UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Integración de OpenSceneGraph en UPV Game Kernel

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Miguel Medina Patón

Tutor: Ramón Mollá Vayá

Curso 2017-2018

Resum

Este treball busca integrar la llibrería de gràfics *OpenSceneGraph* en el motor de videojocs *UPV Game Kernel* de manera transparent amb la finalitat de afegir nova funcionalitat, millorar la ja existent i facilitar el canvi a una llibrería de gràfics distinta en el futur. El correcte funcionament de la integració s'ha comprovat en un videojoc que ja utilitza el *UPV Game Kernel*: *Space Invaders*.

Paraules clau: OSG, OpenSceneGraph, videojocs, UPV Game Kernel, UGK, Space Invaders

Resumen

En este trabajo se busca integrar la librería de gráficos *OpenSceneGraph* en el motor de videojuegos *UPV Game Kernel* de forma transparente con el fin de añadir nuevas funcionalidades, mejorar las ya existentes y facilitar el cambio a una librería de gráficos distinta en el futuro. El correcto funcionamiento de la integración se ha probado en un videojuego que ya usa el *UPV Game Kernel*: *Space Invaders*.

Palabras clave: OSG, OpenSceneGraph, videojuegos, UPV Game Kernel, UGK, Space Invaders

Abstract

This project inteds to transparently integrate the *OpenSceneGraph* graphics library into the *UPV Game Kernel* videogame engine in order to add new functionalities and improve the already existing while facilitating the switch to a different graphics library in the future. The correctness of the integration has been tested on a videogame that already uses the *UPV Game Kernel*: *Space Invaders*.

Key words: OSG, OpenSceneGraph, videogames, UPV Game Kernel, UGK, Space Invaders

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

El motor de videojuegos *UPV Game Kernel* o *UGK* se creó en 2008 como material para la asignatura optativa Animación por Computador y Videojuegos (ACV) en el máster de Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital (MIARFID) de la ETSINF. Éste está basado en el código de *Space Invaders OpenGL*, un videojuego de código abierto que emula al clásico con el mismo nombre. De este videojuego se extrajo la estructura básica del motor y después se adaptó el videojuego para que usara éste motor. A lo largo de los años se han ido mejorando funcionalidades y añadiendo nuevas al motor como fábricas de objetos, gestión de memoria e inicialización y carga de niveles codificados en archivos HTML. Sin embargo en varios aspectos sigue en el mismo estado que el videojuego original, concretamente en el apartado gráfico. La finalidad de este proyecto es mejorar esta parte del motor integrando una librería de gráficos. Ésto también mejorará el rendimiento del motor y el uso de memoria de éste.

1.1 Motivación

El tema de este trabajo fue escogido debido a la intención del autor de trabajar en el área de los gráficos por computador y en especial los videojuegos. Este proyecto ha sido una excelente oportunidad para eso mismo, además de poder mejorar el motor de videojuegos propio de la UPV.

1.2 Objetivos

El objetivo principal de este trabajo es integrar la librería de gráficos *OpenSceneGraph* en el motor de videojuegos *UPV Game Kernel*, creando una interfaz que sirva de intermediaria entre las dos para poder acceder a las funciones de OSG de forma transparente y facilitando así el cambio a una librería de gráficos distinta en el futuro o la comparación de varias en igualdad de condiciones.
Otros objetivos son refactorizar el videojuego *Space Invaders* para adaptarlo a las modificaciones de *UGK*, aprender a usar la estructura de datos del grafo de escena y ganar experiencia programando en *C++*.
El éxito del proyecto se medirá en lo que la nueva versión de *Space Invaders* usando la nueva implementación del grafo de escena se aproxime a la versión anterior. Aquí sigue una lista de objetivos concretos a cumplir:

- El videojuego no necesita realizar ninguna llamada a *OpenGL* ni *OSG* en el proceso de dibujado de la escena.
- Las mallas de los personajes se muestran en la escena tal y como indica el videojuego.
- Los modelos se mueven como les indica el videojuego.
- La cámara de osg se coloca en la posición indicada por el videojuego con los parámetros correctos.
- Se han podido usar luces definidas como un elemento del grafo de escena.
- Se ha conseguido estandarizar los personajes no estándar y mostrarlos en osg (esto se explica en la sección de Análisis del problema).

1.3 Metodología

1.3.1. Sistema operativo

En este aspecto no hay elección posible. Aunque *OpenSceneGraph* es compatible con la mayoría de sistemas operativos *UPVGameKernel* y *SpaceInvaders* son sólo compatibles con Windows en su versión actual. Está previsto añadir compatibilidad con otras plataformas en versiones futuras.

1.3.2. Control de versiones

UGK está alojado en un servidor de *SubVersion (SVN)* en la *ETSINF* por lo que, aunque *Git* es una herramienta mejor para el control de versiones, se ha seguido usando *SVN*. Sin embargo está previsto migrar el proyecto a *Git* en un futuro próximo. La inclusión de la funcionalidad de este TFG libera de uno de los escollos que dificultan esta migración.

1.3.3. Entorno de desarrollo

Hay dos tipos de entornos de desarrollo disponibles: programas dedicados como *VisualStudio* o *Eclipse* y herramientas separadas. En el segundo grupo se encuentran editores de texto como *VisualStudio Code* y *Notepad++*, compiladores como *MinGW* y *g++*, y herramientas de depuración como *PDB* o *MinGW*.

Debido a la complejidad añadida de aprender a usar múltiples programas al usar herramientas separadas, se ha decidido usar un entorno de desarrollo completo. De entre las opciones disponibles se ha elegido *VisualStudio* debido a que *UGK* y *SI* han sido desarrollados en éste, lo que evita el coste de migrar los proyectos a un entorno distinto.

1.3.4. Generación de código

La librería de *OSG* utiliza scripts de *CMake* para gestionar la generación de código independientemente de la plataforma. Debido a esto hay dos opciones disponibles para

generar el proyecto de VisualStudio para compilar *OSG*: Utilizar el programa *CMake* o la integración de *CMake* de *VisualStudio*. Debido a que las instrucciones en la página del proyecto de *OSG* sólo explican cómo usar la primera, ésta es la opción que se ha escogido. En el apéndice B se explica con detalle los pasos a seguir para usar *CMake* y *VisualStudio* para compilar la librería de *OpenSceneGraph*.

1.3.5. Librería de gráficos

Un objetivo a largo plazo de *UGK* es tener disponible múltiples librerías de gráficos entre las que poder elegir sin necesidad de modificar el código. Este proyecto es el primer paso hacia ese objetivo por lo que cualquiera de las librerías candidatas (*OSG, OGRE, TORQUE* o *IRRLICHT*) era una buena elección. En la sección de Estado del arte se desarrolla las características de éstas.

1.4 Estructura de la memoria

El resto de ésta memoria está dividida en las siguientes seis secciones y tres apéndices:

Estado del arte Dónde se muestran las APIs gráficas y librerías de gráficos más populares para ser comparadas entre ellas y se resume el funcionamiento del grafo de escena de la librería de gráficos *OSG* con una descripción de las funciones que ofrece.

Análisis del problema En esta sección se explica la estructura básica del motor de videojuegos desarrollado en la UPV, el *UGK* y se describen los problemas que han llevado a la realización de este proyecto.

Diseño de la solución Dónde se muestran los cambios y adiciones planeados para la integración de *OSG* en el grafo de escena de *UGK* y se describe el plan de trabajo desarrollado para llevarlo a cabo.

Resultados Aquí se enumeran los objetivos de los especificados en esta memoria que se han cumplido con éxito. También se revisa el plan de trabajo que se mostró en la sección de diseño de la solución.

Conclusiones En esta sección se cierra el proyecto con un comentario sobre los resultados obtenidos.

Propuestas para trabajos futuros Dónde se explican las diversas vías que se pueden tomar para expandir sobre este proyecto.

Apéndice A: Historia de UPVGameKernel Texto que explica el origen del motor de videojuegos de la UPV.

Apéndice B: Instalación de OpenSceneGraph Dónde se detallan los pasos a seguir para compilar e instalar la librería de *OpenSceneGraph* usando *VisualStudio* y *CMake*.

Apéndice C: Diccionario Aquí se definen conceptos y términos que pueden resultar ajenos sin conocimientos de programación de videojuegos o informática gráfica.

1.5 Convenciones

- Las menciones a clases y fragmentos de código se muestran en la fuente `courier`
- Las siglas, palabras en inglés y anglicismos están escritas en *cursiva*.
- Los términos que aparecen en el apéndice C: Diccionario contienen un enlace a éste cuando aparecen por primera vez en el documento.

CAPÍTULO 2
Estado del arte

2.1 Interfaces gráficas

Con el fin de facilitar la creación de aplicaciones que usen gráficos por ordenador se han creado intefaces que permiten acceder al *hardware* de gráficos de forma abstracta. Las tres principales *API*s son las siguientes:

OpenGL

OpenGL[1] Es una *API* de código abierto multilenguaje y multiplataforma para la representación de gráficos 2D y 3D. El proyecto fue iniciado por la empresa *Silicon Graphics* en 1991. A partir del cierre de ésta la organización sin ánimo de lucro *Khronos Group* tomó las riendas del proyecto. *OpenGL* dispone dos versiones especializadas en sistemas empotrados (*OpenGL ES*) y navegadores web (*WebGL*). La primera es un subconjunto de la API principal diseñado para su uso en dispositivos con menor potencia computacional como teléfonos móviles o tabletas. La segunda es una API en *JavaScript* para crear gráficos en 2D y 3D en navegadores sin tener que instalar complementos.
Las tres distintas APIs permiten el uso de gráficos acelerados por *hardware* en prácticamente cualquier dispositivo.

Direct3D

Ésta ha sido y es la *API* más usada para la representación de gráficos 2D y 3D en sistemas *Windows* y en las consolas *XBox*. Forma parte de *DirectX*: una colección de *API*s[2] desarrollada por *Microsoft* para la creación de aplicaciones multimedia. Surgió al mismo tiempo que *OpenGL*. Entre los dos ha habido un contínuo debate sobre cuál ofrece mejor rendimiento y facilidad de uso, que ha ido cambiando con las mejoras constantes por parte de ambos lados a lo largo de los años.

Vulkan

Vulkan es la nueva (fue publicada en 2016) *API* desarrollada por *Khronos Group* [3] en colaboración con numerosas empresas desarrolladoras de productos multimedia enfocada a aplicaciones con gráficos 3D de alto rendimiento, con prioridad en la paralelización

[1] https://opengl.org/
[2] https://docs.microsoft.com/en-us/windows/desktop/directx
[3] https://www.khronos.org/vulkan/

(algo con lo que las *API*s gráficas anteriores, especialmente *OpenGL* han tenido problemas), el mejor reparto de la carga de trabajo entre la *GPU* y la *CPU*, y la independencia de la plataforma. Aunque lleva poco tiempo en uso ya se han publicado aplicaciones que mejoran considerablemente el rendimiento de sus versiones que usan *OpenGL* o *Direct3D*.

2.2 Librerías gráficas

Las interfaces mencionadas, aunque muy potentes, son de bajo nivel y requieren mucho trabajo para llevar a cabo tareas sencillas. Por esto han surgido múltiples librerías que buscan abstraer y automatizar el uso de éstas. Aquí se presentan las más populares.

Horde3D

Horde3D[4] es un motor de *renderizado* optimizado para la representación de escenas con un gran número de elementos y centrado en videojuegos. Con este fin la librería incluye también un editor de escenarios. Tiene soporte para animaciones, efectos gráficos avanzados, etc.

Figura 2.1: Captura de pantalla de la demo *Chicago* de Horde3D

OGRE

OGRE3d[5] ofrece una librería de gráficos que prioriza la flexibilidad sobre el rendimiento. Debido a esto es compatible con un gran número de lenguajes de programación

[4] http://www.horde3d.org/
[5] https://www.ogre3d.org/

(aunque sólo C++ es soportado oficialmente) y no es dependiente de ninguna API de gráficos ni sistema operativo.

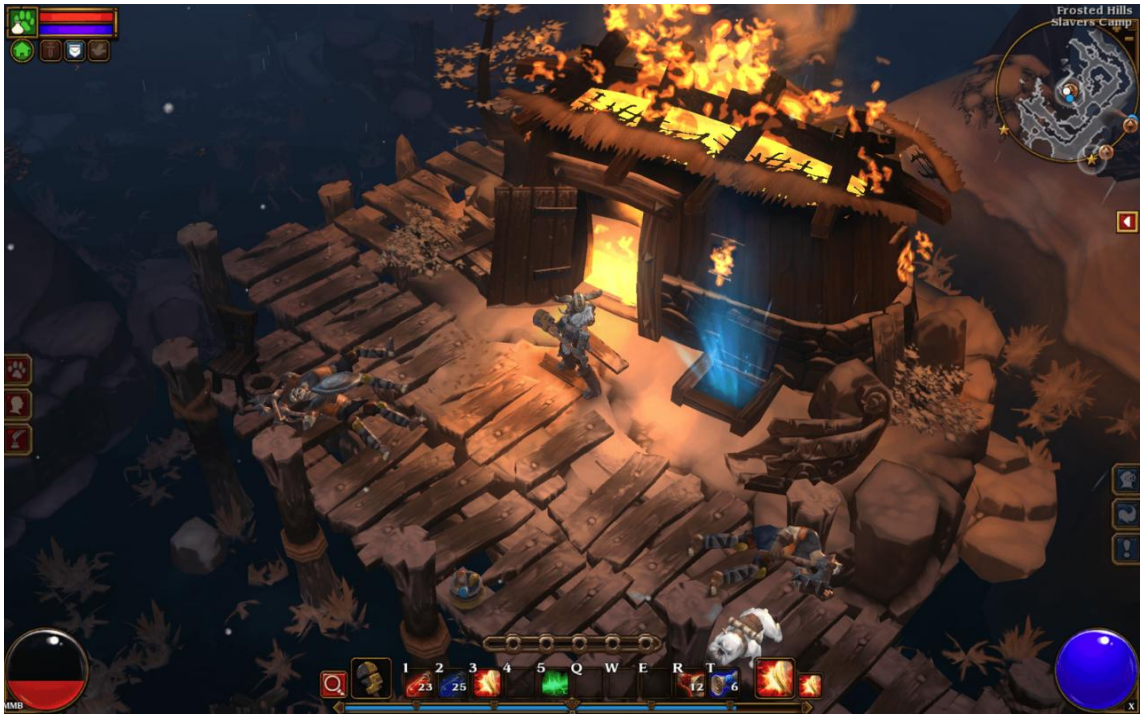

Figura 2.2: Captura de pantalla del videojuego *Torchlight II*, realizado con OGRE3D

IRRLICHT

La peculiaridad principal de esta librería es su independencia de la plataforma. Es compatible con múltiples versiones de *Direct3D* y *OpenGL* y, en caso de no tener disponible ninguna de las dos, puede utilizar su propio programa de *renderizado* capaz de funcionar en *Windows* y *MacOSX*, aunque con un rendimiento reducido[6]. Está escrito en C++ pero dispone de adaptaciones a Java, C# y VisualBasic entre otros.

OSG

OpenSceneGraph es una librería de gráficos en tiempo real de código abierto que encapsula *OpenGL* y está programada en C++. A diferencia de las alternativas, esta librería está construída por completo alrededor del concepto del grafo de escena (como implica su nombre).
OSG comenzó cómo el grafo de escena básico de un simulador de ala-delta desarrollado por Don Burns en 1998. En 1999 Robert Ostfield, compañero de trabajo de Burns, tomó el grafo de escena y lo convirtió en un proyecto independiente de código abierto que llamó *OpenSceneGraph*. A lo largo del año 2000 el proyecto comenzó a ganar popularidad y hasta el día de hoy se ha usado en numerosas aplicaciones [7] y se han añadido numerosas funcionalidades y mejoras cómo soporte a sombras, partículas y animaciones. Robert Ostfield ha anunciado recientemente el inicio del proyecto para portar *OSG* a la API de gráficos *Vulkan*.

[6] http://irrlicht.sourceforge.net/?page_id=45
[7] http://www.openscenegraph.org/index.php/gallery/use-cases

Figura 2.3: Videojuego interactivo en la Tencent Games Carnival de 2014
Este minijuego creado con OSG permitía a los asistentes de la feria luchar contra un monstruo manejando una espada que transmitía sus movimientos al videojuego.

Predecesores de OSG

OpenSG

Aunque de nombre casi idéntico, no hay que confundirla con *OpenSceneGraph*. Ambos proyectos surgieron al mismo tiempo de forma independiente y con cualidades muy similares. Sin embargo sólo *OpenSceneGraph* es mantenido actualmente, llevando más de un año sin actualización alguna en la fecha de redacción de esta memoria.[8]

OpenGL Performer

Ésta librería fue en la que se basaron *OpenSceneGraph* y *OpenSG*[9]. Propiedad de la empresa *Silicon Graphics*, fue creada específicamente para la visualización de grandes escenas con un gran número de elementos para simuladores de vuelo y similares. El cierre de la compañía y, por lo tanto, del proyecto en 2009 ayudó al aumento de popularidad de las librerías de código abierto basadas en ella.

[8] https://sourceforge.net/projects/opensg/

[9] https://web.archive.org/web/20071224141002/http://www.sgi.com/products/software/performer/

Comparativa de librerías gráficas

En esta sección se comparan las librerías gráficas mencionadas según cinco criterios: Características generales, capacidades para efectos avanzados, formatos de imagen soportados y formatos de malla soportados.
Se han excluído de esta comparativa *OpenSG* y *OpenGL Performer* dado que, cómo se ha explicado en la sección anterior, estos proyectos ya no están siendo mantenidos.

Características generales

En este apartado se muestran lado a lado los tipos de licencia usados por cada librería, los lenguajes de programación en los que se encuentran disponibles y las APIs gráficas y plataformas con las que son compatibles.
El aspecto más importante de esta tabla es la licencia usada por cada librería. *UGK* es una librería de código abierto mantenida por profesores y estudiantes de la ETSINF con la finalidad de que pueda ser usada por cualquiera para cualquier aplicación por lo que el uso de una licencia que limitara este aspecto obligando a cambiar la licencia propia o forzando a pagar *royalties* quedaría excluida inmediatamente. Respecto a los otros apartados todas las librerías usan *C++* y *OpenGL* y son compatibles con *Windows*, *Linux* y *MacOSX* por lo que son adecuadas para su uso en *UGK*.

	Licencia	**Lenguaje(s)**	**API(s) gráficas**	**Plataformas compatibles**
OSG	OpenSceneGraph Public License (LGPL)	C++	OpenGL, OpenGL ES	Windows, Linux, MacOSX, Android, iOS
OGRE	MIT License	C++, Python, Java	Direct3D 9, Direct3D 11, OpenGL, OpenGL ES, WebGL	Windows, Linux, MacOSX Android, iOS, XBOX 360, PS3
Horde3D	Eclipse Public License	C++	OpenGL	Windows, Linux, MacOSX
OpenGL Performer	Proyecto sin soporte			
OpenSG	Proyecto sin soporte			
IRRLICHT	zlib License	C++, VisualBasic, Delphi, Java	Direc3D, OpenGL, propio	Windows, Linux, MacOSX, Solaris,

Utilidades avanzadas

En la siguiente comparativa, se muestran las capacidades gráficas avanzadas de las bibliotecas empleadas en el estudio preliminar. El hecho de que una librería no disponga de las herramientas para un efecto concreto no significa que no lo soporte sino que el programador deberá realizar un esfuerzo mayor para conseguirlo al tener que programar esa funcionalidad por su cuenta. Las herramientas más importantes son los *LOD* (siglas en inglés para "nivel de detalle"), que permiten cambiar la resolución de texturas en la escena dependiendo de la distancia a la cámara para ahorrar recursos, las partículas, que permiten crear efectos como fuego o humo y resultan difíciles de implementar desde cero, el *Bump Mapping*, que permite simular características tridimensionales como bultos

o arrugas en texturas en dos dimensiones y el *HDR* (siglas en inglés para "alto rango dinámico", que permite simular el modo en que el ojo humano responde al contraste entre luces y sombras.
También hay que mencionar en este apartado la facilidad para encontrar esta información en la documentación de cada librería. En este aspecto *IRRLICHT* se merece una mención especial por la calidad de su documentación.

Figura 2.4: Modelo del videojuego *Monster Hunter* dibujado con *OpenSceneGraph*
Aquí se muestran múltiples capacidades de la librería *OSG* como sombreado, oclusión ambiental e iluminación.

	Materiales fuera de código	**Mezclado multi-textura y multipaso**	**Animación de texturas**	**Efectos multipaso**	**Material LOD**	**Partículas**
OSG	No	Sí	Sí	No	Sí	Sí
OGRE	Sí	Sí	Sí	Sí	Sí	Sí
Horde3D	No	No	No	No	Sí	Sí
IRRLICHT	No	No	Sí	No	Sí	Sí

	Skyboxes	**Billboarding**	**Luces dinámicas**	**Bump Mapping**	**Parallax Mapping**
OSG	No	Sí	Sí	Sí	No
OGRE	Sí	Sí	Sí	Sí	Sí
Horde3D	No	No	Sí	Sí	Sí
IRRLICHT	Sí	Sí	Sí	Sí	Sí

	Niebla	Luz volumétrica	HDR
OSG	Sí	Sí	Sí
OGRE	Sí	Sí	Sí
Horde3D	Sí	No	Sí
IRRLICHT	Sí	Sí	Sí

Tabla 2.1: Efectos soportados por cada librería.

Figura 2.5: Captura de pantalla de la demo de luces dinámicas y bump mapping de IRRLICHT

Formatos de imagen soportados

Aquí se muestran los formatos de imagen que son capaces de manejar cada librería sin el uso de herramientas externas. Es importante el soporte de los formatos de imagen más comunes como *BMP*, *JPEG* y *GIF* pero son aún más importantes los formatos especializados en el almacenamiento de texturas como *DDS*, que permite ser descomprimido por la GPU o las texturas volumétricas, que son texturas que abarcan un volumen en lugar de un área. Este tipo de texturas se utilizan para el *renderizado* de volúmenes o para conseguir efectos similares a los de la oclusión ambiental por una fracción del coste computacional.

	BMP	JPEG	PNG	TGA	GIF	DDS
OSG	Sí	Sí	Sí	Sí	Sí	Sí
OGRE	Sí	Sí	Sí	Sí	Sí	Sí
Horde3D	Sí	Sí	Sí	Sí	Sí	Sí
IRRLICHT	Sí	Sí	Sí	Sí	Sí	No

	Tex. Volumétricas	RGB	PCX	PSD	WAL	PPM
OSG	No	Sí	No	No	No	No
OGRE	Sí	No	No	No	No	No
Horde3D	No	No	No	No	No	No
IRRLICHT	No	Sí	Sí	Sí	Sí	Sí

Tabla 2.2: Formatos de imagen soportados por cada librería

Formatos de malla soportados

	.obj	.3ds	.dae	.b3d	.ms3d	.bsp	.md2	.x	COLLADA
OSG	Sí	Sí	No	No	No	No	Sí	Sí	Sí
OGRE	Sí	Sí	No	No	Sí	No	No	No	No
Horde3D	No	No	No	No	No	No	No	No	Sí
IRRLICHT	Sí	No	No	Sí	Sí	Sí	Sí	Sí	Sí

Tabla 2.3: Formatos de mallas soportados por cada librería.

2.3 Resultados de la comparación

Plataformas

Todas las librerías comparadas utilizan como mínimo *OpenGL* por lo que todas son compatibles con Windows, Linux y MacOSX. Sin embargo *Horde3D* se queda detrás del resto al no tener soporte para para dispositivos móviles, dado que no soporta *OpenGL ES* ni *WebGL*, aunque la compatibilidad con estas APIs se encuentra en desarrollo. Por otro lado cabe destacar *OGRE* e *IRRLICHT* por la gran selección de plataformas soportadas, especialmente *OGRE*, que ha sido usado incluso en consolas como *XBOX360* y *PS3*.

Licencia

Aunque cada librería hace uso de licencias distintas todas tienen las mismas características principales. Ninguna de las librerías comparadas requiere de pago alguno para el uso de éstas ni la distribución de productos realizados con ellas. Tampoco fuerzan la aplicación de su licencia al resto del programa que las use.
La mayor diferencia reside en cómo tratan la distribución de la propia librería en el caso de que haya sido modificada: *OSG* y *Horde3D* lo permiten bajo la condición de que la librería se distribuya sólo en forma de código objeto (librerías compiladas). En caso de distribuirse el código fuente la licencia de la librería se aplica también al código modificado. *OGRE3D* e *IRRLICH* permiten la distribución del código fuente, modificado o no, para cualquier uso.
Ninguna de las librerías comparadas ofrece ninguna garantía de fiabilidad ni rendimiento.

Capacidades y facilidad de uso

Aquí es donde se ve mayor diferencia entre los candidatos.
En cuanto a capacidades, la librería superior es *OGRE* seguida por *IRRLICHT* y *OSG*. *Horde3D* se queda un poco detrás aunque esto es comprensible dado que el resto de librerías han existido durante mucho más tiempo (*Horde3D* publicó su versión 1.0 en 2017). Sin embargo la mayoría de efectos más comunes como *HDR*, niebla, *bump mapping*, luces dinámicas o partículas tienen soporte directo en todas las librerías. Además, como se ha indicado antes en las tablas, que no tenga soporte directo no significa que no se pueda conseguir ese efecto sino que para realizarlo se requiere más trabajo.

También es importante el número de formatos de imágenes y modelos con los que cada una es compatible. En cuanto a formatos de imagen todos soportan los más comunes con mención especial para *IRRLICHT*, que soporta también muchos formatos menos populares.

Respecto a los modelos hay dos grupos: por un lado *OSG* e *IRRLICHT* soportan un gran número de formatos sin necesidad de ninguna herramienta externa. Por el otro *OGRE* y *Horde3D* utilizan un formato concreto al que hay que convertir usando programas no incluídos en el paquete de la librería. Ambos enfoques tienen ventajas y desventajas. Un formato único simplifica la carga y procesado de modelos y hace el código más fácil de mantener pero encontrar la herramienta adecuada para convertir un formato concreto puede llegar a convertirse en un problema. No depender de un sólo formato resulta muy práctico para el usuario pero más difícil de mantener y más dado a errores.

Conclusiones

Todas las librerías analizadas son potentes desde el punto de vista del resultado visual en pantalla, siendo *OSG* y *OGRE3D* los cuales han sido utilizados en el desarrollo de videojuegos por estudios profesionales[10] [11]. Cómo ya se ha explicado antes, el objetivo a largo plazo del proyecto de *UGK* es poder usar cualquiera de éstas librerías a voluntad del usuario por lo que ésta decisión no es más que elegir por cuál empezar. La elección de *OSG* se debe a que es la librería más fácil de usar desde el punto de vista de alguien que no ha usado una librería de gráficos con anterioridad y su diseño centrado en el grafo de escena la hace muy útil para aprender a usar esta estructura. Otra razón para la elección de *OSG* es que *UGK* ya tiene abstracción del grafo de escena lista para funcionar con *OSG*, realizada en un intento anterior de integrar dicha librería sin éxito.

2.4 Funcionamiento de OpenSceneGraph

Ahora que se ha determinado con qué librería se va a llevar a cabo el proyecto es importante explicar el funcionamiento de ésta y de su base: el grafo de escena.

La base de *OSG* es la estructura de datos conocida como el Grafo de Escena.
Los nodos son el elemento básico del grafo de escena. Éstos representan un elemento de la escena (un modelo, una luz, una cámara, etc.) y el grafo está formado por las relaciones entre estos nodos. El grafo está estructurado como un árbol, con un nodo raíz del que el resto de nodos dependen, y nodos hoja, que albergan los modelos. Entre la raíz y las hojas hay una variedad de nodos, cada uno con una función, que afectan a sus nodos hijo.

Funcionamiento del grafo de escena

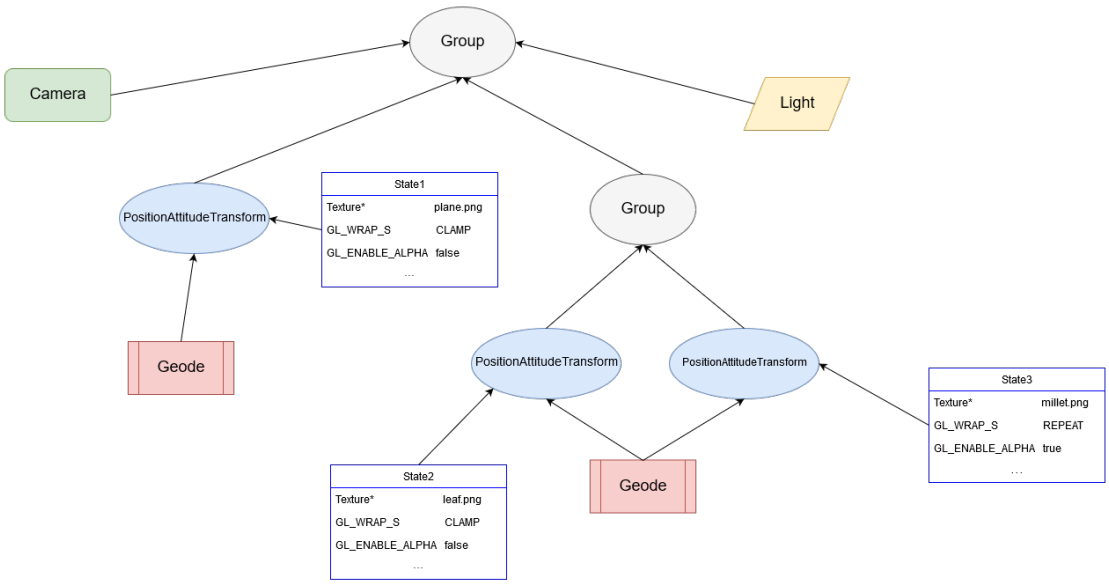

Figura 2.6: Esquema básico de un grafo de escena de OSG

[10] http://www.openscenegraph.org/index.php/gallery/use-cases/170-interactive-game-on-tencent-game-carnival-
[11] https://www.ogre3d.org/showcase

Group

La función del nodo *Group* es, cómo indica su nombre, agrupar varios nodos. En *OSG* los atributos de un nodo se pasan a los de sus hijos por lo que tener múltiples nodos agrupados permite cambiar las propiedades de varios nodos a la vez.

Geode

Geode es la abreviatura de *Geometry descriptor*, que significa "Descriptor de geometría". Estos nodos son los que contienen los modelos a ser renderizados. Los nodos de descripción de geometría no pueden tener nodos hijo, ya que estos han de ser las hojas del grafo de escena.

PositionAttitudeTransform / MatrixTransform

Estos dos nodos cumplen la misma función: aplicar una transformación geométrica a sus nodos hijo. Siendo distintas las formas de acceder y modificar la matriz de transformación de éstos. Para dibujar una escena el programa recorre el árbol en ”profundidad-primero” empezando por el nodo raíz. Cada vez que encuentra un nodo de transformación (`PositionAttitudeTransform`) ésta se aplica a todos sus nodos hijo. Al llegar a un nodo hoja (los cuales contienen las mallas a dibujar) éste ya ha recibido todas las transformaciones que le corresponden.

Camera

El nodo *Camera* representa una cámara en la escena que mostrará el subgrafo que tenga asignado como hijo en el objetivo que se le asigne. Éste objetivo puede ser una ventana, una porción de la misma o una textura para ser procesada o mostrada en la escena más adelante. El uso de este nodo no es totalmente necesario ya que al crear una vista se genera una cámara automáticamente.

View

Aunque no es un nodo la clase *View* (Vista) es la más importante. Esta clase encapsula el proceso de creación y gestión de ventanas e inicia los recorridos del grafo.

Light

Éste nodo representa una luz en la escena. Puede ser un punto, un cono, una luz direccional, etc. Un nodo Luz asignado a un grafo afecta a todos los nodos de éste, sean sus hijos o no. El uso de éste nodo no es totalmente obligatorio ya que al crear una vista se crea una luz básica, aunque es recomendable crear una propia si se quiere tener control sobre ésta.

Sets de estados

Todos los nodos tienen un atributo que encapsula el set de estados de los atributos de OpenGL. Éstos incluyen la textura que se va a usar y sus propiedades, el método de mapeado, el uso de luces y transparencias, etc. El que se aplica al dibujar la malla es el

del último nodo padre que lo ha especificado. Sin embargo ésto sólo sobreescribe los parámetros indicados, lo que permite definir atributos en los nodos padre que se aplicarán a todos sus hijos hasta que alguno los cambie.

Tan sólo con estos nodos básicos y la clase Viewer*OSG* es capaz de crear una escena y mostrarla en pantalla, siendo la librería misma la que crea y gestiona la ventana. La librería tiene muchas más funciones para crear cualquier efecto que se desee: sombras, generación de texturas, niveles de detalle, animaciones, etc. Estas funciones avanzadas están compartimentadas en componentes llamados *NodeKits*. La librería de *OSG* no los necesita para funcionar salvo que se vaya a usar una de sus funciones, permitiendo adaptarla a las necesidades del proyecto.

NodeKits

Aquí hay una lista de los *NodeKits* disponibles y una breve descripción de la función de cada uno.

osgParticle Como indica el nombre esta extensión tiene utilidades para crear efectos de partículas.

osgText Otorga soporte de fuentes FreeType y TrueType para mostrar texto en la escena.

osgFX Permite implementar efectos especiales.

osgShadow Soporte para múltiples técnicas de dibujado de sombras.

osgManipulator Permite interactuar con los objetos de la escena.

osgTerrain Esta extensión tiene funciones para el dibujado de terrenos usando mapas de altura e imágenes cartográficas.

osgAnimation Ayuda en la realización de animaciones de múltiples tipos.

osgVolume Añade soporte para dibujado de alto volumen usado en aplicaciones como la visualización de datos.

Procesamiento del grafo de escena

OSG recorre el grafo tres veces por cada imagen generada. Cada pasada realiza una función distinta:

Actualización

En este recorrido el programa actualiza los valores que hayan cambiado desde la última llamada de dibujado, como sets de estados o matrices de transformación.

Manejo de eventos

Aquí *OSG* comprueba si se han dado ciertos eventos (pulsaciones del teclado, ratón o eventos emitidos por otros nodos) y responde a ellos.

Recortado

El último recorrido determina qué nodos están fuera del campo de visión de las cámaras a las que estén asignadas para optimizar el dibujado.
Debido a que *OSG* paraleliza los recorridos del grafo, cambiar valores de los nodos puede causar problemas. Por esto la librería ofrece una lista de *callbacks* modificables que son llamados en cada nodo durante los recorridos de *OSG* para modificar las propiedades de éstos de forma segura.

CAPÍTULO 3
Análisis del problema

3.1 Funcionamiento de UGK

La base de *UPVGameKernel* es la clase `CCharacter`. Ésta clase es la base sobre la que se construyen todas las entidades de un videojuego sean concretas, como cámaras, luces, personajes, etc. o abstractas como es el caso del gestor del bucle principal del juego o el gestor de un grupo de personajes, como por ejemplo una flota de naves.
Para gestionar la creación y almacenamiento de personajes, texturas y mallas el motor usa el patrón de "Fábrica y almacén de objetos". Esto es: el motor tiene dos clases para gestionar cada objeto (Personajes, texturas, etc.): un "Almacén" y una "Fábrica". En caso de necesitar un objeto nuevo el motor solicita uno al almacén y, si hay disponibles, se lo entrega. En caso de no haber ningún objeto disponible el almacén llama a la fábrica para que cree uno nuevo. Después, cuando el motor ya no necesita dicho objeto, en lugar de ser eliminado se desactiva y se guarda en el almacén para uso futuro. Esto evita llamar constantemente al gestor de memoria para construir o destruir objetos y evita la fragmentación de la memoria dinámica, reduciendo de paso la probabilidad de fallos de memoria dinámica no referenciada.
Aquí se describen los componentes principales de *UGK*:

3.1.1. El grafo de escena

La implementación del grafo de escena original de *UGK* no es realmente un grafo. Consiste en una doble cola que almacena punteros a los objetos `CCharacter` que forman la escena. Cuando se llama al método `Render()` éste recorre la cola secuencialmente y llama al método del mismo nombre de cada personaje. Aparte de almacenar las referencias a todos los personajes de la escena también realiza la función de purga. Esto consiste en recorrer la cola de personajes y borrar las referencias a los que se encuentran inactivos.

3.1.2. Personajes

Los personajes son representados por la clase `CCharacter`. En *UGK* un personaje es cualquier entidad que exista en el videojuego, desde personajes, cámaras o luces a los gestores de texturas o mallas.
La clase `CCharacter` contiene referencias a los sonidos, la malla y el índice de la textura que utiliza (más información sobre el uso de las texturas en la sección Texturas), la información sobre su *AABB* para la detección de colisiones, la máquina de estados de la inteligencia artificial y los atributos del personaje al que representa como posición, rotación, escala, velocidad y aceleración.

Estos atributos son utilizados a la hora de dibujar el personaje en pantalla. Es la propia clase `CCharacter` la que se encarga de llevar a cabo el dibujado de su modelo en la escena aplicando las transformaciones y la textura a la malla. Además el método de dibujado no está implementado en el motor y ha de ser definido en el código del juego que utiliza *UGK*.

3.1.3. Mallas

Los modelos o mallas se almacenan en objetos de la clase `CMesh3D`. Éstos objetos contienen la información de su malla en un objeto `Model_3DS`, que guarda los datos de la malla como vectores de vértices, normales, etc. tras cargarlos de un archivo en formato *.3ds* . También guardan el nombre y el camino relativo del archivo desde el cual se ha cargado el modelo.

3.1.4. Texturas

UGK usa la clase `CTexture` para almacenar las texturas. Dentro de la clase hay una estructura `TTexture` que contiene sus atributos y su número de identificación de *OpenGL*. Además contiene métodos para cargar texturas a partir de imágenes en formato *.tga* y *.bmp*.
Es importante destacar también que en *UGK* las texturas no son accedidas del mismo modo que los modelos. El gestor de texturas no entrega al personaje una referencia a la textura para uso posterior sino que, cuando `CCharacter` llama al método `Render()`, el personaje solicita al gestor que introduzca la textura con el índice indicado en el estado de *OpenGL*, aplicándose así en su malla en el momento de dibujado mediante el método `CTexture::SetTexture()`.

3.1.5. Cámaras

La clase `CCamera` es una extensión de `CCharacter` que contiene los atributos propios de una cámara como las distancias máximas y mínimas de visión, el punto de foco o el ángulo del campo de visión. La cámara en *UGK* puede cambiar entre proyección en perspectiva y ortogonal. Además tiene soporte para animaciones usando curvas de Bezier y la generación de imágenes estereoscópicas, usadas en las gafas de realidad virtual.
El manejo de la cámara se realiza con varios métodos que modifican sus atributos como `LookFrom()` o `LookAt()`. Después estos cambios se aplican a la matriz de proyección de *OpenGL* mediante la función `SetLook()`.

3.1.6. Ventanas

UGK dispone de soporte para la creación de ventanas con las apis *WIN32* y *GLUT* pero la creación y gestión de los eventos de entrada se ha de implementar en el videojuego ya que el motor no dispone de una.

Otros elementos importantes del motor son `UGKParser`, que permite leer archivos HTML para inicialización o carga de niveles, `Quadtree`, una estructura de datos usado para detectar colisiones de forma eficiente mediante un árbol cuaternario para dividir el espacio del juego y un sistema de simulación de físicas independiente de las imágenes por segundo del juego. Para unir y coordinar todos los componentes del motor *UGK* utiliza un sistema de mensajería y temporizadores.

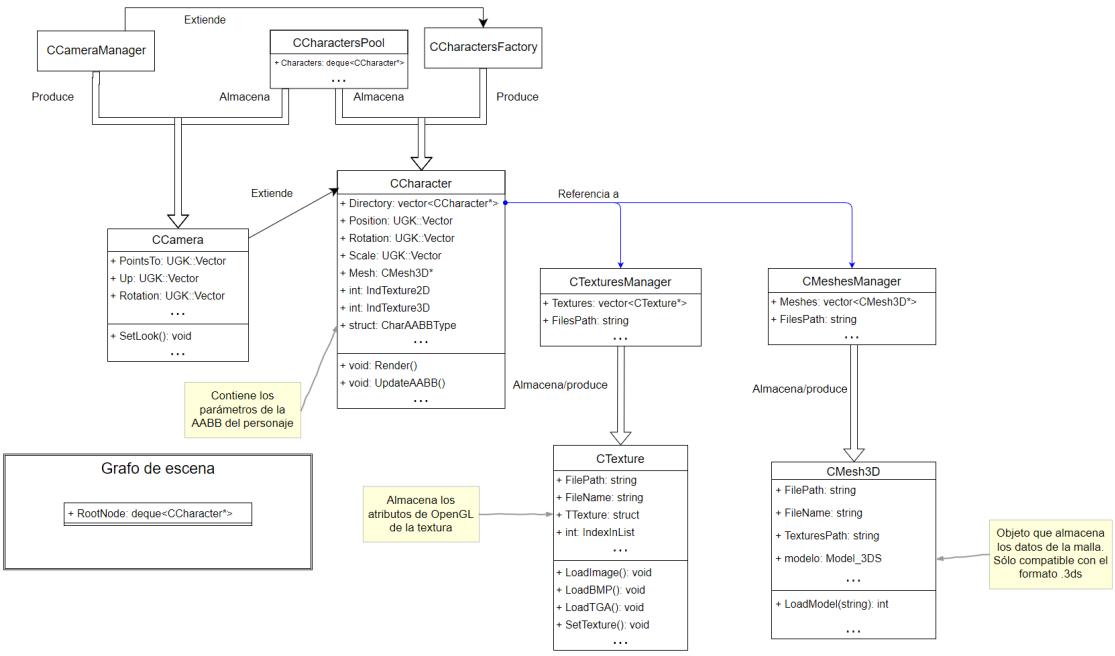

Figura 3.1: Esquema de la gestión de personajes, mallas y texturas

3.2 Problemas principales

UPVGK ya tiene una implementación muy básica de un grafo de escena con *OSG* pero una gran parte del motor y todo el videojuego de *Space Invaders* no están preparados para usarlo. El grafo propio de *UGK* realiza tan sólo una pequeña parte de las funciones que debería desempeñar, con la creación y gestión de la ventana dividida entre el motor y el videojuego que lo usa, el proceso de *renderizado* repartido en múltiples niveles y secciones del motor y con soporte para muy pocos formatos de mallas y texturas.
En su estado actual el motor y el videojuego *Space Invaders* aún tienen partes unidas de tal modo que separarlos resulta complicado. El éxito definitivo de este proyecto se dará cuando el videojuego funcione con una abstracción de un grafo de escena sin información alguna sobre su implementación. Cuando se consiga hacer funcionar otro videojuego con esta versión de *UGK* igual que *Space Invaders* se confirmará el éxito del proyecto. Precisamente en el momento de desarrollo de este trabajo un compañero de la ETSINF está trabajando en actualizar el videojuego *Hyperchess*, un videojuego de ajedrez en 3D, a la versión actual de *UGK*. Este videojuego fue creado con lo que en su momento era el prototipo de *UGK* y muchas de las funciones que ahora están abstraídas en el motor se encuentran en el código fuente del videojuego. Una vez terminada la actualización de *Hyperchess* éste será un candidato excelente para probar la nueva implementación del grafo de escena.

3.2.1. Renderizado y grafo de escena

La mayor carencia del motor en su estado actual reside en su implementación del grafo de escena. Éste no actúa realmente como un grafo, sino como un vector que almacena los personajes de la escena sin ningún tipo de jerarquía ni relación entre ellos. Tampoco realiza las funciones para las que un grafo de escena está diseñado para facilitar, cómo el recortado por campo de visión o la formación de subgrafos para mostrar modelos compuestos.

3.2.2. Personajes

A la hora de dibujar la escena en pantalla el grafo recorre el vector llamando al respectivo método de dibujado de cada personaje, en el que cada uno tiene la capacidad de modificar el estado de *OpenGL* de forma individual. Esto hace muy difícil la depuración del proceso de dibujado y obliga a implementarlo de nuevo cada vez que se añade un personaje nuevo.
Además *SpaceInvaders* tiene personajes que no están soportados por el motor y su implementación reside por completo en el código del videojuego. Estos son los siguientes:

Brick Este personaje representa cada ladrillo individual de los obstáculos (*Búnkeres*) que se encuentran entre la nave del jugador y las enemigas. Al crearse un *Búnker* éste es introducido en el grafo de escena pero sus ladrillos individuales no, por lo que el recorrido de éstos para la detección de colisiones y dibujado se hace por completo fuera del grafo de escena.

Bonus Son objetos que se aparecen de vez en cuando en lo alto de la pantalla y bajan lentamente para ser recogidos por el jugador para conseguir un potenciador temporal. Éstos están implementados completamente en el juego sin intervención del motor. Ni siquiera pertenecen a la clase `CCharacter` y tienen un gestor propio con una fábrica y almacén dedicada exclusivamente a ellos. Por consiguiente tampoco están presentes en el grafo de escena.

Background Es una esfera con una textura que engloba toda la escena para tener una imagen de fondo desde cualquier ángulo. Esta entidad tampoco hereda de `CCharacter` ni se encuentra en el grafo de escena.

3.2.3. Materiales

UGK no dispone de materiales como tales. Su soporte para texturas es muy básico. Tan sólo permite cargar una imagen en formato *.tga* o *.png* y sus atributos de *OpenGL* como el modo de mapeado son determinados en el código en el momento de carga.

3.2.4. Cámara

La cámara es tratada en el motor como un personaje más y, por lo tanto, es gestionada desde la clase *CCharacterManager*, sin embargo la cámara tiene también un gestor propio: *CCameraManager*. Este segundo gestor no es necesario.

3.2.5. Luces

UGK no tiene ningún soporte para manipular luces. En *SpaceInvaders* las luces de la escena son luces estáticas introducidas directamente en la inicialización del juego con una llamada a *OpenGL*.

3.2.6. Ventana

La ventana creada por *UGK* no puede ser desplazada ni redimensionada y siempre se muestra encima de cualquier otra ventana en el escritorio, esté ésta en primer plano o no.

CAPÍTULO 4
Diseño de la solución

4.1 Modificaciones

Gracias al esquema de *Fábrica-almacén* empleado por el motor para la gestión de mallas, texturas y personajes la implementación de éstos está separada del resto del motor, por lo que adaptarlos a *OSG* ha resultado sencillo.

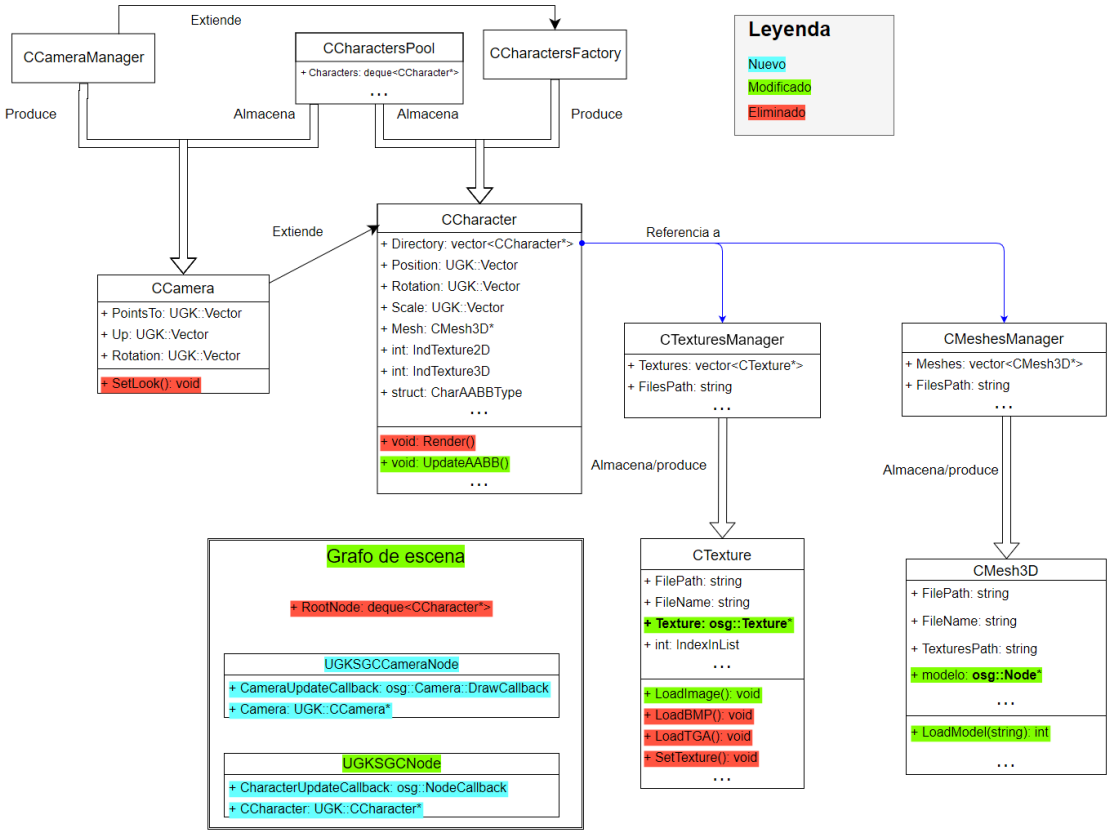

Figura 4.1: Esquema con las modificaciones principales realizadas a *UGK*

4.1.1. Modelos

Dentro de la clase `CMesh` se guarda la información del modelo: el nombre del archivo, la dirección de éste y un puntero a la propia malla. En su estado original la malla se

almacenaba en un objeto `Model_3ds` que contenía los vértices, normales y el resto de datos de la malla tras cargarlos usando código propio, sólo compatible con el formato *.3ds*.
Para adaptar la clase a *OSG* se han realizado dos cambios: primero se ha cambiado el puntero de la malla a un puntero de referencia de *OSG* que apunta a un nodo. Después se ha sustituído el código para cargar modelos *.3ds* por la función `readNodeFile` de la extensión *osgDB*. Esta función es capaz de cargar múltiples formatos de malla distintos y los convierte en un nodo básico de *OSG* con un nodo `Geode` como hijo, el cual contiene la malla cargada. Después se asigna el nodo creado al puntero de la clase `CMesh`.

4.1.2. Texturas

La clase `CTexture` seguía el mismo esquema que `CMesh`, sin embargo los cambios realizados han sido ligeramente distintos.
La referencia a la textura se ha cambiado por un puntero a un objeto `CTexture` de *OSG* y el método para cargar un archivo de imagen ha sido sustituido por el propio de *OSG*. Esto permite cargar muchos más formatos de imagen que los que era capaz antes.
Además se ha cambiado el método `SetTexture` para que tome un nodo como parámetro y le asigne la textura referenciada junto con sus parámetros al set de estados de dicho nodo.
Como solución temporal los parámetros de *OpenGL* de la textura están definidos en el código sin posibilidad de cambio. En un futuro sería preferible poder cambiarlos externamente como propiedades abstractas de la textura, sin ninguna referencia a *OSG* ni *OpenGL*.

4.1.3. Nodo personaje de UGK

Éste nodo tiene la finalidad de representar a un personaje de *UGK* en el grafo de escena.

La clase `UGKOSGCNode` es una extensión del nodo `PositionAttitudeTransform`, que representa una transformación de posición, rotación y escala que se aplica a su subgrafo. Todos los nodos de *OSG* tienen capacidad para albergar un set de estados de *OpenGL* que se aplican a todas las mallas de su subgrafo. Por lo tanto, aunque el nodo en cuestión sólo representa la transformación de su personaje, éste también alberga la información sobre la textura que se aplicará a su modelo y cómo se procesará a la hora del dibujado.
Esta extensión añade dos miembros: una referencia al personaje el cual representa y un *callback* que actualiza la posición, rotación y escala del personaje, la textura y el modelo según los parámetros del personaje referenciado.
El uso del *callback* es necesario para actualizar el nodo debido al modo en que *OSG* gestiona los recorridos del grafo, tal y cómo se ha ilustrado en la presentación de la librería. De este modo todos los nodos toman la posición, rotación y escala de su personaje durante el recorrido de actualización, evitando así condiciones de carrera que podrían producirse si se cambiaran estos valores en cualquier otro momento.
Así, cuando se le solicita a la fábrica crear un personaje, además de crear un objeto `CCharacter` crea además un nodo de personaje, le da una referencia al `CCharacter` y lo introduce en el grafo de escena como hijo del nodo raíz. Gracias a este esquema las modificaciones de la propia clase `CCharacter` han sido mínimas.
Tal y cómo está configurado ahora sólo se pueden añadir modelos simples al grafo. Sin embargo *OSG* soporta modelos complejos usando transformaciones que sean hijas de otras transformaciones. Una propuesta para proyectos futuros es añadir la capacidad pa-

ra introducir nodos *UGKSGCNode* al grafo de escena como hijos de otros nodos que no sean la raíz para soportar personajes complejos.

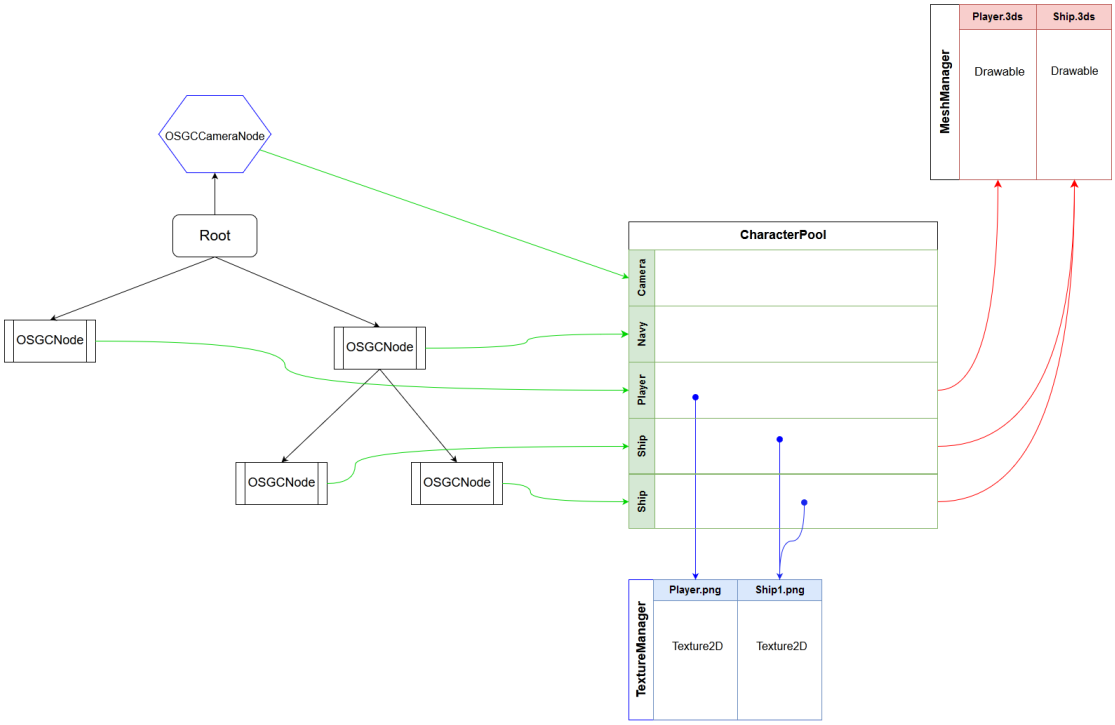

Figura 4.2: Esquema del grafo de escena de UGK con OSG

4.1.4. Personajes

La única modificación que ha sido necesaria en la clase `CCharacter` ha sido el cómo referencia a su textura. Antes la clase `CCharacter` usaba su referencia textura para introducirla en el estado de *OpenGL* cuando ejecutaba la función de renderizado.
Ahora esto no es necesario ya que esta función ha sido relegada al grafo de escena, por lo que el personaje sólo almacena la referencia a su textura para ser accedida por el grafo a la hora de dibujar la escena.
Además, `CCharacter` extrae la caja contenedora (AABB de sus siglas en inglés *Axis Aligned Bounding Box*) de su malla para dársela al sistema de detección de colisiones. Esto antes se llevaba a cabo recorriendo los vértices para encontrar los puntos en los extremos en cada eje de coordenadas pero con *OSG* la clase `CCharacter` ya no tiene acceso a los vértices de su malla. Por lo tanto para conseguir la caja se ha usado un objeto `BoundingBox` de *OSG* que luego es expandido al tamaño de la malla.

4.1.5. Cámaras

La clase `CCamera` se ha modificado para que, en lugar de manipular la matriz de proyección de *OpenGL* según sus parámetros, sirva de almacén de los atributos de la cámara para ser leídos por su respectivo nodo de cámara en el grafo de escena cuando sea necesario.

4.1.6. Ventanas

Al igual que la mayoría de los cambios indicados anteriormente la adaptación de la clase `CWindow` consiste en convertirla en un almacén para los parámetros de la ventana para pasárselos a *OSG*.
UGK utiliza la API *WIN32* para crear ventanas y recibir la entrada del teclado y el ratón. *OSG* permite insertar la vista en una ventana de *WIN32*, sin embargo eso requiere más trabajo que crear una ventana propia de *OSG*. Por lo tanto se ha decidido crear una ventana a parte inicialmente y, si es posible, integrarla más adelante en la propia ventana de *WIN32*. En caso de no poder realizarse la integración de la ventana puede servir para un proyecto futuro. En ese caso habría que decidir si seguir usando *WIN32*, cambiar a otra API distinta o utilizar la propia de cada implementación del grafo de escena que esté usando en el momento.

4.2 Adiciones

4.2.1. Nodo de cámara genérico

Con el fin de que las cámaras de *UGK* se comporten como un personaje más se ha creado la clase `UGKOSGCCameraNode`, análoga a la clase `UGKOSGCNode`, que toma los parámetros del objeto `CCamera` al que está enlazado y actualiza la cámara de *OSG* a dichos parámetros. Sin embargo este nodo alberga algunas diferencias respecto al nodo de personaje. Concretamente este nodo utiliza un *callback* distinto llamado `osg::Camera::DrawCallback`. Este callback es llamado después del recorrido de actualización pero antes que el recorrido de dibujado. Esto permite actualizar los atributos de la cámara en el grafo en el momento adecuado de forma segura.

4.2.2. Luces

OSG representa las luces usando el nodo `LightSource`. A través de este nodo se pueden cambiar los atributos de su luz, como el color difuso, especular o ambiente y el tipo de luz (puntual, direccional, global, etc.). Este nodo no debe tener hijos salvo si es un único nodo con una malla (`Geode`) para darle una representación física de la luz a la escena, como por ejemplo una bombilla.
Una vez en el grafo todos los nodos de éste se verán afectados por la luz. Al igual que cualquier otro nodo se puede modificar la posición de una luz en la escena asignándola como hija de un nodo `PositionAttitudeTransform`.
De este modo las luces son representadas en *UGK* como una extensión de la clase `CCharacter` que, en lugar de albergar una referencia a una malla y una textura tendrá una referencia a un nodo `LightSource` y almacenará los atributos necesarios de su luz para asignárselos al nodo en el grafo.

4.3 Plan de trabajo

El proyecto se divide en dos partes: la de análisis y preparación y la de ejecución.

4.3.1. Fase de análisis y preparación

En esta fase se preparan las herramientas a utilizar y se analiza el problema con el fin de determinar cómo se va a llevar a cabo la fase de ejecución.

Compilar e instalar herramientas

Esta fase consiste en la compilación e instalación de las librerías de *OSG* y *UGK* además del videojuego *Space Invaders* para que todas funcionen correctamente en el PC de trabajo.

Análisis de características *OSG*

Aquí se prepara una lista de las características de *OSG*.

Análisis de características *UGK*

Dónde se recogen las características del motor *UGK* en el apartado de gráficos para ser comparadas con las de *OSG*.

Comparación *OSG-UGK*

Aquí se comparan las características de ambas librerías para determinar cuales son las funcionalidades de las que carece *UGK* que se pueden suplir con *OSG*.

Análisis de integración

En esta fase se estudian ambas librerías para determinar cómo se va a adaptar *UGK* para que utilice *OSG*.

Aprendizaje de *OSG*

Esta fase está dedicada al aprendizaje del uso de *OSG* para la creación de un grafo de escena y sus componentes.

4.3.2. Fase de ejecución

Adaptación de las mallas

Aquí se modifica el código que gestiona las mallas para que éstas sean compatibles con el nuevo grafo de escena y utilicen el componente de *OSG*, *osgDB*, para la carga de mallas.

Adaptación de las texturas

En esta fase se sigue el mismo procedimiento seguido con las mallas para adaptar las texturas a *OSG*.

Adaptar la clase `CCharacter`

La clase `CCharacter` será modificada para que utilice la nueva implementación de las mallas y texturas y para que el grafo de escena de *OSG* pueda acceder a sus atributos.

Adaptar la clase `CCamera`

Aquí se modifica la clase `CCamera` con el fin de que interactúe con el grafo de escena del mismo modo en que lo hace `CCharacter`.

Adaptar la ventana al nuevo grafo de escena

En esta fase se realizan los cambios a la creación y gestión de ventanas de *UIGK* para que sea compatible con los cambios realizados en el grafo de escena.

Añadir luces al grafo de escena

Dónde se añadirá soporte al grafo de escena para crear y gestionar luces.

4.3.3. Tareas adicionales

Aparte de estas fases se ha realizado la redacción de la memoria, que tiene lugar durante toda la duración del proyecto, y la preparación para la presentación de la defensa, que tiene lugar la semana antes de ésta.

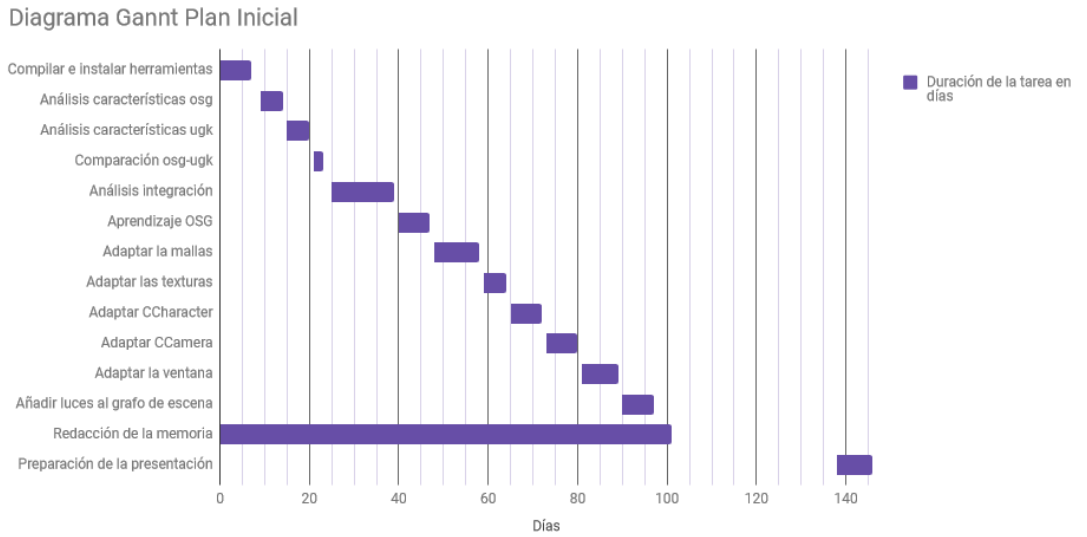

Figura 4.3: Planificación inicial del proyecto

CAPÍTULO 5
Resultados

En este capítulo se muestran los resultados obtenidos como consecuencia de aplicar el plan de trabajo anterior. Así mismo, se mostrarán qué objetivos se han alcanzado, justificando tanto los motivos que han permitido cumplirlo como las razones por las cuales no se han podido alcanzar algunos de ellos, que se dejarán para trabajos futuros en los siguientes TFGs que se propongan para continuar esta línea de trabajo el año que viene. Así mismo, se acompañará una evaluación del plan de trabajo mostrado en la sección de análisis del problema y del grado de adecuación al real, justificando las razones por las que se han producido las discrepancias.

5.1 Objetivos cumplidos

Eliminación de llamadas a *OSG* y *OpenGL* en el videojuego

Se ha modificado el motor de modo que todo el proceso de *renderizado* sea gestionado íntegramente desde el grafo de escena. En lugar de necesitar llamar a los métodos `Render()` de cada personaje ahora es el grafo de escena el que toma los atributos de los personajes y los aplica a sus mallas respectivas a la hora de dibujar la escena. Esto ha permitido simplificar la implementación de nuevos personajes en el futuro ya que ha dejado de ser necesario que cada uno implemente su propio método `Render()`. Además este cambio en la estructura del motor permitirá el cambio de librería de gráficos de forma sencilla en el futuro.

Las mallas de los personajes se muestran en pantalla tal y como indica la escena

Se ha conseguido que los personajes estándar (estos son los que no estaban implementados únicamente en el videojuego) se dibujen en la escena en sus posiciones definidas en los archivos *HTML* usados para definir la escena. Sin embargo algunos modelos aparecen en la escena con la orientación incorrecta. No se ha podido encontrar la causa de esta rotación indebida. Además los personajes que aparecen en la escena después de la carga inicial no se muestran en pantalla, como por ejemplo los disparos de las naves.

Los modelos se mueven como les indica el videojuego

La posición de los modelos que representan los personajes en la escena se actualiza correctamente a lo largo del tiempo tal y como cambian en el transcurso de la partida. Esto se ha conseguido gracias a las extensiones de los nodos de *OSG* que conectan los

objetos `CCharacter` de *UGK* con el grafo de escena, actualizando sus atributos en cada recorrido de éste.

La cámara de *OSG* se coloca en la posición indicada por el videojuego con los parámetros correctos

Se ha conseguido crear una cámara de *UGK* e introducirla en el grafo de escena mediante la extensión del nodo `osg::Camera`. Sin embargo no se ha logrado manipularla ni actualizar sus atributos durante el transcurso de la partida.

5.2 Revisión del plan de trabajo

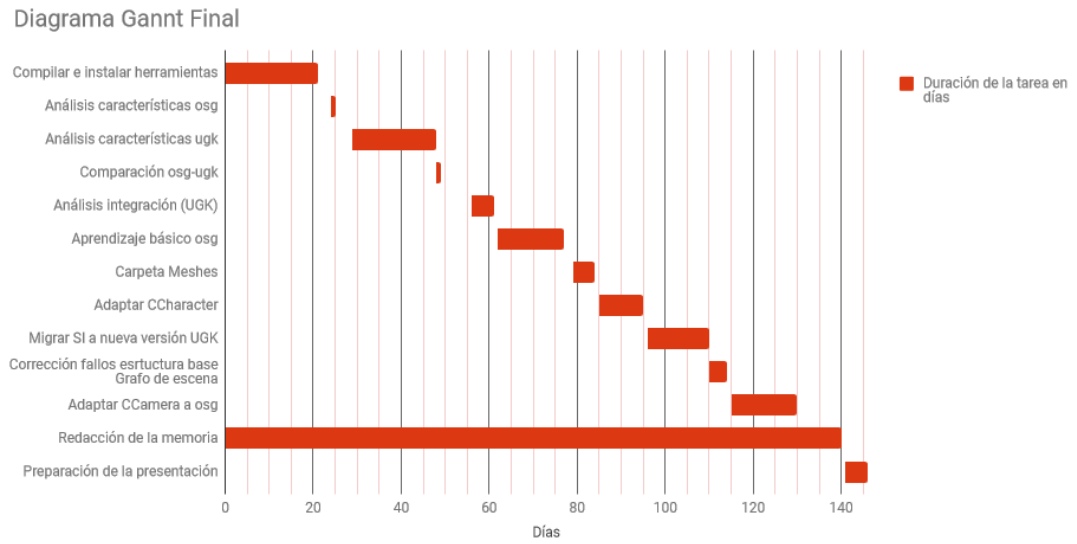

Figura 5.1: Progreso real del proyecto

Como se puede ver en la figura 5.1 la fase de preparación y análisis ha necesitado mucho más tiempo del esperado. Especialmente las tareas de compilación de las herramientas, análisis de *UGK* y el aprendizaje de *OSG*.
El retraso en la fase de compilación se debe a la poca familiaridad del autor en la compilación de grandes librerías usando *Visual Studio* y *CMake* y de sus numerosas opciones de configuración. Es por esto que se ha añadido el apéndice B: Instalación de *OSG* con la esperanza de que estas instrucciones le ahorren tiempo a alguien.
El análisis de *UGK* necesitó más tiempo del planeado debido a que se subestimó la duración de ésta. El mismo caso se ha dado con el tiempo necesitado para el aprendizaje del uso de *OSG*.
En la segunda fase surgieron dos problemas mayores y uno menor. El problema menor se refleja en la tarea "Corrección fallos estructura base del grafo de escena". Eso de debe a que se descubrió un fallo en el diseño inicial del grafo de escena que fue necesario corregir.
El primero de los retrasos mayores fue la adaptación de *Space Invaders* para que fuera compatible con la nueva implementación del grafo de escena. En este apartado fue necesario realizar más cambios de los previstos, como la eliminación del juego de los personajes implementados fuera de *UGK* o la creación de una nueva ventana para visualizar la escena de *OSG*. El segundo fue la modificación de las cámaras de *UGK*, que no consiguió finalizarse con éxito.

Estos retrasos han provocado que no se hayan podido realizar las modificaciones a las ventanas ni la implementación de las luces.

CAPÍTULO 6
Conclusiones

Aunque no se han cumplido todos los objetivos propuestos sí se ha alcanzado la meta principal del proyecto, que es adaptar *UGK* para que utilice el grafo de escena de *OpenSceneGraph* a través de una interfaz propia que permita el cambio entre implementaciones de forma sencilla en el futuro. Sin embargo una parte de la propia adaptación a *OSG* no se ha podido finalizar con éxito. El siguiente paso a seguir desde este punto sería finalizar la adaptación de *OSG*, esto es terminar la adaptación de las cámaras, el manejo y creación de ventanas y la implementación de luces en el motor. Tras esto se podría introducir otra librería de gráficos con el fin de probar que la interfaz creada en *UGK* es lo bastante genérica como para facilitar la adaptación de una librería distinta. Otro paso importante sería terminar la adaptación de *Space Invaders* para que funcione correctamente con la nueva versión de *UGK*.

CAPÍTULO 7
Propuestas para trabajos futuros

Aquí se presenta una lista de posibles proyectos que se pueden llevar a cabo tras la conclusión de éste.

7.1 Correcciones y adiciones

Como ya se ha mencionado en la conclusión algunos aspectos del nuevo grafo de escena no se han podido hacer funcionar correctamente en *UGK*, como por ejemplo la manipulación de las cámaras en la escena, el soporte para luces o la creación y el manejo de ventanas. Estas funcionalidades son necesarias si se desea que el motor se use en la creación de videojuegos por lo que sería recomendable terminar de implementarlas. Respecto a las ventanas se pueden tomar varias vías. Probablemente la más recomendable sería crear una abstracción para la creación de ventanas en *UGK* que permita usar múltiples interfaces dado que cada librería de gráficos tiene su sistema propio para la gestión de ventanas. Otra posibilidad es seguir usando la interfaz *WIN32* e integrar las ventanas de las librerías a esta. Esto es posible al menos con *OSG* pero se desconoce si el resto dispone de dicha capacidad.
Una funcionalidad que podría ser muy beneficiosa pero que se quedó fuera de este proyecto es el uso del propio grafo de escena para la detección de colisiones. Aunque se desconoce si todas las librerías disponibles que usan grafos de escena lo soportan se sabe que *OSG*, *OGRE3D* e *IRRLICHT* sí lo hacen. Esto permitiría prescindir de la estructura `Quadtree`, que básicamente es un duplicado del grafo de escena dedicado exclusivamente a la detección de colisiones.

7.2 Hacer UGK compatible con Linux y MacOSX

Por el momento *UGK* tan sólo es compatible con el sistema operativo *Windows*. La flexibilidad a la hora de elegir la plataforma es una cualidad muy importante en un motor de videojuegos por lo que este proyecto beneficiaría en gran medida al motor.

7.3 Añadir nuevas librerías gráficas

La finalidad de este proyecto ha sido permitir el cambio entre librerías gráficas con facilidad por lo que el siguiente paso lógico sería añadir soporte para otra de las librerías analizadas en la sección de Estado del arte. Las librerías recomendadas por el autor para realizar el siguiente paso son *OGRE3D* e *IRRLICHT* dado que son las que presen-

tan mejores capacidades y disponen de buena documentación y comunidades grandes y activas.

7.4 Finalizar la adaptación de Space Invaders

Una buena parte de los problemas que se han encontrado en el proceso de transición al nuevo grafo de escena se deben a problemas con el código del propio videojuego, especialmente con los personajes creados fuera de la estructura proporcionada por *UGK*. Para que *Space Invaders* sea jugable con la nueva versión de *UGK* va a ser necesario más trabajo para adaptar el código del videojuego.

Bibliografía

[1] Rui Wang, Xuelei Quiang. *OpenSceneGraph Begginer's Guide*. PACKT Publishing, 2010

[2] Página web del proyecto OSG. Consultado en http://www.openscenegraph.org/ 12/06/2018

[3] Página web del proyecto OGRE3D. Consultado en https://www.ogre3d.org/ 12/06/2018

[4] Página web del proyecto Horde3D. Consultado en http://www.horde3d.org/ 12/06/2018

[5] Página web del proyecto OpenSG. Consultado en https://sourceforge.net/projects/opensg/ 12/06/2018

[6] Archivo que contiene la ya cerrada página de OpenGL Performer. Consultado en https://web.archive.org/web/20071224141002/http://www.sgi.com/products/software/performer/ 12/06/2018

[7] Referencias y tutoriales para la programación en C++. Consultado en http://www.cplusplus.com 15/04/2018

[8] Referencias y tutoriales para el uso de Visual Studio. Consultado en https://msdn.microsoft.com/en-us/library/dn762121.aspx 15/04/2018

APÉNDICE A
Historia de UPVGameKernel

En el curso 2008-2009, arrancó la asignatura optativa Animación por Computador y Videojuegos (ACV) en el máster de Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital (MIARFID) dependiente del Departamento de Sistemas Informáticos y Computación (DSIC).
Con el fin de poder realizar las prácticas de la asignatura, se partió de un videojuego ya existente denominado Space Invaders *OpenGL*. Este videojuego sigue estando todavía disponible en *Source Forge* (https://sourceforge.net/projects/spaceinvadersgl/)
Este videojuego fue registrado en este dominio el 20 de febrero de 2003 y la última actualización es del 9 de abril de 2013, por lo que ha quedado descatalogado. Space Invaders OpenGL es un clon 2D / 3D del videojuego original de Taito Corporation lanzado comercialmente en el año 1978 denominado Space Invaders. S.I. OpenGl se desarrolló empleando directamente el API OpenGL. Está desarrollado íntegramente en C / C ++ sobre sistema operativo Windows.
Las primeras experiencias de desarrollo de contenido en las prácticas de la asignatura ACV fueron refactorizando el código e imponiendo cierto orden en el interior del proyecto. El código se fue fragmentando en más clases, ordenándose de forma jerárquica, extrayendo funcionalidades comunes, ... de forma que se realizó un directorio al que se le denominó API en el que se iban sacando funcionalidades que pudieran ser necesarias en otros videojuegos y que también se empleó en el S.I. De hecho, un Proyecto Fin de Carrera denominado Hyperchess, del año 2009, fue uno de los primeros videojuegos en el que se empleó la clase *parser* tanto de inicialización como de nivel fuera del contexto del S.I. Pero la incorporación de nuevas funcionalidades era muy farragosa y siempre pasaba a través del profesorado de la asignatura, por lo que se vio la necesidad de migrar el videojuego a un servidor de control de versiones. De esta manera, las aportaciones de los alumnos quedarían disponibles para años siguientes, pudiendo ir ampliando la funcionalidad y mejorando el código año tras año. Hay que tener en cuenta que se partía de un *spaguetti code* en el que se había empleado tanto texto en inglés como en francés y en el que coexistían tanto código en lenguaje C como C++. La primera actualización del código inicial de este videojuego data del 19 de octubre del año 2011.
La primera vez que se realizó el trasvase de funcionalidad desde el videojuego S.I. a este API interna fue el 11 de octubre de 2012, un año después de haber comenzado con la refactorización del videojuego original de Source Forge. Poco a poco se fueron incorporando nuevas funcionalidades como la gestión de ventanas, mejoras de parseado, gestión de texturas, mallas, periféricos, funcionalidades comunes a una clase básica denominada *Character*, gestión de sonidos, detección de colisiones, álgebra, gestión del tiempo, incorporación de mecanismos de comunicación internos entre los objetos, mensajería,... y en general todo tipo de funcionalidad propia de un videojuego.
En el transcurso, comenzó la asignatura Introducción a la Programación de Videojuegos

(IPV) en el grado de informática dependiente de la Escuela Técnica Superior de Ingeniería en Informática (ETSIInf) que requería de un videojuego sobre el que poder desarrollar unas prácticas. El S.I. fue su candidato. Al año siguiente, ACV desaparecía del máster MIARFID y en su lugar aparecía la asignatura Motores de Videojuegos, lo cual obligaba a tener un motor que tocara todos los puntos que se emplean en un videojuego, versátil, que enmascarara a todos los motores empleados por debajo y que mostrara la trastienda de un motor de videojuegos empleado por un videojuego real (IPV).
Pronto se vio la necesidad de ir pasando el uso de las APIs externas que se empleaban en el proyecto al API interna e ir enmascarándolas debajo de una capa propia que independizara el proyecto de un API en concreto. Así, cada alumno podía emplear el API que más le gustara, sin necesidad de tener que tocar el código original del videojuego. No hacerlo así significaba disponer de diferentes versiones del juego no compatibles entre sí, emplear compilación condicional que ofuscaba el código y lo hacía farragoso de entender y lo alejaba de su uso original pedagógico.
Cuando la cantidad de funcionalidad comenzó a ser elevada, se vio la necesidad de extraer dicha funcionalidad a un API externa al videojuego de forma que realmente fuera un proyecto diferente y no estuviera dicha API fuertemente condicionada por el único videojuego al que servía. A este API se le denominó UPV Game Kernel (UGK). La primera incorporación de código al UGK data de mayo de 2014.
Actualmente este API se emplea para el desarrollo de TFG, TFM, prácticas en una asignatura de grado y otra de máster.

APÉNDICE B
Instalación de *OpenSceneGraph*

Aquí se listan los pasos necesarios para instalar y usar la librería de *OpenSceneGraph* (OSG) en *Windows* usando *VisualStudio* y *CMake*.

B.1 Estructura de archivos

Aunque no es necesario sí es muy recomendable para facilitar el resto del proceso organizar los archivos de OSG del siguiente modo:

1. Crear una carpeta raíz llamada *OSG-VERSION* dónde *VERSION* es el número de versión de OSG.

2. Dentro de la carpeta raíz crear tres carpetas: *3rdParty*, que contendrá las dependencias, *OpenSceneGraph*, donde residirá el código fuente de la librería y *build*, dónde se instalará el resultado de la compilación para ser usado en otros proyectos.

B.2 Generar archivos de proyecto y solución

Para poder compilar el código fuente primero hay que crear un archivo de solución para *VisualStudio* usando *CMake* siguiendo estos pasos:

1. Iniciar *CMake*

2. Introducir en los campos *Where is the source code* y *Where to build the binaries* la dirección de la carpeta *OpenSceneGraph* y hacer click en el botón *configure*.

3. Esto mostrará una ventana dónde se elige para qué programa y versión generar el proyecto y los bits de la máquina de destino. Es muy importante elegir éstos correctamente.

4. Entonces se mostrará una lista de opciones, la mayoría de las cuales no es necesario cambiar. Es muy importante cambiar la dirección de la opción *CMAKE_INSTALL_PREFIX* a la dirección de la carpeta *build* Si se desea compilar los ejemplos hay que activar la opción *BUILD_OSG_EXAMPLES*.

5. Una vez se tenga la configuración deseada sólo queda clicar el botón *Generate*. Esto generará los archivos del proyecto en la carpeta especificada en *Where to build the binaries*.

B.3 Compilar en Visual Studio

En la carpeta de destino debería haber ahora un archivo llamado *OpenSceneGraph.sln*.

1. Abrir *OpenSceneGraph.sln* en *VisualStudio*.
2. Una vez abierta la solución, compilar el proyecto *ALL_BUILD*. (Esto llevará entre treinta y cuarentaycinco minutos dependiendo de la potencia del ordenador).
3. Compilar el proyecto *INSTALL*. Esto instala la librería compilada en la carpeta indicada en el campo *CMAKE_INSTALL_PREFIX*. Esta es la carpeta que habrá que referenciar para usar OSG en otros proyectos.

APÉNDICE C
Diccionario

Axis Aligned Bounding Box (AABB) Caja que representa el volumen ocupado por un objeto en la escena. Los vértices de esta caja son paralelos a los ejes del marco de referencia global.

Callback Función que se pasa como argumento a otra función con el fin de que se ejecute en un momento dado. Normalmente en el caso de que se dé un evento concreto o se cumpla cierta condición en el estado del programa.

Curvas de Bezier Curva paramétrica definida por un conjunto de puntos.

Graphics Processing Unit GPU Componente de un computador especializado en el procesado y generación de imágenes.

Motor de videojuegos Conjunto de herramientas y código diseñados para facilitar el desarrollo de videojuegos.

Oclusión ambiental Técnica de renderizado para calcular cómo de expuesto está un punto de la escena a la luz ambiental.

Parser Programa que analiza e interpreta texto para convertirlo en instrucciones que el computador pueda entender.

Renderizado Proceso de generación de imágenes mediante un programa informático a partir de la definición de una escena.