UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Normalización y uso del UPV Game Kernel

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Luís Taverner Sanmartin

Tutor: Ramón Pascual Mollá Vayá

2021 / 2022

Resumen

Una de las tareas más exigentes a las que un programador sin experiencia debe enfrentarse en el mundo real es trabajar con un repositorio de código de tamaño considerable. Leer y entender grandes programas diseñados por otras personas es una tarea ardua, introducir cambios sin degradar la calidad del proyecto y respetando su filosofía de diseño lo es incluso más.

UPV Game Kernel (UGK) es un proyecto de creación de un motor de videojuegos que pretende ser pedagógico, utilizable como banco de trabajo para soporte de prácticas en clase y como recurso a emplear en trabajos de fin de grado relacionados con los videojuegos.

Este trabajo de fin de grado tiene como objetivo normalizar y mejorar las interfaces de los diferentes módulos ya existentes en el motor de videojuegos UGK. Las interfaces existentes han sido analizadas identificando deficiencias de diseño que puedan derivar en dificultades para poder ser utilizadas de forma correcta. Además de abordar los problemas de diseño de las interfaces, se identifican e implementan funcionalidades que los usuarios necesitan para poder desarrollar sus videojuegos y que estaban ausentes en UGK.

Palabras clave: videojuego, juego, C++, Motor de Videojuegos, gestión de memoria, UGK , patrones, interfaces.

Resum

Una de les tasques més exigents a les quals un programador sense experiència ha d'enfrontar-se al món real és treballar amb un repositori de codi de grandària considerable. Llegir i entendre grans programes dissenyats per altres persones és una tasca difícil, introduir canvis sense degradar la qualitat del projecte i respectant la seva filosofia de disseny ho és fins i tot més.

UPV Game Kernel (UGK) és un projecte de creació d'un motor de videojocs que pretén ser pedagògic, utilitzable com a banc de treball per a suport de pràctiques a classe i com a recurs a utilitzar en treballs de fi de grau relacionats amb els videojocs.

Aquest treball de fi de grau té com a objectiu normalitzar i millorar les interfícies dels diferents mòduls ja existents al motor de videojocs UGK. Les interfícies existents han sigut analitzades identificant deficiències de disseny que pugin derivar en dificultats per poder ser utilitzades correctament. A més d'abordar els problemes de disseny de les interfícies, s'identifiquen i implementen funcionalitats que els usuaris necessiten per poder desenvolupar els seus videojocs i que estaven absents a UGK.

Paraules clau: Videojoc, joc, C++, Motor de Videojocs, gestió de memòria, UGK , patrons, interfícies.

Abstract

One of the most daunting tasks an inexperienced programmer must face in the real world is dealing with a legacy codebase of considerable size. Reading and understanding a big program designed by other developers can be hard, applying changes without degrading the quality of the project and respecting the original design philosophy can be even harder.

UPV Game Kernel (UGK) is a game engine creation project that aims to be a pedagogic tool, fit to be used as a workbench at laboratory classes and to be used for undergraduate final projects in which video games are developed.

This project aims to standardize and improve the interfaces of the different modules already present in the UGK videogame engine. The existing interfaces have been analyzed to identifying design deficiencies that may lead to difficulties in order to use them correctly. In addition to tackling interface design problems, functionalities that users need to implement their own videogames and that were absent in UGK are identified and implemented.

Keywords : videogame , game , C++, Game engine, memory management, UGK , patterns, interfaces.

Índice

Índice de figuras

1 Introducción

En los últimos diez años la barrera de entrada al desarrollo de videojuegos se ha reducido considerablemente.

Esto se debe, principalmente, a tres factores: plataformas de distribución digital, redes sociales y la disponibilidad de motores de videojuegos comerciales de forma gratuita.

Las plataformas de distribución digital ofrecen a los desarrolladores una infraestructura capaz de generar ingresos monetarios suficientemente voluminosos como para convertir el desarrollo independiente en una ocupación viable.

Las redes sociales permiten a los autores publicitar su producto haciéndolo llegar al mayor número de personas posibles, además democratizan el acceso a los conocimientos necesarios para el desarrollo de videojuegos, propician el encuentro entre personas con intereses comunes que acaban formando grupos de desarrollo y proporcionan soporte emocional, fundamental en el proceso de creación de un videojuego.

Las plataformas de distribución digital combinadas con las redes sociales dotan a los individuos de una infraestructura de distribución que les permite competir con las empresas tradicionales. El uso de esta infraestructura requiere una inversión o riesgo iniciales prácticamente nulos.

El tercer factor, tema de este trabajo, son los motores de videojuegos.

La gran evolución de los motores de videojuegos modernos no es el motor en sí mismo, si no las herramientas que lo rodean.

La construcción de un videojuego es un proceso abierto e iterativo, cuando más ágil sea el proceso de diseño y más fácil sea introducir cambios, mejor calidad se obtendrá en el resultado final, pues los artistas y diseñadores necesitan libertad y tiempo para desarrollar su trabajo.

El éxito de los motores gratuitos actuales como *Unity*[1] o *Unreal Engine*[2] radica en que proporcionan todas las herramientas necesarias para que un artista o un miembro creativo del equipo sin conocimientos técnicos pueda crear contenido de una forma fácil con una curva de aprendizaje suave, reduciendo la necesidad de aprender instrumentación específica, la cual resulta difícil de utilizar por personas sin conocimiento técnico.

En lo que concierne a los programadores, los motores de videojuegos permiten centrarse en problemas de alto nivel teóricamente más complejos que añaden el verdadero valor al producto final, como pueden ser, por ejemplo, algoritmos relacionados con las mecánicas de juego.

Pero, a pesar de todas las ventajas que ofrecen los motores comerciales, no se debe de perder de vista las siguientes cuestiones: son sistemas de gran complejidad y hacen

[1] https://unity.com/
[2] https://www.unrealengine.com/

dependientes al equipo de desarrollo del conocimiento y la propiedad intelectual de una entidad externa.

Dentro de la Universidad Politécnica de Valencia, se ha desarrollado un motor de videojuegos con orientación pedagógica llamado UPV Game Kernel y abreviado como *UGK*.

UGK tiene la finalidad de permitir a sus usuarios desarrollar videojuegos y aplicaciones gráficas interactivas. El entorno de trabajo abstrae y gestiona los problemas de bajo nivel para que el usuario pueda centrarse en el diseño de la lógica de la aplicación. Se utiliza también como banco de pruebas para investigar e implementar algoritmos gráficos, trabajos de fin grado y tesis de máster.

1.1 Motivación

Las motivaciones detrás de este trabajo surgen, por un lado, del interés personal del autor en comprender la organización interna de un videojuego, y, por otro lado, de las necesidades de mantenimiento que han ido apareciendo según el proyecto UGK ha envejecido y crecido a lo largo de los años. De una forma más concreta, las motivaciones de este proyecto quedan resumidas en los siguientes puntos:

- UPV Game Kernel necesitaba ser revisado para identificar sus deficiencias y proponer soluciones a las mismas.
- Los motores comerciales convierten el proceso de desarrollo en una caja negra, hay que seguir enseñando y ejercitando las técnicas necesarias para desarrollar un videojuego para no ser en el futuro totalmente dependientes de la tecnología privada de empresas externas.
  Los problemas que debe de solucionar un motor de videojuegos requieren de experiencia y la toma de muchas decisiones de diseño.
  Al ser estos problemas tan amplios y generalistas es difícil encontrar soluciones concretas en la literatura existente.
  Si la universidad consiguiera transmitir estos conocimientos sobre programación y diseño de software a los alumnos mediante una herramienta propia, el alumno obtendría una ventaja respecto a los estudiantes de otros centros.
- UPV Game Kernel o futuros desarrollos propios de la Universidad Politécnica de Valencia se pueden utilizar como herramienta de aprendizaje mucho más accesible que los motores populares, que son mucho más complejos y opacos para los alumnos de grado.

1.2 Objetivos

Los objetivos propuestos a continuación tienen como finalidad subsanar las deficiencias del programa que amenazan la viabilidad y continuidad del proyecto, y aquellas que impiden a los usuarios emplear UGK de forma satisfactoria.

- Analizar las deficiencias en la gestión de memoria y evaluar soluciones para solventarlas. Esto incluye tanto identificar errores existentes en la implementación del motor como deficiencias en las interfaces que propician que el usuario cometa errores de gestión de memoria en las aplicaciones.
- Analizar estructura interna del motor del videojuego, junto con sus dependencias externas para detectar problemas potenciales, mejoras a introducir y ordenarlas por prioridad de impacto en la usabilidad del motor por parte de los desarrolladores de videojuegos
- Identificar funcionalidades necesarias pero que no existen en el motor, las cuales los usuarios se vieron forzados a implementar en aplicaciones desarrolladas en el pasado.

1.3 Estructura de la memoria

Hemos considerado que, dada la naturaleza dispersa y heterogénea del conjunto de problemas abordado en este proyecto, es conveniente utilizar una estructura diferente a la de un trabajo de fin de grado habitual. Este documento se organiza comenzando por una introducción común para toda la obra, esta introducción contiene la motivación y objetivos a alcanzar por este trabajo y una exposición y crítica al estado del arte. A continuación de esta parte inicial común, el trabajo diverge en una serie de subapartados que abordan cada uno de los subproblemas que componen este proyecto. Cada uno de los subapartados contienen un análisis del problema, seguido de una exposición de sus posibles soluciones de entre las cuales se seleccionará una de ellas, seguidamente se documentará el diseño, desarrollo, implementación y resultados de la solución elegida para solventar el subproblema. El documento finaliza exponiendo una conclusión común para todo el trabajo y proposición de posibles trabajos futuros. Esperamos que esta estructura modular, al proporcionar independencia entre las distintas partes de la obra, permita al lector localizar y adquirir la información que sea de su interés de una forma rápida y que le resulte de la máxima utilidad posible.

2 Estado del arte

Según ha incrementado la potencia del hardware disponible y el presupuesto para el desarrollo de las empresas, los motores de videojuegos han aumentado en tamaño y complejidad. Los motores actuales incorporan muchos módulos y componentes complejos diferentes. Hoy en día es habitual emplear componentes licenciados de terceros en vez de realizar todo el desarrollo de manera interna.

Entre los subsistemas más sofisticados que conforman los motores actuales, destacan el motor de renderizado, el sistema de físicas y el sistema de inteligencia artificial. Estos módulos están claramente delimitados y resuelven problemas pertenecientes a dominios concretos, que requieren gran conocimiento experto en cada área de interés. A continuación, se expone brevemente en qué consiste cada uno de ellos.

2.1.1 Sistema de Físicas

Gracias a este sistema las entidades que forman parte del mundo del juego son capaces de interactuar entre ellas de una forma realista y dinámica.

Este componente es el responsable de la animación y el comportamiento de gran parte de los detalles de las escenas de un videojuego. Permite la simulación de cuerpos sólidos, cuerpos blandos y también permite que los objetos intercambien energía cinética entre ellos y estén sujetos a fuerzas como la fricción o la gravedad.

Esta parte del motor también se encarga de simular, detectar y resolver la colisión entre objetos.

Los sistemas de físicas utilizados en los videojuegos priorizan el rendimiento por encima de la exactitud y la rigurosidad. Es suficiente con que el resultado sea convincente a los ojos del jugador [1].

Uno de los desafíos a la hora de diseñar un sistema de estas características es que resulte estable. La credibilidad y el aspecto del mundo se resienten si los objetos tienden a comportarse de formas extrañas como, por ejemplo, salir despedidos a grandes velocidades después de colisionar unos con otros o cuando permanecen en un estado de vibración incesante.

Actualmente este componente no suele desarrollarse internamente y es común emplear software de terceros para implementar esta parte del motor, siendo algunos de los motores de físicas más conocidos Havoc[3], Bullet[4] y PhysX[5].

[3] https://www.havok.com/
[4] https://github.com/bulletphysics/bullet3
[5] https://developer.nvidia.com/gameworks-physx-overview

2.1.2 Motor de renderizado

Este sistema se encarga de producir las imágenes que visualiza el jugador por pantalla, para lograr este objetivo el motor de renderizado administra y controla el hardware de aceleración gráfica y sus recursos.

Las tarjetas gráficas actuales son sistemas de procesamiento paralelo que toman como entrada una serie de listas de datos y producen como resultado imágenes bidimensionales. La manera en la que se combinan los datos de entrada para generar un valor de color para cada píxel es definida por el programador mediante programas llamados *shaders*. Estos programas son ejecutados por el procesador gráfico en diferentes etapas del proceso de ensamblado de la imagen.

Para conseguir un rendimiento satisfactorio con una tasa de imágenes por segundo elevada el motor de renderizado debe de gestionar eficientemente los recursos del hardware de vídeo.

Hoy en día los videojuegos emplean gran cantidad de técnicas de iluminación complejas y efectos de postprocesado diferentes. Conseguir que todas estas técnicas y efectos funcionen de forma conjunta es una tarea complicada, muchas veces es necesario realizar varias pasadas de renderización y combinar una serie de imágenes intermedias para conseguir el resultado final que acabará visualizando el usuario. El motor de renderizado es la parte del programa donde se implementan todos estos algoritmos gráficos.

2.1.3 Sistema de Inteligencia Artificial

El sistema de inteligencia actual es el encargado de actualizar el estado lógico de las entidades del juego.

En el pasado, el comportamiento de las entidades de un juego, como personajes no jugables o enemigos, se implementaban utilizando máquinas de estado simples y patrones de movimientos preestablecidos.

En la actualidad, los consumidores esperan que las entidades que habitan el mundo virtual se comporten de una manera mucho más compleja. Los enemigos de un videojuego realista actual son capaces de reaccionar a los cambios en su entorno y a las acciones del jugador.

Existen variedad de algoritmos y soluciones que pueden ser utilizadas para implementar el sistema de inteligencia artificial de un videojuego [2], como, por ejemplo, lógica fuzzy o arboles de decisión; este último es utilizado por la serie Halo [3], de Bungie Inc.

A la hora de diseñar el comportamiento de la inteligencia artificial siempre predomina, por encima del realismo y la exactitud, la necesidad de crear una experiencia de juego satisfactoria y conseguir un rendimiento elevado.

2.1.4 Apuesta por la Importación de recursos

En un videojuego, si bien la infraestructura tecnológica es indispensable, no es el único elemento que contribuye al valor del producto final. La calidad de los apartados artísticos, la calidad del diseño de las mecánicas del juego y el volumen de contenido posible se incrementan durante el desarrollo si las herramientas de software permiten al personal creativo del proyecto la posibilidad de experimentar y modificar los recursos del juego constantemente con un coste reducido en tiempo y esfuerzo. Por este motivo, los motores de videojuegos actuales están siendo orientados a la creación de contenido y a los autores de este.

En el pasado, debido a las limitaciones técnicas y a la falta de estandarización, muchos estudios de videojuegos decidían diseñar y utilizar sus propios formatos propietarios para almacenar los recursos de los juegos. Este enfoque requería desarrollar programas independientes del código del motor, que debían de procesar todos los recursos y convertirlos a los correspondientes formatos propietarios.

En el plano técnico, esta manera de manejar los datos suponía un coste de mantenimiento considerable, pues los programas de conversión debían de ser desarrollados por los programadores del proyecto. Si el motor soportaba varias plataformas esto implicaba escribir varias colecciones de programas de conversión, una por cada plataforma diferente. Esta manera de trabajar hacia difícil acomodar cambios, una modificación en la estructura de un formato interno utilizado por el motor conllevaba reescribir los programas de conversión para ese formato.

Para los artistas este enfoque también resultaba problemático, pues estos dependían de los programadores del juego y de entornos de desarrollo automatizados mediante scripts para poder introducir los recursos en el juego. Para poder visualizar y probar el resultado final de su trabajo, los artistas se veían forzados a aprender a utilizar herramientas complejas y de carácter técnico que limitaban su capacidad creativa. También se daba el caso de que algunos formatos propietarios del juego debían editarse con programas internos de la empresa, inferiores en calidad y más difíciles de utilizar que los programas de arte populares a los cuales los artistas estaban acostumbrados.

En la actualidad, los motores de videojuegos están optando por un modelo de importación de recursos [4]. Este modelo consiste en soportar los formatos más populares y comunes de manera que estos archivos puedan ser cargados directamente por el videojuego sin necesidad de programas externos. Esto permite a los artistas utilizar los programas a los que ya están acostumbrados, con los cuales son más productivos y les evita tener que perder tiempo en cuestiones técnicas. Los motores y las herramientas de desarrollo que se construyen siguiendo esta filosofía suelen permitir importar recursos empleando interfaces intuitivas dentro de los editores de contenido del juego.

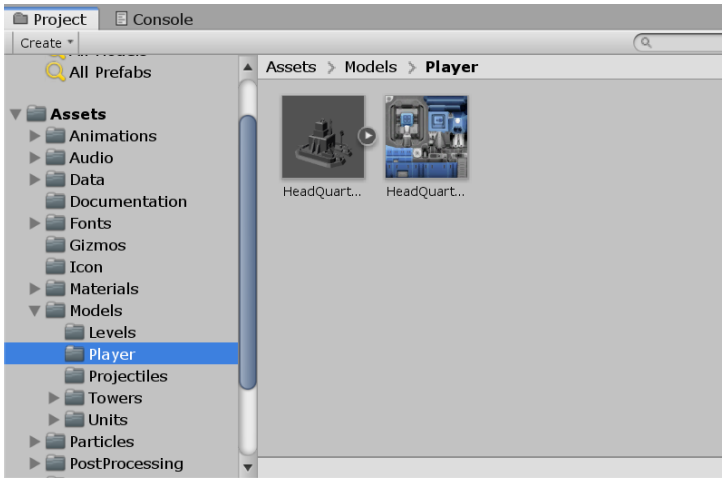

Ilustración 1 Unity Project View

Utilizando como ejemplo Unity, se puede ilustrar cómo funciona el proceso de importación en una herramienta actual. Para importar recursos dentro de Unity simplemente se debe de copiar los ficheros dentro de la carpeta de recursos del proyecto y se importaran de forma automática. Unity también es capaz de detectar las modificaciones y vuelve a importar los archivos modificados de forma autónoma. La interfaz del editor de niveles cuenta también con el menú Project View, un explorador de ficheros que permite administrar y utilizar recursos de forma intuitiva.

3 Análisis de UGK

El análisis ha sido realizado desde el punto de vista de un usuario, esto es, asumiendo el rol de un usuario final se han identificado las deficiencias y los puntos a mejorar de UGK que suponen un mayor contratiempo a la hora de desarrollar un videojuego empleando la herramienta.

La exploración e identificación de los problemas que afectan a los usuarios se ha realizado principalmente utilizando los siguientes dos métodos: El primero, se ha analizado el código de juegos anteriores desarrollados con UGK; el segundo, se han desarrollado juegos propios con la herramienta.

Dentro del primero método, se ha analizado el código de dos videojuegos desarrollados previamente con UGK: *HyperChess* [5] y *Space Invaders*.

Gracias a este método, se identificaron funcionalidades de las cuales UGK carecía, y que eran indispensables para desarrollar una aplicación. La necesidad de incluir estas funcionalidades en UGK resultó evidente cuando se observó como los usuarios se veían forzados a realizar sus propias implementaciones de las mismas áreas de código en cada uno de sus videojuegos. Dichas áreas de código no realizaban tareas relacionados con la lógica de la aplicación si no con la interacción y la gestión de recursos mediante llamadas al sistema o librerías de terceros incluidas en UGK.

Entre las funcionalidades ausentes más relevantes, se detectó la inexistencia de un gestor del bucle principal del juego y la carencia de un módulo de gestión de entrada que se encargara de leer el input generado por el jugador (teclado, ratón y joystick).

Gracias al segundo método, desarrollo de videojuegos propios utilizando UGK, se descubrieron las dificultades a las cuales un usuario debe enfrentarse a la hora de comenzar un nuevo proyecto.

Entre otros problemas, se observó como la falta de un archivo de configuración global de UGK dificulta la distribución del motor en forma de librería precompilada, lo cual supone un gran inconveniente para los usuarios.

En los siguientes subapartados se exponen en detalle las deficiencias que han sido descubiertas como resultado de este análisis y que hemos considerado que deben ser solventadas de manera prioritaria.

Los problemas identificados en el análisis realizado y que no han sido abordados en este proyecto por limitaciones de ámbito, tiempo o por considerar que cuentan con una prioridad menor, serán enumeradas y descritas brevemente en el apartado Trabajos Futuros.

3.1 Gestión de memoria dinámica

3.1.1 Análisis del problema

Una de las funcionalidades más comunes con la que cuentan los lenguajes de programación modernos son los colectores de basuras. Un colector de basuras es un proceso, perteneciente al entorno de ejecución del lenguaje, que se ejecuta en segundo plano y recupera los bloques de memoria que el programador ya no está utilizando. La presencia de un colector de basuras elimina la necesidad de gestionar la memoria de forma manual.

El lenguaje utilizado por UGK, C++, no cuenta con un colector de basuras y por lo tanto la responsabilidad de gestionar la memoria recae sobre el programador.

A la hora de gestionar memoria de forma manual, los problemas que hay que evitar son los siguientes:

- Utilizar regiones de memoria después de haber sido liberadas.

  Si esto sucede, en el mejor de los casos nuestro programa simplemente finaliza. En el peor de los casos el programa continúa ejecutándose y las regiones de memoria que han sido corrompidas dan pie a errores difíciles de reproducir y depurar.

- Usar regiones de memoria antes de solicitarlas.

  Misma problemática que la expuesta en el punto anterior.

- No liberar memoria al acabar de ser utilizada, fuga de memoria.

  En esta situación, la memoria utilizada crece de forma continua hasta consumir la totalidad del espacio disponible.

Propietarios de la memoria, Semántica de los punteros.

El propietario de un objeto es el bloque de código sobre el cual recae la responsabilidad de liberar la memoria del objeto cuando este deje de utilizarse. La propiedad del objeto puede transferirse mediante operaciones de asignación, llamadas a funciones y también utilizando los valores de retorno de una función [6].

Unos de los principios importantes a la hora de gestionar la memoria de un programa de manera correcta es determinar quién es el propietario de cada objecto [7]. A la hora de escribir un bloque de código, el programador debe de identificar que objetos son propiedad del bloque y cuales no lo son. Si el programador comete una equivocación a la hora de identificar quien es el dueño de un objeto, aparecen errores que causan perdidas de memoria o comportamiento indefinido del programa.

El principal inconveniente a la hora de utilizar punteros tradicionales es que no tienen ninguna semántica asociada a la propiedad de la memoria. Los punteros no documentan ni denotan quien es el propietario de la memoria a la cual apuntan [8].

Observemos algunos ejemplos de este problema en las interfaces públicas de UGK:

```
//Fichero UGKCharacter.h
class UGK_API CCharacter :public RTDESK_CEntity
{
	//(…)
	std::vector <CSound*> Sound;

	inline void SetSound(CSound * sound, unsigned int Index)
	{
	 Sound[Index] = sound;
	}

	inline CSound * GetSound(unsigned int Index)
	{
	    return Sound[Index];
	}

	//(…)

}/*fin CCharacter */
```

Ilustración 2 Ejemplo Interfaz con punteros

Como se aprecia en el código de este ejemplo, la clase `CCharacter` cuenta con un vector de punteros a objetos de la clase `Sound`.

Con esta interfaz el usuario no tiene forma de saber si la función `SetSound` adquiere la propiedad sobre el objeto apuntado por `sound`. No podemos saber tampoco si la función `GetSound` está transfiriéndonos la propiedad del objeto que nos está devolviendo. Este tipo de métodos que toman como argumento o devuelven punteros, y no cuentan con documentación adicional, son predominantes a lo largo de las clases que componen la interfaz pública de UGK. A la hora de utilizar interfaces como estas el usuario solo tiene dos opciones, o bien intentar gestionar la memoria de los objetos y arriesgarse a crear

errores en el programa, o bien asegurarse que el programa continue funcionando, dejando sin gestionar la memoria y permitiendo fugas de memoria.

Ausencia de automatización en la gestión de memoria.

Otra deficiencia presente en las clases de UGK es la ausencia de destructores que liberen las dependencias y los recursos cuando el objeto se destruye.

Esta vez utilizamos como ejemplo la clase `CPhyObject` :

```
//Archivo UGKPhysicsObjectUGK.h
/**
* Define the class with the information of a physics object:
* Geometry, material, linear velocity, angular damping
* and the transformations
*/
class CPhyObject: public CBasicPhyObject
{
	public:
	CPhyMaterial* material;

	CPhyObject(CPhyMaterial* m);
	/// Get the trasnformation Matrix of the physical object
	virtual void getTransformationMatrix(float* matrix) = 0;
	/// Turns the Plane
	virtual void setTurnPlane(float angle, float x, float y, float z) {};
};
```

Ilustración 3 Ejemplo clase sin destructor

Esta clase, contiene una dependencia de tipo `CPhyMaterial`. Como sucede de forma habitual en las clases de UGK, no existe un destructor que libere el objeto al cual apunta la variable `material`, por lo tanto, la creación de entidades físicas en el juego implica una pérdida de memoria por cada objeto creado. Además, al igual que el ejemplo anterior, no es posible determinar con certeza si efectivamente `CPhyMaterial` es el dueño del objeto `material`.

Inicialización en múltiples pasos.

En las clases de UGK donde se utiliza la composición de objetos mediante punteros, frecuentemente se permite que el usuario construya un objeto en un estado inconsistente. Inmediatamente después de construir el objeto, los punteros a sus componentes quedan sin inicializar o son inicializados a valores nulos. El resultado de llamar a los métodos de la clase sobre un objeto en estado inconsistente es el lanzamiento de excepciones o la corrupción de memoria.

3.1.2 Identificación de posibles soluciones

Estas son algunas de las maneras más comunes de solventar los problemas de gestión de memoria analizados en el apartado anterior o

Evitar la utilización de punteros y memoria dinámica.

Una buena forma de simplificar las interfaces y reducir los problemas de gestión de memoria es simplemente prescindir de los punteros donde sea posible. Si la penalización de rendimiento se puede asumir, utilizar referencias y realizar copias internas, o pasar los argumentos por valor.

En las clases que utilizan punteros como mecanismo de composición de objetos, pero no se pretende utilizar ningún tipo de polimorfismo, simplemente eliminar dichos punteros y remplazarlos por una instancia del objeto.

```
// UGKPhysicsObjectUGK.h
class CPhyObject: public CBasicPhyObject
{
	//(…)
public:
	CPhyMaterial* material;

	//(…)

}

// UGKPhysicsMaterialUGK.h

class CPhyMaterial
{
public:
	UGKALG_NUMERIC_TYPE staticFriction,
			    dynamicFriction,
			    restitution;
	//(…)
};
```

Ilustración 4 Ejemplo composición mediante punteros

Como se puede observar en este caso, `CPhyMaterial` no es más que una estructura con 3 campos, si se elimina el puntero y se hace que `CPhyObject` contenga un `CPhyMaterial`, se simplifica el diseño y se eliminan posibles problemas de gestión de memoria.

```
class CPhyObject: public CBasicPhyObject
{
public:
	CPhyMaterial material; /* CPhyObject ahora contiene un CPhyMaterial*/

}
```

Documentar mediante comentarios la propiedad de la memoria.

En todas las interfaces en las cuales se utilicen punteros, comentar de forma clara y precisa quien es el propietario de la memoria y que transferencias de propiedad se producen al utilizar una función o método.

```
class CPhyObject: public CBasicPhyObject
{
public:
#ifdef CD_USE_QUADTREE
	QuadtreeRoot		*QTRoot; /*unowned*/
#elif defined CHAR_USE_SWEEP_AND_PRUNE
#endif
	CPhyMaterial* material; /*owned*/
	/**
	@param m , (Transfer full ownership)
	*/
	CPhyObject(CPhyMaterial* m);
};
```

Ilustración 5 Interfaz con propiedad de memoria comentada

En este fragmento de código se está indicando mediante comentarios que el miembro `material` es propiedad de la clase `CPhyObject`, también se indica que la clase no es dueña del objeto `QuadtreeRoot` apuntado por `QTRoot`.
Los comentarios también documentan como al realizar la llamada al constructor de la clase se delega la responsabilidad de gestionar el tiempo de vida del argumento `CPhyObject *m` al código de la clase, por lo tanto, el usuario que realiza la llamada transfiere la propiedad del objeto y debe de abstenerse de gestionar la memoria de `CPhyObject *m` después de realizar la llamada.

```
	class UGK_API CMeshesManager
	{

		/**
		@brief return 3D mesh with id M
		@returns (transfer none), pointer to 3D mesh
		*/

		CMesh3d* GetMesh(unsigned int M);
	}
```

Ilustración 6 Interfaz con propiedad de memoria comentada (II)

Mediante este bloque de código se ilustra como denotar la transferencia de propiedad de un objeto devuelto por un método. En este caso el comentario indica al usuario que no es el propietario del objeto `CMesh3d` devuelto por la función `GetMesh`, pues la clase `CMeshesManager` tiene delegada la responsabilidad de cargar y descargar las mallas en 3D durante el tiempo de ejecución del programa.

Utilizar punteros inteligentes y RAII.

RAII, abreviatura inglesa de *"Resource acquisition is initialization"*, es un idioma de programación característico de C++ cuya finalidad es automatizar la gestión de los recursos del programa.

En C++, las clases cuentan con un método destructor el cual es ejecutado de forma automática antes de que se libere, o "destruya", la memoria asignada al objeto. Esta llamada al destructor se realiza de forma automática en el caso de los objetos locales. Cuando la ejecución de un programa llega al final de un bloque de código, se ejecutan todos los destructores de los objetos locales pertenecientes al mismo, esto sucede incluso cuando la salida del bloque se produce a causa de una excepción.

El concepto clave detrás del idioma RAII es ligar el tiempo de vida de un recurso al tiempo de vida de un objeto local, liberando el recurso en el destructor del objeto.
De esta manera se automatiza la gestión del recurso y se evitan filtraciones.
Los recursos no son solo memoria dinámica, también pueden ser archivos abiertos, conexiones a una base de datos y en general cualquier entidad que requiera ser adquirida y liberada.

```
//Ejemplo Clase que emplea el idioma RAII
//Gestion automatica de un fichero de texto
class SmartFile {
private:
	std::fstream fs;
	inline SmartFile() { ; } //prohibido crear una instancia sin archivo asociado
public:
	SmartFile(const char* name) { fs.open(name, std::fstream::out); }
	~SmartFile() { fs.close(); } //al destruirse se cierra el fichero
	void write(std::string text) { fs << text; }; //interfaz para escribir
};

int main() {
	{
		SmartFile foo("./foo"), bar("./bar");
		foo.write("foobar");
		bar.write("foobar");
	} //Al salir del bloque Se ejecutan los destructores de foo y bar de forma
	    automatica Incluso si se han generado excepciones dentro del bloque.
	return 0;
}
```

Ilustración 7 Ejemplo de clase RAII

Este ejemplo de código ilustra la estructura básica de un objeto implementado usando RAII. El recurso, en este caso un fichero, se adquiere en el constructor `SmartFile()` y se libera en el destructor `~SmartFile()` , por lo tanto, siempre que se creen instancias de la clase `SmarFile` como variables locales, los recursos utilizados por la clase se gestionarán de manera automática.

Los punteros inteligentes, del inglés *smart pointers*, son clases que envuelven y encapsulan a los punteros tradicionales. El adjetivo “inteligentes” proviene del hecho de que se implementan utilizando el idioma RAII, por lo tanto, la gestión de la memoria a la cual apuntan se realiza de forma automática.

En el código donde se emplean los punteros inteligentes, los propietarios de los objetos quedan indicados de manera explicita y se obliga al programador a tener presente que transferencias de propiedad se están realizando cuando se manipulan dichos objetos. Un programa que utiliza punteros inteligentes es menos propenso a los malentendidos a la hora de determinar a que parte del programa corresponde la responsabilidad de gestionar el tiempo de vida de cada objeto y por lo tanto se minimizan los errores de gestión de memoria.

La *Standard Template Library,* a la cual nos referiremos por sus siglas inglesas (STL), forma parte del estándar de C++ y proporciona una colección de punteros inteligentes que cubren las necesidades típicas de un programador a la hora de lidiar con los distintos modelos de propiedad de la memoria que pueden presentar los objetos de un programa.

Utilizando la STL, para los casos en los que el fragmento de código si que sea dueño y responsable de la memoria del objeto, utilizamos el puntero inteligente `std::unique_ptr`, que se encargara de liberar la memoria de forma automática.

En el caso de que el bloque de código no se sea dueño de la memoria del objeto, se pueden continuar utilizando punteros normales.

Queda aún por abordar el problema que supone mayor dificultad a la hora de gestionar el tiempo de vida de un objeto, la propiedad compartida de un objeto. Cualquier programa complejo tiene objetos que son utilizados por distintas partes del programa, y por lo tanto estos objetos compartidos solo deben de ser liberados cuando ya no sean alcanzables por ninguna parte del programa que los utilice. Este problema se soluciona utilizando la clase `std::shared_ptr`. Este puntero inteligente mantiene un contador de referencias, compartido por todas las instancias de `std::shared_ptr` que apuntan al mismo objeto. En el constructor de esta clase se incrementa el contador de referencias, y en el destructor se decrementa dicho contador y en caso de valer cero, se libera la memoria del objeto al que se está apuntado.
Utilizando `std::shared_ptr` obtenemos un mecanismos de gestión de memoria automatizado con una funcionalidad equivalente a la de un colector de basuras, como con el que cuentan lenguajes como Java o Phyton.

Anotar punteros utilizando Templates

Una solución intermedia entre refactorizar todo el proyecto o simplemente usar comentarios tradicionales, es anotar el código utilizando templates.

Las templates, o plantillas en español, se utilizan para implementar código genérico en C++. En resumidas cuentas, las templates son plantillas que el programador escribe y que el compilador utiliza para autogenerar código.

La idea consiste en escribir templates que, al ser evaluadas por el compilador, no generen ningún código adicional, el cometido de estas templates es documentar de forma explícita quien es el propietario de un puntero.

Este tipo de templates ya se encuentran disponibles en las diferentes implementaciones de la librería GSL, Guidelines Support Library.

Utilizando la librería GSL, podemos indicar como un puntero es propietario de la memoria a la que apunta, utilizaríamos la template `gsl::owner<T *>`

```
gsl::owner<CPhyMaterial *> a;

//El Código resultante generado por esta template es:

CPhyMaterial *a;
```

Ilustración 8 template sin efecto

En este ejemplo se puede ver como la template `gsl::owner` desaparece una vez evaluada por el compilador y el código resultante es simplemente la sentencia contenida entre los símbolos <>.

Con esta técnica se consiguen los siguientes beneficios frente al uso de comentarios tradicionales:

- Integración con la sintaxis de C++, las anotaciones resultan más naturales.
- Se evita que la documentación quede desactualizada respecto al código.

Podemos proporcionar información útil a las herramientas de análisis estático del código.

3.1.3 Solución propuesta

Refactorizar la totalidad del repositorio es inviable y no creemos que los punteros inteligentes se ajusten a la filosofía de diseño inicial con la que se desarrolló UGK. Todas las interfaces ya existentes en UGK deberían de ser documentadas utilizando comentarios de texto o templates para eliminar cualquier ambigüedad o malentendido que puedan sufrir los usuarios finales a la hora de gestionar la memoria de sus juegos.

Proponemos también que todas las interfaces nuevas que se añadan a UGK utilicen tanto punteros inteligentes como el idioma RAII para gestionar la memoria. También proponemos que los programadores que contribuyan al proyecto UGK en el futuro no deben de limitarse a utilizar punteros inteligentes sino también deben de prestar especial atención en la fase de diseño a identificar los propietarios de cada objeto almacenado en memoria y pensar como comunicar esta información a los usuarios finales de manera clara y efectiva.

En resumen, la solución propuesta sería mejorar la documentación de la propiedad de la memoria en el código antiguo y comprometerse a utilizar, en el código que se añada en el futuro, un estilo de C++ moderno propio de estándares actuales como C++11 y C++20, que se caracteriza por primar estrategias de gestión automática de memoria, dejando de lado la gestión de punteros de forma manual.

3.2 Gestión de la compilación condicional

3.2.1 Análisis del problema

Dentro del código de UGK se utiliza la compilación condicional de forma habitual.

El lenguaje C++ cuenta con un preprocesador, el cual toma como entrada el texto que conforma el código original del programa, según ha sido escrito por el programador, y realiza substituciones textuales, produciendo como salida el código final que utilizará el compilador para generar el código máquina.

Las sustituciones textuales realizadas por el preprocesador pueden ser controladas de forma condicional mediante la definición de constantes. Esta funcionalidad puede ser utilizada para excluir bloques de código del proceso de compilación. Esta técnica se conoce como compilación condicional.

```
/*Ejemplo de compilación condicional*/
/* La definicion de la constante "DECIR_FOO" determina que código se compila*/
#define DECIR_FOO
#ifdef DECIR_FOO /*Si DECIR_FOO esta definido */
       printf("FOO"); /*esta sentencia se compilara*/
#elif /*Si DECIR_FOO NO esta definido*/
       printf("BAR"); /*esta sentencia se excluye del programa y NO se compila*/
#endif
```

Ilustración 9: Ejemplo compilación condicional

Generalmente, se emplea el uso de directivas `#ifdef` para intentar solucionar los siguientes problemas en tiempo de compilación:

- Activar y desactivar funcionalidades del programa.
- Soportar diferentes plataformas y sistemas operativos.
- Seleccionar implementaciones alternativas.
- Excluir código de depuración y prueba en versiones de producción.

El Infierno #ifdef

El abuso de la compilación condicional tiene un impacto significativamente negativo en la calidad del código [9] [10].

Si no se utilizan con cautela, los bloques de código compilado condicionalmente comienzan a multiplicarse, a anidarse unos dentro de otros y además aparecen dependencias entre las condiciones lógicas de los bloques, lo que acaba degenerando en un código muy difícil de leer y mantener.

La principal finalidad con la que se recurre al uso de la compilación condicional en UGK es integrar diversas implementaciones alternativas encargadas de soportar diferentes sistemas operativos, algoritmos y librerías de software.

Para hacernos una idea de la cantidad de implementaciones alternativas existentes dentro de UGK listamos las categorías más significativas, junto a la colección de constantes que se pueden definir para cada una de ellas:

Implementaciones dependientes del sistema operativo: `UGKOS_WINDOWS`, `UGKOS_LINUX`, `UGKOS_OSX`, `UGKOS_ANDROID`

Implementaciones diferentes del sistema de físicas: `UGKPHY_PHYSX` `UGKPHY_ODE`, `UGKPHY_HAVOK`, `UGKPHY_BOX2D`, `UGKPHY_UGK`

Implementaciones diferentes del sistema de algebra lineal: `UGKALG_API_EIGEN`, `UGKALG_API_GLM`, `UGKALG_API_BLAS`, `UGKALG_API_FIXED_POINT`

Implementaciones diferentes grafos de escena: `UGKSG_OGRE` `UGKSG_UGK` `UGKSG_CRY` `UGKSG_IRL` `UGKSG_TORQUE` `UGKSG_OSG`

Sin ser este un listado completo de todas las opciones de compilación, solo con los casos mencionados estaríamos hablando de cuatrocientas ochenta versiones diferentes del mismo programa que deben de ser mantenidas y probadas.

Enlazando con UGK

Las definiciones de las interfaces públicas que expone UGK a sus usuarios, mediante archivos de cabecera, están afectadas por el fenómeno de la compilación condicional. Se puede identificar claramente el problema examinando el ejemplo de la .

UGKPhysicsObjectUGK.h - UGKPhysicsObjectODE.h

```
class CPhyObject: public CBasi
{
public:
#ifdef CD_USE_QUADTREE
        QuadtreeRoot        *QTRoo
#elif defined CHAR_USE_SWEEP_AND_PRUNE
#endif
        CPhyMaterial material;
```

```
class CPhyObject : CBasicPhyOb
{
public:
    CPhyMaterial* material;
    Vector dimension,
                Position,  ///< 3
                /// 3D space coor
                PositionPrev,
                /// 3D speed coor
                Speed,
                /// 3D accelerati
                Acceleration,
                /// 3D space orie
                Rotation,
```

Ilustración 10 Compilación condicional, implementaciones alternativas

Este ejemplo corresponde a la implementación de la clase `UGK::CPhyObject` , esta clase forma parte de la interfaz pública de la librería UGK y representa los atributos físicos de una identidad que forma parte de la simulación del juego.
Como se puede ver, tanto los miembros como los métodos que forman parte de la clase cambian según la implementación elegida, este problema se repite a lo largo de las clases que conforman las interfaces públicas de la librería.

Si el usuario no cuenta con una copia idéntica sin modificaciones de los archivos de cabecera con los cuales se compiló la versión de UGK que está utilizando, el proceso de enlazado fallará. Al no existir un archivo de configuración centralizado, el usuario no tiene forma alguna de saber con certeza como enlazar correctamente su aplicación. UGK actualmente no puede distribuirse como una librería precompilada, la única forma de enlazar UGK y la aplicación del usuario para generar un ejecutable, es que el propio usuario compile tanto su programa como el motor UGK.

Inteligibilidad del código fuente

La presencia de directivas `#ifdef` entrelazadas en el código lo hacen difícil de leer y mantener.

La lectura del programa se ve interrumpida por bloques de código inactivos que no tienen ninguna relevancia para el programador haciéndole perder tiempo. Los bloques inactivos además dificultan identificar la estructura lógica del código ya que menoscaban la efectividad de elementos estilísticos como la sangría, la agrupación y la ordenación de las líneas de código.

3.2.2 Soluciones propuestas

Dado el tamaño y años de desarrollo invertidos en UGK, las soluciones más viables son paliativas. el objetivo de aplicar estas soluciones es evitar que el problema continue expandiéndose e intentar mejorar la calidad del programa realizando refactorizaciones localizadas. Consideramos como inviables las soluciones que impliquen la reestructuración sistemática de todo el proyecto ya que la ausencia de tests automatizados convierte el proceso de refactorización demasiado complejo y propenso a generar errores.

Pasamos ahora a exponer las diferentes soluciones propuestas.

Archivo de configuración único

Para solucionar los problemas relacionados con el enlazado y la compilación, la solución más rápida y efectiva a corto plazo es centralizar todas las definiciones del preprocesador en un único archivo de cabecera, que contenga la configuración de todo el proyecto. Para implementar esta solución será necesario asegurarse de que las definiciones son únicas y no existen repeticiones en los archivos de cabecera, y que los programadores respetan y utilizan correctamente el archivo de configuración único.

Para que el archivo de configuración único sea una solución factible y funcional, se debe comprobar que cualquier modificación futura cumple esta convención y respeta la configuración global del proyecto.

Implementaciones Alternativas en diferentes ficheros

La idea clave de esta solución es separar cada variante de una implementación en su propio fichero de código, de manera que los bloques de código correspondientes a cada implementación alternativa no se encuentren entrelazadas en el mismo archivo. Esta solución tiene un impacto positivo en la legibilidad del código.

Organizando los ficheros del proyecto de esta manera se elimina la necesidad de utilizar directivas `#ifdef` y en su lugar se gestiona que archivos compilar utilizando el gestor de propiedades de Visual Studio, creando en las configuraciones que se necesite para el programa.

Reducir el número de variaciones posibles

El proceso de diseño de un programa consiste en tomar decisiones que limitan y acotan las opciones de implementación para poder manejar mejor la complejidad. Si el código cuenta con demasiadas implementaciones alternativas esto indica que se está evitando tomar decisiones de diseño. El coste de esta indecisión es elevado, pues cada variación existente del programa tiene que ser probada y mantenida.

Consideramos que las variaciones presentes en UGK son excesivas, no es factible mantenerlas todas y hacen difícil trabajar con la herramienta. Se debe de evaluar que implementaciones alternativas pueden ser eliminadas del proyecto para reducir el tamaño y la complejidad de este.

3.3 Bucle principal y gestor de escenas lógicas

3.3.1 Análisis del problema

El bucle principal es un componente indispensable en un videojuego. Una vez finalizada la inicialización del sistema al comienzo del programa, toda la ejecución del juego sucede dentro del bucle principal.

Dentro del bucle principal se ejecutan las siguientes tareas esenciales para el funcionamiento de la aplicación:

- Gestionar la entrada del usuario
- Atender los eventos y señales del sistema operativo
- Ejecutar la lógica y la simulación del mundo del juego
- Dibujar los gráficos.
- Reproducir el sonido.
- Asegurar una velocidad de ejecución adecuada del juego.

Al comienzo de este proyecto, UGK no contaba con esta funcionalidad, por lo tanto, los usuarios del motor se veían forzados a realizar su propia implementación de esta estructura de control básica.

Uno de los problemas más importantes a los que se enfrentan los usuarios cuando son obligados a implementar su propio bucle principal es que muchas de las tareas críticas que realiza el bucle son directamente dependientes de las plataformas donde se ejecuta la aplicación. Por ejemplo, un usuario debería de gestionar directamente la cola de mensajes de la API de Windows en su aplicación, o gestionar la sincronización temporal del bucle utilizando llamadas del sistema exclusivas de Windows.

La ausencia de un bucle principal es una clara deficiencia funcional de UGK que necesita ser solventada.

3.3.2 Solución propuesta

La solución elegida ha sido añadir a UGK un director centralizado de la lógica del videojuego. Este controlador implementa un bucle principal donde se gestionan todas las funcionalidades que dependen del hardware o de librerías de terceros, de esta manera abstraemos del usuario cualquier problema que dependa de una plataforma concreta.

Como parte de la interfaz con la que el usuario trabaja con el director para establecer la estructura lógica de su juego, se ha añadido una clase para representar lo que se ha denominado como “escena lógica” del juego.

La escena lógica representa un estado del juego, al que comúnmente los jugadores se refieren como “pantalla”: pantalla de juego, menú de inicio, pantalla de créditos, etc. El usuario puede especificar y añadir sus propias escenas extendiendo y sobrescribiendo la clase base utilizada para representar una escena y añadiéndola a la lista de escenas disponibles. Utilizando la clase director el usuario puede cargar escenas en su aplicación y transmitir entre ellas.

Podemos considerar el conjunto formado por la clase director y todas las clases de escenas lógicas como un autómata finito que gestiona la lógica de la aplicación.

3.4 Gestor de entradas

3.4.1 Análisis del problema

UGK carece de un sistema de gestión de entradas funcional. Anteriores aplicaciones realizadas con el motor, como *HyperChess* [5] o *Space Invaders*, optaron por implementar sus propios sistemas de gestión de entradas, para ello, utilizaron la API del Sistema operativo Windows.

UGK debe de ofrecer a los usuarios una interfaz simplificada para la gestión de entradas, pues no es posible implementar un subsistema de estas características en el espacio de la aplicación sin romper la encapsulación que aísla la lógica del juego del código de bajo nivel que gestiona los recursos del hardware.

Si para poder gestionar los periféricos de entrada, el usuario se ve obligado recurrir a llamadas del sistema operativo, la lógica del juego dependerá de una plataforma concreta, como puede ser Windows u otros sistemas operativos.

También es incorrecto utilizar las librerías de terceros empleadas por el motor en el espacio de código del juego. Estas librerías de terceros, como por ejemplo SDL o OpenGL, deberían ser dependencias del motor, no de la aplicación. Esta forma de vinculación "transitoria" entre las librerías de terceros y el código del usuario impediría que en un futuro UGK pudiera utilizar un conjunto de tecnologías diferentes.

La gestión de la entrada, o input, del jugador en los videojuegos tradicionales presenta una serie de problemas y restricciones que condicionan el diseño de este subsistema.

El diseño de un gestor de entradas convencional está sujeto a los siguientes requisitos:

- La lectura de las entradas debe realizarse a bajo nivel, mediante llamadas al sistema operativo o interfaces del hardware.
- Los dispositivos de entrada pueden ser muy variados: teclados, ratón, mando (joystick)
- La información del estado de las entradas es accedida por muchas secciones de código diferentes distribuidas a lo largo del programa.
- La lectura y actualización de los estados de las entradas debe realizarse en un punto concreto del bucle principal.

3.4.2 Solución propuesta

Se propone la implementación de un objeto global estático que proporcione la interfaz necesaria para gestionar los dispositivos de entrada.

La interfaz del objeto sería accesible desde todas las partes del programa, de manera que la lógica del juego podría consultar el estado de los inputs desde cualquier parte de la aplicación.

El objeto contaría con un método de actualización que leería y actualizaría el estado de todos los inputs, esta función sería llamada en cada iteración del bucle principal por el director lógico del juego. De esta forma la actualización de las entradas se realizaría en un solo punto del programa y las entradas siempre estarían actualizadas para el código del usuario en todos los frames.

Debido a la cantidad de métodos de entrada: diferentes, teclado, gamepads, volantes; se propone el uso de una estructura que modelaría un controlador ideal, y sería la interfaz uniforme que utilizaría la lógica del juego, para consultar el estado de los periféricos de entrada.

3.5 Integración de UGK con clases del usuario

Los motores también se encargan del emplazamiento y gestión de la memoria del programa, una de las implementaciones más comunes para lograr este objetivo es la utilización de pools de objetos preemplazados, para evitar en la medida de lo posible, la utilización del emplazamiento de memoria dinámica y la fragmentación de memoria.

Uno de los problemas que hay que resolver cuando la herramienta que se está desarrollando proporciona gestores de memoria es cómo integrar las clases de datos definidas por los usuarios en las estructuras, módulos e interfaces proporcionadas por el entorno de trabajo.

Para realizar esta integración se suele utilizar el patrón factoría, también conocido como constructor virtual. El patrón consiste en delegar la responsabilidad de instanciar los objetos en una clase intermedia. Los métodos que componen la interfaz de esta clase factoría solo utilizan referencias a una clase base definida por el motor y permite instanciar clases derivadas definidas por el usuario. Esta clase es inyectada en los módulos del sistema correspondientes y esto permite que la librería o entorno de trabajo pueda utilizar clases definidas por el usuario.

UGK necesita realizar este tipo de integración en las clases que componen su sistema de gestión de objetos de juego. Este sistema ofrece al usuario una interfaz mediante la cual destruir y crear entidades del juego.

En UGK las entidades de juego derivan de la clase `UGK::CCharacter`. Para utilizar sus propias clases derivadas, el usuario puede extender e implementar los métodos de la factoría `UGK::CCharactersFactory`.

Las instancias `UGK::CCharacterFactory` son inyectadas dentro de objetos de tipo `UGK:: CCharactersPool`, los cuales implementan y gestionan una lista de objetos reutilizables.

Analizando la interfaz de `CCharactersFactory` se han identificado dos inconveniencias que pueden resultar molestas para los usuarios.

```
typedef unsigned int UGKOBJM_CharacterType;
class UGK_API CCharactersFactory
{
protected:
	map<UGKS_String, UGKOBJM_CharacterType> CharacterDict;
public:
	CCharactersFactory();
	~CCharactersFactory();

	inline void InitDict(const UGKS_String CHARS_Tags[],
				unsigned int Size);
	//(…)
	inline UGKOBJM_CharacterType Lexeme2Token(UGKS_String Name);

	CCharacter* GeneralCreateNewCharacter(int Type, int subType);

	virtual CCharacter* createNewCharacter(int Type, int subType) = 0;

	virtual int getNumDistinctCharacterTypes();
};
```

Ilustración 11 Interfaz clase CCharactersFactory

El primer inconveniente que se percibe en esta interfaz es que el usuario debe de mantener y actualizar su implementación de la factoría si incluye nuevas clases en el juego o elimina alguna ya existente.

Para identificar el segundo problema se necesita analizar también la forma en que la clase `UGK::CCharactersPool y UGK::CCharacter` están implementadas:

```
/* Interfaz CCharacter, UGKCharacter.h */
class UGK_API CCharacter :public RTDESK_CEntity
{
	/* Virtual method that allows the object to be reseted as if it was created as
	   a completely new object by a constructor method
	   Please, move all initialization actions to this method.
	   This method is used by the pool/factory everytime a new object is asked for */
	virtual void Init();
}

/* Implementación CCharactersPool, UGKCharactersPool.cpp */
UGK::CCharacter* CCharactersPool::get(int Type, int subType)
{
	CCharacter* c;
	//(…)
	if (Pool[Type][subType].empty())
	//(…)
	else
	{
		Pool[Type][subType].pop_front();
		CharactersAmount--;
		c->Init(); //Reset the character to its very beginning as if it was just
				created right now
		/(..)
		return c;
	}
}
```

Ilustración 12 Interfaz CCharactersPool y CCharacter

La gestión de memoria realizada por el pool se limita a reutilizar instancias de objetos ya existentes almacenadas en una lista. Esta implementación tiene serias limitaciones y puede provocar diversos errores de gestión de memoria.

Este diseño intenta sustituir el uso de los constructores y destructores de objetos con un simple método `Init()`. El procedimiento de inicialización y destrucción de objetos es un mecanismo complejo de C++, que implica llamar en un orden determinado a los constructores y destructores de las clases base, así como llamar a los constructores de los miembros del objeto. Implementar el método `Init()` para cada objeto es una tarea difícil, propensa a errores y no funciona para clases que formen parte de una jerarquía de herencia compleja.

Esta implementación solo utiliza punteros de tipo `CCharacter`, lo cual obliga a utilizar *dowcasting* de forma extensiva en la lógica de la aplicación. Un *downcasting* a una clase derivada equivocada provoca corrupción de la memoria y errores difíciles de depurar.

Hay que resaltar como `UGK::CCharactersPool` no tiene ningún control sobre como emplazar la memoria de los objetos y además su diseño parece asumir que las factorías utilizan memoria dinámica. Por lo tanto, no se evita completamente el sobrecoste que suponen las llamadas del sistema para reservar memoria y la fragmentación de esta.

3.5.1 Solución propuesta

Se propone la implementación de un tipo de factoría donde las propias clases sean capaces de auto registrarse. Estas nuevas factorías estarían disponibles para el conjunto del programa y no sería necesario inicializarlas ni inyectarlas como dependencias en ninguna otra clase. De esta forma no se requiere que el usuario realice ninguna tarea adicional para poder utilizar sus clases dentro de UGK ni se le exige que implemente y mantenga sus propias factorías.

Para conseguir este comportamiento se pretende explotar la inicialización de las variables estáticas que C++ realiza antes de que comience la ejecución de la función `main()` del programa [11].

Se propone implementar verdaderos pools de objetos empleando para ello bloques de memoria contiguos previamente reservados. También se propone abandonar el uso del método `Init()` y utilizar en su lugar los propios constructores y destructores de cada clase.

Para construir instancias de objetos reutilizando memoria previamente reservada se emplearán el operador *placement new* y llamadas explicitas a los destructores de las clases.

4 Diseño de la solución

En este apartado se expone el diseño de las soluciones adoptadas para solventar los problemas previamente analizados en el apartado ANÁLISIS DE UGK.

Se pretende dar una explicación de alto nivel poniendo énfasis en la arquitectura, esquema de clases y estructuras de datos utilizadas en la solución.

4.1 Bucle principal y gestor de escenas lógicas

Se ha añadido al programa la clase `UGK::LogicScene`. La interfaz `LogicScene` consiste únicamente del método virtual puro, `runStep`, el cual es ejecutado por el director lógico del juego en cada iteración del bucle principal. El cometido del método `runStep` es actualizar toda la lógica y simulación del mundo del juego.

Para implementar sus propias escenas el usuario debe de extender la clase `LogicScene` e implementar el método `runStep`. La adquisición y liberación de recursos pertenecientes a la escena, como pueden ser modelos tridimensionales y texturas, pueden ser codificadas, respectivamente, en el constructor y destructor por defecto de la clase.

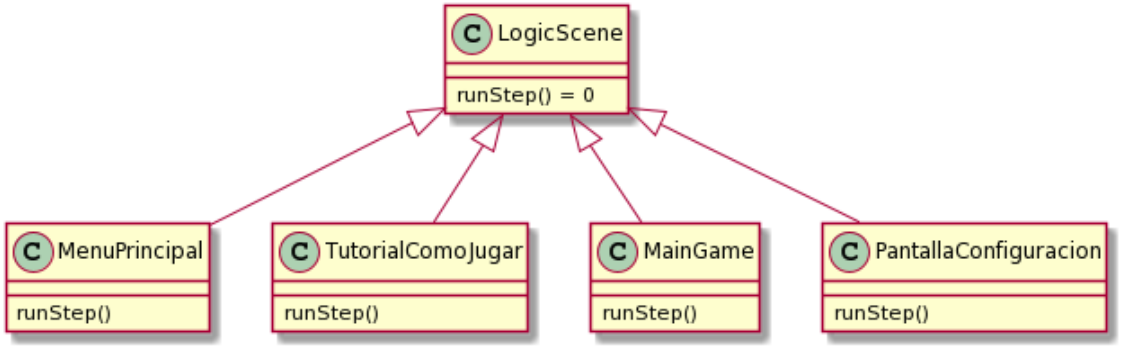

Ilustración 13 Ejemplo Jerarquía de Escenas de un juego

Director Lógico

El director lógico del juego esta implementado en la clase `UGK::Director`. Esta clase utiliza un patrón *singleton* por lo tanto solo puede existir una única instancia durante la ejecución del programa.

Los métodos que componen la interfaz pública de `UGK::Director` permiten al usuario cargar instancias de la clase `UGK::LogicScene` y transitar entre ellas. Podemos

considerar el conjunto formado por `Director` y `LogicScene` como una máquina de estados que gobierna la lógica de la aplicación.

El director se encarga de llamar al constructor de la escena cuando se transita hacia esta, adquiriendo y cargando en memoria todos los recursos requeridos. Cuando se transita a otra escena diferente, el director ejecuta el destructor de la escena actual, liberando los recursos utilizados en la escena previa.

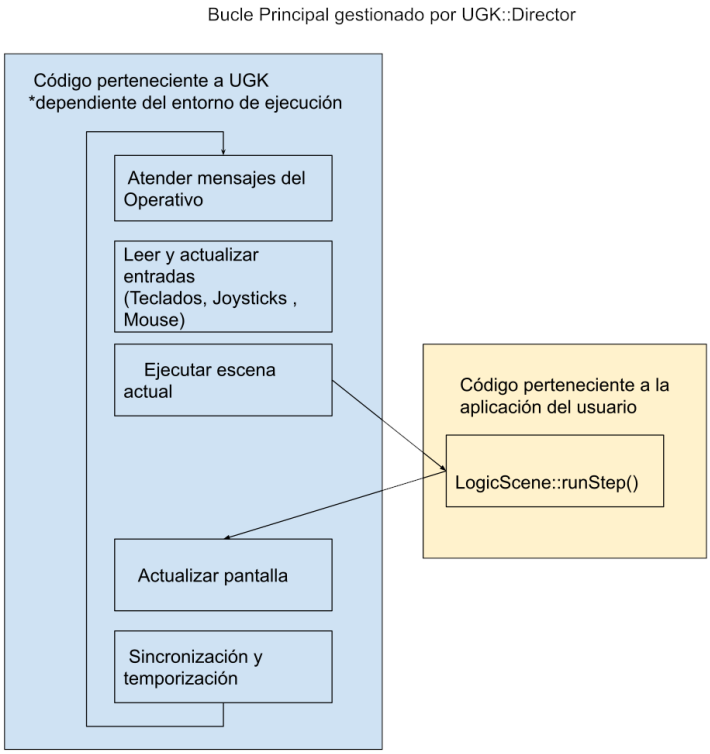

Ilustración 14 Bucle Principal de UGK

Como se observa en el diagrama, el bucle principal del juego codificado dentro de `UGK::Director`, se encarga de gestionar todas las tareas dependientes del hardware y llama al código del usuario para actualizar la lógica del juego.

4.2 Gestor de entradas

Se han definido una serie de funciones estáticas globales que permiten leer el estado de las entradas de los jugadores desde cualquier punto del código de la aplicación.

El contexto y los subsistemas necesarios para leer los eventos de entrada se inicializan en el momento en que alguno de los métodos que componen la interfaz son utilizados por primera vez.

La interfaz está compuesta por las siguientes funciones estáticas:

- *EasyUpdateInput():*

  Se encarga de leer y actualizar el estado de todas las entradas. Este método es llamado por UGK en cada iteración del bucle principal, con la finalidad de que el código de la aplicación siempre tenga disponible el estado actualizado de las entradas al utilizar las funciones *EasyReadGamePad*.

- *GamePad EasyReadGamePad():*

  Devuelve el estado del mando principal, almacenado dentro de una estructura de tipo *Gamepad*.

- *GamePad EasyReadGamePad(numero):*

  Devuelve el estado del mando enumerado. En caso de no existir se devuelve un GamePad con estados por defecto y marcado como invalido.

Se designa como mando principal al joystick de entrada más antiguo que continúa conectado al sistema. En caso de no existir ningún joystick conectado, el teclado del sistema asume el rol de mando principal.

Estructura GamePad

El componente más importante de esta interfaz, que nos permite simplificar la complejidad de lidiar con diferentes dispositivos de entrada, es la estructura *GamePad*.

GamePad representa un mando ideal con un número de botones, crucetas, gatillos y joysticks fijos.

Todos los eventos generados por cualquier clase de periférico de entrada existente, como pueden ser teclados, mandos Dualsock, mandos Xbox360, joystick arcade, etc.; son mapeados a campos de la estructura *GamePad*.

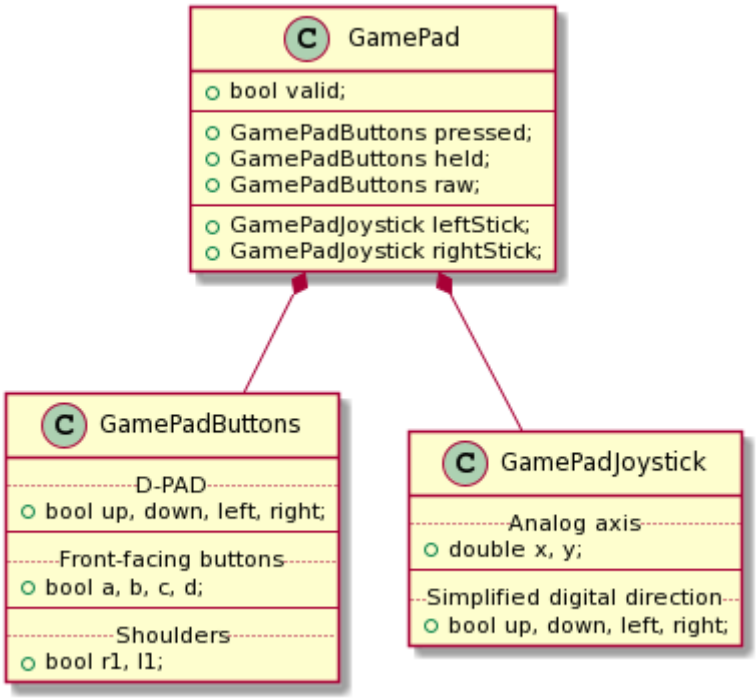

Ilustración 15 Clase GamePad

Como se puede observar en la Ilustración 15, `Gamepad` proporciona al usuario la posibilidad de distinguir entre botones que han sido presionados durante un solo frame, cuyo estado se almacena en el campo `pressed`, *y* botones que están siendo mantenidos durante más de un frame, cuyo estado se almacena en el campo `held`.

Para conveniencia del usuario los niveles de entrada analógicos leídos para los ejes de los joysticks se discretizan, de forma que, pueden ser tratados como la entrada de una cruceta analógica.

4.3 Factorías con mecanismo de registro automático

A continuación, se describe la implementación de la solución propuesta en el apartado 3.5.1.

Factorías

Se ha definido una plantilla genérica `UGK::Factory<>` dentro de la cabecera `UGKFactory.h`.

Esta plantilla instancia una factoría para una clase base concreta. Este tipo base será utilizado como interfaz común para manipular los objetos creados por esta especialización de `UGK::Factory<>`. Por ejemplo, mediante `UGK::Factory<CCharacter>` se pueden instanciar clases derivadas de `CCharacter` y los punteros utilizados por la interfaz de la factoría serán del tipo `CCharacter *`.

Las factorías son clases estáticas globales que pueden ser accedidas desde cualquier parte del programa.

La interfaz de `UGK::Factory<>` se ha diseñado alrededor de dos ideas principales: las clases deben de poder referenciarse simplemente por su nombre y la factoría permite utilizar los constructores de la clase para crear instancias nuevas o emplazar objetos reutilizando memoria.

```
template <typename Base>
class Factory {
	private:
		typedef std::unordered_map<Typeid, BluePrints_<Base>> TypesMap;
	public:
		/**
		* @return OWNED pointer to new object
		* @retval nullptr if subclass does not exist
		*/
		static Base* create(std::string &name);

		/* emplace object at the given memory block */
		static bool emplace(std::string &name, void *mem);

		/* get an human readable string with all factory type names */
		static std::string getTypesList();

		/* Register a derived class */
		static bool add(BluePrints_<Base> &bluePrints);
};
```

Ilustración 16 Factoría genérica

En este resumen simplificado de la interfaz están presentes los dos métodos estáticos que permiten crear objetos: `create()`, que emplea memoria dinámica; y `emplace()`, que construye el objeto en el bloque de memoria proporcionado por el usuario.

Toda la información que se necesita para la creación de objetos está representada por la estructura genérica `BluePrints<>`. Para cada tipo derivado que ha sido registrado, existe un `BluePrints<>` almacenado en el mapa privado de la factoría.

Reflexión: la clase BluePrints<>

El tipo genérico `BluePrints<>` es la estructura que almacena toda la información necesaria sobre una clase para poder ser instanciada por una factoría. Contiene el nombre de la clase registrada, punteros a su constructor y el tamaño en bytes que ocupa una instancia.

`BluePrints<>` intenta suplir, de forma rudimentaria, la ausencia de un mecanismo de reflexión en el lenguaje C++.

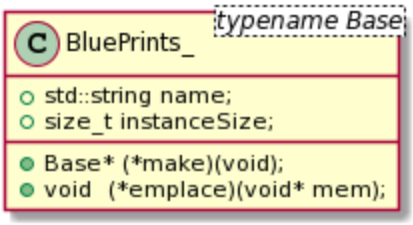

Ilustración 17 Clase descriptor BluePrints_

Auto registro:

El auto registro de las clases se implementa explotando el mecanismo de inicialización de las variables estáticas de C++.

Cuando el programa comienza su ejecución, las variables estáticas son inicializadas antes de llamar a la función `main()`, que es el punto de entrada de la aplicación. Utilizando este mecanismo, podemos lograr que las funciones encargadas de registrar las clases en las factorías sean ejecutadas de forma automática cuando se inicia el programa.

Para que esta solución funcione, es necesario que los mapas de las factorías sean inicializados antes que las variables estáticas que realizan el registro, pero el estándar de C++ no establece ningún orden de inicialización entre las variables estáticas definidas en diferentes unidades de traducción [12].

La solución a este problema es emplear el idioma *construct on first use* [13]. Se envuelven los mapas estáticos dentro de una función también estática y se asegura su inicialización antes de que sean utilizados por primera vez.

```
template <typename Base>
class Factory{
		//Wrapping the map inside a function
		//prevents the "Static Initialization Order Fiasco"
		static TypesMap& p_getMap(void) {
			static TypesMap map_;
			return map_;
		}

}
```

Ilustración 18 Construct on first

Para utilizar el mecanismo de auto registro, los usuarios solo deben de incluir la macro `#REGISTER( BaseClass, ClassName )` dentro de la definición de sus clases.

```
class MyCharacter : public UGK::CCharacter {
	public:
	#REGISTER( UGK::CCharacter, MyCharacter )
}
```

`#REGISTER` comprueba mediante aserciones estáticas que su segundo argumento `ClassName` es exactamente el nombre de la clase del usuario y que `ClassName` deriva de la base `BaseClass`, de no cumplirse estas condiciones, se detiene la compilación y se informa al usuario del problema.

La macro define un método estático `RegisterInitialize()` dentro de la clase del usuario, que será utilizado para realizar el registro en la factoría cuando se inicialicen las variables estáticas del programa.

Dentro de `RegisterInitialize()` se instancia una especialización de la plantilla `TypeBluePrints_<>` para esta clase derivada.

`TypeBluePrints_<>` es la clase donde está declarada la variable estática cuya inicialización provoca el registro de la clase en la factoría:

```
	/* Create Blueprints for a distinc type of object */
	template <typename Base, typename Derived>
	struct TypeBluePrints_ : public BluePrints_<Base>{
		static bool register_entry;
	};
	//Header Only self-register statics
	template <typename Base, typename Derived>
	bool TypeBluePrints_<Base, Derived>::register_entry =

		Derived::RegisterInitialize();
```

Ilustración 19 Auto registro

Como se observa, la inicialización de la variable estática `bool TypeBluePrints_<> ::register_entry` llama a la función `Derived:: RegisterInitialize()`, que registra la clase `Derived` en `UGK::Factory<Base>`. Por lo tanto, UGK es capaz de crear instancias de este tipo derivado mediante factorías de la siguiente forma:

```
/* Utilizando memoria previamente reservada*/

UGK::Factory<Base>::emplace("MyClass", poolMemory_ptr);

/* Nueva instancia utilizando memoria dinámica*/

MyClass *o = UGK::Factory<Base>::create("MyClass");
```

Ilustración 20 Ejemplo de uso clase Factory<>

Como indica el estándar de C++, Los miembros estáticos de una plantilla solo se definen e inicializan sí su así lo requiere [14]. Para asegurar su inicialización, se ha utilizado el miembro estático dentro del constructor de `TypeBluePrints_<>`

```
TypeBluePrints_::TypeBluePrints_(std::string name_) { (void)register_entry;}
```

Ilustración 21 forzar inicialización variable estática

5 Caso de Uso: Simulador de vuelo Arcade.

Para poder evaluar las interfaces y funcionalidades de UGK desde el punto de vista de un usuario y probar la implementación de las soluciones descritas en el apartado 4, se ha desarrollado un simulador de vuelo Arcade.

Esta aplicación es enlazada con la librería estática `UPVGameKerneld.lib` para generar el ejecutable final del juego.

Debido a la falta de tiempo, solo han podido ser implementadas las funcionalidades más básicas, necesarias para ejercitar y depurar las mejoras introducidas en UGK.

Al tratarse de un juego arcade, no se pretende realizar una simulación realista y rigurosa sino proporcionar una experiencia de juego accesible y centrada en la acción. El proposito del juego es derribar los aviones enemigos presentes en el mapa, utilizando para ello las armas de las que dispone nuestro caza de combate.

Ilustración 22 Cámara 3D (I)

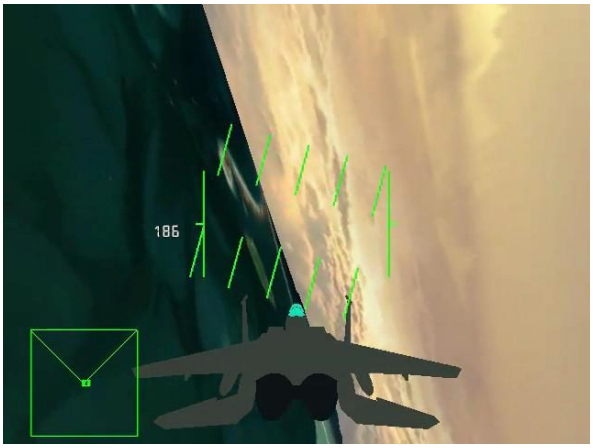

Ilustración 23 Cámara 3D(II)

Movimiento y Cámara

Se ha implementado en el simulador una clase cámara que soporta rotaciones con tres grados de libertad.
La interfaz de rotación de la cámara consiste en una función que permite especificar una rotación local, respecto a la orientación actual de la camera, utilizando ángulos de Euler.

```
void EasyCamera::localRotate(float yaw, float pitch, float roll);
```

Ilustración 24 Interfaz rotación cámara

Los ángulos de Euler de la rotación local son convertidos en una matriz que es aplicada a la matriz de vista actual. Los ángulos de Euler correspondientes a la orientación final son extraídos de la matriz resultante de esta operación utilizando el algoritmo descrito en [16]

HUD

Ilustración 25 HUD

Ilustración 26 HUD(II)

Los aviones enemigos visibles son marcados utilizando indicadores verdes cuadrados acompañados de un contador que informa al jugador de la distancia a la cual se encuentra el objetivo. El indicador cambia a color rojo cuando se fija el objetivo.

Para implementar esta interfaz, se recuperan las coordenadas tridimensionales de los aviones enemigos y se realiza una proyección al espacio de la pantalla, empleando para ello las matrices de proyección y vista actuales que están siendo utilizadas para dibujar la escena.

Misiles

El usuario es capaz de seleccionar un objetivo entre los enemigos visibles y disparar misiles trazadores.

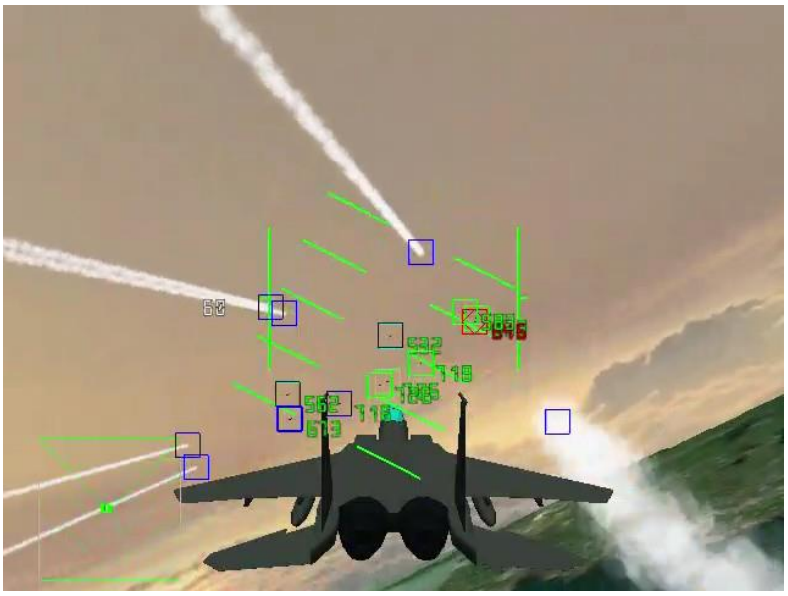

Ilustración 27 Misiles

Los misiles y sus estelas son objetos creados dinámicamente y para su gestión se ha implementado en el juego un prototipo de manejador de entidades.
Este manejador de entidades hace uso de las factorías descritas en el apartado 4.3, mediante las cuales es capaz de crear y destruir objetos reutilizando un bloque de memoria previamente reservado, evitando el uso de memoria dinámica.

6 Conclusión

En este proyecto se ha realizado un análisis del motor de videojuegos desarrollado internamente en la facultad de informática de la Universitat Politècnica de València.

Se han identificado y documentado los problemas más importantes con los que contaba el programa, que amenazan la viabilidad del proyecto, y se han propuesto soluciones concretas para todos ellos.

Se ha identificado y solventado una de las deficiencias funcionales más importantes con las que contaba UGK, la ausencia de un bucle principal. Dicha estructura de control ha sido implementada en una clase que se encarga de gestionar todas las tareas relacionadas con la plataforma de ejecución en cada iteración. Con la solución que ha sido implementada, el usuario ya solo debe preocuparse de desarrollar la lógica de su videojuego y gestionar los recursos utilizados por las escenas de su juego.

A lo largo de este proyecto se ha ejercitado UGK, desarrollando pequeños ejemplos de prueba comprobando de esta manera que el código ya existente continuaba compilándose y ejecutándose en entornos actuales. Esto es importante pues el proyecto UGK es complejo y cuenta con gran cantidad de dependencias externas empaquetadas y distribuidas junto al proyecto en forma de librerías estáticas y librerías dinámicas. Hemos comprobado que efectivamente estas librerías y el proyecto UGK continúa ejecutándose correctamente en las últimas versiones del sistema operativo Windows.

A nivel personal, esta trabajo ha servido para conseguir experiencia a la hora de leer y mantener un repositorio de código de tamaño considerable. El autor siempre ha considerado que la mayor barrera entre el mundo académico y la industria es enfrentarse a programas complejos de gran tamaño y ser capaz de entenderlos e introducir modificaciones respetando las convenciones y estándares utilizados por la organización creadora del programa. En ese sentido creemos que la experiencia resultará útil en el mundo laboral.

6.1 Relación con los estudios cursados

Al ser UGK un programa complejo desarrollado en C++, se han utilizado y ejercitado de forma extensa gran parte del conocimiento adquirido en las asignaturas de programación cursadas en la UPV.

La asignatura Estructura de Datos y Algoritmos (EDA), nos ha proporcionado los conocimientos necesarios para poder trabajar cómodamente con conceptos como herencias, genericidad y polimorfismo. Aun siendo usuarios noveles del lenguaje C++, la asignatura EDA ya nos había introducido al conocimiento teórico existente detrás de la mayoría de las funcionalidades del lenguaje.

La asignatura Algoritmia nos ha permitido hacer estimaciones intuitivas del coste computacional del código que estábamos leyendo.

El conocimiento adquirido en la asignatura Ingeniería del Software (ISW) ha sido indispensable. Hemos sido capaces de comprender de forma eficiente el funcionamiento de grandes partes del programa gracias a su descripción en términos de patrones de diseño. La mayoría del trabajo realizado en este proyecto ha girado en torno a esta asignatura, pues en gran medida a consistido en realizar refactorizaciones para mejorar la calidad del código existente.

A la hora de lidiar con todo el código relacionado con los gráficos tridimensionales y a la hora de desarrollar aplicaciones y videojuegos para probar los cambios introducidos, las asignaturas Sistemas Gráficos Interactivos (SGI) y Algebra Lineal (ALG) han resultado indispensables.

7 Trabajos Futuros

A continuación, listamos algunas de las funcionalidades con las que aún no cuenta UGK y creemos que sería necesario implementar:

- Remplazar el código obsoleto que utiliza el modo inmediato de OpenGL por código moderno que utilice *Vertex Buffer Objects* y variables uniformes.
- Diseñar un sistema de renderizado que soporte técnicas de iluminación más complejas que las proporcionadas por el OpenGL de tubería fija utilizado actualmente por el motor.
- Implementar un sistema de animaciones de modelos tridimensionales
- Añadir soporte para quaternions y utilidades que permitan interpolar entre dos orientaciones tridimensionales distintas.
- Revisar el diseño de las interfaces de programación para que resulten más fáciles e intuitivas de utilizar para el usuario.

Bibliografía

[1] J. Thomas, "Advanced Character Physic".

[2] P. Michele, "The use of Fuzzy Logic for Artificial Intelligence in Games," Department of Computer Science, University of Milano, Milano, 2012.

[3] T. Tommy, "Outsmarting The Covenant: The AI of Halo 2," 2017. [Online]. Available: https://medium.com/the-cube/theaiofhalo2-33e824209a4c.

[4] R. Green, "Sponsored Feature: The All-Important Import Pipeline," [Online]. Available: https://www.gamedeveloper.com/programming/sponsored-feature-the-all-important-import-pipeline.

[5] E. Nogueroles Bertó, " HyperChess. Videojuego de ajedrez configurable empleando el API UGK," Universitat Politècnica de València. Escola Tècnica Superior d'Enginyeria Informàtica, Valencia, 2018.

[6] The GNOME Project, "Memory Management," 06 03 2021. [Online]. Available: https://web.archive.org/web/20210306043239/https://developer.gnome.org/programming-guidelines/stable/memory-management.html.en.

[7] I. Turner-Trauring, "Object ownership across programming languages," 26 01 2017. [Online]. Available: https://codewithoutrules.com/2017/01/26/object-ownership/.

[8] E. Lavesson, "c ownership semantics," [Online]. Available: https://ericlavesson.blogspot.com/2013/03/c-ownership-semantics.html. [Accessed 23 08 2021].

[9] H. &. C. G. Spencer, "#ifdef Considered Harmful, or Portability Experience with C News," in *USENIX Summer*, 1992, 1992.

[10] F. Deißenböck, "Living in the #ifdef Hell," 27 10 2015. [Online]. Available: https://www.cqse.eu/en/news/blog/living-in-the-ifdef-hell/. [Accessed 01 08 2020].

[11] A. Zimmerer, "Header-Only Self-Registering Classes for Factories in C++," 2 Mayo 2020. [Online]. Available: https://www.jibbow.com/posts/cpp-header-only-self-registering-types/. [Accessed 10 Noviembre 2020].

[12] "What's the "static initialization order fiasco"?," c++-faq, [Online]. Available: http://www.cs.technion.ac.il/users/yechiel/c++-faq/static-init-order.html.

[13] "How do I prevent the "static initialization order fiasco"?," c++-faq, [Online]. Available: http://www.cs.technion.ac.il/users/yechiel/c++-faq/static-init-order-on-first-use.html.

[14] ISO, "13.9.2 Implicit instantiation," in *Programming Language C++. [Working draft]*.

[15] C. Preschern, Patterns to Escape the #ifdef Hell, In Proceedings of the 24th European Conference on Pattern Languages of Programming (EuroPLoP), 2019.

[16] M. Day, "Extracting Euler Angles from a Rotation Matrix".