UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Juego de Estrategia en Tiempo Real con Agentes Inteligentes y Planificación

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Miguel Arturo Martínez Gutiérrez

Tutor: Oscar Sapena Vercher

Curso 2020 - 2021

Resumen

El presente proyecto consiste en la realización de un juego de estrategia en tiempo real que le permite al jugador construir estructuras, recolectar recursos y entrenar unidades para la posterior defensa de su base de operaciones, que será atacada por hordas con regularidad. El jugador no podrá controlar sus unidades durante el periodo de defensa, siendo éstas controladas por un sistema de inteligencia artificial basado en planificación. El desarrollo de este trabajo se plantea sobre el contexto del interés del alumno por el área del desarrollo de videojuegos y la aplicación práctica de técnicas de inteligencia artificial.

El juego tiene como finalidades emplear los conocimientos adquiridos en el área de la inteligencia artificial y aplicarlos a un área operativa como pueden ser los videojuegos; al igual que permitir la expansión de un portafolio personal.

Palabras Clave: Agentes Inteligentes, Planificación, Estrategia en Tiempo Real, RTS, Unity, Agentes Reactivos, Sistema Multiagente, Juego.

Abstract

This work consists in the development of a real time strategy game that lets the player build structures, collect resources and train units that will be used in the subsequent defense of the player's base of command; this base will be attacked by hordes of enemies every certain amount of time. The player will not be able to control his units during the defense phase as this units will be under the control of an artificial intelligence based on planning techniques. The development of this work is based on the context of the student's interest in game development and the practice of artificial intelligence.

The game aims to expand the student's knowledge in the area of AI applied to video games and at the same time expand the student's personal portfolio of developed games.

Keywords: Planning, Real Time Strategy, RTS, Unity, Reactive Agents, Multi-agent Systems, Game.

Agradecimientos

A mis padres Jesús Martínez y Laura Gutiérrez, y a mi pareja Wanda Lavanga por estar conmigo desde el principio de mi carrera, apoyándome en los momentos difíciles y celebrando mis logros.

A mi tutor Oscar Sapena por la paciencia, disponibilidad y ayuda mostrada durante estos arduos meses de trabajo.

Y, por último, a mis amigos más cercanos por estar presentes cuando los necesitaba e incluso cuando no, pero sin quienes estos últimos cuatro años no hubiesen sido lo mismo.

Tabla de contenidos

Tabla de ilustraciones

Capítulo 1

Introducción

Los juegos de estrategia en tiempo real (RTS) disfrutaron de una época dorada entre los años 1995 y 2010; durante este periodo se produjeron joyas del género como Age Of Empires, StarCraft y Command & Conquer; sin embargo, hoy en día este género parece estar en una fase de decadencia debido a la falta de originalidad en las propuestas de sus sucesores. Podemos notar en los ejemplos mencionados como el gameplay loop de todos es muy similar; en estos productos el jugador empieza con un grupo de unidades básicas que solo pueden construir y recolectar recursos. A medida que avanza el juego se van desbloqueando edificios y unidades más fuertes y con capacidades de pelea mayores hasta que el jugador decide enviar sus unidades a batallar en el campamento enemigo, y, dependiendo del tipo y de la cantidad de unidades, el ataque puede resultar exitoso o no. A partir de ese resultado se replantean y ejecutan nuevas estrategias hasta que alguno de los contrincantes pierda todas las unidades y edificios. Consideramos que con pequeñas alteraciones al gameplay loop de este género, junto con la asistencia de sistemas avanzados de inteligencia artificial, se podría generar una sensación novedosa y refrescante al jugar los RTS, como ya se intentó en juegos como Homeworld o There Are Billions; estas alteraciones podrían venir de ideas básicas de otros juegos más populares en estos momentos, como Totally Accurate Battle Simulator (TABS), los distintos juegos de Tower Defense o el ya mencionado There Are Billions.

TABS nos presenta una mecánica alternativa y muy popular, que no tienen los juegos típicos de estrategia. A diferencia de los RTS clásicos, TABS no permite el control de las unidades, lo único que está bajo el control del jugador es la posición, el número y el tipo de unidades durante el inicio de una partida; una vez que el jugador realiza estas acciones, se inicia una simulación de batalla donde él es solamente un espectador que analiza lo que funciona y lo que no de su planteamiento inicial; muy parecido a lo que ocurre con juegos de Tower Defense donde el jugador, una vez posiciona sus torres, no tiene decisión sobre a quién atacan estas. Este juego sin embargo tiene un problema que muchas veces puede resultar frustrante: la inteligencia artificial de las unidades es muy básica, su funcionamiento consiste en atacar a la

unidad más cercana y, en el caso de no tener ninguna en su rango de acción, moverse hacia adelante; lo que ocasiona que muchas veces no elija la mejor unidad para atacar, que no se coordinen varias unidades para asaltar a una que está sola, o que se queden los soldados atorados en alguna irregularidad del mapa. Este sistema funciona suficientemente bien para el planteamiento simple que propone TABS, pero si quisiéramos extrapolarlo a un juego más complejo como los RTS nos vamos a encontrar con muchas carencias que serían contraproducentes para los objetivos que queremos lograr en el campo de los juegos de estrategia.

Este trabajo propone la creación de un juego RTS que mezcle la idea de los juegos clásicos de estrategia, con las características de TABS de limitar al jugador al asiento del espectador durante ciertas fases del juego. Utilizando el motor de desarrollo de Unity y el lenguaje de programación C#, se plantea el desarrollo de un juego cuyo gameplay loop esté dividido en dos fases; una fase inicial, donde el jugador controla unidades que recolectan recursos y construyen edificios para crear unidades más fuertes, con la finalidad de defender una estructura clave en el campamento del jugador, y una segunda fase donde una vez creados los soldados y estructuras, el jugador pierde el control de las unidades y este es traspasado a un sistema inteligente, que a través de técnicas de agentes inteligentes reactivos y planificación, mueve y coordina las unidades para defenderse de los enemigos que atacarán por hordas. El papel del jugador en esta segunda fase consistirá principalmente en analizar cómo funciona, y cómo no, la estrategia desplegada en la primera etapa, aprender de esta y prepararse de nuevo para la primera fase, que volverá a comenzar cuando la horda previa termine.

1.1 Motivación

Nuestro juego, Galactic Defense, es un proyecto impulsado por motivaciones personales y profesionales, desarrollado con la finalidad de poder dar un paso importante en la aplicación práctica de todos los conocimientos teóricos obtenidos durante los estudios universitarios y de poder emprender la creación de proyectos de mayor escala.

En el terreno personal, a lo largo de la última década una de mis pasiones más influyentes ha sido la de diseñar y desarrollar videojuegos, pasión que me impulsó a estudiar la carrera de Ingeniería Informática para poder obtener las herramientas necesarias para incursionar en las

empresas más importantes de este sector. En los últimos cuatro años he desarrollado una serie de juegos pequeños de menor complejidad que no tenían enemigos móviles y mucho menos inteligentes; esperaba el momento de hacerme con herramientas que me permitieran aumentar la sofisticación y el dinamismo de mis juegos. Durante los últimos dos cursos de la rama de Computación aprendí a crear sistemas inteligentes, de planificación y de aprendizaje. Estos conocimientos no nos lo enseñaron enfocados al ámbito de los videojuegos; sin embargo, era fácil extrapolarlos a esa área; de tal forma que este proyecto representa no solo el juego más grande que he creado hasta los momentos, si no también, la forma de aplicar muchas de las técnicas que me impartieron en clase al área que más me apasiona.

Este proyecto está impulsado también por motivos profesionales. El sistema Goal Oriented Action Planning (GOAP) es uno de los modelos de inteligencia artificial para videojuegos más utilizados en la industria. Este trabajo nos permite, no solo realizar un análisis exhaustivo de este modelo y su funcionamiento, si no también, desarrollar los conocimientos teóricos para la creación de un sistema GOAP y aplicarlos a un juego propio. Este desarrollo va a permitir la inclusión de un juego a un portafolio, que no solo evidencie la capacidad de trabajar en proyectos grandes, sino que también demuestre la posesión de herramientas y conocimientos para desenvolverse con comodidad en un género muy diferente a los previamente desarrollados.

1.2 Objetivos

Generales:

1. Crear un proyecto que combine los conocimientos adquiridos durante la carrera en la rama de computación, con las técnicas existentes en el ámbito del desarrollo de videojuegos.

2. Ilustrar la potencialidad de la Inteligencia Artificial aplicada al entretenimiento.

Específicos:

1. Sentar las bases para que un sistema inteligente controle a un grupo de agentes, de tal forma que la escalabilidad de las acciones que pueden tomar estas

unidades sea sencilla y no requieran de muchas horas para planificar todos los comportamientos deseados.

2. Crear un prototipo funcional y entretenido de un juego con menú, gameplay y condiciones de derrota y victoria.
3. Ampliar el catálogo de géneros de juego y herramientas con las que trabajar en un futuro.
4. Crear un sistema inteligente GOAP, de tal forma que se pueda extrapolar sin dificultad a otros proyectos.
5. Optimizar un sistema inteligente GOAP utilizando programación con hilos en Unity.
6. Realizar un juego completo con modos de juego, más tipos de unidades y gráficos en 3D.

1.3 Metodología

En primer lugar, se realizó un estudio previo de la bibliografía pertinente sobre el sistema Goal Oriented Action Planning (GOAP), así como varios proyectos donde se aplicaron este y algunas de sus variantes.

Posteriormente, se procedió a la creación del documento de diseño del juego, donde se establecieron las mecánicas, el estilo del arte, el estilo de la interfaz de usuario y una primera versión de la narrativa para el juego.

Por último, se procedió al desarrollo del juego utilizando las herramientas que mencionaremos a continuación para la organización, programación, diseño de arte y desarrollo del mismo.

1.4 Estructura

La memoria de este trabajo contiene un total de ocho capítulos en los que se describe toda la producción del juego Galactic Defense. Con el fin de facilitar la comprensión del trabajo, se va a proporcionar a continuación un breve resumen de cada apartado.

La introducción sirve como punto de partida para comprender el trabajo, aquí podemos encontrar las motivaciones para la realización del mismo, los objetivos que se pretenden cumplir con el proyecto y la metodología utilizada. Después expondremos las herramientas empleadas, un breve capítulo donde se describen de forma muy resumida los softwares utilizados. El siguiente capítulo nos introduce en el análisis teórico de las tecnologías que se utilizaron, más específicamente presenta el sistema de Goal Oriented Action Planning y el modelo Bird-oid Object; cómo funcionan, quién los creó y cuál es su finalidad.

Los siguientes dos capítulos son los más importantes, puesto que nos detallan todo el desarrollo del proyecto. Primero establecemos un análisis del problema, cuál es el juego que se quiere crear, y cuáles los requisitos que se pretenden satisfacer con el mismo. Después se habla de forma detallada del desarrollo, empezando por el documento de diseño donde están explicadas las mecánicas a implementar, el diseño del arte que se quiere emplear, las unidades, sus acciones, los enemigos y el objetivo del juego, entre otras; posteriormente se detalla la arquitectura de clases del juego, la forma en la que se creó la inteligencia artificial, y las diferentes mecánicas explicando en detalle las clases y métodos que se utilizan.

Luego, encontramos la conclusión del trabajo que informa si los objetivos fueron alcanzados; en caso contrario, proporciona una explicación de los problemas y errores que surgieron durante el desarrollo del trabajo. Y también tenemos aquellos puntos que podrán trabajarse en un futuro para mejorar o ampliar de alguna forma este proyecto.

Los últimos dos capítulos pertenecen a las referencias y al glosario de términos empleados en la memoria.

Capítulo 2

Estudio estratégico

Con la finalidad de entender mejor el planteamiento del proyecto y la razón de las decisiones de diseño que se tomaron para crear este juego, consideramos necesario analizar los productos con los que se puede comparar Galactic Defense. Estos son los más importantes del género RTS y aquellos que han generado algunas de las mecánicas de juego más interesantes a lo largo de los años.

No podemos hablar de la categoría de estrategia en tiempo real sin mencionar al que se conoce como el padre del género, Dune 2: The building of A Dynasty, muchas de las mecánicas más representativas de los RTS dieron sus primeros pasos en este juego, como el Fog Of War, la recolección de recursos, el gameplay especial para ser usado con ratón y la construcción desde cero de una base de operaciones; mecánicas que incluso los juegos de hoy en día siguen usando sin mucha variación (Lee, 2001).

Ilustración 1. Portada y captura de gameplay del juego Dune 2: The building of A Dynasty

Para el momento en el que salió Dune 2: The building of A Dynasty, los juegos de RTS no eran mundialmente conocidos y tenían un nicho relativamente pequeño; no obstante, de la

mano de la misma compañía, Westwood Studios, surgió otra joya del género que marcó el camino de los juegos de real time strategy hacia la fama: la serie de Command & Conquer (C&C)[1]. Siguiendo los pasos de su antecesor, estos juegos se basan en las mismas mecánicas de jugabilidad, la misma disposición de las interfaces e incluso los mismos objetivos a conseguir en una campaña; sin embargo, lo que realmente consiguió que esta serie obtuviese más de 450 millones de dólares vendiendo más de 15 millones de copias, fue su modo multijugador, que permitía que pudiese ser jugado cientos de veces sin que la jugabilidad perdiese su valor (Larson, 2002).

No obstante, C&C no es considerado la cúspide del género, puesto que sobre las bases que asentó Dune 2: The building of A Dynasty y que mejoró la serie de Command & Conquer, nació la mundialmente conocida serie de Age Of Empires (AOE)[2], una saga que logró vender más de 25 millones de copias y amasar un total de un billón de dólares en el proceso (Ali, 2020). Este juego no trajo ninguna mecánica sumamente novedosa a la mesa, ya que el proceso de construir bases, acumular ejércitos y destruir a los enemigos había sido utilizado y refinado por muchos otros antes de la concepción de AOE; sin embargo, fueron los pequeños detalles implementados por Ensemble Studios los que le dieron mucha más profundidad a las posibilidades estratégicas dentro del juego, combinado con la representación histórica de civilizaciones reales como la Romana, la Mesopotámica o el Shogunato japonés, lo que consiguió darle al juego ese carácter innovador que le permitió convertirse en uno de los pilares fundamentales de los RTS (Madsen, 2020).

Ilustración 2. Portada y captura de gameplay del juego Age Of Empires

[1] https://www.ea.com/games/command-and-conquer
[2] https://www.ageofempires.com/

El último juego que logró alcanzar una trascendencia global y que tuvo mucho éxito fue Starcraft 2[3], un RTS de la serie de Starcraft. A diferencia de los otros juegos antes mencionados, Starcraft 2 no se basó en ninguna mecánica nueva que le diera protagonismo en la escena RTS; más bien la innovación que este juego trajo consigo estaba alejada de las mecánicas, los soldados, los gráficos, etc.; la empresa creadora del juego: Blizzard Entertainment[4] realizó grandes inversiones de capital para popularizar la idea de los RTS como un e-sport, teniendo a Starcraft y Starcraft 2 como la bandera de este género en la escena competitiva. De esta forma, Starcraft llegó a ser uno de los e-sports más populares del mundo impulsando incluso la idea de los e-sports como competencias profesionales de alta envergadura. Se llevaron a cabo torneos con premios que llegaron a alcanzar la cantidad de 700 mil dólares (Scholz, 2019).

Ilustración 3. Portada y captura de gameplay del juego Starcraft

A parte de los juegos previamente mencionados, los más importantes del género RTS durante la era dorada del mismo, se han desarrollado otros de menor envergadura y con públicos objetivos mucho más reducidos, pero que han intentado revolucionar la experiencia de los fanáticos de este género de estrategia. Es necesario mencionar juegos como Totally Accurate Battle Simulator[5] que se basa en la simulación de enfrentamientos más que en la administración de recursos y construcción de bases, y se centra en que el jugador pueda disfrutar de batallas que por diseño rayan en lo ridículo y lo gracioso. También se debe destacar el juego There Are Billions[6], que permite al jugador enfrentarse en un modo de clásico RTS a miles de enemigos

[3] https://starcraft2.com/en-us/
[4] https://www.blizzard.com/en-us/
[5] https://landfall.se/totally-accurate-battle-simulator
[6] http://www.numantiangames.com/theyarebillions/

zombis; cuya dificultad se encuentra en el número más que en la estrategia y que presentan un espectáculo visual cuando las fuerzas del jugador se enfrentan a grupos masivos de enemigos.

2.1 Crítica al estado del arte

Como hemos podido constatar, los juegos de RTS, después de la creación de Dune 2: The Building of A Dynasty, no lograron aportar grandes innovaciones en las mecánicas básicas del género; la idea siempre fue la misma, al igual que los objetivos y la forma de alcanzarlos. Los mayores avances ocurrieron en el aspecto gráfico o en el surgimiento del modo competitivo en línea. Muy a diferencia de los otros géneros que alcanzaron fama durante la década de los 2000, como los First Person Shooters (FPS), Role Playing Games (RPG) o plataforma; los RTS se quedaron estancados en sus ideas casi desde la concepción del género, perdiendo así de forma paulatina la base de fanáticos que lograron amasar con las primeras iteraciones de estas series.

2.2 Propuesta

Con el diseño de Galactic Defense proponemos retomar muchas de las mecánicas clásicas del género de RTS, pero decantándonos por una idea poco intuitiva que plantean algunos juegos de estrategia más recientes: el jugador a veces solo quiere aguardar ansiosamente el desarrollo y resultado de la estrategia escogida y observar cómo los enfrentamientos ocurren en la pantalla, así como sucede con TABS o There Are Billions. Por ende, la propuesta consiste en un juego de estrategia que tiene a su vez una sección del gameplay loop donde el jugador queda limitado a ver cómo funcionan sus decisiones en el desarrollo de la batalla y esperar lo mejor.

Nombre	**Año de lanzamiento**	**Gráficos**	**Características principales**
Dune 2: The Building of a dynasty	1992	2D top down sprites	Primer Juego del género. Pionero en construcción y administración de recursos. Primer uso del Fog Of War.
Command & Conquer	1995	2D top down sprites	Primer juego en introducir el modo multijugador competitivo.
Age of Empires	1997	2D isometric sprites	Uso de civilizaciones reales. Mayor profundidad estratégica.
Starcraft 2	2010	3D models	Multijugador competitivo con torneos internacionales. Fue uno de los primeros juegos reconocidos como e-sport.
Totally Accurate Battle Simulator	2016	3D models	Juego de estrategia que no se basa en la administración de recursos ni en la microgestión de unidades. Las unidades se mueven solas una vez colocadas en el mapa. El entretenimiento del jugador se basa en el espectáculo visual que ofrecen los soldados al pelear.

There Are Billions	2017	2D isometric sprites	El jugador pelea contra hordas enormes, conformadas por cientos de zombis.
Galactic Defense	2021	2D top down sprites	El jugador controla la recolección y administración de recursos al igual que la creación de tropas. En el momento del enfrentamiento entre los aliados y los enemigos el jugador toma un rol más pasivo y disfruta del espectáculo y realiza nuevos planes.

En síntesis, luego de analizar los juegos ya mencionados y de evaluar sus virtudes y defectos, decidimos realizar un proyecto donde se incluyeran las propuestas más acertadas y se corrigiera su defecto más notable: la falta de innovación en la dinámica del juego que lleva sin evolucionar desde su concepción. Decidimos explorar una de las opciones posibles, que consiste en importar la mecánica principal de TABS, un simulador de estrategia, y aplicarla a un juego de estrategia en tiempo real.

Capítulo 3

Análisis de la tecnología utilizada

Para el desarrollo del videojuego hicimos uso de la anteriormente mencionada arquitectura de planificación GOAP y del modelo de simulación de movimiento artificial Boids.

3.1 GOAP

El sistema de Goal Oriented Action Planning (GOAP) es un sistema de inteligencia artificial creado por el ingeniero Jeff Orkin en 2005 para darle vida a los enemigos de F.E.A.R., un videojuego de disparos en primera persona, desarrollado por Monolith Productions. Como explica Orkin en el documento divulgativo, el propósito de este sistema es dotar al juego con una experiencia de combate intenso y excesivo, inspirado en las películas de acción de la época (Orkin, 2006).

Con la finalidad de explicar la composición y el funcionamiento del sistema GOAP vamos a dividir su estudio en 3 partes diferentes, pues este sistema es básicamente una combinación de una máquina finita de estados, un árbol de búsqueda que emplea A* y un planificador derivado del STRIPS planner.

3.1.1 STRIPS

STRIPS es un sistema de planificación creado en la universidad de Stanford en 1970, su nombre viene del acrónimo “STanford Research Institute Problem Solver”. Este planificador consiste en tres elementos fundamentales: metas, acciones y estados. Siendo las metas aquellos

estados a los que se quiere llegar, partiendo de un nodo inicial y ejecutando acciones para ir generando diferentes estados intermedios.

Los estados en STRIPS están formados por un conjunto de literales que representan la situación del mundo en un momento dado, estos estados se generan cuando una acción se ejecuta sobre uno previo cambiando sus propiedades o literales.

Las acciones están compuestas por dos elementos fundamentales: precondiciones y efectos. Las precondiciones establecen aquellos valores que tiene que tener el estado actual para que se pueda ejecutar dicha acción, y los efectos son los cambios que se aplican al estado actual para generar uno nuevo, una vez que se haya ejecutado dicha acción; esto último ocurre con la aplicación de dos listas, una de borrado que elimina conocimientos del mundo que ya no son vigentes en el nuevo estado y una lista de agregado que añade los nuevos conocimientos que se generan con la ejecución de la acción (Nilsson, 1998).

3.1.2 ¿Qué es Goal Oriented Action Planning?

GOAP es un sistema que introduce una solución elegante y eficiente a la indeseable complejidad de los sistemas inteligentes. Marcando una diferencia con las inteligencias artificiales de la época; que estaban basadas en máquinas finitas de estados (FSM por sus siglas en inglés), con tantos estados como acciones tenían los personajes; GOAP emplea únicamente tres estados ya que, como dice Orkin, “todo lo que hace una I.A es moverse por ahí y ejecutar animaciones”. Estos estados son: “GOTO”, o ir a algún sitio, “Animate”, ejecutar una animación que sería lo mismo que realizar una acción y “Use Smart Object”, que sería muy similar a “Animate”, pero empleando objetos específicos como, por ejemplo, armas de fuego. Orkin quería no solamente optimizar la toma de decisiones para que esta fuese más dinámica y eficiente, sino también, simplificar el código de la inteligencia artificial para que les resultase más fácil a los desarrolladores y diseñadores ampliar el catálogo de acciones que podían ejecutar los enemigos; para esto, separó la lógica de la toma de decisiones de la FSM, puesto que, en los sistemas antiguos, con cada set de acciones nuevas introducido en la I.A, el número de conexiones entre estados que había que realizar se volvía cada vez más complejo. (Orkin, 2006)

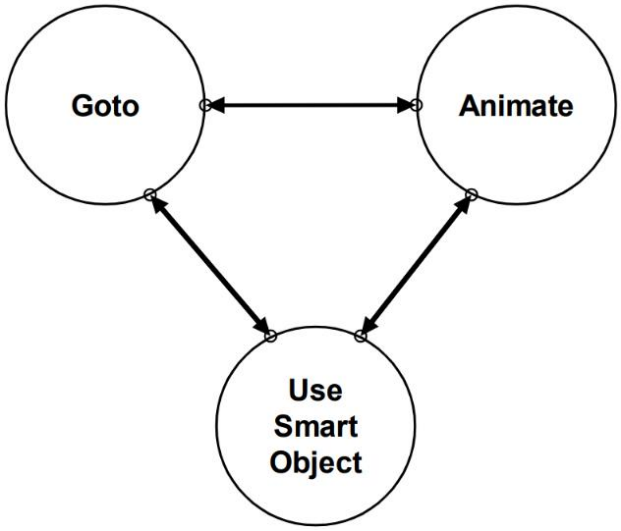

Ilustración 4. Máquina finita de estados diseñada por Orkin

Las acciones que va a realizar la I.A en los estados Animate y Use Smart Object se definen mediante un sistema de planificación basado en STRIPS. Se crean de forma separada las posibles acciones a realizar como: atacar, atacar cuerpo a cuerpo, patrullar, fuego de cobertura, abrir una puerta, entrar por la ventana, etc. Estas acciones se crean con cinco componentes: precondiciones, efectos, costo por acción, precondiciones procedurales y efectos procedurales. Tanto las precondiciones como los efectos funcionan de forma muy similar a como se definen en el sistema STRIPS; una acción no puede ejecutarse si el estado actual no cumple todas las precondiciones necesarias y una vez se ejecute se genera un nuevo estado, teniendo este almacenados los efectos de la acción. Ahora bien, para poder agilizar la toma de decisiones y permitir priorizar el uso de algunas acciones sobre otras, se agregó a cada acción un coste, dependiendo de cuales acciones consideraban los diseñadores como las más convenientes de ejecutar o las que tuviesen más lógica en ciertos contextos. Orkin también introdujo el uso de precondiciones procedurales; estas son precondiciones que no se pueden definir de forma sencilla con el uso de un literal en el estado, y que necesitan cálculos más complejos que aumentarían la carga del procesador si se ejecutaran de forma constante. Por último, como el sistema STRIPS es un planificador que no toma en consideración que una acción no se ejecuta de forma instantánea, y en el ámbito del videojuego F.E.A.R. era necesario que las acciones tomaran su tiempo, se añadió el uso de efectos procedurales. Estos se conectan con la máquina finita de estados y le indican al personaje que tiene que empezar la animación de ejecución de la acción, y terminar dicha animación antes que se inicie la siguiente acción del plan (Orkin, 2006).

3.1.3 GOAP como evolución a la FSM

En la época en la que se creó el sistema GOAP, la inteligencia artificial presente en los videojuegos estaba compuesta casi en su totalidad de máquinas de estado finito, con decenas de acciones y metas conectadas entre sí para formar un comportamiento complejo que pudiera entretener a los usuarios del juego. Debido a esta complejidad, añadir acciones se volvía cada vez más difícil, ya que aumentaban los nodos de la máquina de estados y la replicación de los comportamientos deseados por los diseñadores requería un análisis muy detallado; por esto, el número de acciones posibles no solía escalar más allá de las decenas. GOAP toma una aproximación diferente ya que, como se comentó con anterioridad, reduce el número de estados a tres y aísla completamente las acciones de la máquina de estados.

Al quedar las acciones separadas de la máquina de estados y aplicadas dentro de un sistema de planificación, se generan una serie de ventajas que no posee el modelo de FSM. En primer lugar, las acciones y las metas quedan separadas por completo, contrariamente a lo que ocurre con el FSM donde una meta está unida a una serie de estados acción, que al ejecutarse conducen a la meta deseada. Si dos enemigos diferentes tenían la misma meta, como por ejemplo atacar al jugador, era necesario crear ramas de estados particulares para cada tipo de enemigo aumentando la complejidad de una sola meta. Usando el sistema GOAP se tiene la capacidad de dotar a cada personaje con un conjunto de acciones ejecutables, y que sea el planificador el que decida qué acciones tomar dependiendo del estado en el que se encuentre cada uno. También podemos destacar la modularidad con la que se pueden crear acciones para los personajes; como no es necesario definir un camino que ejecute una acción nueva, el único trabajo que se tiene que realizar es definir la acción de forma aislada, con sus precondiciones, efectos, costos, etc.; el planificador incluirá dicha acción en el plan, en caso de necesitarla.

La otra ventaja importante del sistema GOAP es la solución dinámica de problemas, pues permite la planificación de acciones para llegar a una meta; si en algún momento un personaje intenta ejecutar una acción, pero esta se encuentra imposibilitada por el jugador o por otro personaje, entonces se marca esta acción como imposible y se vuelve a llamar al planificador para que intente encontrar otra forma de alcanzar la meta deseada. Esto sería muy difícil de lograr con una FSM, puesto que sería necesario programar de forma manual cada posible escenario en el que la I.A fallase en realizar una acción, y diseñar un plan de contingencia para

cada una; llegando incluso a ser necesarios planes de contingencia para los mismos planes de contingencia, lo que aumentaría significativamente la complejidad del autómata. (Orkin, 2006)

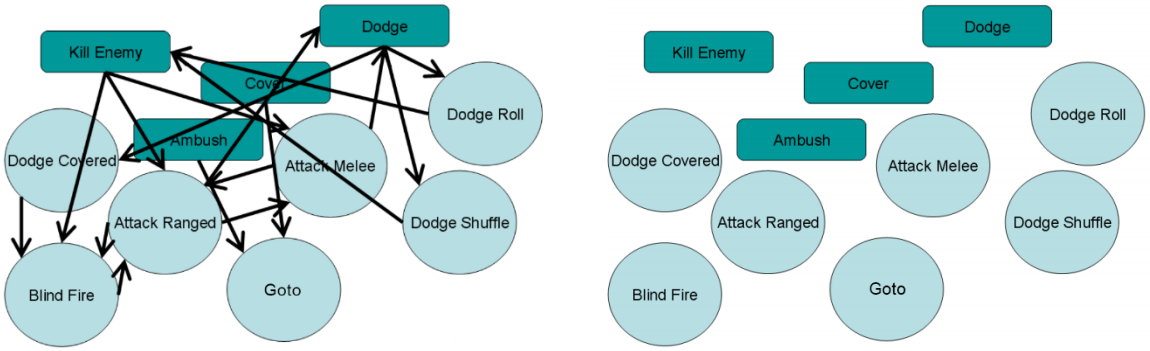

Ilustración 5. Representación gráfica de un FSM (izq.) y el modelo GOAP (dcha.) extraída del documento de Orkin (Orkin, 2006)

3.1.4 Usos de GOAP en la industria de videojuegos moderna

El sistema GOAP fue creado para el juego F.E.A.R en 2005, recibiendo numerosas críticas positivas por el dinamismo y realismo que ofreció su I.A a los jugadores; no obstante, las exigencias han aumentado en los 16 años transcurridos desde su desarrollo y, como es de esperarse, este sistema ha sido renovado constantemente por las distintas empresas líderes en el desarrollo de videojuegos; sin embargo, los componentes e ideas principales del sistema GOAP se han mantenido en su mayoría inmutables con el paso del tiempo.

Una de las empresas más importantes que ha empleado este sistema para darle vida a los contrincantes en un juego, ha sido Monolith Productions[7]en el software Shadow Of Mordor[8], cuya inteligencia artificial fue aclamada por la crítica, ya que no solamente hacía que los enemigos se comportasen de forma realista, sino también que tuvieran la capacidad de crear historias personalizadas durante los enfrentamientos entre el jugador y sus distintos enemigos. Para este juego se utilizó el sistema GOAP, con algunas variaciones en los componentes lógicos

[7] https://lith.com/
[8] https://www.shadowofwar.com/shadow-of-mordor/

que permitieron a las unidades comunicarse unas con otras, y realizar acciones que serían mejor ejecutadas por una sola unidad que por varias (Orkin, n.d.), (Conway et al., 2017).

High Moon Studios[9] también hizo uso del sistema GOAP en los juegos Transformers: War For Cybertron, Transformers: Fall Of Cybertron y Transformers: Rise of the Dark Spark; estos juegos contaban con una variación en el sistema GOAP que permitía que las acciones se agruparan de forma jerárquica para que el planificador, en vez de elegir acción por acción, escogiera grupos inmutables de acciones. Esto les permitió a los diseñadores controlar los comportamientos de la I.A. de forma manual y que se ejecutaran ciertas acciones en un orden preconcebido (Thompson, 2016).

Otros juegos notables que usan el sistema GOAP son Rise of the Tomb Raider[10], Shadow of the Tomb Raider[11], Deus Ex: Human Revolution[12] y Empire: Total War[13].

3.2 Flocking Algorithm

El algoritmo de Flocking, también llamado BOID o Bird-oid object, es un modelo computacional de coordinación y simulación de movimientos de animales en grupo, creado por Craig Reynolds en 1986, que busca simular la manera en la que se mueven las bandadas de pájaros o los cardúmenes de peces.

Este modelo funciona usando un comportamiento emergente, en el que todos los agentes del grupo parecen estar actuando de forma coordinada por un ente en común; no obstante, cada uno de estos tiene un comportamiento propio y está actuando de manera independiente; reaccionando a los demás agentes que tiene en su entorno.

[9] https://www.highmoonstudios.com/
[10] https://square-enix-games.com/en_EU/games/rise-of-the-tomb-raider-20-year-celebration
[11] https://tombraider.square-enix-games.com/en-us
[12] https://square-enix-games.com/en_EU/games/deus-ex-human-revolution
[13] https://www.totalwar.com/

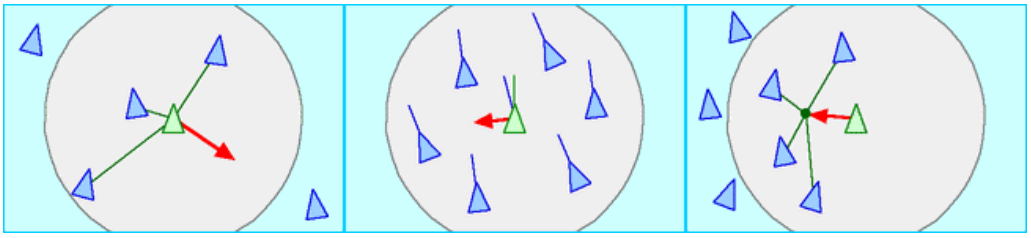

Ilustración 6. Representación gráfica de los comportamientos del sistema BOID

Para poder obtener este modelo de conducta, Reynolds creó tres sencillos comportamientos para que los agentes se pudieran mover por el entorno: separación, alineamiento y cohesión. El primero describe un comportamiento en el que el agente se va a alejar de otros objetos de sus inmediaciones, lo que le permitirá desplazarse sin chocar con otros agentes y al mismo tiempo esquivar obstáculos. El alineamiento lo capacita para moverse siempre en una dirección similar a la media de las direcciones de sus pares en un rango determinado. Y, por último, la cohesión lo habilita para moverse a un punto cercano a la posición media de sus compañeros. Aunque estos comportamientos puedan requerir de acciones aparentemente contrarias, como alejarse de sus compañeros, pero a la vez estar cerca de ellos, se utiliza un sistema de prioridad que cohesiona estas tres conductas y especifica cuál de ellas preferir (Reynolds, 1995).

En conclusión, para alcanzar la calidad de los comportamientos deseados hicimos uso de las tecnologías que nos parecieron más idóneas. Tanto el algoritmo BOID como el modelo GOAP nos dan la posibilidad de implementar comportamientos inteligentes para los agentes del juego, y gracias a las características particulares de ambos modelos podemos crear agentes con personalidades muy diferentes. El algoritmo BOID nos da agentes con una sensación más salvaje, más animal, incluso más errática, permitiéndonos así proporcionar al usuario la sensación de que está tratando con verdaderas hordas de zombis y ayudando a dar una mayor inmersión en el juego. Muy diferente es el modelo GOAP, que permite dotar de mayor coordinación a los distintos agentes; genera acciones más calculadas y complejas, de manera que, los agentes no van en una dirección en particular, pueden quedarse quietos e incluso retirarse del campo de batalla si lo ven necesario.

Capítulo 4

Desarrollo del proyecto

A continuación, trataremos de forma detallada la elaboración del juego, partiendo de la conceptualización y los detalles teóricos que hay que concretar antes de poder empezar a desarrollarlo en su respectivo motor gráfico. Luego expondremos los dos sistemas de inteligencia artificial que definieron el comportamiento de los agentes, al igual que la estructura de clases que permitió que el juego se controlase como los RTS clásicos y la estructura de objetos, escenas y otros elementos dentro del motor Unity.

4.1 Plan de trabajo

Este proyecto fue realizado siguiendo una serie de pasos que van desde la concepción de la idea hasta la realización del prototipo del producto final. Esta secuencia se organizó siguiendo la metodología ágil Kanban; la elección de esta metodología se debió a la necesidad de optimizar el tiempo disponible y a que el único recurso humano involucrado en el proyecto era el alumno que firma esta memoria. La metodología Kanban permite la mayor eficiencia de las horas de trabajo y el flujo de desarrollo más simple y directo posible.

El primer paso para realizar el trabajo fue la creación de un tablero Kanban que tuviese los componentes básicos: Tareas pendientes, tareas en proceso, tareas realizadas y backlog. La columna de tareas en proceso tenía un límite correspondiente al estimado para realizar en una semana. Las tareas se dividieron en cuatro categorías diferentes: programación, diseño, arte e investigación. Las tres primeras categorías no requieren más explicación que la que su nombre indica y serán ampliamente desarrolladas en posteriores apartados; en cuanto a la categoría investigación, se refiere a las tareas que llevaron a encontrar toda la información con respecto a las tecnologías necesarias para el proyecto, las cuales también serán suficientemente expuestas.

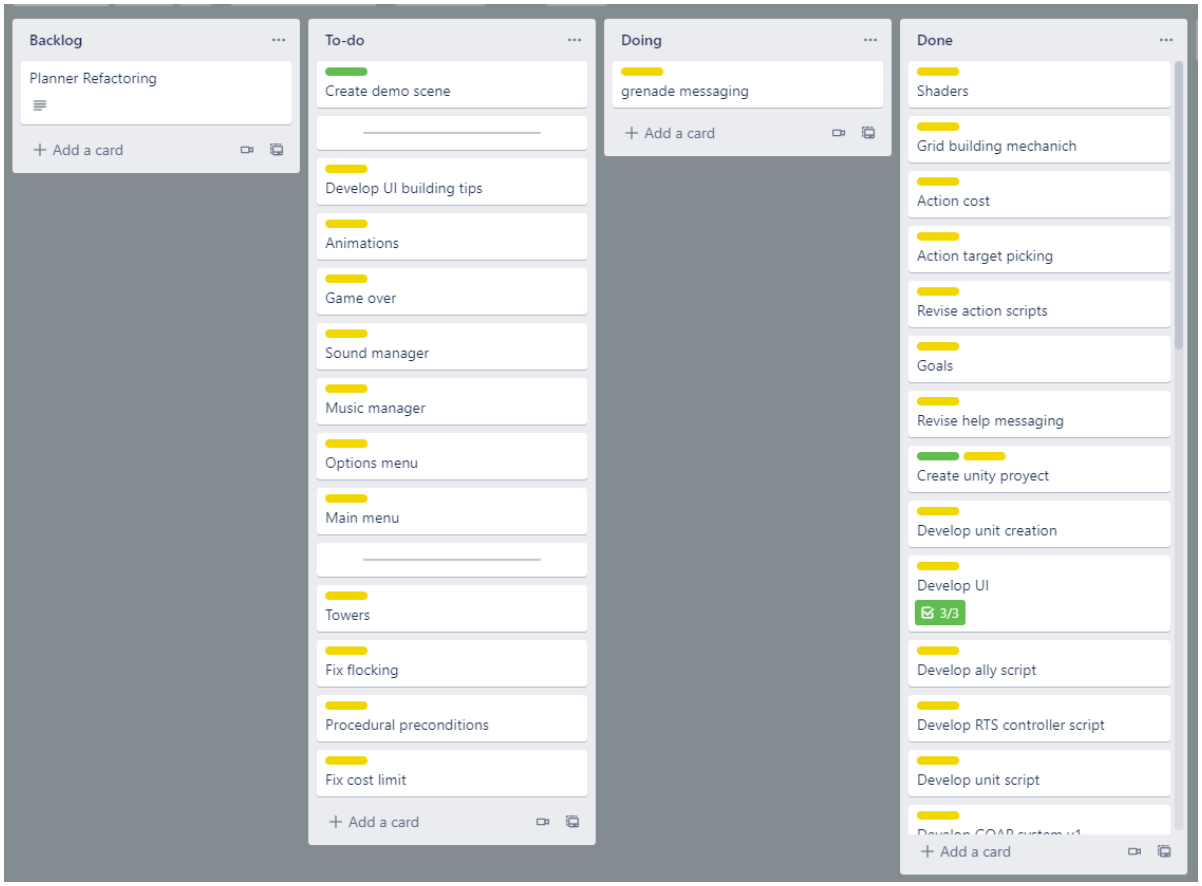

Ilustración 7. Captura del tablero Kanban utilizado por el alumno

Antes de poblar de tareas el tablero Kanban se realizó un paso previo que fue la conceptualización del juego y la creación del documento de diseño, disponible en una sección posterior de esta memoria. Este documento fue elaborado a lo largo de unos pocos días utilizando las experiencias previas en el ámbito del desarrollo de videojuegos; para ello se hizo un proceso de brainstorming y análisis de los requisitos fundamentales que normalmente se encuentran en los juegos de RTS más importantes, además de analizar cuáles eran las necesidades del planteamiento inicial del juego. En este documento se registraron todas las áreas, tanto de programación como de diseño, jugabilidad y arte.

La siguiente tarea importante fue la investigación de las tecnologías de GOAP y de BOID, esto nos llevó aproximadamente un par de semanas y empleamos medios audiovisuales y escritos, entre los cuales destacamos los documentos divulgativos escritos por Jeff Orkin (GOAP) (Orkin, 2006) y Craig Reynolds (BOID) (Reynolds, 1995); también investigamos ciertas mecánicas de RTS implementadas en Unity, como la construcción en cuadrícula, o la

implementación de ScriptableObjects para distintos objetos del juego, todo esto se fue anotando en un documento de texto que nos sirvió de referencia para poder aplicarlo una vez terminada la fase de investigación.

El paso siguiente fue la creación de un diagrama de clases utilizando el lenguaje UML. Estimamos cuales iban a ser las clases necesarias para la ejecución de todas las mecánicas del juego; este diagrama fue modificado a lo largo del desarrollo a medida que íbamos creando clases que no habían sido estimadas en el planteamiento original; sin embargo, nos sirvió de base para iniciar el proceso de desarrollo y llenar el tablero Kanban con actividades para la creación de cada una de las clases en el juego.

Los siguientes pasos permitieron la elaboración del juego, empezando por la creación de una escena en Unity que sirviera de testeo para las distintas mecánicas; luego, la implementación del modelo GOAP; la creación de los agentes BOIDs y, por último, la implementación de las mecánicas de estrategia en tiempo real. Todas y cada una de estas partes del desarrollo están explicadas con más detalles en las siguientes secciones de este capítulo. En esta etapa realizamos los trabajos de arte que le dieron el estilo que deseábamos para juego siguiendo el documento de diseño previamente creado. Esta fue la etapa más larga del trabajo; requirió alrededor de tres meses para su finalización.

Por último, se realizó un periodo de testeo de la aplicación para poder buscar errores tanto en la inteligencia artificial, y las decisiones que tomaba como en las mecánicas del juego.

4.2 Herramientas Utilizadas

Durante el desarrollo del presente proyecto se utilizaron varias herramientas, para la programación y el ensamblaje del juego, la elaboración del arte, la organización del trabajo y la redacción de la memoria que acompaña al proyecto.

- Unity: Motor de desarrollo de videojuegos en tiempo real que utiliza C#, publicado en 2005 por Apple Inc. Este motor es uno de los más populares de la industria debido a su plan de costos accesible, al igual que su sencilla e intuitiva interfaz. La versión que usamos fue la 2020.1.9f1.

- Visual Studio 2019: Entorno de desarrollo Integrado (IDE) desarrollado por Microsoft en 1997. Es compatible con una gran cantidad de lenguajes de programación; sin embargo, para este proyecto se programó únicamente en C#.
- Trello: Es un software web de administración de proyectos que utiliza el sistema Kanban para el registro de actividades con grupos y tarjetas.
- Adobe Photoshop: Un software de edición de imágenes rasterizadas, creado por Adobe Inc., cuya principal función es la de retocar imágenes o crear composiciones. Sin embargo, su gran librería de pinceles y herramientas de dibujo nos permitió dar texturas y detalles a los elementos visuales del juego.
- Adobe Illustrator: Es un software de creación de gráficos vectoriales que nos permitió la creación de elementos limpios y escalables para su posterior rasterización en photoshop y Unity. Al igual que Photoshop, este software está desarrollado por Adobe Inc.
- Google Docs: Se trata de un software gratuito de procesamiento de texto online, desarrollado por Google, con el que creamos el documento de diseño del juego y con el que también escribimos la memoria.
- Google Drive: Es un servicio de almacenamiento de archivos en la nube desarrollado por Google, en él se alojaron todos los archivos pertenecientes al juego, de esta forma los tuvimos organizados y accesibles desde cualquier plataforma.
- Draw.io: Es un software de diagramación desarrollado por Seiber-Media que permite la creación de, entre otras cosas, diagramas UML; lo que nos facilitó el diseño de la arquitectura de clases del juego.

4.3 Diseño del juego

El primer paso consistió en crear el documento de diseño del juego (GDD, por sus siglas en inglés), en el que se especifican de forma concreta todos los detalles importantes como el tipo de gameplay, los controles, el tipo de arte a utilizar, la música y efectos de sonido, y los personajes con sus posibles acciones.

4.3.1 Gameplay

El gameplay se dividirá en dos fases principales por las que transitará el jugador de forma constante a lo largo del desarrollo de la partida. La primera, a la que denominaremos Fase de planificación, es aquella en la que el jugador moverá sus unidades a plenitud para conseguir recursos, construir defensas y entrenar soldados para que estos puedan defender el objetivo. En la segunda, Fase de defensa, el jugador se moverá por el escenario analizando los sucesos que se desarrollan frente a él de manera automática y planeando su próxima estrategia.

En la Fase de planificación el jugador controlará sus unidades con mecánicas de Point and Clic; las unidades que el jugador seleccione se moverán a donde él elija con los botones del ratón y realizarán distintas acciones dependiendo de las unidades seleccionadas. El jugador también podrá controlar ciertos edificios que se encargaran de crear las unidades que este decida. Esta primera fase estará temporizada por un reloj visible en la interfaz del juego y una vez que se termine el tiempo, la fase llegará a su fin. El jugador también podrá ser capaz de finalizar antes de tiempo esta etapa.

Durante la Fase de defensa el jugador no podrá controlar sus unidades, el éxito o fracaso de la partida dependerá de si el jugador supo posicionar y gastar de forma correcta sus recursos; cualquier fallo en el diseño de las defensas o en el posicionamiento de las unidades ocasionará que el campamento que debe defender sea destruido. Los soldados se moverán y actuarán conforme una inteligencia artificial con elementos de sistemas multiagentes reactivos; esto quiere decir que dependiendo de lo que puedan observar en su entorno tomarán distintas decisiones. Esta fase finaliza cuando el jugador haya sido derrotado (el edificio principal haya sido destruido) o cuando todas las unidades enemigas en esta fase hayan quedado fuera de combate.

4.3.2 Mecánicas

El jugador selecciona sus unidades con el clic del ratón o arrastrando para seleccionar en área. Posteriormente, podrá elegir en el menú de las unidades la acción que desea que realicen, y luego con el clic derecho del ratón apuntará al sitio donde quiere que las unidades realicen dicha acción. En caso de no seleccionar ninguna, se ejecutará la que está predefinida: moverse hacia el punto.

4.3.2.1 Mapa

El mapa en el que se desarrolla el juego es un plano de tamaño 250x250 unidades. Para el desarrollo de este proyecto, el mapa se creó de forma manual con la finalidad de evitar complicar de manera innecesaria el trabajo. Un trabajo futuro podría incluir la creación procedural del mapa. El mapa actual está poblado por distintos nodos, agrupaciones de recursos, de los que el jugador tendrá que extraer los materiales con los que construirá sus estructuras y unidades de defensa; los nodos están dispersos por el terreno, pero es necesario que el jugador tenga cerca de su zona de inicio al menos un nodo de cada recurso para que pueda empezar la expansión de su territorio. El mapa también dispone de algunos detalles visuales que no afectan las mecánicas, solo sirven para hacer el mundo jugable más inmersivo.

4.3.2.2 Creación de estructuras y unidades

El jugador a lo largo de la fase de planificación, tendrá la capacidad de ordenar la construcción de distintas estructuras y unidades que le ayudarán a prepararse para la siguiente fase; estas construcciones requerirán que el jugador disponga de los recursos necesarios, los cuales serán distintos para cada estructura o unidad; de igual manera, tendrán un retardo entre el momento en el que se inicia la acción de construir y el momento en el que ésta finaliza.

La opción para la construcción de cualquiera de los edificios se encontrará en los botones que se muestran cuando se tiene seleccionado el Centro de mandos. La construcción de unidades solo estará disponible cuando se seleccione el edificio Barracón, que no está preconstruido en el mapa, a diferencia del Centro de mandos que si lo está; el edificio Barracón solo podrá ser construido por el jugador cuando este disponga de los recursos suficientes.

No existe un límite cuantitativo para la construcción de edificios, pudiendo el jugador crear cuantos Barracones desee o los que le quepan en el mapa. Sin embargo, hay reglas en cuanto al sitio donde se pueden ubicar ciertas estructuras; no es posible colocar estructuras ocupando puestos que ya ocupan otros edificios; no se permite la creación de dos estructuras recolectoras de recursos a una distancia menor de la establecida, limitando de esta forma la cantidad de recursos disponibles; por último, tampoco se permite que el jugador construya estructuras muy alejadas de sus otras edificaciones obligándolo así a una expansión progresiva.

4.3.2.3 Recolección de recursos

Al igual que otros juegos de RTS, en Galactic Defense es necesario que el jugador administre, recolecte y almacene recursos que servirán para poder construir nuevas estructuras y unidades. Una mecánica que diferencia este juego de otros juegos del mismo género es que no es necesario disponer de unidades que se dediquen a transportar u obtener los recursos, solo es necesario construir unas estructuras cerca de las zonas ricas en recursos y estos se recolectan solos a un ritmo predefinido. Los recursos a recolectar son energía y metal; la energía se recolecta usando plantas termoeléctricas que se pueden colocar exclusivamente en las zonas donde hayan Chimeneas volcánicas; y el metal se extrae con refinerías que tendrán que colocarse en nodos visualmente marcados por cristales de color azul.

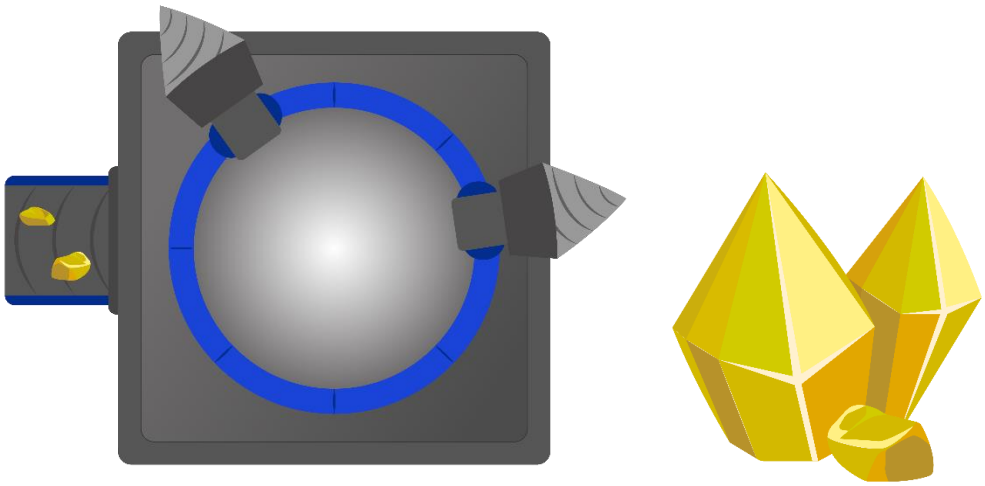

Ilustración 8. Estructura recolectora y recurso metal

4.3.3 Cámara

La cámara tendrá un formato Top-Down o cenital, siguiendo el estilo de juegos como The Escapist, Hotline Miami o RimWorld. La cámara podrá desplazarse por el mapa controlada por el jugador; no obstante, el territorio permanecerá oculto hasta que el jugador visite esa zona (Fog Of War).

La cámara se desplazará sobre el terreno de juego mostrando segmentos del territorio, ya que no le será posible abarcar todo el mapa. Si el jugador lo desea podrá acercar y alejar la cámara para ampliar zonas específicas, pudiendo así seleccionar unidades particulares o ver la acción con más detalle.

4.3.4 Controles

El movimiento de la cámara se efectuará con las flechas de dirección, con las teclas WASD o moviendo el cursor a alguno de los bordes de la pantalla. El acercamiento de la cámara se realizará con la rueda del ratón.

Las unidades aliadas se podrán seleccionar durante la fase de planificación al hacer clic en ellas o al arrastrar y hacer una selección zonal de varias unidades; una vez seleccionadas estás, se desplegará un conjunto de opciones entre las que será posible elegir: reparar, moverse o patrullar; una vez elegida la acción el jugador con el botón derecho del ratón podrá seleccionar el sitio donde desea que se ejecute dicha acción y las unidades obedecerán siempre y cuando sea posible.

Las unidades y algunas estructuras dispondrán en la interfaz de usuario de una botonera con todas las acciones disponibles para el jugador. Las estructuras tendrán disponibles acciones de construcción y las unidades, de movimiento.

4.3.5 Unidades

Las unidades existentes en el juego estarán divididas en dos bandos; las aliadas, aquellas que puede controlar el jugador y las enemigas, de las que se tiene que defender. En las primeras predominara el color azul y en las enemigas el ocre, de forma que sea fácil diferenciarlas en cualquier momento de la partida.

4.3.5.1 Aliados

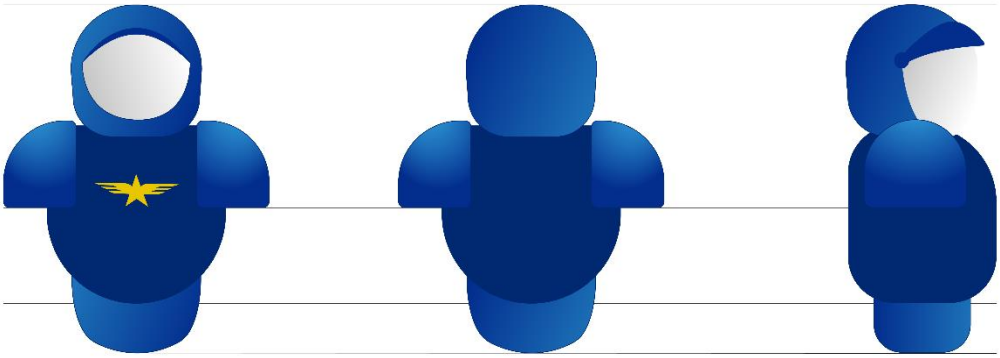

Ilustración 9. Propuesta de diseño de los soldados aliados

Los soldados son las unidades móviles que el jugador puede crear para defenderse, estas se pueden desplazar por el terreno de juego, pueden patrullar, curarse y atacar, entre otras actividades. Las acciones que puede ordenar el jugador a los soldados son reducidas, debido a que ellos tomarán sus propias decisiones cuando la fase de defensa comience; no obstante, el jugador si puede enviarlos a realizar un patrullaje; ordenar que se curen de los daños recibidos en rondas pasadas y que incorporen soldados nuevos en el escuadrón.

Los soldados cuando están bajo el control directo del jugador podrán desplazarse, curarse y patrullar. Mientras los soldados estén controlados por el sistema de I.A. las unidades podrán desplazarse, atacar a distancia con armas de fuego, atacar cuerpo a cuerpo, atacar con explosiones en el área, reparar estructuras, curarse, recargar munición y pedir ayuda a los demás escuadrones.

Las estructuras, o unidades estáticas, disponibles para los aliados cumplirían una de tres funciones, tendrían labores defensivas o de recolección y construcción.

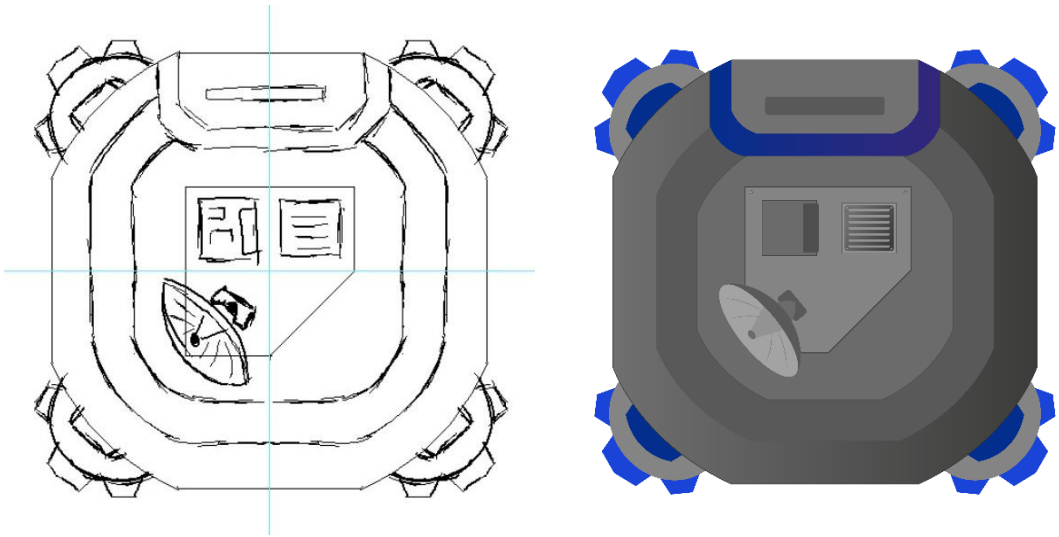

Ilustración 10. Boceto y diseño final del Centro de mandos

El Centro de mandos estará creado desde el inicio de la partida, es el único edificio que el jugador no puede construir y es el más importante del juego, puesto que si es destruido el juego finaliza de inmediato con una derrota. Desde el Centro de mando se construyen todas las demás estructuras.

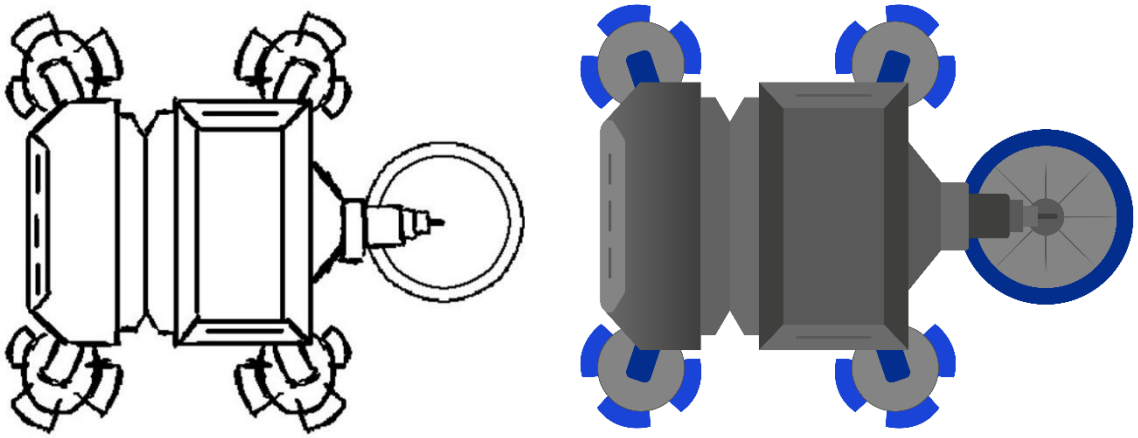

Ilustración 11. Boceto y diseño final del Barracón

El Barracón es el edificio donde el jugador ordenará la construcción de los soldados, esta es su única función, y puede haber tantos barracones como el jugador pueda crear con los recursos disponibles y el espacio permitido.

Las torres funcionan como otra unidad de defensa, pero a diferencia de los soldados, las torres serán estáticas, éstas atacaran siempre a las unidades enemigas más cercanas y proveerán al jugador de una mayor visión de la zona del mapa en la que se ubiquen.

Las murallas son obstáculos destruibles para los enemigos los cuales tendrán que atravesarlas para poder llegar al objetivo final.

4.3.5.2 Enemigos

Ilustración 12. Diseño final de personajes enemigos

Los zombis son unidades muy básicas y su comportamiento es muy sencillo: se mueven hacia la estructura principal que el jugador tiene que defender y destruyen todo a su paso, tomarán las estructuras como prioridad por encima de los soldados y la estructura principal como prioridad absoluta si la tienen en su rango de visión. Es posible que algunos zombis deambulen sin un rumbo definido mientras que otros estarán concentrados en el objetivo, de esta forma tendrán un comportamiento más natural y “salvaje”.

4.3.6 Arte

El arte del juego está hecho con sprites con una resolución de 300 pixeles por sprite. El estilo del arte de las unidades, estructuras, enemigos y terreno está inspirado en los juegos RimWorld y The Escapist; con figuras muy simples, poco más que una silueta a color del objeto; los sprites de las estructuras son una visión cenital de las mismas, mientras que las de los soldados están creados como si estuvieran vistos de perfil.

El diseño de los detalles de los soldados está fuertemente inspirado en los Space Marines de Starcraft y Warhammer 40k, con hombreras prominentes dando una sensación de una armadura sumamente gruesa y resistente a altos calibres de munición.

Las estructuras del Centro de mando y del Barracón están inspiradas también en estos dos juegos con algunos cambios para ajustarlos más al estilo general de Galactic Defense.

4.4 Arquitectura del sistema

Dividiremos este capítulo en dos partes con la finalidad de facilitar el análisis de los sistemas que componen el juego. En la primera parte realizaremos una explicación a grandes rasgos de los tres módulos de la arquitectura del sistema y cómo se relacionan entre sí. En la segunda, profundizaremos en la discusión de las características del módulo de Real Time Strategy; debido a su importancia para este trabajo y su extensión, el análisis detallado del módulo GOAP al igual que el del módulo BOID se va a realizar en las secciones siguientes.

4.4.1 Módulos

La implementación de las mecánicas del juego requirió alrededor de 50 clases que dividimos desde un principio en tres módulos diferentes: Módulo GOAP, módulo BOID y módulo RTS. El primero realiza todo el funcionamiento de la inteligencia artificial de los agentes aliados del jugador: la planificación con A*, el funcionamiento de la máquina finita de estados, las acciones que puede ejecutar el agente y la comunicación entre los distintos agentes. El segundo está encargado de los agentes antagonistas; contiene todas las clases necesarias para poder recrear el modelo BOID de Reynolds, los comportamientos generales, los comportamientos más específicos, filtros de reconocimiento de objetos y el controlador de los agentes enemigos. Por último, tenemos el módulo RTS que va a cargar con todas las mecánicas propias de un juego de estrategia en tiempo real, el movimiento de la cámara, el render de la interfaz de usuario dinámica, la recolección de recursos, la construcción de estructuras, el control manual de las unidades del jugador y el temporizador para cambiar entre las dos fases del juego.

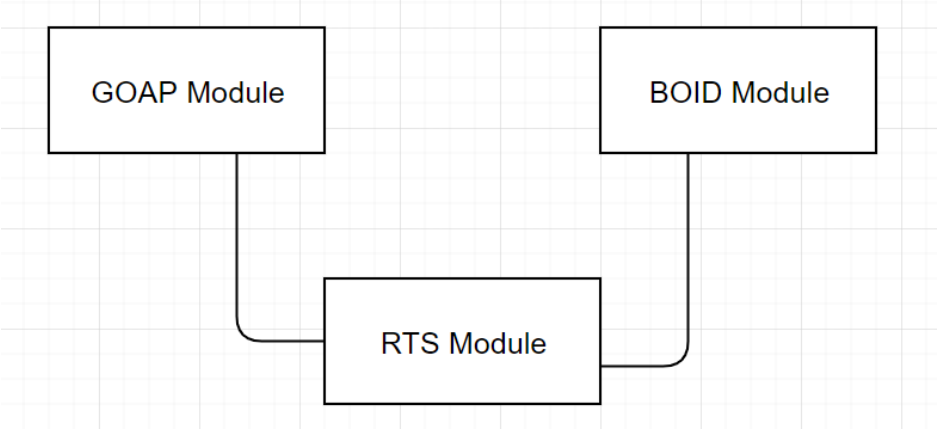

Ilustración 13. Diagrama de módulos del proyecto

Estos tres módulos están pensados para poder implementarse de forma independiente unos de otros; en parte porque la dinámica del juego así lo exige, pues el uso de la I.A. ocurre cuando el jugador ya no puede controlar sus unidades y uno de los módulos queda relegado a muy pocas acciones; y en parte porque todo se interconecta dentro de una escena de Unity, a través de GameObjects que reciben las acciones de varios módulos diferentes. El GameObject de Soldier, la unidad aliada, permite conectar el módulo RTS con el GOAP, ya que cuando el jugador crea nuevos soldados, estos pueden recibir acciones provenientes del módulo RTS mientras el jugador siga en control de las unidades. Una vez se pasa a la segunda fase estos soldados reciben acciones del módulo GOAP. Algo similar pasa con los enemigos, el módulo RTS contiene el temporizador que marca el cambio entre una fase y otra, y con este cambio se genera un número definido de unidades enemigas que están controladas por el módulo BOID.

4.4.2 Clases

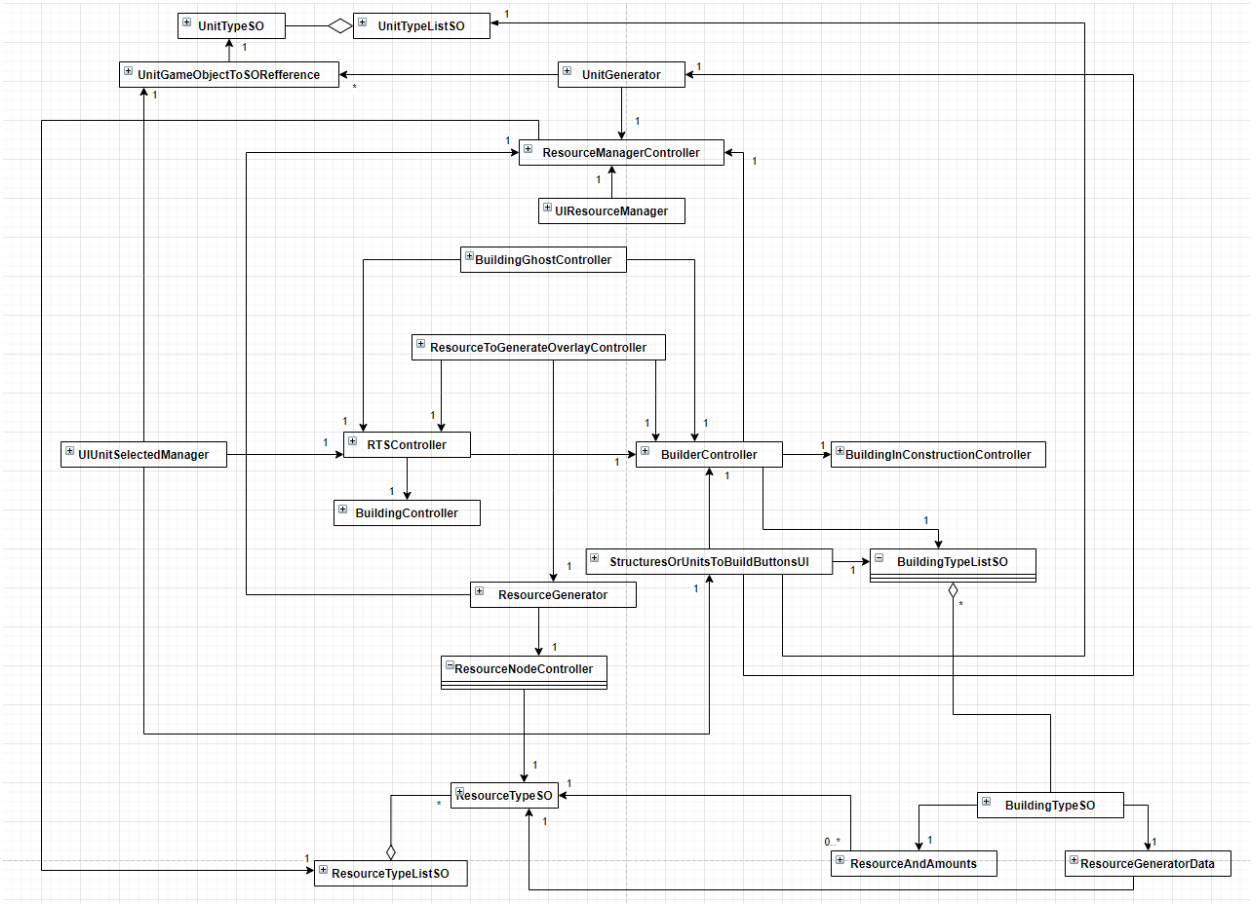

Ilustración 14. Diagrama de clases UML del módulo RTS

El módulo RTS tiene un total de 27 clases, cada una de ellas es de vital importancia para el funcionamiento correcto del sistema de real time strategy diseñado previamente; no obstante, vamos a analizar a fondo las clases que consideramos más prioritarias debido a sus diversos funcionamientos; las demás, las mencionaremos de tal manera que se entienda su función en el esquema, sin mencionar los detalles menores.

Probablemente la clase más importante de todo el módulo es RTSController. Esta clase es la encargada de recibir el input del jugador y discernir qué acción quiere ejecutar, teniendo en cuenta que el input puede ser similar para distintas acciones. También tiene entre sus funciones realizar el movimiento de la cámara, seleccionar las unidades y estructuras del juego y calcular las coordenadas del ratón dentro de la escena de Unity; es decir, todo lo que lleva al control

básico del juego sin incluir mecánicas más complejas como la construcción o recolección de recursos.

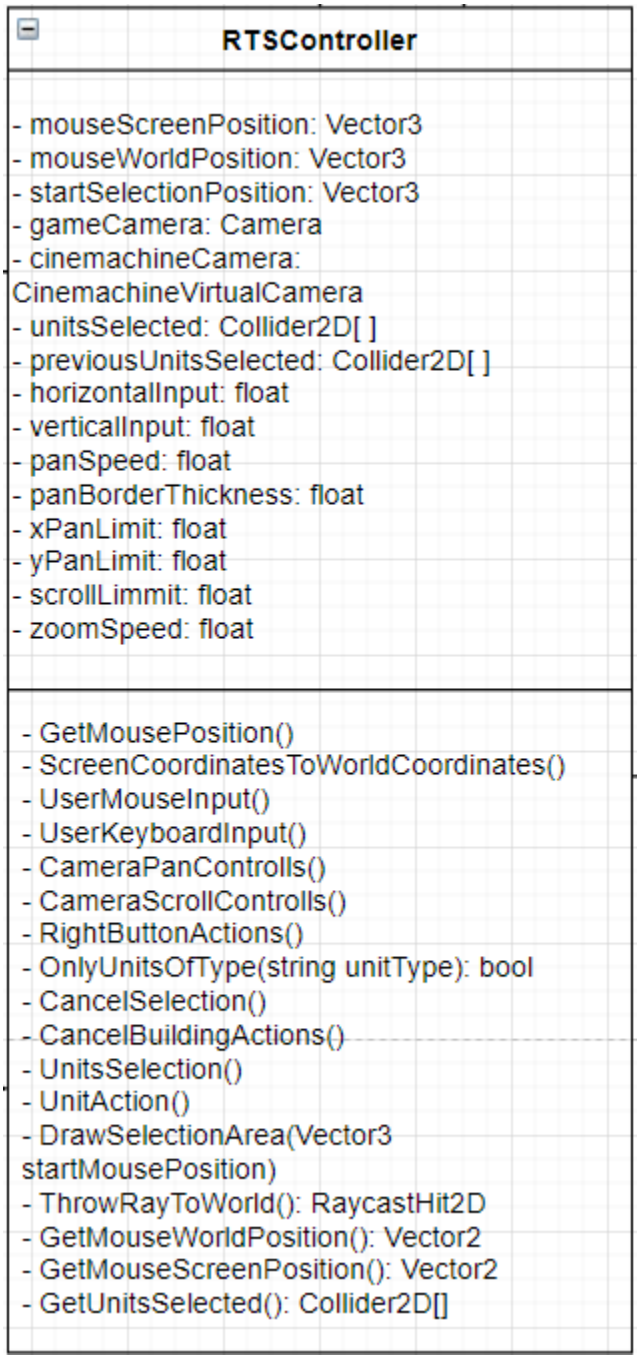

Ilustración 15. Diagrama UML detallado de la clase RTSController

BuilderController es otra de las clases importantes para la ejecución de las mecánicas en el juego; partiendo de la clase StructuresOrUnitsToBuildButtonsUI, se encarga de: conocer qué estructura o unidad quiere construir el jugador, verificar que el sitio donde quiere ubicarla cumple con las reglas de construcción diseñadas e iniciar la construcción del mismo usando la clase BuildingInConstructionController para hacer que la construcción tarde el tiempo predeterminado.

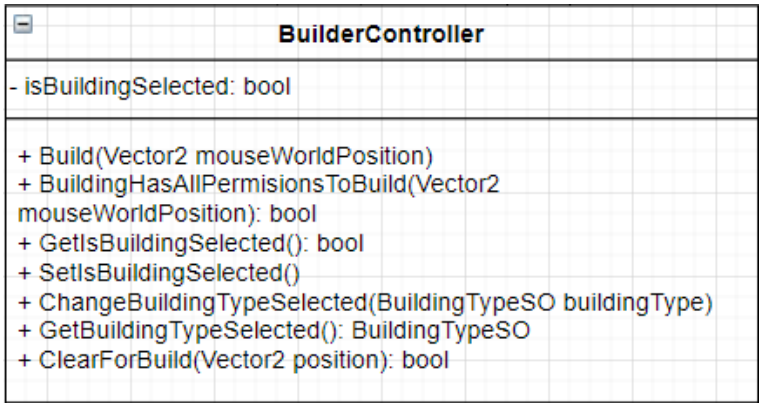

Ilustración 16. Diagrama UML detallado de la clase BuilderController

Una de las mecánicas más importantes en un RTS es la administración de recursos, esto incluye la recolección de los mismos y su posterior inversión en más solados o estructuras para expandir la base del jugador; todas estas acciones se realizan a través de ResourceManagerController, esta clase permite que cuando se va a construir una unidad o estructura nueva, se verifique si hay recursos suficientes para ello, y cuando se construye se restan los recursos de las arcas del jugador. ResourceManagerController se relaciona con ResourceGenerator, una clase que tienen todos los objetos del tipo estructura y que permite generar la cantidad de recursos que cada edificio tiene determinada por su posición en el mapa.

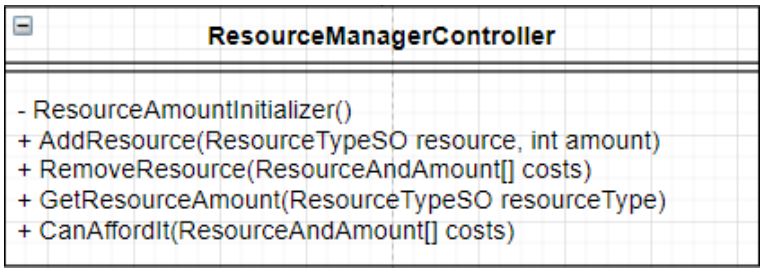

Ilustración 17. Diagrama UML detallado de la clase ResourceManagerController

Para mantener la información acerca de los tipos de estructuras, unidades y recursos existentes en el juego se usaron una serie de clases: ResourceTypeSO, BuildingTypeSO y UnitTypeSO. Estas contienen la información de costos, nombres, recursos recolectados y mucha más.

Por último, para poder mostrar toda la información necesaria en la UI, y permitir su interactividad, se usaron las siguientes clases: ResourceToGenerateOverlayController, UIUnitSelectedManager, BuildingGhostController, StructuresOrUnitsToBuildButtonsUI y UIResourceManager. Estás nos permitieron: mostrar toda la información de las unidades o estructuras seleccionadas, dar acceso a la botonera de acciones para cada unidad y edificación, mostrar el temporizador de la fase de planificación, mostrar el sitio donde el jugador va a construir y cuántos recursos va a obtener si decide construir en un lugar en particular, y ocultar toda la información irrelevante para las necesidades del jugador en los distintos momentos de la partida.

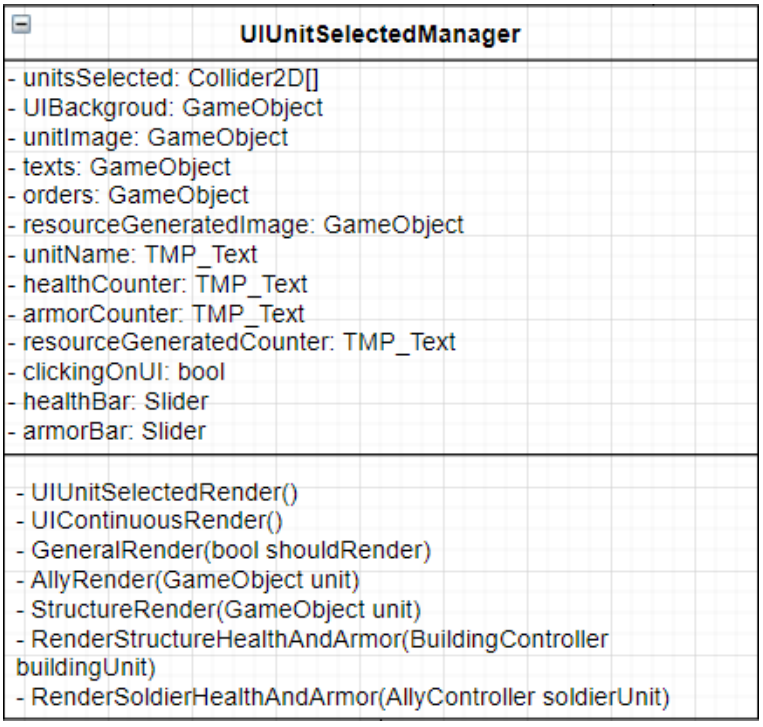

Ilustración 18. Diagrama UML detallado de la clase UIUnitSelectedManager

4.5 Aplicación del algoritmo GOAP en Unity

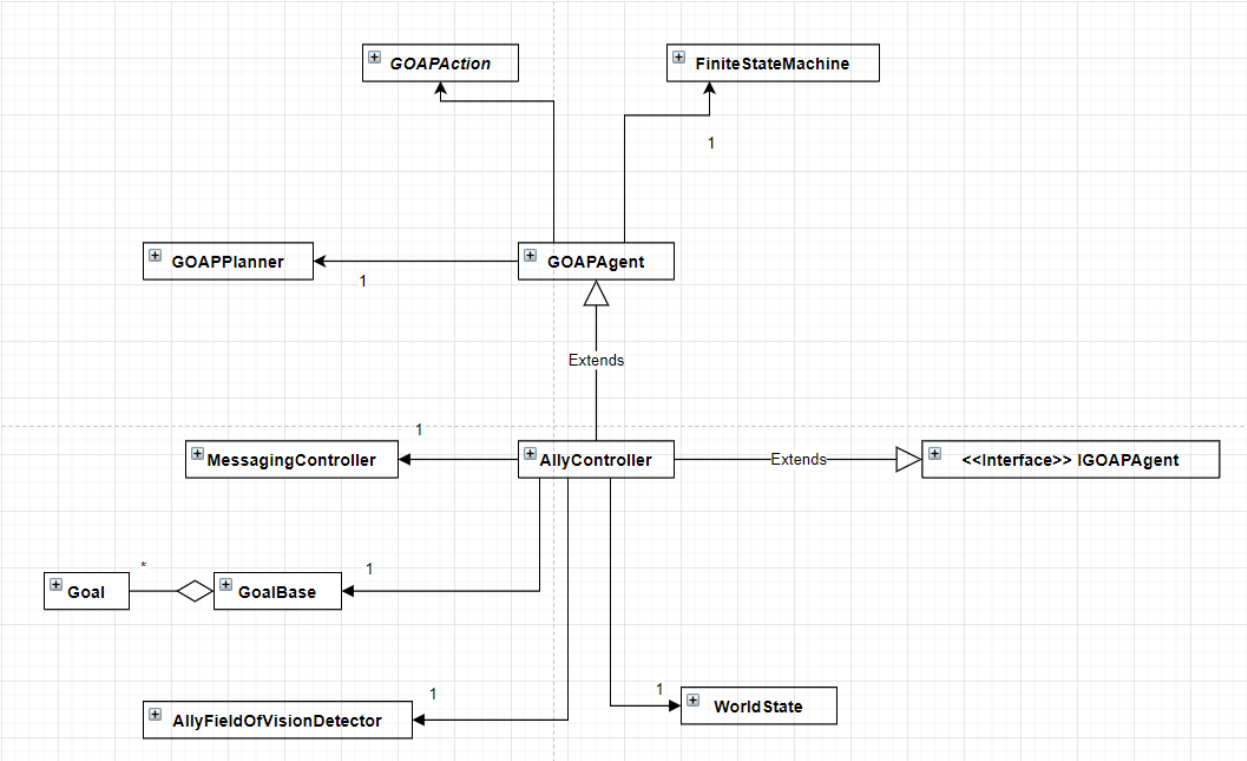

Ilustración 19. Diagrama de clases UML del módulo GOAP

El sistema de inteligencia artificial GOAP necesita de cuatro componentes fundamentales: una máquina finita de estados, encargada de decidir si generar un plan, moverse o realizar alguna acción; un planificador A* que busque el mejor plan de acción entre aquellos posibles; las acciones a realizar y los agentes que realicen las acciones.

4.5.1 FSM

La máquina finita de estados tiene como ya se describió previamente tres estados, el creador del modelo GOAP diseñó estos de forma diferente a como se van a aplicar en este proyecto. Jeff Orkin definió los tres estados como GOTO, Animate y Use Smart Object, cuyas

funciones dentro de su sistema ya se analizaron en profundidad en un capítulo anterior; nosotros cambiaremos Animate y Use Smart Object por Idle y Action.

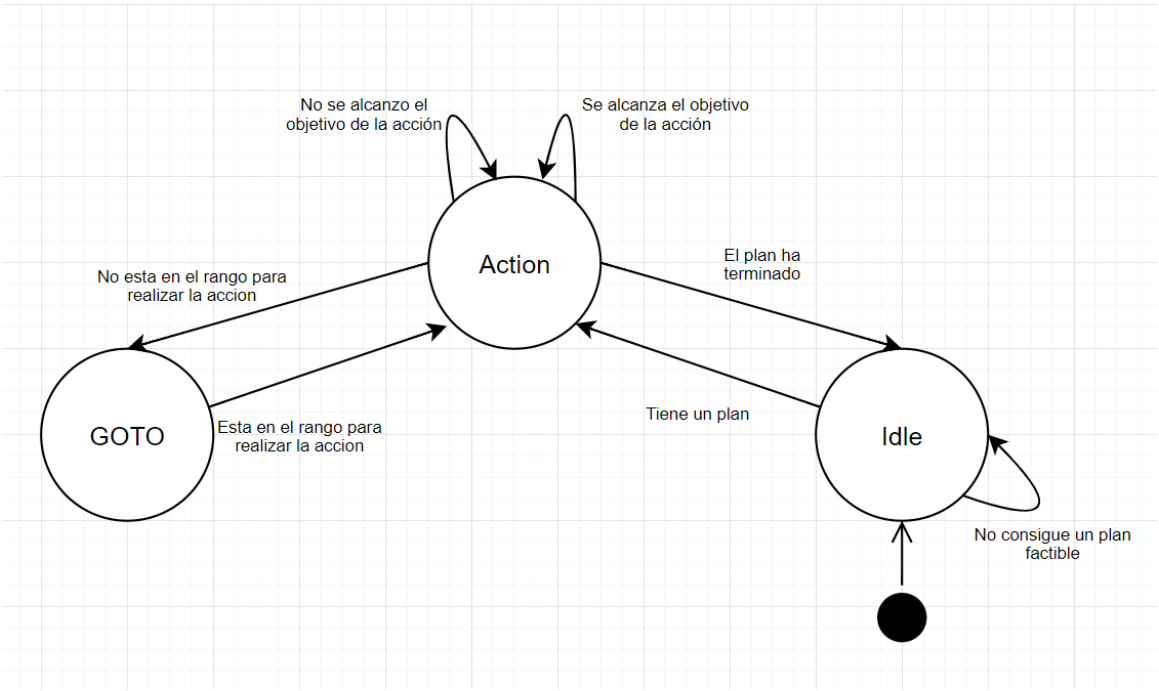

Ilustración 20. Diagrama de transición de estados

Idle, como su indica su nombre en inglés, es un estado que usaremos de base cuando el agente no tenga ningún plan con acciones pendientes de ejecutar y servirá para indicar que es necesario llamar al planificador para obtener un nuevo plan. En este estado también se obtienen, a través del agente, los hechos que puede percibir este de su entorno y las metas que tiene activas por cumplir.

Action, o acción en español, es aquel donde se ejecuta cada una de las acciones del plan del agente. Primero se verifica que exista un plan para evitar que se entre en el estado Action sin uno para ejecutar. Después se verifica si existe alguna acción sin finalizar en el plan del agente y en caso de que sea así, se confirma si el agente está en la capacidad de realizar dicha acción o es necesario que se desplace a algún lado para llevarla a cabo; si el agente se encuentra en un rango que le permita realizar la acción que se está ejecutando, se invoca el método PerformAction() de la acción; este método retorna el booleano correspondiente a si se pudo o no realizar de forma satisfactoria la acción, en caso de que no se pudiera realizar, esto le indicaría al agente que es necesario que se aborte el plan y se cree uno nuevo, con hechos actualizados

del mundo visible del agente, por lo tanto se cambia del estado Action, otra vez al estado Idle. En caso de que la acción hubiese podido llevarse a cabo, se marca como finalizada una iteración de la acción y se vuelve a entrar en el estado Action; si la última iteración de la acción completa el objetivo, esta se desencola del plan y se ejecuta la siguiente; en caso contrario se vuelve a ejecutar la misma acción. Esta doble comprobación al ejecutar la acción, nos permite evitar el apilamiento de acciones repetidas acortando el plan del agente; de esta forma, tomando como ejemplo el proyecto del juego, si un agente tiene que atacar cinco veces a un enemigo para eliminarlo, en vez de tener un plan con cinco acciones de Atacar se tiene una sola que se repetirá hasta que el enemigo haya sido eliminado y el agente haya comprobado que el target de la acción ya no se encuentra presente.

Por último, tenemos el estado GOTO que es el estado al que va a recurrir el agente si necesita desplazarse a algún punto en concreto para realizar una acción, como recoger objetos o atacar cuerpo a cuerpo. El agente mantiene en memoria una variable con la última acción a realizar por el estado Action y busca si esta acción tiene un objetivo, si el objetivo ya no existe porque fue eliminado por otra unidad, el agente vuelve a pasar al estado Idle para crear un nuevo plan; si por el contrario el objetivo sigue existiendo en el contexto actual, el estado GOTO invoca al método MoveTowardsTargetOfAction que realizará la acción de desplazarse hasta que la unidad y su objetivo estén a una distancia igual o inferior que el rango que requiere la acción en particular.

4.5.2 Planificador A*

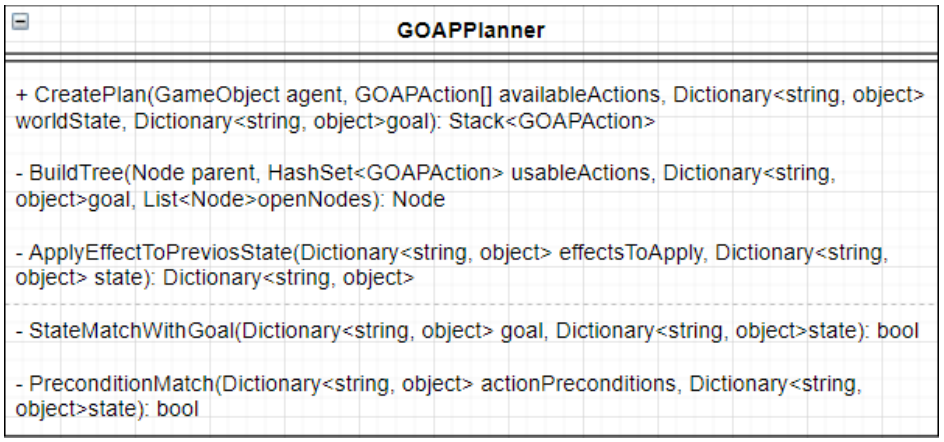

Ilustración 21. Diagrama UML detallado de la clase GOAPPlanner

Una vez que el agente se encuentre en el estado Idle el FSM va a llamar a la clase GOAPPlanner, esta clase es la encargada de crear un plan de acción a partir de: una serie de acciones disponibles, el estado del mundo del agente y una serie de objetivos. El plan de acción va a partir del estado en el que se encuentra el agente hasta satisfacer uno de los objetivos.

El planificador pasa por dos etapas fundamentales. La primera es la de depuración, en la que el planificador, a partir de la lista de las acciones disponibles para el agente, descarta aquellas que no pueda realizar en ese momento; para esto hace un chequeo de precondiciones complejas, es decir precondiciones que se van a revisar una sola vez durante la planificación ya que son más costosas que otras y sus valores son más complejos que un simple booleano o un enum. Después de este proceso quedan las acciones que el agente si puede realizar y se pasa a la construcción del árbol de búsqueda. En la segunda etapa, la construcción del árbol de búsqueda, el nodo raíz corresponde con el estado actual del mundo, y se van buscando las acciones cuyas precondiciones están satisfechas por el nodo raíz; estas acciones son agregadas a una lista de nodos frontera y la raíz, en caso de estar en la lista de nodos frontera, es eliminada de la misma. Posteriormente, se busca la acción con menor coste total estimado, que se calcula a partir de la suma del coste calculado de la acción y el valor heurístico considerado, y se expande llamando al método BuildTree de forma recursiva, cambiando el nodo raíz por el nuevo nodo acción a expandir y cambiando el estado del mundo por otro modificado por los efectos que tiene la acción del nodo a expandir. Los bucles infinitos se evitan comparando un nodo padre con el nodo hijo que en caso de ser iguales se salta dicho nodo.

La recursividad termina cuando se consigue llegar a un nodo con un estado que satisfaga la condición de la meta seleccionada.

El planificador ha sido desarrollado para no tener la necesidad de crear planes largos, la máxima cantidad de acciones continuas en un solo plan dentro del modelo que hemos creado es de tres acciones. Respetamos así los criterios de Orkins, pues según él, para poder mantener el dinamismo y la eficiencia necesaria dentro de un juego en tiempo real, el planificador no puede tomarse más que unos pocos milisegundos en calcular el siguiente plan. De esta forma utilizamos un estimador heurístico basado en grafos de planificación relajados, donde a partir de un estado inicial aplicamos todas las acciones posibles a este, obteniendo un grupo de estados resultado; luego, evaluamos si entre ellos alguno satisface el objetivo deseado y, en caso de que así sea, se asigna la suma del coste de las acciones necesarias para llegar a la solución como el coste heurístico de expandir el nodo; ahora, si no se alcanza la solución entre los estados obtenidos,

se le aplican a estos todas las acciones posibles obteniendo un nuevo grupo de estados y se vuelven a comparar con el estado objetivo. Se considera que la solución es inalcanzable desde un nodo en el momento en el que obtenemos dos conjuntos de estados iguales y se asigna un valor heurístico infinito positivo (Hoffman & Nebel, 2001).

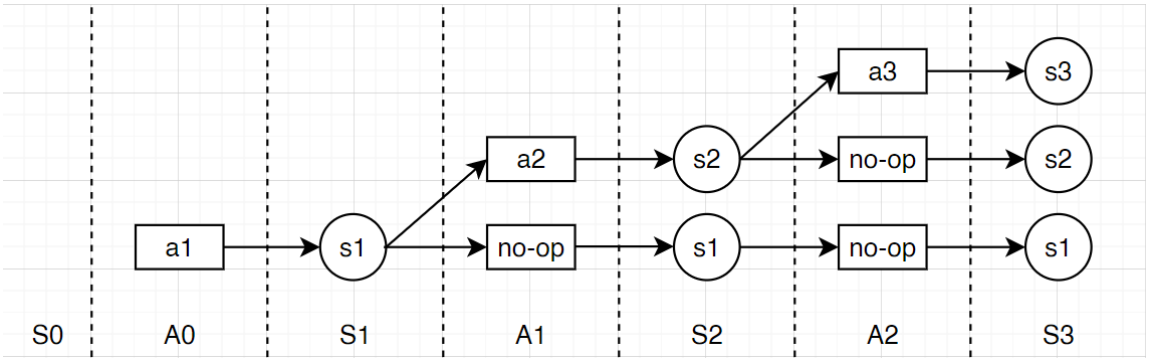

Ilustración 22. Diagrama ilustrativo del grafo de planificación relajado

4.5.3 Acciones

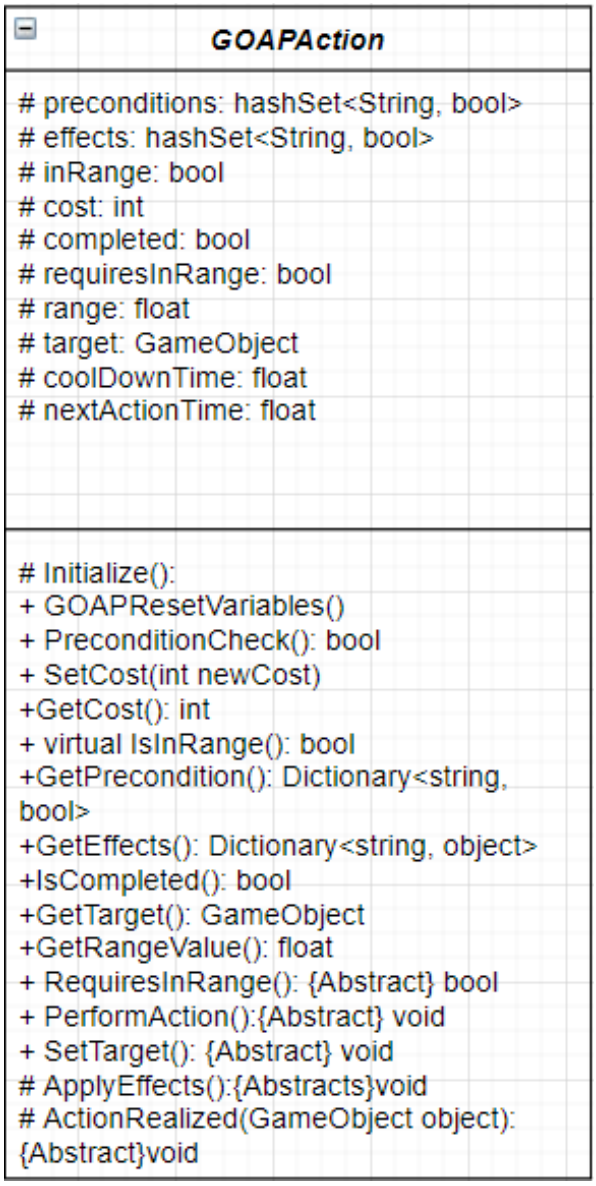

Ilustración 23. Diagrama UML detallado de la clase GOAPAction

Las acciones son probablemente la parte más importante del sistema GOAP y, a la vez, la parte más fácil de implementar gracias a la forma en la que está diseñado el sistema. Disponemos de una clase abstracta que va a servir de plantilla para todas las acciones que se van a diseñar, proporcionándoles de esta manera los métodos y variables necesarios para el correcto funcionamiento del planificador y de la máquina de estados. Las acciones van a tener un diccionario de precondiciones y uno de efectos, ambos tienen como clave una cadena de

caracteres y como valor un booleano que nos permite identificar con mucha facilidad si se cumplen o no las precondiciones; detalle que es necesario porque el sistema GOAP carece de borrado de hechos. Las acciones también dispondrán de valores como el rango necesario para realizar la acción, el coste, el objetivo y un tiempo de espera entre una acción y otra. La creación de esta clase abstracta facilita mucho el diseño de las acciones, puesto que solo se tienen que completar los métodos abstractos para que estas se incorporen de forma satisfactoria en el planificador, sin que el diseñador se preocupe por la implementación de las acciones que ejecutará el agente.

También tendrán un método que definirá la realización de estas en tiempo real en el entorno de Unity. Este método conectará toda la parte interna de la máquina de estados, el planificador, las precondiciones y los efectos con el mundo real; es decir, lo que ve el usuario en la pantalla.

Es importante recordar que las acciones tienen dos estados fundamentales cuando se van a ejecutar: Acción realizada y Acción completada. El primero define si la acción se ejecuta de forma correcta dentro de la escena de Unity; por ejemplo, en el caso de que un soldado quiera atacar a un enemigo la acción se marca como realizada cuando le hace daño a este, si el enemigo es eliminado antes de poder hacerle daño; por ejemplo, si otro agente lo elimina primero, entonces la acción se marca como no realizada y se procede a calcular un nuevo plan. Acción completada define si una acción se ejecutó hasta lograr llegar al objetivo de la misma; usando el ejemplo anterior, la acción de atacar se marca como completada cuando el enemigo es eliminado; es decir, que no solo fue dañado sino que el daño recibido ocasionó su eliminación, por ende, el efecto establecido en la acción se consolida y se puede proceder a la siguiente acción del plan; en caso de que la acción no este marcada como completada, se vuelve a ejecutar; siguiendo con el ejemplo utilizado, si el soldado ataca al enemigo y no lo elimina con esa acción, puesto que tiene más salud que la que le resta un ataque del soldado, esta se vuelve a ejecutar hasta que la salud del zombi quede reducida a cero.

A continuación, analizaremos a fondo las acciones que tienen disponibles los agentes:

AttackAction: Esta es la acción que realiza el agente cuando ataca a distancia a los enemigos, para poder realizar esta, necesita tener munición suficiente para eliminar al enemigo marcado como target de la acción y tener a este último en el rango de alcance del arma. Esta acción se marca como realizada cuando le hace daño al enemigo y como completada cuando el enemigo es eliminado.

GrenadeAttackAction: Esta se puede realizar cuando el agente soldado tiene munición de granada disponible y tiene al menos un enemigo en su rango, antes de ejecutar la acción va a avisar con un mensaje a todos los demás soldados que va a lanzar una granada, de tal forma que si los demás agentes habían decidido lanzar una también se les bloquea esta opción, así evitamos que lancen más de una granada por vez y se desperdicie munición. El costo de esta acción disminuye entre más enemigos tenga en su rango de lanzamiento, de tal forma, que si el agente tiene 50 enemigos al frente es más posible que elija esta acción antes que atacar con el fusil. Esta acción se marca como realizada y completada en el momento en el que se lanza la granada sin necesidad de alguna comprobación extra.

MeleeAttackAction: Esta es la última acción de ataque que tienen los soldados, tiene como precondición únicamente que el agente tenga enemigos en su rango de visión. Para el cálculo del coste de la acción se utiliza la distancia entre la unidad y el enemigo objetivo, al igual que la cantidad de munición del soldado, la cantidad de enemigos y el monto de vida del agente. Este ataque inflige 50 puntos de daño, equivalente al 50% de la vida de los enemigos. Esta acción se marca como realizada cuando le hace daño al enemigo objetivo y como completada cuando el mismo es eliminado.

SelfHealAction: A diferencia de las otras acciones, esta se centra en la recuperación de puntos de vida del agente, tiene como precondición la necesidad de que la unidad tenga menos puntos de vida que un valor umbral determinado; el cálculo de su coste dependerá de la diferencia entre la vida del agente y el umbral mínimo salud, también dependerá de la cantidad enemigos próximos, en caso de haberlos. Esta acción se marca como realizada cuando el agente logra recuperar una porción de vida y como completada cuando el agente tiene la vida al 100%.

RetreatAction: Esta acción se ejecuta cuando el agente está muy cerca de una gran cantidad de enemigos, no tiene munición suficiente para todos y, además está herido; en un caso como este el agente se moverá hacia el Centro de mando (HQ). La única precondición necesaria para que se ejecute esta acción es que el agente se encuentre herido y con la salud por debajo del umbral mínimo. El costo de esta acción se calculará en base a la cantidad de enemigos en el rango del jugador, la distancia que los separa, la cantidad de munición y la vida disponible. Esta acción se marcará como realizada y completada en el momento en el que el agente llegue al punto pautado para la retirada.

RepairAction: Para que se pueda considerar esta acción dentro del planificador es necesario que alguna estructura esté dañada. Todas las estructuras tienen la posibilidad de

comunicarse con los agentes y señalar que tienen los puntos de salud por debajo del umbral establecido para ellas. La acción se marca como realizada cuando el agente logra recuperar una porción de la vida de las estructuras y como finalizada cuando la salud de la estructura está al 100%.

ReloadAction: Esta acción necesita que el soldado tenga menos munición que el umbral determinado. El costo de esta acción se calcula a partir de la diferencia entre la munición disponible y el umbral, la cantidad de enemigos en el rango de vista del agente y la cantidad de vida del mismo. La acción se marcará como realizada y completada cuando el soldado tenga su munición recuperada en su totalidad.

HealRequestAction: Esta es una de las pocas acciones enfocada solo en la comunicación con otros agentes, se puede realizar si un soldado se encuentra herido con la vida por debajo del umbral de salud determinado. El agente herido envía un mensaje a todos los demás soldados comunicando que necesita ayuda y provocando que el WorldState de los otros agentes tenga un hecho de que un aliado necesita curación. Una vez enviado el mensaje, la acción se marca como realizada y completada. El cálculo de costes de esta acción involucra la distancia del agente con los enemigos y la distancia del soldado con agentes aliados.

HealOtherUnitsAction: Esta funciona como continuación de HealRequest, tiene como requisito que el soldado que va a realizar la acción haya recibido un mensaje de solicitud de curación. Los detalles de esta acción son muy parecidos a los de SelfHealAction pero con una unidad objetivo distinta a la unidad que realiza la acción. El cálculo del costo de esta acción tiene como principales componentes la distancia entre el soldado que necesita ayuda y el soldado que ejecuta la acción, y la distancia del agente enemigo más próximo.

Nombre de la acción	Precondición	Efecto	Cálculo del coste variable
AttackAction	EnemyDetected = true HasAmmo = true	EnemyDetected = false	Costo = (shootingRange – distanceToTarget) + (targetHealth/ ammo)
GrenadeAttackAction	EnemyDetected = true HasGrenade = true	EnemyDetected = false	Costo = maxEnemiesToKill – enemiesInSight
MeleeAttackAction	EnemyDetected = true	EnemyDetected = false	Costo = (targetHealth / soldierHealth) + (ammo / meleeDamage)
SelfHealAction	AgentHurt = true	AgentHurt = false	Costo = (health / healthThreshold) + enemiesInSight
RetreatAction	EnemyDetected = true AgentHurt = true	Retreating = true	Costo = (health / healthThreshold) + (ammo / enemiesInSight)
RepairAction	StructureNeedsRepair = true	StructureNeedsRepair = false	Costo = enemiesInSight + distanceToStructure
ReloadAction	HasAmmo = false	HasAmmo = true	Costo = (ammo / ammoThreshold) + enemiesInSight + (shootingRange – distanceToTarget)
HealRequestAction	AgentHurt = true	AgentHurt = false AllyNeedsHealing = true (en otros agentes)	Costo = (health / healthThreshold) + (closestEnemyDistance / closestAllyDistnace)
HealOtherUnitsAction	AllyNeedsHealing = true	AllyNeedsHealing = false	Costo = (distanceToTarget / distanceToEnemy)

4.5.4 Agente

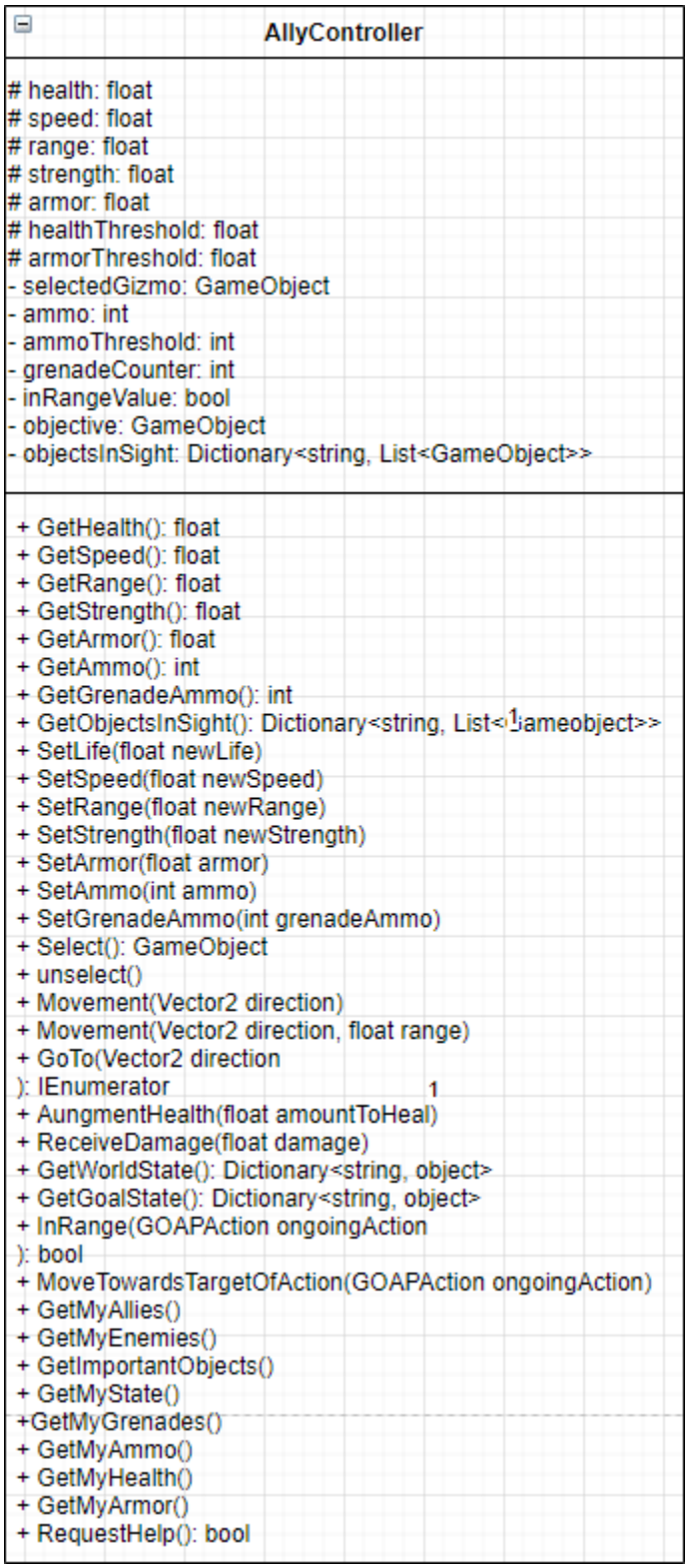

Ilustración 24. Diagrama UML detallado de la clase AllyController

Los agentes representan a las unidades soldado del juego, tendrán a su disposición una serie de métodos que les permitirán desplazarse a los sitios requeridos por el estado GOTO del FSM, y observar el mundo a su alrededor; para esto último van a utilizar un componente del motor Unity llamado CircleCollider2D que representa su “rango de visión”; todo componente que entre en este rango será analizado y categorizado como aliado, enemigo u objeto de interés. También poseen métodos de propiocepción que les permiten conocer su nivel de salud, armadura y munición. Todos estos datos se agregan a un diccionario de la clase WorldState, de la que se hablará en el siguiente apartado.

4.5.5 WorldState

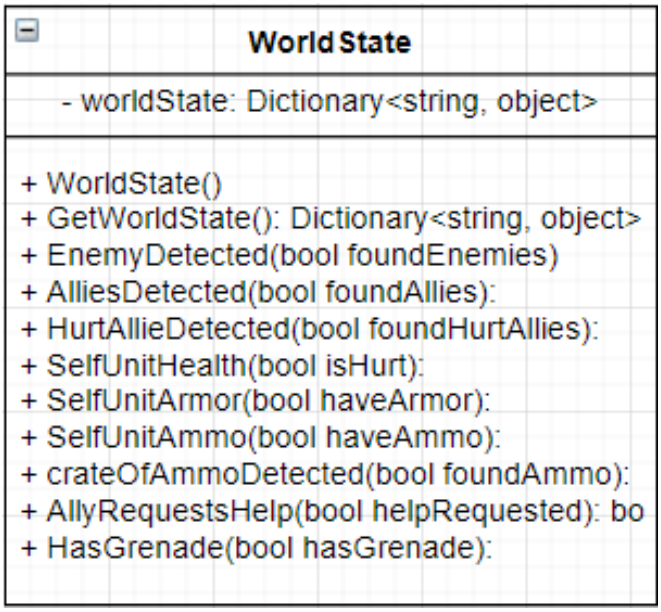

Ilustración 25. Diagrama UML detallado de la clase WorldState

Cada uno de los agentes activos en el juego tiene un WorldState personal compuesto por los elementos visibles de cada unidad, de los que ya hablamos en un punto anterior. Esta clase está diseñada con dos propósitos: el primero, hacer más limpio el código de los agentes teniendo toda la información del mundo que los rodea, y la forma de agregar esa información separada en una clase distinta; el segundo es unificar las claves de las precondiciones que van a utilizar los agentes y las acciones, encapsulando estas cadenas de caracteres tras métodos con menos posibilidad de equivocación. De esta forma, si dos tipos de unidades ven a un enemigo, en vez

de agregar al WorldState la clave EnemyDetected y exponerse a que haya una equivocación y se escriban dos claves diferentes para los dos tipos de agente, se encapsula esta tras un método llamado EnemyDetected, que agrega la misma cadena todas las veces con el valor booleano que se le proporcione como parámetro. Los métodos de esta clase verifican si existe la clase en el diccionario y, en caso de que así sea, se modifica la clave ya existente en vez de agregarla de nuevo, evitando de esta manera excepciones en la ejecución del código.

4.6 Aplicación del algoritmo BOID en Unity

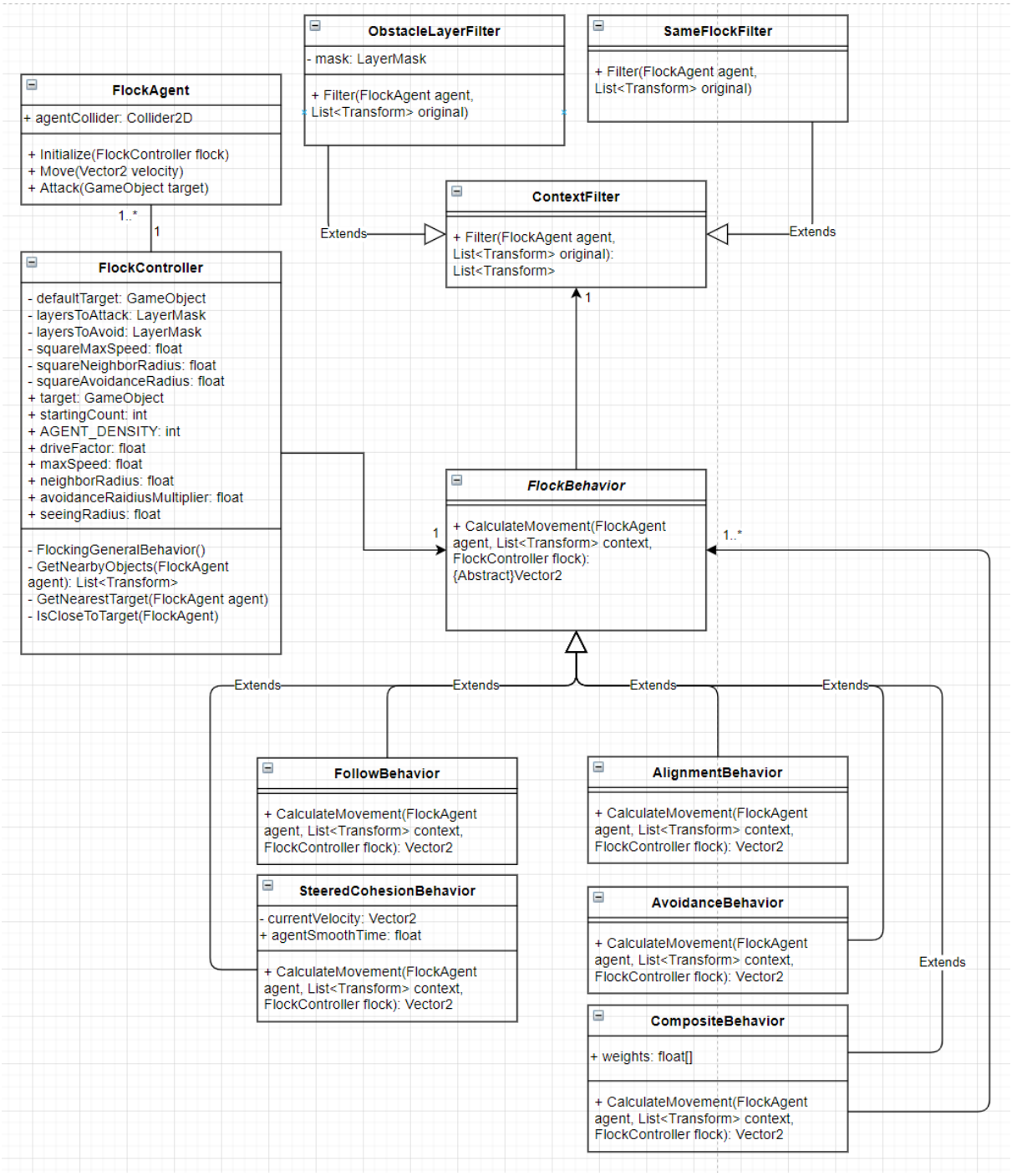

Ilustración 26. Diagrama de clases UML del módulo BOID

Como se mencionó antes, el algoritmo BOID es un modelo con el que se puede replicar el comportamiento de grupos de animales, como bandadas de aves o cardúmenes. Para esto se utilizan tres clases de comportamientos principales: separación, alineamiento y cohesión; y se utiliza un ente que enlace estos comportamientos y les asigne valores de prioridad.

La implementación que utilizamos en Unity emplea un objeto Flock que actúa de objeto padre de los agentes; vale la pena señalar que este no ejercerá de coordinador de los agentes, puesto que eso iría en contra del principio fundamental del algoritmo. Este objeto y su correspondiente clase FlockController, servirán como iteradores que irán uno a uno por cada agente Flock e invocaran su método CalculateMovement, que será el encargado de realizar los cálculos necesarios para el movimiento de su respectivo agente; también realizará una detección de los objetos cercanos a cada agente y añadirá estos a una lista a la que llamaremos Context, que servirá de contexto para el cálculo de los comportamientos.

Cada uno de los agentes dispondrá de una serie de clases, derivadas de una clase padre llamada FlockBehavior, que representan cada uno de los comportamientos posibles: AlignmentBehavior, AvoidanceBehavior y SteeredCohesionBehavior; también tendremos FlockAgent, CompositeBehavior y FollowBehavior como clases extra de las que también vamos a hablar más en detalle.

Todos los comportamientos dependen de la detección que se realiza de los objetos cercanos a cada agente; sin embargo, como la detección inicial de estos objetos no distingue entre los tipos de GameObjects dentro del rango, en la lista se pueden encontrar objetos con los que no queremos que el agente interactúe, como el plano que representa el suelo o los adornos del terreno que no tienen ninguna utilidad especial. Para evitar problemas y aligerar la carga de trabajo de los agentes, creamos la clase abstracta ContextFilter que hereda de la clase ScriptableObject; esta clase tendrá un método Filter, cuyo propósito es recibir la lista original sin filtrar de los objetos cercanos al agente y devolver la lista solo con los objetos que cumplan las características del filtro.

El comportamiento de alineación se consigue con la clase AlignmentBehavior, ésta utiliza la lista Context para identificar a todos los agentes que se encuentran a su alrededor, y, posteriormente, sumar todos los vectores dirección de estos agentes en una variable Vector2; una vez hecho esto, calcula la media del vector suma, la que corresponde con la dirección media a donde está apuntando el grupo, y la retorna.

La cohesión de la horda se obtiene empleando la clase SteeredCohesionBehavior, que a partir de la misma lista Context, una vez filtrada para obtener solo a los agentes del grupo, calcula la media de la posición de los demás agentes, sumando todas las posiciones de estos y dividiéndola entre el número de agentes a calcular; luego de conseguida la media, se calcula el vector dirección que necesita seguir el agente para llegar a la posición media de su grupo y se suaviza este vector dirección usando el método SmoothDamp de la clase Vector2 de Unity; este se utiliza para evitar movimientos bruscos y poco convincentes cuando el agente intente corregir su posición.

El último de los comportamientos base del modelo BOID es separación o evitación; para conseguirlo se utilizó la clase AvoidanceBehavior. Como las otras dos clases, esta hace uso de la lista Context filtrada para obtener aquellos objetos que se desea evitar; para cada objeto de la lista se calcula el cuadrado de la magnitud de la distancia entre el agente y el objeto y, en caso de que sea menor que el cuadrado de la distancia máxima de separación del agente, es decir que el objeto se encuentre más cerca del agente de lo que deseamos, se realizan dos sencillos pasos; primero, se aumenta en una unidad un contador de objetos para esquivar y luego se suma a un vector bidimensional el vector distancia entre el agente y el objeto. Con estos cálculos, realizados para todos los objetos que se desean esquivar, se promedia el vector donde se sumaron todas las distancias y se obtiene el vector resultante que será el que usará el agente para evitar los obstáculos.

Por ahora, nuestros agentes enemigos pueden permanecer juntos, moverse en la misma dirección y evitar chocar entre ellos y con su entorno; sin embargo, todavía no conseguimos que los enemigos tengan un objetivo definido y que se muevan en la dirección de este, para ello creamos la clase FollowBehavior. Esta clase filtra la lista Context para obtener solo objetos de tipo Estructure y Ally, consigue de entre estos el más cercano y lo asigna como objetivo principal; en caso de no conseguir ningún objeto de estos tipos en el rango de detección, el objetivo por defecto es el Centro de mando del jugador. Posteriormente a esta detección y filtrado, se calcula el vector dirección entre el agente y la estructura, y se retorna este.

Hasta el momento, tenemos cuatro clases distintas de comportamientos y, como se explicó previamente, algunos de estos van a intentar mover al agente en direcciones contrarias; por este motivo necesitamos una clase que fusione por prioridades a estas clases, obtenga un vector combinado de movimiento y se lo retorne a FlockController para su aplicación en Unity. Para realizar esta fusión hemos utilizado la clase CompositeBehavior, que recibe en una lista de

tipo FlockBehavior todos los comportamientos que se van a aplicar al agente y, en un array de floats, todos los pesos o prioridades que tendrán los comportamientos. A partir de la lista y del array se realizarán los siguientes cálculos: la obtención del vector movimiento de cada uno de los comportamientos y la multiplicación de estos por los pesos correspondientes. En el caso de que el cuadrado de la magnitud de este vector sea mayor que el cuadrado del peso correspondiente al comportamiento, se realizara una normalización del vector usando el método Normalize de Unity, y al vector normalizado se le multiplicara nuevamente el peso del comportamiento que le corresponde. Finalmente, se sumarán todos estos vectores en un Vector2 llamado CompositeMovement y éste se retorna a FlockController para que se le aplique al agente y él realice el movimiento.

Es importante destacar que, tanto para nuestro proyecto, como para las futuras ampliaciones de este, los agentes van a necesitar aplicar un mismo comportamiento a distintos tipos de objetos y con diferentes prioridades. En el caso de este trabajo, los agentes tienen dos comportamientos diferentes de tipo AvoidanceBehavior, puesto que deben evitar obstáculos a la vez que evitan a otros agentes. Para conseguir esto sin sacrificar la modularidad y limpieza del código, se empleó para los comportamientos una estructura de datos propia de Unity llamada ScriptableObject; de esta forma, con un solo script de Unity, representando la clase AvoidanceBehavior, podemos crear dos ScriptableObjects del mismo tipo, asignarles filtros distintos para que uno de los objetos represente el comportamiento de esquivar otros agentes y el otro represente el de esquivar obstáculos; y luego introducirlos en CompositeMovement con dos valores de prioridad diferentes. Gracias a esto también se reduce la reutilización de código, puesto que al diseñar estas clases como subclases de ScriptableObjects a cada instancia se le pueden asignar valores de variables distintos desde el inspector de Unity.

Cuando se combinan todas estas clases, el resultado es una simulación de movimiento con apariencia “salvaje”, con agentes que a veces muestran comportamientos independientes y aleatorios, pero con un patrón grupal fácilmente distinguible, obteniendo así, esa sensación de horda zombi de las películas o juegos.

4.7 Desarrollo detallado en Unity

Para facilitar el análisis del desarrollo realizado dentro del motor de juegos Unity, hemos decidido dividir esta sección del trabajo en los diferentes elementos técnicos presentes en la carpeta del proyecto Assets; así, podemos analizar cada tipo de elemento, cómo y porqué se creó y qué valor tiene para la composición final.

4.7.1 Scenes

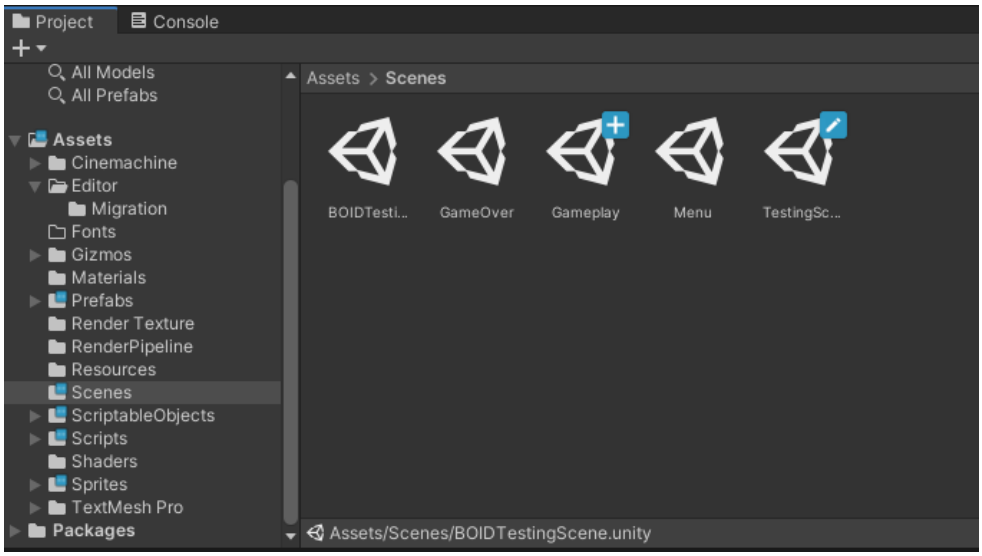

Ilustración 27. Captura del motor Unity de la carpeta Scenes

Las escenas en Unity son las distintas secciones interactuables de un juego. Para este trabajo empleamos tres escenas; la primera corresponde al menú, que permite la interacción con una serie de botones para ver los créditos, modificar opciones del juego e iniciar el mismo; la escena de jugabilidad llamada GameplayScene es aquella donde se desarrollan todas las mecánicas del juego, interactúan los sistemas inteligentes creados para el proyecto y donde el jugador pasará la mayor parte del tiempo. Por último, tenemos la escena final que mostrará una pantalla de victoria o derrota dependiendo de los resultados obtenidos en la escena anterior.

Las escenas están compuestas en su totalidad por GameObjects, que son objetos con propiedades físicas dentro del motor y a los que se les pueden agregar componentes; cómo por ejemplo, SpriteRenderers, Colliders y también scripts especificando comportamientos no preestablecidos en Unity. Estos GameObjets están todos organizados en una jerarquía que permite el anidamiento de objetos dentro de objetos, para agruparlos por funcionalidades o por cualquier característica que el diseñador vea necesaria.

4.7.2 Prefabs

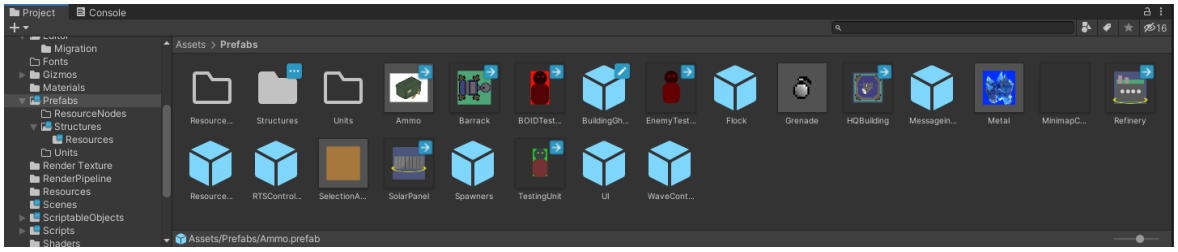

Ilustración 28. Captura del motor Unity de la carpeta Prefabs

Cuando el diseñador decide que un GameObject es muy importante y va a ser instanciado varias veces a lo largo del juego, o va a ser necesario en varias escenas, puede convertir este objeto en un objeto Prefab (un término para acortar Prefabricated), al hacer esto todos los componentes, características y valores que tiene el objeto se guardan en una carpeta con el nombre Prefabs y el diseñador puede agregar manualmente, o por código, con más facilidad cualquier número de copias independientes que necesite.

En nuestro caso, la carpeta Prefabs contiene los siguientes objetos:

RTSController: Este es el objeto en el que recaen todas las funciones y clases necesarias para que el juego se comporte como un RTS, este Prefab tiene la clase RTSController que, como ya hemos explicado, es la encargada de registrar todos los inputs del jugador durante el juego, los clics para seleccionar las unidades, las teclas y los movimientos del ratón para mover la cámara y hacer zoom en el terreno de juego, entre otras muchas acciones.

SelectionArea: Este es un objeto sencillo que es controlado por RTSController y permite visualizar el campo de selección de las unidades por el jugador; es básicamente el rectángulo que aparece cuando el usuario clica y arrastra el ratón sobre el terreno.

ResourceManager: Contiene la clase ResourceManagerController, que tiene como labor controlar los recursos del jugador durante la partida con acciones como aumentar o quitar recursos, cuando el jugador los obtenga o los gaste construyendo estructuras.

BuildingGhost: Es un objeto que muestra la estructura que el jugador va a construir en una zona del mapa para que este pueda planificar dónde ubicarla, también funciona como una pista visual para que el jugador sepa de forma intuitiva si tiene permitido construir un edificio en una zona determinada o no.

UI: Es un objeto que contiene toda la interfaz de usuario necesaria durante la escena de juego, como el cronómetro, el minimapa, la pantalla de recursos del jugador o la botonera de acciones de cada unidad y estructura. Las clases que tiene el Prefab UI están encargadas de extraer la información de otras clases como ResourceManagerController y hacerla visible en la interfaz para el jugador.

WaveController: Es un objeto con el que controlamos el temporizador del juego y a la vez cambiamos entre las distintas fases del gameplay.

Por último, tenemos todos los objetos necesarios para el gameplay del juego como estructuras, unidades, enemigos, nodos de recursos, etc. Cada uno de estos tiene su propio Prefab que nos permite instanciar el número de objetos que la computadora pueda manejar durante la ejecución del juego.

4.7.3 ScriptableObjects

Los ScriptableObjects son contenedores de datos dentro de Unity que permiten a los desarrolladores almacenar grandes cantidades de información, sin tener que crear GameObjects dentro de la escena del juego para cada uno de los grupos de datos que se van a almacenar. Esto permite a los desarrolladores crear clases para almacenar datos cuando se van a acceder a estos de forma continuada, o cuando se necesitan crear distintos objetos con el mismo tipo de datos.

Para el desarrollo de nuestro juego estos ScriptableObjects fueron muy útiles, puesto que facilitaron la creación de clases genéricas para los distintos tipos de recursos y estructuras que el jugador puede recolectar y crear, permitieron también realizar la estructura modular de algunas

partes de la interfaz de usuario; como por ejemplo, los botones de construcción que dependen de los edificios disponibles y la pantalla en la que aparecen los recursos que se diseñaron para el juego.

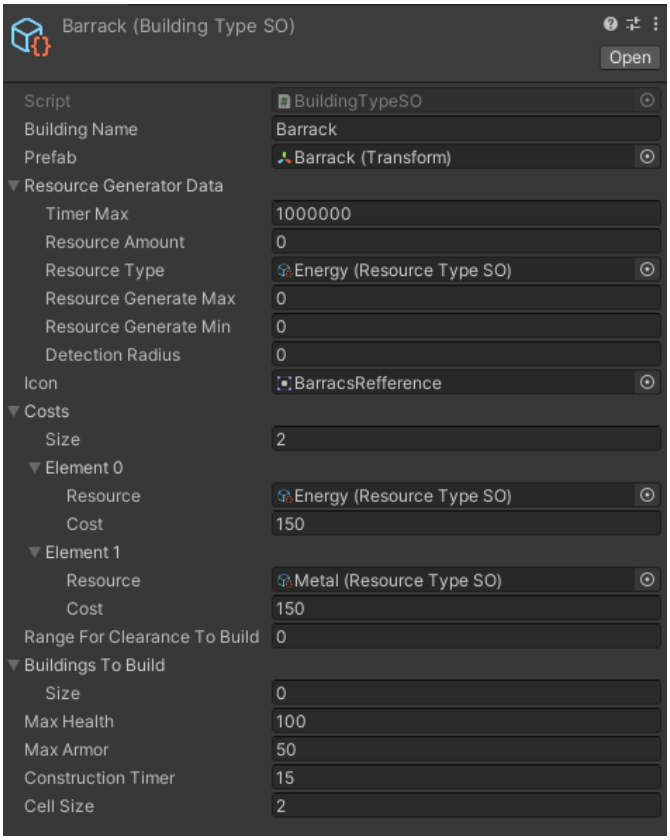

Ilustración 29. Captura del motor Unity del ScriptableObject Barrack

Los ScriptableObjects utilizados en el trabajo son:

ResourceTypeSO: Permite almacenar un nombre y un icono. Con este se crearon los recursos de Metal y Energy que necesita el jugador para construir. Cada vez que el jugador recolecta o gasta un recurso de alguno de estos tipos, se llaman a estos ScriptableObjects para obtener sus datos.

BuildingTypeSO: Éste almacena el nombre, el objeto Prefab, el recurso que produce la estructura de recolección y el tiempo que tarda en hacerlo, el icono para la botonera y el costo

que hay que pagar para construirlo. Cada uno de los edificios que se usan en el juego pertenecen a este tipo de ScriptableObjet.

UnitTypeListSO: Se usa para almacenar el objeto Prefab, el nombre y los recursos que cuesta construir una unidad. Para el alcance del proyecto solo se creó una unidad con este tipo de ScriptableObject: la unidad Soldier, sin embargo, quedan las bases asentadas para ampliar el juego y crear con mucha facilidad más tipos de unidades.

4.7.4 Cinemachine

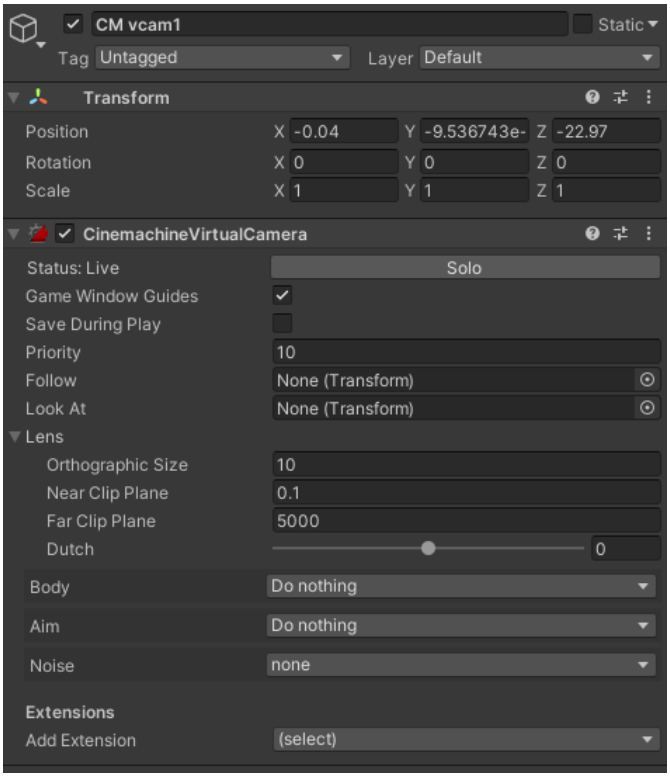

Ilustración 30. Captura del motor Unity de Cinemachine

Aunque Unity tiene entre sus elementos bases un sistema de cámaras muy útil, estos tienen limitaciones cuando se intentan realizar algunas acciones o es necesario crear clases para poder aplicar ciertas modificaciones a la cámara. Para evitar complicar la estructura del juego decidimos optar por un plugin que amplía cuantiosamente las opciones para la cámara y facilitará el movimiento de la misma usando métodos de la clase RTSController.

4.7.5 Scripts

Ilustración 31. Captura del motor Unity de la carpeta Scripts

Los scripts de Unity representan la parte más compleja e importante de todo el trabajo, pues es en la carpeta scripts donde se ubican todas las clases pertenecientes a la arquitectura del juego previamente descrita; las que se encargan del funcionamiento del mismo.

4.7.6 Otros

Las demás carpetas dentro de Assets almacenan archivos con finalidades muy variadas e importantes, pero su explicación es bastante más sencilla. La carpeta Sprites es donde se almacenan todos los recursos visuales necesarios para el juego, las imágenes de los soldados, enemigos, estructuras, la interfaz, botones y el terreno del juego, entre otros. Fonts es la carpeta donde se almacenan las fuentes tipográficas utilizadas. Y RenderTexture y Shaders son las carpetas donde almacenamos una serie de clases que describen cómo se realiza la renderización de algunos elementos dentro del juego; permitiendo así, disponer de efectos visuales más complejos, como el que mostramos cuando se está construyendo un edificio.

Tres meses de trabajo desarrollando modelos de inteligencia artificial y mecánicas de juegos de estrategia en tiempo real, combinado con arte propio, interfaces intuitivas y dinámicas al igual que otros elementos de diseño; ensamblado todo dentro del motor gráfico Unity, culminaron en la producción del juego Galactic Defense.

Capítulo 5

Resultados

A continuación, presentaremos una serie de capturas con la finalidad de mostrar el prototipo desarrollado, sus mecánicas y características visuales.

En la Imagen 1 podemos apreciar la distribución de la interfaz de usuario usada en el juego; en la parte superior tenemos el temporizador que muestra cuanto tiempo le queda al jugador para completar sus defensas antes de que empiece el ataque de la horda zombi. En la parte inferior se muestra la información necesaria para realizar acciones, administrar recursos y visualizar el mapa; en la esquina inferior izquierda tenemos los recursos disponibles en las arcas del jugador, al igual que la cantidad de recursos que se generan por segundo; en la esquina inferior derecha mostramos el minimapa, que nos permite ver una versión simplificada de las estructuras y soldados; y por último, en la parte inferior central podemos ver la información perteneciente a la unidad seleccionada en el momento, las acciones que se pueden realizar con esta, la vida y la armadura.

Imagen 1. Captura del juego mostrando la interfaz

En las siguientes imágenes podemos ver en funcionamiento las mecánicas de construcción de estructuras; en la imagen 2 mostramos la cuadricula de organización sobre la que se van a construir las distintas edificaciones; también podemos apreciar, en color verde, el sprite de la estructura a construir, marcando la zona sobre la cual se va a realizar la acción correcta. Podemos notar como las reglas de construcción previamente especificadas se están aplicando en las imágenes 3, 4 y 5; en ellas se ve como el jugador no tiene permitido construir encima de un nodo de recursos, encima de otra estructura o cerca de una estructura del mismo tipo ya existente. Para diferenciarlas de la correcta, el sprite se muestra en color rojo.

Imagen 2. Captura del juego mostrando una construcción correcta

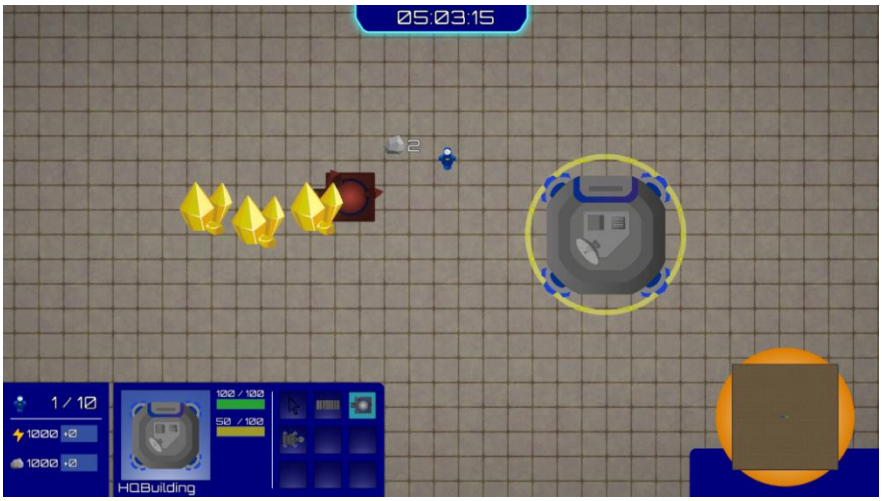

Imagen 3. Captura del juego mostrando una construcción incorrecta sobre un recurso

Imagen 4. Captura del juego mostrando una construcción incorrecta sobre otra estructura

Imagen 5. Captura del juego mostrando una construcción incorrecta cerca de otra estructura

En la imagen 6, podemos apreciar como un par de soldados están defendiendo el Centro de mandos de una pequeña horda zombi, se ven una serie de cambios en la interfaz con respecto a otras imágenes: en primer lugar, notamos que ya no hay un temporizador dado que no es necesario contabilizar el tiempo, pues la fase en la que se encuentra el juego terminara cuando los soldados eliminen a todos los zombis; también podemos notar que, como no hay ninguna unidad seleccionada, no se muestra la información de ningún soldado o estructura. Por último, podemos ver que la acción que están desarrollando los agentes es la de atacar a distancia debido a que tienen suficiente munición y los zombis están lejos.

Imagen 6. Captura del juego mostrando dos soldados atacando a distancia

En la Imagen 7 podemos ver a un soldado atacando cuerpo a cuerpo a un agente zombi, el soldado tomó la decisión de realizar esta acción puesto que no tenía munición y estaba cerca de estos por lo que recargar la munición le llevaría a la muerte.

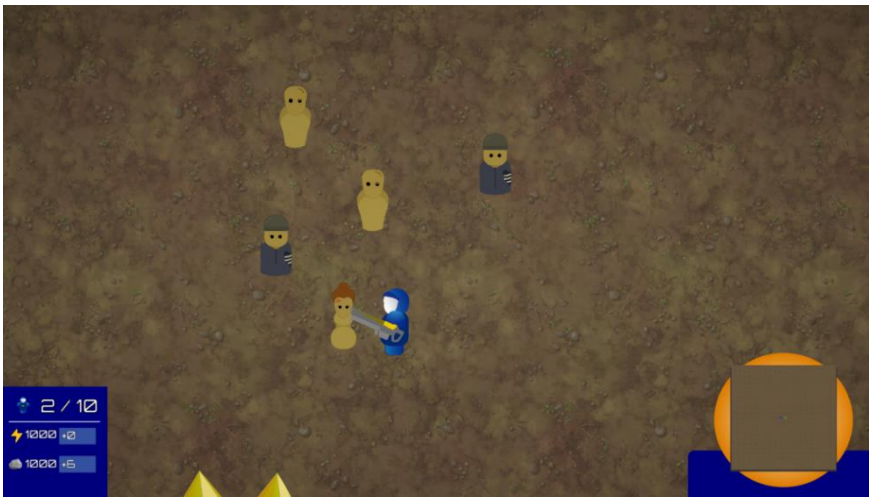

Imagen 7. Captura del juego mostrando un soldado atacando cuerpo a cuerpo

En la imagen 8 podemos ver a un soldado realizando otra de las acciones disponibles, en este caso se está curando, la vida de la que dispone es inferior al valor programado como límite inferior antes de que considere necesario y prioritario recuperarse. Nótense las cruces verdes que simbolizan la acción.

Imagen 8. Captura del juego mostrando un soldado recuperando salud

Por último, podemos apreciar un soldado sin zombis en su rango de visión y sin acciones pendientes por realizar (Imagen 9), lo que significa que se encuentra en el estado Idle del FSM y, por lo tanto, está realizando la acción de patrullaje que le indico el jugador en la fase previa.

Imagen 9. Captura del juego mostrando un soldado patrullando entre dos puntos

Capítulo 6

Conclusiones

El desarrollo de este trabajo fue, sin lugar a dudas, más complejo de lo que se planteó inicialmente; es un proyecto que, desde su concepción, plantea un número de retos difíciles que solo fueron desvelando su complejidad a medida que iba avanzando la realización del mismo. El primero de esos retos fue la creación de un juego de un género que requiere de mucho análisis y balanceo para que no deje de ser entretenido y pierda el interés del usuario con el tiempo. El segundo desafío, fue crear un sistema de inteligencia artificial partiendo de unos conocimientos con los que no estábamos muy familiarizados, pues una parte de estos no están incluidos en la carga académica de nuestra carrera; son propios de los modelos de sistemas inteligentes utilizados en la industria de videojuegos.

A pesar de estas dificultades, consideramos que la mayoría de los objetivos planteados al inicio de este trabajo, y que están expuestos en el comienzo de la memoria, fueron alcanzados. Logramos desarrollar el sistema inteligente que controla las unidades aliadas y las enemigas, de tal forma que la escalabilidad de las acciones que estas unidades pueden desarrollar es sencilla y alcanzar todos los comportamientos que se pueden conseguir no requiere de muchas horas de planificación. Se logró crear un prototipo funcional y entretenido, aunque no final del juego planteado; con menú, juego y condiciones de derrota y victoria. Se consiguió también ampliar el catálogo de géneros de juego y herramientas con las que se pueden trabajar de ahora en adelante, al igual que crear un sistema inteligente como el GOAP, el cual se puede extrapolar sin mucha dificultad a otros proyectos. Los objetivos no logrados fueron la optimización del sistema inteligente GOAP utilizando programación con hilos en Unity y la realización del juego completo con modos de juego, más unidades y gráficos en 3D.

Aunque se hayan completado la mayoría de los objetivos planteados, consideramos que hubo errores que podríamos haber evitado y que hubiesen permitido el logro de los demás objetivos; estos quedan ahora reseñados en el apartado destinado a las futuras ampliaciones que puede tener el proyecto. Los principales errores recayeron en la planificación y en la conceptualización. El propósito inicial del proyecto, fue crear un juego que mezclase los conocimientos adquiridos durante la carrera en la rama de computación y los conocimientos

existentes en el ámbito del desarrollo de videojuegos. La falta de entendimiento de los sistemas inteligentes que podían aplicarse en Unity sobre el marco de tiempo planteado, ocasionó demoras en la conceptualización final del tema del juego. Llegamos a considerar que sería fácil crear un juego que utilizará el lenguaje PDDL para un sistema de planificación que funcionara como “storyteller” procedural, lenguaje que no es aceptado por Unity y para el que no existe un parser en C#.

6.1 Relación del trabajo con los estudios cursados

A lo largo del trabajo hemos utilizado una gran cantidad de las herramientas adquiridas durante los cuatro años de la carrera de informática; estas nos ayudaron a crear un juego, no solamente funcional y eficiente, sino también estéticamente correcto y con algoritmos complejos que mejoran enormemente la calidad del producto final. Para resaltar lo útiles que han sido los estudios cursados vamos a dar una pequeña explicación sobre aquellos conocimientos que sentimos fueron los más relevantes a lo largo del proyecto.

Uno de los conocimientos que nos resultó más útil fue el paradigma de Programación Orientada a Objetos (POO), que utilizamos en todos los scripts creados en Unity. La POO es el paradigma principal para la creación de videojuegos. Aunque el lenguaje C# no haya formado parte de la formación profesional recibida en la universidad, no ha sido difícil aprender de forma autodidáctica las reglas y convenciones de este, puesto que no varían mucho de los otros lenguajes del mismo paradigma que nos enseñaron en materias como Introducción a la informática y la programación o Programación.

También nos fueron muy útiles los conocimientos que nos impartieron en la asignatura Estructura de datos y algoritmos, donde aprendimos el uso de listas, diccionarios, pilas y colas que fueron de mucha ayuda para la creación de la máquina finita de estados que utilizó el sistema GOAP. El desarrollo de este sistema se benefició también de la información recibida en Teoría de autómatas y lenguajes formales, donde aprendimos acerca del funcionamiento de las máquinas finitas de estados con sus nodos y transiciones.

Queremos destacar también los aportes que tomamos de las asignaturas Agentes inteligentes; Técnicas, entornos y aplicaciones de inteligencia artificial y Sistemas inteligentes;

pues estos nos sirvieron de base para el entendimiento del sistema de planificación fundamentado en STRIPS, que utiliza el sistema GOAP; lo que aprendimos en estas asignaturas nos permitió intuir el funcionamiento de GOAP, las acciones con efectos y precondiciones para la planificación y la búsqueda de un plan efectivo con el algoritmo A*.

Siendo Galactic Defense un software de entretenimiento con un tipo de usuario específico, fue necesario crear una interfaz clara e intuitiva que le permitiese acceder a todas las funcionalidades y entender lo que estaba pasando en la pantalla; esto requirió que aplicásemos los conocimientos adquiridos en la materia Interfaces persona computador, utilizando varias de las reglas de la Gestalt para poder generar la interfaz final que se ve en el juego.

En términos de organización, el juego no presentó una complejidad excesiva, puesto que el proyecto fue desarrollado por una sola persona; sin embargo, fueron necesarios los conocimientos adquiridos en Gestión de proyectos para poder organizar todas las tareas pertinentes para el desarrollo del juego, utilizando el sistema Kanban. En relación a la organización de las clases necesarias para el gameplay del juego, estructurar el proyecto con un modelo UML probó ser un método muy efectivo, puesto que nos facilitó la programación de las clases siguiendo las especificaciones previamente planteadas en el documento de diseño, esto no hubiese sido posible sin los conocimientos obtenidos en Ingeniería del software.

En los cuatro años de carrera y 250 créditos de estudios no podemos decir que haya habido algún área o alguna asignatura que no haya tenido cabida en este proyecto y aunque no se resaltan todas en esta memoria, el proyecto práctico las tiene todas presentes.

Capítulo 7

Futuras ampliaciones

Al juego se le pueden hacer muchas ampliaciones y mejoras; aunque las bases para la inteligencia artificial y las funcionalidades del gameplay ya están sentadas, quedan muchas áreas en las que se puede trabajar.

- El sistema GOAP tiene sus funcionalidades básicas completas, selecciona entre todas las acciones disponibles para las unidades y crea un plan para satisfacer las metas de los agentes; sin embargo, la cantidad de acciones creadas para las unidades es poca cuando se compara con las posibilidades que un jugador de cualquier RTS puede desarrollar; por lo tanto, crear más acciones para que el sistema GOAP pueda desarrollar planes diferentes para otras situaciones, podría brindarle mucho contenido y diversidad al juego.

- Por ahora el juego utiliza solamente como unidades de defensa a las torres y a los soldados. Cuando comparamos este con juegos de la talla de TABS, Age Of Empires o There Are Billions nos damos cuenta de que el número de unidades con diferentes usos y cualidades es muy superior a las dos unidades de Galactic Defense; por ello, la creación de más unidades; tanques; soldados, más fuertes pero lentos, más débiles pero rápidos o con distancia de visión más larga, puede enriquecer las estrategias adoptadas por los jugadores.

- El juego tenía concebida una historia detallada que justificaba el funcionamiento en hordas de los enemigos, y la imposibilidad de controlar a las unidades cuando venían estos; no obstante, por motivos de tiempo no se pudo implementar una narrativa detallada.

- Un último detalle que podría darle al proyecto más complejidad, elegancia y jugabilidad es la posibilidad de más modos de juego; por ahora el único modo de juego es supervivencia, donde el jugador tiene que resistir la embestida de un número determinado

de hordas; pero de contar con un modo historia, un modo infinito y un modo multijugador podría lograrse que el juego pasará de ser un proyecto universitario a un producto comercializable.

Referencias

1. Ali, Z. (2020, 02 11). *Age Of Empires Has Sold 25 Million Units, Generated Over $1 Billion In Revenue*. TheGamePost. https://thegamepost.com/2020/02/11/age-of-empires-sold-25-million-units-generated-over-1-billion-revenue/
2. Conway, C., Higley, p., & Jacopin, E. (2017, 10 09). *Goal-Oriented Action Planning: Ten Years of AI Programming* [In this 2015 GDC talk, AI Programmers Chris Conway, Peter Higley and Eric Jacopin revisit the Goal-Oriented Action Planning method of game AI programming to see how it's held up for the last 10 years, and how it influenced the AI of Middle Earth: Shadow of] [Video]. YouTube https://www.youtube.com/watch?v=gm7K68663rA
3. Hoffman, J., & Nebel, B. (2001). The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research, 14*, 253-302.
4. Larson, P. (2002, 03 18). *Command and Conquer*. STS 145. https://web.stanford.edu/group/htgg/sts145papers/plarson_2002_1.pdf
5. Lee, J. A. (2001, 03 22). *The Dune Series*. STS 145. http://web.stanford.edu/group/htgg/sts145papers/jlee_2001_2.pdf
6. Madsen, H. (2020, 10 01). *How Age of Empires Changed the RTS Genre Forever*. ScreenRant. https://screenrant.com/age-empires-rts-genre-influence-changed/
7. Nilsson, N. J. (1998). *Atificial Inteligence: A New Synthesis*. Morgan Kaufmann Publishers, Inc. https://doi.org/10.1016/C2009-0-27773-7
8. Orkin, J. (n.d.). *Goal-Oriented Action Planning*. Alumni MIT Media Lab. http://alumni.media.mit.edu/~jorkin/goap.html
9. Orkin, J. (2006). Three States and a Plan: The A.I. of F.E.A.R. *Game Developers Conference*, *2006*(4). http://alumni.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf
10. Reynolds, C. (1995, 06 29). *BOIDS*. red3d. https://www.red3d.com/cwr/boids/
11. Scholz, T. M. (2019). *eSports is Buisness: Management in the world of competitive gaming*. Palgrave Macmillan. 10.1007/978-3-030-11199-1_2
12. Thompson, T. (2016, 06 10). *HTN Planning in Transformers: Fall of Cybertron* [In previous videos, we have discussed the Goal Oriented Action Planning (GOAP) framework: a game AI method adopting classical STRIPS-based planning for the F.E.A.R. trilogy. In this video, we look at the evolution of planning in games, focussing on the Tr] [Video]. YouTube. https://www.youtube.com/watch?v=kXm467TFTcY

Glosario

Real Time Strategy: Se traduce como estrategia en tiempo real, es un género de juego que se caracteriza por necesitar que el jugador desarrolle estrategias de cierta complejidad para poder progresar; y también, por que las decisiones de los contrincantes se tomen al mismo tiempo sin tener turnos definidos para cada uno de los participantes del juego.

Gameplay Loop: Análisis a gran escala de las mecánicas que se usan en el juego; usualmente los juegos dependen de la repetición ordenada de ciertas mecánicas o acciones, y aquellas que componen la jugabilidad principal del juego se consideran el gameplay loop.

Fog Of War: Componente visual característico de los juegos de RTS. Consiste en un impedimento visual que le oculta al usuario elementos de la escena de juego, para evitar que tenga toda la información disponible de forma fácil; aumenta la dificultad del juego y obliga a los jugadores a proceder con cautela.

E-Sport: Competiciones multijugador de videojuegos, destacan por tener una alta popularidad en la población contemporánea.

Comportamiento emergente: Es una conducta que no se puede reducir a las propiedades de un elemento aislado de un sistema, sino que surge cuando interactúan dos o más de ellos. El resultado es distinto a la suma de las propiedades individuales de cada uno de ellos.

Metodología Kanban: Su traducción del japones es “letrero” o “tarjeta”. Es una metodología ágil caracterizada por un tablero de tareas organizadas por columnas; estas representan las diferentes etapas por las que debe pasar una tarea antes de su finalización.

Generación Procedural: Es un método de crear datos de forma algorítmica en vez de manual. Usualmente se consigue a partir de la combinación de recursos concebidos por un desarrollador, algoritmos y aleatoriedad generada a partir de computadoras.

Motor de videojuegos: Es un framework de desarrollo diseñado específicamente para la creación de videojuegos, usualmente dispone de librerías para soportar y combinar recursos gráficos con simulaciones físicas. Su propósito fundamental es facilitar el proceso de desarrollo de videojuegos.