InvasionEngine 2: Motor gráfico para crear videojuegos mediante scripts

Abel Cano Quintana

Resumen— Este proyecto tiene como objetivo el diseño, implementación y test de un motor eficiente y extensible de juegos 3D sencillos, con soporte para periféricos, audio y controlable mediante scripts externos. El motor nace de una motivación personal de conocer el funcionamiento interno de un motor de juegos y aprender todo un conjunto de tecnologías transversales asociadas. Aparentemente, los resultados obtenidos son favorables en escenas simples, pese a que el sistema necesitaría un desarrollo más amplio para poderse comparar equitativamente con otros motores existentes.

Palabras clave— Direct3D, DirectX 11, Extensiones, Intérprete de scripting, Kinect, Mono, Motor de juegos, OpenAL, XInput.

Abstract— This project aims to design, implement and test an efficient and extensible 3D game engine, with support for peripherals and 3D audio, fully controllable through external scripts. The engine comes out from a personal motivation of knowing the internal functioning of a game engine and learning a whole set of associated transversal technologies. Apparently, most of results obtained are favorable in simple 3D scenes although the system would need a larger development in order to be able to compare it equitably to other existing engines.

Index Terms— Direct3D, DirectX 11, Kinect, Script interpreter, Mono, Game engine, OpenAL, Plugin, XInput.

1 Introducción

Hoy en día los videojuegos cautivan a un gran público a través de tramas fantásticas y gráficos cada vez más realistas. Una parte significativa del éxito es debido a los motores, herramienta importante en el sector de los videojuegos. Estos motores plasman conceptos abstractos y muy técnicos para dar vida a proyectos creativos de la mano de un público imaginativo menos entendido. Es esa complejidad técnica subyacente la que me ha generado interés para llevar a cabo el presente proyecto. En una visión retrospectiva, es evidente la evolución creciente que han tenido los juegos desde sus inicios, desde el popular Pong [1]. En la actualidad, existe una fuerte apuesta por la inmersión de los jugadores en el juego, ya no sólo a nivel sonoro y visual, sinó en la posibilidad de hacerles formar parte de él y transmitirle emociones [2]. El jugador ahora tiene la posibilidad de verse a sí mismo dentro del juego, moverse libremente por un escenario sin necesidad de mandos, observar objetos de fantasía mediante realidad aumentada o incluso ver a través de los ojos de su héroe favorito utilizando unas gafas de realidad virtual. Se puede decir que el propio usuario ha pasado a ser un controlador más; se le ha integrado dentro del juego. Ya no se conforma con ver en una pantalla una imagen 2D o 3D, ahora quiere tener la capacidad de formar parte de la realidad del juego, existiendo una clara intención de ir cada vez más allá y mejorar su experiencia. Se trata de un concepto sencillo y natural, pero considerablemente complejo a nivel técnico. No obstante, todo esto no es sólo gracias a los motores. Tales adelantos técnicos en la industria del videojuego no habrían sido posibles sin los recientes avances en capacidad de cómputo de las máquinas de los últimos años o la existencia de periféricos y sensores como Kinect [3].

1.1 Estado del arte

En la actualidad se encuentran disponibles una amplia variedad de motores de juego, sean comerciales u Open Source. Entre los más destacados del sector se encuentran Unreal Engine [4] de Epic Games, CryEngine de la alemana Crytek, Frostbite de la sueca DICE, Anvil Next de Ubisoft Montreal y Unity 3D [5] de Unity Technologies.
Tanta variedad no es fruto de la casualidad. La gran cantidad de motores existentes no viene sólo dada por las prestaciones de los mismos ya que, en esencia, todos comparten comportamientos similares [Tabla 2]. La diferencia radica en cómo han decidido abordar ciertas cuestiones como la gestión de recursos, la propia implementación de algoritmos y/o formatos de fichero. Asímismo, cada juego es diferente y necesita realizarse con un motor que se adapte a sus necesidades individuales. Un juego hiperrealista necesitará de un motor potente y pesado. En el caso de un juego sencillo, un motor complejo no hará más que enlentecer su desarrollo y posterior funcionamiento. A menudo la razón que impulsa a las compañías a desarrollar un motor propio no es otra que poseer tecnología propia que se ajuste a sus necesidades técnicas. Otro motivo podría ser el no tener que depender de terceras partes y reducir costes cuando se van a producir varios juegos. Los motores son herramientas que se pueden reutilizar en varios proyectos si son los suficientemente generalistas. En términos gráficos, el API Direct3D [6] és el sistema de representación más usado. La versión más moderna es la 12, aunque la mayoría de sistemas aún

- *Correo electrónico de contacto: abel.cano@e-campus.uab.cat*
- *Mención realizada: Ingeniería del Software.*
- *Trabajo tutorizado por: Enric Martí Gòdia (DCC)*
- *Curso 2016 / 2017*

utilizan el 11 por razones de compatibilidad con sistemas operativos y adaptadores gráficos más antiguos. Es también destacable que la mayoría de los motores anteriormente citados también implementan una interfaz alternativa basada en OpenGL. Esto les permite correr en sistemas operativos alternativos a Microsoft Windows como Mac y Linux. En temas de audio hay una cierta divergencia pues suelen incluir implementaciones basadas en interfaces nativas de la plataforma en cuestión. En lo que todos coinciden es en la inclusión de un entorno de desarrollo y generador de mapas. Esta funcionalidad no estará presente en el proyecto pues no se ajusta al tiempo permitido. A nivel de scripting no existe tampoco un estándar y los entornos de programación van desde LUA (en el caso de CryEngine y Source) a .NET de Unity o lenguajes gráficos propios (UnrealScript / Blueprints). Finalmente, la capacidad de un motor de funcionar en múltiples plataformas resulta en la actualidad un factor decisivo, que evita que el programador de juegos tenga que realizar múltiples versiones del mismo juego para funcionar en distintas plataformas. Por todo ello, y por la posibilidad de crear juegos propios, con una herramienta a medida y que ayude a entender desde los fundamentos cómo funcionan y cuales son los entresijos de un motor, se ha dado cabida a este proyecto. InvasionEngine 2 tiene como origen el motor dedicado *InvasionEngine One* del juego "Invasión" [7], el cual fue una primera incursión en el desarrollo de motores gráficos para la asignatura de *LIS* (*Laboratorio Integrado del Software*).

1.2 Marco de actuación y objetivos

Este proyecto tiene como objetivo desarrollar una nueva versión del motor gráfico InvasionEngine, el cual realizará representaciones 3D eficientes con soporte para audio 3D y permitirá el control de periféricos externos como mandos de juego, teclado y ratón. Además, al programador se le ofrecerá el control de todas y cada una de las funcionalidades del motor por medio de accesos al hardware de bajo nivel en las interfaces basadas en scripts. Asímismo, su diseño modular facilitará la creación de módulos y extensiones alternativas para expandir sus capacidades y suplir futuras carencias. Esto permitirá añadir al sistema nuevas características o plataformas de ejecución sin que los juegos generados deban modificarse lo más mínimo, manteniendo una total independencia de la plataforma de ejecución. En un TFG paralelo, por ejemplo, se desarrolla una interfaz para controlar y obtener información de dispositivos Kinect de Microsoft directamente desde el sistema de scripts de IE2.
Para llevar a cabo este ambicioso objetivo, se ha descompuesto el objetivo principal en subobjetivos, clasificados por plugins o extensiones [Fig. 1]. Estas extensiones son:

- ***Scripting***: el cual permitirá la interpretación, arranque y ejecución de los juegos del usuario.
- ***Renderer gráfico 3D a bajo nivel:*** el cual se encargará de la representación de las escenas 3D.
- ***Administración y reproducción de efectos sonoros 3D y música***: gestionará el hardware de audio, efectos sonoros y decodificación de música
- ***Contenedor multimedia IEF:*** Ficheros de datos
- ***Administrador de ventanas***: Ventanas de error y dónde se pintará finalmente la imagen renderizada.
- ***Funciones generales del motor***: Incluye logs, funciones comunes y mensajes de depuración.
- ***Gestión de periféricos de E/S:*** Teclado, ratón y periféricos de control de juegos como mandos, palancas y volantes, etc.

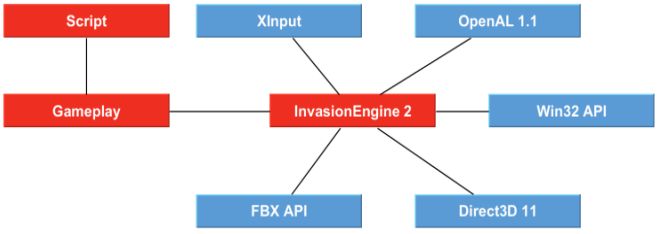


Figura 1 – Tecnologías utilizadas para el desarrollo del motor

A continuación se presenta el artículo, el cual está compuesto por seis apartados. En los dos primeros se expondrán objetivos y metodología. Después, etapas previas, diseño, implementación y test de la aplicación; subdivididas estas últimas en siete subpartes, para dar paso, finalmente, a resultados, conclusiones y líneas futuras de desarrollo.

2 METODOLOGÍA

Dada la naturaleza dinámica de un motor, en la cual se extienden y modifican constantemente funcionalidades, una metodología tradicional tipo cascada no se adapta a la mecánica de trabajo requerida porque no permite volver a etapas previas a coste razonable. Por lo tanto, se ha escogido una metodología ágil y adaptada para satisfacer las exigencias del proyecto. Cabe destacar que el aspecto diferenciador más significativo de esta respecto a otras metodologías ágiles es que las iteraciones vienen dadas por la implementación de funciones y no por tiempo de sprint. Se trata de un proceso iterativo en espiral que se repetirá tantas veces como funcionalidades se tengan que implementar. Por cada funcionalidad se realiza internamente un ciclo de desarrollo que consta de las siguientes cinco fases o etapas: Test (script) de referencia, diseño, implementación, binding y validación. Si durante alguna parte del ciclo interno algo debe reformularse, puede volverse atrás con facilidad, pues los cambios serán los suficientemente pequeños para no afectar al resto de tareas. A continuación se define la actividad de cada etapa:

- ***Test***: Consiste en generar un script de referencia mediante un símil al *Test-Driven Development* que compruebe la correcta ejecución de la funcionalidad que se implementará y validará en la presente iteración.
- ***Diseño***: Se diseñan nuevas clases o se modifica la estructura de las actuales para lograr la nueva funcionalidad planeada.

- ***Implementación***: Se programan/implementan las funcionalidades diseñadas.
- ***Binding***: Se realiza la unión de las funciones de bajo nivel del motor con las de alto nivel del script, a través de una biblioteca runtime o "tiempo de ejecución" en que, a modo de puente, se ejecutarán las internal calls necesarias para el funcionamiento. Este mecanismo se describirá con detalle más adelante en el punto 4.1.
- ***Validación***: Se comprueba que el script implementado en la primera fase se ejecuta correctamente y genera el resultado esperado sin que por ello dejen de funcionar otros scripts de iteraciones anteriores.

3 Preparación

Antes de empezar cualquier diseño e implementación, se realiza un análisis previo y exhaustivo de los requisitos del proyecto. Con este paso quedarán reflejadas con claridad y exactitud todas las funcionalidades que deberá incluir el programa, clasificadas en funcionales y no funcionales. También se incluirá un diagrama de objetivos AND-OR. Este gráfico será importante pues determinará la prioridad de todos y cada uno de los objetivos. Estas informaciones quedarán reflejadas en un documento SRS. También se llevarán a cabo documentos que reflejen los errores que vayan siendo encontrados a lo largo del desarrollo y de la gestión de la configuración del proyecto. En este último se mostrará la organización del proyecto, buenas prácticas de trabajo y cómo manipular el repositorio.

4 Diseño e Implementación

Este apartado detalla cómo se ha diseñado el motor y cómo es su implementación. Para un mejor entendimiento, los módulos se explican en apartados propios. Se dividirán en los siguientes: Motor de scripts, arquitectura de extensiones, render gráfico Direct3D 11, plugin XInput [8], extensión de audio 3D y contenedor multimedia IEF.

4.1 Motor de scripts

El motor de scripts es la parte del programa que permite al usuario desarrollar juegos con InvasionEngine 2. Asímismo le dará la capacidad para extender la aplicación con nuevos algoritmos y funciones personalizadas no contempladas en el proyecto. InvasionEngine 2 basa su sistema de scripting en The Mono Open Source Project [9], un conjunto de herramientas a código abierto cuyo desarrollo fue iniciado por la compañía Ximian. Mono contiene bibliotecas, herramientas y compiladores que implementan un clon de .NET compatible y multiplataforma, que permitirá llevar los scripts a plataformas diferentes a Windows (como Linux, Mac o incluso Android). Los programas no necesitarán ni recompilarse. Una tecnología competidora del mismo tipo sería Java, dónde es común ejecutar un mismo paquete de aplicación en diferentes sistemas operativos sin que se produzcan conflictos. Mono provee compiladores de Basic, C# y Python, entre otros, un tiempo de ejecución común para todos y reimplementaciones compatibles de las bibliotecas de funciones originales. Lo más interesante, es su capacidad de scripting. Más allá de ejecutar programas .NET, Mono es capaz de actuar como intérprete esclavo de scripts y embutir sobre un programa de C/C++ su entorno para lograr dotarlo de capacidades de scripting. El host C++ expone al script invitado sus funciones de bajo nivel, de forma parecida a una syscall de un sistema operativo, pudiendo controlar también su flujo de ejecución [10]. Asimismo, podrá manipular y acceder a las propiedades, variables, objetos y funciones del script, todo a un coste computacional reducido. Esta capacidad resulta muy interesante de cara al desarrollo de un motor de juegos. Cuando el script invitado realiza una llamada a una de estas funciones del motor, el compilador JIT y el tiempo de ejecución de Mono la atraparán, cederán el control al código de nuestro motor (host C++ vinculado) para que realice una determinada acción o evalúe un resultado y lo devuelva. Como es lógico pensar, las operaciones típicas que realiza un videojuego (gameplay) son muy pesadas y desde un contexto de alto nivel implican un sobrecoste en CPU excesivo (overhead). Además, el programador de juegos no querrá lidiar con las crípticas llamadas al API del sistema operativo. Mono puede ayudar aquí. Se puede diseñar e implementar dicho conjunto de rutinas en C++ a bajo nivel usando APIs del calibre de Direct3D 11 u OpenAL. A continuación, se les oculta al programador de juegos, abstrayendo su complejidad y generando macrofunciones que realicen tareas que le puedan ser útiles. Dibujar un modelo, cargar una textura o reproducir una secuencia de audio de manera automática por poner algunos ejemplos. Mientras que realizar muchas pequeñas llamadas al API genera gran coste y sobrecarga, una única llamada unificada no, evitando así generar un impacto excesivo sobre el sistema. Cabe destacar que IE2 no es el único sistema que hace uso de Mono como tecnología de scripting. Unity es un ejemplo de motor, ampliamente popular, que recurre a Mono para el mismo fin.

4.2 Arquitectura de extensiones

Teniendo en cuenta el apartado anterior en el que se ha expuesto el funcionamiento de los puentes syscall/internal call, se podrían considerar las extensiones a bajo nivel (gráficos 3D, audio, entrada, etc.) como si fuesen los controladores del sistema operativo virtual. Estas extensiones se implementan como proyectos independientes en forma de bibliotecas de enlace dinámico DLL. Serán las encargadas de exponer a los scripts sus

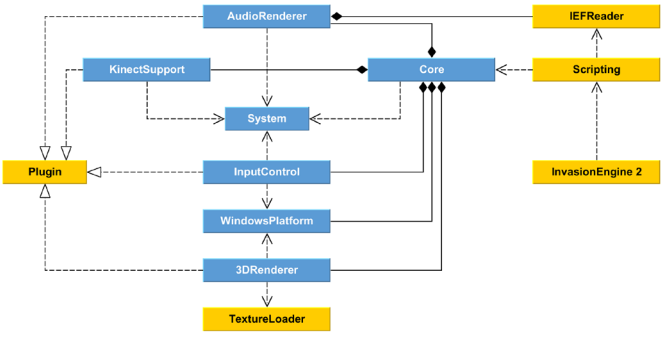


Figura 2 – Escena en modo wireframe (alambres)

interfaces o APIs para realizar determinadas tareas. El programador de scripts llamará a un punto de entrada de función definido en nuestro runtime de InvasionEngine 2, quién a su vez, realizará la correspondiente llamada interna al motor. El plugin "Core" accederá a continuación al controlador correspondiente para ejecutar la acción y/o elaborar una respuesta [Fig. 2]. Es importante destacar que el punto de entrada del runtime no es más que una interfaz, lo que significa que se podría sustituir un controlador de bajo nivel por otro compatible sin que el script se viera afectado lo más mínimo (no haciendo falta, tan siquiera recompilar). Estos puntos de entrada de runtime o "bindings" se definen para cada iteración dispuesta en la metodología. Para más información sobre la arquitectura del software, consultar anexo [Fig. 12].

4.3 Renderer gráfico Direct3D 11

El sistema de rendering 3D se basa en el API de renderizado a bajo nivel de Direct3D 11 [6], dentro del subconjunto de Microsoft DirectX. Esta extensión tiene como objetivo ocuparse de los cálculos gráficos y de la abstracción de las complejas llamadas de Direct3D 11. Además, el sistema de rendering tiene funciones para identificar los monitores y adaptadores gráficos instalados; inicializarlos y comprobar la memoria de vídeo disponible, la cual, en algunas ocasiones puede condicionar las aplicaciones subyacentes. Las rutinas de inicialización incluyen la creación del estado del rasterizador, el cual establece, entre otras cosas, el tipo de pintado (si es FILL-SOLID (Sólido) o WIREFRAME (alambres)) y variables como CullMode, el cual determinará la forma de dibujo de las partes ocultas del gráfico [Fig. 3].

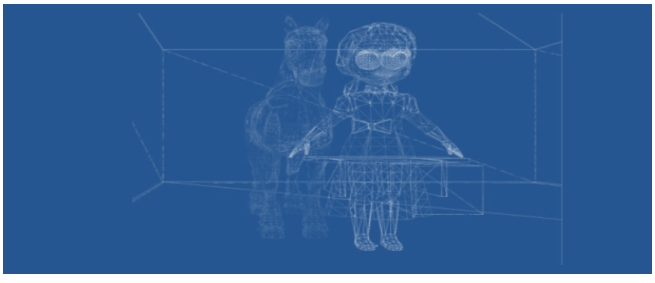


Figura 3 – Escena en modo wireframe (alambres)

La extensión incorporará también funciones como "LoadContent", que aceptará vectores de vértices e índices. Con esta información, se generarán las estructuras y descriptores de búfer necesarios para programar el pipeline gráfico y aceptar el tipo de datos recibidos. Cabe destacar que la programación de dicho pipeline se hará completamente por medio de Shaders HLSL (High Level Shader Language) pues a partir de Direct3D 10 (Shader Model 4.0) desaparece el antiguo sistema de tuberías fijas tradicionales y los programas-shader pasan a ser la única forma de control posible del adaptador de gráficos [11]. La extensión hace por defecto uso del API de Direct3D 11 (Shader Model 5.0) pero si no hay hardware compatible disponible se intentará iniciar con una versión del API gráfico más antigua como Direct3D 10.1 (Shader Model 4.1) a modo de fallback. En caso de persistir un error de inicialización, se volverá a probar una inicialización usando el entorno más reciente que soporte la gráfica (10.0 – SM 4.0, 9.0c – SM 3.0 o 9.0 – SM 2.0). Esto es posible gracias al compilador de Shaders "fxc.exe" integrado en el set de herramientas de compilación de Visual Studio, el cual tiene una opción de compilación que permite la traducción de Shaders entre versiones. Por ejemplo, "vs_4_0_level_9_1" el cual corresponde a un intérprete de Vertex Shader 4.0 (DirectX 10) pero generaría código compilado compatible con 2.0, y por lo tanto, DirectX 9. Si aun así no se encuentra un adaptador gráfico compatible, IE2 está programado para recurrir a renderizado por software WARP. WARP (*Windows Advanced Rasterization Platform*) de DirectX, es un rasterizador software de alto rendimiento que simula la existencia de un entorno gráfico compatible y emula el hardware de la tarjeta gráfica faltante usando software. El uso de WARP garantiza que el motor podrá funcionar incluso en las peores condiciones, aunque a costa de una fuerte penalización en rendimiento y velocidad de ejecución [12].

4.3.1 Cámara 3D

La cámara se controla desde el script, el cual podrá situar la misma en una posición 3D determinada del espacio. El motor se encargará de calcular automáticamente el target focus (si fuera necesario) para mantener la cámara recta y alineada, añadiéndole dinámicamente un factor α (foco = posición + α * distancia) proporcional al movimiento. A partir de los datos de cámara, la función de actualización realizará los cálculos matriciales pertinentes para generar la nueva matriz de visualización. Este tipo de cálculos 3D deben realizarse en CPU y están presentes a lo largo de todo el módulo de representación. Se llevan a cabo mediante instrucciones paralelas SIMD-SSE2 dispuestas en forma de funciones intrínsecas dentro de la biblioteca abierta DirectXMath.h de Microsoft. Aunque existen varias formas de proyección (como la ortogonal y la axonométrica), en el motor la única disponible actualmente es la perspectiva. De manera análoga, el programador podrá configurar el ángulo de proyección, la relación de aspecto y los valores para near y far. Los ángulos siempre serán expresados en grados, quedando relegada al motor la tarea de realizar internamente las conversiones pertinentes cuando se necesite operar en radianes.

4.3.2 Carga de texturas

El formato de texturas soportado por IE2 es el DDS (DirectDraw Surface), el cual admite internamente varios algoritmos de compresión de texturas admitidos por hardware (como el DXT). Estos ficheros quedarán embutidos dentro de contenedores IEF para mantener organizadas y concentradas todas las texturas. Microsoft implementa un cargador de texturas DDS en forma de biblioteca sencilla "DDSTextureLoader". El motor cargará desde un contenedor IEF (explicado en la sección 4.6) el bloque DDS en memoria, el cual será automáticamente decodificado y transferido a memoria de vídeo. Como dato adicional, se ha utilizado la herramienta de Microsoft "DirectX Texture Tool" para generar las texturas DDS de las demo-script, pues las originales se encontraban en

formato PNG (Portable Network Graphics). Una vez cargadas, las texturas se filtrarán usando un filtrado anisotrópico simple al momento de dibujarse. Cuando una textura deba extenderse a lo largo de un polígono, el mecanismo de repetición predeterminado será de tipo TEXTURE_ADDRESS_MIRROR [13]. [Fig. 4]

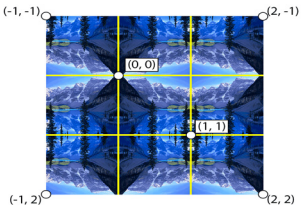


Figura 4 – Textura en espejo (ADDRESS_MIRROR)

4.3.3 Iluminación global y colorización

El programador podrá seleccionar un coeficiente decimal de luz (siendo 0.0 totalmente oscuro y 1.0 iluminado). Cualquier valor mayor a 1.0 saturará el color resultante generando una sobreexposición a la luz. La coloración de la luz y el brillo se calculan en el Pixel Shader de la etapa final de la tubería o pipeline, obteniendo el color de píxel procedente de la textura y multiplicándolo por el color de la luz y factor de intensidad. [14], [15]

4.3.4 Ciclo de ejecución gráfico

Una vez por ciclo de dibujo, el usuario podrá llamar desde el script al método `Clear()` para borrar la imagen generada en el fotograma anterior, redibujar los objetos pertinentes mediante llamadas a `Render()` y terminar presentando el fotograma por pantalla con el método Present(). El método render también recalcula la matriz de traslación, escala y rotación 3D del objeto y la envía a la GPU. A continuación, se envían al sombreador de vértices los conjuntos de vértices e índices (`IASetVertexBuffer / IASetIndexBuffer`) conjuntamente con la topología (que siempre será de triángulos). El motor proseguirá inicializando el sombreador de píxeles, al cual se le enviarán las variables de estado de dibujo con `UpdateSubResources()` y las texturas a aplicar (si las hay, con `PSSetShaderResources()`). [Fig. 5]

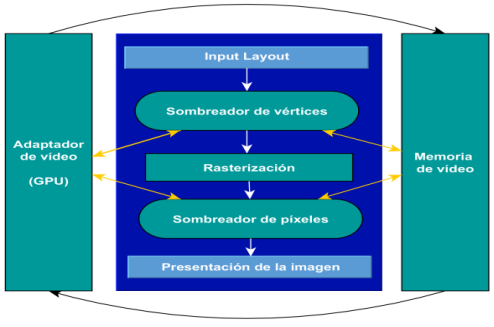


Figura 5 - Tubería gráfica (pipeline)

4.3.5 Carga de modelos 3D

Uno de los objetivos y puntos clave del proyecto era el soporte de renderizado de modelos 3D complejos. Hasta el momento, el motor solo disponía de una clase para dibujar cubos texturizados (que podían escalarse, posicionarse y rotarse). Este cubo representa una primitiva. Aunque es posible dibujar objetos compuestos de varias primitivas mediante código script, el resultado obtenido es lento y ofrece una muy pobre calidad visual. Para dibujar entidades 3D más realistas o detalladas, el desarrollador recurre normalmente a programas de modelado 3D como Blender, Maya, 3D Studio Max o Cinema 4D. Estos permiten generar en 3D y de manera considerablemente más fácil objetos que luego pueden ser representados por aplicaciones gráficas. Los ficheros 3D contendrán información acerca de la ubicación de los vértices 3D del objeto en el espacio y las aristas de conexión entre los mismos, además de otra mucha información adicional. A estas colecciones y agrupaciones de vértices y aristas se las llama "meshes" o "polygon meshes".En el mercado, existe un amplio abanico de formatos 3D. Algunos de los más conocidos son el OBJ de WaveFront [16], el PLY de Stanford [17] (muy utilizado en impresoras 3D), ambos ASCII, el antiguo 3DS de Autodesk 3D Studio Max y el FBX Universal [18] (también de Autodesk). El formato que se eligió finalmente fue el FBX, pues en la industria de los videojuegos se ha convertido en casi un estándar de facto y porque se diseñó para funcionar como fichero de intercambio entre varios programas. Admite agrupaciones de objetos, información de iluminación, cámaras, propiedades de escena y animación, a diferencia de los dos primeros. En su variante binaria, ocupa menos espacio que los formatos ASCII anteriores, y mientras estos listan la información 3D de una manera puramente secuencial, el formato FBX está diseñado para ofrecer lecturas rápidas de ciertas zonas de interés, al estar internamente representado con estructuras en árbol (nodos) [19]. Desde Autodesk, se ofrece una biblioteca FBX gratuita en forma de SDK para C++. Es la biblioteca que se ha utilizado para recuperar los datos de los modelos 3D. Cabe destacar que, como se puede apreciar en la figura [Figura 6], el adaptador del plugin 3D del intérprete FBX del motor está implementado en una clase "FBXModelLoader" que hereda sus funciones de la abstracta "ModelLoader". Esto significa que en líneas de desarrollo futuras se podrían implementar formatos de archivo 3D distintos, como por ejemplo, un hipotético "OBJModelLoader" para visualizar archivos de WaveFront. [Fig. 6]

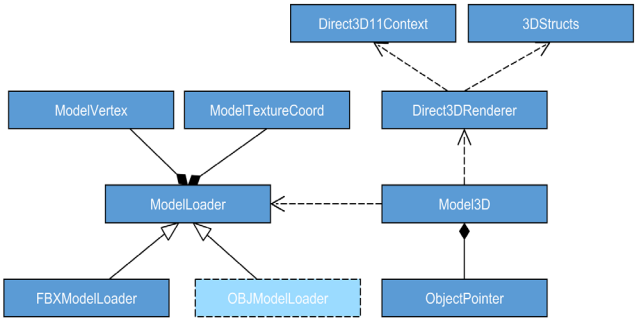


Figura 6 - Estructura de clases del cargador 3D

4.4 Plugin XInput

Como en casi todos los programas, es necesario controlar la entrada de información del teclado o ratón por parte del usuario. En videojuegos encontramos además, dispositivos de E/S alternativos como mandos y joysticks para mejorar la experiencia del jugador. En el sistema operati-

vo Windows se pueden encontrar dos APIs distintas para leer información desde estos dispositivos: DirectInput y XInput. DirectInput es la opción más compatible puesto que fue el primer API y por lo tanto es el más antiguo. Como dato adicional apareció por primera vez en Windows 95 y funciona con mandos, palancas y volantes anteriores a los fabricados en el año 2006. El API moderno de Microsoft (XInput) fue creado inicialmente para la videoconsola XBOX 360 pero fue rápidamente extendido a Windows. La interfaz de este API es mucho más limpia. Soporta dispositivos más nuevos, permitiendo leer información en pocas llamadas. Con XInput se puede obtener información de características específicas del dispositivo, como la presión de los dedos sobre los gatillos de un mando, controlar el vibrador, etc. sin necesidad de recurrir a montones de bibliotecas de terceros. Por último, admite un abanico mucho más amplio de dispositivos de juego (mandos, joysticks, volantes, cambios de marchas, alfombras de baile, micrófonos USB, guitarras de Guitar Hero… [Fig 7]). Todo con un conjunto de llamadas unificadas [8]. Para realizar las pruebas de funcionamiento se disponía de dos mandos de XBOX 360 inalámbricos con su correspondiente adaptador para PC. Para probar la extensión de entrada y salida basada en XInput se realizó un test unitario específico sobre terminal, con el que se detectaron varios errores (vibrador y gatillos de presión.) El sistema desarrollado para el motor permite el control de hasta 4 dispositivos simultáneos (que es el límite actual de XInput). El plugin permite controlar los dos vibradores del mando por separado y obtener información útil adicional como el tipo de energía que se está utilizando (cable directo, pilas, batería…) y el nivel de batería restante (si el dispositivo se está usando con baterías).

Figura 7 - Dispositivos XInput (mando y volante)

4.5 Extensión de audio 3D

La extensión de audio tiene como objetivo la gestión y abstracción del audio del juego que se está ejecutando. Antes de iniciar ninguna implementación o diseño de esta extensión, se llevó a cabo una búsqueda APIs de audio que tuvieran buena documentación, aceptación y reputación entre los programadores de la industria. Las más destacadas fueron DirectSound, XAudio 2 y OpenAL [20]. De las tres opciones, se eligió OpenAL, porque exponía una interfaz de programación nativa en C, razonablemente sencilla y muy similar a OpenGL. DirectSound fue descartada por ser antigua y carecer actualmente de soporte y XAudio 2 porque realizaba lo mismo que OpenAL pero no era multiplataforma. OpenAL es abierta, compatible y con aceleración de audio 3D, lo cual resulta muy interesante para el proyecto. Fue diseñada originalmente por Loki y adquirida poco después por Creative Labs para reproducir sonidos 3D en sistemas operativos basados en Windows Vista o superior. En este momento, existen corriendo cuatro implementaciones distintas de la especificación OpenAL, aunque compatibles entre sí. La original de Loki y Creative Labs (Sample Implementation), la de StrangeSoft, la de Adalin AeonWave y la de Rapture3D, siendo las tres primeras gratuitas y la última comercial cerrada y de pago.

Cabe destacar que en un inicio se empezó a trabajar con la implementación original (Sample implementation), pero pronto se pasó a utilizar la de StrangeSoft debido a que la implementación original de Loki/Creative tenía algunos fallos conocidos que, por falta de mantenimiento, no habían sido resueltos. Algunas webs de fabricantes de juegos recomendaban utilizar la implementación de StrangeSoft cuando la implementación original fallaba. El resultado con la biblioteca de StrangeSoft fue muy positivo pues efectivamente se mantenía actualizada y ofrecía excelente compatibilidad.

La referencia de programación de OpenAL se tomó de la especificación de OpenAL 1.1 disponible en la web oficial de la biblioteca [20]. A continuación se explicarán algunas de las funcionalidades más importantes que hemos implementado en nuestra extensión y su funcionamiento.

4.5.1 Mezcla de audio

La extensión de audio debe ser responsable de mezclar, controlar, reproducir y administrar los múltiples efectos de sonido, voces, ruidos y pistas de música de un juego. Aunque a simple vista pueda parecer que todos los anteriores tipos de sonido tienen un tratamiento idéntico (porque todos son objetos de audio), existen diversas razones por las que deben ser tratados de maneras muy diferentes. Por ejemplo, los efectos de sonido son habitualmente muy cortos (como el chirrido de una puerta, una pisada, etc.) y se necesita muy poca memoria para describirlos en su totalidad. Esto conlleva a que habitualmente residan completos en memoria RAM, permitiendo una reproducción sin esperas, casi instantánea. Pero por otra parte, también tenemos pistas de música ambiental. Una pista de audio puede durar varios minutos. Si consideramos un sampling típico de calidad de CD (44.100 Hz estéreo de 16 bits) el audio necesitará 176,4 KB por cada segundo de audio. Si el flujo de música dura 5 minutos, necesitaríamos mantener aproximadamente 53 MB de datos en la memoria RAM, lo cual puede resultar excesivo.

4.5.2 Técnica de streaming o doble búffer

La técnica de streaming o doble búfer circular supone una solución al problema anterior. Supóngase un fichero de audio de un total de 176 MB (1000 segundos con una tasa de bits redondeada a 176 KB/s). La solución reside en crear en memoria RAM un array con sólo dos bloques o búferes de memoria contiguos (de 176 KB de tamaño cada uno). En memoria sólo se cargan, por tanto, los primeros 352 KB del flujo de audio. A esta etapa se la llama "Buffering". Se procede a iniciar la reproducción del primer bloque. Cuando esta se completa, el sistema de audio (en

este caso OpenAL) avisará, pero no se producirá ningún corte en el flujo audible pues aún queda contiguo un segundo búfer de repuesto con información reproducible. Mientras este segundo búfer de repuesto se consume, el motor de audio tendrá tiempo de sobreescribir el primero, ya agotado, con los siguientes 176 KB de información, que corresponderían a un tercer bloque de datos. Así, y de manera sucesiva, los búferes se irán reciclando cíclicamente hasta que el flujo completo se haya reproducido. Obsérvese que el consumo total de RAM será ahora siempre de 352 KB, independientemente de la longitud del flujo de música original. Mientras que esta técnica solventa el problema de consumo de memoria física, el consumo de espacio en disco podría seguir representando un problema si se tiene en cuenta que 75 minutos de música implican 700 MB de datos. Es por esto que adicionalmente, en InvasionEngine 2 existe una capa más que es posible aplicar a los flujos de audio: la de compresión.

Se evaluaron algunos de los códecs de audio más comunes en el mercado (MP3 y WMA, por poner algunos ejemplos), pero solían tener costes de licenciamiento bastante elevados. Al final, se escogió OGG Vorbis [21] de Xiph.org pues está libre de patentes y es gratuito. Entre sus cualidades destaca soporte de audio multicanal 5.1 y calidad de decodificación bastante superior a MP3. Este formato es ampliamente utilizado en bandas sonoras de videojuegos. El usuario del motor podrá elegir entre el ya implementado LPCM (Linear PCM sin compresión) con cabecera WAV de Microsoft y el nuevo OGG Vorbis (con compresión). Un códec de audio con soporte para Streaming heredará de una clase llamada "MediaStreamer" de manera similar a como ocurría con los modelos 3D. Siguiendo esta directriz, se podrían añadir nuevos códecs en el motor sin realizar cambios en la arquitectura del sistema. En la [Fig. 8] se muestra la estructura de clases.

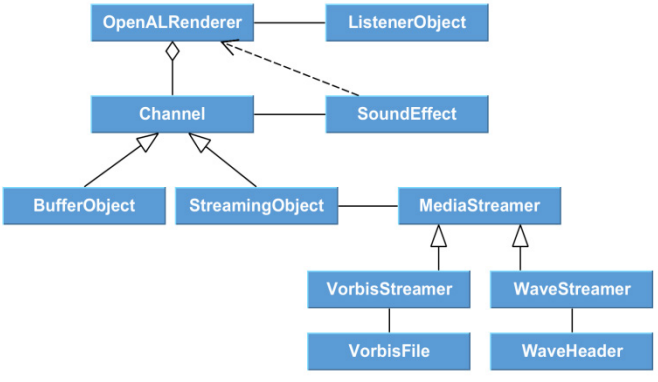


Figura 8 - Estructura de clases del administrador de audio

4.5.3 Sonidos puntuales

Además de música y efectos sonoros, a menudo el programador tendrá que trabajar con efectos sonoros de reproducción esporádica como por ejemplo las voces de diálogo de personajes de juego. Las líneas de texto son lo suficientemente cortas como para poder mantenerse íntegras en memoria, pero no tiene sentido almacenarlas ahí permanentemente pues se utilizan en ocasiones muy concretas y a menudo ni se repiten a lo largo del juego. Un sistema de streaming aquí no tendría sentido, pero probablemente mantener estos sonidos precargados tampoco. El método de actuación será cargar la voz íntegramente en memoria cuando se requiera y desecharla una vez la reproducción haya finalizado.

4.5.4 Estrategia de canales de audio

La extensión de administración de audio que se ha programado basa su estrategia en un patrón resource-pool de canales. Estos canales residen en una clase "OpenALRenderer", la cual contiene internamente una lista variable de "pistas" que pueden albergar un sonido mezclable y reproducible en cualquier momento. Cuando un canal ya no haga falta, deberá ser liberado. En el momento de solicitar a OpenALRenderer un nuevo canal, se verificará si es posible reciclar algún canal que se haya liberado con anterioridad. Si todos están ocupados, se procederá a añadir uno nuevo. Aunque no hay un número máximo de canales en dicha lista (se pueden añadir tantos canales como se crea oportuno), sí existe un valor máximo de polifonía. La polifonía máxima indica el número máximo de canales que se pueden mezclar y reproducir a la vez. Esto dependerá en gran medida del hardware disponible en la máquina. Mientras un chip de sonido integrado normal (como el típico Realtek HD de la mayoría de placas base de hoy día) admite una mezcla máxima por hardware de 8 canales, una tarjeta de sonido dedicada como la Sound Blaster X-FI de Creative admitirá hasta 128 canales o voces simultáneas. Si el motor de audio se encuentra con la necesidad de reproducir más canales de los permitidos por el hardware (por ejemplo 10 sobre una Realtek de 8) mezclará los canales restantes por software. La polifonía máxima que se puede conseguir es de 32 canales simultáneos (extendidos mediante mezcla por software en CPU). En el caso especial de que el hardware dedicado admita más de 32 canales (como el caso anterior de la tarjeta X-FI de 128), la polifonía máxima será la soportada por el hardware. En este caso, de 128 voces, quedando desactivada la mezcla software por exceder los 32 máximos. En cualquier caso, una vez superada la polifonía máxima, los nuevos sonidos no se reproducirán y quedarán en silencio hasta que exista "espacio de mezcla" en el pool.

4.5.5 Aplicar efectos

Para aplicar un efecto a un sonido o flujo de música, se deberá obtener el canal de mezcla asociado. Esto se consigue en nuestro motor mediante una función `GetChannel()` y a continuación se le aplicará el efecto deseado a ese ID (posición 3D, volumen, decay, tono, velocidad, etc). Las operaciones que impliquen cálculos 3D (como la posición u orientación del sonido) siempre deberán hacerse sobre fuentes de sonido mono. Intentar aplicar un efecto 3D a una fuente de sonido estéreo o surround no provocará error, pero tampoco causará ningún efecto. Es importante destacar que ésta es una limitación propia de OpenAL y no de nuestro plugin de audio del motor. Esta limitación no afecta a cambios de volumen o tono, que si pueden ser aplicados a fuentes multicanal como estéreo o surround.

4.5.6 Concepto de micrófono 3D.

En gráficos 3D se habla comúnmente de concepto de cámara, la cual puede estar en una posición, con una orientación y up-vector. El micrófono 3D de InvasionEngine 2 tiene las mismas propiedades que una cámara de gráficos 3D, pero en este caso, aplicadas al sonido. El programador de scripts obtendrá un resultado sonoro 3D realista si aplica los mismos valores de la cámara gráfica al micrófono sonoro. Si por ejemplo este micrófono se aleja de las fuentes emisoras de sonido en el juego, el volumen y claridad del sonido irán decayendo. Sólo puede haber un micrófono en la escena ya que, desde un punto de vista lógico, es imposible estar en dos sitios a la vez.

4.6 Contenedor multimedia IEF

Normalmente, un videojuego o programa multimedia contiene infinidad de archivos auxiliares (texturas, mapas, sonidos, músicas, datos de modelos 3D…) Si en el directorio del juego de destino se colocan tal cantidad de ficheros, el rendimiento del sistema operativo en tiempos de acceso y búsqueda sobre la ruta de trayecto del programa podría resentirse considerablemente. Además, no resulta nada práctico ensuciar el directorio de instalación de la aplicación con miles y miles de ficheros. Mientras los archivos pequeños se leen de uno en uno causando una gran sobrecarga en el sistema operativo, un fichero grande podría leerse a ráfagas largas, en poco tiempo y usando pocos recursos. Por esta razón hemos desarrollado el '*InvasionEngine File*'. IEF es un fichero contenedor de datos sencillo y sin compresión, el cual contiene una lista de ficheros embutidos y punteros de dirección absolutos dentro del propio fichero dónde se almacenan. Un acercamiento similar podría ser un fichero TAR o ISO, los cuales pueden simular en cierta medida una especie de disco virtual compacto. Para poder generar estos ficheros, también hemos desarrollado una utilidad empaquetadora simple en línea de comandos con lenguaje C. La contraparte desempaquetadora está escrita en C++ directamente como una extensión más del motor, la cual podrá acceder a los datos de estos paquetes de manera transparente al programador devolviendo los datos cargados directamente en memoria RAM. Adicionalmente, la mencionada extensión cacheará de manera automática recursos en memoria para evitar accesos a disco de manera continuada cuando se le pida repetidamente un mismo fichero. Esta función de caché puede ser controlada manualmente o desactivada, siempre a criterio del programador.

5 Test y Scripts

Como se comentó anteriormente, el test de aplicación consiste en la realización de pequeños scripts a modo de referencia que comprueben el correcto funcionamiento de las funciones implementadas y expuestas por el motor. Esto es posible debido a que los scripts pueden ejecutar código normalmente y acceder a las funciones externas expuestas desde las extensiones. A todos los efectos, pueden considerarse pruebas de caja negra. A continuación podrán observarse tres scripts de demostración. En la primera [Fig. 9] se muestra una escena 3D construida a partir de primitivas (paredes, suelo y techo) y modelos 3D. En la segunda [Fig. 10], un laberinto 3D generado a partir de un TXT, el cual marcaba con '*' los lugares en los que debían dibujarse paredes y finalmente, en la tercera [Fig. 11] una escena 3D con una luz ambiental de color.

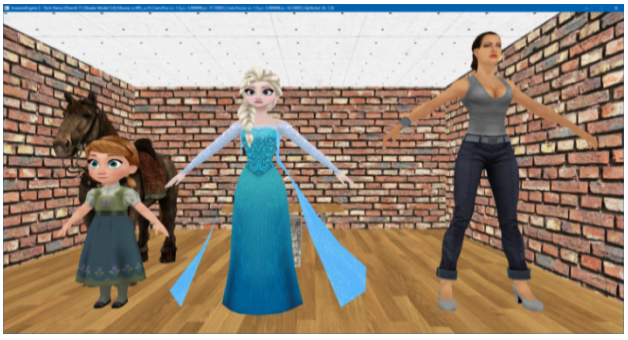


Figura 9 - Test de representación de modelos 3D
Los modelos de "Frozen" Anna y Elsa son propiedad de The Walt Disney Company".

Figura 10 - Laberinto 3D generado a partir de un TXT

Figura 11 - Escena con luz de colorización

6 Resultados

El objetivo de las pruebas que se mostrarán a continuación era mostrar el rendimiento sin carga de los tres motores anteriores (Unity, Unreal Engine 4 e InvasionEngine 2), todos ellos en su versión de 32 bits. Se realizó el mismo test-demo (un cubo 3D con textura sólida y música de fondo) con las últimas versiones disponibles de los tres sistemas. La configuración de audio de salida estaba en estéreo y el viewport gráfico generado tenía una salida 1080p @ 60 Hz (1920x1080) con sincronización vertical activada. Para monitorizar las métricas se utilizó la herramienta de perfilado de rendimiento que trae Visual Studio consigo. Para obtener más información acerca Para

más información sobre Unreal Engine® 4 [4] y Unity® [5] consultar sus respectivas webs oficiales. Los resultados obtenidos pueden verse a continuación [Tabla 1] [Fig. 13].

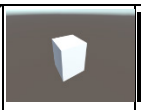
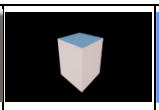
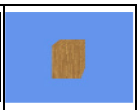
ESPECIFICACIONES / MOTOR		 Unity 5.6.1f1 (32 bits)		 Unreal Engine 4.1.12 (32 bits)		 Invasion Engine 2 (32 bits)	
CPU	Mínimo	0%	CPU	0%	CPU	0%	CPU
	Normal	1%	CPU	5%	CPU	1%	CPU
	Máximo	2%	CPU	12%	CPU	2%	CPU
RAM	Mínimo	44.280	KB	107.992	KB	27.272	KB
	Normal	52.112	KB	107.992	KB	27.276	KB
	Máximo	52.288	KB	517.884	KB	27.276	KB
GPU	Mínimo	30%	GPU	12%	GPU	15%	GPU
	Media	43%	GPU	75%	GPU	16%	GPU
	Máximo	60%	GPU	99%	GPU	19%	GPU
Resolución	Valor	1920 x 1080		1920 x 1080		1920 x 1080	
FPS	Mínimo	60	FPS	60	FPS	60	FPS
	Normal	60	FPS	60	FPS	60	FPS
	Máximo	60	FPS	60	FPS	60	FPS
Disco	Valor	29.230	KB	206.427	KB	69.457	KB

Tabla 1 – Métricas comparativas entre motores

Los tres motores obtuvieron los 60 FPS que se esperaban. A nivel de rendimiento de CPU y uso de memoria, Unity e InvasionEngine 2 se encontraban muy igualados. Esto podría deberse a que ambos se encuentran programados a bajo nivel usando C++ y utilizan un motor de scripting basado en Mono. A nivel de uso de GPU, InvasionEngine 2 mantiene los porcentajes de uso más bajos. No obstante, hay que recordar que a diferencia de Unity o Unreal, InvasionEngine 2 no dispone de programas Shader complejos (como algoritmos de iluminación avanzada), lo cual explicaría las enormes discrepancias en porcentajes de uso con los otros primeros. Por otro lado, se puede observar que Unreal Engine 4 mantiene unos porcentajes de uso de GPU increíblemente altos para dibujar únicamente un cubo (75%), saturando por completo el pipeline gráfico a 99% en determinados momentos. Para conocer la razón de este comportamiento se debería de inspeccionar el funcionamiento interno de dicho motor, aunque una posible razón podría ser que intente generar la mejor imagen posible con el tiempo de GPU disponible (que variaría en función del adaptador gráfico y el ancho de banda disponible en el hardware), aunque eso implique repetir reiteradamente ciertas funciones de cálculo intensivo. En términos de consumo de memoria RAM, InvasionEngine 2 también demuestra ser el más eficiente seguido de Unity, que obtiene un segundo puesto. Aquí Unreal Engine 4 pierde por completo doblando a Unity y triplicando a InvasionEngine 2. En algunos momentos, quizás por tareas de gestión interna, genera picos de uso de memoria RAM que alcanzan los 512 MB. Este consumo de memoria tan alto podría tener origen en los complejos algoritmos de representación 3D de los que dispone y que dotan de increíble realismo gráfico a los juegos hechos con el mismo. Tal vez, y dado que la escena 3D es asombrosamente simple (cubo 3D) la razón de este consumo sea en una precarga en memoria de todos los efectos que es capaz de producir, independientemente de si se usan o no. En caso de que fuera así, no sería un motor no indicado para juegos sencillos o máquinas con bajos recursos, lo cual favorecería ampliamente a InvasionEngine 2 o Unity. Se ha comprobado el espacio en disco que ocupaban las tres versiones de la demo una vez compiladas y empaquetadas. Aquí Unity demostró ser el más compacto de los tres, con una carpeta de binarios de aproximadamente 30 MB. El segundo puesto lo obtuvo InvasionEngine 2 con 70 MB, quedando último Unreal Engine 4 con 206 MB. La cantidad de espacio ocupada en disco por Unreal Engine 4 podría apoyar la hipótesis anterior de que este motor empaqueta efectos y características aunque no se usen. Por otra parte, la diferencia de espacio en disco utilizado entre InvasionEngine 2 y Unity podría radicar en la versión del sistema de scripting Mono utilizado. Mientras InvasionEngine 2 es capaz de interpretar scripts basados en .NET 4.0 completo, Unity está limitado a una versión recortada de .NET 2.0. Para concluir, se podría dar por bueno casi un empate técnico entre InvasionEngine 2 y Unity a pesar de las carencias del primero si se tiene en cuenta que las funcionalidades del segundo son superiores a las del primero debido a las restricciones de tiempo con las que disponía el proyecto. Es conveniente señalar que, por otra parte, Unreal Engine 4 no obtiene una buena puntuación cuando se trata de hacer juegos sencillos. Este motor quedaría reservado para realizar juegos con grandes gráficos y alta complejidad técnica; escenario en el cual, Unity e InvasionEngine 2 probablemente quedarían fuera de lugar.

7 CONCLUSIONES

Se ha implementado un motor de juegos primitivo con capacidad de scripting para satisfacer una inquietud personal, profundizando en el funcionamiento base de uno y comparando su rendimiento con los motores de Unreal Engine 4 y Unity. Para ello se ha fragmentado el objetivo en varios objetivos menores que han sido asumidos gradualmente. Los más críticos a implementar se enuncian a continuación:

- **El motor de scripting:** se basa en un entorno Mono manipulado en tiempo de ejecución con las funciones a bajo nivel del motor expuestas.
- **Gráficos 3D:** La realización de un módulo de representación de escenas 3D, a partir de modelos FBX texturizados multicapa programado en DirectX 11 mediante Shaders HLSL.
- **Audio 3D:** El soporte para la reproducción de efectos sonoros 3D y decodificación de música OGG Vorbis basada en la biblioteca abierta de OpenAL.
- **Cuarto**: El control de dispositivos de entrada (mandos, palancas, etc) con XInput.

Tras testear InvasionEngine 2 contra otros dos competi-

dores (Unreal Engine 4 y Unity) los resultados se han mostrado muy favorables para el primero. Se ha conseguido un motor liviano y con rendimiento aceptable en escenarios sencillos, por lo que el propósito se ha cumplido. Nuevamente, es importante destacar que resulta difícil hacer una comparación debido a que a nivel de prestaciones no se pueden igualar por falta de prestaciones.

Durante la implementación del motor cabe destacar que surgieron dos problemas externos importantes: el primero era un error que afectaba al compilador de C++ de Visual Studio 2015. En determinadas circunstancias, el sistema operativo Windows mataba el proceso del compilador debido a un error interno del mismo. Este fallo era conocido y estaba documentado en la web oficial de Microsoft y se solucionaba instalando la actualización 1 de Visual Studio 2015. El segundo fallo importante que apareció fue un error de segmentación de memoria debido a que el descodificador de OGG Vorbis requería el uso de estructuras C alineadas en direcciones de memoria múltiples de 8 bytes, pero en la documentación de Ogg Vorbis no se mencionaba nada, de manera que al compilar en modo RELEASE en Visual Studio fallaba inexplicablemente en algunas ocasiones, puesto que la alineación automática no siempre era múltiple de 8.

Dar por concluido un motor de juego también es algo complicado pues debemos estar abiertos a continuas mejoras y nuevas funcionalidades. El hardware es cada vez más potente y los usuarios más exigentes. El motor implementado es primitivo comparado con los motores existentes debido a restricciones temporales del proyecto. Quedarían por implementar un abanico de funcionalidades para poder competir en cuanto a prestaciones con otros. Las siguientes funcionalidades que se deberían incorporar en un futuro serían:

El soporte completo de iluminación (incluyendo luz emisiva, difusa y especular), un entorno de desarrollo propio con editor de código script y mapas/escenas 3D, otro plugin 3D para sistemas operativos Linux y Android basado en OpenGL, la inclusión de una pila de red para la comunicación multijugador, un mini sistema de físicas, control de colisiones y un plugin alternativo a XInput para sistemas operativos no Windows.

AGRADECIMIENTOS

A mi compañero Albert Castell por aceptar mi reto de enfrentarse a un motor nuevo y a las tecnologías Kinect.

A mi tutor Enric Martí por aceptar mi TFG, proporcionarme distintos puntos de vista durante su realización y prestarme los equipos Kinect necesarios para llevar a cabo algunas partes.

A Estefanía Riera, por darme pautas para afrontar los retos que supone un TFG de estas características.

A Yolanda Benítez, por aconsejarme en la realización del dossier específico de software.

A todos mis amigos de «Primera fila» por acompañarme a lo largo de esta difícil carrera y darme ánimos para superar este TFG.

Y finalmente a mi familia, por entender la dificultad del proyecto que tengo entre manos y tomarse con buen humor mis jornadas de estudio y trabajo maratonianas.

BIBLIOGRAFÍA Y REFERENCIAS

[1] Wikipedia. "Pong" Disponible en: https://es.wikipedia.org/wiki/Pong (último acceso Junio de 2017)

[2] Eva Hudlicka. "Affective Game Engines: Motivation and Requirements" 4th International Conference on the Foundations of Digital Games (2009) Disponible en: http://works.bepress.com/eva_hudlicka/2/ (último acceso: Marzo de 2017)

[3] Microsoft. "Kinect" Disponible en: http://www.xbox.com/es-ES/kinect (último acceso: Junio de 2017)

[4] Unreal Engine 4.15. "Making Something Unreal" Disponible en: https://www.unrealengine.com/what-is-unreal-engine-4 (último acceso: Mayo de 2017)

[5] Unity3D. "Aprender con Unity" Disponible en: https://unity3d.com/es/learn (último acceso: Mayo de 2017)

[6] Microsoft, "The Direct3D reference library" Disponible en: https://msdn.microsoft.com/en-us/library/windows/desktop/ff476080(v=vs.85).aspx (último acceso: Marzo de 2017)

[7] "Proyecto Invasión (LIS) y página oficial" Disponible en: http://invasion.uab.es/index.php/ie (último acceso: Marzo de 2017).

[8] Microsoft Developer Network. "XInput Programming Reference" Disponible en: https://msdn.microsoft.com/es-es/library/windows/desktop/ee417005(v=vs.85).aspx (último acceso: Mayo de 2017)

[9] The Mono Project. Documentación oficial de scripting con Mono. Disponible en: https://www.mono-project.com/docs/advanced/embedding/scripting/ (último acceso: Marzo de 2017).

[10] Seth Juarez, Andreia Gaita. "Scripting with Mono Embedded API." Descripción: Introducción al scripting con Mono y demostración del mismo con el motor gráfico Unity. Material visual en vídeo disponible en: https://channel9.msdn.com/Events/dotnetConf/2015/Scripting-with-MonosEmbedded-API (último acceso: Marzo de 2017)

[11] Taking Initiative. "DirectX 10: HLSL and Basic Shader Reflection" Disponible en: https://takinginitiative.wordpress.com/category/graphics-programming/hlsl/ (último acceso: Mayo de 2017)

[12] Microsoft. "Windows Advanced Rasterization Platform". https://msdn.microsoft.com/en-us/library/windows/desktop/gg615082(v=vs.85).aspx

[13] Wikipedia. "Mapeado de texturas" Disponible en: https://es.wikipedia.org/wiki/Mapeado_de_texturas (último acceso: Mayo de 2017)

[14] 3D Game Engine Programming. "Texturing & Lighting in DirectX 11" Disponible en: https://www.3dgep.com/texturing-lighting-directx-11/ (último acceso: Mayo de 2017)

[15] RasterTek (DirectX 11 tutorial). "Ambient Lighting." Disponible en: http://www.rastertek.com/dx11tut09.html (último acceso: Mayo de 2017)

[16] Wikipedia. "Wavefront .obj file." Disponible en: https://en.wikipedia.org/wiki/Wavefront_.obj_file (último acceso: Mayo de 2017)

[17] Paul Bourke. "PLY – Polygon File Format" Disponible en: http://paulbourke.net/dataformats/ply/ (último acceso: Mayo de 2017)

[18] Autodesk. "Adaptable file format for 3D animation software" Disponible en: https://www.autodesk.com/products/fbx/overview (último acceso: Mayo de 2017)

[19] Autodesk FBX SDK Help. "Importing a scene and initializing the importer" Disponible en: http://help.autodesk.com/view/FBX/2015/ENU/ (último acceso: Mayo de 2017)

[20] Creative Labs. "OpenAL Programmers Guide (PDF)" https://openal.org/documentation/ (último acceso: Marzo de 2017).

[21] Xiph.org. "OGG Vorbis" Disponible en http://vorbis.com/ (último acceso: Mayo de 2017)

APÉNDICE

A1. DIAGRAMA DE FLUJO DE FUNCIONAMIENTO DEL MOTOR

La siguiente figura muestra un diagrama de flujo del funcionamiento del motor. Puede observarse la secuencia de arranque del motor, la carga en memoria de un script y posterior inicialización de plugins. Cuando la secuencia de arranque termina, comienza el bucle de ejecución del script. Las llamadas desde el mismo que se efectuen al motor normalmente son síncronas, pero el usuario puede acceder a varios plugins y/o ejecutar múltiples códigos concurrentemente meditante el uso de hilos o threads. Para obtener información ampliada, consultar los apartados 4.1 y 4.2.

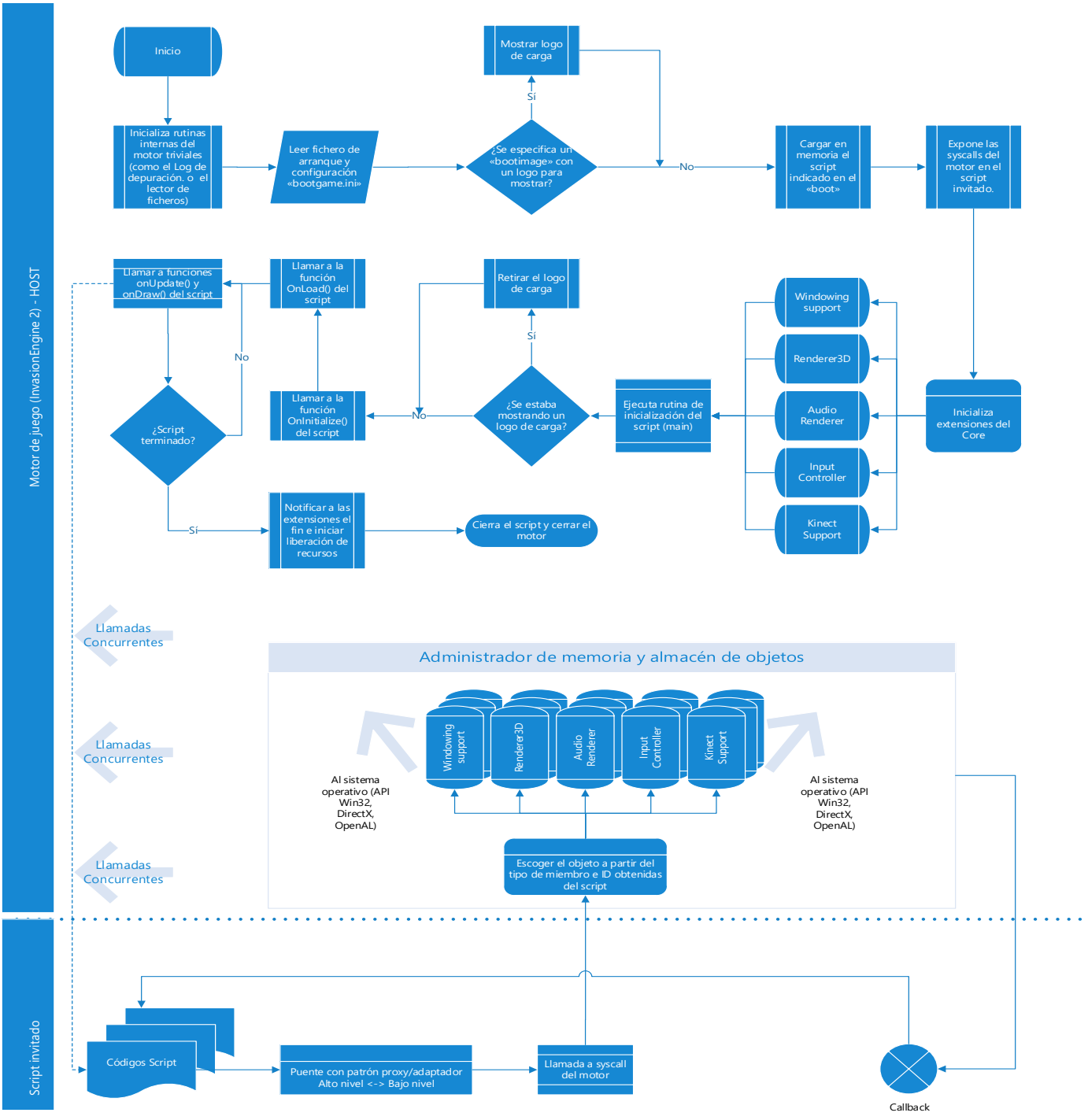


Figura 12 – Diagrama de flujo del motor

A2. CUADRO COMPARATIVO DE FUNCIONALIDADES ENTRE MOTORES GRÁFICOS

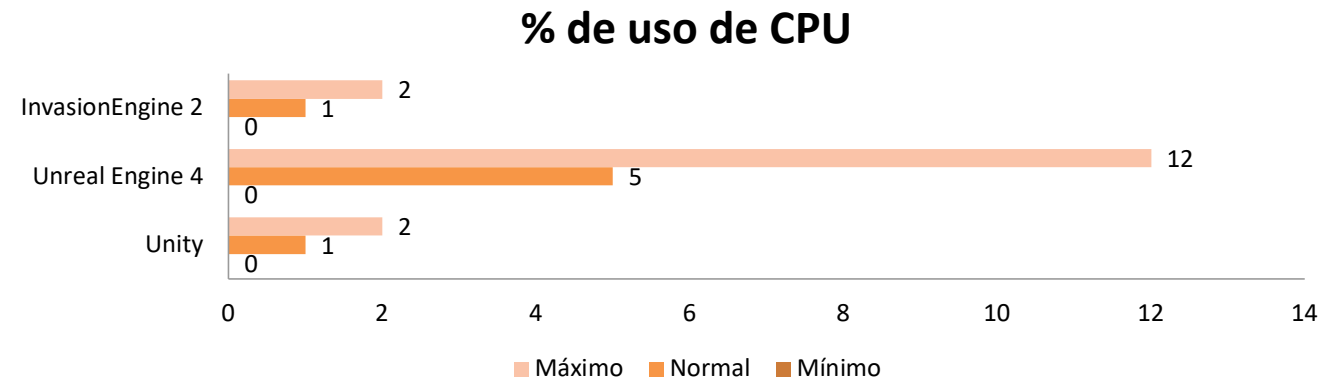
A continuación se muestra un cuadro comparativo de algunos de los motores comentados en el apartado de estado del arte (4.1).

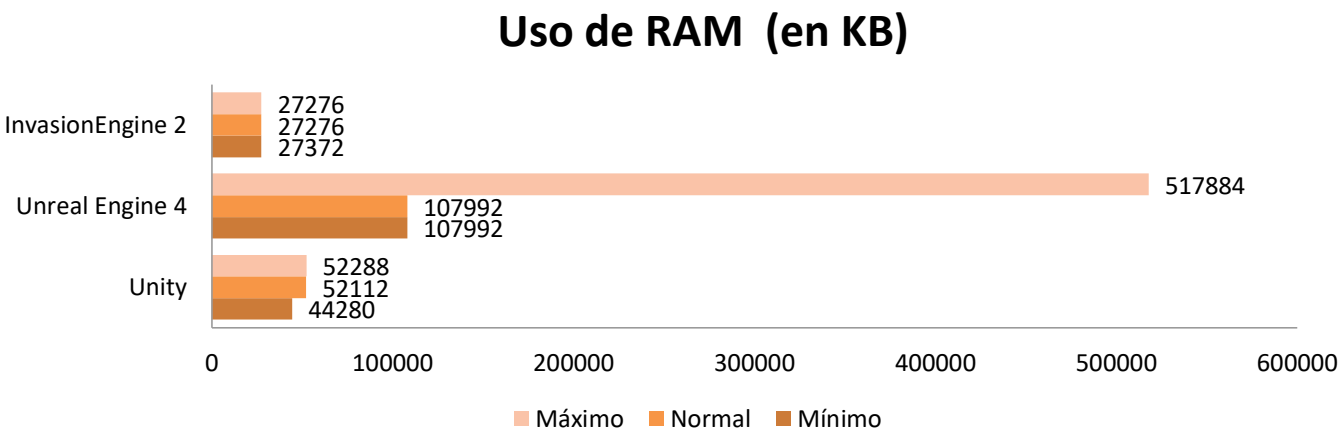
MOTOR / CARACT.	CryEngine (Crytek)	UnrealEngine (Epic Games)	Source (Valve)	Unity Tech.	InvasionEngine2
2D	Sí	Sí	No	Sí	Sí
3D	DirectX 9-12	DirectX 9-12	DirectX 9	DirectX 9-11	DirectX 9-11
	OpenGL	OpenGL	OpenGL	OpenGL	-
Audio 3D	Sí	Sí (OpenAL)	Sí	Sí	Sí (OpenAL)
Scripting	LUA	UnrealScript C++	LUA	C# (Mono)	C#, Basic, Python (Mono)
Multiplataforma	Sí	Sí	Sí	Sí	No[2]
Editor interactivo WYSIWYG	Sandbox Editor	Unreal Editor	Hammer	Unity Editor	No
Soporte Kinect	Sí	Sí	No	Sí	Sí

Tabla 2 – Tabla comparativa entre funciones del motor

A3. GRÁFICOS DE RENDIMIENTO ENTRE MOTORES (RESULTADOS)

Comparativa de InvasionEngine 2 (en índices de rendimiento) con los motores Unreal Engine 4.12 y Unity 5.6. Este gráfico es una versión ampliada de la tabla 1 del apartado de resultados (6).
Valores más bajos indican mejores puntuaciones.





A3. GRÁFICOS DE RENDIMIENTO ENTRE MOTORES (CONTINUACIÓN)

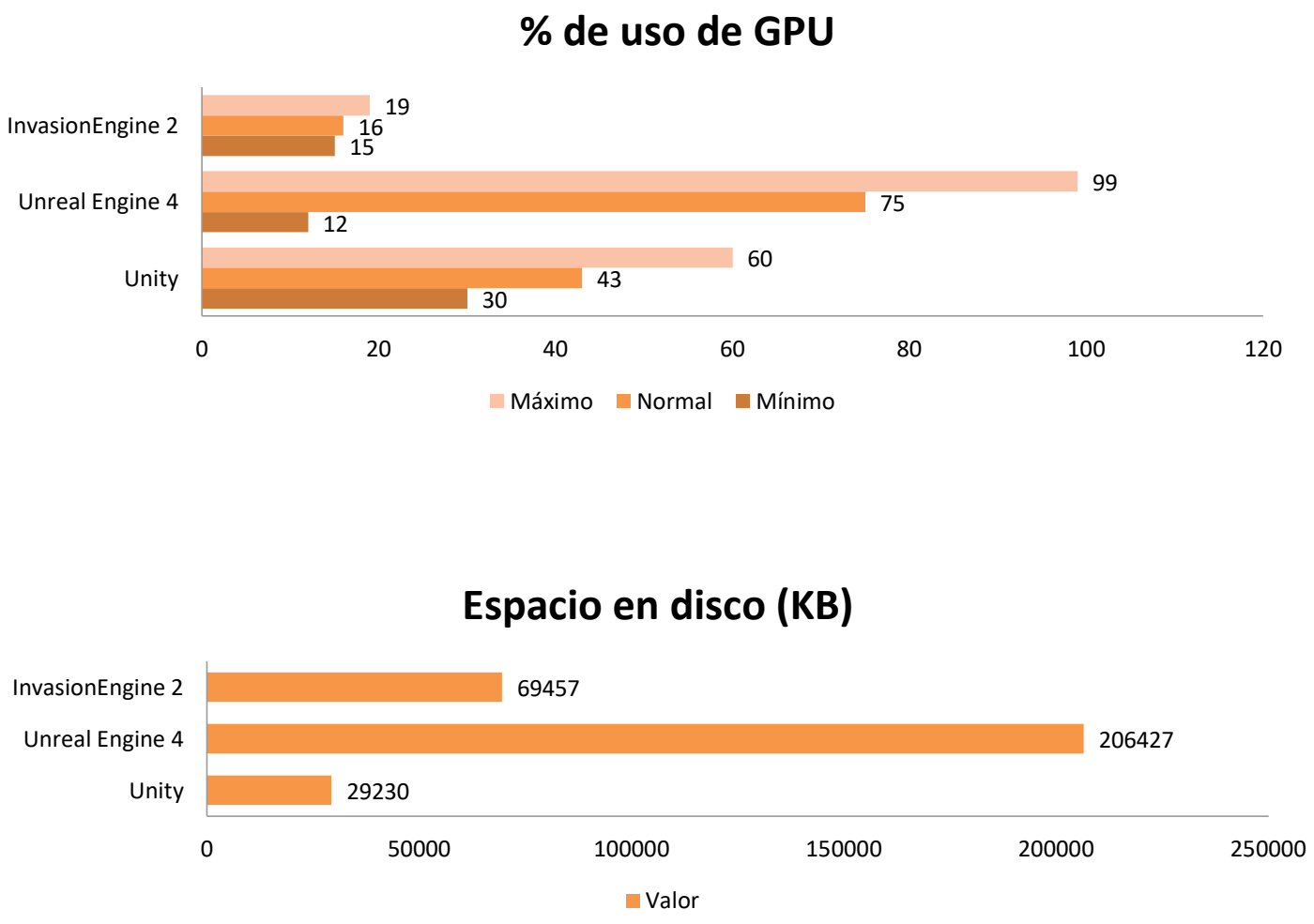


Figura 13 – Uso de recursos del motor