**
1	8.33
2	8.34
5	8.34
10	8.60
20	8.40
50	8.45
100	8.90
500	12.75
1000	330.15

Cuadro 5.1: Tabla de tiempo medio de procesado por fotograma de elementos con consultas

Nº elementos (sin consultas)	**Tiempo(ms)**
1000	10.05

Cuadro 5.2: Tabla de tiempo medio de procesado por fotograma de elementos sin consultas

Como puede observarse, hemos visto que el sistema escala de manera aceptable hasta las 100 consultas simultáneas. Sin embargo, a partir de esa cantidad el rendimiento empieza a verse afectado. Hemos querido añadir como dato adicional el tiempo medio de procesado de fotograma para el caso de 1000 elementos sin componente de CLIPS, para que se observe que la diferencia de rendimiento se debe, precisamente, al uso de nuestra implementación y no a la propia saturación de elementos en escena. Si bien estos resultados de rendimiento no son nada desalentadores, ya que 100 consultas simultáneas y constantes es un caso bastante extremo en el uso habitual que se le pueda dar a un sistema como este, debemos hacernos a la idea de que en un uso real deberemos limitar las consultas al máximo. Una buena manera sería agrupar varias consultas en una sola, usando algún tipo de gestor en escena. Aunque también hay que tener en cuenta que esto supondría una complejidad extra para el diseñador de niveles de la que pretendíamos alejarnos en este proyecto.

5.2– Potencial de uso

Y, ahora que ya sabemos qué tal rinde la implementación, hablemos de su potencial de uso. Ya que, como mencionábamos hace un momento, el número de consultas que podemos realizar de manera simultánea, aunque limitado, es suficiente para la mayoría de situaciones reales en la que se quiera emplear el sistema.

Un área donde sin duda el potencial de uso de este sistema experto se vería realmente potenciado sería en videojuegos donde se necesite constante evaluación del estado actual de la partida. Por ejemplo, en juegos de alta estrategia donde la cantidad de posibilidades para la máquina son muy numerosas, disponer de un sistema experto de fondo capaz de estar evaluando constantemente la situación y obtener conclusiones de manera rápida facilitaría el desarrollo de una inteligencia artificial eficiente.

Otra situación donde disponer de CLIPS como entorno de fondo supone una gran ayuda, en este caso al desarrollo. Son en las situaciones donde, como comentábamos en la introducción de esta documentación, un diseñador de niveles debe contemplar una cantidad de variables muy numerosa y necesita poder tener un acceso rápido y ordenado a estas. Por citar un ejemplo, en el género de las aventuras gráficas existe una metodología desarrollada por Ron Gilbert que, como explica en [Gil15], permite obtener un diagrama de eventos y condiciones que deben darse en un juego para que este avance según el plan, aunque su implementación suele ser muy laboriosa y consume una gran cantidad de recursos. En este caso CLIPS actuaría como una capa de abstracción para el diseñador, ya que se podría crear un conjunto de reglas generales para el seguimiento de diagramas como los citados en [Gil15] y, a partir de ahí, hacer una traslación directa a la lógica de juego haciendo uso de estructuras similares a

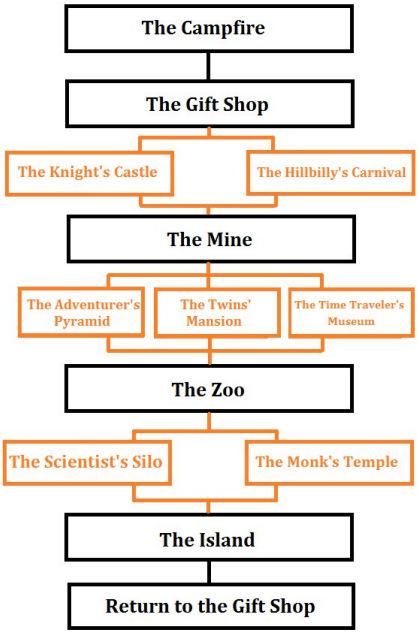

Figura 5.1: Un ejemplo de un diagrama Gilbert

las que planteamos en nuestro proyecto de ejemplo.

Además de estos dos ejemplos citados, no es complicado encontrar situaciones en las que un sistema experto pueda ser de utilidad tanto para un videojuego como para su desarrollo. Y disponer de todo el potencial de CLIPS de una manera directa y sencilla cubre todas estas posibilidades, por lo que creemos que el potencial de la implementación aquí ofrecida es bastante alto dada las aplicaciones prácticas que pueden surgir de él.

5.3– Limitaciones y mejoras posibles

Aun así, este trabajo dista de ser perfecto y, además de las ya mencionadas limitaciones en rendimiento, presenta varios problemas y limitaciones que, probablemente, podrían solventarse con un enfoque diferente al que hemos adoptado.

El más notable de estos problemas es, una vez más, es el sistema de consultas. Y no por los que ya hemos mencionado. Si no porque, tratándose de uno de los objetivos de este trabajo facilitar a los diseñadores su tarea, encontramos varios aspectos en los que la actual implementación de las consultas no cumplen completamente con este cometido.

Por un lado, en nuestra implementación solo hemos ofrecido funciones expuestas para asuntos habituales, tales como consultar un hecho y

añadirlo o retirarlo de una base de conocimiento, sin embargo, para cualquier otra consulta algo más compleja el diseñador de niveles depende completamente de sus conocimientos de CLIPS para resolverla. Una posible solución para esto sería ampliar la librería de funciones que se exponen al usuario, aunque probablemente siempre nos quedase alguna situación sin cubrir en la que este problema volvería a salir a la luz.

Otro asunto problemático con las consultas es su propia naturaleza. Estas se basan en el comando `eval` de CLIPS, lo cual presenta bastantes limitaciones propias del sistema, como no poder acceder a variables locales. Esto recorta de manera drástica las posibilidades que se le pudieran dar a estas consultas al no poderse realizar ninguna acción de relativa complejidad. Para estas situaciones deberemos buscar soluciones alternativas haciendo uso del sistema de reglas y de routers ya que, de momento, CLIPS no ofrece ninguna otra posibilidad.

Y, siguiendo con las consultas, encontramos otro problema – aunque bastante menor – en la comprobación que hace CLIPS de que estas están bien formadas. Siempre que hagamos una consulta, CLIPS hará una comprobación de que es un comando válido que la función `eval` pueda ejecutar. En caso de no ser así se lanzará una excepción. Hasta aquí todo normal, no debería ser un problema capturar una excepción con un bloque `try-catch`. Pero lo es. Por defecto, Unreal Engine 4 tiene desactivada la captura de excepciones, por lo que no podemos hacer uso de estos bloques y, cada vez que se haga una consulta que CLIPS considere inválida, nuestro juego se cerrará con una excepción. Por supuesto, hay una solución evidente: activar nuevamente el uso de estos bloques. Aunque, como no todo podría ser tan fácil, activar estos bloques provoca una gran cantidad de incompatibilidades en el momento de exportar el proyecto final, además del tiempo que lleva compilar el motor al completo para que trabaje con esta opción. Por esto mismo hemos decidido no poner solución a este problema, ya que ello derivaría en problemas de muchísima mayor complejidad y, una vez bien formadas las consultas, esto deja de resultar un problema. Eso sí, puede resultar una pérdida de tiempo bastante irritante si, por culpa de un paréntesis mal colocado, tenemos que reiniciar todo el entorno de desarrollo de Unreal Engine 4.

Por último, algo que ya se comentábamos en el capítulo anterior, es que esta solución solo es válida para entornos Windows. Por lo que las opciones de exportación tan amplias que ofrece Unreal Engine 4 se ven recortadas si empleamos esta implementación sin más. Para poder hacer frente a eso deberíamos ampliar la implementación dando soporte a todas las plataformas. Aunque, como es lógico, esto llevaría un trabajo y tiempo bastante considerable.

5.4– Futuras investigaciones

Y, como no podría ser de otra manera, un Proyecto de Fin de Carrera no estaría completo si no hablamos de todos los frentes que quedan abiertos para la investigación.

Las opciones, sin lugar a dudas, no son pocas. Dado que el enfoque de este proyecto era el de lograr la posibilidad de emplear CLIPS y todo su potencial en Unreal Engine 4, no hemos llegado a indagar realmente en todo este amplio abanico de aplicaciones. Como ya se ha comentado en este mismo capítulo, las aportaciones que puede hacer CLIPS al desarrollo de videojuegos no son pocas y, por tanto, un estudio pormenorizado para cada caso sería necesario para sacar el mayor provecho posible de esta integración.

Además, también hemos visto que han quedado ciertos problemas que con casi total seguridad puedan resolverse si se realiza un estudio en mayor profundidad de los mismos. Por esta parte podríamos decir que la propia integración en su estado actual, aunque plenamente funcional, aún deja espacio a la investigación de la misma.

Por último, una vía de investigación muy interesante con la que podría continuarse el estudio iniciado en este proyecto es la integración de una herramienta para la definición de sistemas expertos que esté desarrollada de manera nativa para Unreal Engine 4 y las implicaciones que esto tendría en el rendimiento. Especialmente si consideramos el enorme impacto que tienen las consultas.

Manual de usuario

CLIPSComponent

1. **Uso**

   CLIPSComponent es una clase de ActorComponent que puede agregarse a cualquier tipo de actor para crear un proceso paralelo que contenga al entorno de CLIPS. Para usarlo tan solo se debe agregar a la lista de componentes de cualquier Blueprint que herede de la clase Actor.

2. **Funciones**

   a) `void CLIPSLoad(FString filePath)`

   Esta función recibe como parámetro la localización de un archivo *.clp*, tomando como ruta base la carpeta Content del proyecto, y lo carga en el entorno de CLIPS.

   b) `void CLIPSBuild(FString code)`

   Esta función recibe como parámetro un código para que sea interpretado por CLIPS.

   c) `FString CLIPSOutput()`

   Esta función devuelve todo el contenido que se haya volcado en la salida del router con el identificador `'unreal'`. Es importante notar que el contenido volcado será acumulativo y no se borrará sin indicación expresa.

   d) `void CLIPSClear()`

   Esta función se encarga de borrar todo el contenido que se haya volcado en la salida del router con el identificador `'unreal'`.

3. **Advertencias**

   Es responsabilidad del usuario asegurarse de la vida del componente, si éste o el objeto al que está agregado son eliminados de maneras no convencionales podrían producirse errores ya que rompería el ciclo de vida natural del entorno de CLIPS.

CLIPSClient

1. **Uso**

   CLIPSClient es una clase de ActorComponent que puede agregarse a cualquier tipo de actor para hacer uso de un proceso paralelo que haya creado previamente la clase CLIPSComponent. Para usarse tan solo se debe agregar a la lista de componentes de cualquier Blueprint que herede de la clase Actor.

2. **Funciones**

   a) `void AssertFact(FString tipo, FString id))`
   Esta función se emplea para realizar una petición de añadir a la base de conocimiento de CLIPS un hecho que siga la forma `(tipo (slot id))`. Recibe como primer parámetro el tipo del hecho y como segundo el identificador.

   b) `void RetractFact(FString tipo, FString id)`
   Esta función se emplea para realizar una petición de retirar de la base de conocimiento de CLIPS un hecho que siga la forma `(tipo (slot id))`. Recibe como primer parámetro el tipo del hecho y como segundo el identificador.

   c) `void CheckFact(FString tipo, FString id)`
   Esta función se emplea para realizar una petición de comprobar si existe en la base de conocimiento de CLIPS un hecho que siga la forma `(tipo (slot id))`. Recibe como primer parámetro el tipo del hecho y como segundo el identificador.

   d) `void EvalQuery(FString query)`
   Esta función se emplea para realizar una consulta personalizada al entorno de CLIPS. Debe tenerse en cuenta que el formateado y las limitaciones son las propias de un comando `eval` en CLIPS. Es importante notar que, en caso de que la consulta no esté bien formado, el editor se cerrará con una excepción cuando ésta se ejecute.

3. **Eventos**

   a) `OnQueryResult(FString result)` Este evento se emplea para recibir la información correspondiente a las consultas realizadas. Entrega como parámetro una cadena de texto donde se incluye la respuesta de CLIPS a la consulta realizada.

4. **Advertencias**

   Debe tenerse especial cuidado con la cantidad y la frecuencia de las consultas, ya que tienen un gran impacto en el rendimiento y, en

algunos casos, podrían llegar a bloquear por completo la ejecución del hilo principal del juego.

Bibliografía

[Cha15] Chatman, Bob, 2015. 'Linking Static Libraries Using The Build System.' `https://wiki.unrealengine.com/Linking_Static_Libraries_Using_The_Build_System`. [Online; accedido por última vez el 28 de agosto de 2015].

[Gil15] Gilbert, Ron, 2015. 'Puzzle Dependency Charts.' `http://grumpygamer.com/puzzle_dependency_charts`. [Online; accedido por última vez el 28 de agosto de 2015].