UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

HyperChess. Videojuego de ajedrez configurable empleando el API UGK

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Esteban Nogueroles Bertó

Tutor: Ramón Pascual Mollá Vayá

Curso: 2017-2018

Resumen

El trabajo consiste en la creación de un videojuego llamado *HyperChess*, que simula el juego del ajedrez. Se ha realizado una reconstrucción en prácticamente su totalidad de una versión anterior desactualizada del juego, que ya no funcionaba. Para ello se ha utilizado la aplicación grafica de UGK (*UPV game kernel*) como motor de soporte. Tiene la posibilidad de ser configurable gracias a archivos externos en formato *.html*, permitiendo así jugar distintas variantes del juego del ajedrez.

Palabras clave: ajedrez, videojuego, configurable, UGK.

Resum

El treball consisteix en la creació d'un videojoc anomenat *HyperChess*, que simula el joc dels escacs. S'ha realitzat una reconstrucció en pràcticament la seva totalitat d'una versió anterior desactualitzada del joc, que ja no funcionava. Per a això s'ha utilitzat l'aplicació gràfica de UGK (*UPV game kernel*) com a motor de suport. Té la possibilitat de ser configurable gràcies a arxius externs en format *.html* permetent així jugar diferents variants del joc dels escacs.

Palabras clave: escacs, videojoc, configurable, UGK.

Abstract

The assignment consists of the creation of a video game called HyperChess, that simulates the game of chess. A reconstruction has been refactored almost entirely of a previous outdated version of the game, which no longer worked. For this, the graphic application of UGK (UPV game kernel) has been used as a support engine. It has the possibility of being configurable thanks to external files in .html format, thus allowing to play different variants of the game of chess.

Keywords: chess, videogame, configuration, UGK.

Tabla de contenidos

1 Introducción 9
1.1 Motivación 10
1.2 Objetivos 11
1.3 Metodología 11
1.4 Estructura de la obra 13
2 Estado del arte 14
2.1 Sistemas y tecnologías 14
2.1.1 Grafos de Escena 14
2.1.2 Interfaces Graficas 17
2.2 Actualidad del ajedrez 19
2.2.1 Ejemplos de variaciones en juegos de ajedrez 19
2.2.2 Similitudes entre el proyecto y la actualidad 21
2.3 Critica al estado del arte 22
2.4 Propuesta 23
3 Análisis del problema 24
3.1 Análisis de requisitos 24
3.1.1 Funcionalidad 24
3.1.2 Interacción usuario juego 25
3.2 Análisis de soluciones 25
3.3 Solución propuesta 26
3.3.1 Motivos de la elección 26
3.3.2 Especificaciones de la solución escogida 27
3.4 Análisis de eficiencia algorítmica 28
3.5 Trabajo y presupuesto 29
3.5.1 Plan de trabajo 29
3.5.2 Calculo de tiempo 32
3.5.3 Calculo del presupuesto 32
3.6 Colaboración 34
4 Diseño de la solución 35
4.1 Análisis de las herramientas 35

4.1.1 Sistema operativo utilizado y lenguaje de programación 35
4.1.2 Entorno de desarrollo 36
4.1.3 UPV Game Kernel 36
4.1.4 Creación o modificación de contenido de autor 38
4.2 Arquitectura del software 39
4.2.1 Capas dentro del videojuego 39
4.2.2 Herencias e inclusiones 41
4.2.3 Métodos recurrentes o altamente utilizados 44
4.3 Implementación 45
4.3.1 Jugador 45
4.3.2 Piezas 46
4.3.3 Gestores 47
4.3.4 Analizadores 48
4.4 Desarrollo 49
4.4.1 Primera implementación 49
4.4.2 Implantación Definitiva 51
5 Resultados 53
5.1 Analizador HTML 53
5.2 Carga de contenido de autor 53
5.3 Movimiento y actualización 54
5.4 Otros Objetivos 55
5.4.1 Carga y eliminación del grafo de escena 55
5.4.2 Utilización de UGK y actualización de la aplicación 55
5.4.3 Interfaz 55
5.4.4 Ampliación 55
6 Conclusiones 56
6.1 Relación con los estudios cursados 56
6.2 Trabajos Futuros 57
6.2.1 Mejor interacción tablero y bonificación 57
6.2.2 Inteligencia artificial 57
6.2.3 Reglas configurables 57
6.2.4 Interfaz 57
6.3 Agradecimientos 58
7 Bibliografía 59

Apéndice 61
Explicación archivos de inicialización HTML 61
Explicación de archivos de nivel HTML 64

1 Introducción

Los videojuegos son una nueva forma de entretenimiento. Los cuales han ido aumentando en popularidad en los últimos años, más en concreto desde el dos mil en adelante. Estos crean nuevas experiencias de entretenimiento, para el usuario, pero nunca olvidando de donde vienen, es decir, de los juegos clásicos de mesa. Siendo uno de ellos, el más conocido, el ajedrez. Siendo este el que, de alguna forma, es para toda aquella persona que quiera un desafío competitivo contra otra persona y que no requiera un gran esfuerzo físico. Aunque también se pueda intentar perfeccionar tu técnica jugando en solitario.

Siempre se puede innovar respecto de las reglas ya establecidas de un juego, es decir, modificar estas mismas. Pero todo eso sería más fácil si no fuera necesario la creación, en muchos casos, de un nuevo tablero para las nuevas reglas. Esto conlleva a que hay que fabricar o modificar uno ya comprado y esto no resulta atractivo a la mayoría de la gente. Por lo tanto, la forma más fácil acaba siendo la digitalización de este y convirtiéndose así en un videojuego. Puesto que es más fácil cambiar una línea de código o unos datos almacenados que crear algo físico desde cero. Además, en este mundo tan globalizado y con tanta información en movimiento. Todo esto más fácil de transmitir gracias a las tecnologías de la información. Las cuales hacen llegar tu nueva visión del producto a otras personas de todo el mundo de forma rápida y eficiente. Aunque solo sea el ya existente juego del ajedrez con nuevas reglas de juego. Siendo que cada nueva aportación tiene que poder ser realizado por otras personas en cualquier parte del mundo sin tampoco necesitar de un gran esfuerzo físico. Por esto es también necesario una digitalización del propio juego, porque nos permite transferirlo a otros públicos de forma más sencilla. Ya sea solo descargando una aplicación vía internet o transfiriéndose de unos a otros a través de una simple memoria USB. Aportando a las nuevas generaciones otra forma de entretenerse con un juego clásico que no tiene que considerarse desfasado. Además, que se puede utilizar este como una forma de enseñanza de algunos aspectos informáticos, como por ejemplo programación, o ya más relacionado con el juego, como concentración y estrategia.

Es importante que una aplicación como un videojuego o como cualquier otra, tenga una configuración. Porque que cada persona siempre ha tenido preferencias distintas respecto de los atributos del mismo concepto, en este caso las reglas del juego. Por lo tanto, también hay que permitir que cada uno aporte una visión más personal del mismo concepto al videojuego. Todo esto se tiene que poder permitir hacer al usuario para que se sienta más atraído hacia el propio videojuego. Es decir, tiene que ser personalizable, incluso de algo tan sencillo como pueden ser la propia apariencia del tablero o de las piezas. Aunque también tienen que poderse modificar las reglas. Estas tradicionales reglas que ya están muy arraigadas en la sociedad. Pero con lo grande que es el mundo y con la cantidad de gente que existe en él, toda comunicada a día de hoy como se ha mencionado, sería un desperdicio no conocer sus propias modificaciones de esas reglas. Por lo tanto, como se ha querido remarcar, una aplicación tiene que ser capaz de adaptarse a su público objetivo incluso si es solo una del juego del ajedrez. Esto es lo que se ha buscado en este trabajo.

1.1 Motivación

Existen distintos motivos por los que se ha escogido este proyecto, de entro los cuales se han puesto tanto personales como profesionales y didácticos. Estos son:

- El desarrollo de un videojuego, aunque sencillo, siempre ha sido una motivación intrínseca. Puesto que siempre se ha tenido una afición hacia el sector de los videojuegos y porque además se está familiarizado con el propio juego del ajedrez. Por lo que hace considerarlo una buena propuesta, teniendo en cuenta que genera una satisfacción personal, cuando se ha completado el proyecto. Porque el desarrollo de aplicaciones me es mayor motivación que el simple desempeño de un sistema en red o su mantenimiento. Por muy pequeña que pudiera llegar a ser la aplicación creada.

- La industria del videojuego es un mercado que se encuentra en pleno crecimiento. Que además de englobar a la informática, también se encuentra en otros sectores como pueden ser el artístico, aprendizaje, simulación, etc. Permitiendo aumentar las experiencias del propio programador.

- La forma de programar de un videojuego no es exactamente la misma de la gran mayoría de aplicaciones que hay en el mercado o que se pueden haber dado en la carrera. Por lo tanto, también expande nuevas formas de comprensión y desarrollo a la hora de resolver problemas en programación.

- El propósito también de esta aplicación es de aprender y enseñar con la misma. Por lo que además de poder aprender nuevas cosas con la creación de esta. También servirá para que otras personas en adelante aprendan del funcionamiento de la misma y puedan aplicarlo en sus propios proyectos en el futuro o también sobre la propia aplicación en sí misma.

- El propio motor gráfico de la universidad, el UGK, está en plena expansión constante y en perfeccionamiento. Por lo tanto, poder hacer un trabajo con esta aplicación, permite experimentar en persona los cambios que en un futuro pueden soportar otras aplicaciones similares. Las cuales se fundamentan en bases en continuo cambio o modificación. Ampliando así la adaptabilidad del propio desarrollador.

1.2 Objetivos

Crear un videojuego basado en el juego del ajedrez que utilice la aplicación de la universidad denominada UGK (UPV *Game Kernel*). Utilizando los grafos de escena y objetos que este brinda, sin dejar visible al resto de usuarios la utilización de sistemas como puede ser el OSG (*Open Source Graph*) u otros que pueda utilizar internamente el UGK.

Diseñar unos analizadores que permitan, mediante archivos externos escritos en html, configurar las características del juego sin necesidad de cambiar nada del código base del propio juego.

Que los objetos del juego realicen las funciones que tendrían que realizar correctamente. Esto incluye que se muevan, se visualicen o se carguen correctamente. Para ello se tiene que tener en cuenta que todos los requisitos de funcionalidad descritos en el capítulo tres sean completados con éxito.

Permitir que la aplicación sea ampliable en nuevos proyectos que puedan aportar más al videojuego en sí y desarrollar de forma que se pueda utilizar como aplicación de enseñanza en la asignatura de programación de videojuegos que imparte esta facultad.

1.3 Metodología

El desarrollo de la aplicación se realiza de la siguiente forma. En primer lugar, lo que se hace es buscar los fallos que existen en la aplicación previa y documentarlos, a la par que se tiene que documentar también respecto de la aplicación que se va a utilizar UGK con todos los sus apartados, funcionalidades, métodos, etc. También se tiene que documentar sobre el funcionamiento de los grafos de escena y de las librerías de OSG.

Después es la instalación de las aplicaciones que serán útiles para la programación y desarrollo del proyecto, este paso depende de lo que ya se tenga instalado o de lo que se necesite a posteriori. Todas las aplicaciones importantes que se utilizaran están analizadas en el capítulo 4 en la sección de herramientas.

Continuando con la implementación. Se tiene que buscar una o más soluciones que se pueden implementar para pasar a las siguientes fases de desarrollo con las ideas más claras. Además, tener en cuenta cómo se encuentra el estado del arte en ese momento y apoyarse también en propuestas o trabajos ya existentes, que sirvan de referencia para tomar una dirección determinada. Pero siempre aportando algo nuevo sin salirse del marco establecido por el proyecto.

Más tarde la implementación pasa a la fase de eliminación de errores previos y la importación de las librerías que sean necesarias para el funcionamiento de la aplicación. Esta parte puede estar en constante uso, puesto que siempre pueden faltar cosas por importar o errores desconocidos que corregir.

Buscar o crear todo el contenido de autor que puede ser utilizado por la aplicación. Si fuera necesario también documentarse respecto de los programas que pueden ser necesarios para editar todo este contenido adicional. Se denominará como contenido de autor todo aquello que no es ni código, ni librerías, ni programas o soluciones. Dentro de este concepto entrarían por ejemplo todo lo que es modelos 3D, imágenes o texturas, sonidos, efectos, partículas, etc. Todo lo que acaba siendo percibido por el usuario.

Después, establecer las pautas a seguir, en las cuales se centra en la refactorización y reconstrucción de los archivos y librerías que tendría el viejo juego. Pero aplicando el nuevo sistema de UGK, que es necesario en este proyecto. Utilizando una serie de puntos fuertes a explotar de la aplicación gráfica, como la capacidad de usar grafos de escena que facilite la inserción de elementos visibles.

Además, planificar y desarrollar como se harán los analizadores que se encargarán de leer los archivos HTML, que se va a poner como archivos de configuración. Se utilizan estos archivos porque UGK tiene ya integrado parte del sistema de lectura de este tipo de archivos. Además, son más fáciles de escribir, a la par que más fáciles de entender por parte del usuario a la hora de configurar el juego.

Más tarde se buscará que todo lo que se haya aportado al proyecto funcione correctamente y que no tenga errores que no permitan el funcionamiento de la aplicación. Ejemplo de esto serian errores de compilación o errores de ejecución que salten desde el principio o medio camino, impidiendo su funcionamiento.

Finalmente se analizarán las partes que no se realizaron en este proyecto y cuáles podrían ser posibles ampliaciones para futuros proyectos.

1.4 Estructura de la obra

La memoria se divide en cinco apartados, seis si se tiene en cuenta la introducción. En los cuales, para que la información sea más fácil de localizar, se hablara por encima a continuación. Para poder conocer en menor medida de que trata cada uno de los mismos.

En el primer capítulo está la introducción. Es el apartado en el que se encuentra este sub apartado y que tiene como finalidad aproximar al lector de la memoria a qué tipo de trabajo está contemplando y como se va a desarrollar la propia memoria a partir de aquí.

En el segundo capítulo está el estado del arte. Donde se exponen como se encuentra las tecnologías relacionadas con el trabajo que se describe en esta memoria. Mostrar en qué puntos se parece lo que se está desarrollando en el ámbito global con lo que se hace en este proyecto. Además, también de hablar del medio y el interés global en su determinada medida sobre la materia del trabajo y que problemas hay con lo que ya está en el mercado actual y que es lo que se va a aportar.

En el tercer capítulo es el análisis del problema. Donde se exponen cuáles son los requisitos y problemas que tiene el proyecto. Para luego hacer un plan que solucione el problema que plantea el proyecto. Además, analizar sobre como de eficiente es el programa creado. Una valoración sobre el trabajo realizado en torno a un presupuesto en caso de que hubiera sido un trabajo remunerado y no un trabajo académico. Además, también se mencionarán las colaboraciones que se ha tenido en la realización de este trabajo.

En el cuarto capítulo es el diseño de la solución. Es aquí donde se muestra de forma más detallada cómo se han implementado las soluciones a lo largo del tiempo. Además de las herramientas que se han ido utilizando para poder completar estas soluciones. También se le hará un análisis a cómo queda la estructura al final de la implementación y de métodos más utilizados.

En el quinto capítulo son los resultados. Donde se materializan los distintos casos de prueba que se han usado para comprobar que el proyecto es funcional y cumple con los objetivos establecidos. Aunque, también se discute sobre los posibles fallos que pueda tener al final o como queda el producto en comparación a las expectativas que se tenían respecto de este.

En el sexto capítulo son las conclusiones. Las cuales es una reflexión global de cómo ha ido el proyecto y reflexionar sobre este y sus objetivos. Además de hacer una relación de todos los conceptos que se han ido trabajando a lo largo del proyecto. Además de ver qué relación tiene con los estudios que ha recibido el alumno por parte de la escuela. Más tarde se sintetizarán que posibles incorporaciones o trabajos relacionados con este se pueden hacer a futuro.

Finalmente, está la bibliografía donde se apunta cuáles han sido los distintos puntos de información utilizados por el autor de la memoria y del trabajo.

2 Estado del arte

En los últimos años se han creado distintas tecnologías y metodologías que han favorecido a la industria del videojuego. A la par que la programación de estos mismos se hace de forma que se agiliza la eficiencia en tiempo de ejecución. Los sistemas más utilizados para la generación de los objetos y entidades tridimensionales son los denominados grafos de escena. Todas estas tecnologías también han aportado a otros campos y es más profundo de lo que se comentara en esta memoria.

No solo se mirarán las tecnologías que se va a utilizar como referencia y como se encuentran en la actualidad. También se examinará algunas de las posibles variaciones que puede tener el juego del ajedrez. Y cómo se encuentran el medio del videojuego en su estado actual respecto del mundo del ajedrez. Además de hacer una reflexión sobre lo que aportara el proyecto.

2.1 Sistemas y tecnologías

Como ya se ha comentado, las tecnologías más utilizadas hoy en día y de las que se va a hacer más eco en este proyecto son los grafos de escena que permitirán introducir tanto los objetos como personajes que se desean, al mismo tiempo que permitirán luego manejarlos e incluso hacerlos desaparecer en el futuro. Además, en este apartado también se comentarán algunas cosas sobre otras tecnologías que son de ayuda a lo hora de hacer este trabajo.

2.1.1 Grafos de Escena

Los grafos de escena son una estructura de datos que consiste en una colección de nodos y la descripción entre estos nodos, a los cuales se hace referencia con conexiones. Son claves en muchos programas gráficos, en los que el grafo de escena ayuda a manejar mejor los objetos y sus transformaciones asociadas. Un grafo bien construido y diseñado es fundamental para un manejo más sencillo de todo el programa. Los grafos de escena se pueden encontrar en su mayor medida en forma de árbol n-ario, donde hay un único nodo padre sobre una escena 3D. Mientras se esconde las características gráficas en los objetos y también, la aplicación de una transformación geométrica sobre un nodo concreto, afecta a todos los nodos hijos. He aquí, la utilidad de los grafos de escena, que permite una buena organización espacial para recorrer y organizar todas las estructuras graficas que se muestra en escena. Además, es robusto, sencillo de usar y escalable.

En adición a los nodos de transformación y representación también se pueden tener nodos que agrupan a otros nodos, como se puede apreciar en la figura 1(*BranchGroup*), donde comparten características comunes. Los cuales normalmente son de pertenencia, haciendo que todo el resto penda de este nodo en concreto. Definiendo así a un grupo en concreto dando nuevas formas de control sobre la escena.

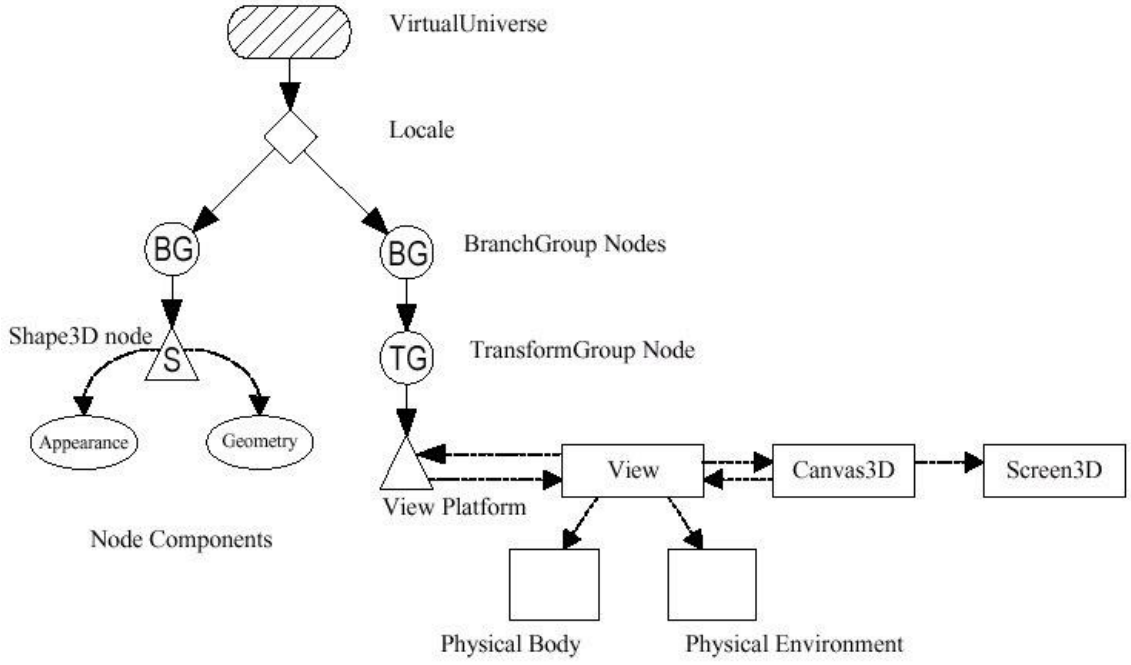

***Figura 1**: Representación del funcionamiento de un grafo de escena.*

Se pueden extraer distintos tipos de nodos de grafo dependiendo de qué particularidades componen ese nodo, existiendo unos conjuntos básicos los cuales no están todos representados en la figura 1. Estos conjuntos pueden ir desde nodos geométricos, los cuales almacenan la información poligonal de los objetos e información de apariencia. Pasando por los nodos de grupo, encargados de agrupar varios nodos hijos, de forma meramente de índole organizativa, hasta facilitar la jerarquía. Además, de los nodos de nivel de detalle, que seleccionan a sus hijos, basándose en la distancia entre el objeto con múltiples niveles de detalle y vista. También nodos de transformación afín, que permite aplicar matrices de transformación, que afectara a la ubicación espacial de sus nodos hijos, muy útiles para objetos móviles. Finalizando con los nodos conocidos como *switch,* que permiten realizar una selección entre los nodos hijos.

Los beneficios de utilizar los grafos de escena suelen ser varios. Partiendo desde un principio de mejora del rendimiento, ya que la selección es más ágil y permite que las reproducciones se hagan de forma conjunta. Puesto que de lo contrario tanto procesador como unidad grafica se podrían ver desbordados por una gran cantidad de datos. Además, esto se transforma en una mejora en la productividad. Puesto que solo se gestionan los bajos niveles gráficos en vez de miles de líneas de por ejemplo *OpenGL* a un número simple de llamadas. Por lo tanto, se pueden beneficiar de una portabilidad y escalabilidad añadidas, por no ser necesarios unas dependencias cuasi totales de las raíces. Puesto que cada grafo y nodo encierra gran parte de las tareas de muestreo y ser capaces de ser gestionados de forma dinámica. [1]

2.1.1.1 Open Scene Graph (OSG)

Es la librería más utilizada a nivel de uso de grafos de escena en el ámbito del software abierto. Es una herramienta grafica de alto nivel, que además es portable, diseñada para el desarrollo de aplicaciones graficas de alto rendimiento como pueden ser los videojuegos, realidad virtual, etc. También en aplicaciones científicas como simulaciones o visualizaciones. Se encuentra orientado a objetos y se construye a partir de las librerías de *OpenGL*, lo cual al final libera al programador o desarrollador de necesitar hacer llamadas graficas de bajo nivel. Provee de muchas utilidades adicionales que permiten un desarrollo mucho más rápido de la aplicación.

El núcleo de esta herramienta ha sido diseñado con solo unas mínimas dependencias del código C++ estándar y *OpenGL*, lo cual le permite ser muy portable a una gran cantidad de plataformas. Todo su código esta publicado bajo la *OpenSceneGraph Public License*[1] que permite la utilización modificación y distribución libre de la misma.

OSG se caracteriza por tener varias librerías que van desde el núcleo de esta que proporciona las clases básicas, pasando por los visualizadores y tratadores de texto y utilidades, hasta los manejadores de partículas y dibujo vectorial.

El propio motor que se utiliza en el proyecto, el UGK (UPV *Game Kernel*), se basa en llamadas a ciertas librerías y de OSG. Por lo tanto, se basa en las mismas directrices de uso de grafos de escena para la representación gráfica. Permitiendo utilizar esta forma de programación en el proyecto y encontrar una forma más eficaz y rápida de programarlo. Se podría utilizar en el videojuego puesto que es de licencia gratis, y no existen muchas librerías graficas focalizadas en los grafos de escena como esta.

2.1.1.2 Java3D

Es un tipo de grafo de escena basado en la 3D API de la *Java platform*. Esta también se ejecuta sobre *OpenGL* o *Direct3D* hasta sus versiones más actuales como es la 1.6.0, más concretamente sobre la versión de Java de *OpenGL* (JOGL). Su mayor característica es la utilización de grafos acíclicos dirigidos como estructura del grafo.

Java3D[2] [3] no es una utilidad más dentro de una API gráfica. Es una encapsulación de la programación grafica orientada a objetos. De esta forma, construye la escena haciendo uso de un grafo de escena, como representación de los objetos que se tiene que mostrar en la misma.

También tiene un amplio soporte para sonido especializado, que permite una mejor representación del sonido 3D en la escena. Compatible con todo tipo de formato estéreo o envolvente.

[1] http://www.openscenegraph.org/
[2] http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138252.html
[3] https://download.java.net/media/java3d/javadoc/1.5.0/index.html

Tendría como característica más loable para este proyecto, el hecho de que es multiplataforma, pero es solo compatible con programación en java.

2.1.1.3 Morphic Interface

Este es un sistema que suele ser utilizado en la programación web y cuya mayor característica es la utilización objetos llamados *morphos*.

Su relación con los grafos de escena es profunda, puesto que cada objeto Mórfico <<*Morph*>>, se puede entender como un nodo en un grafo de escena. El marco mórfico, es el encargado de gestiona la representación, de este tipo de grafo de escena. La aplicación construye un grafo de escena compuesto por *Morphs*. Entonces la aplicación modifica los <<*Morphs*>> en respuesta a los cambios en los objetos de dominio, que representan a estos <<*Morphs*>>. El marco luego actualiza la pantalla, al renderizar los <<*Morphs*>> modificados y cualquier otro <<*Morphs*>> que también deba volver a dibujarse, como los que se superponen con los <<*Morphs*>> modificados.

No es recomendable para este trabajo, puesto que no tiene mucho sentido usar una API con ciertas licencias, si el proyecto no va a ser cargado en un entorno en la red.

2.1.2 Interfaces Graficas

Para la creación de todo programa o aplicación gráfica es necesario la utilización de unos sistemas que permitan comunicar la aplicación con el hardware gráfico. Aquí es donde intervienen las interfaces gráficas o API gráfica. Algunas de las API más conocidas son las siguientes.

2.1.2.1 OpenGL

Una de las más utilizadas en todo el todo el mercado. Es de código abierto y multiplataforma. Permite soporte tanto en gráficos dos dimensiones como en tres dimensiones. *OpenGL*[4] cuenta además con dos versiones cada una especializada en un sistema distinto. El primero, especializado en sistemas empotrados, se considera un grupo de API diseñadas para su uso en dispositivos con menos capacidad computacional que otros sistemas como los ordenadores de sobremesa. Ejemplo de esto serían los teléfonos móviles inteligentes o las tabletas. El segundo de los sistemas estaría más especializado en páginas *Web*. El cual utiliza una base en JavaScript para la creación de sus gráficos en los navegadores *Web* sin la necesidad de instalar ningún software extra. [2]

[4] https://opengl.org/

OpenGL es utilizado por el API UGK para ciertas funciones visuales. Trabajando de forma que el programador en muchas ocasiones no necesita recurrir a sus métodos. De la utilización de *OpenGL* se encarga en gran medida el propio API UGK que sería el que internamente lo usa y luego facilitaría al programador los métodos resultantes. Aunque de vez en cuando también es necesario realizar ciertas especificaciones o llamadas a este por parte del desarrollador, si la implementación así lo requiere.

2.1.2.2 Direct3D

Patentada y desarrollada por *Microsoft*. Esta API es la más usada para realizar representaciones en sistemas operativos que sean propiedad de *Microsoft*. Es decir, sistemas como *Windows* o *Xbox System*. Es parte integral de *DirectX*[5], el cual es una agrupación de otras API que tienen la función de hacer representaciones multimedia para gráficos en dos y tres dimensiones. Básicamente la competencia de *OpenGL* y una opción que sería también igual de valida que esta. Anuqué en este este caso no se utiliza porque ya se usa *OpenGL*. Esto es debido a su uso escondido en el API UGK. Además, el proyecto no tiene la intención de utilizar API graficas directamente, sino que se tiene que utilizar el motor en el cual luego se encargue personalmente de usar estas API Graficas de forma oculta.

2.1.2.3 Metal

Metal[6] es una API de bajo nivel y baja sobrecarga, que sirve para el acelerado por hardware, de gráfico 3D. También tiene una función de cálculo de sombreado, dentro de unas funciones de aceleración, que son utilizadas de forma paralela al núcleo del programa. Fue desarrollado por *Apple Inc.* y que se estrenó en *iOS 8*. Metal combina funciones similares a las de *OpenGL* debajo del funcionamiento de su API. Está destinado a brindar a todos los sistemas de *Apple*, de algunos de los beneficios de rendimiento de API similares para otras plataformas, como puede ser el ejemplo de Vulkan y DirectX 12. Obviamente no utilizada en este proyecto porque se utiliza Windows, esto mejor explicado en el capítulo 4.

[5] https://docs.microsoft.com/es-es/windows/desktop/directx
[6] https://developer.apple.com/metal/

2.2 Actualidad del ajedrez

Volviendo al tema principal, el juego del ajedrez se podría decir que siempre ha estado presente desde el día de su creación hasta nuestros días. Pasando a lo largo del tiempo por distintas fases, como podría ser la de un juego para los intelectuales, hasta su denominación como deporte. Finalmente, como medio de entretenimiento lúdico y casual, para un público mucho mayor que el que podía tener en el pasado. Esto se debe en gran medida a las tecnologías de la información, que han hecho que sea mucho más asequible. Debido a que no siempre hay que realizar un desembolso de capital, no solo para adquirir el juego físico en sí, sino también, en algunos casos para jugarlo. Ahora es algo trivial como descargarse una aplicación que reproduzca este juego de la forma que sea.

Muchas aplicaciones que mueven el juego del ajedrez se basan normalmente en el juego original, aportando algo mínimamente distinto como podría ser la ayuda de un jugador virtual o distinto al propio usuario. Esto también implicaría a una inteligencia artificial en caso de que la soporte y que pueda ofrecer distintos niveles de dificultad. Aunque también podrían contener otras variables que se comentaran a continuación.

2.2.1 Ejemplos de variaciones en juegos de ajedrez

En la actualidad existen distintas variaciones[7], algunas de las cuales quedan registradas en libros o páginas web. Estas variantes no tienen que seguir los estándares prefijados del juego del ajedrez y cada una cambia de alguna forma el juego. Por ejemplo, no se tienen ni que tener el mismo tablero que tendría el juego original en este caso. Un ejemplo de esto sería el de los tableros hexagonales en vez de cuadrados como en la siguiente figura:

Figura 2: *Imagen de la variante hexagonal del ajedrez*

[7] http://www.chessvariants.com/

En el cual las reglas de desplazamiento no serían iguales a las del ajedrez convencional. Aunque se pueden realizar otros cambios interesantes sobre el mismo concepto de tablero cuadrado sin necesidad de cambios de forma, pero si pudiendo cambiar configuración y piezas a añadir como es el caso de esta variante:

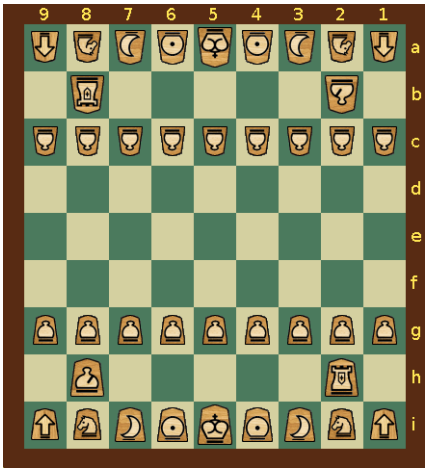

Figura 3: *Representación de la variante que se asemeja al ajedrez japonés*

La variante del ajedrez que se puede apreciar en la figura 3, es una versión occidental copiando la distribución y propiedades del ajedrez japonés o *Shogi*. En el cual se empiezan con un rey, una torre, un alfil, dos <<generales dorados>>, dos <<generales plateados>>, dos caballos, dos lanceros y nueve peones. Aunque no solo de cambios de piezas se puede tratar el juego, sino que, además se puede remodelar las reglas de forma que no se pueda apreciar que es un juego del ajedrez a simple vista, sino que se convierte en otro juego manteniendo ciertas similitudes con el original.

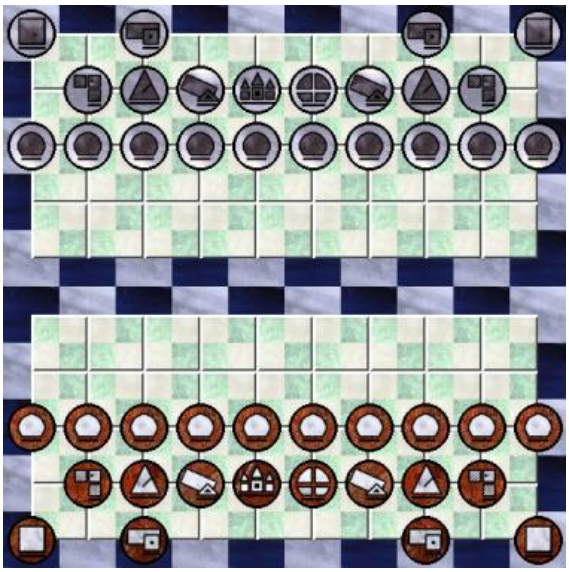

Figura 4: *Representación de la variante indoeuropea del ajedrez*

En la figura 4 se puede apreciar que tanto distribución, como tablero y fichas no son como las reglamentarias y cada una además se coloca en la intersección de los cuadrados no en el cuadrado en sí, a la par que el tablero tiene una separación central entre un ejército y otro con el que tienen que interactuar estos dos para pasar de un lado al otro.

Existen otras variantes que también dejan pocas fichas en el tablero o modifican este para que sea como el mapa de un lugar, plano de una casa, estructuras tridimensionales, etc. Existen muchas variantes que se pueden localizar en internet y otras que hasta han salido en el cine. Pero todas con la misma característica de enfrentar a dos ejércitos o a veces más, por obtener la victoria. Donde siempre cuando uno mata o se come a la ficha más importante o representativa del tablero, que suele ser el rey, el otro pierde.

2.2.2 Similitudes entre el proyecto y la actualidad

En estos momentos solo con una búsqueda por internet es fácil encontrar el juego del ajedrez de forma digital. No solo es que ya venía en algunas versiones de *Windows* como juego de prueba, sino que en varias páginas web que se focalizan en ofrecer juegos de navegador, también se puede encontrar el juego del ajedrez de forma fácil y accesible para muchas personas, la de la figura 5. De esta forma no es complicado, para estas, echar una partida con el único requerimiento de una conexión a internet. Aunque en los últimos años y con el crecimiento de los *Smart Phones*, se ha generado un aluvión de aplicaciones para el teléfono móvil. el cual ahora tiene también, una extensa variedad de aplicaciones que mueven el juego del ajedrez en el teléfono móvil, sin mucho esfuerzo añadido para los dispositivos.

El proyecto se encuentra aproximadamente en esta situación, puesto que se desarrolla una aplicación, que, aunque no es para móvil, permite al usuario utilizar todas las herramientas que le brinda el ordenador para poder jugar una partida al ajedrez con otra persona. Puesto que el multijugador es una fase muy importante en todo juego de mesa. Las aplicaciones que se han ido fomentando a lo largo de este corto espacio de tiempo han sido enfocadas en aprovechar la conectividad de internet, que se tiene ahora, para permitir que mucha gente se conecte a lo largo del mundo en pos de jugar unos contra otros.

Aunque en este proyecto no se va a brindar una funcionalidad a través de internet. Se puede apreciar cómo se permite que el juego cumpla con todas las reglas de las que dispone el juego del ajedrez. Implementado la aplicación para poder jugar entre dos personas simultáneamente, algo que es lo común en toda aplicación que se precie de este juego. Donde muchas otras coinciden con la de este proyecto es en permitir realizar partidas en tiempo real de juego, respetando los pertinentes turnos de cada uno y garantizando que cuando uno se alce vencedor. Haciendo saber, ya sea por el método pertinente, quien ha ganado y que se acaba la partida.

Figura 5: *Imagen de un videojuego cualquiera que simula el juego del ajedrez*

2.3 Critica al estado del arte

Se ha mostrado, que en la actualidad hay una gran variedad de posibles variantes en torno al juego del ajedrez. Además de como con las tecnologías de la información cada persona es capaz de jugar con otra que se encuentre en otra parte del globo terráqueo. Pero sigue siendo en casi todos los casos, una aplicación estática que solo permite realizar el juego del ajedrez tal y como fue concebido, sin dejar a los propios jugadores o usuarios elegir.

Casi ningún producto relacionado con el ajedrez a día de hoy permite a sus usuarios modificar nada respecto del propio juego en sí. Son rígidos y poco flexibles y visualmente muy similares entre todos ellos, como se puede apreciar en la figura 5. Además, en ocasiones parecen copias unos de otros, en incluso mecánicas como la de poner un listado de movimientos o como notifican a los usuarios toda acción realizada anteriormente. Pero por encima de todo y lejos de la interfaz, no permiten jugar a nada más que no sea el juego clásico del ajedrez, un juego que tiene más de cien variantes y que puede expandirlas infinitamente.

Las limitaciones a la hora de personalizar por parte del usuario son palpables. No tienen, en muchos casos, a la hora de querer probar nuevas formas de jugar al ajedrez, que simplemente cambiar la dificultad. Siendo que estos propios usuarios son luego los que podrían diseñar una nueva forma de jugar al ajedrez. Pero no serían capaces de ponerlo en práctica si no tienen un tablero físico propio o desarrollan ellos desde cero una nueva aplicación para ello.

La carga de trabajo para alguien que quiera jugar de otra forma al juego del ajedrez que no sea la estándar, suele ser más alta de lo normal y solo impide que no se pueda experimentar y difundir otras formas de jugar.

Haciendo que al final todas estas nuevas variantes se pierdan en un parcial olvido y que no se pueda innovar en algo tan poco estático como tendría que ser un juego. Puesto que la introducción de nuevas formas no acabaría destruyendo las que ya se encuentran, incluyendo la estándar.

2.4 Propuesta

Por los problemas comentados anteriormente, el proyecto realizado tiene como base principal la de permitir configurar al gusto del usuario, como quiere que sea la partida que va a realizar y como la distribución dentro de la misma. Es hacer partícipe al usuario a la hora de la personalización, permitiéndole que pueda realizar la partida como a él más le guste y por ello pudiendo innovar más en todo que sea modificable.

Entonces, no solo se tiene la idea de que el usuario cambie la posible distribución. Si no que tenga un abanico de posibilidades, que luego realicen un cambio sustancial en cómo se desarrolle el juego. Por ejemplo, cambiando tanto las posiciones de las piezas en el tablero, como también permitir que este tenga la forma deseada y que no sea el típico cuadrado de ocho por ocho que tiene toda variante estándar del ajedrez. Además, permitir que luego todo elemento visual de la escena también sea moldeable y permitir a los usuarios, si quieren, una visualización tanto más plana, como más tridimensional.

En definitiva, que la apariencia tanto visual como estratégica del juego, no sea la convencional. Por lo tanto, esta aplicación sea posible de ampliar a más ámbitos dentro del ajedrez y de la jugabilidad en sí. Para que no solo sea una aplicación lúdica, si no también ser educativa al final del ciclo de vida de esta.

3 Análisis del problema

Aunque en el estado del arte se ha podido apreciar cual es el motivo por el que se ha creado este proyecto. Un análisis más exhaustivo y sintetizado del problema, permite luego explicar cuál ha sido el trabajo a realizar.

3.1 Análisis de requisitos

Es necesario desglosar cuáles son los apartados que debe cumplir el proyecto terminado para satisfacer sus las necesidades del consumidor. En estas se basarán tanto en la funcionalidad como en la interacción usuario juego.

3.1.1 Funcionalidad

La funcionalidad es la parte en la que al cumplirse esta, permite el buen funcionamiento de la aplicación. Por lo tanto, no existen errores respecto de los conceptos básicos que tiene que cumplir la aplicación, descritos a continuación.

- Modernizar como tal la aplicación anterior con el mismo nombre. Es decir, que es funcional en con librerías y sistemas actuales.
- La aplicación tiene que ser capaz de leer los archivos *.html* de parametrización que se encuentran en las carpetas de *configuration* y *levels*. Además de hacer cumplir los parámetros que esos archivos se le especifique.
- Ser capaz de cargar todos los elementos de autor como son modelos, texturas, etc. Permitiendo ser visualizador en la escena del videojuego. Además, que puedan ser cambiados en los directorios y que representen mínimamente como se tendría que ver en la escena.
- Las piezas se mueven con fluidez por la superficie del mapa/tablero y realizan los movimientos pertinentes de cada pieza en concreto. Además de ser capaces de acabar con otras piezas y estas desaparecer de la escena.
- Utilizar las herramientas de la API UGK de la universidad y no tener errores durante la ejecución. Además, ser capaz de utilizar gran parte de las funcionalidades que ofrece permitiendo así insertar o eliminar objetos del grafo de escena sin que el usuario se percate de ello.

3.1.2 Interacción usuario juego

La parte de interacción es la de requisitos que tiene que cumplir el proyecto para que el usuario se sienta más cómodo al utilizar la aplicación.

- Tener una interfaz mínima, con una ventana, que permita al usuario interactuar con ella. Pero sin necesidad de hacer todos los cambios por consola o por los archivos modificados.
- Haber alguna forma de puntero en pantalla. Lo cual permita al usuario, utilizando únicamente un periférico, mover las fichas.
- El usuario ser capaz de interactuar con la cámara. Pero hacer de una forma u otra que los cambios de perspectiva no modifiquen el juego como tal.

3.2 Análisis de soluciones

Existen distintos tipos de soluciones al problema principal. La modernización de la aplicación anterior se debe de ver desde distintos puntos de vista. Puesto que la mejor solución no queda clara. Teniendo en cuenta que hay que modificar una aplicación que no gastaba para nada la API UGK y ahora sí la tiene que utilizar.

Para este proyecto se pensaron en tres soluciones posibles. Todas ellas validas, aunque cada una con un distinto tipo de dificultad que al final determina cual es la escogida. Pero eso se aclarará en el siguiente subapartado.

Primera solución. Teniendo en cuanta que ya existe una aplicación anterior hecha y que esta era funcional en su momento. La solución pasaría por actualizar todas las librerías que existen y hacer las modificaciones pertinentes en el código fuente que permitieran introducir la API UGK sin estropear las partes que ya funcionan. Además, modificar los archivos para reducir las funcionalidades que fueron implementadas por código propio y ser así más dependiente del UGK.

Segunda solución. Crear una base ya diseña con la API UGK, que implemente los conceptos más básicos que tendría que tener una aplicación. Para que fuera operativa. Donde más tarde se le irían añadiendo los distintos archivos de la aplicación original y donde pondrían las librerías actualizadas. Manteniendo todo aquello que es juego y dejando la funcionalidad solo en manos de la implementación previa.

Tercera solución. Teniendo en cuenta que la API UGK ya ha sido utilizada en anteriores proyectos. Además, es una aplicación de uso académico, que permite a los alumnos aprender a utilizarla con juegos no muy difíciles de modificar. Por lo tanto, otra opción es la utilización de otro juego que ya usé la API UGK y remodelarlo. Utilizándolo como base y luego ir modificando los archivos existentes para que el funcionamiento sea como la aplicación deseada. A la par que se le añade algún archivo de la aplicación anterior que modificara una funcionalidad únicamente del juego y no de la infraestructura.

3.3 Solución propuesta

La solución que finalmente se escoge entre las tres anteriores es la tercera. Esto es debido a una serie de factores que tienen que ver con el tiempo, el aprendizaje, la facilidad y funcionalidad. Además, luego se explicará en mejor profundidad que consiste y que contiene la solución seleccionada.

3.3.1 Motivos de la elección

Aunque en principio se intentó realizar la primera solución porque se veía la solución más rápida a priori. Luego esta pasó a remarcar que la cantidad de fallos que tenía por su desactualización y la dificultad de introducir una infraestructura basada en UGK sobre una aplicación que funcionaba de forma totalmente distinta. Se pensó entonces en realizar la segunda, pero otra vez el tiempo influencio en la decisión, puesto que es un proceso largo el de crear la infraestructura desde cero y sin ningún referente. Es por esto que tener un referente que funcionaba y que incorporaba la aplicación era una forma más rápida de desarrollar el proyecto.

La fase de aprendizaje también fue relevante a la hora de escoger, puesto que la aplicación anterior se caracteriza por utilizar librerías que el desarrollador no había utilizado nunca antes. Además, como ya habían quedado descrito antes, la aplicación estaba desactualizada y la forma que tenía el programa de utilizarlas también era muy distinto a cómo funciona en la actualidad. La tercera solución no necesitaba de fases largas de aprendizaje puesto que ya se está familiarizado con el sistema, el cual no es complicado de entender.

Por ende, esto conlleva a comprender que el nuevo sistema no es solo más fácil de aprender, sino que además su funcionamiento es más sencillo. Esta facilidad que tiene la infraestructura en UGK, puede provenir de que está hecho para la enseñanza. Las formas en las que se reestructuran las distintas funciones y métodos a la par que los objetos, hacen muy intuitiva la utilización del mismo.

Finalmente, la aplicación elegida para la tercera solución, ya era funcional antes. Esto hace que no se necesite un proceso de solución de errores de compilación ni de ejecución previos en un principio. La adhesión de nuevos elementos y el cambio de los ya existentes con anterioridad ocasionara otros errores. Pero se descarta la posibilidad de que el fallo sea la base y no los archivos modificados, por lo tanto, añade seguridad a la hora de la implementación, puesto que se está más consciente del trabajo hecho.

3.3.2 Especificaciones de la solución escogida

La solución escogida como ya se ha mencionado antes es la tercera de entre todas las presentadas anteriormente. Esta es la de reconstruir el videojuego de *HyperChess* usando como plantilla un videojuego que ya utilizara la API UGK, en este caso el videojuego a usar es el *Space Invader* el cual ya es utilizado en la asignatura de <<introducción a la programación de videojuegos>> impartida en esta universidad.

La solución es una reestructuración del propio juego plantilla y la adhesión de los elementos que caracterizaban la aplicación de *HyperChess* al nuevo formato. El juego plantilla cuenta con un sistema en forma de capas que tiene una serie de recursos o <<*characters*>> que son creados, modificados, insertados o eliminados de la escena, por las distintas fases y sistemas de control que tiene el juego. Donde cada <<*Character*>>, además de basarse en un modelo hecho en la API UGK, también tiene sus funciones propias a la par que es capaz de interactuar con las capas y con otros como él.

Los inconvenientes que tiene este sistema son, que el funcionamiento de la aplicación plantilla no es idéntico al que puede tener la aplicación resultado de la reconstrucción por lo tanto algunos archivos se tienen que modificar. Otro inconveniente es que hay que añadir todos los elementos que pertenecen al propio juego de *HyperChess* y que van a ser necesarios, pero no son compatibles con la nueva infraestructura. Además, que los archivos incensarios, puesto que no tiene nada que ver con el juego final, tienen que ser eliminado y la vez que también se eliminan otras funciones que solo interactuaban con ellos.

Las posibles soluciones para estos problemas pasan por especificar cuáles son los nuevos parámetros que van a necesitar los objetos que se modifican. Un ejemplo es la inclusión de un identificador de color entre los <<*Characters*>> que son piezas. Al mismo tiempo, la forma en la que tiene que distribuirse por la escena las piezas, no es igual a como se le indicaba en la configuración, cuando eran naves espaciales. Anadir también los nuevos archivos como puede ser cada pieza en particular o la especificación base de pieza en sí misma. Pueden estar basadas en otros ejemplos que ya estén en el juego y que de esta forma se haga más fácil la compatibilidad cuando son introducidos. Ejemplo de esto puede ser basar la estructura que tenía ya alguna nave en particular para luego añadirle todos los elementos que tiene una pieza especifica. Finalmente, la eliminación de todo el contenido innecesario es algo que se tiene que hacer simplemente quitando los archivos o librerías pertinentes cuando estos ya no sean necesarios y quede claro que ya han sido sustituidos por las nuevas.

Es por estos casos que el desarrollo pasa por una serie de fases necesarias para la solución de los problemas, mejora de la eficacia a la hora de implementar el código y simplificación de las tareas.

En primer lugar, está la copia de la gran mayoría de los archivos del videojuego plantilla y la reestructuración de estos en el proyecto para que ningún fallo de compilación pueda ocurrir. Siempre pensando que todo elemento no tiene que hacer alusión al juego plantilla por lo que también habrá que renombrar algunos elementos en el proceso.

En segundo lugar, modificar todo elemento que se pueda utilizar pero que en su momento es demasiado especifico o hace demasiadas alusiones al juego plantilla y no es compatible con lo que se quiere introducir a continuación.

En tercer lugar, añadir todos los nuevos elementos que son necesarios y que construyen la nueva funcionalidad que se quiere diseñar. Como ya he mencionado antes, no es necesario copiarlos totalmente, puesto que no funcionaran, pero tampoco empezar de cero. Se permite crear un estado intermedio entre recuperación de contenido de la aplicación vieja, un poco de inspiración de la plantilla e innovación propia del desarrollador.

En cuarto lugar, esta referenciar todo elemento nuevo que se ha añadido al proyecto y además hacer posible ciertas comprobaciones de que estos elementos nuevos interactúan con las distintas capas como es necesario en un estado controlado. Ejemplo de esto es la inserción de la pieza <<peón>> y luego más tarde utilizar los archivos de configuración para apreciar que se encuentran o se insertan en escena.

En quinto lugar, se encuentra la eliminación de todos los contenidos sobrantes que ya no serán necesarios en el futuro proyecto. También se tendrá que asegurar que la eliminación de esto no genere errores y por lo tanto corregir los que surjan en este preciso momento.

En sexto lugar, se encuentra otro apartado que no es excluyente del anterior, ni necesariamente consecutivo de ese, siendo posible encontrarse este antes. Pero se realizará después para tener una implementación más fluida. Por tanto, es la de terminar de implementar todas las funcionalidades que tiene que realizar todos los elementos, para que el proyecto funcione como tiene que ser. Es la parte en la que se espera que el proyecto quede pulido como modo final.

3.4 Análisis de eficiencia algorítmica

El programa ejecuta toda su fase grafica con la forma de un grafo de escena. Esto en el sentido de la implementación, lo hace más fácil, pero a la par también lo hace más eficiente. La utilización de grafos de escena, como ya ha quedado registrado con anterioridad, permite que las transformaciones u otras acciones sobre un nodo padre se trasmitan a los nodos hijos. Pero esto lejos de ser solo una forma de simplificar la programación también simplifica la computación puesto que solo uno tiene que hacer el cálculo y todos los otros seguirlo tal y como ya estaban. Esto simplifica las operaciones que tendría que realizar el ordenador, puesto que no todos los nodos se tienen que calcular al unísono.

Sin embargo, aunque se quiera simplificar la ejecución del programa usando un mejor sistema gráfico. No sería recomendable usar esta aplicación en dispositivos que tengan una capacidad muy baja. Puesto que los sistemas de mensajería, contacto y bucle principal, que mantienen todas las piezas. Podría hacer que un número alto de envíos, recepciones, interacciones y cargas en grandes cantidades, no fuera procesado a la velocidad deseada.

3.5 Trabajo y presupuesto

El trabajo a realizar en este proyecto, no es un trabajo remunerado como tal. Pero se puede analizar tanto las horas dedicadas a cada tarea en particular, como las dedicadas en general. Además, realizando al final una valoración de cómo sería el trabajo completo y asignarle un valor en proporción a todo el esfuerzo realizado, como si de un trabajo remunerado se tratase.

3.5.1 Plan de trabajo

Se analizarán las distintas partes del trabajo realizado y al mismo tiempo se le asignara una cantidad de tiempo que se pensaba sería necesaria para realizar esta función.

3.5.1.1 Fase de preparación

- **Análisis de la API UGK:**
  Donde se espera obtener mayor conocimiento de cómo es el funcionamiento de la aplicación base que se desea utilizar para el desarrollo del videojuego. En un principio esto no tendría que ser muy complicado, pero cuando se realiza este trabajo, la aplicación también está en constante cambio. Por lo que es posible alguna modificación en cierto momento durante el propio desarrollo.
  Teniéndolo en cuenta, se estima un tiempo de catorce horas promedio.

- **Analizar aportaciones OSG:**
  Como el sistema de OSG está escondido en lo profundo de la aplicación base, no se cree muy necesario tener un estudio exhaustivo sobre este grafo de escena. Aunque este también se está interacción se esté modificando en el momento de realizar este proyecto. Se estima un tiempo promedio de cuatro horas.

- **Instalación de las aplicaciones:**
  Se necesitan una serie de aplicaciones para la implementación y compilación del videojuego, por lo que se estima el tiempo en función de lo que se tarde en descargar e instalar dichas aplicaciones. Este tiempo no está sujeto a las capacidades del programador, sino a la velocidad de descarga de la línea estándar y de la velocidad de procesador a la hora de instalar. Se estima un tiempo promedio de ocho horas.

- **Copiado del proyecto plantilla:**
  Esta tarea consiste únicamente en la cargar de todos los archivos que son del proyecto plantilla, en el directorio donde se piensa crear el proyecto de este trabajo. Es necesario no copiar los archivos que son únicamente usados por *visual studio* para la creación y administración del proyecto, puesto que estos se crearan luego. Por lo que hay que ser selectivo a la hora de copiar. Se estima un tiempo promedio de dos horas.

- **Creación del proyecto:**
  Es la parte en la que se crea el proyecto con la aplicación de *visual studio* y todo lo que conlleva. Es decir, la creación de la <<*solution*>> y de todos los archivos del proyecto que son necesarios para su funcionamiento. Además, ordenar internamente en el programa todos los archivos en los distintos directorios ficticios dentro de la <<*solution*>> y el referenciación a donde se encuentran las librerías. Finalizando con la configuración del proyecto. Se estima un tiempo promedio de seis horas.

- **Instalación y actualización de librerías:**
  Se necesitan una serie de librerías de distintos tipos para el buen funcionamiento y compilación del videojuego. También, teniendo en cuenta que las librerías de UGK, pueden sufrir modificaciones durante el desarrollo. Se estima un tiempo promedio de cuatro horas.

3.5.1.2 Fase de Implementación

- **Modificado de los archivos plantilla:**
  Es la parte en la que se modifica todo el código de los archivos internos, donde se hacía referencia a la aplicación plantilla. Donde se transforma o borra dependiendo del caso, todo elemento o funcionalidad que fuera muy específica de la aplicación plantilla y que no va a servir en el futuro. Un ejemplo muy simple, es el de renombrar toda referencia que hay hacia objetos que luego se pueden usar, pero luego tendrán que tener otro nombre. Se estima un tiempo promedio de seis horas.

- **Inserción de las nuevas funciones del juego:**
  Es la creación de nuevas funcionalidades que solo valla a tener el nuevo proyecto y que no tuviera el proyecto plantilla. Estas son las que más tarde van a ser necesarios cuando se añadan los distintos archivos que son específicos de este juego. Esta parte siempre puede aparecer en cualquier parte del desarrollo puesto que a medida que se desarrollan estos archivos también se desarrollan estas nuevas funcionalidades para los viejos. Se estima un tiempo promedio de catorce horas.

- **Inserción de los nuevos archivos y personajes:**
  Esta parte consiste en la de insertar los nuevos archivos en el proyecto. Aunque, no todos son creados de cero y pueden ser importados del viejo proyecto o basados en algún archivo de la aplicación plantilla, pero en mucha menor medida. También, en esta fase es cuando se pasa por todas las capas que existen en el proyecto y se va referenciando o añadiendo como partes funcionales, dependiendo de que hace cada capa, los nuevos archivos. Un ejemplo para que se entienda, seria donde en la parte de los analizadores se referencia una pieza y se le dice a ese analizador que hacer cuando lea algo de esa pieza en particular. Se estima un tiempo promedio de veinte horas.

- **Implementación de las funciones específicas:**
  Se trata de desarrollar todos los archivos de específicos y sus funciones nuevas. Permitiendo al juego realizar las funciones que tiene que desempeñar al final del desarrollo del proyecto. Esto es mucho más costoso y la base de la programación. Puesto que se crean funciones nunca hechas y el resultado debe de ser funcional. Ejemplo de esto seria las funciones que hacen moverse a las piezas en el mapa, teniendo en cuenta que pieza es y las restricciones que eso conlleva. Aunque también teniendo en cuanta que la forma en la que se desarrolla tiene que ser mínimamente eficiente para no crear algo inmanejable. Se estima un tiempo promedio de veinticuatro horas.

- **Crear o modificar archivos de autor:**
  Este apartado va a centrarse en adaptar todo el contenido de autor, es decir todo lo que son mayas, sonidos, imágenes, etc. Todo lo que haga falta para el nuevo proyecto, incluyendo nuevos archivos *.html* que configuren las nuevas directrices que tiene que tener el trabajo o nuevos niveles, por ejemplo. Esta parte es más artística en algunos casos y más funcional en los casos en los que se quiere comprobar algo. Se estima un tiempo promedio de veinte horas.

3.5.1.3 Fase de depuración:

Esta fase no está en contraposición con la fase de desarrollo, puesto que en muchas ocasiones se realizan acciones de esta, mientras se está realizando al mismo tiempo una implementación. Aunque aquí se expondrá como si fueran una sección final para no complicar conceptos.

- **Eliminación de todo contenido innecesario:**
  En este punto se enmarcaría todas las acciones de borrado que se realizarían tanto en archivos como en referencias. Donde no se afectaría al buen funcionamiento de la aplicación, ni por errores de compilación, ni por errores de ejecución, ni por la pérdida de funcionalidades. En este apartado entran todo tipo de archivos, desde los que podían todavía existir de la aplicación plantilla, hasta esos archivos que en un principio se iban a usar y al final han sobrado. Se estima un tiempo promedio de seis horas.

- **Eliminación de errores:**
  Esta parte, más que centrarse en la corrección de errores de programación o compilación, aunque no menos importantes. Se tiene que focalizar en encontrar y solucionar todo error de ejecución que pueda ocurrir en el videojuego. De forma que el jugador no encuentre ningún problema a la hora de realizar las fases de carga o cualquier acción sencilla. Se estima un tiempo promedio de ocho horas.

- **Comprobación del videojuego:**
  Esta parte es en la que se ejecutara el videojuego y se revisara que todo es correcto. Por otra parte, se puede tomar capturas de su funcionamiento, para mostrar luego como es, al público. Se estima un tiempo promedio cuatro horas.

3.5.2 Calculo de tiempo

El tiempo total, sería la suma de todos los tiempos que se han estimado, en el conjunto de todas las tareas a realizar. Todo ello basado en los promedios de tiempo que se han estimado a priori, pensando en cómo iría el trabajo.

La suma total de esta estimación ascendería a unas 140 horas. Teniendo en cuenta una valoración optimista del recorrido a la hora de implementar el proyecto.

En la figura 6, que representa el diagrama de Gantt de esta planificación, se puede apreciar las distintas fases y como se espera que se desarrollen, sin mayor complicación. Todo medido en horas dobles, puesto que es múltiplo de dos, para que no sea demasiado largo.

3.5.3 Calculo del presupuesto

El cálculo del presupuesto, en caso de que fuera un trabajo remunerado, sería la suma del coste de compra de las aplicaciones, gastos adjuntos y las horas de trabajo.

En este caso, como las aplicaciones que se han utilizado son gratis y no hay ninguna licencia de ningún derecho que comprar. Los gastos pasan a ser directamente las horas de trabajo, más una pequeña cantidad a modo de colchón de gastos por si ocurre cualquier imprevisto. Es decir, teniendo en cuenta que el valor medio de un informático son unos 7€ la hora. El cálculo estimado de coste sería de unos 980€ por el tiempo consumido, más una pequeña cantidad de 40€ como colchón de gastos.

También se tendrían que tener en cuanta espacio y mantenimiento de este. Teniendo en cuenta que un piso despacho o piso barato, se alquila de media por unos 230€/mes y que se estima que el proyecto dure cuatro semanas. Entonces este precio también se le tiene que añadir. Además de gastos de luz que se redondean en unos 88€/mes por tarifas estándar. Por lo tanto, también habría que añadir este gasto.

Quedando en una cantidad que si se tiene en cuanta todos los gastos ascenderían a un total de unos 1338€.

figura 6: Diagrama de Gantt de la planificación original

3.6 Colaboración

Menciones a la colaboración que se ha tenido a lo largo de este TFG (trabajo final de grado) con otro TFG realizado por el alumno de esta misma escuela, Miguel Medina Patón.

Puesto que simultáneamente a la creación de este trabajo, el suyo consistía en la modificación del propio API UGK, para la inserción del sistema OSG en el ya mencionado.

Por lo que, todos los cambios realizados por el compañero también han sido transmitidos durante este desarrollo, para el mejor funcionamiento de este. Gracias a la utilización de una herramienta de control de versiones, la cual esta especificada en el capítulo cuatro. A la par que se mantenía contacto vía internet con el correo de la UPV. Para conocer que había hecho cada uno. Aunque al final no haya habido una colaboración directa entre las partes dentro del proyecto.

4 Diseño de la solución

Las herramientas que se han utilizado a lo largo del desarrollo, son una serie de utilidades, bibliotecas y software de diseño o compilación. De los cuales se hablará en profundidad en este capítulo. Donde además se remarcará como fue estructurado el proyecto y como funcionaria internamente en mayor o menor medida.

4.1 Análisis de las herramientas

Como ya se ha comentado, todas las herramientas utilizadas a lo largo del desarrollo del proyecto han tenido una determinada funcionalidad y por tanto aquí se explicará porque se escogieron y en que han servido. Analizadas cada una dentro del ámbito en el que se encontraba.

4.1.1 Sistema operativo utilizado y lenguaje de programación

El sistema operativo que se utilizó durante el proyecto es el de *Windows*. Puesto que el proyecto tampoco dejaba más opciones. Esto es debido a que el propia API UGK no es compatible a día de hoy con otros sistemas operativos que no sea *Windows*. Por lo que no solo el desarrollo es en ese sistema operativo, sino que también el producto final va destinado a ese sistema operativo. Incapacitando cualquier otra alternativa posible.

Dos partes de los mismo pasaría con el lenguaje de programación utilizado. Se ha utilizado el lenguaje de programación C++ para todo el proyecto. Puesto que UGK está programado en C++ y no hay más remedio que utilizar este. El cambio de lenguaje podría ser realizado. Pero es demasiado costoso, puesto que habría que hacerlo todo de nuevo, y algo que a fin de cuentas es innecesario. Como lenguaje de programación es eficiente y el desarrollador ya lo conoce, por lo que no tiene que aprender nada nuevo de este.

También se utiliza otro lenguaje de programación como auxiliar para la configuración. Este sería el lenguaje HTML. Esto sería debido a que la modificación de los archivos en lenguaje HTML no es muy complicada. Puesto que se puede hacer hasta con el propio blog de notas de *Windows*. Pero sobre todo se utiliza, porque es muy intuitivo entender cómo se organizan las directrices de configuración dentro de estos. Además, el propio UGK ya tiene el analizador que lee este tipo de archivos y los convierte en información para el videojuego.

4.1.2 Entorno de desarrollo

Para el desarrollo del proyecto se ha utilizado *Visual Studio*. Esto es debido en parte a que el videojuego original del *Hyperchess* estaba desarrollada en *Visual Studio*. Otro motivo es que el videojuego plantilla y el propio API UGK también se han implementado en *Visual Studio*. Por esto ya se tiene un punto solución y también la forma de la estructura interna del proyecto. Todo utilizando el *Visual Studio*. Esto hace que utilizando el *Visual Studio* no sea necesario realizar otra estructura de los archivos, ni ninguna configuración nueva. A su vez, este entorno de desarrollo ofrece muchas utilidades que pueden ser muy beneficiosas a la hora de implementar la aplicación. Algunas de estas son:

- Detección de errores de compilación durante la propia fase de escritura. A la par que también detecta algunos errores que se pueden dar en la fase de ejecución y otros errores que pueden crear problemas, como es el de referencias mal usadas en una lista.
- Sistemas de autocompletado. Que permiten al programador ahorrar tiempo y esfuerzo en escribir los métodos. Además, que también permiten a este no necesitar recordar cómo se llaman todos los métodos, atributos, definiciones, etc. Ni tampoco conocerlos todos.
- Compila el programa y lo ejecuta en un entorno seguro y controlado. Donde se le comunica al programador que excepciones han ocurrido durante la ejecución de la aplicación y de qué tipo son.
- Permite insertar puntos de interrupción en el código. Los cuales permitirán al programador saber si la aplicación realiza lo que se esperaba de ella.

También se ha utilizado para la administración de la versión el servicio de *SubVersion* o SVN, más concretamente el *TortoiseSVN*. Este servicio permite a través de la red, cargar o descargar las versiones y modificaciones que se hayan hecho en los archivos y el código del proyecto. También permite recuperar versiones anteriores si la modificación aplicada no fue una solución eficaz. Este servicio está integrado en el explorador de *Windows* y por tanto su manejo es como el del propio explorador. Por tanto, no necesita de un aprendizaje específico para él. Además, esta tan integrado que es consciente de todas las modificaciones que se hacen con el Visual Studio y viceversa. Lo cual simplifica a la hora de hacer modificaciones.

4.1.3 UPV Game Kernel

La base del API UGK es la de la aproximación computacional a un grafo de escena. Puesto que en verdad no es un grafo de escena como tal. Ya que es en verdad una doble cola que almacena punteros de una clase vital para todo el API como es la *UGKCharacter*. Siendo esto lo que forma la escena como tal. Donde cada llamada general a un método como puede ser el *update*, lo que hace en verdad es recorrer la cola sucesivamente y llamar al método de cada personaje en concreto. A la par que también se encarga de ir recorriéndola para encontrar los elementos que han dejado de estar activos e ir purgándolos.

4.1.3.1 Personajes

En UGK los personajes son construcciones de la clase *CCharacter* de *UGKCharacter*. Donde cualquiera de estos es una entidad que existe en el propio videojuego con una serie de funciones. Pueden ser desde personajes interactivos, pasando por cámaras o luces, hasta gestores de cualquier funcionalidad sea visual u organizativa. Cada personaje tiene referencias a todos los atributos que le son necesarios. Estos pueden ir desde la maya y textura que utiliza, pasando por los detectores de colisiones a la propia inteligencia artificial. Sin olvidarse lo más básico como la posición, escala, valores de velocidad, etc. Todos los atributos se cargan para su muestreo por pantalla. Además de ejecutar las distintas funcionalidades que pueden tener los personajes. Esto es porque tiene una serie de métodos básicos que todo personaje puede tener y son invocados en cada ciclo de ejecución. Por lo tanto, es la propia clase *Character* la que se encarga de hacer aparecer cada personaje en escena, pero es cada personaje el que tiene que especificar en su código como se hace esa aparición. Algo parecido pasa con el *update* (actualización) que sería el encargado de modificar valores o acciones a medida que avanza el juego.

4.1.3.2 Factorías

El motor que hace de simulador de un grafo de escena, utiliza una serie de factorías y almacenes por decirlo de alguna forma que se encargan de la gestión de objetos durante la ejecución. Esto es, cuando objeto no existe y es necesario, se crea en la factoría para dárselo a la escena o guardarlo en el almacén. De esta forma cuando el motor necesita de un objeto, este es extraído del almacén sin necesidad de crear uno nuevo. Solo cuando no existe ese tipo de objeto en el almacén, entonces se vuelve a llamar a la factoría para que cree uno nuevo. Por su parte los objetos que dejan de ser utilizados son enviados a al almacén de nuevo y queda en estado de espera. Esto convierte al denominado almacén o piscina <<*pool*>> en un tipo de factoría intermedia que no destruye ni crea, pero agiliza las cosas. Gracias a este sistema se puede ahorrar en memoria dinámica, puesto que se evitan un exceso de creaciones y destrucciones. Evitando a su vez los posibles errores derivados de estos [3]. En el videojuego las factorías tienen las instrucciones de cómo se crea cada personaje genérico, para que luego este sea moldeado a los valores especificados más tarde en el analizador.

4.1.3.3 Analizadores

Los analizadores de UGK, son los que se utilizan luego para crear los analizadores del videojuego. A su vez este se basa en el código de los analizadores *HTMLlite* para la lectura de los archivos. Permitiendo así convertir código en un lenguaje especifico (como puede ser el HTML que se utiliza para la configuración), en contenido que pertenezca al videojuego.

Esto es como el análisis de un compilador, en el cual se detectan las palabras clave como en un analizador léxico, el orden en el que se encuentran como en un analizador sintáctico y si su contenido es aceptable a los patrones específicos como hace un analizador semántico. Todos estos valores especificados por el programador a la hora de hacer los analizadores específicos del videojuego.

4.1.3.4 Ventanas e interfaces

El motor de UGK tiene la capacidad de crear ventanas. Esta capacidad le viene dado gracias a que utiliza como soporte de creación de ventanas las API WIN32 y GLUT. Pero cada evento de entrada o salida dentro de estas, se tiene que especificar en el código del videojuego. Puesto que UGK no tiene un sistema que trate esto de forma general. Por lo tanto, cada videojuego que utiliza el motor puede tener un funcionamiento distinto. Todo depende de cómo se haya programado la interacción usuario interfaz. Por ejemplo, en este proyecto se permiten varios idiomas de interfaz, almacenados en los archivos HTML que se han comentado.

4.1.4 Creación o modificación de contenido de autor

Como se ha comentado anteriormente. El contenido de autor es todo ese material de diseño propio o externo, que es percibido por los sentidos del jugador y que es lo que realmente va a apreciar cuando interactúe con el videojuego. Estos son por ejemplo texturas, modelos 3d, sonidos, etc.

4.1.4.1 Imágenes

Se engloban en este apartado todo tipo de imagen que se vaya a utilizar en la escena ya sea una foto o una textura. Puesto que se ha utilizado el mismo programa para todas ellas.

El programa utilizado para la creación y modificación de las imágenes durante el desarrollo del proyecto ha sido el GIMP. Esto es debido en primer lugar a que es un programa gratuito y no se necesita de licencia, por lo que se ahorra ese dinero. Además, es una aplicación con la que ya se había trabajado con anterioridad. Por lo que no es necesario aprender su funcionamiento básico, ahorrando ese tiempo. Por ultimo permite guardar las imágenes sin información del canal de color, lo cual es necesario para poder cargar esas imágenes en el juego.

4.1.4.2 Mallas

Para el procesamiento de los modelos 3d y sus mayas se utilizado el programa conocido como *blender*. Esto es así por dos motivos, los cuales comparte con el programa de tratamiento de imagen seleccionado, conocimiento gratuito y tener una licencia libre. La otra razón seria que permite exportar los modelos al formato que procesa el motor, siendo este el punto 3DS.

4.1.4.3 Sonido

Para el sonido se encontraría un poco con la misma situación. Debido a su licencia gratuita y que ya se ha manejado con anterioridad, se utilizaría para editar archivos de sonido el programa *Audacity*. Además de ser un programa bastante completo y que cumple a la perfección con lo que sería necesario para el proyecto, aunque solo este limitado a unos pocos formatos.

4.2 Arquitectura del software

Todo programa consta de una estructura que le sirve no solo para organizarse sino también para mejorar su funcionamiento interno. Aquí se analizarán las distintas capas de las que se encuentran en el proyecto, cómo funcionan las herencias entre los distintos archivos y que métodos son recurrentes o necesarios.

4.2.1 Capas dentro del videojuego

El proyecto no consta de capas como se han enseñado en la carrera. Es decir, capas que solo se comunican con alguna otra y su funcionamiento podría estar restringido a una única tarea. Todo esto en una estructura de pila o más piramidal, aunque existan otras. En contrapartida el proyecto lo que hace, sería juntar en algún grupo todo un tipo de funcionalidades, que sí que tendrían una función en concreto. Pero que pueden hacer más funciones internas dependiendo del archivo o lo que se quisiera que hiciera. Ejemplo de ello sería el archivo que además de leer mensajes también realiza otras funciones internas, mientras los devuelve. También podrían considerarse todo aquel archivo que se encuentra en un grupo y que realiza alguna tarea que se podría considerar de otro. Aunque no es lo normal y cada grupo o capa se encarga de una serie de funciones, que normalmente acaban interactuando con todas las otras capas.

Las capas se han diferenciado en grandes grupos. Dependiendo de su funcionalidad, así como de lo que aporta cada una en unos cinco grupos que se verán a continuación. Algunos de los cuales no son diseños propios del videojuego en sí.

Sino que también existen ciertas partes que pueden pertenecer al propio motor o a librerías que se utilicen en el proyecto.

Las distintas capas se podrían diferenciar en estos apartados:

- Analizadores o *Parsers*: En esta parte, el juego usa los métodos basados en la API UGK para modificarlos de forma que pueda utilizarlos de una forma más específica para este juego en particular. Se utilizan para cargar todos los elementos necesarios o leer todas las directrices que contiene los archivos de configuración en formato *.html*.
- Controladores o *Controlers*: De diseño propio, son los encargados de manejar algún aspecto particular de los <<*Characters*>>, que tienen un denominador común o una funcionalidad, que necesite de la colaboración u obediencia de los otros objetos. Como a modo de ejemplo el controlador llamado <<*army*>> tiene la obligación de acabar con todas las demás piezas de un mismo color cuando el rey, de estas, ha muerto.
- Sistema de mensajes: los mensajes pueden ser mandados y recibidos por cualquier objeto que se encuentre en el juego. Independientemente de su tipo, forma u objetivo dentro del mismo. Pero se necesita un sistema que sea el que sepa a quien hay que mandarle el mensaje y coordine todos los envíos que se van realizando, este sistema se mantiene apartado en general del resto puesto que no tiene que interactuar más allá de la gestión de mandar y recibir los mensajes. También está basado en UGK.
- Visualizador: Es el encargado en todo momento de la visualización. Está incorporado en el propio UGK y se puede designar con que sistema de visualización o motor se quiere ver incluyendo al propio UGK. Siendo una característica de todo elemento visible la necesidad de tener un método de visualización o dibujo que invoque a los métodos de esta parte.
- Piscina o *Pool*: Todo objeto de la escena puede ser añadido o eliminado en cualquier momento, pero no por ello, hay que reconstruir en su totalidad o eliminar dicho objeto solo cuando es necesario. Por lo tanto, es también importante tener una capa que sea capaz de insertar o extraer cada elemento en el momento que sea necesario para la escena sin necesidad de destruirlo como tal.

Todas las capas en general interactúan de una u otra forma con la escena tal y como se aprecia en la figura 7. Tanto si es a modo de especificación o si es para el control de algún aspecto de esta.

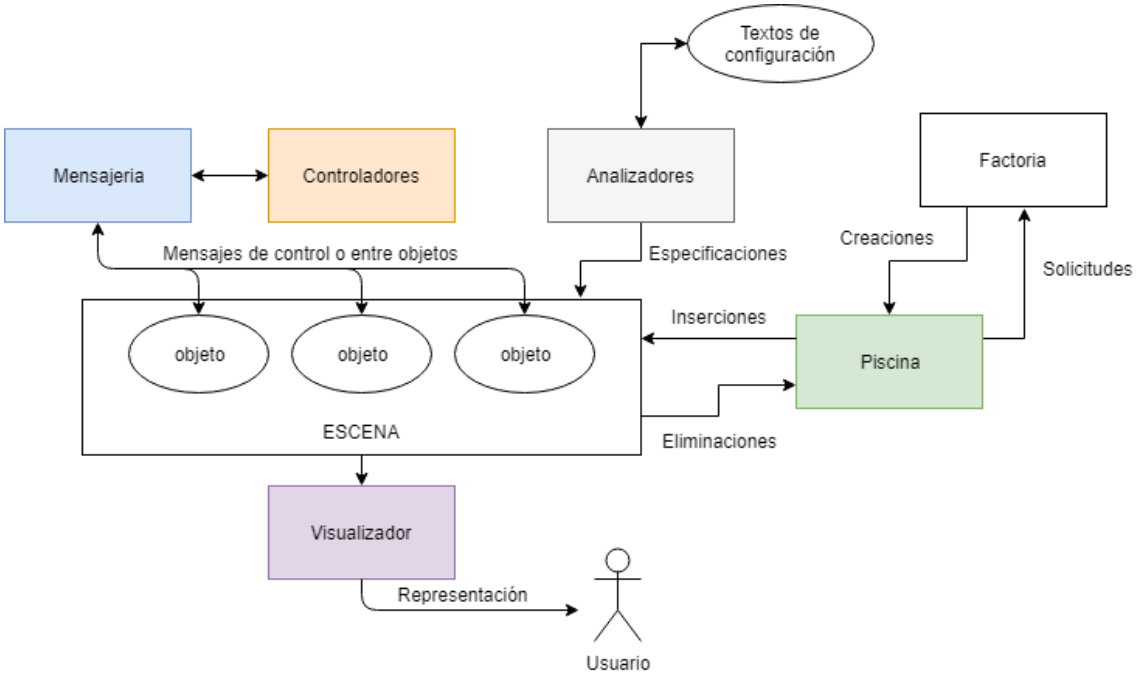

figura 7: Inclusiones y herencias del motor UGK

4.2.2 Herencias e inclusiones

Todo proyecto que se precie tiene archivos que heredan de otros. Esto simplifica la implementación puesto que todo lo heredado del padre no hay que volverlo a escribir. Además, existen las inclusiones a otros archivos, ya sea para utilizar sus métodos, como para utilizar sus estructuras.

El proyecto tiene una jerarquía definida, tanto en herencias como en inclusiones. Donde todo archivo se apoya en otros para realizar así su tarea. Esta estructura no se encuentra solo en el código implementado para este proyecto, sino que también se encuentra en el propio motor. Teniendo en cuanta eso, UGK tiende a utilizar o al menos focalizarse en convertirlo todo en un <<*Character*>>, es por esto que muchas partes acaban heredando de esta clase. Puesto que esta todo enfocado en la creación de un grafo de escena hecho de estos. Por lo que es normal que todo el sistema penda de esta clase.

Dentro de UGK existen una gran variedad de inclusiones. Todas ellas formando un entramado, como se puede apreciar en la figura 8, que permita realizar todas las funciones como mostrar por pantalla, representar la interfaz, colocar cámaras, materializar personajes, reproducir sonidos, etc. Siendo así la base del API UGK.

-

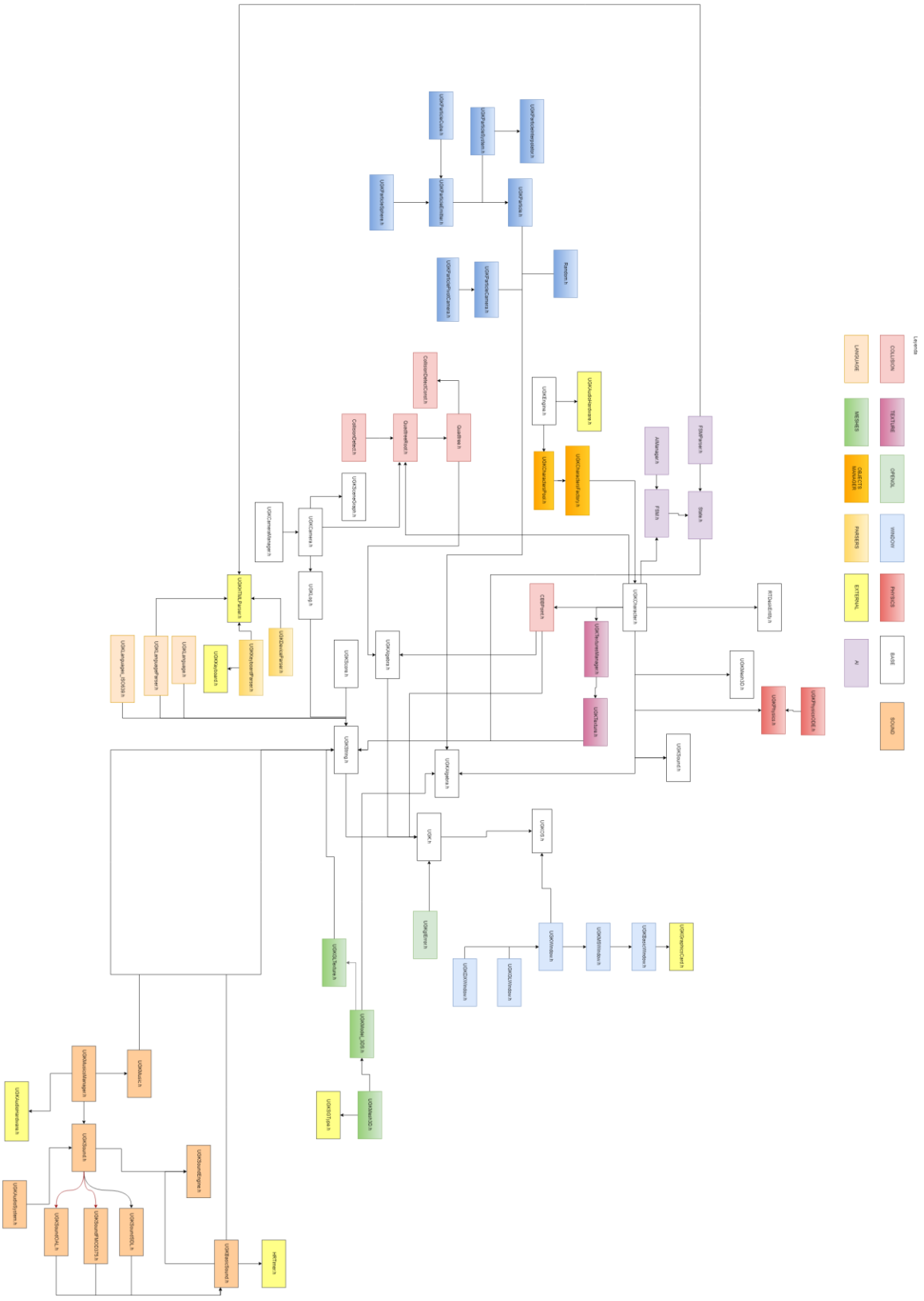

figura 8: Inclusiones y herencias del motor UGK

El proyecto también tiene una serie de herencias relevantes a comentar. Como en el motor UGK, el proyecto también hereda mucho de la clase *CCharacter*. La cual básicamente sirve como base para que todos los objetos que hay por la escena tengan unas características comunes. Estas características comunes van a servir para luego durante la implementación, utilizar funciones comunes, que serán de gran ayuda. Por ejemplo, la utilización del método que le dice cuál es su maya o de la variable que luego indica cuál es su tipo. Por lo tanto, cuando se busca algo en concreto, ya se sabe dónde se tiene que encontrarse. También sirve de ayuda a no tener que escribir más líneas de código de la cuenta. Puesto que, si todo el motor se intenta basar en esa clase, el proyecto también lo tiene que estar.

Existe una clase también importante en el proyecto aparte de *CCharacter*. Esta clase es la de *CPiece*, que básicamente es una herencia de la *CCharacter*, pero que añade métodos y variables que son necesarios a lo largo de todo el proyecto. También hay otras clases que se encargan de ciertos apartados, como la interfaz, pero básicamente también tiene una correlación con la clase *CCharanter*. Como se puede apreciar en la figura 9 la estructura de herencias dentro de las clases del proyecto suele estar bastante relacionada a excepción de ciertos casos. Estos casos suelen ser herencias a otras clases del motor UGK y que por tanto al final también están relacionadas de cierta forma.

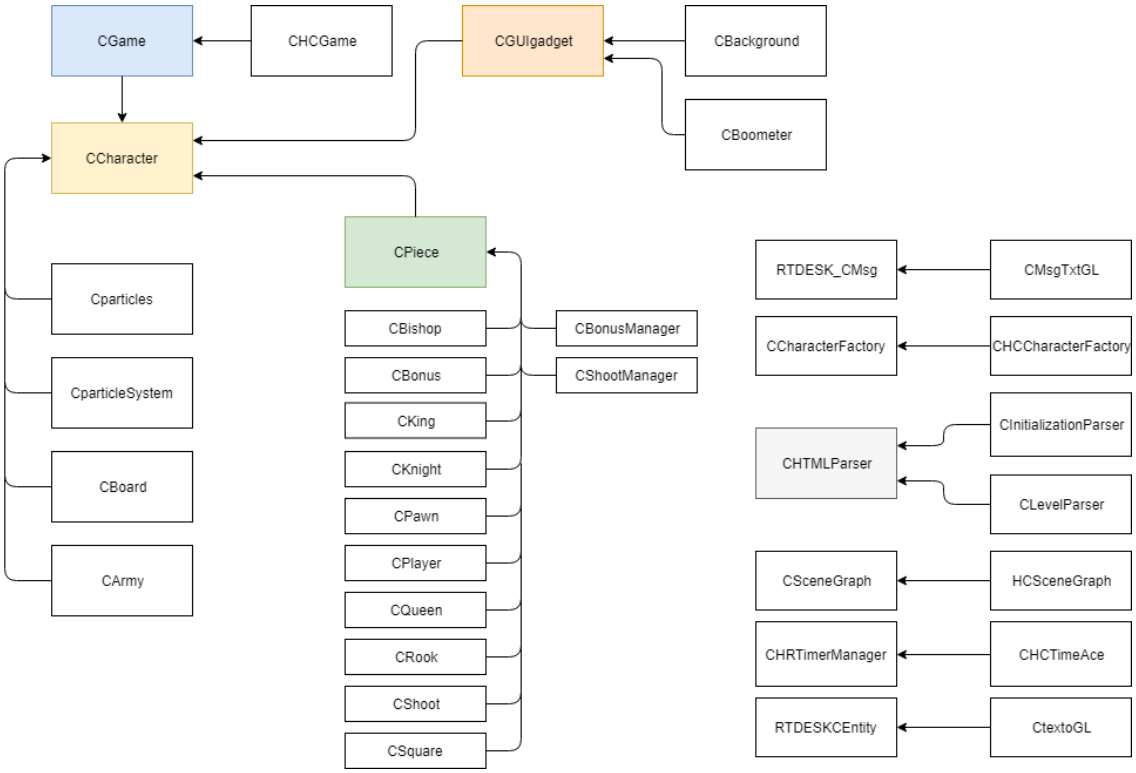

figura 9: Herencias en algunas de las clases del proyecto

4.2.3 Métodos recurrentes o altamente utilizados

La forma en la que se ejecuta el propio videojuego hace necesaria la utilización de algunos métodos que suelen ser llamados con relativa frecuencia. Esto es debido a que, como videojuego, la aplicación tiene un bucle continuo que se realiza constantemente durante la fase de ejecución. Esto también es debido a que la propia representación en pantalla se tiene que actualizar cada cierto tiempo, como la información de todo comportamiento físico en el juego.

Por lo que básicamente todo objeto que se muestra en pantalla va a tener un método asegurado, que se va a repetir una y otra vez mientras este en ejecución. Este método es el de *Render*, que tiene la función de mostrar por pantalla el objeto al que pertenezca el método en particular. Por lo que no todo método *Render* será igual a los demás, aunque se puede dar el caso de que un objeto tenga el mismo método *Render* que otro. En especial si son objetos con una misma representación, como son las piezas. Pero se pueden dar casos donde la representación se tenga que hacer de forma muy distinta, por lo que cada método es un mundo.

Otro método que se puede encontrar a menudo es el método *Update*. Esto es debido a que muchos objetos durante la ejecución tienen que cambiar algún aspecto o variable propios. Por poner un ejemplo, si un objeto se mueve, este tiene que estar cambiando continuamente su variable de posición, lo cual también es necesario para su representación. Por lo tanto, el método *Update* en muchos casos es necesario puesto que, sin él, el propio videojuego no avanzaría. También remarcar que este método se ejecuta en el bucle principal del videojuego como lo hace *Render*.

Un tercer método que se puede encontrar con frecuencia es un método que se hereda de la clase pieza y que se denomina *MovValid*. Este método tiene la función de responder si las coordenadas enviadas a esa pieza en concreto son válidas o no. En caso afirmativo devuelve *True* y se puede mover esa determinada ficha a esa localización. De lo contrario devuelve *False* y no se tendría que mover. Este método se puede utilizar para más cosas, ya que solo tiene la función de comprobar si una localización es viable para esa pieza en particular o no. Cada pieza tiene su propia implementación del método y en algunos casos siempre devolverán que es falso, puesto que esa pieza no se mueve. Ejemplo de esto sería la pieza *Square*, la cual no se tiene que mover de su sitio.

4.3 Implementación

En este apartado lo que se muestra es la forma en la que se implementaron algunas funciones necesarias. Explicando cómo son cada una en particular. Pero sin insertar código en la memoria, puesto que no es relevante para la explicación de estas.

4.3.1 Jugador

El concepto del jugador ya existía de antemano en el proyecto plantilla. Pero en ese proyecto era como cualquier otro personaje de la escena, solo que, en vez de tener pensamiento propio o inteligencia artificial, respondía a los comandos que se le mandan por teclado. En vez de eso, en este proyecto tenía la función de árbitro y cursor. Es decir, que es el encargado de gestionar la partida y de mandar las ordenes a las otras piezas en función de factores como el turno, el color, etc. Cambiando su forma y apariencia además de todas las funciones. De todos los cambios y nuevas formas de gestión se hace una lista como la siguiente:

- Movimiento: Se modifica tanto la función de movimiento en cada dirección (arriba, abajo, izquierda o derecha) como las variables a modificar en el *update*. Siendo así que le permite moverse libremente por la escena. En forma de saltos de longitud respectivo a la escala de las piezas. Esto es también debido a que hay una variable que ajusta esta escala para todas las piezas. Ejemplo de esto es si la escala es uno, es decir no hay escala, el cursor se moverá distancia uno a cada orden de movimiento que reciba.
- Eventos: Se modifican la recepción de eventos que tenía. En la versión actual recibe todos los eventos por el teclado, otra cosa se interpreta por mensaje. En cada evento que recibe, con su identificador correspondiente, este también recibe un marcador. El marcador tiene la función de indicar cuando se está apretando la tecla. De forma que en el manejador de eventos se sepa esto. Aunque en este caso la acción se realice cuando la tecla se ha dejado de apretar, es decir se ha levantado el dedo. Puesto que ocurría que se podían llegar a mandar siete o más evento en una pulsación de tecla. Lo cual generaba un grave error de coordinación. Además, el más importante de todos que es el evento de selección, se quedaba ejecutándose hasta el infinito.
- Mensajes: El sistema de mensajes en el jugador está básicamente para dos cosas. Una es la selección de unidades y de dar ordenas, mientras que la segunda es la de sincronizarse con el juego. La primera y más relevante a la hora de hacer funcionar el juego, se activa cuando recibe el evento de selección (accionado por la tecla *Enter* del teclado). En ese momento realiza la comprobación de si existe alguna selección actualmente o no. Esto lo sabe porque lleva un registro de las selecciones. En caso negativo busca en el grafo de escena la pieza que se encuentra en su posición y le manda un mensaje indicando que se tiene que quedar seleccionada. Por lo contrario, si ya hay una selección hecha, buscara si hay una pieza en su posición en el grafo de escena y si es del color contrario al de la ficha seleccionada.

En caso de no haber nada en su posición o de que lo anterior sea correcto, mandara un mensaje a la ficha seleccionada para que procese si el movimiento es válido para ella. En caso de que la pieza le conteste con un mensaje de aprobación el jugador podrá cambiar el turno al otro color de las fichas. En el caso de que reciba un mensaje negación por parte de la ficha, realizará un cambio interno en su registro de selecciones y ninguna ficha estará seleccionada. Otro punto, es que, si el jugador vuelve a seleccionar la ficha seleccionada, esta se deselecciona, de la misma forma que cuando se falla al indicar un objetivo. Por último, el sistema de mensajes de actualización servirá para que el propio jugador sepa cuando ha terminado la partida o lo que pueda decir cada ejército, el juego actual, etc.

4.3.2 Piezas

Las piezas son una parte importante del proyecto, puesto que es la sección más visible de este. Son las piezas las que se desplazan por la escena y las que tienen como objetivo derrotar a las otras. Aunque estas sean siempre manejadas por el jugador. Es por eso que se tiene que tener en cuenta que toda aportación a estas será relevante en cómo se desarrolla una partida en el videojuego. Estos objetos *Character* tienen cierta similitud respecto de las naves en el proyecto plantilla. Pero se modifica casi todo el comportamiento de las mismas incluyendo sus métodos más relevantes. Los cambios se pueden ver como:

- Mensajería: Si se tiene en cuenta cómo se comunicaba el jugador con las piezas, estas no son muy diferentes. Se utiliza un sistema muy parecido, solo que estas tienen que comunicarse con dos entes. Uno es el jugador, el cual tiene que ser contestado si manda una orden de posicionamiento. Esto se hace mediante la comprobación de las coordenadas enviadas. De modo que si es correcto se le envía el mensaje de confirmación, en caso contrario uno de negación o de deseleccionar, de forma que ambos se deseleccionan y por tanto se comienza desde el principio. El otro ente es su ejército el cual tiene las funciones de administrador y también de juez y árbitro. Es decir que, si reciben un mensaje de muerte por parte de este, la pieza tiene que obedecer y morirse. Por su parte el rey también tiene que confirmarle al ejército que ha muerto.
- Comprobación: las piezas tienen dos formas de comprobación. Una es la de consultar su estado en todo momento e ir actualizándolo en función de las directrices que recibe, ya sea por mensaje o por factores internos modificados en un método. El otro método de comprobación es el de saber si un movimiento es válido. Este método es individual de cada pieza y tiene que implementarse en cada código respectivamente. En este método las piezas tienen que tener en cuenta tres cosas: la escala, su característica de movimiento particular y las otras piezas. Lo primero es más fácil porque ya existe la variable que registra a que escala se encuentran todas las piezas y cuantas unidades tienen que moverse. Lo segundo es más complicado, puesto que cada una tiene una peculiaridad única.

Esto es por ejemplo que los peones cuando salen puedan moverse dos casillas adelante o que ataquen de lado o como los caballos que saltan sobre todas las piezas o como los alfiles que se mueven en diagonal siempre. La tercera es comprobada por todas a excepción del caballo que solo se preocupa de donde aterriza. Pero que para ello se necesita ver todo el grafo de escena para saber si existe una pieza que se encuentre en la trayectoria de la pieza que se quiere mover. Si todo es correcto entonces el método devolverá que es así.

- Movimiento: aquí también tiene mucha influencia el método *update*, puesto que es el encargado de realizar la ejecución de movimiento cada vez que es ejecutado. Esto se realiza solo cuando se encuentra en fase de movimiento. Siendo que además la fase de movimiento tiene sub fases internas. Estas fases sirven para monitorizar cómo se desarrolla el desplazamiento de cada pieza. Por lo que cuando recibe el mensaje de movimiento afirmativo pasa a la fase de movimiento uno. Estas fases son tres puesto que el movimiento se realiza en tres partes. Primero se levanta la ficha hasta cierta altura. Luego se desplaza hasta la posición designada. Finalmente aterriza en el tablero. Este sistema es para que las piezas no tropiecen con las otras a menos que sea la que se tiene que comer. Puesto que de lo contrario podrían cocar con más de una.
- Colisiones: Este sistema utiliza en gran parte de UGK y del sistema de colisiones que usa. Esto es debido a que aun habiendo un método de colisión en cada pieza. Este método solo tiene la función de gestionar lo que pasa cuando colisiona. Detectar la colisión es parte del motor. De todas formas, es muy útil puesto que comunica con qué objeto se ha colisionado. Esto permite saber si se encuentra en fase de movimiento (por si es una pieza que va a comerse a otra). En caso de que lo anterior sea correcto entonces la ficha que recibe la colisión tiene el deber de destruirse. En caso de que sea el rey tiene que mandar un mensaje al ejercito de que ha muerto.

4.3.3 Gestores

Los gestores, aunque no muchos, tienen una serie de funciones muy importantes en el juego. Algunos gestores del videojuego plantilla, como el gestor de disparos, no son utilizados. Pero no son eliminados puesto que ha futuro podrían ser utilizados para expandir el proyecto. Sin embargo, los que son utilizados, véase porque son modificados o creados desde cero, administran partes importantes del funcionamiento de todos los otros objetos de la escena. Es el caso por ejemplo del ya comentado ejército. El cual tiene la función de comunicar las piezas, lanzar los métodos que sean pertinentes en cada situación, como una actualización porque el ejército ha cambiado algo o eliminar piezas si esto fuera necesario. Por ejemplo, cuando muere el rey tiene que recorrer todos los miembros de su ejército en el grafo de escena e ir eliminándolos hasta que no quede ninguno. También está el gestor de mensajes, aunque vasado en el sistema de mensajes de UGK, tiene las especificaciones de cómo tiene que ser cada mensaje y como se tiene que mandar. Además, está el gestor del teclado, que es el encargado de leer las teclas y mandar el evento correspondiente a cada objeto en particular. Finalmente, también se puede encontrar el gestor del juego, que es el que lo crea y gestiona el bucle principal del juego.

4.3.4 Analizadores

Los analizadores han sido modificados respecto de los que existían en el videojuego plantilla. Estos nuevos analizadores tienen las funciones anteriores más alguna añadida o modificada para la ocasión. Estas se podrían describir en los siguientes puntos:

- Color: teniendo en cuenta que este apartado no existía previamente, se ha añadido a los nuevos parámetros posible. De esta forma si se inserta variables a una determinada pieza, se tiene en cuanta cuál es su color. Puesto que un ejército o color, para ser exactos, no tiene que ser igual al otro en parámetros. Esto sirve también cuando se crea un ejército donde se especificará cual es el color y el luego él se encargará de dárselo a sus piezas.
- Ejército o *Army*: aunque ya se ha hablado mucho de este gestor en respecto a varias de sus funciones. Otra cosa muy importante es cuando se realizan sus especificaciones, una de estas es la distribución de las piezas por la escena. Puesto en forma de líneas una encima de la otra, cuando se lee de los archivos HTML, estas líneas están con un código de números que especifican que pieza se encuentra en cada parte. Ejemplo de esto sería que si se lee una línea tal que así <<111121111>>, se estaría indicando que hay una línea horizontal en la escena de ocho peones y una torre en medio de estos. Insertando en la escena las respectivas piezas que valla a medida que las va leyendo. De esta forma el que crea el archivo HTML lo tiene más fácil para describir la disposición en el tablero. Esto se verá mejor en el apéndice.
- Tablero: De forma casi idéntica a como el ejército distribuye sus piezas por la escena. Existe también el tablero que tiene como función la de distribuir las casillas de forma en la que se lo indique cada línea. Donde después se registrarán cada línea a función de cómo se ha parametrizado. Esto es útil puesto que se pueden crear tableros que no sean cuadrados exactos.
- Casos individuales: aunque existan los otros sistemas para introducir objetos en la escena, como son mediante su gestor. También existe la posibilidad de inserción en escena de objetos individuales que no pertenezcan a nada ni a nadie y que puedan ser meramente decorativos o bien que sean piezas individuales. Todos ellos también tienen su sistema para ser parametrizados e introducidos.
- Selector de nivel: Esta aportación nueva es sencillamente un parámetro en la configuración, que permite seleccionar cual es el nivel a ejecutar entre todos los que se tienen a nuestra disposición. Si no existiera ese nivel entre los archivos HTML que los definen. Entonces sencillamente no arrancaría el juego.
- Idioma: algo ya existente en el proyecto anterior se ha retocado para cargar algunos mensajes que antes no hacía y que el primer idioma en aparecer sea el español.
- Reglas del juego y movimientos específicos de las fichas: Estas modificaciones, aunque necesarias para mejorar la experiencia de juego, no han sido implementadas en esta versión. Por lo que se dejaran para futuras implementaciones.

4.4 Desarrollo

En este apartado se recorre como ha sido el desarrollo del proyecto a lo largo del tiempo. Comentando si se siguió el plan de desarrollo especificado en el capítulo 3. Comentando los problemas que ha habido o los cambios realizados.

4.4.1 Primera implementación

Aunque se comenta en el capítulo 3, que la solución a escoger es la de hacer una refactorización a un videojuego plantilla, con los elementos para hacer este proyecto. La verdad es que no fue la primera solución escogida. La primera vez se consideró que se tenía que realizar una actualización del juego original de *HyperChess*. Para más tarde añadirle las funcionalidades de UGK. Como se puede apreciar en la figura número 10, la cual es un diagrama de Gantt reducido de cómo fue el desarrollo, existió una serie de fases, las cuales fueron: Análisis de la API UGK, Analizar aportaciones OSG, Instalación de aplicaciones, Instalación y Actualización de librerías, Modificado de los archivos desactualizados, Inserción de nuevas funciones y eliminación de errores. Podría ponerse más fases, pero estas nunca fueron ni si quiera comenzadas, por lo que se considera irrelevantes.

El problema empezó con que entender el funcionamiento tanto de UGK, como de la versión actual de OSG, se volvieron más costosas de lo esperado. Teniendo en cuenta que UGK está en constante cambio también. Aunque instalar las aplicaciones necesarias para el desarrollo fuera lo único que se completó correctamente. El resto de factores no salió como estaba planeado, ni cerca. La aplicación original tenía todas las librerías desactualizadas y las referencias a estas también estaban mal. Se tuvo que descargar todo, e ir poco a poco añadiendo. Esta tarea no se llegaría a completar por descartar esta implementación y porque se necesitaban muchas librerías distintas. Además de que algunos de las utilizadas cambiaban radicalmente en su uso de cómo se gastaban en el juego original a como se gastaban en la actualidad. Desde un principio se empezó con el tratamiento de errores, en paralelo al resto del desarrollo, puesto que los problemas aparecieron desde el mismo instante en el que se empezó con el proyecto. No se tendrían en cuenta si no fueran ajenos a los problemas que pudiera se pudieran crear por parte de las propias modificaciones realizadas. Pero cuando se empezó con el desarrollo el proyecto del juego original ya estaba desactualizado, en muchos aspectos, y tenía alrededor de unos mil quinientos errores. Todos ellos de distinta índole. Por lo que se corregían errores a la par que se intentaba insertar liberaras, modificar archivos desactualizados o insertar las funciones nuevas. Todos estos puntos se dejaron porque la base del propio juego original ya no era válida y la única solución era empezar desde cero. Puesto que muchos métodos que se gastaron en su momento, en la actualidad o no existían como tal o no funcionaban como antes. Además de errores que nunca se llegó a entender porque eran. Al final de todo el proceso cuando se descartó esta solución aun existían alrededor de quinientos errores.

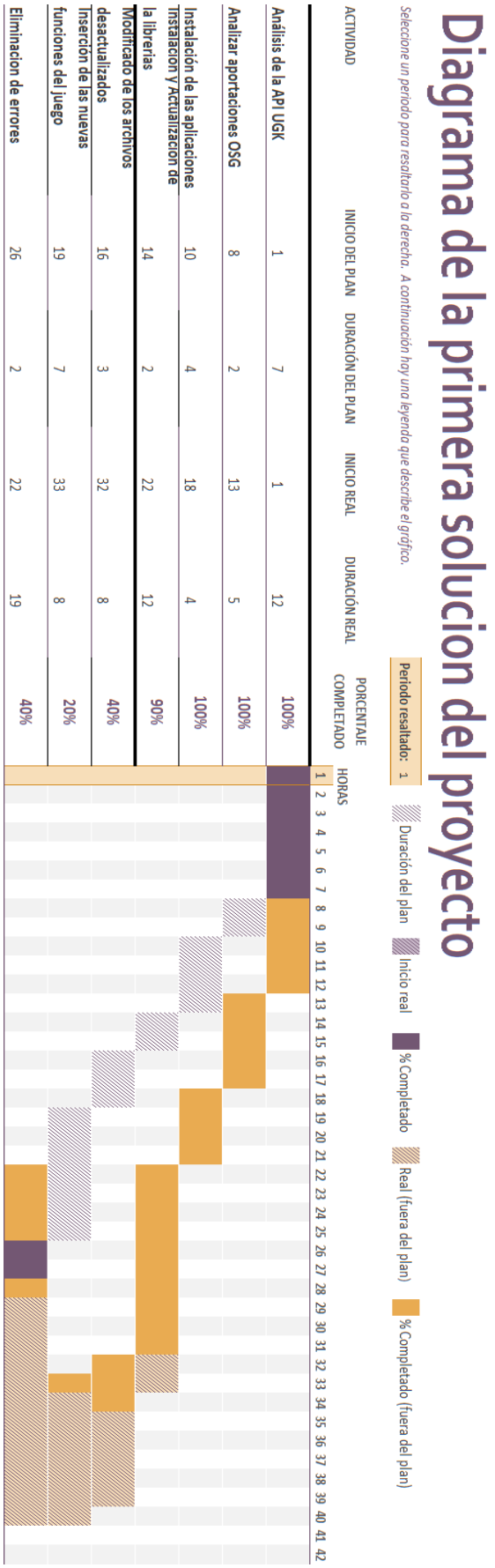

figura 10: Diagrama de Gantt de la primera solución fallida

4.4.2 Implantación Definitiva

La segunda implementación y definitiva, es la solución que si se especifica en el capítulo 3. Por lo que no existe motivo por el que volver a explicar todos los apartados de esta. Sin embargo, se realizará un análisis sobre como fue el desarrollo respecto de lo planeado en el capítulo 3 y cuáles fueron las diferencias. También se podrá apreciar con el diagrama de Gantt de la figura 11 que representa como fue el desarrollo. Tener en cuenta que ese diagrama no va en horas como los anteriores sino del doble de horas respecto de la figura 6.

En primer lugar, como se había pasado por un desarrollo anterior, las fases preliminares de preparación se desarrollaron en poco o nada de tiempo. Estas fases son las de analizar UGK y OSG a la par que instalar las aplicaciones necesarias. La de copiado y creación del nuevo proyecto si se tuvieron que realizar y duraron algo más de lo esperado. Sobre todo, crear el proyecto nuevo, puesto que se tubo introducir todas las opciones e insertar todos los archivos y carpetas uno a uno. Algunas librerías nuevas para UGK dieran algún problema que se solucionó con algo de tiempo.

Después vendrían las fases de desarrollo más importantes. Teniendo en que se tuvo que añadir más cosas de lo esperado y las veces que había que repetir la escritura de código para algunos métodos, declaraciones, etc. El proceso de desarrollo se alargó mucho más de lo planeado. Se estima que casi unas tres veces más de los previsto. Además, desde el momento en que comenzó el desarrollo, también se empezó con el control de errores. Esto es debido a que el proyecto plantilla copiado tenía una serie de errores desconocidos. Además, durante el propio desarrollo del juego aparecieron otros errores que se derivaban del propio motor o de la versión que se estaba utilizando.

Las fases de comprobación también tuvieron partes de implementación. Puesto que aparecieron errores no planificados durante el desarrollo. Los cuales tenían que ver más con la forma de funcionar el juego que con errores de compilación o ejecución. Estos problemas serán comentados en mayor profundidad en el capítulo 5 donde se analizan las pruebas realizadas.

El desarrollo en general de esta solución no fue tal y como se había planeado. Costando mucho más tiempo de lo que se planifico. Teniendo en cuenta que los cambios sobre UGK no son definitivos incluso después del desarrollo de este proyecto. Por lo que puede ocurrir alguna modificación de ultima ahora.

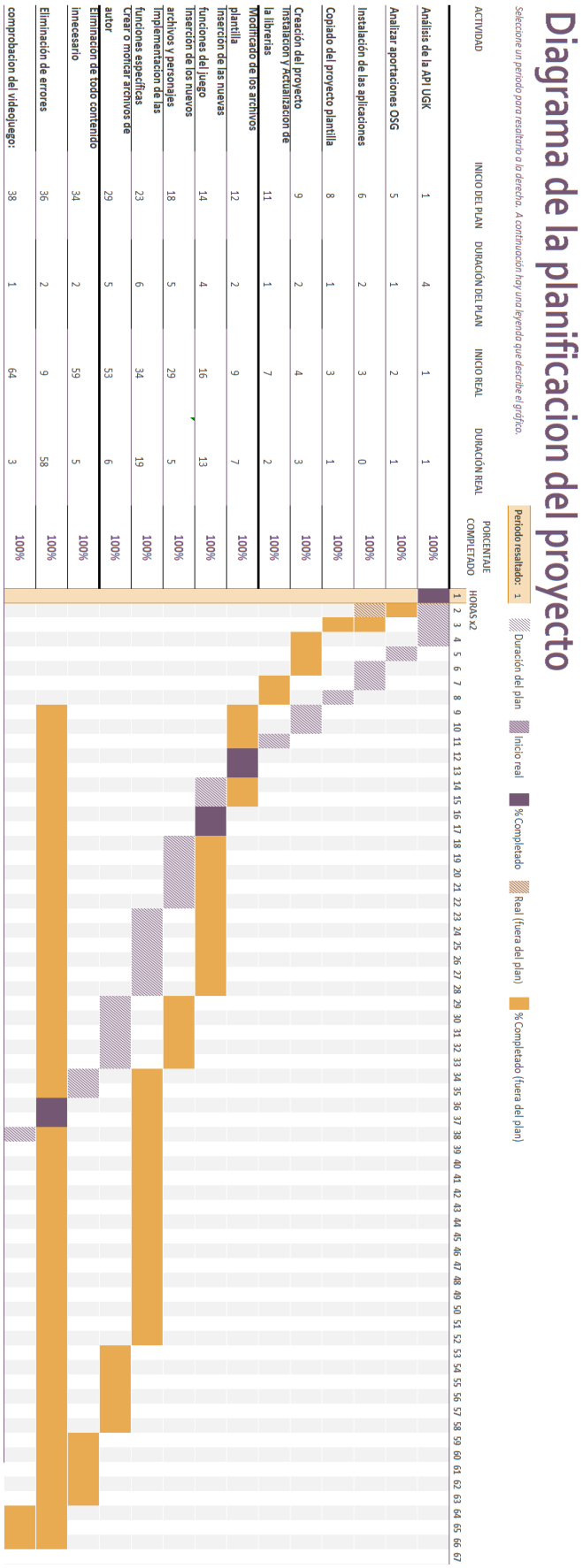

figura 11: Diagrama de Gantt de la solución definitiva

5 Resultados

Las pruebas que se realizan sobre el proyecto, tienen la intención de averiguar si se completaron los objetivos y requisitos de forma correcta. El objetivo de este capítulo es documentar todo lo que ha salido bien o mal en las pruebas. Es decir, se explicará el contenido de la prueba realizada para comprobar algún objetivo y luego se añadirá una parte, en la que se discute si ha sido fructífera o si se ha tenido que realizar alguna modificación extra.

5.1 Analizador HTML

Para comprobar si el juego es capaz de cargar correctamente todo lo parametrizado en los archivos HTML, lo primero que se realiza es una carga del videojuego normal. Esto es vital puesto que, si no carga, es que alguna línea escrita en los archivos HTML está mal y por lo tanto no carga. Después se comprueba que lo que estaba por defecto se ha cargado. Finalmente, después de todo se modifica los archivos HTML para apreciar si han realizado los cambios como se esperaba.

Resultados

La prueba no ha tenido grandes incidentes como tal. En primer lugar, no cargaba como era debido, pero esto era debido a que una línea del archivo de configuración estaba mal escrita. Se solucionó rápidamente puesto que es algo del archivo HTML y no del propio analizador. El resto de la prueba fue realizado correctamente y por lo tanto toda modificación realizada en el archivo también era representada en pantalla.

5.2 Carga de contenido de autor

Se tiene que ser capaz de insertar los elementos deseados por el usuario en la escena y estos ser percibidos correctamente. Por lo tanto, todo modelo, textura o sonido se tiene que reproducir en el videojuego de forma fidedigna a como es en la realidad. Para comprobar esto lo más sensato es introducir elementos nuevos en la escena y luego asígnales este contenido. Para más tarde modificarlos por otros y comprobar si son correctos los cambios.

Resultados

La carga de los contenidos no ha tenido como tal ningún problema. Pero existen problemas en ciertos puntos a la hora de la visualización. Esto es que todo elemento se ha cargado en la escena, pero algunos no se visualizan como es debido. Esto es probablemente debido a algún problema con el motor.

Dando como resultado que las texturas plasmadas en los modelos 3D no se visualizan como tal. El resto de elementos se cargan y perciben como es debido.

5.3 Movimiento y actualización

Todos los objetos interactivos del videojuego, es decir las piezas, tienen que ser capaces de moverse por la pantalla y el tablero. Además, ser capaces de actualizar todos sus parámetros mientras se desarrolla la acción en la partida. Para comprobar esto se necesitan tres pruebas. La primera es ver si la función de selección creada para el jugador, es funcional o no. Para esto se valdrá de puntos de ruptura en el código que se activen durante la ejecución del juego, en las líneas de código específicas que se tienen que ejecutar. Después realizar algunas directrices de movimiento y ver si de verdad realizan correctamente lo que se les específica y es representativo de esa pieza. Tal y como se puede apreciar sobre el funcionamiento de un alfil en la figura número 12. Finalmente, realizar otros puntos de interrupción en el código, pero esta vez comprobar si se ha modificado los valores internos correctamente.

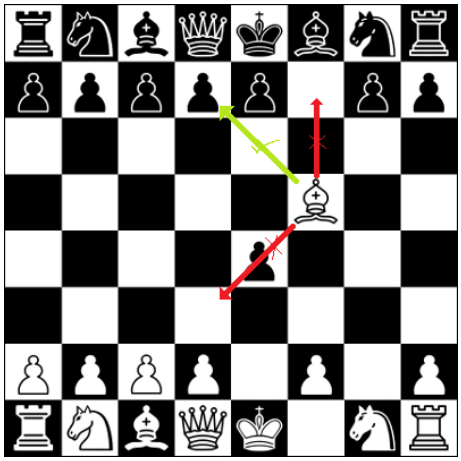

figura 12: Movimiento de un alfil. En diagonal y sin saltar por encima.

Resultados

El resultado del experimento ha sido el correcto después de algún ajuste de última hora. Se ha conseguido que los personajes se seleccionen como es debido, pero se ha tenido que corregir un problema que generaba más mensajes de selección de los que era necesario. Solucionado esto el segundo error corregido ha sido entorno al movimiento de las fichas. Pues estas al desplazarse podían pasarse de la posición objetivo, debido a la velocidad que podrían llevar. Para corregir esto, se ha puesto una variable de error, en la cual, si el objetivo está entre la medida exacta más o menos el error, frena y se recoloca en la posición objetivo. El resto de objetivos se ha conseguido sin ningún problema.

5.4 Otros Objetivos

Aquí se comentan los otros objetivos que no tiene una prueba específica para comprobar si se ha conseguido completar con éxito.

5.4.1 Carga y eliminación del grafo de escena

Si se tiene en cuenta que, durante la ejecución del programa, es necesario en varias ocasiones recorrer el grafo de escena para encontrar los objetos, sin encontrar ningún problema a esto. Se puede afirmar que la carga de objetos en el grafo de escena es correcta y que funciona como se deseaba en un principio. Además, recorrer este grafo no es demasiado costoso y resulta un método rápido de inserción.

5.4.2 Utilización de UGK y actualización de la aplicación

La aplicación en todo momento utiliza el API UGK y por lo tanto este objetivo se ha conseguido completar con éxito. Es por esto que la modernización o actualización de la aplicación original de *HyperChess* se ha conseguido. Puesto que la nueva aplicación no tiene errores de librerías anticuadas o problemas con las nuevas. También se desenvuelve bien con los nuevos cambios en UGK, por lo tanto, se considera que tiene la refactorización más actualizada.

5.4.3 Interfaz

Se ha conseguido tener una interfaz mínima y funcional basada en la interfaz que tenía el videojuego plantilla. Por lo tanto, no es un gran avance en este objetivo. Sin embargo, como el único interés que se tenía era que fuera funcional, se puede considerar completado. Aunque se hayan podido quedar varias cosas por añadir. Al menos se ha conseguido que el idioma español se cargue correctamente, algo que no hacía antes.

5.4.4 Ampliación

Anuqué no un objetivo muy relevante en la implementación. Se ha tenido en cuenta que se puedan realizar futuros trabajos basados en este proyecto y que esta aplicación se pueda utilizar para la enseñanza. Esto se ha conseguido manteniendo un esquema que ya se encontraba en la aplicación plantilla y que por lo tanto hace similares en forma a ambos proyectos. Tampoco se ha creado ningún método que pueda ser demasiado problemático a la hora de las ampliaciones.

6 Conclusiones

El trabajo realizado ha sido una escala constante en el trabajo del desarrollador. En el cual se ha tenido que cambiar de plan original a mitad desarrollo. Esto se ha debido en gran parte a la falta de experiencia del alumno entorno a proyectos con una gran cantidad de problemas. Por otra parte, también ha servido para aprender a darle un nuevo giro al problema si este se vuelve demasiado inabarcable y de esa forma no detenerse en ningún momento. Cuanto más rápido se percate uno de los problemas y de las posibles soluciones más rápido podrá reaccionar. Por otra parte, el cambio total de una estructura ya existente, como la del videojuego plantilla, requiere de un cierto tiempo de reflexión y aprendizaje. Esto ayuda a tener más agilidad a la hora de entender otras estructuras en el futuro.

Aunque los objetivos se hayan completado parcialmente todos de alguna forma. No deja de existir la sensación de que se tendría que haber realizado más. Puesto que no todo está perfecto y seguramente se tenga que dejar más trabajo a las futuras aportaciones a este videojuego. Aunque toda la base para esas futuras aportaciones está realizada y funciona. El proyecto en sí es una forma de aprendizaje muy valiosa que no considero que se pueda dar por acabado en mucho tiempo. Puesto que siempre se le podrá añadir más cosas. Es un proyecto que enseña otras formas de resolver un problema o nuevas estructuras, como el ya mencionado grafo de escena. Además, resulta una experiencia más visual e intuitiva que, si solo fuera código, puesto que permite experimentar en vivo con conceptos que todos entienden. Por lo tanto, es una forma más fácil de entender que es lo que se está realizando y cuál es el objetivo. Ejemplo de todo esto, sería ver cómo se desarrolla el movimiento de una pieza cuando todo el código funciona correctamente, algo visible y fácil de entender.

Por otra parte, también ha servido como base para aprender a trabajar en una entidad o en colaboración con otra gente. Puesto que en muchas ocasiones se tenía que ser consciente de los cambios que podían realizar las otras personas que trabajaban sobre el motor.

6.1 Relación con los estudios cursados

El trabajo en sí mismo ha tenido muchos factores relacionados con todo lo impartido a lo largo de la carrera. En él se han podido desarrollar diversas formas de estructuración de código, tal y como se ha visto en la carrera. Ejemplos de esto son las pilas, grafos, etc. Todas ellas dadas en la asignatura de estructuras y algoritmos. También se ha estado desarrollado una ingeniera del software, sobre todo en las partes que necesitaban más una actualización. A la par que también se tenía una cierta planificación de proyecto, tal y como se enseña en la carrera, en la asignatura de gestión de proyectos. Además, también sirvieron los conocimientos sobre gráficos que se me impartió en la rama de computación en la asignatura de Gráficos por ordenador.

6.2 Trabajos Futuros

Aquí se plasman algunos trabajos que se podrían realizar en el futuro, teniendo en cuenta las ideas que se actuales de lo que puede ser ampliable.

6.2.1 Mejor interacción tablero y bonificación

Hay que tener en cuenta que la estructura del tablero ya venía siendo algo complicada en sus orígenes cuando el proyecto comenzaba. Aunque se haya hecho mucho por mejorar este apartado. El tablero sigue siendo una entidad que no tiene mucho en relación con el grafo y además no tiene ninguna interacción con las piezas. Algo que se tendría que mejorar y que, además tendría que permitir realizar alguna función con el elemento de bonificación o <<*bonus*>> que en estos momentos no se utiliza. Aportando así algo nuevo al juego.

6.2.2 Inteligencia artificial

El juego carece de una inteligencia como tal y es dependiente de que dos jugadores humanos re pongan a jugar simultáneamente. Por lo que se consideraría oportuno que se creara una inteligencia artificial mínima, para que no fuera necesaria la interacción entre dos personas, permitiendo así partidas en solitario. Aunque también no estaría de más que esa inteligencia tuviera distintos niveles de dificultad.

6.2.3 Reglas configurables

Teniendo en cuenta la capacidad que se tiene de leer los archivos HTML, sería una buena aportación que además se pudieran especificar las reglas del juego. Esto viene bien para cuando se cree nuevos personajes o se quiera cambiar las reglas ya existentes. Porque por el momento no es posible modificar las reglas puesto que se encuentran fijas en el código del juego. Seria inventar una metodología para pasarle las reglas a la aplicación y poderlas modificar a gusto del usuario.

6.2.4 Interfaz

La interfaz en estos momentos es mínima. Sería deseable que se pudiera expandir para agregar nuevas capacidades a la ya existente. Esto serian nuevas formas de configuración o pantalla. Además, permitir que durante el juego se permita el control con el ratón y no solo con el teclado.

6.3 Agradecimientos

Quisiera agradecer a la universidad por todos los recursos que ha puesto a mi disposición a lo largo de mi aprendizaje y sobre todo en el material de consulta que he podido utilizar durante este. Además de todas las facilidades que han puesto para que pueda hacer y presentar este trabajo.

Quisiera agradecer también a mi tutor por estar siempre accesible cuando era necesario. Además de aconsejarme y ayudarme, en todo lo relacionado al trabajo que he necesitado y no tener reparos en decirme en lo que me he equivocado.

Finalmente quisiera agradecer a mis padres todo el apoyo y ayuda que me han dado tanto a lo largo de la carrera, como cuando realizaba este proyecto. Gracias a ellos he podido tener más tiempo para mis estudios y dedicarle más esfuerzo a terminar el proyecto, que el que le habría dedicado, si tuviera que depender de un trabajo remunerado.

7 Bibliografía

[1] Juan Camilo Ibarra López, Fernando De la Rosa. Creación Interactiva de Grafos de Escena para Aplicaciones Gráficas 3D. 2008.

[2] Inkyun Lee, Hwanyong Lee, and Nakhoon Baek. Research on Implementation of Graphics Standards Using Other Graphics API's. 2011.

[3] Andrei Alexandrescu. Modern C++ Design Generic Programming and Design Patterns Applied. 2004.

[4] Robert Cecil Martin. Agile Software Development Principles, Patterns, and Practices. 2006.

Apéndice

Aquí se adjuntará una información y código, en el cual se verá más fácilmente cómo funciona la parametrización de los archivos HTML. Además de ir acompañado en algún momento de imágenes que así lo representen.

Explicación archivos de inicialización HTML

Existen dos tipos importantes de archivos HTML en el proyecto. Uno es el de inicialización el cual se describirá aquí. El otro es el de nivel que se mostrara más en adelante.

Lo primero que hace el archivo de inicialización es direccionar el programa hacia los directorios donde se encuentran los distintos elementos del juego.

```
<HTML>
<HEAD>
        <TITLE>HyperChess</TITLE>
        <VERSION>1.0.0</VERSION>
        <TYPE>Initialization</TYPE>
</HEAD>

<BODY>
  <DIR>
    <AI>AI</AI>
    <MUSIC>sounds</MUSIC>
    <SOUNDS>sounds</SOUNDS>
    <TEXTURE>images</TEXTURE>
    <GEOMETRY>models</GEOMETRY>
    <LANGUAGE>languages</LANGUAGE>
    <LEVEL>Levels</LEVEL>
    <LEVEL_STR>0</LEVEL_STR>
    <CONFIG>configuration</CONFIG>
  </DIR>
```

Primero se puede ver el encabezado de todo archivo que describe que juego es, que versión y que archivo es. Esto es importante para que el programa lo coja.

Luego aparece el cuerpo del archivo y con el nuestro primer punto que son los directorios. En los que se puede apreciar que cada apartado tiene su respectiva carpeta como puede ser la de música, texturas, modelos, lenguajes, etc. Aunque existe uno en particular que es el <<LEVEL_STR>>, que es una variable que indica en qué nivel comienza el juego, así el jugador puede elegir la variedad que desea y allá sido descrita en ese nivel en particular.

Después vienen las especificaciones de cada uno de los personajes que vallan a aparecer en el videojuego. Si no se parametriza algo aquí el personaje cogerá su valor por defecto. Aunque a veces ese valor es nulo.

```
<CHARACTER>
    <NAME>BISHOP</NAME>
    <MESHW>1bishop.3ds</MESHW>
    <MESHB>1bishop.3ds</MESHB>
    <TEXTURE2DW>texBishopw1.bmp</TEXTURE2DW>
    <TEXTURE2DB> texBishopw1.bmp</TEXTURE2DB>
    < TEXTURE3DW>white.bmp</TEXTURE3DW>
    <TEXTURE3DW>black.bmp</TEXTURE3DW>
</CHARACTER>
```

Aquí se muestra como se los personajes se encuentran encasillados en su propia denominación. Donde lo primero y más importante es el nombre. Por lo que, si no se pone primero, el resto de la parametrización no funciona, puesto que no sabe a qué personaje se refiere. Más tarde se especifican parámetros más propios de cada personaje y que son comunes a todos los mismos personajes de la escena. Es decir que gracias a esta parametrización todos los <<bishops>> del mismo color, tendrán el mismo repertorio de texturas y modelos. También como se puede apreciar se pueden diferenciar entre color blanco y negro por la letra W y B situados al final de la palabra. Por lo que se puede apreciar en este ejemplo todo <<bishop>> tiene el mismo modelo, pero dependiendo del color tiene una textura u otra.

El tablero también es un personaje como tal:

```
  <CHARACTER>
    <NAME>BOARD</NAME>
    <TEXTURE2DW>pixel_vert.bmp</TEXTURE2DW>
    <TEXTURE2DB>pixel_vert2.bmp</TEXTURE2DB>
    <TEXTURE3DW>pixel_vert3d.bmp</TEXTURE3DW>
    <TEXTURE3DB>pixel_vert3d2.bmp</TEXTURE3DB>
  </CHARACTER>
```

Por lo que como antes se puede apreciar que dependiendo de que cuadrado se use, este tiene una textura u otra. También dependiendo de si la representación es en 2 o 3 dimensiones.

Por ultimo vendrían los parámetros de colisiones. Donde se especificaría que elemento colisionara con otros. Es decir que un personaje solo colisionara con otro que se encuentre en su lista de colisiones de lo contrario pasara de él. Estas colisiones sirven por ejemplo para saber si una pieza se comerá a otra u otras implementaciones que pueden salir del contacto directo entre piezas.

```
  <COLLISION_TABLE>

    <COLLISION>
      <CHARACTER>BISHOP</CHARACTER>
      <COLLIDED>KING</COLLIDED>
      <COLLIDED>KNIGHT</COLLIDED>
      <COLLIDED>PAWN</COLLIDED>
      <COLLIDED>QUEEN</COLLIDED>
      <COLLIDED>ROOK</COLLIDED>
    </COLLISION>
```

Siguiendo con el ejemplo del <<bishop>>. Se puede apreciar que daca personaje se encuentra encasillado en una colisión en particular. Donde otra vez el nombre va primero y luego todos los otros personajes con los que colisiona. Esto se repetirá en todos los personajes de la misma forma, puesto que la tabla de colisiones solo afecta al personaje parametrizado. Ejemplo de esto seria, cuando el <<bishop>> colisione con el <<knight>>, si el segundo no tiene una tabla de colisiones, entonces la colisión solo la detectara el primero, pero no el segundo.

Finalmente se cierra el archivo con las referencias:

```
  </COLLISION_TABLE>

</BODY>
</HTML>
```

Para decir que es el final del archivo, como todo archivo HTML que se precie.

Explicación de archivos de nivel HTML

Aquí se explicarán peculiaridades que presentan los archivos de nivel. En los cuales se parametriza la forma en la que se va a encontrar el videojuego cuando se habrá por parte el usuario. También decir que todo elemento que se modifique en este archivo a diferencia del de inicialización, solo afectara a un personaje en particular. Siendo que si se especifica algún parámetro que se describió en el de inicialización como puede ser el modelo o la textura, este cambio machacara al de la inicialización, pero solo afectara a un personaje especifico.

El encabezado en casi igual al del archivo de inicialización, solo que en este se especificara que es de nivel y que nivel es.

```
<HEAD>
  <TITLE>HyperChess</TITLE>
  <VERSION>1.0.0</VERSION>
  <TYPE>Level</TYPE>
  <LEVEL>1</LEVEL>
</HEAD>
```

Aquí se especificará un personaje en especial que es el juego en sí mismo. Esto es porque se tiene que parametrizar para el arranque de este y de ciertas funciones únicas de este dependiendo del nivel en el que se encuentre.

```
  <CHARACTER>
    <NAME>GAME</NAME>
    <RENDERMODE>3D</RENDERMODE>
    <SIMULATIONMODE>CONTINUOUS</SIMULATIONMODE>
    <BOUNCE>1</BOUNCE>                <! --Bounce Mode: 0->off, 1->active-->
    <TIMERENDER>16.67</TIMERENDER>
    <TIMEUPDATE>16.67</TIMEUPDATE>
  </CHARACTER>
```

Aquí es donde se es donde se especifica qué tipo de renderizado va a tener, que tipo de simulación va a ser entre otros como puede ser el tiempo de cada ciclo de actualizado, etc.

Después se pueden representar personajes por separado. Luego ya se verá cuando van en conjunto. Estos personajes son individuales, aunque tiene las mismas características que los otros a menos que las parametrizaciones los cambien. Esto pude realizarse para hacer pruebas o un tipo de juego en el cual haya pocas fichas distintas entre ellas.

```
<CHARACTER>
    <NAME>BISHOP</NAME>
    <COLOUR>WHITE</COLOUR>
    <MESH>ejemplo.3ds<MESH>
    <POSITION>
      <X>1.0</X>
      <Y>0.0</Y>
      <Z>0.0</Z>
    </POSITION>
</CHARACTER>
```

Aquí a modo de ejemplo se crearía un <<bishop>> individual que tendría sus propias variables. De la misma forma que era importante el nombre en la inicialización, aquí también será importante el color, puesto que si no se especifica la pieza pasaría a default y no interactuaría con nada, sería como de decoración. Además, que se usa también para cargar los valores predefinidos de las piezas de ese color. Finalmente se modifica tanto su malla, que como se ha dicho solo le afecta a él y su posición en la escena. Esta posición es respecto del sistema de representación del videojuego en la escena.

Después vendría otro de los personajes más importantes, este es el ejército o *Army*. En el cual se especifican una gran distribución de otros personajes por la escena a la par que les da una serie directrices generales para todos.

```
  <CHARACTER>
    <NAME>ARMY</NAME>
    <COLOUR>WHITE</COLOUR>
    <NUMLINES>5</NUMLINES>
    <FROM>
      <X>-4.0</X>
      <Y>-4.0</Y>
      <Z> 0.0</Z>
    </FROM>
    <LINE>00000000</LINE>
    <LINE>00000000</LINE>
    <LINE>00000000</LINE>
    <LINE>11111111</LINE>
    <LINE>23365332</LINE>
    <SPEED>
      <X>0.001</X>
      <Y>0.001</Y>
      <Z>0.001</Z>
    </SPEED>
  </CHARACTER>
```

Aquí se puede apreciar como el ejército tiene un color, el cual se lo transmitirá a todas las piezas que pertenecen a este. Su posición respecto del eje de coordenadas en la escena y las líneas de ejército que se van a crear. Además de a qué velocidad se desplazan en la escena dichas fichas.

Las líneas son importantes puesto que cada una representa que piezas se van a poner alineadas en ese momento. Teniendo en cuenta que los ceros son piezas inexistentes. Mientras que los otros números son asignados a piezas determinadas. Siendo de la siguiente forma: uno es el peón, dos es la torre, tres el caballo, cuatro el alfil, cinco la reina y seis el rey. En la siguiente figura se puede apreciar como que daría el resultado de este ejército.

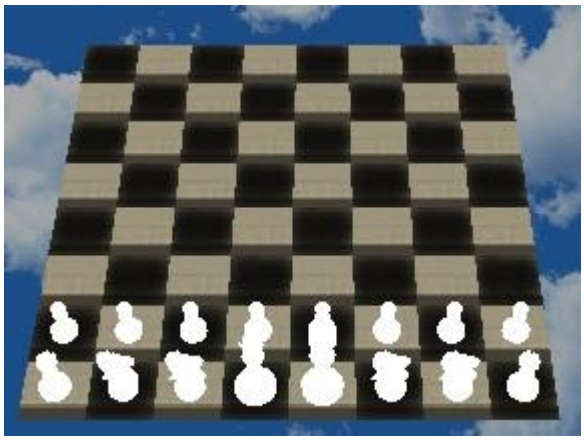

figura: Representación ejército blanco

Finalmente, está la representación del tablero en el videojuego, el cual tiene las posibilidades de ser configurable como lo es el ejército.

```
<CHARACTER>
  <NAME>BOARD</NAME>
  <TYPE>USER</TYPE>
  <POSITION>
    <X>-4.0</X>
    <Y>-6.0</Y>
    <Z> 0.0</Z>
  </POSITION>
  <LINE>21212121</LINE>
  <LINE>12121212</LINE>
  <LINE>21212121</LINE>
  <LINE>12121212</LINE>
  <LINE>21212121</LINE>
  <LINE>12121212</LINE>
  <LINE>21212121</LINE>
  <LINE>12121212</LINE>
</CHARACTER>
```

De la misma forma en la que se representaba el ejército este también tiene líneas que representaran la distribución de los cuadrados sobre la escena. También apreciar que el tablero no necesita color puesto que no es necesario, pero si un parámetro que defina su tipo. Esto es debido a que si no se especifica el tablero pasara a predeterminado que es igual a no visualizarse.

De la misma forma que los números tenían una relevancia en las líneas del ejército, el tablero también tiene su codificación. Siendo los ceros iguales a que no haya cuadrados. Mientras que el resto de números tiene una configuración como la siguiente: uno es cuadrado blanco, dos es cuadrado negro, tres es cuadrado bono blanco y cuatro es cuadrado bono negro. Aunque estos dos últimos que se han mencionado, no tiene más que una representación del color, puesto que los bonos no están habilitados en esta versión del videojuego. En la siguiente figura se puede apreciar cómo se visualiza el tablero en la escena.

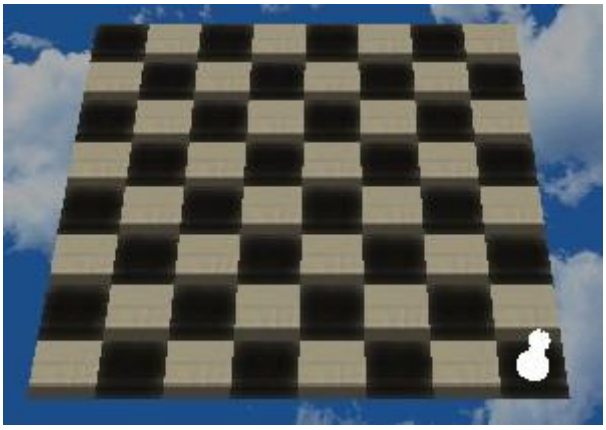

figura: Representación tablero cuadrado

Aunque por mostrar otro ejemplo de tablero y de cómo queda también tendremos este otro.

```
<CHARACTER>
  <NAME>BOARD</NAME>
  <TYPE>USER</TYPE>
  <POSITION>
    <X>-4.0</X>
    <Y>-6.0</Y>
    <Z> 0.0</Z>
  </POSITION>
  <LINE>00000001</LINE>
  <LINE>00000012</LINE>
  <LINE>00000121</LINE>
  <LINE>00001212</LINE>
  <LINE>00012121</LINE>
  <LINE>00121212</LINE>
  <LINE>01212121</LINE>
  <LINE>12121212</LINE>
</CHARACTER>
```

Que como se puede apreciar introduce ceros para quitar cuadrados de la escena. Siendo estos unos cuadrados que no existen y que por lo tanto tampoco interactúan en la escena.

Quedando como resultado la siguiente figura. En la cual se puede apreciar que se ha creado un tablero con una forma triangular recta.

Figura: Representación de un tablero triangular