Producción de un Videojuego Multijugador en Unity Combinando los Géneros MOBA y RTS

Tecnología e Implementación

Diego Gálvez Ruiz

Maximiliano Miranda Esteban

Fernando Monasterio Martín

Facultad de Informática
UNIVERSIDAD COMPLUTENSE DE MADRID

Proyecto de Sistemas Informáticos

Madrid, Junio de 2014

Director: Prof. Dr. D. Federico Peinado Gil

Co-director: Prof. Dr. D. Fernando Rubio Diez

Autorización

Los abajo firmantes, matriculados en la asignatura Sistemas Informáticos de la Facultad de Informática, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, los contenidos audiovisuales incluso si incluyen imágenes de los autores, la documentación y/o el prototipo desarrollado durante el curso académico 2013-2014 bajo la dirección del Dr. Federico Peinado Gil y la codirección del Dr. Fernando Rubio Diez, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Diego Gálvez Ruiz

Maximiliano Miranda Esteban

Fernando Monasterio Martín

Dedicatoria

Queremos dedicar este proyecto a todos aquellos ingenieros e ingenieras en informática que quisieron dedicarse a la producción y desarrollo de videojuegos y que actualmente, por causas de fuerza mayor, están trabajando en algo completamente diferente.

Resumen

En los últimos años los videojuegos independientes han obtenido un crecimiento sin precedentes, tanto en número de desarrollos como en volumen de ventas dentro de la industria del ocio electrónico. Todos los análisis suelen coincidir en dos puntos clave: el crecimiento global de esta industria y el desarrollo de nuevas herramientas que simplifican en gran medida muchos de los factores que intervienen en el desarrollo de videojuegos.

El ejemplo más claro de estas herramientas son los entornos de desarrollo comerciales de videojuegos, como son CRYENGINE, Unreal Engine, o Unity. De estas alternativas, para el presente proyecto, se optó por Unity usando el lenguaje de programación C# por varias razones, las principales han sido: ofrece una curva de aprendizaje que resulta satisfactoria desde el primer momento, hay un gran volumen de documentación oficial (tanto de Unity como de C#) y la existencia de una inmensa comunidad de usuarios de esta herramienta en Internet, ofreciendo un amplio abanico de posibilidades ante cualquier problema.

De entre todas las alternativas que se estudiaron, se optó por desarrollar un videojuego que uniera los géneros RTS y MOBA haciendo uso de una jugabilidad asimétrica, esto es, uniendo las jugabilidades de ambos géneros en una misma partida pero sin mezclar mecánicas, de manera que el jugador pueda optar por jugar de una manera u otra, consiguiendo un concepto nunca visto en ningún videojuego lanzado al mercado anteriormente. El género RTS (siglas del inglés *Real Time Strategy*) o juego de estrategia en tiempo real, ha triunfado desde principios de los noventa, y el género MOBA (siglas del inglés *Multiplayer Online Battle Arena*) o campo de batalla multijugador en línea, que nació como una variación del primero, pronto se posicionó como uno de los géneros más populares. Los aspectos pertenecientes al diseño del videojuego se encuentran documentados en la memoria del mismo título con subtítulo *Ingeniería del Software y Diseño del Videojuego* (Cuesta Boluda, Cuesta Boluda, & Rodríguez-Osorio Jiménez, 2014).

Los dos modos de juego se caracterizan por poseer elementos claramente diferenciados, como el control de la cámara o de los personajes, etc. En el modo RTS se ha trabajado con el sistema de navegación y *pathfinding* de Unity y se han implementado, entre otras cosas, múltiples formas de selección de unidades, un algoritmo de elección de múltiples destinos, un complejo algoritmo de cálculo de distancias para evitar el costoso uso de físicas, y el comportamiento de las distintas unidades de cada raza y sus correspondientes edificios. Por otro lado, la implementación del MOBA se ha enfocado más en las habilidades y ataques básicos, ya que son más complejos que los de las unidades RTS.

En otros aspectos más generales, se han trabajado con *shaders* Cg/HLSL, con herramientas que ofrece Unity como el sistema de partículas y el motor de físicas. También se han utilizado herramientas que extienden Unity como NGUI para crear menús mucho más eficientes, y sobre todo Photon*,* que ofrece un servicio de gestión de partidas online, para una de las partes más importantes y complejas como es la sincronización de ambos modos de juego a través de la red.

Palabras clave: juego, C#, Photon, multijugador, online, shader, minimapa, NGUI.

Abstract

In recent years, independent games have gained unprecedented growth, both in number of developments and in sales volume within the electronic entertainment industry. All reviews usually agree in two key points: the global growth of this industry and the development of new tools that simplify many factors involved in the development of videogames.

The clearest example of these tools are the videogame commercial development environments such as CryENGINE, UDK, o Unity. From all these alternatives for this proyect we chose Unity, using the programming language C#. We did this for many reasons, the main reasons are: it offers a learning curve that is satisfying from the beginning, there is a large volume of official documentation (both Unity and C#) and the existence of a large users community from this internet tool, offering a wide range of solutions for any problems.

Of all alternatives, we chose to develop a videogame that joins the RTS and MOBA genre making use of the asymmetric gameplay, that is, joining the gameplay of both genres in one same game but without mixing the mechanics. This way the player can chose to play one way or the other creating a concept that never existed before. The RTS genre (Real Time Strategy) is successful since the early nineties and the MOBA genre (Multiplayer Online Battle Arena) was born as a variation of the first and quickly positioned itself as one of the most popular genres. The issues related to game design are documented in the report of the same title with subtitle Software Engineering and Game Design.

The two game modes are characterized by distinct features including camera, character control, etc. In the RTS mode we worked with the navigation and pathfinding system from Unity and we have implemented, among other things, multiple forms of unit selection, an algorithm to select multiple destinations, a complex algorithm to calculate distances to prevent the costly use of fisics, and the behavior of different units depending on race and their corresponding buildings. On the other hand, the implementation of MOBA was focused more into the skills and basic atacks because it is more complex compared to the RTS units.

In other general aspects, we have worked with Gg/HLSL shaders, with tools that Unity offers such as the particle system and the PhysX fisics system. We have also used some tools that extend Unity like NGUI to create more efficient menus, especially Photon, offering online management service to one of the most important and complex parts such as the synchronization of both game modes through the network.

Keywords: game, C#, Photon, multiplayer, online, shader, minimap, NGUI.

Agradecimientos

Esta es una gran oportunidad para dar las gracias a todas aquellas personas que nos han apoyado y ayudado, dándonos todo el cariño necesario para poder realizar la carrera y el proyecto.

Primeramente, queremos dar las gracias a Federico Peinado Gil y a Fernando Rubio Diez, nuestros directores del proyecto, por habernos guiado este año. A Federico, que también ha sido nuestro profesor en *Laboratorio de Programación de Sistemas* y en *Ingeniería del Software* y ha sido tanto un profesor simpático y cercano como exigente y bueno para prepararnos lo máximo para la vida laboral. A Fernando, primero por haber accedido a un proyecto que no tenía pensado hacer y segundo por su enorme esfuerzo y dedicación en el proyecto y por habernos dado la oportunidad de aprender de él, ya que si como director de proyecto es impresionante, como persona es aún mejor.

Una ayuda que ha sido imprescindible para poder desarrollar este videojuego es la de Jacobo Estévez Arias, David Vortrefflich Pérez y Fernando Marco Figueroa, estudiantes del Grado en Diseño y Desarrollo de Videojuegos de ESNE, ya que desde el principio han trabajado muy duramente en la parte artística de este proyecto. También, queremos agradecer su ayuda al final del proyecto a Luis Álvarez Castañón, Luis Suárez Álvarez y Santiago Rico Bayón, estudiantes del Grado en Diseño y Desarrollo de Videojuegos de ESNE.

También, por supuesto, queremos dar las gracias a nuestros padres, que nos han ayudado desde pequeñitos a formarnos como personas y nos han motivado y, sobre todo, dado la oportunidad de cursar esta carrera. También por haber sido nuestro apoyo en los momentos más difíciles y estresantes, ya no sólo de la vida, sino de la carrera: gracias a ellos hemos podido finalizar una carrera tan exigente como es Ingeniería en Informática.

Además, damos las gracias a nuestras parejas por habernos ayudado en los momentos más complicados de la carrera y habernos soportado en momentos tan decisivos y desquiciantes como son los exámenes de esta complicada carrera universitaria.

Para finalizar, queremos dar las gracias a la Facultad de Informática de la Universidad Complutense de Madrid, a nuestros profesores y compañeros de carrera por habernos acompañado curso tras curso; y a los trabajadores de la facultad con los que hemos compartido lugar de trabajo durante estos años, especialmente a los trabajadores de la biblioteca.

Índice General

Autorización i
Dedicatoria ii
Resumen iii
Abstract iv
Agradecimientos v
Índice General vi
Índice de Figuras viii
Índice de Fragmentos de Código xii
1. Introducción 1
   a. Motivaciones 1
   b. Interés 2
   c. Objetivos 2
   d. Composición de Grupos de Trabajo 3
2. Revisión y Estado del Arte 4
3. Diseño del Videojuego 6
      i. Ficha resumen 6
      ii. Breve descripción 6
4. Tecnologías empleadas 9
   a. Unity 9
      i. El editor 10
      ii. Arquitectura por componentes 14
      iii. El bucle de juego 15
      iv. Física 18
      v. Materiales 23
      vi. Sistemas de partículas 27
      vii. Navegación 30
   b. Scripting en C#, Monodevelop y VS 2012 38
   c. UnityVS 39
   d. NGUI 40
   e. Photon Network 42
   f. Herramientas de diseño 46
5. Análisis y diseño del proyecto 47
   a. RTS 49
      i. Selección de unidades 49

ii. Las unidades de los ejércitos 59
iii. Los edificios 79
iv. ArmyController 87
v. Algoritmo de bandada 92
vi. Navegación y Pathfinding 97
vii. Cámara RTS 99
viii. Herramienta para la medición de distancias 103
b. MOBA 110
i. Diseño software del héroe 110
ii. La cámara en tercera persona 129
c. Común 135
i. El sistema multijugador online 135
ii. Niebla de guerra 142
iii. El Minimapa del HUD 145
iv. Shaders implementados 150
v. Sistemas de partículas en NewDetroit 157
vi. Menús e Interfaz de Usuario 162
6. Discusión 164
7. Conclusiones 166
8. Referencias Bibliográficas 167
9. Apéndices 168
a. Métricas 168
i. Métricas a nivel de implementación 168
ii. Métricas a nivel de juego 169
b. Arte 170
i. Ejército Orco 170
ii. Ejército Robot 172
iii. Assets del Escenario 173
c. Manual de Instrucciones Básicas 178
e. Capturas del juego 180
g. Glosario de términos 187

Índice de Figuras

Figura 4.1: Aspecto de la pestaña *Project* en el proyecto del juego. 10

Figura 4.2: Aspecto de la pestaña *Hierarchy* en uno de los escenarios prototipo del juego. 11

Figura 4.3: Aspecto de la pestaña *Scene* en uno de los escenarios prototipo del juego. 11

Figura 4.4: Control de la orientación de la cámara en la pestaña de escena. 12

Figura 4.5: Modos de manipulación de los objetos en el entorno tridimensional de la escena. 12

Figura 4.6: Aspecto de la pestaña *Game* en uno de los escenarios prototipo del juego. 12

Figura 4.7: El objeto *Dralien ArmoredSpiders* en el inspector de Unity. 13

Figura 4.8: Esquema visual del ciclo de juego en Unity. 16

Figura 4.9: Ejemplo del componente *Transform* en el editor de Unity. 18

Figura 4.10: Aspecto del componente *Rigidbody* en el inspector de Unity. 19

Figura 4.11: Aspecto del componente *Sphere Collider* en el inspector de Unity. 20

Figura 4.12: Aspecto de un material en el inspector de Unity. 24

Figura 4.13: Ejemplo de una textura simple de color de 512x512 píxeles de resolución en el inspector de Unity. 24

Figura 4.14: Ejemplo de mip maps en una textura. 25

Figura 4.15: Ejemplo de 3 texturas para un mismo material. 26

Figura 4.16: Las 3 texturas anteriores aplicadas al modelo. 26

Figura 4.17: Detalle del efecto de un mapa de Normales aplicado a un modelo sin mapa de color. 26

Figura 4.18: Ejemplo de un emisor de partículas en Unity. 28

Figura 4.19: Ejemplo del animador de partículas en Unity. 29

Figura 4.20: Ejemplo del procesador de partículas en Unity. 29

Figura 4.21: Ejemplo del colisionador de partículas en Unity. 30

Figura 4.22: Aspecto de las opciones de la pestaña *Navigation* en el editor de Unity. 31

Figura 4.23: Aspecto del componente *Nav Mesh Agent* en el editor de Unity. 34

Figura 4.24: Aspecto del componente *Nav Mesh Obstacle* en el editor de Unity. 35

Figura 4.25: Opción *OffMeshLink* Generation de Unity. 36

Figura 4.26: Opciones de *bakeado* del *OffMeshLink*. 36

Figura 4.27: Aspecto de varios *OffMeshLink* en una malla de navegación. 37

Figura 4.28: Generación manual del *OffMeshLink* en Unity. 37

Figura 4.29: Resultado de la creación manual de un *OffMeshLink*. 38

Figura 4.30: Aspecto de la herramienta Monodevelop. 39

Figura 4.31: Aspecto de la herramienta Microsoft Visual Studio 2012. 40

Figura 4.32: Asppecto de Microsoft Visual Studio 2012 con UnityVS integrado. 40

Figura 4.33: Boton NGUI en la barra de opciones. 41

Figura 4.34: Disposición de carpetas de Photon en la pestaña *Hierarchy* de *Unity*. 42

Figura 4.35: Opciones de conexión de Photon en editor de Unity. 43
Figura 4.36: Configuración de Photon.. 43
Figura 4.37: *enlace* de los componentes en el inspector de Unity. 45
Figura 5.1: Aspecto de varias unidades sin seleccionar. 49
Figura 5.2: Aspecto de varias unidades seleccionadas. 50
Figura 5.3: Ejemplo de selección simple. 52
Figura 5.4: Ejemplo de selección múliple por cuadrado. 54
Figura 5.5: ejemplo de la proyección de los puntos del rectángulo de pantalla (izquierda) en el cuadrilátero del mundo (derecha). 55
Figura 5.6: Ejemplo de selección múltiple por tipo de unidad... 58
Figura 5.7: Diagrama de clases simplificado de las unidades del *RTS*....................................... 60
Figura 5.8: Diagrama de estados de la clase UnitController. 65
Figura 5.9: Diagrama de estados de las unidades recolectoras del RTS. 68
Figura 5.10: Diagrama de estados de las unidades ingenieras del RTS..................................... 72
Figura 5.11: Diagrama de estados de las clases *UnitBasicArtillery* y *UnitHeavyArtillery*. 77
Figura 5.12: Aspecto gráfico de los 3 tipos de edificios de cada ejército (almacén de recursos, torretas y base). 79
Figura 5.13: Aspecto gráfico de una torreta neutral conquistable (izquierda), y del proceso de conquista por unidades ingenieras (derecha)... 79
Figura 5.14: Diagrama de clases simplificado de los edificios de recursos del RTS. 81
Figura 5.15: Diagrama de clases simplificado de las torres. 82
Figura 5.16: Diagrama de estados de las Torres. 86
Figura 5.17: Ejemplo visual del algoritmo de rotación de posiciones de destino....................... 93
Figura 5.18: Ejemplos visuales del resultado final del algoritmo de cálculo de múltiples destinos. 94
Figura 5.19: Creación del *OffMeshLink* en uno de los escenarios del juego. 99
Figura 5.20: Cámara RTS sin zoom. 99
Figura 5.21: Cámara RTS con zoom. 100
Figura 5.22: Variables del script *CameraRTSController*. 100
Figura 5.23: UML del *script CameraRTSController*. 102
Figura 5.24: Aspecto de la burbuja de visión en una unidad en el editor de Unity. 103
Figura 5.25: Ejemplo de las posiciones de unidades de dos ejércitos. 108
Figura 5.26: Muestra visual del recubrimiento de la matriz de distancias de la clase *DistanceMeasurerTool*. 109
Figura 5.27: Muestra de la matriz de distancias de la clase *DistanceMeasurerTool*. 109
Figura 5.28: Diagrama de componentes del héroe 111
Figura 5.29: Máquina de estados del héroe... 113
Figura 5.30: UML del script *HeroeController* 115
Figura 5.31: Componentes del ataque básico 116

Figura 5.32: UML del script *BasicAttack.cs* 117
Figura 5.33: Componentes de las habilidades activas con colisionador 118
Figura 5.34: Componentes de las habilidades activas con colisionador de partículas 118
Figura 5.35: UML del script *SkillAttack*... 118
Figura 5.36: Componentes de las habilidades pasivas.. 118
Figura 5.37: UML del script *SkillDefense* 119
Figura 5.38: UML del script *ParticleDamage.cs* 121
Figura 5.39: UML del script *OrcController*.. 124
Figura 5.40: UML del script *RobotController*.. 128
Figura 5.41: Parámetros del script *ThirdPersonCamera.cs.* 129
Figura 5.42: UML del script *ThirdPersonCamera.*.. 132
Figura 5.43: Aspecto visual del script *CameraMOBAController* (1). 133
Figura 5.44: Aspecto visual del script *CameraMOBAController* (2). 133
Figura 5.45: Aspecto visual del script *CameraMOBAController* (3). 134
Figura 5.46: UML del script *CameraMOBAController.* 134
Figura 5.47: Menú de creación o selección de una sala 137
Figura 5.48: Menú de selección de equipo y modo de partida... 138
Figura 5.49: Menú de selección de equipo y modo de partida... 138
Figura 5.50: Componentes de un objeto en red. 139
Figura 5.51: Aspecto de la niebla de guerra... 144
Figura 5.52: Minimapa con tropas enemigas (1). 145
Figura 5.53: Minimapara con tropas enemigas (2). 146
Figura 5.54: Minimapa con tropas enemigas (3). 146
Figura 5.55: Minimapa con edificios enemigos (1). 146
Figura 5.56: Minimapara con edificios enemigos (2). 147
Figura 5.57: Minimapa con edificios enemigos (3). 147
Figura 5.58: Matriz en el minimapa (1)... 149
Figura 5.59: Matriz en el minimapa (2)... 149
Figura 5.60: Matriz en el minimapa (3)... 150
Figura 5.61: Matriz en el minimapa (4)... 150
Figura 5.62: Ejemplo del aspecto que produce el *shader* *BumpedDiffuseColorTextureSpecularColor*. 152
Figura 5.63: Ejemplo del aspecto que produce el *shader* *BumpedDiffuseIndepTextureSpecularOutlinedViewPos*.. 153
Figura 5.64: Ejemplo del aspecto que produce el *shader* *BumpedIndepTextureSpecularRefactionOutlinedViewPos*.. 154
Figura 5.65: Ejemplo del aspecto que produce *el shader Alpha Color Noise Beat Color Diffuse*. 155

Figura 5.66: Ejemplo del aspecto del *shader Alpha Color Noise Beat Color Diffuse* en un edificio del juego. 155
Figura 5.67: Textura de ruido *PerlinNoise* para el *Alpha Clip* del *shader Alpha Color Noise Beat Color Diffuse*. 155
Figura 5.68: Secuencia ejemplo del aspecto que produce el *shader UnitDie Timer*. 156
Figura 5.69: Textura de ruido *PerlinNoise* para el *Alpha Clip* del *shader UnitDie Timer*. 156
Figura 5.70: Aspecto de los 3 sistemas de partículas expuestos en el ejemplo anterior en el inspector de Unity. 157
Figura 5.71: Aspecto del sistema de partículas de la explosión del explorador goblin. 158
Figura 5.72: Aspecto de la explosión en el inspector de Unity. 159
Figura 5.73: Secuencia de imágenes que ilustra el aspecto de esta explosión en el juego. 159
Figura 5.74: Sistema de partículas de la habilidad 1 de Rob Render. 159
Figura 5.75: Sistema de partículas de la habiliad 2 de Rob Render. 160
Figura 5.76: Aspecto de los dos sistemas de partículas de la habilidad 3 de Rob Render. 160
Figura 5.77: Sistema de partículas de la habilidad 1 de Skelterbot. 161
Figura 5.78: Sistema de partículas de la habilidad 2 de Skelterbot. 161
Figura 5.79: Sistema de partículas de la habilidad 3 de Skelterbot. 162
Figura 5.80: Interfaz, panel principal del juego. 162
Figura 5.81: Interfaz, panel de selección de sala. 163
Figura 5.82: Interfaz, panel de selección de rol - boton desplegado. 163
Figura 5.83: Interfaz, panel de selección de rol - personajes de ejercito. 163
Figura 9.1: Presentación de la banda de los Reb Lobster. 170
Figura 9.2: Aspecto de los edificios del ejército goblin. 171
Figura 9.3: Presentación de la banda de los Skelters. 172
Figura 9.4: Aspecto de los edificios del ejército robot. 173
Figura 9.5: Aspecto de las torretas neutrales. 173
Figura 9.6: Gasolinera del escenario. 174
Figura 9.7: Edificio del distrito goblin. 174
Figura 9.8: edificio del barrio pobre de NewDetroit. 174
Figura 9.9: Catedral neo-gótica de NewDetroit. 175
Figura 9.10: Puente principal de la ciudad de NewDetroit. 175
Figura 9.11: Set de árboles de NewDetroit. 175
Figura 9.12: Parada de Metro de NewDetroit. 176
Figura 9.13: Dispensadores de periódicos de NewDetroit. 176
Figura 9.15: Set de assets 2 de NewDetroit. 177
Figura 9.14: Set de assets 1 de NewDetroit. 177
Figura 9.16: Motos Choper de los Red Lobster. 177

Índice de Fragmentos de Código

Código 4.1: Ejemplo de un evento de entrada en un colisionador. 21

Código 4.2: Ejemplo de aplicación de un impulso de fuerza a un *Rigidbody*. 22

Código 4.3: Ejemplo del uso de físicas para "golpear" a una unidad y que salga volando. 23

Código 4.4: Ejemplo de *script* de Unity para una unidad instanciada remotamente usando Photon. 45

Código 4.5: Instrucción para instanciar un objeto remótamente con Photon. 46

Código 5.1: Método *Awake* de la clase *CSelectable*. 51

Código 5.2: Métodos de selección y deselección de la clase *CSelectable*. 52

Código 5.3: Selección simple. 53

Código 5.4: Lanzamiento de rayos para calcular el cuadrilátero proyectado en el mundo. 54

Código 5.5: Bucle que determina si una unidad está dentro del cuadrilátero. 56

Código 5.6: Cálculo de la distancia de un punto a un segmento determinado por 2 puntos. 56

Código 5.7: Cálculo simplificado del signo de la distancia de un punto a un segmento determinado por 2 puntos. 57

Código 5.8: Atributo *screenPosition* de la clase *ControllableCharacter*. 57

Código 5.9: Método *CreatingSquare2* para la selección múltiple de unidades según su posición en coordenadas de pantalla. 58

Código 5.10: Ejemplo de código de una de las últimas clases en la jerarquía de las unidades del *RTS*. 61

Código 5.11: Método *Update* de la clase *UnitController*. 64

Código 5.12: función *SearchForAnEnemy* de la clase *UnitArtillery*. 74

Código 5.13: Fragmento de la clase *UnitHeavyArtillery* encargado del cambio de estado por medio de corrutinas. 76

Código 5.14: función *SetCanConstruct* de los ingenieros para establecer la posibilidad de la construcción. 83

Código 5.15: función *StartConstruct* de los ingenieros para establecer el comienzo de la construcción de edificios. 84

Código 5.16: función *Conquest* de la clase *NeutralTower*. 85

Código 5.17: función *Repair* de la clase *BuildingController*. 86

Código 5.18: Inserción de una nueva mina de recursos en la lista de minas de *ArmyController*. 88

Código 5.19: Inserción de un nuevo almacén de recursos en la lista de *ArmyController*. 89

Código 5.20: Ejecución del movimiento con ataque automático en la clase *ArmyController*. 90

Código 5.21: Elección de puntos de patrulla para los exploradores en la clase *ArmyController*. 90

Código 5.22: Finalización de elección de puntos de patrulla en la clase *ArmyController*. 91

Código 5.23: Nueva Torre o nuevo Almacén de Recursos 91
Código 5.24: Algoritmo de selección de múltimpes destinos. 97
Código 5.25: función *GoTo* de la clase *UnitController* que establece el destino al componente *NavMeshAgent*. 98
Código 5.26: Velocidad aumentada del desplazamiento en los ejes X y Z de la cámara. 101
Código 5.27: Movimiento de la cámara por medio del botón central (rueda) del ratón. 101
Código 5.28: Movimiento de la cámara por medio del teclado. 102
Código 5.29: Zoom de la cámara. 102
Código 5.30: Clase *CTriggerVisionSphere* que detecta las colisiones entre unidades. 104
Código 5.31: Atributos de la clase *DistanceMeasurerTool*. 106
Código 5.32: Primera parte del algoritmo de búsqueda de la clase *DistanceMeasurerTool*. ... 107
Código 5.33: Segunda parte del algoritmo de búsqueda de la clase *DistanceMeasurerTool*. .. 107
Código 5.34: Máquina de estados del héroe 114
Código 5.35: Función *RotateTowars* de Unity. 114
Código 5.36: Rotación suave del héroe 114
Código 5.37: Detección de colisiones y daño en el ataque básico 116
Código 5.38: Código del script *SkillAttack.cs* 120
Código 5.39: Código del script *SkillDefense.cs* 121
Código 5.40: Animaciones del orco 123
Código 5.41: Instanciación de las habilidades del orco 124
Código 5.42: Animaciones del robot 127
Código 5.43: Instanciación de las habilidades del robot 128
Código 5.44: Modificación del tipo de *controller*. 129
Código 5.45: Código *OnEnable*. 130
Código 5.46: *OnEnable* posiciones central y superior. 130
Código 5.47: *OnEnable* rotaciones en el eje y del héroe y de la cámara. 131
Código 5.48: *OnEnable* nuevo ángulo de la cámara. 131
Código 5.49: *OnEnable* altura de la cámara. 131
Código 5.50: *OnEnable* halla la rotación de tipo *Quaternion* con recpecto al eje y. 131
Código 5.51: *OnEnable* posición de la cámara con respeto al plano XOZ. 131
Código 5.52: *OnEnable* actualiza la altura de la cámara. 131
Código 5.53: *OnEnable* mira hacia el héroe. 132
Código 5.54: *SetUpRotation* desviación entre la posición de la cámara y el centro del héroe. 132
Código 5.55: *SetUpRotation* rotación de la cámara. 132
Código 5.56: Configuración de Photon 135
Código 5.57: Menú de creación o selección de sala 137
Código 5.58: Menú de selección de equipo y modo de partida 138
Código 5.59: Instanciación de objetos en Photon. 139

Código 5.60: Modificación de la posición y rotación a través de Photon. 140
Código 5.61: Aplicación de Fuerza y daño a una unidad a través de red................................ 141
Código 5.62: Llamada a una función RPC. 141
Código 5.63: Modificación de los atributos de defensa a través de red. 141
Código 5.64: Implementación del script *FogOfWarUnit.* 143
Código 5.65: Implementación del script *ForOfWarPlane.* 144
Código 5.66: Actualización de las unidades aliadas en el minimapa. 148
Código 5.67: Actualización de las unidades enemigas en el minimapa. 148
Código 5.68: Actualización de las casillas de la matriz en el minimapa. 149
Código 5.69: Propiedades del *shader BumpedDiffuseColorTextureSpecularColor*. 152
Código 5.70: Cálculo del ancho de la línea de contorno en el *shader BumpedDiffuseIndepTextureSpecularOutlinedViewPos*.. 153
Código 5.71: fragmento del shader *UnitDie Time* donde se realiza la transformación del *Alpha Clip* en base al tiempo. 156

1. Introducción

a. Motivaciones

Como se ha comentado en el resumen de la presente memoria, en los últimos años ha habido un gran crecimiento en el número de videojuegos desarrollados de forma independiente, es decir, sin la financiación de grandes editores o *publishers*.

Un buen ejemplo de este interés creciente, se vive en la Facultad de Informática de la Universidad Complutense de Madrid, donde cada vez más estudiantes se aventuran a realizar videojuegos para las asignaturas de prácticas de programación, o de *Ingeniería del Software* y, cómo no, como proyecto de *Sistemas Informáticos* suponiendo un colofón al final de una carrera tan exigente como es Ingeniería en Informática.

Un ejemplo de este tipo de estudiantes somos los que hemos realizado el presente proyecto, cuyo germen se gestó durante el pasado año gracias al haber participado todos juntos en el mismo grupo en la asignatura de *Ingeniería del Software*, desarrollando, con la ayuda de los profesores Dr. Federico Peinado Gil y Dr. Samer Hassan, un videojuego matamarcianos de nombre *Project_Chandra*[1] con la tecnología XNA de Microsoft. Según avanzaba el proyecto de esta asignatura, teníamos la intención cada vez más fuerte de realizar un videojuego en 3D como proyecto fin de carrera. Esta intención fue rápidamente bien recibida por D. Federico Peinado, sin embargo, debido a la ambición con la que se quería dotar a este proyecto, se pensaba que un grupo de tres personas se antojaba escaso, por lo que se intentó durante todo el verano conseguir el apoyo de un segundo director de proyecto, y así D. Fernando Rubio Diez ofreció su visto bueno y comenzó el desarrollo de este proyecto combinado por dos grupos de tres personas.

Aunque la mayoría de las industrias, incluida la del videojuego, se ha visto afectada por una situación de recesión, esta última, a la que nos acercamos, muestra un crecimiento significativo y la mayoría de las compañías siguen mostrando beneficios y buenos datos sobre ventas en el mercado. Esto es, en parte, gracias al aumento de nuevas oportunidades de negocio debido a la irrupción de modelos de negocio como el *freeToPlay*, en el que se ofrece el juego sin coste para el jugador y se consigue recaudar gran cantidad de dinero gracias, tanto a la venta de complementos para el propio juego, como a la gran aceptación de público que acaban consiguiendo. Dichos datos son un aliciente para nuestra decisión de afrontar el reto de desarrollar un videojuego con una futurible visión comercial, ya que tomando como ejemplo un juego de este tipo, *League of Legends*, las pretensiones y motivaciones son bastante ilusionantes.

El juego mencionado en el párrafo anterior se trata de un MOBA, y sirve como ejemplo perfecto de que un videojuego con monetización tipo *free-to-play* puede tener un gran éxito comercial logrando atraer una comunidad muy numerosa, tanto de jugadores habituales como otros más ocasionales, gracias a una jugabilidad que destaca por su sencillez inicial y por su exigencia para alcanzar niveles superiores de dificultad. Por supuesto, gracias al haber formado

[1] http://projectchandra.co.nf/

una comunidad tan amplia, permite obtener grandes beneficios de forma distinta a la forma clásica de venta de juegos.

Más allá de este hecho, por afición, gusto o atracción a nivel personal o grupal, esta industria del videojuego nos aporta suficiente motivación para pretender sumergirnos en ella.

La decisión del modo de juego derivó de nuestros propios gustos y del conocimiento actual de esta industria. A nivel personal y grupal la idea de mezclar dos modos de juego tan atractivos, ya no sólo para nosotros, sino para la comunidad *gamer* en general, fue una de las mayores motivaciones, ya que a día de hoy la unión de jugadores desempeñando las labores de cada modo es algo inexistente, calificando este proyecto de novedoso y único debido a la idea original que materializa.

b. Interés

La forma de plantear y visualizar el proyecto no tiene un punto final en el momento de finalización del curso. Desde el principio se estableció un sistema de *Ingeniería del Software* con el fin de poder continuar con la evolución del proyecto tras la finalización del curso. Por esto se desarrolló una *Gestión de Proyecto* eficiente y que va más allá de la idea de proyecto de fin de carrera, centrándose en potenciar un desarrollo eficaz y competitivo en la industria española para tener planificado un rumbo y un progreso que permita crecer dentro de esta industria.

Esta idea ha ejercido de influencia a la hora de tomar decisiones de configuración de la estructura de trabajo y ha producido un aumento en el interés de adentrarse en la industria del videojuego. Además ha resultado de impulso el hecho de que el Ministerio de Industria, Energía y Turismo publicó recientemente el lanzamiento de ayudas al sector del videojuego y adelantaron que se abriría la convocatoria de ayudas. Este es el resultado del cumplimiento de uno de los compromisos recogidos en el Plan de Impulso de la Economía Digital y los Contenidos Digitales integrado dentro de la Agenda Digital para España aprobada por el Consejo de Ministros en febrero de 2013. Así mismo, se espera que el resultado final satisfaga las necesidades del sector y conseguir el mayor impacto de las ayudas.

Gracias a todo esto y al proyecto innovador en el que nos encontramos, vemos la posibilidad de buscar el hueco dentro de la industria que nos permita cumplir con los objetivos establecidos.

c. Objetivos

Decidido el estilo del videojuego a realizar durante el proyecto, el siguiente paso fue fijar los objetivos a cumplir durante el proyecto. Principalmente se fijó tener a final de curso un nivel jugable entre cuatro jugadores, divididos en dos equipos, y comprobar el resultado de la unión innovadora entre los modos MOBA y RTS.

Además, no sólo se planteó como objetivo poder combatir entre los jugadores, sino también, y gracias a la disponibilidad inusual de contar con gente de distintos ámbitos, dar a los jugadores variedad en la elección de juego: héroes, ejércitos, etc...

Pero los objetivos no acaban ahí, el hecho de ser un proyecto ambicioso, especial y con una idea novedosa que lo convierte en un videojuego único al mezclar los modos MOBA y RTS, nos marcan el objetivo de evolucionar el juego de forma correcta para convertirlo, en lo posible, en un videojuego con una gran jugabilidad y con el don de enganchar a jugadores con facilidad.

d. Composición de Grupos de Trabajo

Para poder realizar un videojuego de semejantes características hace falta un grupo de personas muy numeroso y entre las cuales tiene que haber gente dedicada al apartado artístico; por esto, durante el proceso de desarrollo han trabajado dos grupos de dos universidades diferentes, uno del Grado en Desarrollo de Videojuegos de ESNE encargado del Arte del videojuego, y otro de Ingeniería en Informática de la Facultad de Informática de la Universidad Complutense de Madrid, encargado de la implementación del videojuego. La idea nació de los componentes de la Universidad Complutense de Madrid y ambos grupos han tenido tareas comunes como la producción y el diseño, habiendo una persona, Maximiliano Miranda Esteban, que ha sido el nexo de unión de estos grupos ya que pertenece a ambas universidades.

Finalmente, el grupo de desarrollo de la Universidad Complutense ha estado integrado por seis alumnos de Sistemas Informáticos. Sin embargo, a la hora de documentar el proyecto se estableció desde el principio una división en dos grupos de tres personas. Por un lado, un grupo ha estado documentando la presente memoria y otro grupo una memoria sobre *Ingeniería Software y Diseño del Videojuego* (Cuesta Boluda, Cuesta Boluda, & Rodríguez-Osorio Jiménez, 2014).

2. Revisión y Estado del Arte

A la hora de decidir el tipo de videojuego que se deseaba desarrollar en este proyecto se optó por un videojuego de género RTS porque se pensaba que gracias a su característica jugabilidad se ajustaba bien a un desarrollo exigible para un proyecto de fin de carrera, y además se entiende que es un género que ha gozado de una excelente popularidad durante muchos años. Además, se pretendía añadir un elemento de originalidad, y se decidió introducir una parte de género MOBA, menos longevo pero con una gran proyección en los últimos años. Tras esta decisión, se llevó a cabo un estudio del estado del arte de los videojuegos pertenecientes a ambos géneros. Este estudio, se encuentra en la otra memoria desarrollada a lo largo de este proyecto sobre *Ingeniería del Software y Diseño del Videojuego* (Cuesta Boluda, Cuesta Boluda, & Rodríguez-Osorio Jiménez, 2014), sin embargo, se ha estimado interesante exponer en este capítulo un breve resumen de este estudio.

Los videojuegos RTS o de estrategia en tiempo real (*Real-Time Strategy*) son videojuegos en los que todos los jugadores se desenvuelven a la misma vez y no por turnos. Y consisten en juegos donde se maneja una civilización, y que poseen características tan variadas como la utilidad de los recursos y las diversas funcionalidades de unidades y construcciones.

Los MOBA, o *Multiplayer online battle arena,* aparecieron como subgénero de los RTS, pero cambiando elementos tan fundamentales como la cámara o el control único de una unidad, eliminando características como la gestión de tropas y recursos, y añadiendo importancia a la evolución del personaje o al uso de habilidades y magia, llegando a divergir tanto, en un breve periodo de tiempo, que pronto se acuñó el término MOBA como un nuevo género independiente.

La unión de estos géneros nos centra en un nuevo subgénero cooperativo, aunque el objetivo heredado de ambos géneros es común, derrotar la estructura principal del enemigo.

El estudio de mercado se ha centrado en videojuegos de ambos géneros que han sido seleccionados debido a su originalidad, influencia o popularidad, como son:

- **MOBA**: *League Of Legends*[2], *Prime world*[3], *Bloodline champions*[4], *Airmech*[5], *Super MNC (Monday Night Combat)*[6], *Smite*[7], *Smashmuck Champions*[8] y *Dota 2*[9].
- **RTS**: *StarCraft 2*[10], *WarCraft III*[11] y *Saga Age of Empires*[12].

[2] http://euw.leagueoflegends.com/
[3] http://en.playpw.com/
[4] http://www.bloodlinechampions.com/
[5] https://carbongames.com/
[6] http://www.uberent.com/smnc/
[7] http://www.hirezstudios.com/smite
[8] https://www.smashmuck.com/
[9] http://es.dota2.com/
[10] http://eu.battle.net/sc2/es/
[11] http://us.blizzard.com/es-es/games/war3/?locale=es-es
[12] http://www.ageofempires.com/

Los juegos mencionados que más similitudes tienen con el videojuego desarrollado para este proyecto en el género RTS son *WarCraft III* y *StarCraft 2*, en los que existe una unidad única que mejora su potencia militar y sube de nivel pero el ejército es controlado por el mismo jugador.

Referente al género MOBA, los juegos más parecidos son *League of legends* y *Smite*, en los que los cambios hacen referencia al número de habilidades para poder usar y la cámara, mucho más perpendicular en *League of Legends* pero muy similar a la cámara de *Smite*.

Sin embargo, es importante destacar que actualmente no existe ningún videojuego en el mercado que combine estos dos géneros de manera asimétrica, esto es, sin mezclar las mecánicas de ambos en una sola jugabilidad, sino manteniendo ambas y permitiendo a varios jugadores jugar de diferente manera en una misma partida.

Todos estos juegos han sido estudiados con mayor detalle en la citada memoria de *Ingeniería del Sofware y Diseño del Videojuego* (Cuesta Boluda, Cuesta Boluda, & Rodríguez-Osorio Jiménez, 2014).

3. Diseño del Videojuego

Uno de las puntos más importantes a la hora de desarrollar un videojuego es el que se reproduce en el Documento de Diseño del Videojuego, donde se expone la perfecta definición del comportamiento del videojuego, como mecánicas jugables, tipos de cámara, unidades que aparecerán, etc. Los géneros MOBA y RTS tienen una gran cantidad de mecánicas jugables diferentes, por lo que este punto es primordial para el buen funcionamiento y evolución del proceso de desarrollo del proyecto durante el curso.

Para esta función, se desarrolló un detallado documento de diseño del videojuego, conocido por las siglas GDD (del inglés *Game Design Document*), que ha servido de guía tanto para preparar la producción del videojuego, como para realizar el diseño software de las diversas funcionalidades que se han desarrollado, convirtiéndose en una pieza fundamental de un proyecto basado en un videojuego ya que contiene los detalles sobre las mecánicas que conforman el juego, y por lo tanto, la especificación de las funcionalidades.

Este documento se encuentra perfectamente plasmado en la otra memoria desarrollada a lo largo de este proyecto sobre *Ingeniería Software y Diseño del Videojuego* (Cuesta Boluda, Cuesta Boluda, & Rodríguez-Osorio Jiménez, 2014), sin embargo, se entiende interesante esbozar las piezas principales de este diseño del Videojuego, para ofrecer una visión más clara de la clase de videojuego que se ha desarrollado para este proyecto.

i. Ficha resumen

Con los siguientes puntos se busca transmitir a grandes rasgos, y desde una primera vista, los puntos básicos del videojuego.

- Título del proyecto: Project NewDetroit (nombre provisional).
- Título del videojuego: Mutant Meat City.
- Género: estrategia, RTS (estrategia en tiempo real) + MOBA (multijugador online en arena de batalla).
- Plataformas: PC y MAC.
- Modos de juego: 100% online, 1vs1, 2vs2.
- Público destinado: jugadores habituales, mayores de 12 años.

ii. Breve descripción

El videojuego se describe como un juego de estrategia que mezcla los géneros MOBA y RTS de manera asimétrica, es decir, no se mezclan mecánicas de ambos géneros en una única jugabilidad, sino que se incluyen las dos jugabilidades de forma simultánea en una misma partida del juego. Esto es así ya que el videojuego está totalmente enfocado al multijugador online de manera que equipos formados por parejas de jugadores se enfrentan en una arena, y cada jugador de estas parejas juega de diferente forma, por un lado, un jugador controla un héroe como si fuera un MOBA y el otro jugador controla a un ejército que asiste a este héroe como un RTS.

De esta forma se mezclan elementos cooperativos y competitivos de ambos géneros (asistirse entre jugadores del mismo equipo y juntos lograr así derrotar al equipo contrario para ganar la partida). El objetivo de los jugadores es destruir la base del ejército RTS del equipo rival.

Las partidas transcurren en arenas de combate de temática variada. Así, la ambientación del videojuego dependerá del escenario en el que se juegue, siendo en general un mundo con estética futurista distópica.

1. RTS

La Cámara de esta parte del juego presenta al mundo desde una vista aérea con una breve inclinación para ofrecer una visión clara del "campo de batalla". El control sobre las tropas se realiza seleccionando unidades con el ratón (clic izquierdo) y lanzando órdenes (clic derecho), otras acciones más específicas se realizan con atajos de teclado (como movimiento con ataque automático, órdenes para construir edificios, etc.).

El jugador cuenta con los siguientes tipos de unidades controlables para su ejército:

- Recolectores: encargados de recolectar los minerales necesarios para crear más tropas y construir edificios.
- Artillería ligera: unidad bélica de medio alcance.
- Artillería pesada: unidad bélica de gran alcance pero de movimientos lentos.
- Exploradores: unidad que se desplaza de forma rápida sobre una montura y que es capaz de saltar por ríos y otros accidentes del terreno.
- Ingenieros: los únicos capaces de conquistar torres de defensa y de construir otros edificios.

Los **tipos de edificios** son los siguientes:

- Base del ejército: es el único lugar donde se pueden crear nuevas unidades, y resulta de vital importancia su defensa, ya que esta es única y cuando se destruye se pierde la partida.
- Torretas del ejército: los ingenieros pueden construir estas torres que atacan automáticamente a cualquier enemigo que entre en su radio de visión.
- Almacenes de recursos: los recolectores, tienen que volver a la base a dejar los recursos cuando estén picando en una mina y "se llenen", este camino de vuelta puede ser muy largo, pero estos almacenes pueden simplificar esta tarea si se construyen cerca de las minas de recursos.
- Torretas neutrales: al comienzo de una partida, por el nivel se encuentran repartidas estas torretas que no atacan a nadie, sino que tienen que ser conquistadas por unidades ingenieras para que comiencen a servir al ejército del jugador.

2. MOBA

La **cámara** de este modo de juego muestra al héroe desde una vista trasera en tercera persona, pudiendo ver el mundo que tiene frente a él mismo. El **control** sobre el héroe se

realiza con las teclas W (movimiento hacia delante), A (movimiento hacia la izquierda), S (movimiento hacia la derecha) y D (movimiento hacia atrás).

El jugador que controla al héroe cuenta con las siguientes opciones jugables:

- Uso de habilidades: accesibles con las teclas numéricas 1, 2 y 3. Pueden ser habilidades mágicas o físicas gastando su correspondiente en mana o adrenalina.
- Uso de ataque básico simple, doble o triple (encadenado): pulsando el botón izquierdo del ratón realiza el ataque básico; si se mantiene pulsado desencadena un ataque básico doble o triple dependiendo del tiempo que esté presionado.
- Recoger tesoros y obtener recompensas: durante la partida, el héroe puede enfrentarse a enemigos NPCs dispuestos en el mundo del juego. que le otorgarán tesoros o recompensas tras derrotarlos.

4. Tecnologías empleadas

Hay diversos motores 3D para el desarrollo de videojuego, como Unity, UDK 4 o CRYENGINE. Se recomienda consultar la memoria sobre *Ingeniería Software y Diseño del Videojuego* de este proyecto (Cuesta Boluda, Cuesta Boluda, & Rodríguez-Osorio Jiménez, 2014).

El proyecto ha sido desarrollado con el motor para videojuegos Unity, utilizando además scripting en C#. También se han utilizado otras tecnologías como UnityVS para poder depurar código en tiempo de ejecución, Photon Network para simplificar la lógica del juego en red y NGUI para el diseño e implementación de la interfaz de usuario del juego.

En este capítulo se hace una breve aproximación a estas tecnologías, haciendo hincapié en los aspectos que más útiles han resultado para el desarrollo de este proyecto, para así poner en contexto el siguiente capítulo sobre Implementación donde se citan constantemente apartados tratados en este capítulo.

a. Unity

Unity es una interfaz gráfica en 3D que sirve de herramienta para crear videojuegos, aplicaciones interactivas, visualizaciones y animaciones en 3D y tiempo real. Unity puede implementar contenido para múltiples plataformas. El editor de Unity es el centro de la línea de producción, ofreciendo un editor visual para crear juegos. El contenido del juego es construido desde el editor y ejecución o *gameplay*, pero fundamentalmente a través de la arquitectura de componentes que conforman Unity. Cuenta con una documentación oficial muy extensa y detallada[13], sin embargo, en los siguientes apartados se resumirán los aspectos o partes del programa que más útiles han resultado en el desarrollo del proyecto.

Para trabajar en Unity siempre se tiene que crear un proyecto. Este a su vez crea una carpeta con los archivos necesarios, donde se indicará los paquetes que contendrá el mismo. Al crearlo se debe guardar como escena. Cada escena consiste en un nuevo nivel para el proyecto. Cuando se guarda la escena queda en una carpeta llamada *Assets* que es creada al principio del proyecto.

Las escenas contienen los objetos del juego. En cada escena se coloca el ambiente que se quiera, los obstáculos, la decoración, el diseño y en si lo que se quiere para el proyecto.

Hay varias formas de reorganizar el diseño de la interfaz de usuario. Unity proporciona diferentes opciones, que se pueden cambiar mediante la selección del menú desplegable que aparece en la parte derecha de la ventana de la aplicación.

Las escenas pueden visualizarse en una de las ventanas aportadas por Unity, allí es donde se puede observar y ejecutar cómo va quedando el proyecto. El proyecto además no tiene iluminación por defecto, este utiliza la cámara principal para ver los objetos en la ventana de juego, aunque todas estas opciones, como la iluminación, pueden añadirse.

[13] http://docs.unity3d.com/Manual/

Así mismo, Unity proporciona vistas donde se encuentra todo lo que contiene el proyecto, como las escenas, carpetas, objetos etc., desde allí también se puede eliminar lo que se quiera, eliminando también su ubicación en el disco duro.

i. El editor

El editor de Unity es clave dentro de la herramienta, ya que representa uno de sus elementos más importantes y más diferenciadores frente a otros motores o herramientas de creación de videojuegos, gracias a su amplitud, versatilidad y generosas opciones. A continuación se exponen brevemente los diferentes elementos que conforman de la interfaz de Unity.

1. Project Browser (explorador del proyecto)

En esta vista se puede acceder y gestionar los archivos que pertenecen al proyecto. El panel izquierdo del navegador muestra la estructura de las carpetas del proyecto. Cuando se selecciona una carpeta de la lista, su contenido se mostrará en el panel a la derecha. Los archivos individuales se muestran como iconos que indican su tipo (*scripts*, materiales, etc). También existe una sección de favoritos donde pueden guardarse artículos de uso frecuente para un rápido acceso.

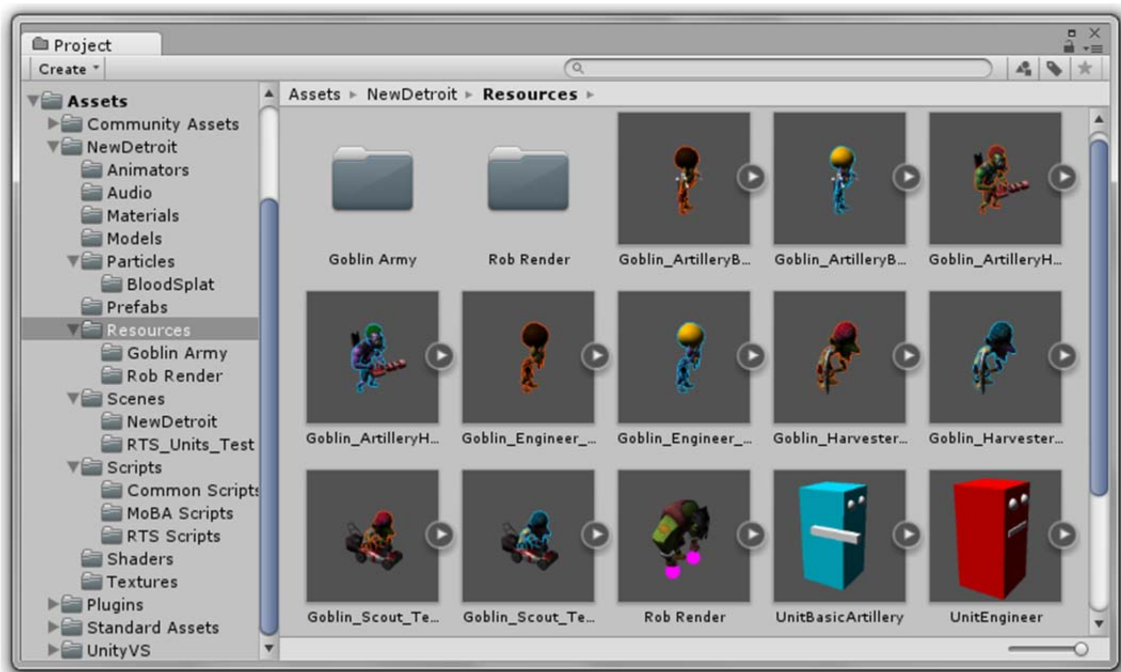


Figura 4.1: Aspecto de la pestaña *Project* en el proyecto del juego.

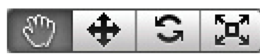
2. Hierarchy (jerarquía)

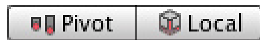
La Jerarquía contiene los *GameObject* de la escena actual. Algunos de estos son modelos 3D, objetos prefabricados, personalizados, etc. que formarán parte de la escena actual del juego.

Figura 4.2: Aspecto de la pestaña *Hierarchy* en uno de los escenarios prototipo del juego.

3. Toolbar (barra de herramientas)

La barra de herramientas se compone de cinco controles básicos. Cada uno hace referencia a cada una de las vistas:

 *Transform Tools* - se utiliza con la vista de escena



Transform Gizmo Toggles - afecta a la pantalla Vista de escena

Play/Pause/Step Buttons - usado con el juego en ejecución

Layers Drop-down - controles para especificar las capas de objetos que se mostrarán en la vista de escena

Layout Drop-down - controles a disposición en todas las vistas

4. Scene View (vista de la escena)

La vista de escena es una ventana con carácter interactivo que permite seleccionar y posicionar (mover, rotar, escalar) el jugador, la cámara, los enemigos, y todos los demás *GameObjects* de la escena actual. Maniobrar y manipular los objetos dentro de la vista de escena son algunas de las funciones más importantes de Unity, por lo que es importante ser capaz de hacerlo rápidamente.

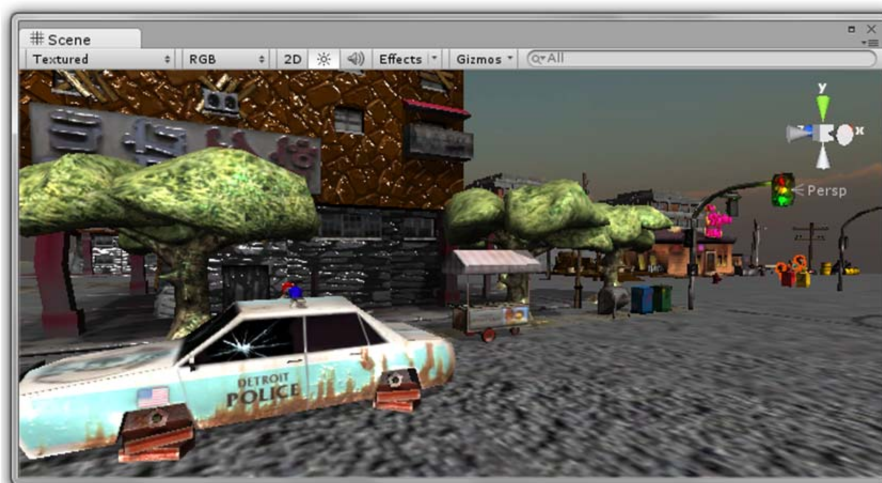
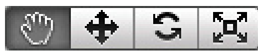


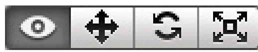
Figura 4.3: Aspecto de la pestaña *Scene* en uno de los escenarios prototipo del juego.

En la esquina superior derecha de la *Scene View* encontramos la orientación actual de la cámara, que también permite modificar rápidamente el ángulo de visión. Cada uno de los "brazos" de color representa un eje geométrico. Se puede hacer clic en cualquiera de los brazos para ajustar la cámara de la propia escena (que no la del juego).

Figura 4.4: Control de la orientación de la cámara en la pestaña de escena.

También encontramos controles que permiten jugar con la perspectiva:

 Haciendo clic y arrastrando se mueve la cámara alrededor.

 Haciendo clic y arrastrando para girar la cámara alrededor del punto de giro actual.

Haciendo clic y arrastrando para acercar o alejar la cámara.

En la creación de juegos se coloca una gran cantidad de objetos. Unity nos ofrece herramientas de transformación en la barra de herramientas para trasladar, girar y escalar *GameObjects* de forma individual o grupal.

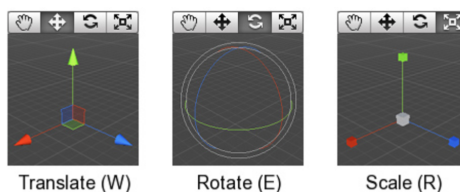


Figura 4.5: Modos de manipulación de los objetos en el entorno tridimensional de la escena.

5. Game View (vista del juego)

Representa, desde su cámara, el juego que se ha creado, en ejecución. En tiempo de ejecución, Unity permite manipular la escena desde el editor, y además manipular valores de los *scripts* y ver cómo afectan estos cambios al propio juego. Estos cambios son temporales, y se restablecen cuando se sale del modo de reproducción. El propio Unity ofrece que el editor de la interfaz se oscurezca para recordar esto.

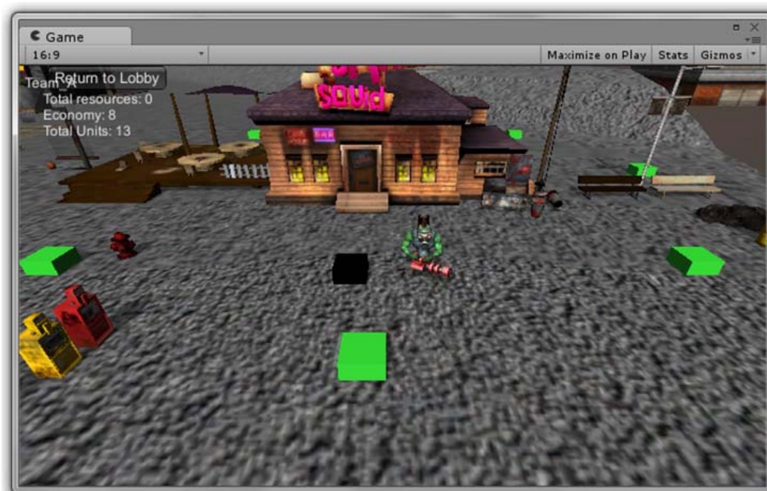


Figura 4.6: Aspecto de la pestaña *Game* en uno de los escenarios prototipo del juego.

En esta ventana se permite modificar el aspecto de la ventana como prueba para distintos monitores, maximizar la reproducción y visualizar el rendimiento del juego (*drawcalls*, *fps*, etc.).

6. Inspector

Los videojuegos en Unity se componen de múltiples *GameObjects* que contienen mallas, *scripts*, sonidos, otros elementos gráficos, etc. El Inspector muestra información detallada acerca de todos estos componentes que tiene el *GameObject* seleccionado, incluyendo sus propiedades. Aquí se puede modificar/configurar la funcionalidad de los *GameObjects* en su escena.

Cualquier propiedad que se muestra en el Inspector puede modificarse directamente, incluso las variables de proceso se pueden cambiar sin modificar el propio *script*. Se puede utilizar el *Inspector* para cambiar el valor de las variables en tiempo de ejecución para hacer pruebas.

En este ejemplo de las arañas *Dralien ArmoredSpider* se aprecian varios de los componentes que conforman el propio *GameObject*:

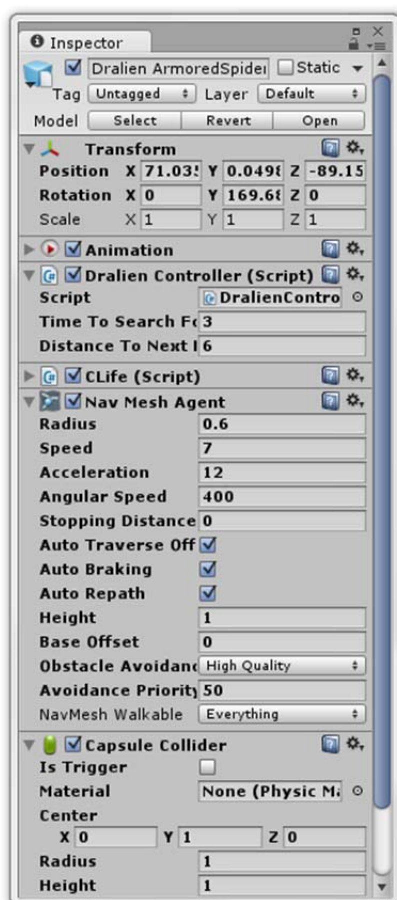


Figura 4.7: El objeto *Dralien ArmoredSpiders* en el inspector de Unity.

- *Transform*: atributos de todos los objetos que representan su posición, rotación y escala en el mundo.
- *Animation*: controlador de las animaciones disponibles en el modelo 3D.

- *DralienController*: *script* que contiene el código con el comportamiento de las arañas. Aquí se aprecia cómo el diseñador puede modificar el valor de los atributos públicos de la clase, en este caso *TimeToSearchForNextDest* que indica la frecuencia (en segundos) con la que la araña busca un nuevo destino, y *DistanceToNextDest* que representa la distancia máxima a la que la araña se desplazará en el siguiente movimiento. Gracias al editor de Unity se pueden modificar los valores en el propio editor sin necesidad de acceder al código fuente de la clase, facilitando mucho la labor de los diseñadores de juego.
- *Nav Mesh Agent*: permite a las arañas moverse por la malla de navegación (ver apartado sobre Navegación en Unity *4.a.vii*).
- *Capsule Collider*: componente físico que permite a las arañas interactuar físicamente con el escenario (ver apartado sobre Física en Unity *4.a.iv*).

ii. Arquitectura por componentes

Los componentes son la base de los objetos y comportamientos en un juego. Son las piezas fundamentales y funcionales de cada *GameObject*. Así mismo, ofrecen un servicio predefinido y pueden comunicarse con otros componentes. En otras palabras, podría decirse que un componente es un objeto definido de acuerdo a unas especificaciones. No importa qué especificación sea, siempre que el objeto adquiera una funcionalidad. Solo cumpliendo correctamente con esa funcionalidad este objeto se convierte en componente y adquiere características como la reutilización.

Un *GameObject* es un contenedor para muchos componentes diferentes. Por defecto, todos los *GameObjects* automáticamente tienen un componente de transformación (*Transform*), que determina la posición en el mundo del *GameObject*, además de su rotación y su escala en los 3 ejes. Sin un componente de transformación, el *GameObject* no tendría un lugar en el proyecto (ver apartado sobre Física en *4.a.iv*).

Se pueden agregar cualquier tipo de componentes a un *GameObject*. Esta es una de las bases de Unity, ya que se pueden crear distintos componentes de forma individual y autoritaria, y únicamente agregarlos a un *GameObject* para dar esa funcionalidad. Se pueden adjuntar cualquier número o combinación de componentes a un solo *GameObject*. Algunos componentes funcionan mejor en combinación con otros.

Entre los distintos tipos de componentes se encuentra algo tan accesible y directo como luces, cámaras, componentes físicos, etc. pero también aparecen componentes que pueden crearse y modificarse por los programadores, como texturas, materiales, *shaders*, *scripts*, etc. que aportan una variabilidad que permite conseguir la funcionalidad y aspecto deseado.

Además, Unity permite activar o desactivar e incluso enlazar o eliminar nuevos componentes en un *GameObject* en tiempo de ejecución, lo que dota al motor de una sobresaliente versatilidad y opciones y simplifica, en gran medida, diseños software que usando únicamente programación orientada a objetos serían mucho más complejos.

iii. El bucle de juego

En este apartado se explica de forma detallada el ciclo de ejecución de Unity[14], es decir, qué métodos se llaman en cada momento, y para qué sirven.

Como puede verse en la Figura 4.8 se distinguen 6 zonas principales en la estructura de Unity:

- Comienzo: las funciones que se encuentran dentro de esta zona se llaman cuando se inicia la escena, y se hace una vez por cada objeto que haya.
- Actualización: cuando se hace la lógica del juego y las iteraciones, animaciones, posiciones de la cámara, etc, hay una serie de eventos diferentes que se pueden usar. Lo más común es realizar todas las tareas dentro de la función *Update()*, aunque también hay otras funciones que se pueden usar.
- Renderizado: las funciones que se encuentran en esta zona se llaman cuando la escena se renderiza.
- Corrutinas: las actualizaciones de las corrutinas normales se ejecutan después de que la función *Update* termine. Una corrutina es una función que puede suspender su ejecución (*yield*) hasta que la instrucción *yield* termine.
- Fin: las funciones que se encuentran en esta zona se llaman en todos los objetos activos de la escena cuando se quiere finalizar el juego.

Entre cada *frame*

- **OnApplicationPause:** Este método es llamado al final de cada frame si se detecta el estado de pausa, normalmente de forma efectiva entre las actualizaciones de *update*. Además, se invocará una nueva actualización extra después del evento *OnApplicationPause* para permitir al juego mostrar gráficos que indiquen el estado de pausa.

Actualización

- **FixedUpdate:** A menudo es llamado con más frecuencia que *Update()*. Se puede llamar varias veces por *frame* si la velocidad de *frame* es baja, y puede no ser llamado si la velocidad de *frame* es alta. Todos los cálculos y actualizaciones de la física ocurren después de esta función. Cuando se aplican cálculos de movimiento dentro de *FixedUpdate()* no se necesita multiplicar los valores por *deltaTime* porque esta función es llamada con un temporizador diferente, que es independiente del tiempo de *frame*.
- **Update:** Es llamado una vez por *frame*. Es la función principal de la actualización del *frame*.

[14] http://docs.unity3d.com/Manual/ExecutionOrder.html

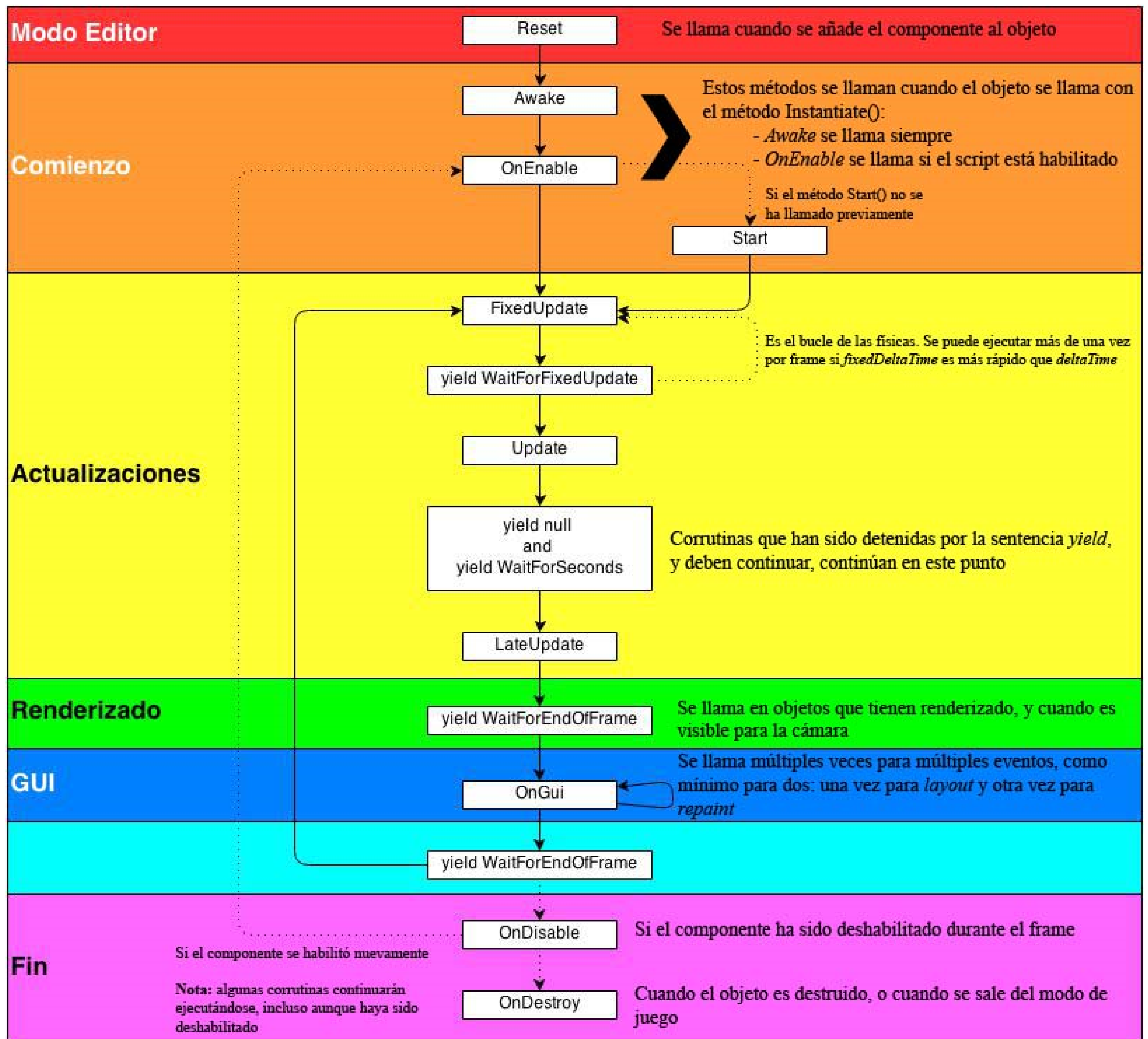


Figura 4.8: Esquema visual del ciclo de juego en Unity.

- **LateUpdate:** Se llama una vez por *frame*, después de que haya terminado la función *Update()*. Algunos cálculos que se realizan en *Update()* se completan cuando *LateUpdate()* empieza. Un uso común para esta función es el seguimiento de la cámara en tercera persona. Si el personaje se mueve y gira dentro de *Update()*, se pueden realizar todos los cálculos del movimiento y la rotación de la cámara en *LateUpdate()*, lo que asegura que el personaje ha completado su movimiento antes de que la cámara siga su posición.

Renderizado

- **OnPreCull:** El *culling* determina qué objetos son visibles para la cámara. Se llama justo antes de que el *culling* tenga lugar.

- **OnBecameVisible/OnBecameInvisible:** Se llama cuando un objeto llega a ser visible/invisible a alguna de las cámaras existentes.
- **OnWillRenderObject:** Se llama una vez para cada cámara si el objeto es visible.
- **OnPreRender:** Se llama antes de que la cámara comience a renderizar la escena.
- **OnRenderObject:** Se llama después de que todo el renderizado regular de la escena esté hecho. Se puede usar la clase *GL* o *Graphics.DrawMeshNow* para dibujar geometrías personales en este punto.
- **OnPostRender:** Se llama cuando la cámara termina de renderizar la escena.
- **OnTenderImage (versión Pro):** Se llama después de que el renderizado de la escena se haya completado para permitir el procesado posterior de las imágenes de la pantalla.
- **OnGUI:** Se llama varias veces por frame en respuesta a eventos GUI. Los eventos *Layout* y *Repaint* se procesan primero, seguido de un evento *Layout* y *keyboard*/*mouse* para cada evento de entrada.
- **OnDrawGizmos:** Se usa para dibujar *Gizmos* en la escena visual para propósitos de visualización.

Corrutina

- **yield:** La corrutina continuará después de que todas las funciones de *Update* hayan sido llamadas en el siguiente frame.
- **yieldWaitForSeconds(2):** Continúa después de un tiempo de retardo, y después de que todas las funciones de *Update* hayan sido llamadas en el frame correspondiente.
- **yieldWaitForFixedUpdate():** Continúa después de que todo el *FixedUpdate* haya sido llamado en todos los scripts.
- **yield WWW:** Continúa después de que una descarga WWW se haya completado.
- **yield StartCoroutine(MyFunc):** Bloquea la corrutina, y espera a que la corrutina *MyFunc* termine primero.

Cuando el objeto es destruido

- **OnDestroy:** Esta función se llama después de la última actualización del *frame* del último objeto existente. El objeto puede ser destruido en respuesta a *Object.Destroy* o al cerrar una escena.

Cuando se sale

- **OnApplicationQuit:** Esta función se llama en todos los objetos del juego antes de salir de la aplicación. En el editor se llama cuando el usuario detiene el *playmode*. En el *web player* se llama cuando el visualizador web se cierra.
- **OnDisable:** Esta función se llama cuando el componente se desactiva o llega a ser inactivo.

Resumen

En cualquier *script* que se use, el orden de ejecución con respecto a lo visto anteriormente es el siguiente:

- Todas las llamadas a *Awake*.
- Todas las llamadas a *Start*.
- **while** (pasos hacia la variable delta time)
    - Todas las funciones *FixedUpdate*.
    - Simulaciones de física.
    - Disparadores *OnEnter/Exit/Stay*.
    - Funciones de colisión *OnEnter/Exit/Stay*.
- La interpolación *Rigidbody* se aplica a *transform.position* y *transform.rotation*
- Eventos *OnMouseDown/OnMouseUp*, etc.
- Todas las funciones *Update*.
- Las animaciones avanzan, se mezclan y se aplican para el *transform.*
- Todas las funciones *LateUpdate*.
- Renderizado

Como se ha dicho anteriormente, hay que tener en cuenta que las corrutinas se ejecutan después de todas las funciones *Update*.

iv. Física

Unity contiene el potente motor de físicas PhysX de NVIDIA. Este apartado se ha resumido en cuatro subapartados, los cuales son de gran importancia para el tratamiento básico de la física en Unity. Se incluye también al final de cada apartado un ejemplo donde se explica la aplicación que se ha hecho de la física al proyecto.

Transform

El componente *Transform* es un atributo de la clase *Object* de la cual heredan todos los objetos de una escena. Su conocimiento es esencial en el uso de esta herramienta y no solo para la física. Sus atributos principales y visibles en el inspector de la escena son *Position*, *Rotation* y *Scale*.

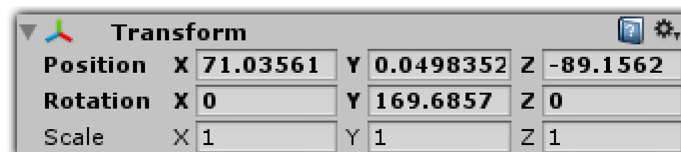


Figura 4.9: Ejemplo del componente *Transform* en el editor de Unity.

Para acceder a cualquiera de estos atributos en C# se hace de la siguiente manera:

- *Object.Transform.Position;*
- *Object.Transform.Rotation;*
- *Object.Transform.Scale;*

Rigidbody

Es un componente que se puede aplicar a un *GameObject* y se usa para el control de su posición mediante simulación de físicas.

Cuando se manipulan los parámetros de un *Rigidbody* se debe trabajar dentro de la función *FixedUpdate*, en vez de *Update* (que es la que se usa por defecto) (ver apartado 4.a.iii). La simulación de físicas se lleva a cabo en cada pasada del código, la función *FixedUpdate* es llamada justo después de cada paso.

En el inspector de Unity se observa de la siguiente manera:

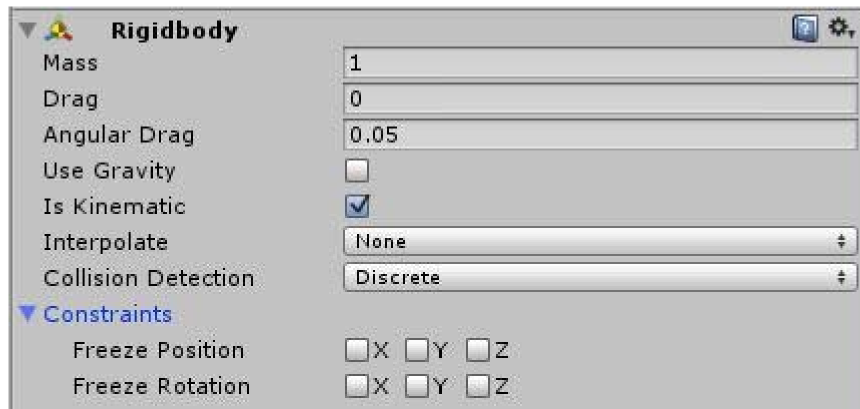


Figura 4.10: Aspecto del componente *Rigidbody* en el inspector de Unity.

Parámetros:

- *Mass*: masa del *GameObject* en cuestión.
- *Drag*: Especifica cuánto afecta la resistencia al aire cuando el objeto se mueve con fuerzas.
  - 0: no hay resistencia al aire.
  - Infinito: el objeto deja de moverse de inmediato.
- *Angular Drag*: Especifica cuánto afecta la resistencia al aire cuando el objeto tuerce.
  - 0: no hay resistencia al aire.
  - Infinito: el objeto gira muy lentamente, pero solo con esto no se puede hacer que deje de girar.
- *Use Gravity*: Influencia o no de la gravedad en el objeto.
- *Is Kinematic*: Es cinemático, es decir, en su movimiento no influyen las fuerzas ni físicas y se limita a la trayectoria en función del tiempo, solo se puede manipular mediante su *Transform*.
- *Interpolate*: Permite suavizar el efecto de ejecutar la física a una velocidad fija. Si el objeto pega sacudidas, seguramente sea por esta variable. Se recomienda activar la interpolación para el personaje principal, pero desactivarlo para todo lo demás.
  - *None*: No se aplica interpolación.
  - *Interpolate*: *Transform* es suavizado basado en el *Transform* del *frame* anterior.
  - *Extrapolate*: *Transform* es suavizado basado en el *Transform* del *frame* siguiente.

- *Collision Detection*: Se usa para que los objetos no pasen rápidamente a través de otros sin haber detectado la colisión.
  - *Discrete*: Es el valor por defecto. Se usa para colisiones normales con otros *colliders*.
  - Continuous: Usa colisión discreta con *colliders* dinámicos (con *Rigidbody*) y usa colisión continua con *static MeshColliders* (sin *rigidbody*).
  - *Continuous Dynamic*: Usa colisión continua contra objetos *con Collision Detection Continuous y Continuous Dynamic*. Usa también colisión continua con *static MeshColliders* (sin *rigidbody*) y con el resto colisión discreta.
- *Constraints*:
  - *Freeze Position*: Detiene el movimiento del *Rigidbody* en las coordenadas X, Y y Z seleccionadas.
  - *Freeze Rotation*: Detiene la rotación del *Rigidbody* en las coordenadas X, Y y Z seleccionadas.

Collider

Es la clase de la que heredan los demás colisionadores básicos, como el *BoxCollider* o el *SphereCollider*. Estos componentes se añaden a los objetos del juego para comprobar cuándo colisionan entre ellos o para evitar que invadan el espacio del otro.

En el caso de que se quiera comprobar que han colisionado, en el componente del *collider* la pestaña *IsTrigger* tiene que estar activada. Además, para que funcione correctamente, uno de los dos componentes que colisionen tiene que tener como componente un *Rigidbody*.

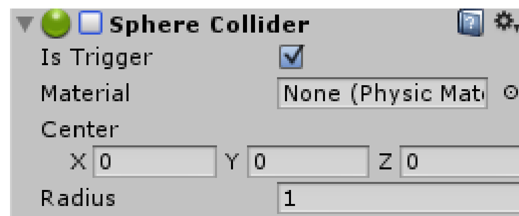


Figura 4.11: Aspecto del componente *Sphere Collider* en el inspector de Unity.

Una vez está activada la pestaña *IsTrigger*, se puede contemplar en los scripts que tenga como componentes los casos en los que colisiona con otro objeto mediante las funciones: ***`OnTriggerEnter()`***, ***`OnTriggerExit()`*** y ***`OnTriggerStay()`***. El área de efecto del colisionador viene determinada por los parámetros que se aprecian en la imagen, en este caso:

- *Center*: Coordenadas locales del colisionador.
- *Radius*: Radio de la esfera.

Las funciones que son invocadas en los eventos de colisión son las siguientes:

- **`OnTriggerEnter(Collider)`**: Esta función es llamada cuando detecta que otro colisionador entra en su área de efecto. El parámetro de entrada es dicho colisionador y no devuelve ningún valor.

- **`OnTriggerStay(Collider)`**: Esta función es llamada en cada paso siempre que haya un colisionador dentro del área de efecto, el cual es el parámetro de entrada de dicha función. Al igual que la anterior, no devuelve ningún valor.
- **`OnTriggerExit(Collider)`**: Por último, esta función es llamada una sola vez cuando el objeto que estaba dentro del colisionador ha salido del área y tampoco devuelve ningún valor.

A continuación se muestra un ejemplo de uso en el proyecto. En concreto corresponde a la colisión de colisionadores que se instancian en los puños de los héroes para comprobar que hemos golpeado otro objeto con colisionador.

```
//Detect all objects that collide with this
void OnTriggerEnter(Collider collisionInfo)
{
    GameObject go = collisionInfo.gameObject;
    nameCollide = go.name;
    if (nameCollide != this.owner.name)
    {
        ControllableCharacter goControllableCharacter =
                            go.GetComponent<ControllableCharacter> ();
        if (goControllableCharacter == null)
            return;
        if (goControllableCharacter.Damage(
                    owner.GetComponent<HeroeController>().attack(P)
           )
        // Damage the enemy and check if it is dead
        {
            owner.GetComponent<HeroeController>().
                  experienceUp(goControllableCharacter.experienceGived);
        }
        hasCollided = true;
        lifeCollide = (int)goControllableCharacter.getLife();
    }
}
```

Código 4.1: Ejemplo de un evento de entrada en un colisionador.

Fuerzas

Se pueden aplicar fuerzas a los distintos *GameObject*. Para ello hay que seguir una serie de pasos:

1. El objeto al que se le va a aplicar la fuerza debe tener un *Rigidbody*, así que si no lo tiene, hay que añadírselo:
    - `object.gameObject.AddComponent<Rigidbody>();`
2. Hay que especificar que le afecten las fuerzas, es decir poner *kinematic* a *false*.
    - `object.rigidbody.isKinematic = false;`
3. Si queremos que le afecte la gravedad, hay que especificarlo:
    - `object.rigidbody.useGravity = true;`
4. Hay que parar el componente *NavMeshAgent*:
    - `object.GetComponent<NavMeshAgent>().Stop(true);`
5. Por último se le añade la fuerza al objeto en cuestión. Para esto, lo mejor es crear un objeto Vector3 que sea la dirección (un vector dirección), normalizarlo y luego aplicarle una fuerza en esa dirección, especificando el tipo de fuerza aplicada (`AddForce(Vector3, ForceMode);`):

```
// Create the direction Vector, from this.position to the object
Vector3 dir = object.transform.position - transform.position;
// normalize it
dir = dir.normalized;
// Give the force values (in the 3 coordinates) and add the type of force
// wanted, impulse in this case.
object.rigidbody.AddForce
(
    new Vector3(dir.x * 0.7f, 3.5f, dir.z * 0.7f),
    ForceMode.Impulse
);
```

Código 4.2: Ejemplo de aplicación de un impulso de fuerza a un *Rigidbody*.

Modos de fuerza:

- *Force*: Añade una fuerza continua al *Rigidbody*, usando su masa.
- *Acceleration*: Añade una aceleración continua al *Rigidbody*, ignorando su masa.
- *Impulse*: Añade una fuerza instantánea al *Rigidbody*, usando su masa.
- *VelocityChange*: Añade un cambio de velocidad instantánea al *Rigidbody*, ignorando su masa.

En el proyecto se ha usado para solucionar muchos problemas. El ejemplo más claro es el de un ataque del orco dando un golpe contra el suelo, aplicando una onda expansiva contra todos los *minions* cercanos. Los *minions* no tienen *Rigidbody* ya que ralentizaría mucho el juego y además no es absolutamente necesario.

En el siguiente recuadro se muestra el uso de las fuerzas sobre el ejemplo comentado. Se puede apreciar la integración que tienen todos los apartados comentados de la física para conseguir una simulación de esta.

```
void OnTriggerEnter(Collider other)
{
    Debug.Log(other.tag);
    if (other.gameObject.name != owner.name)
    {
        if (other.tag == "Player")
        {
            HeroeController script = other.GetComponent<HeroeController>();
            script.Damage(GetDamage(),'M');
        }
        else if (other.tag == "Minion")
        {
            if (!unitList.Contains(other))
            {
                // For damage
                UnitController otherUC = other.GetComponent<UnitController>();
                otherUC.Damage(GetDamage(), 'M');

                // For add a force to the minions so they can fly
                other.gameObject.AddComponent<Rigidbody>();
                other.rigidbody.isKinematic = false;
                other.rigidbody.useGravity = true;

                other.GetComponent<NavMeshAgent>().Stop(true);
                Vector3 dir = other.transform.position - transform.position;
                dir = dir.normalized;

                other.rigidbody.AddForce
                (
```

```
                        new Vector3(dir.x * 2.0f, 5.0f, dir.z * 2.0f),
                        ForceMode.Impulse
                    );
                    otherUC.Fly();
                    unitList.Add(other);
                }
            }
        }
    }
```

Código 4.3: Ejemplo del uso de físicas para "golpear" a una unidad y que salga volando.

En este código la parte interesante se encuentra en la detección sobre si el atributo "tag" del objeto "other" es un *Minion* (`else if (other.tag == "Minion")`). Se comprueba si el objeto colisionado ya ha sido colisionado mediante una lista denominada "unitList", si el objeto ya estaba en la lista no se le aplica ninguna fuerza. En caso contrario se le aplica el daño del ataque, se le añade un componente *RigidBody* necesario para la física y se alteran los atributos de este componente para que la función *AddForce* aplique su efecto de manera efectiva. El propio *minion* al que se le aplica la fuerza es consciente de cuándo tiene que destruir su *rigidbody* mediante la función `Fly()` que actualiza su estado.

v. Materiales

Los materiales[15][16] se utilizan sobre los *GameObjects* y son fundamentales para definir el aspecto visual que muestran los objetos en el juego (ver Figura 4.12). Incluyen texturas y/o colores acompañados de un *shader* que determina cómo se representan estos en una malla o *sprite* (ver apartado siguiente 4.a.v.2), por lo que sin ningún tipo de material no se pueden visualizar estos componentes, y en el juego el *GameObject* aparecería como una simple mancha (normalmente negra o rosa).

Entre las características o propiedades fundamentales y más utilizadas de cualquier material encontramos el propio *shader*, el color principal (*main color*) y la base.

Dentro del editor de Unity, resulta muy sencillo modificar un material en base a los atributos usados en el *shader* del que hace uso. Para las texturas, se permite modificar el *tileado* de ésta (el número de veces que se repite la textura) tanto a lo ancho, como a lo alto y su desplazamiento (u *offset*). Para atributos numéricos del *shader*, se puede establecer su valor, especificando el número, o mediante un *slider*. Y para colores, Unity ofrece varios modos de paletas de color para seleccionar uno.

También se pueden ajustar distintas propiedades de la imagen, como pueden ser la translucidez, la intensidad de la sombra y el brillo.

1. Texturas

La relación entre material y textura (a menudo también llamado mapa) es fundamental para el aspecto del *GameObject* en cuestión. Los materiales se utilizan para colocar texturas en sus *GameObjects*. No se puede agregar una textura directamente sin material, y al hacerlo se creará implícitamente un nuevo material. Así mismo, necesitaremos escoger un *shader*, y obviamente, la textura (o texturas) para mostrar junto a él.

[15] http://docs.unity3d.com/Manual/Materials.html
[16] http://docs.unity3d.com/Manual/class-Material.html

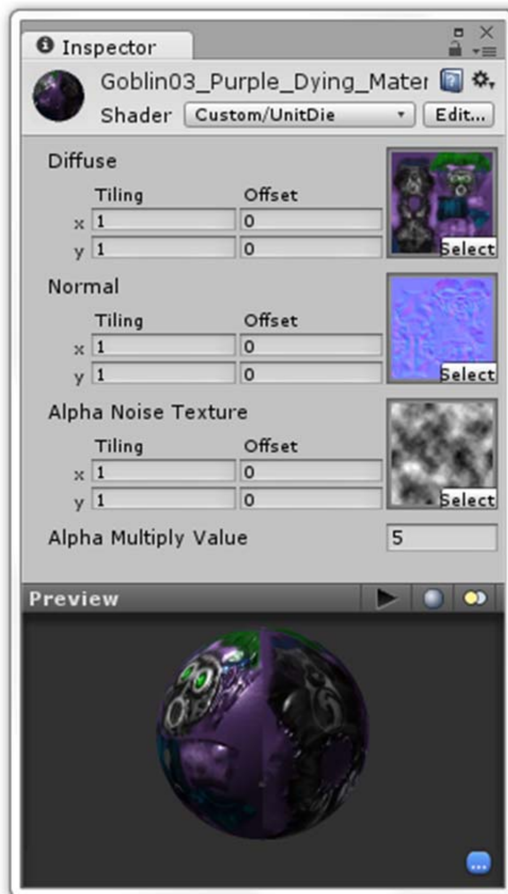


Figura 4.12: Aspecto de un material en el inspector de Unity.

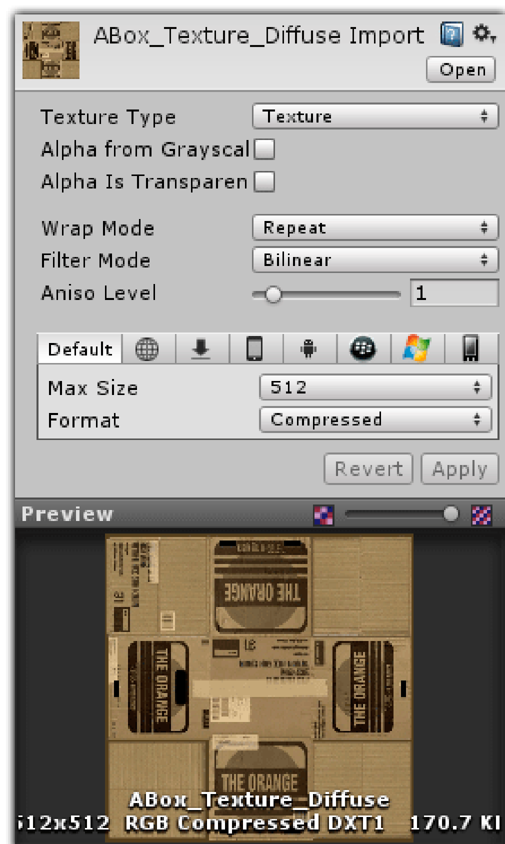


Figura 4.13: Ejemplo de una textura simple de color de 512x512 píxeles de resolución en el inspector de Unity.

Lo primero que se puede apreciar es la elección del tipo de textura, ya que pueden ser de varios tipos. Se pueden escoger texturas que activan canales de color en cierto formato, en forma de *sprite*, para crear reflejos sobre ellas, detalles de su luz y controlar de forma específica los parámetros de la textura, teniendo un total control sobre ella.

Existen otras propiedades, como el modo *Wrap*, en el que se puede configurar cómo se comporta la textura: repeticiones de la textura (azulejos) o bordes de sujeción de la textura que podemos estirar. O la posibilidad de generar un canal de transparencia alfa para obtener distintos valores de la imagen de la luz y de oscuridad de la textura.

También existe un modo de filtro, es decir, cómo se filtra la textura cuando se estira o se encoge por las transformaciones en 3D: la textura se puede ver por bloques de cerca, se puede ver borrosa de cerca, o se puede desdibujar entre los diferentes niveles *mip*. Los niveles *mip* son colecciones de imágenes dehttp://es.wikipedia.org/wiki/Gr%C3%A1ficos_rasterizados mapas de bits que acompañan a una textura para aumentar la velocidad de renderizado. Gracias a esta técnica (*Mipmapping*) se evita en gran medida el *aliasing* que ocurre cuando el número de píxeles que ocupa una textura en pantalla es menor al número de *texels* (resolución real de la textura).

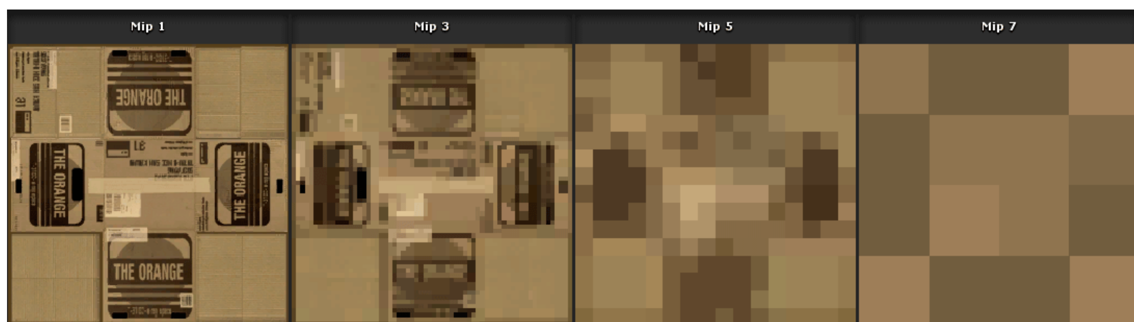


Figura 4.14: Ejemplo de mip maps en una textura.

Por último, se puede aumentar la calidad de la textura durante la visualización de la textura en un ángulo pronunciado. Esta propiedad (*Aniso Level*) es muy útil para las texturas del suelo y de tierra. Así mismo se puede configurar el tamaño máximo de la textura y su formato.

a. Tipos de Texturas

Existen múltiples tipos de texturas (o tipos de mapeado), los más importantes son:

- *Diffuse Map*: también conocido como mapa de color, es la textura que da color al material, reemplaza la superficie de un objeto por la textura 2D asociada.
- *Specular Mapping*: la textura especular varía el brillo y el color de la luz especular (la que “rebota” en el objeto). Esta textura suele estar en tonos grises donde las zonas más blancas representarán zonas más brillantes en el objeto.
- *Normal Mapping*: modifica el sombreado del objeto (modificando la normal de un punto dado de una malla en función del color de la textura en ese punto), dando la sensación de volumen al variar el sombreado en función de la luz que reciba el modelo.

Otros tipos de mapeados muy utilizados son el mapeado de iluminancia (para objetos que se auto-iluminan independientemente de las luces de la escena), el mapeado de transparencia (para marcar zonas transparentes, por ejemplo, para dibujar pelo en un plano) y el mapa de desplazamiento (para alterar la geometría de una malla, e incluso para generar una mayor densidad de geometría por teselación).

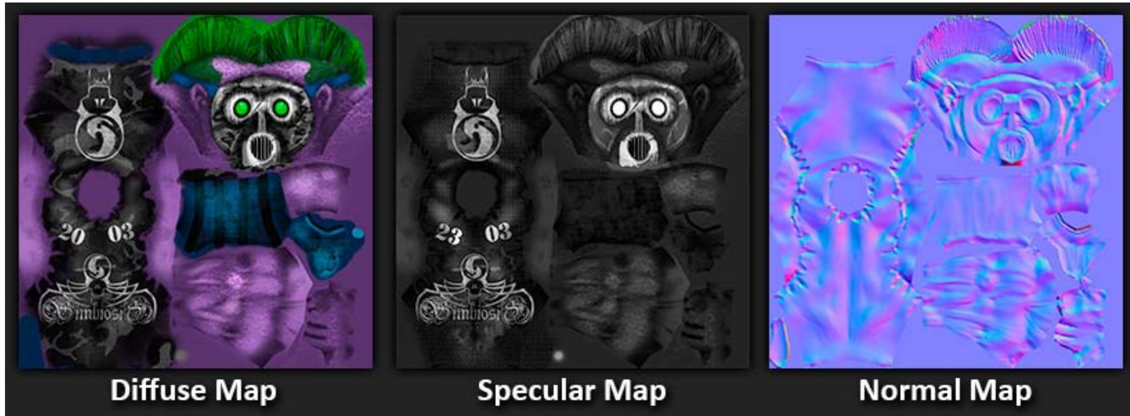


Figura 4.15: Ejemplo de 3 texturas para un mismo material.

Figura 4.16: Las 3 texturas anteriores aplicadas al modelo.

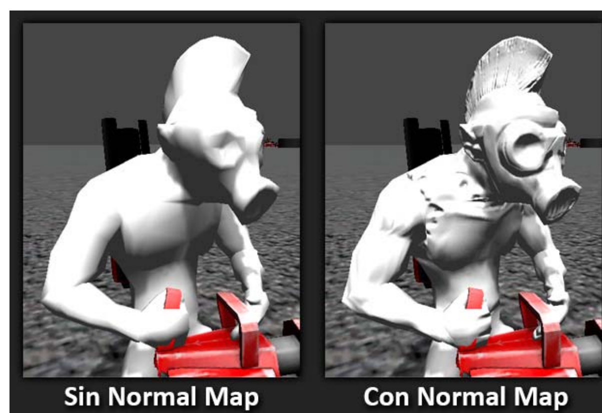


Figura 4.17: Detalle del efecto de un mapa de Normales aplicado a un modelo sin mapa de color.

2. Shaders

El *shader* es el programa gráfico que determina aspectos y características como el sombreado específico que se aplica al material, reflejos, apariencia metálica, imagen especular, brillos suaves, brillos especulares, etc. Además, dependiendo del tipo de *shader*

escogido se tienen en cuenta factores como la curvatura de la superficie, la cantidad de luz, el ángulo de la cámara, etc. para conseguir los reflejos y el matiz exacto que se quiera obtener. Como puede verse, esta propiedad es primordial para conseguir al detalle cualquier tipo de material.

Los materiales y los *shaders* poseen una gran relación entre ellos. Los *shaders* contienen la información que indica el tipo de propiedades a utilizar y los materiales permiten ajustar y aplicar las propiedades mediante el inspector de Unity. Después de crear un material, lo primero que se debe decidir es qué Shader se quiere utilizar en este.

Los *shaders* en Unity[17] se programan en **Cg** (o C for Graphics), un lenguaje de alto nivel desarrollado por *Nvidia* y *Microsoft* muy similar a HLSL de *Microsoft*, por eso muchas veces el lenguaje es nombrado como Cg/HLSL y permite la programación de vertex shaders (programas que se ejecutan en la GPU por vértice de una malla 3D) y pixel shaders (programas que se ejecutan en la GPU por cada pixel de la pantalla).

Una de las características principales de los *shaders* es que definen las propiedades que se muestran en el inspector. Además, cada material contiene los datos de los *sliders* (valor numérico a través de un rango), de los colores y de las texturas, para seleccionarlas. Un dato importante es el hecho de que un *shader* puede ser utilizado en varios materiales, pero un material no puede utilizar varios shaders.

En Unity, existen una gran cantidad de *shaders* por defecto, que se agrupan en varias categorías: *Normal* (texturas opacas), *Transparent* (texturas parcialmente transparentes), *TransparentCutOut* (objetos con áreas completamente opacas y otras completamente transparentes), *Illuminated* (objetos que tienen partes con emisión de luz) y *Reflective* (objetos con texturas opacas que reflejan un entorno). En cada grupo, los shaders están clasificados por complejidad, desde el más simple (*VertexLit*) al más complejo (*Parallax Bumped with Specular*).

Además de los principales *shaders* del juego, existen otras categorías que nos ayudan a conseguir otros fines más específicos, como son: *FX* (iluminación y efectos de agua), *GUI* (visualización de la interfaz gráfica de usuario), *Nature* (árboles y terreno), *Particles* (efectos del sistema de partículas), *Render FX* (efectos de cielo), *Toon* (representación de dibujos animados).

vi. Sistemas de partículas

Los sistemas de partículas consisten en una fuente de emisión de planos texturizados que siempre muestran sus caras hacia la cámara. Las partículas son esencialmente imágenes 2D puestas en el espacio 3D. Con un texturizado apropiado permiten generar efectos como fuego, humo, lluvia, hechizos o de hojas.

A continuación se detalla el funcionamiento de los sistemas de partículas en Unity[18], más adelante, en el apartado 5.c.v, se muestran ejemplos detallados de sistemas de partículas creados para el proyecto.

[17] http://docs.unity3d.com/Manual/Shaders.html
[18] http://docs.unity3d.com/Manual/class-ParticleSystem.html

Un sistema de partículas se compone de tres componentes separados: emisor de partículas, animador de partículas y un procesador de partículas. Se puede utilizar un emisor de partículas y de procesador juntos si se pretende crear partículas estáticas.

Además de existir la opción de utilizar estos tres componentes para crear un sistema de partículas, existe la opción de añadir un componente único que engloba los tres componentes ya mencionados. De este modo, todas sus propiedades equivalen al conjunto de las mismas. Los tres componentes separados reflejan la verdadera estructura de un sistema de partículas.

Por último, también encontramos un componente de colisión, añadible de forma individual si utilizamos los componentes por separado, si no, viene incorporado como propiedad en el sistema de partículas conjunto.

1. Emisor de partículas

El componente correspondiente es *Ellipsoid Particle Emitter*. Es el emisor básico, como se mencionó en el apartado introductorio. Este componente puede definir los límites para las partículas, es decir, dar a las partículas un valor mínimo y máximo a su tamaño, tiempo de vida y número de partículas inicial. Así mismo, permite indicar la velocidad inicial con respecto al mundo en el que se está trabajando y con respecto al sistema de partículas. También se puede definir la variación de velocidad en referencia a los tres ejes.

Además es posible configurar el espacio que ocupan las partículas y la duración de su emisión, cíclica o solo un disparo, así como el rango máximo de ocupación donde se van emitiendo partículas.

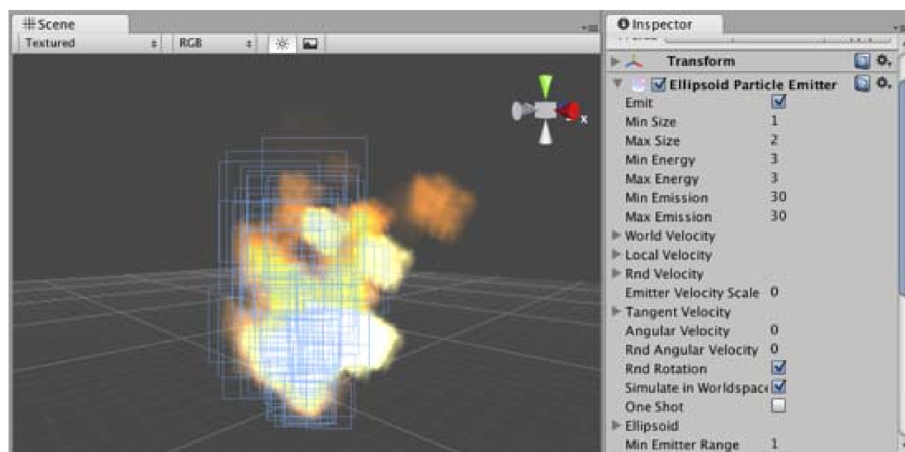


Figura 4.18: Ejemplo de un emisor de partículas en Unity.

2. Animador de partículas

El componente correspondiente es *Particle Animator*. Los animadores de partículas permiten aportar dinamismo en la emisión de partículas. En particular, permiten cambiar o variar de forma automática el color de sus partículas, así como, aplicar fuerzas externas y rotación sobre cada una de ellas. También permiten variar el tamaño de las partículas, lo que acrecienta ese dinamismo mencionado. Como última propiedad, permiten destruir el sistema de partículas cuando acaba de emitir.

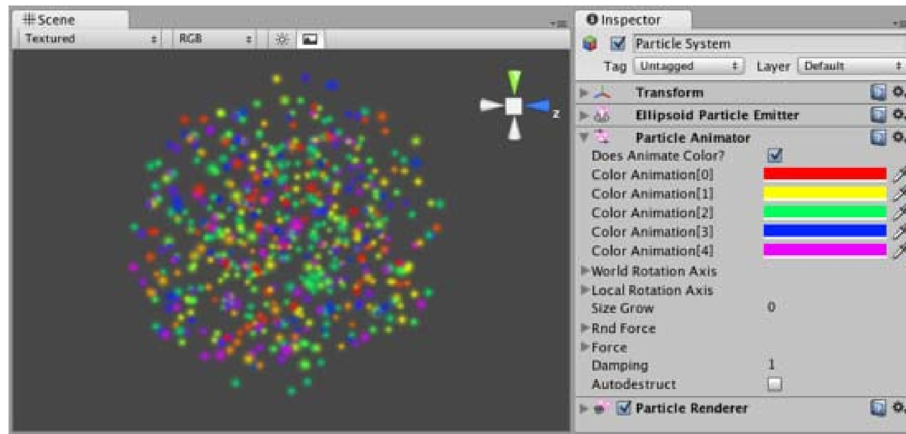


Figura 4.19: Ejemplo del animador de partículas en Unity.

3. Procesador de partículas

El componente correspondiente es *Renderer Animator*. Todo sistema de partículas requiere representadores de partículas que se mostrarán en la pantalla y que favorecen a una aceptación visual adecuada al contexto que se encuentre.

En este apartado se puede añadir un material a cada una de las partículas, así como concretar su integración con el escenario, aplicando sombras y luces dependiendo de su opacidad. También es posible modificar su escala en referencia a su velocidad o su ancho.

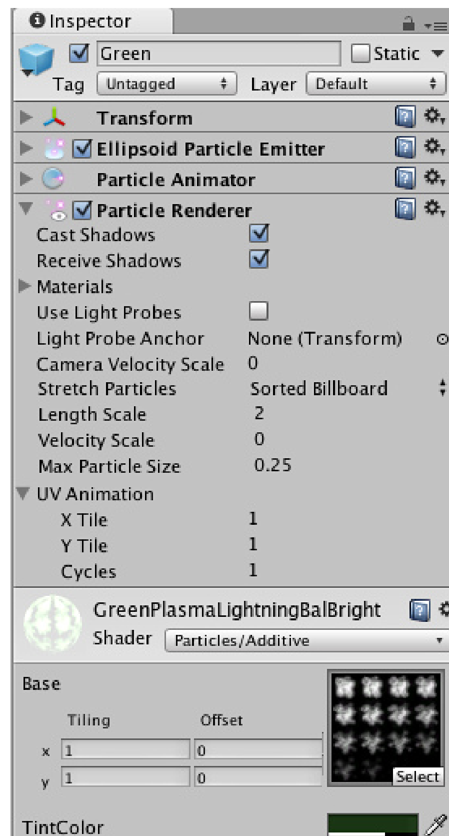


Figura 4.20: Ejemplo del procesador de partículas en Unity.

4. Colisión de partículas

El componente correspondiente es *World Particle Collider*. Aquí se pueden controlar todas las posibles características de un choque entre una partícula y cualquier posible objeto.

Es posible especificar la aceleración o deceleración de una partícula tras una colisión, controlar si la partícula debe ser eliminada porque pierda cierta cantidad de energía y cuál es ese cierto punto, así como especificar con qué puede o no colisionar. La característica más especial con respecto a los otros componentes es la opción de enviar un mensaje si se produce una colisión, la cual puede ser capturada en el script.

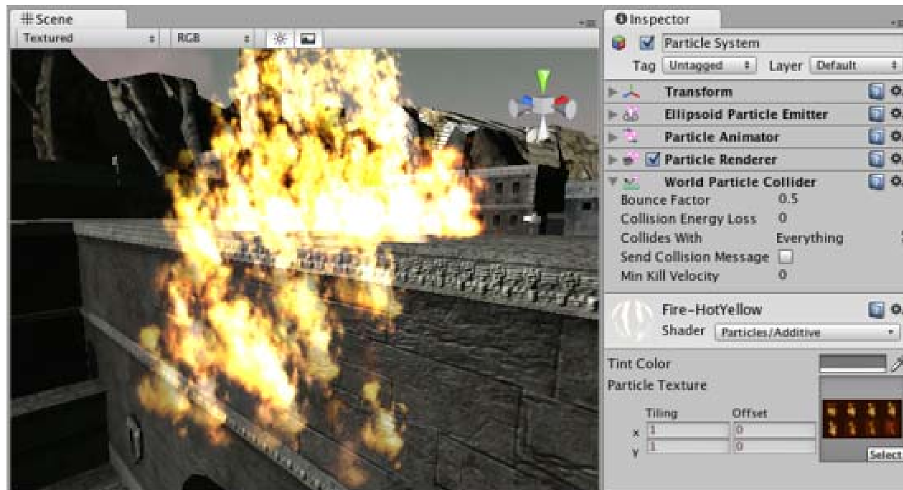


Figura 4.21: Ejemplo del colisionador de partículas en Unity.

vii. Navegación

Para la realización de un videojuego *RTS* es imprescindible un sistema de *pathfinding* (sistema de búsqueda de caminos). Éste usa una malla de navegación (*Navigation mesh*), la cual de ahora en adelante se nombrará de la forma en que más se le conoce: *Navmesh*; además se usarán los términos *Navmesh agent* para obstáculos móviles y *off-mesh link* para enlaces entre diferentes *navmesh*.

Este sistema es absolutamente necesario para muchos tipos de videojuegos, y en especial para videojuegos *RTS*, ya que se desarrollan en escenarios con múltiples agentes móviles o unidades que tienen que desplazarse sorteando obstáculos para llegar desde un origen hasta un destino marcado de forma que no se choquen con ninguno de estos obstáculos y eligiendo un camino determinado (el más corto a ser posible). Para esto se hace necesario desarrollar una lógica de control para que las unidades puedan tomar la decisión correcta en cuanto al camino que tienen que tomar, haciéndose así necesario saber qué puntos son transitables y cuáles son los caminos posibles desde el punto origen al destino.

Unity implementa desde la versión 3.5 un sistema de *pathfinding* en el propio editor del programa[19], simplificando muchísimo la programación de este tipo de sistemas para videojuegos. Al principio se optó por estudiar varios sistemas distintos, y aunque todos coincidian en usar algoritmos A* para la búsqueda de caminos, todos divergían y se complicaban a la hora de generar la malla de navegación (en algunos se generaba en tiempo de ejecución y en otros era necesario establecerla préviamente), sin embargo, tras estudiar la propia de Unity se optó por usar esta, por su simplicidad de uso, lo accesible de su documentación y las posibilidades y tipos de componentes que ofrece.

[19] http://docs.unity3d.com/Manual/Navigation.html

1. Navmesh

La *Navigation mesh* (o malla de navegación) es uno de los elementos más importantes de un sistema de *Pathfinding*, ya que define una manera de representar qué partes son transitables y qué partes no lo son. Ésta, en Unity, es conocida como *NavMesh* y sólo se crea una vez y se guarda, de tal forma, que si se mete algún obstáculo nuevo que no se vaya a mover en el escenario (por ejemplo, un edificio en mitad del mapa), se tiene que crear de nuevo ya que tiene que tener en cuenta el nuevo obstáculo introducido.

El área transitable queda definida por los polígonos de la *navmesh* y las rutas posibles se forman dando saltos entre polígonos adyacentes. Esta *Navmesh* tiene distintos atributos que se pueden ver en la pestaña *Navigation* del editor, este menú tiene 3 secciones: *Object*, *Bake* (el proceso de cálculo de la malla, traducido como cocción), y *Layers*:

Figura 4.22: Aspecto de las opciones de la pestaña *Navigation* en el editor de Unity.

- *Objects*: en este menú se encuentran las opciones sobre navegación que tiene cualquier objeto de la escena.
    - *Navigation Static*: para marcar si el objeto se incluye en la malla.
    - *OffMeshLink Generation*: para indicar si el objeto debe de tenerse en cuenta para generar los saltos entre mallas.
    - *Navigation Layer*: determina a qué capa de la *NavMesh* va a pertenecer el objeto.
        - *Default*: la que se usa por defecto para el área transitable.
        - *Not walkable*: la que se usa como no transitable, es decir, el objeto no se puede transitar por ningún lado.
        - *Jump*: la que se usa para determinar saltos, en el caso de que se pueda.
- *Bake*: aquí se encuentran las opciones de generación de la *NavMesh*.
    - *Radius*: determina cuán cerca puede llegar un *character* o agente a un obstáculo del escenario. Además es la distancia mínima que tiene que haber entre dos obstáculos.
    - *Height*: se refiere a la altura del "techo" del *NavMesh*.

- *Max Slope*: establece el umbral donde una rampa del suelo se convierte en muro y deja de ser transitable.
- *Step Height*: altura máxima en la cual hay una protuberancia o paso en la superficie del suelo que es ignorado por el *character* (cualquier paso superior a esta altura será intransitable en vez de una continuación de la misma zona).

- *Layers*: se puede dotar de una mayor credibilidad a los agentes que se mueven por la *NavMesh* gracias al uso de capas en esta, por ejemplo, partes donde el desplazamiento sea más costoso, más lento, o sea más o menos aceptable bajo determinadas circunstancias. En esta pestaña se pueden seleccionar las capas y seleccionar su configuración.

Funciones:

- *CalculatePath*: calcula un camino entre dos puntos, devuelve true si encuentra el camino:

```
static bool mesh.CalculatePath(transform.position, target.position, -1, path);
```

  - Donde:
    - SourcePosition: posición inicial.
    - Target position: posición final.
    - PassableMask: especifica las capas que entran para calcular la ruta.
    - Path: camino resultante.

- *CalculateTriangulation*: calcula y devuelve una sencilla triangulación del *navmesh*:

```
static NavMeshTriangulation CalculateTriangulation();
```

- *FindClosestEdge*: Localiza el borde más cercano del *navmesh*, devuelve true si encuentra un borde:

```
static bool FindClosestEdge(Vector3 sourcePosition, NavMeshHit hit, int
passableMask);
```

  - Donde:
    - SourcePosition: posición inicial.
    - Hit: contiene las propiedades del hit más cercano.
    - PassableMask: especifica las capas que entran para calcular el borde más próximo.

- GetLayerCost: devuelve el coste de atravesar una capa:

```
static float GetLayerCost(int layer);
```

  - Donde:
    - Layer: índex de la capa

- *SetLayerCost*: determina el coste de una capa.

```
static void SetLayerCost(int layer, float cost);
```

- Donde:
  - *Layer*: índex de la capa.
  - *Cost*: coste de la capa.

- *GetNavMeshLayerFromName*: devuelve el index de una capa, si no la encuentra devuelve -1:

```
static int GetNavMeshLayerFromName(string layerName);
```

- Donde:
  - *LayerName*: nombre de la capa

- *Raycast*: traza un rayo de un punto de origen a otro destino del *navmesh*. Devuelve true si el rayo acaba antes del destino y false en otro caso.

```
static bool Raycast(Vector3 sourcePosition, Vector3 targetPosition, NavMeshHit
hit, int passableMask);
```

- Donde:
  - *SourcePosition*: posición inicial.
  - *Hit*: contiene las propiedades del rayo resultante.
  - *PassableMask*: especifica las capas que entran para calcular el rayo.

- *SamplePosition*: prueba el *navmesh* más cercano a un punto específico

```
static bool SamplePosition(Vector3 sourcePosition, NavMeshHit hit, float
maxDistance, int allowedMask);
```

- Donde:
  - *SourcePosition*: posición inicial.
  - *Hit*: contiene las propiedades de la localización resultante.
  - *maxDistance*: la máxima distancia.
  - *PassableMask*: especifica las capas que entran para calcular la localización.

2. Navmesh Agent

Los *GameObjects* que van a desplazarse por el terreno, por la malla de navegación, necesitan tener un componente *Navmesh Agent* que es el encargado de buscar los caminos de la malla y controlar los movimientos del personaje.

Este componente está fuertemente ligado al *pathfinding* y tiene unos atributos bien definidos:

- *Radius*: radio de la unidad, que difiere con respecto al radio del objeto real.
- *Speed*: velocidad máxima a la que la unidad se puede mover para llegar a su destino.
- *Aceleration*: aceleración máxima de la unidad.

- *Angular* Speed: velocidad de rotación máxima (grados dec/s).
- *Stopping distance*: distancia con el destino a la cual la unidad comenzará a decelerar.
- *Auto Traverse OffMess Link*: para automatizar el movimiento de entrada y salida de *Off-Mesh link* (explicado más adelante).
- *Auto repath*: para adquirir una nueva ruta si la que se está llevando es inválida en un momento determinado.
- *Auto braking*: para hacer el frenado automático.
- *Height*: La altura de la unidad (Sólo usado para modo debug).
- *Base offset*: desplazamiento vertical de la geometría de colisión con respecto a la geometría real.
- *Obstacle avoidance type*: nivel de calidad de evitación de obstáculos. Hay none, low, medium, good y high. Se ha usado el mayor nivel (high).
- *Avoidance priority*: los agentes de menor prioridad que el de este no son tomados en cuenta a la hora de encontrar obstáculos. Cuanta menor sea el número, mayor es la prioridad.
- *Navmesh walkable*: especifica los tipos de capas que la unidad puede atravesar.

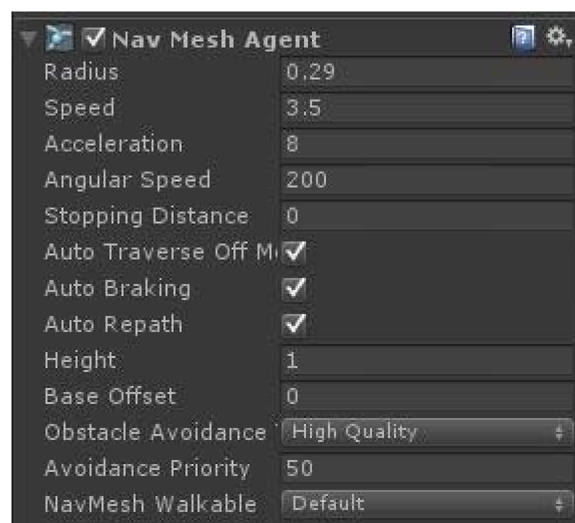


Figura 4.23: Aspecto del componente *Nav Mesh Agent* en el editor de Unity.

Funciones

- *CalculatePath*: calcula el camino en el *navmesh*, devuelve false si no encuentra ninguno:

```
bool CalculatePath(Vector3 targetPosition, NavMeshPath path);
```

  - Donde:
    - *Target position*: posición final.
    - *PassableMask*: especifica las capas que entran para calcular la ruta.

Por ejemplo, de la forma siguiente, primero se coge el componente agente de la unidad, segundo se crea una variable de tipo *NavMeshPath* (que será el camino elegido) y tercero se llama a la función del agente llamada *CalculatePath,* el camino devuelto es *path*:

```
agent = GetComponent<NavMeshAgent>();
NavMeshPath path;
```

```
agent.CalculatePath(target.position, path);
```

- *FindClosestEdge*: localiza el borde más cercano del *Navmesh*, devuelve false si no encuentra ninguno:

```
bool FindClosestEdge(NavMeshHit hit);
```

  - Donde:
    - *Hit*: contiene las propiedades del borde encontrado.
- *Raycast*: lanza un rayo a una posición del *navmesh* sin mover al agente, devuelve true si hay un objeto en la trayectoria del rayo:

```
bool Raycast(Vector3 targetPosition, NavMeshHit hit);
```

  - Donde:
    - Target position: posición final del rayo.
    - Hit: especifica las propiedades del obstáculo encontrado, si es que hay uno.
- *SetPath*: asigna un camino nuevo al agente, devuelve true si se ha asignado correctamente:

```
bool SetPath(NavMeshPath path);
```

- *Stop*: Detiene al agente en medio del recorrido del camino:

```
void Stop(bool stopUpdates);
```

  - Donde:
    - stopUpdates:
      - *True*: el agente se detiene de inmediato, sin afectar el resto del juego.
      - *False*: el *navmesh agent* controla la desaceleración del agente.

3. Nav Mesh Obstacle

Los objetos móviles que no necesitan un sistema de navegación y que tienen que ser un obstáculo a evitar por los personajes que tienen el componente *Navmesh Agent*, tienen el componente *Navmesh Obstacle*. Este consta de un radio (*Radius*) y altura (*Height*) para definir el volumen del *GameObject* en cuestión, umbral de movimiento (*Move Threshold)* que es un umbral de actualización para los objetos que tienen activado *Carve*: un booleano que sirve para que el objeto pueda o no cortar la malla de navegación.

Figura 4.24: Aspecto del componente *Nav Mesh Obstacle* en el editor de Unity.

4. Off-mesh Links

El área transitable no está definida por un sola *navmesh*. Off-mesh links se usa para poder transitar entre distintas mallas.

Hay que poner enlaces entre las mallas de navegación para que las unidades salten de unas a otras. Por ejemplo, si se quiere saltar un río hay que ponerlo entre dos mallas distintas y hacer enlaces entre estas para que la unidad vaya de una a otra. En el ejemplo de más adelante se verá en azul el mapa y en rojo lo que luego será el río.

Se puede hacer de dos formas:

- Autogenerada: por la cual Unity genera los enlaces entre estas dos mallas. Para esto hay que seleccionar el suelo e ir a *Navigation* (se puede encontrar en Window → Navigation):
  - Hay 3 pestañas:
    - *Object*: hay que habilitar la opción *OffMeshLink Generation* para que Unity genere automáticamente los enlaces.

Figura 4.25: Opción *OffMeshLink* Generation de Unity.

    - *Bake*: el apartado importante es *Generated Off Mesh Links*, hay dos opciones:
      - *Drop Height*: altura máxima de caída del agente.
      - *Jump Distance*: distancia máxima de salto.

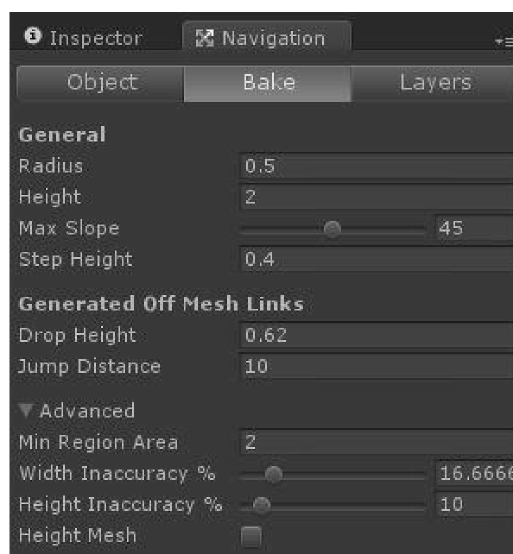


Figura 4.26: Opciones de *bakeado* del *OffMeshLink*.

- *Layers*: aparecen las capas generadas.

Se pulsa en el botón *Bake* (no en la pestaña) y se generarán los links como en la figura siguiente, en este caso hemos seleccionado las dos mallas, por lo que podrá saltar en las dos direcciones:

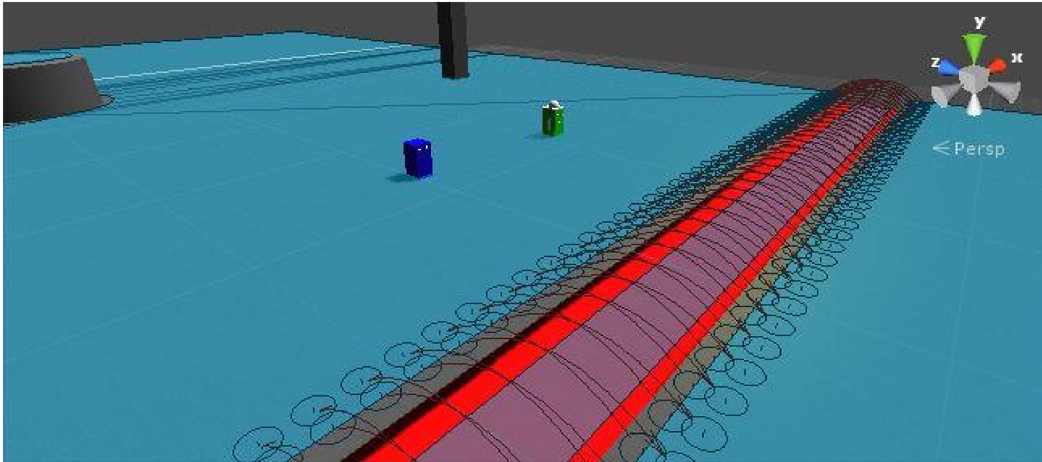


Figura 4.27: Aspecto de varios *OffMeshLink* en una malla de navegación.

- *Manual*: en vez de dejar a Unity que elija los enlaces entre las mallas, se puede hacer manualmente. Hay que crear dos *GameObject*: uno en la posición inicial del salto y otro en la posición final del salto y meter a uno de ellos el componente *Off Mesh Link*.

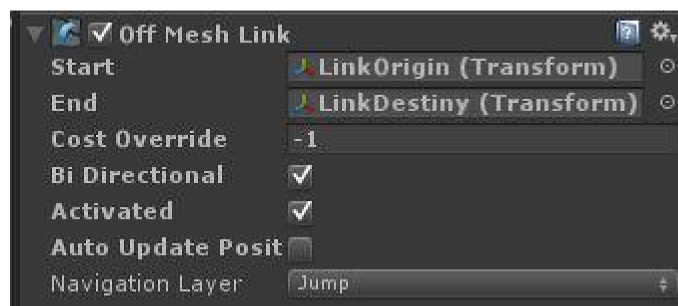


Figura 4.28: Generación manual del *OffMeshLink* en Unity.

- *Start*: objeto origen del salto.
- *End*: objeto destino del salto.
- *Cost Override*: coste para calcular el camino, si es negativo será ignorado.
- *Bi Directional*: si el salto es bidireccional.
- *Activated*: activado estará disponible para el *navmesh pathfinder*.
- *Auto Update Position*: determina si tiene que actualizarse automáticamente cuando se modifican Start/End.
- *Navigation Layer*: determina a qué capa pertenece el *link*.

Después se pulsa, como en la opción autogenerada, en *Navigation* el botón *Bake* y se crea el *link* como puede verse en la Figura 4.29.

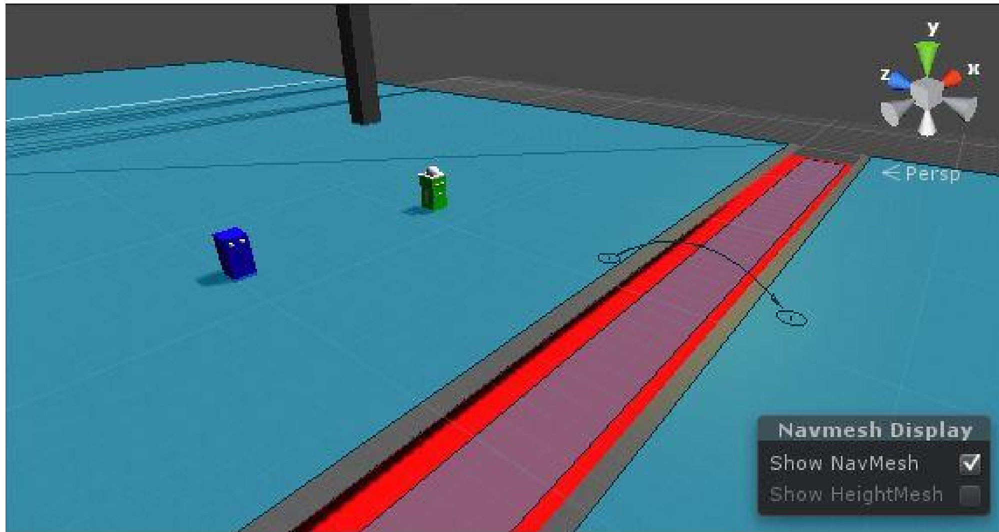


Figura 4.29: Resultado de la creación manual de un *OffMeshLink*.

b. Scripting en C#, Monodevelop y VS 2012

El lenguaje elegido para implementar el proyecto es C#. Dado que dicho lenguaje no se usó durante la carrera por ninguno de los miembros del grupo de trabajo, ha sido necesario un proceso inicial de estudio del lenguaje, y a esto ha sido de gran ayuda el libro *Learning C# by Developing Games with Unity 3D Beginer's Guide* (Norton, 2013). Se trata de un lenguaje orientado a objetos que forma parte del framework .NET de Microsoft y que fue creado para desarrollar aplicaciones de Windows y aplicaciones web. Es una evolución de Microsoft C y Microsoft C++, con mejoras como control de versiones, eventos y liberación de memoria, así como seguridad de tipos.

Hay otros lenguajes que se pueden usar en Unity para implementar los proyectos software, como por ejemplo *UnityScript* y *Boo*. Ahora bien, los más utilizados son C# y *UnityScript* (muy similar a *JavaScript*). Para el presente proyecto se eligió C# por diversas razones:

- Existe mucha más documentación: *UnityScript* es solo un lenguaje de scripting diseñado específicamente para Unity y para encontrar soluciones a algún problema en *UnityScript* hay que buscarlo para *JavaScript*, ya que es un lenguaje muy similar y es mayor la comunidad de este último, pero el código podría no funcionar sin tener que modificarlo. Además, ¿por qué empezar a estudiar un lenguaje de programación limitado para Unity en vez de hacerlo en un verdadero lenguaje de programación de propósito general?
- Mayor flexibilidad para usar scripts en C# que en *UnityScript:* cualquier archivo C# que haya en las carpetas del proyecto de Unity, son accesibles sin necesidad de añadirlos a un *GameObject*. En cambio si es un archivo *UnityScript*, para que funcione debe ser agregado a un *GameObject.*

Las reglas de codificación son específicas en C#: este es un lenguaje estrictamente tipado y mientras se escribe código Unity es capaz de encontrar errores inmediatamente. En cambio *UnityScript* no es un lenguaje estrictamente tipado y se puede estar escribiendo código inválido sin saber que lo es hasta que no se ejecute.

Monodevelop y VS 2012

Monodevelop (Figura 4.30) es un IDE (Entorno de Desarrollo Integrado) multiplataforma y principalmente diseñado para C# y otros lenguajes de .NET. Se sincroniza con Unity de manera que puede usarse para depurar el código, con *breakpoints*, *watches*, etc. Este entorno se usó mucho en las primeras etapas de desarrollo del proyecto, pero más adelante se utilizó Microsoft Visual Studio 2012 (Figura 4.31), ya que es un entorno conocido por todos y se había trabajado mucho con él durante la carrera. Además es un entorno que ofrece las mismas funcionalidades que Monodevelop, aunque tiene la desventaja de no tener depuración por sí solo con Unity. Debido a esto se usa el plugin UnityVS que se explica más adelante (4.c).

Con cualquiera de estos dos entornos se puede inspeccionar el proyecto de Unity, de tal manera que si se añade, elimina o modifica un script en Unity aparecerá automáticamente en la jerarquía de Monodevelop y Visual Studio. Igualmente, si se hace una de estas operaciones en uno de los IDE, Unity se actualizará automáticamente.

c. UnityVS

UnityVS[20] es un plugin para Unity que sirve para conectar el depurador de Microsoft Visual Studio con Unity, de tal forma que se pueda depurar estos scripts y añadir *breakpoints*, *watches*, etc (Figura 4.32). Se integra perfectamente con el proyecto de Unity de manera que cuando se carga un fichero de código desde el proyecto, Unity abre Microsoft Visual Studio configurado con UnityVS de forma automática.

Permite trabajar con los lenguajes de programación de Unity (C#, *UnityScript* y *Boo)*. Además UnityVS mantiene características como el *syntax highlighting* (resaltado de sintaxis), *IntelliSense* (para marcar errores mientras se edita el código) y permite visualizar la jerarquía de los ficheros de código en el proyecto de Unity.

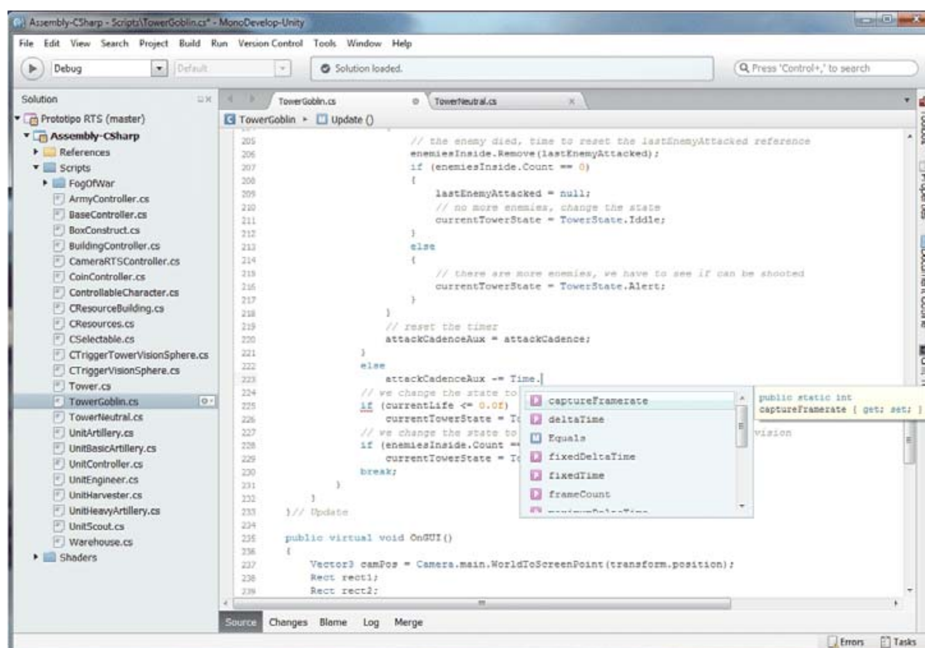


Figura 4.30: Aspecto de la herramienta Monodevelop.

[20] http://unityvs.com/

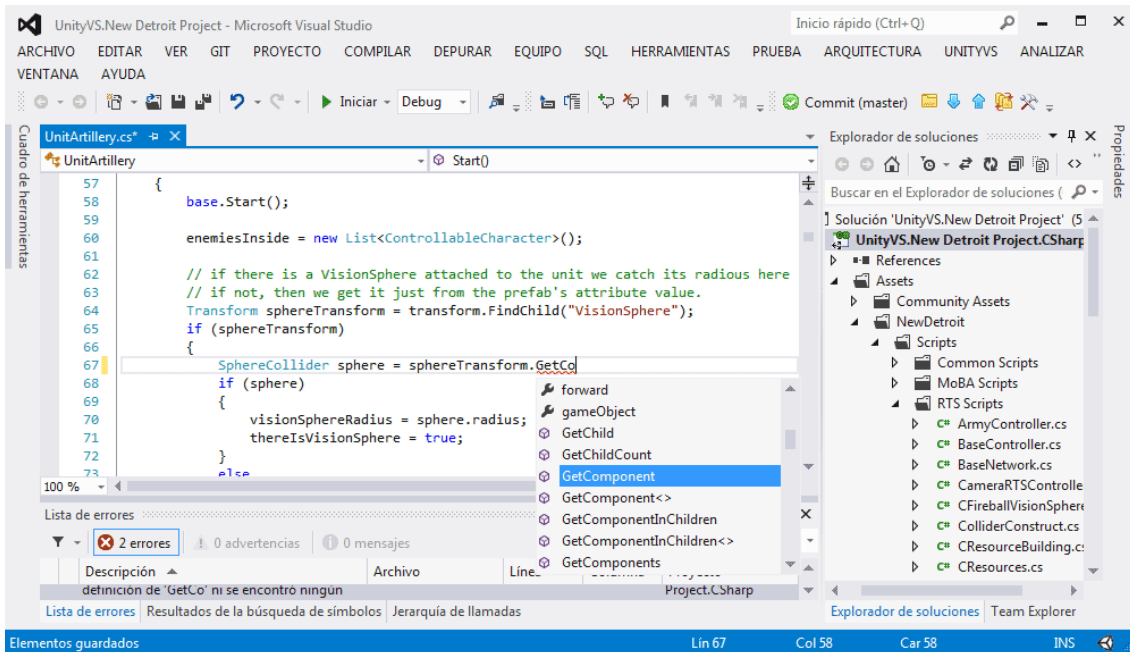


Figura 4.31: Aspecto de la herramienta Microsoft Visual Studio 2012.

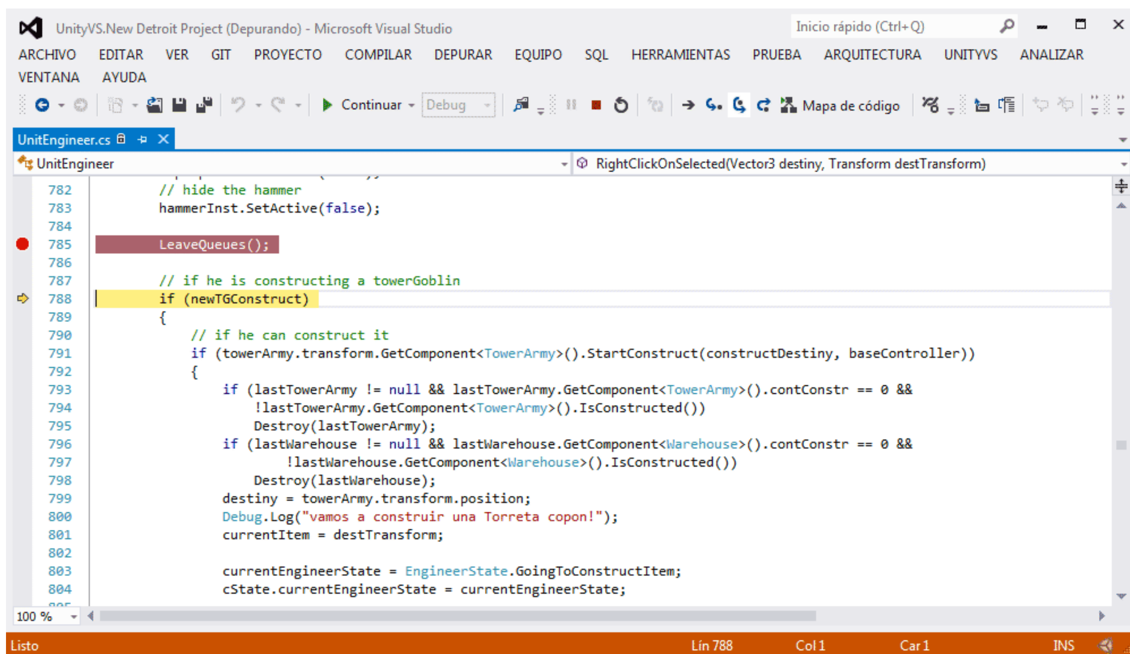


Figura 4.32: Asppecto de Microsoft Visual Studio 2012 con UnityVS integrado.

d. NGUI

NGUI (*Next-Gen UI Kit*) es un sistema de interfaz de usuario basado en notificaciones por eventos y en el principio KISS ("Keep It Simple, Stupid"). Ofrece un código simple, minimalista y fácilmente comprensible para crear tu propia interfaz y con alto rendimentio. Tiene completa integración en Unity y con su inspector de objetos, diseño modular basado en componentes y, lo mas importante, sus componentes solo generan una llamada para dibujarse (esto a la hora de renderizar la escena visible es una mejora muy importante).

Integración en el proyecto

Para integrar la interfaz NGUI en el proyecto se puede descargar de la *Asset Store* e importarla directamente en el proyecto. En este proyecto se ha usado la última verisón ya que esta va a ser incluida en la versión 4.5 Unity. Cuenta con suficientes escenas montadas con ejemplos de uso así como documentación al respecto para construir tu interfaz en pocos pasos.

Una vez importadas las carpetas en nuestro proyecto aparece en la barra de opciones del editor el botón "NGUI" desplegable:

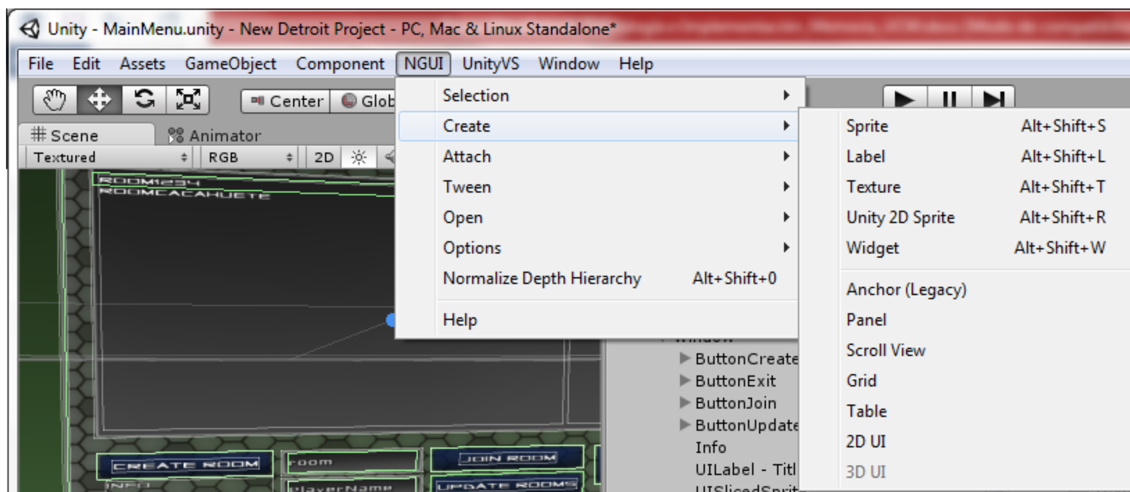


Figura 4.33: Boton NGUI en la barra de opciones.

Como se puede ver en la figura anterior el nuevo boton ofrece opciones para crear objetos funcionales en la escena. Estos objetos incluyen los scripts que se añaden como componentes y cumplen funcionalidades distintas. En la opción de "Attach", también visible en la imagen, se puede añadir estos scripts a objetos que ya estén en la escena. Cabe destacar los siguientes objetos:

- *Sprite*: dibuja en la escena una parte del *Atlas* (objeto que contiene todas las texturas que se van a dibujar en la interfaz en una sola textura).
- *Label*: es usado para imprimir texto por la pantalla. También utiliza las fuentes de letras extrayendo del objeto Atlas.
- *Texture*: se utiliza para extraer una parte del atlas y utilizarlo como textura en la escena.
- *Widget*: su uso mas común es el de integrar la funcionalidad de eventos de la interfaz con los scripts que se le añadan como componentes, de este modo se puede extender su funcionalidad.
- *Panel*: se añade automáticamente a la escena y se encarga de generar la geometría. Tiene una cámara asociada y se encarga de atender todas las llamadas de sus objetos para ser dibujados. Se pueden añadir más de uno para controlar interfaces que no compartan nada entre sí.

Más adelante, en el capítulo 5.c.vi, se hace una breve descripción del uso de esta herramienta para el menú del videojuego.

e. Photon Network

Photon Network es una plataforma de conexión para videojuegos en tiempo real, de acceso gratuito hasta cien conexiones simultáneas. Proporciona una interfaz para su uso y software ejecutable para crear conexiones locales. De necesitar más instancias dispone de unas tarifas por uso durante distintos periodos de tiempo.

Tiene especial interés en este proyecto debido a que la herramienta Unity tiene integración con Photon Network y tutoriales disponibles para cualquier usuario.

Configuración en el proyecto

Para integrar Photon Network en el proyecto de Unity debe descargarse de la Asset Store, que incluye demos con tutoriales que muestran su correcto funcionamiento.

La importación de los archivos de Photon en el entorno Unity incluye una serie de carpetas con la siguiente disposición:

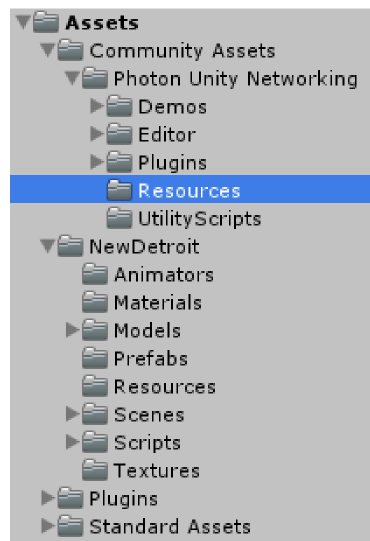


Figura 4.34: Disposición de carpetas de Photon en la pestaña *Hierarchy* de *Unity*.

La carpeta importante es la señalada en la imagen denominada "Resources". Esta carpeta contiene un único elemento el cual nos abre en el inspector las opciones de conexión de Photon.

En la pestaña Host Type hay 3 alternativas de interés: *OfflineMode*, *SelfHosted* y *PhotonCloud*. El primero sirve para que las directivas de Photon no requieran estar conectado para instanciar objetos así como la necesidad de conectarse a una red. El segundo utiliza el *Server Port* y *Server Adress* especificado en el inspector junto con una aplicación externa al editor que gestionará la red de conexión en modo local. La tercera opción conviene configurarla con otro menú distinto, de lo contrario en *server Adress* deberíamos poner "app-eu.exitgamescloud.com" ya que en este caso el servidor a usar es europeo, y en *AppID* se introduce la ID proporcionada por una cuenta que debemos tener en la web de *photon cloud*.

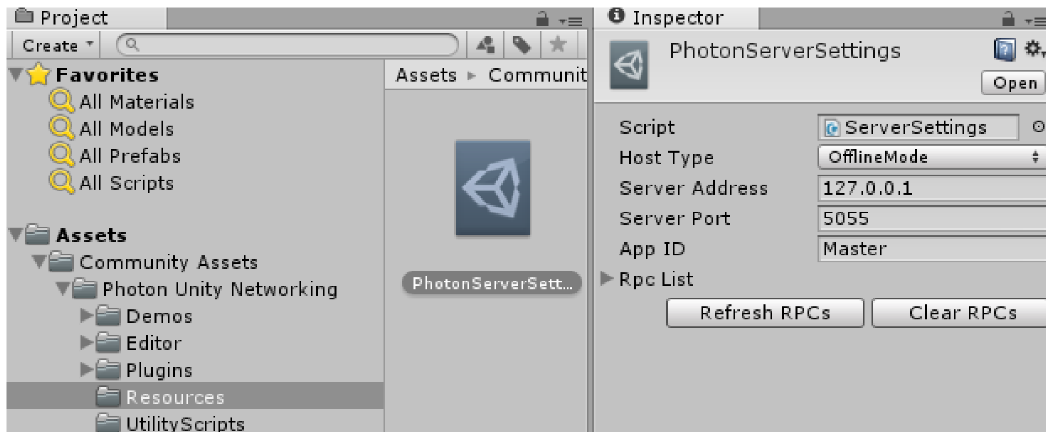


Figura 4.35: Opciones de conexión de Photon en editor de Unity.

- ***PhotonCloud***: Como se ha mencionado antes hay un modo alternativo de configurar el modo *PhotonCloud*. Para ello hay que acceder a la opción *Window -> Photon Unity Networking* en la barra de menús del editor.
- Una vez abierto pulsamos en *setup* y aparecerá una ventana como la siguiente en la que seleccionamos la región deseada e introducimos la *AppID*.

Figura 4.36: Configuración de Photon.

- ***SelfHost***: Para utilizar este modo se necesita la aplicación *ExitGames Photon Server SDK* (gratuita a través de la página oficial[21]). Una vez descargado en la carpeta descomprimida hay que utilizar la versión del sistema operativo dentro de la carpeta “deploy”. Ahí se ejecuta el archivo “PhotonControll.exe”. Esta aplicación aparecerá en la barra de tareas donde se le debe especificar que lance la aplicación *LoadBalancing(my Cloud)* y seleccionar la dirección de IP. Con esto se pueden arrancar varias instancias y conectarlas entre sí.

[21] https://www.exitgames.com/en/OnPremise/Download?sdk=windows8

La carpeta *deploy* debe estar tal y como se descomprime, no debe ser extraído nada de la carpeta.

Uso y componentes

La principal función de *Unity Photon Networking* consiste en pasar objetos de una instancia del juego local a otra instancia que esté conectada a la misma red. Para ello, Photon consta de un componente que se encarga de visualizar el objeto al que vaya acoplado para extraer información de él. Este componente se llama *Photon View*.

Para que extraiga los datos de la escena, es necesario agregarlo a un objeto vacío de la misma y agregárselo al objeto o prefabricado que se quiera instanciar con un flujo de datos. Cuando éste es añadido a un objeto que no sea el vacío de la escena, requiere de un script del cual observará los datos que se quiera transferir. Dicho *script* contiene el siguiente esquema:

```
using UnityEngine;
using System.Collections;

public class UnitNetwork : Photon.MonoBehaviour
{
    // Scripts wich contains the local prefab.
    CSelectable selectableScript;
    Unit unitScript;
    FogOfWarUnit fogOfWarScript;
    NavMeshAgent navMes;

    void Awake ()
    {
        // When this script awakes it takes the scripts asociated to the prefab
        selectableScript = GetComponent<CSelectable>();
        UnitScript = GetComponent<Unit>();
        fogOfWarScript = GetComponent<FogOfWarUnit>();
        navMes = GetComponent<NavMeshAgent>();

        if (photonView.isMine)
        {
            // In this case the instance of the game is local -> active these
            // scripts
            selectableScript.enabled = true;
            engineerScript.enabled = true;
            fogOfWarScript.enabled = true;
            navMes.enabled = true;
        }
        else
        { // In this case the instance is remote, deactive this.
            selectableScript.enabled = false;
            engineerScript.enabled = false;
            fogOfWarScript.enabled = false;
            navMes.enabled = false;
        }
        gameObject.name = gameObject.name + photonView.viewID;
    }

    // This function does the sending and receiving of data
    void OnPhotonSerializeView (PhotonStream stream,
                                PhotonMessageInfo info)
    {
        if (stream.isWriting)
        {
```

```
            // Local player -> send our data, in this case the
            // position and rotation
            stream.SendNext(transform.position);
            stream.SendNext(transform.rotation);
        }
        else
        {
            // Remote player -> receive the data
            correctPlayerPos = (Vector3)stream.ReceiveNext();
            correctPlayerRot = (Quaternion)stream.ReceiveNext();
        }
    }

    private Vector3 correctPlayerPos = Vector3.zero; // linear interpolation
towars this position
    private Quaternion correctPlayerRot = Quaternion.identity; //linear
interpolation towars this rotation

    // this función makes the linear interpolation (lerp) of the position and
    // the rotaiton.
    void Update ()
    {
        if (!photonView.isMine)
        { // update the remote object
            transform.position = Vector3.Lerp(transform.position,
correctPlayerPos, Time.deltaTime * 5);
            transform.rotation = Quaternion.Lerp(transform.rotation,
correctPlayerRot, Time.deltaTime * 5);
        }
    }
}
```

Código 4.4: Ejemplo de *script* de Unity para una unidad instanciada remotamente usando Photon.

Una vez esté listo el script, lo único que queda por hacer, como se ha mencionado antes, es añadirlo al componente *Photon View* del mismo objeto, como puede verse en la Figura 4.37.

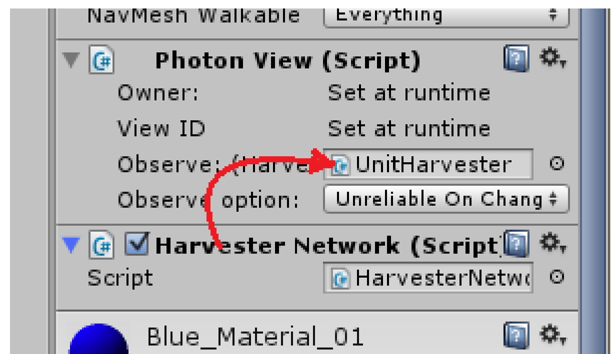


Figura 4.37: *enlace* de los componentes en el inspector de Unity.

Hay que añadirle el componente que contiene el script de network como ilustra la figura anterior, no el script.

Como punto final, para instanciar los prefabricados mediante Photon, estos han de encontrarse en una carpeta del proyecto llamada *Resources*, pues en esas carpetas es donde Photon buscará el nombre del prefabricado que se quiera instanciar mediante la siguiente sentencia:

```
PhotonNetwork.Instantiate("PlayerPrefab", Vector3.Zero, Quaternion.identity, 0);
```

Código 4.5: Instrucción para instanciar un objeto remótamente con Photon.

f. Herramientas de diseño

Para la realización de los distntos componentes gráficos que forman parte de este proyecto han sido necesarias varias herramientas de diseño. Aunque esta parte de la producción no se entiende que forma parte de un proyecto de Sistemas Informáticos, si que se estima interesante apuntar una breve referencia a las dos herramientas más utilizadas en este apartado:

- **Photoshop**: es una herramienta que permite editar y crear gráficos, siendo el software de diseño más popular en la actualidad. En nuestro proyecto, ha resultado de vital importancia a la hora de crear las texturas para los modelos 3D. También ha resultado de gran ayuda a la hora de tratar numerosas imágenes, inlcuidas las aportadas en esta memoria, gracias a las opciones que ofrece para editar y exportar imágenes de distintos formatos.
- **Autodesk 3D Max**: es una herramienta para hacer modelos 3D, pero también permite realizar animaciones con esos modelos. Al igual que el Photoshop, ha sido indispensable a la hora de crear los modelos de las unidades y los héroes con sus animaciones, y de los *assets* del escenario.

5. Análisis y diseño del proyecto

Al inicio del proyecto se desarrollaron 2 prototipos en paralelo para comenzar una implementación modular mientras se avanzaba con el diseño del juego y del software. Se consideró que esto fue una experiencia positiva y necesaria, ya que permitió adquirir experiencia y soltura con Unity a la vez que se desarrollaron scripts que más adelante pasaron a formar parte del proyecto principal. Estos prototitpos son versiones primigenias tanto de la parte RTS como de la parte MOBA. Cuando estas versiones cumplían con una funcionalidad mínima se unieron en un proyecto principal que fue llamado *Project_NewDetroit*.

Organización del proyecto en Unity

Cuando se comenzó a trabajar en el proyecto de Unity que se pretendía que fuera el final (dejando de lado ya los prototipos), se establecieron unos criterios para la organización de archivos y carpetas dentro de éste, para evitar una navegación caótica por el proyecto cuando creciera en complejidad.

Unity establece por defecto una estructura de carpetas muy básica cuando se crea un nuevo proyecto, automáticamente se crea una carpeta llamada *Assets* donde irán todos los archivos que formarán el proyecto. En la creación del proyecto, se fijó una estructura a seguir para mantener un orden lógico. Este orden es básicamente la creación de 3 carpetas, para separar los assets que fueran usados de la comunidad de Unity (*Community Assets*) y los propios que Unity permite importar (*Standard Assets*), de los propios (carpeta *NewDetroit*) creados a lo largo del desarrollo.

Dentro de la carpeta de *assets* propios se establece una subdivisión en carpetas en función del tipo de *asset*.

- ***Assets***: carpeta por defecto en Unity
  - ***Community Assets***: *assets* descargados de Internet, por ejemplo del *Asset Store* de Unity.
    - **Photon Unity Networking**: assets de Photon.
  - **NewDetroit:** assets propios.
    - ***Animators***: archivos de animaciones *.controller.*
    - ***Audio***: *assets* de audio.
      - ***Music***: archivos de música, principalmente *.mp3.*
      - ***Sound Effects**:* archivos de efectos de sonido, normalmente *.wav.*
    - ***Fonts***: archivos de fuentes de texto.
    - ***GUISkins**:* assets gráficos para la interfaz de usuario.
    - ***Materials:*** materiales creados en Unity *.mat.*
    - ***Models:*** modelos importados de *props* y personajes, *obj*, *fbx*, etc.
    - ***Particles:*** prefabricados de sistemas de partículas.
    - ***Prefabs:*** archivos de prefabricados cuya extensión de archivo es *.prefab.*

  - ***Resources:*** prefabricados que se utilizarán para instanciarse por Photon y por búsqueda desde código fuente.
  - ***Scenes:*** ficheros de escenas *.unity.*
  - ***Scripts:*** ficheros de código Fuente *.cs.*
    - ***Common Scripts***: código fuente perteneciente a la parte común del juego.
    - ***MOBA Scripts***: código fuente perteneciente a la parte MOBA del juego.
    - ***RTS Scripts***: código fuente perteneciente a la parte RTS del juego.
  - ***Shaders:*** ficheros de código fuente de *shaders.*
  - ***Textures:*** (texturas importadas de *props* y personajes, *tga*, *png*, etc.
- ***Standard Assets:*** *Assets* propios de *Unity*, por ej. agua, partículas, etc.

Nominación de *Assets*

Además de establecer una jerarquía de carpetas, también se estableció una regla para nombrar los propios *assets*. Facilitando así su búsqueda en la propia herramienta de búsqueda del editor de Unity. Esta regla es básicamente la siguiente:

```
[NombreSeparadoPorMayúsculas]_[fecha-version]_[TipoDeAsset]_[DatosAuxiliares]
```

Sin embargo, para los objetos *prefabricados* se opta por usar una nomenclatura especial, el nombre completo con los espacios que este tenga. Esta decisión se toma por la importancia que tienen este tipo de *assets*, objetos que forman parte del propio juego, y que por lo tanto, aparecen en la pestaña de jerarquía de Unity.

Ejemplo, archivos del héroe *Rob Render:*

- ***Animator***: Assets/NewDetroit/Animators/RobRender_Animator.controller
- **Modelo**: Assets/NewDetroit/Models/RobRender_13,12,19_Model.fbx
- **Texturas**:
  - Assets/NewDetroit/Textures/RobRender_Texture_Diffuse&Specular.tga
  - Assets/NewDetroit/Textures/RobRender_Texture_Normal.tga
- **Material**: Assets/NewDetroit/Materials/RobRender_Material.mat
- ***Prefab***: Assets/NewDetroit/Resources/Rob Render.prefab
- ***Scripts***:
  - Assets/NewDetroit/Scripts/Common Scripts/FogOfWarUnit.cs
  - Assets/NewDetroit/Scripts/MoBA Scripts/RobController.cs
  - Assets/NewDetroit/Scripts/Common Scripts/ThirdPersonController.cs
  - Assets/Community Assets/Photon Unity Networking/Plugins/PhotonNetwork/PhotonView.cs

a. RTS

El desarrollo de la parte RTS del juego ha supuesto un gran esfuerzo y dedicación a lo largo de todo el proceso de desarrollo, tanto por lo característico de su jugabilidad como por los distintos tipos de unidades que se han desarrollado.

Para comenzar, al contrario de lo que suele ocurrir en la mayoría de videojuegos, en un RTS no se tiene un control directo y automático sobre los avatares del juego, sino que múltiples unidades han de tener un comportamiento semi-automático en base a las órdenes dadas por un mismo jugador.

Además el hecho de que existan varios tipos de unidades con diferentes características y habilidades exige de un diseño específico y más complejo de lo que en principio pudiera parecer. A todo esto hay que unir la existencia de varias edificaciones con su propia lógica y características.

Por último, también ha requerido gran esfuerzo el diseño de una lógica de control sobre todo el ejército de un equipo, tanto en el control de las unidades (con su selección, y ejecución de órdenes), como edificios, gestión de recursos y economía del juego.

i. Selección de unidades

Las unidades y muchos edificios son seleccionables, es decir, pueden ser seleccionados para más adelante realizar acciones características de cada uno de ellos. Por ejemplo, si se quiere que una unidad ataque un enemigo en concreto, primero hay que seleccionarla con el botón izquierdo del ratón y después elegir el enemigo a atacar; es decir, para ordenar a una unidad que realice una acción en concreto en un momento determinado primero hay que seleccionarla.

Hay dos tipos de selección: uno individual o simple, en el que solamente se selecciona una unidad y otro múltiple en el que se pueden seleccionar más unidades. Las dos se gestionan inicialmente desde el script *ArmyController.cs* (desde donde se recoge el evento del clic del ratón) (ver apartado 0).

Al seleccionar una unidad, varía levemente su aspecto para que el jugador identifique que la unidad en cuestión ha sido seleccionada (ver figuras Figura 5.1 y Figura 5.2). Este cambio de aspecto puede ser una variación del color en el *outline* (reborde) o un leve cambio en la tonalidad del material del modelo. La lógica que gestiona esto se encuentra en el script *CSelectable*.

Figura 5.1: Aspecto de varias unidades sin seleccionar.

Figura 5.2: Aspecto de varias unidades seleccionadas.

1. Componente CSelectable

Este componente es un script que se le añade a una unidad o edificio para agregarle la funcionalidad necesaria para que sea seleccionable; más adelante, cuando en *ArmyController.cs* se están seleccionando unidades, lo que se mira entre otras cosas, es si tiene este componente *CSelectable*.

Lo más reseñable de los atributos del script son los necesarios para cambiar el color del modelo (el del equipo, cada uno tendrá uno diferente) del *outline* o de todo el modelo en cuestión y el grosor de las unidades que tengan *outline* (puede haber unidades que no lo tengan). Estos son varios atributos de tipo *Color* y una referencia al modelo. Además se tiene una referencia al componente *UnitController* que lo contiene. También contiene un atributo booleano que identifica si la unidad está o no seleccionada: *selected*, y una referencia a un clip de sonido, que la unidad reproducirá cada vez que se seleccione: *sfxSelected* de tipo *AudioClip*.

A la hora de instanciar un objeto de este tipo se llama al método `Awake()` (Código 5.1) de *CSelectable.cs*, donde lo primero que se comprueba es si el modelo donde ha sido añadido este componente es un edificio o una unidad. Si es una unidad de un ejército, se sabe que tienen *outline* que cambiará de color al seleccionarse, se recoge el grosor original del *outline* (*outlineWidth*) por si se desea variar más adelante y también se recoge el número de materiales que tenga asociado el modelo (hay unidades que pueden tener varios materiales, por ejemplo, un material para el cuerpo y otro material para un arma). Si en cambio es un edificio, variará el color del material. En cualquier caso, se resetea el color principal en base al equipo al que pertenezca la unidad, esto se hace accediendo a una matriz estática que guarda los colores asociados a todos los equipos que pueden participar en una partida (`teamColor = TeamsColors.colors[ GetComponent<CTeam>().teamColorIndex ];`). Además se guarda el tipo en *selectType*, y se obtiene la referencia al componente *UnitController*.

```
void Awake ()
{
    model = transform.FindChild("Model");
    if (model)
    {
        selectType = 0;
        // if there is a "model" children in the object it is a unit
        numberOfMaterials = model.renderer.materials.Length;
        outlineWidth = model.renderer.material.GetFloat("_OutlineWidth");
        for (int i = 0; i < numberOfMaterials; i++)
            model.renderer.materials[i].SetColor("_OutlineColor", Color.black);
    }
    else
```

```
    {
        // if not, it is a building
        if (renderer.material.HasProperty("_DiffuseColor"))
        {
            selectType = 1;
            origColor = renderer.material.GetColor("_DiffuseColor");
        }
        else if (renderer.material.HasProperty("_AlphaColor"))
        {
            // is a buildable (construible) building
            selectType = 1;
            origColor = Color.white;
        }
        else
        {
            selectType = 2;
            origColor = renderer.material.color;
        }
    }
    teamColor = TeamsColors.colors[GetComponent<CTeam>().teamColorIndex];
    selected = false;
    unitReference = GetComponent<UnitController>();
}
```

Código 5.1: Método *Awake* de la clase *CSelectable*.

La lógica para cambiar el aspecto al seleccionarse y deseleccionarse, se encuentra en los métodos `SetSelected()` y `SetDeselected()` (Código 5.2). En *SetSelected* primero se comprueba que el objeto no está ya seleccionado, sino, se marca como seleccionado y en función del tipo de objeto que sea (*selectType*), unidad o edificio, cambia el color del *outline* (`model.renderer.materials[i]. SetColor ("_OutlineColor", teamColor);`) en el primer caso, y el color del material en el segundo (`renderer.material. SetColor ("_DiffuseColor", teamColor);`). Por último reproduce el efecto de audio del atributo *sfxSelected*. Para deseleccionar se ejecuta el método *SetDeselect* con una lógica muy parecida al anterior pero que en vez de cambiar el color por el del equipo (*teamColor*) lo cambia por el original (*origColor*).

```
public void SetSelected ()
{
    selected = true;
    switch (selectType)
    {
        case 0:
            for (int i = 0; i < numberOfMaterials; i++)
                model.renderer.materials[i].SetColor("_OutlineColor",
                                                     teamColor);
            break;
        case 1:
            this.renderer.material.SetColor("_DiffuseColor", teamColor);
            break;
        case 2:
            this.renderer.material.color = new Color(teamColor.r + 0.4f,
                                      teamColor.g + 0.4f, teamColor.b + 0.4f);
            break;
    }

    if (unitReference)
        unitReference.isSelected = true;
}
```

```
public void SetDeselect ()
{
    selected = false;
    switch (selectType)
    {
        case 0:
            for (int i = 0; i < numberOfMaterials; i++)
                model.renderer.materials[i].SetColor("_OutlineColor",
                                                     Color.black);
            break;
        case 1:
            this.renderer.material.SetColor("_DiffuseColor", origColor);
            break;
        case 2:
            this.renderer.material.color = origColor;
            break;
    }

    if (unitReference)
        unitReference.isSelected = false;
}
```
Código 5.2: Métodos de selección y deselección de la clase *CSelectable*.

2. Selección simple

Como ya se ha comentado, tanto la lógica de selección simple como de selección múltiple se realiza en la clase *ArmyController*. En el caso de la selección simple (ver Figura 5.3) lo primero que se mira es si se ha pulsado el botón izquierdo del ratón, esto se controla con el evento *Input.GetMouseButtonDown(0)* y se guarda la posición de la pulsación en un atributo de la clase (`(Vector2)lastClick = Input.mousePosition`) para más adelante, cuando se levanta el botón izquierdo del ratón, controlado con el evento `Input.GetMouseButtonUp(0)`, ver si la posición del ratón es la misma (si no se ha arrastrado *marcando un rectángulo*), además se pone a *true* la variable booleana `mouseButtonPreshed` y se guarda `lastCrowdAngle = 0`.

Figura 5.3: Ejemplo de selección simple.

Se lanza un rayo desde la pantalla hasta la posición pulsada para ver con que choca, es decir, sobre dónde se ha hecho clic. Al recoger la información que indica sobre qué ha chocado este rayo, se comprueba que este objeto tenga un componente de tipo *CSelectable*, en este caso, se mira si es una unidad o un edificio y si es del mismo equipo que el del jugador que está seleccionando. En el caso de que sea una unidad, hay que comprobar que no estén pulsadas las teclas *control* ni *shift*, ni sea un doble clic, ya que forman parte de la selección múltiple. Si es así (ver Código 5.3: Selección simple.), hay que deseleccionar todas las unidades que estén seleccionadas, se añade la unidad seleccionada a la lista *unitSelectedList* (atributo de la clase

que contiene las referencias a las unidades seleccionadas) y se llama al método *SetSelected* del componente *CSelectable*. También se guarda el momento en el que se realiza la pulsación para controlar más adelante si la siguiente pulsación pertenece a un doble clic o no. En caso de que sea un edificio y del mismo equipo, tienen que deseleccionarse todas las unidades, comprobar qué edificio se ha pulsado y marcarlo como seleccionado (de forma similar a las unidades).

```
myRay = Camera.main.ScreenPointToRay(Input.mousePosition);
if (Physics.Raycast(myRay, out myHit, 1000f, layerMask))
{
    CSelectable objSel = myHit.transform.GetComponent<CSelectable>();
    if (objSel != null)
    {
        // Check if the object is an army unit
        UnitController unitCont = objSel.GetComponent<UnitController>();
        // Check if unit is in the same team than this ArmyController
        if (unitCont != null && unitCont.GetTeamNumber() == teamNumber)
        {
            // left control is not preshed (not multi-selection)
            // deselect the previous selected units
            if ( (Input.GetKey(KeyCode.LeftControl) ||
                  Input.GetKey(KeyCode.RightControl) ||
                  doubleClickStart == -1.0f) &&
                  !Input.GetKey(KeyCode.LeftShift) &&
                  !Input.GetKey(KeyCode.RightShift))
            {
                DeselectAll();
            }
        ...
        if ( !Input.GetKey(KeyCode.LeftShift) &&
             !Input.GetKey(KeyCode.RightShift))
            DeselectAll();
        unitDoubleClick = u;
        doubleClickStart = Time.time;
        // Insert the unit in the selected list
        unitSelectedList.Add(u);
        // mark the unit as selected
        u.GetComponent<CSelectable>().SetSelected();
    }
}
...
```

Código 5.3: Selección simple.

3. Selección múltiple

Como se comenta en la memoria sobre *Ingeniería Software y Diseño del Videojuego* de este proyecto (Cuesta Boluda, Cuesta Boluda, & Rodríguez-Osorio Jiménez, 2014) hay distintos modos de selección múltiple. Para todos ellos primero se comprueban las circunstancias de cómo y dónde se ha hecho clic, para poder diferenciar entre estos métodos de selección múltiple.

a. Selección múltiple absoluta

Este tipo de selección ocurre cuando se ha hecho clic y se mantiene el botón presionado mientras se arrastra el ratón, dibujando un recuadro en pantalla. Este evento se captura gracias a guardar la posición del ratón cuando se presiona por primera vez el botón del ratón. Mientras se arrastra el ratón con el botón presionado, se va comprobando si cada una de las unidades del ejército están dentro de este recuadro (ver Figura 5.4).

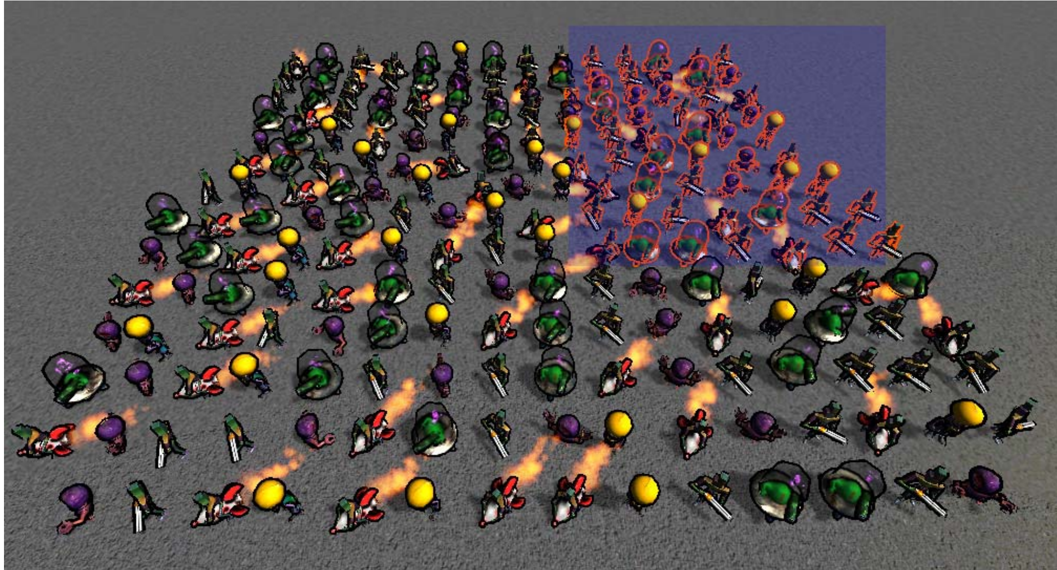


Figura 5.4: Ejemplo de selección múliple por cuadrado.

También se comprueba si es la primera vez que se entra, se deseleccionan todas las unidades (a no ser que esté alguna tecla *shift* pulsada, en su caso los cuadrados de selección se acumulan) y se crea el primer punto de selección, estos puntos se guardan en un *array* de cuatro posiciones, que representan cada uno de los cuatro puntos del cuadrado que se "dibuja" en pantalla.

Para encontrar las unidades que están dentro de este cuadrado de selección se han implementado dos alternativas diferentes en los métodos CreatingSquare y CreatingSquare2, que pasan a detallarse a continuación.

CreatingSquare

El primer método consiste en proyectar las cuatro esquinas del cuadrado que se dibuja en pantalla a través de cuatro rayos que salen de la cámara del juego hacia el mundo, a continuación se recogen las posiciones 3D donde han chocado estos rayos con el suelo del nivel (ver Código 5.4), y por último se comprueba si las posiciones de cada unidad están dentro de este área determinada por el cuadrilátero proyectado en el mundo del juego (cuyas esquinas se almacenan en un array de 4 posiciones llamado *squareSelectionPointsProyected*) (ver Figura 5.5).

```
// 4 rays are launched in order to capture the 4 vertices of the quadrilateral
for (int i = 0; i < squareSelectionPointsProyected.Length; i++)
{
   // ray:
   myRay = Camera.main.ScreenPointToRay(squareSelectionPointsScreen[i]);
   // pick the position
   if (Physics.Raycast(myRay, out myHit, 1000f, layerMask))
       squareSelectionPointsProyected[i] = myHit.point;
}
```

Código 5.4: Lanzamiento de rayos para calcular el cuadrilátero proyectado en el mundo.

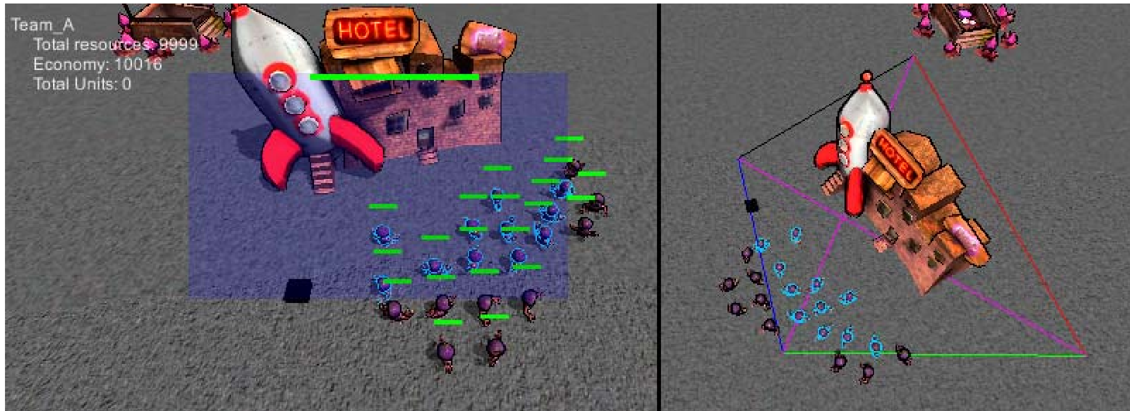


Figura 5.5: ejemplo de la proyección de los puntos del rectángulo de pantalla (izquierda) en el cuadrilátero del mundo (derecha).

Tras capturar las coordenadas del cuadrilátero proyectado, se calculan las coordenadas del centro de este cuadrilátero a través del cálculo de la intersección de las dos rectas formadas por las coordenadas pares e impares de dicho cuadrilátero para poder así calcular más adelante las distancias de las unidades al este centro. Por último, también se calcula el radio de la circunferencia circunscrita del cuadrilátero.

La última parte del algoritmo consiste en determinar una por una, si cada unidad está dentro de este cuadrilátero (ver Código 5.5). Para simplificar este costoso bucle, se van descartando casos por simplicidad de cálculo. En primer lugar se descartan los casos más simples a través del cálculo de las distancias en los ejes x e y, con el centro del cuadrilátero, si esta es mayor que el radio del propio cuadrilátero ya se sabe que están fuera de su área, así, con dos restas y dos comprobaciones se descarta la mayoría de casos. A continuación se calcula la distancia euclídea entre la posición de la unidad y el centro del cuadrilátero (este cálculo exige dos multiplicaciones y una raíz cuadrada), si esta distancia es menor al radio significa que la unidad esta fuera.

A continuación viene el último y más costoso caso que consiste en medir la distancia de la unidad a cada uno de los cuatro segmentos que componen el cuadrilátero, si las cuatro distancias son positivas (o negativas si los segmentos se entienden en sentido contra horario) significa que el punto está dentro del polígono, en este caso, se marca a la unidad como seleccionada y se introduce en la lista de unidades seleccionadas. Este cálculo necesita varias multiplicaciones, una división y una raíz cuadrada[22] (ver Código 5.6), sin embargo, lo único necesario es el signo resultante de esta fórmula, por lo que se puede simplificar el denominador y así eliminar el cálculo de la división y de la raíz cuadrada (ver Código 5.7).

```
// check one y one if the unit is inside the quadrilateral
int count = unitList.Count;
for (int i = 0; i < count; i++)
{
    // 2D position of the unit in the world floor
    Vector2 unitPos = new Vector2(unitList[i].transform.position.x,
        unitList[i].transform.position.z);

    // 1st: Manhattan distance
    if ( Mathf.Abs(unitPos.x - squareCenter.x) <= radius &&
         Mathf.Abs(unitPos.y - squareCenter.y) <= radius )
```

[22] http://geomalgorithms.com/a02-_lines.html

```
    {
        // 2nd: euclidean distance
        // distance between the unit and the center of the selection square
        float dist = Vector2.Distance(unitPos, squareCenter);

        if (dist < radius)
        {
            // 3rd: distance unit to segments of the square
            int contNeg = 0, contPos = 0; float d = 0.0f;
            for (int j = 0; j < 4; j++)
            {
                /*d = DistancePointToSegment(
                    new Vector2(squareSelectionPointsProyected[j].x,
                                squareSelectionPointsProyected[j].z),
                    new Vector2(squareSelectionPointsProyected[(j + 1) % 4].x,
                                squareSelectionPointsProyected[(j + 1) % 4].z),
                    unitPos
                );*/
                // this alternative is less expensive:
                d = SignPointToSegment
                (
                    squareSelectionPointsProyected[  j          ],
                    squareSelectionPointsProyected[ (j + 1) % 4 ],
                    unitPos
                );
                if (d < 0)
                    contNeg++;
                else
                    contPos++;
            }
            if (contNeg == 4 || contPos == 4)
            {
                // the unit is inside the selection square
                // we add it to the selection list
                if (!unitSelectedList.Contains(unitList[i]))
                    unitSelectedList.Add(unitList[i]);
                // and mark it as selected
                unitList[i].GetComponent<CSelectable>().SetSelected();
            }
            else
            {
                // delete the unit from the selection list
                unitSelectedList.Remove(unitList[i]);
                // and mark it as deselected
                unitList[i].GetComponent<CSelectable>().SetDeselect();
            }
        }
    }
}
```

Código 5.5: Bucle que determina si una unidad está dentro del cuadrilátero.

```
// returns the distance of the point to the segment
// A is the first point of the segment
// B is the second point of the segment
// p is the point
private float DistancePointToSegment (Vector2 A, Vector2 B, Vector2 p)
{
    return (((B.x - A.x) * (A.y - p.y) - (A.x - p.x) * (B.y - A.y)) /
        (float)(Mathf.Sqrt( (B.x - A.x) * (B.x - A.x) +
                            (B.y - A.y) * (B.y - A.y) )));
}
```

Código 5.6: Cálculo de la distancia de un punto a un segmento determinado por 2 puntos.

```
// calculating the sign of the distance from a point to a segment
private float SignPointToSegment (Vector3 A, Vector3 B, Vector2 p)
{
    return ((B.x - A.x) * (A.z - p.y) - (A.x - p.x) * (B.z - A.z));
}
```

Código 5.7: Cálculo simplificado del signo de la distancia de un punto a un segmento determinado por 2 puntos.

CreatingSquare2

El segundo método hace uso de la función *WorldToScreenPoint* de la clase *Camera*[23], que devuelve la posición en coordenadas de pantalla de cualquier objeto dada su posición en el mundo. Todas las unidades tienen un atributo de tipo Vector3 que guarda esta posición en pantalla y que se actualiza en cada ciclo de juego (ver Código 5.8). De esta manera, comprobar que esta posición devuelta está dentro de un rectángulo dibujado en pantalla simplifica en gran medida el primer algoritmo planteado y se evita el uso de rayos y otros cálculos presumiblemente muy costosos (ver Código 5.9).

```
// the screen position of the character
public Vector3 screenPosition;

public virtual void Update ()
{
    screenPosition = Camera.main.WorldToScreenPoint(transform.position);
}
```

Código 5.8: Atributo *screenPosition* de la clase *ControllableCharacter*.

```
// alternative version which avoids the (presumably expensive) use of rays
private void CreatingSquare2 ()
{
    // update the positions of the vertices of the screen square
    UpdateSelectionPointScreen();

    Vector3 unitScreenPos;
    int count = unitList.Count;
    for (int i = 0; i < count; i++)
    {
        // capture the screen position of the unit
        unitScreenPos =
               unitList[i].GetComponent<ControllableCharacter>().screenPosition;
        // check if the position if inside the square we are creating
        if (unitScreenPos.x >= squareSelectionPointsScreen[0].x &&
            unitScreenPos.y <= squareSelectionPointsScreen[0].y &&
            unitScreenPos.x <= squareSelectionPointsScreen[2].x &&
            unitScreenPos.y >= squareSelectionPointsScreen[2].y)
            // select the unit
        {
            // we add it to the selection list
            if (!unitSelectedList.Contains(unitList[i]))
                unitSelectedList.Add(unitList[i]);
            // and mark it as selected
            unitList[i].GetComponent<CSelectable>().SetSelected();
        }
        else
        {
            // delete the unit from the selection list
            if ( !Input.GetKey(KeyCode.LeftShift) &&
                 !Input.GetKey(KeyCode.RightShift) )
            {
```

[23] http://docs.unity3d.com/ScriptReference/Camera.WorldToScreenPoint.html

```
                if (!unitSelectedList.Contains(unitList[i]))
                    unitSelectedList.Remove(unitList[i]);
                // and mark it as deselected
                unitList[i].GetComponent<CSelectable>().SetDeselect();
            }
            // and mark it a de-selected
            unitList[i].GetComponent<CSelectable>().SetDeselect();
        }
    }
} // CreatingSquare2 ()
```

Código 5.9: Método *CreatingSquare2* para la selección múltiple de unidades según su posición en coordenadas de pantalla.

b. Selección múltiple por tipo de unidad

Se realiza dentro del mismo *if* que la selección simple, cuando se comprueba que sea una unidad y que sea del mismo equipo del jugador que la selecciona. Primero se mira si está pulsada alguna tecla *control* y si es así primero se añade a la lista *unitSelectedList* y se marca como *selected;* después se realiza un bucle *foreach* donde se agrega a la lista *unitSelectedList* y se marcan como *selected* todas las unidades que sean del mismo tipo que la seleccionada con el ratón. Si no está pulsada ninguna tecla *control* entonces se mira si es el primer clic de un doble clic, si no lo es hay que ver si es un doble clic viendo el intervalo de tiempo que ha pasado desde el primero; en el caso de que sea un doble cl clic ick se realiza lo mismo que se hace cuando se pulsa alguna tecla *control*.

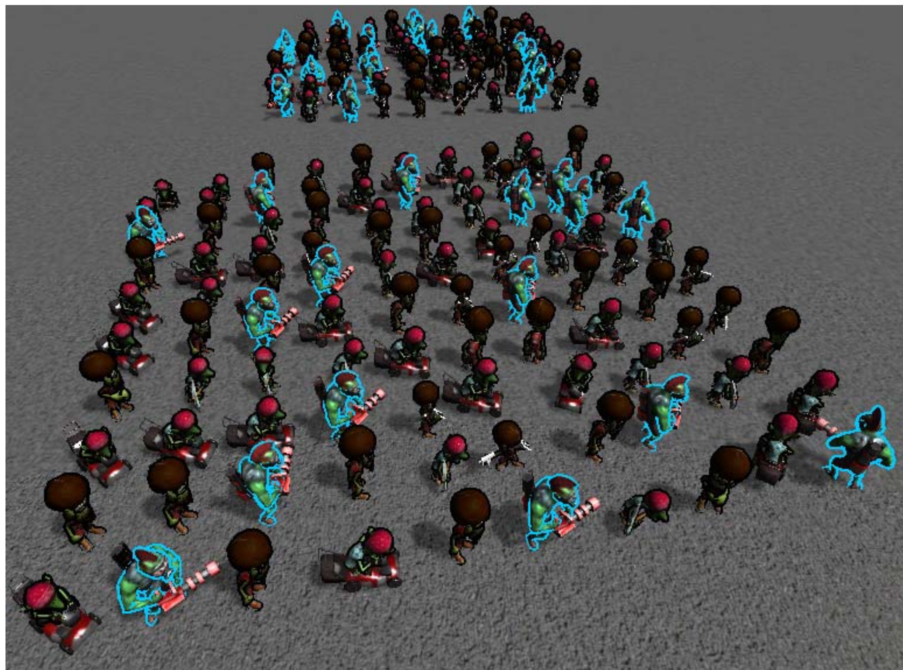


Figura 5.6: Ejemplo de selección múltiple por tipo de unidad.

c. Selección múltiple selectiva

Para hacer esta selección lo que se ha hecho es comprobar antes de cada *DeselectAll ()* si se está pulsando alguna tecla *shift*, además en *CreatingSquare2 ()* se comprueba cuando se deseleccionan unidades al hacerse más pequeño el cuadrado.

ii. Las unidades de los ejércitos

1. Resumen general

En este tema se describe el diseño e implementación general de las unidades RTS, centrándose en las funcionalidades comunes de todos los tipos de unidades, el diagrama de clases implementadas y su funcionalidad con una máquina de estados. En los apartados posteriores se describen el diseño e implementación concreto de cada tipo de unidad.

Además se recomienda consultar la documentación sobre Diseño del Videojuego tratada en la otra memoria desarrollada de este proyecto sobre *Ingeniería Software y Diseño del Videojuego* (Cuesta Boluda, Cuesta Boluda, & Rodríguez-Osorio Jiménez, 2014) para obtener una mejor visión sobre el diseño de cada tipo de unidad y sus características dentro del videojuego.

Las principales características de las unidades son la capacidad de ser seleccionadas y recibir órdenes del jugador, como desplazarse por la malla de navegación del escenario sorteando obstáculos y a otras unidades, y atacar, tanto a unidades como edificaciones enemigas.

a. Componentes

Los principales componentes de las unidades de los ejércitos son el *NavMeshAgent* de Unity y el que contiene el *script* de la clase *UnitController*, de la que heredan todos los tipos de unidades. Este *script* contiene, por ejemplo, el código necesario para que las unidades se muevan por el escenario del juego, paren cuando llegan a su destino, ataquen a enemigos, reciban daño, reciban curación, etc.

Por su lado, el componente *NavMeshAgent* es el que permite a la unidad desplazarse por la malla de navegación sorteando tanto obstáculos fijos (como edificios, árboles, etc.) como otras unidades. Contiene algunos atributos para dotar de "credibilidad" a la unidad afectando a la manera en la que se mueve por el escenario, como la velocidad de desplazamiento, fuerza de fricción con otras unidades, velocidad de rotación, etc. (ver apartado 4.a.vii sobre la malla de navegación).

Además, las unidades de los ejércitos también tienen otros componentes:

- **CLife**: es un *script* que tienen todos los objetos del juego que tienen vida, como las propias unidades o los edificios destruibles. Su implementación es muy básica, ya que solo contiene los atributos necesarios para representar la vida actual, total máxima y métodos para recibir daño, consultar si sigue vivo, etc.
- **CTeam**: *script* muy básico que indica la pertenencia de la unidad a un ejército. Este componente es compartido tanto para las unidades de los ejércitos, como para los héroes y para las torretas.
- **CSelectable**: otro *script* que tienen todos los objetos que son seleccionables por el jugador. Se encarga de cambiar el aspecto de la unidad cuando esta está seleccionada (y cuando se des-selecciona).
- **UnitRemote/Network**: *script* que define el comportamiento de la unidad para el juego en red (descrito en el capítulo 5.c.i).

b. Diagrama de clases

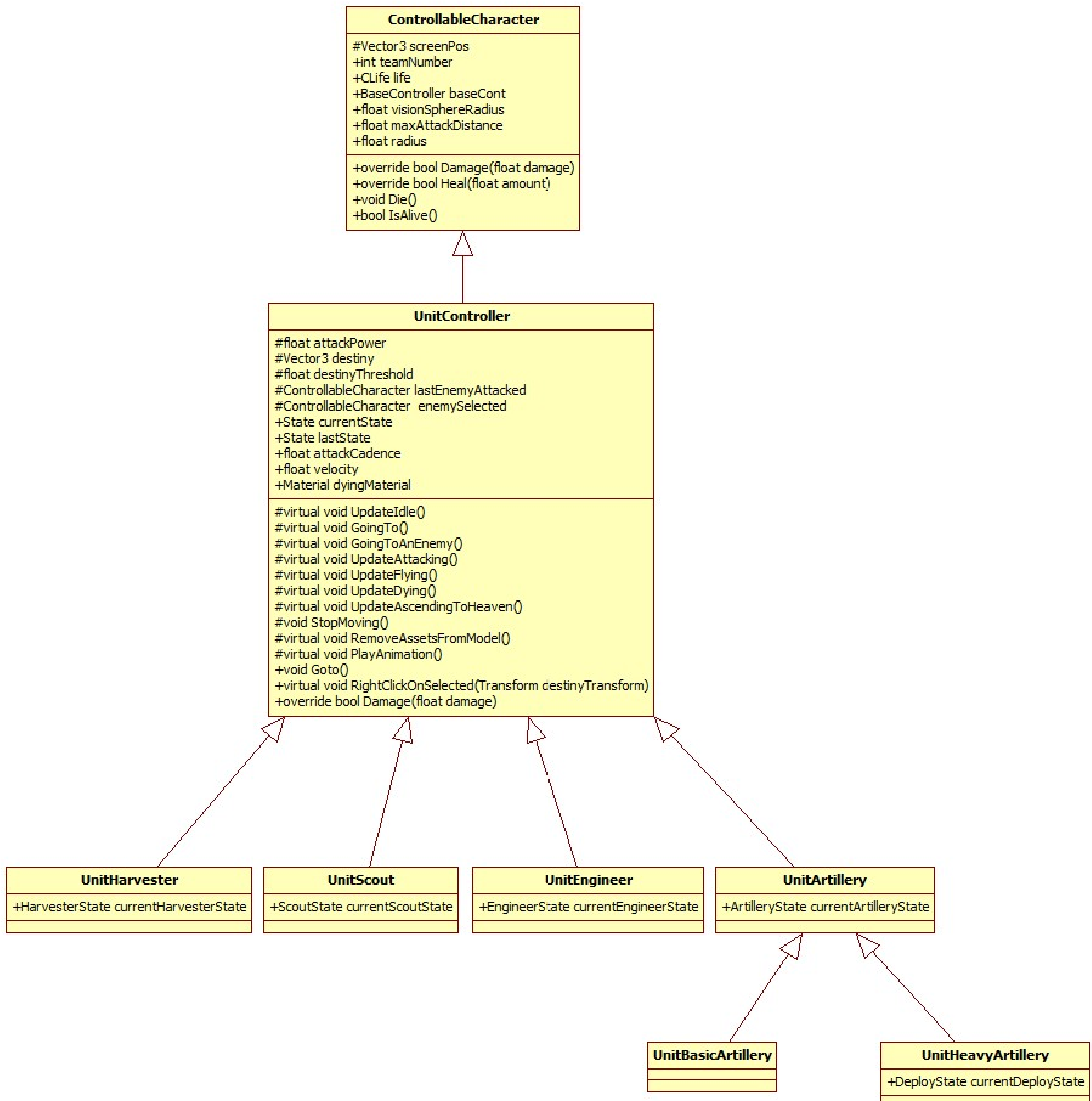


Figura 5.7: Diagrama de clases simplificado de las unidades del *RTS*.

Tanto la clase padre de las unidades (*UnitController*) de los ejércitos como los héroes (*HeroController*) del juego, heredan de una clase padre llamada *ControllableCharacter* que contiene los atributos y métodos propios a todos los tipos de unidades controlables por el jugador del juego (como por ejemplo las referencias a los componentes *CLife* y *CTeam*, o métodos como el *Damage* para recibir daño).

De la clase *UnitController* heredan las clases que definen el comportamiento específico de cada tipo de unidad, sobreescribiendo los métodos esenciales para sus acciones concretas. Además estas clases contienen una nueva máquina de estados que extiende a la de la clase padre.

Por último, de cada una de las clases concretas, heredan otras clases más por cada ejército diferente en el juego. Esto es necesario porque aunque dos unidades de un tipo tengan las mismas cualidades y funcionalidad, su modelado y armas son diferentes, y las referencias a los *dummys* y *GameObjects* que representan sus armas y accesorios son exclusivas de ese tipo de unidad de ese ejército en concreto. Estas referencias se recogen en el método **Awake ()** (ver apartado del bucle de juego en Unity 4.a.iii). En el siguiente ejemplo (Código 5.10) de la clase de artillería ligera del ejército Goblin se ve como estas clases hijas son bastante simples:

```
public class UnitBasicArtilleryGoblin : UnitBasicArtillery
{
   public override void Awake ()
   {
       base.Awake();

       numberOfWeapons = 2;
       // if the refences to the dummies of the model haven't been set in the
       // editor, we search for them now:
       if (dummyLeftWeapon == null)
           dummyLeftWeapon = transform.FindChild("Bip001/Bip001 Pelvis/Bip001
Spine/Bip001 Spine1/Bip001 Neck/Bip001 L Clavicle/Bip001 L UpperArm/Bip001 L
Forearm/Bip001 L Hand/Mano IZQ/WeaponLeft");
       if (dummyRightWeapon == null)
           dummyRightWeapon = transform.FindChild("Bip001/Bip001 Pelvis/Bip001
Spine/Bip001 Spine1/Bip001 Neck/Bip001 R Clavicle/Bip001 R UpperArm/Bip001 R
Forearm/Bip001 R Hand/Mano DER/WeaponRight");
       if (dummyLeftWeaponGunBarrel == null)
           dummyLeftWeaponGunBarrel =
dummyLeftWeapon.FindChild("GunBarrelLeft");
       if (dummyRightWeaponGunBarrel == null)
           dummyRightWeaponGunBarrel =
dummyRightWeapon.FindChild("GunBarrelRight");
       if (dummyBat == null)
           dummyBat = transform.FindChild("Bip001/Bip001 Pelvis/Bip001
Spine/Arma Blanca/Cylinder002");

       if (dummyLeftWeapon)
           leftWeapon = dummyLeftWeapon.gameObject;
       if (dummyRightWeapon)
           rightWeapon = dummyRightWeapon.gameObject;
       if (dummyBat)
           baseballBat = dummyBat.gameObject;
   }
} // class UnitBasicArtillery
```

Código 5.10: Ejemplo de código de una de las últimas clases en la jerarquía de las unidades del *RTS*.

c. La clase UnitController

Como ya se ha comentado, el *script* que contiene la clase *UnitController* es el principal componente de los objetos que representan las unidades de los ejércitos del *RTS*. Esta clase contiene el comportamiento que comparten todos los tipos de unidades, sea cual sea su clase, e implementa la máquina de estados general de las unidades. A continuación se describe la

clase en general y más adelante se hará un mayor inciso en la citada máquina de estados, ya que, por su importancia, se considera que merece una mención más concreta.

Principales atributos:

- *AttackPower*: potencia del ataque.
- *AttackCadence*: indica la cadencia de ataque (cada cuántos segundos se ataca).
- *Velocity*: velocidad de la unidad (viene determinado por el atributo *speed* del componente *NavMeshAgent*).
- *Destiny*: variable de tipo *Vector3* que indica el último destino marcado.
- *DestinyThreshold*: *float* que representa la distancia umbral en la que la unidad se puede parar antes de llegar a su destino.
- *DyingMaterial*: referencia al material que se establece en la propia unidad cuando esta muere y asciende para desaparecer.
- *LastEnemyAttacked*: referencia al último enemigo atacado.
- *EnemySelected*: referencia al enemigo seleccionado para atacar.

Principales métodos:

- *Awake* y *Start*: métodos pertenecientes al ciclo de Unity (ver capítulo correspondiente: 4.a.iii). Aquí se establecen algunas referencias, se inicializan algunos parámetros, se introduce la unidad en la estructura correspondiente de la herramienta de medición de distancias (ver capítulo correspondiente: 5.a.viii) y se comienza a reproducir la animación de *idle* de la unidad.
- *Update*: método perteneciente al ciclo de Unity. Aquí se invoca a los métodos correspondientes de la máquina de estados.
- *OnGUI*: método perteneciente al ciclo de Unity. Contiene el código que pinta la barra de vida de la unidad.
- *GoTo*: recibe un *Vector3* con el siguiente destino que tiene la unidad. Se invoca, por ejemplo, cuando un jugador, con la unidad seleccionada, le ordena desplazarse a un punto del escenario. Actualiza el atributo *destiny* de la clase y se lo indica al componente *NavMeshAgent* (GetComponent<NavMeshAgent>(). SetDestination(destiny);), actualiza el estado a *GoingTo* y reproduce la animación de andar.
- *StopMoving*: detiene la navegación por la malla de la unidad (GetComponent<NavMeshAgent>().Stop();), actualiza el estado a *Idle* y reproduce la animación de *idle*.
- *RightClickOnSelected*: se ejecuta cuando la unidad está seleccionada por el jugador y se hace clic con el botón derecho del ratón. Es uno de los métodos más importantes, ya que puede alterar de manera abrupta el comportamiento de la máquina de estados de la unidad (al ser un evento asíncrono de la máquina pues se puede ejecutar en cualquier instante de tiempo independientemente del estado en el que se encuentre). Comprueba sobre qué objeto se ha hecho clic y en qué lugar, y en función de esto se realiza una acción u otra (por ejemplo, si se ha seleccionado

una unidad enemiga, se actualiza la referencia de *enemySelected* y se cambia el estado a *GoingToAnEnemy*). Éste es un método virtual y se sobreescribe en las clases hijas.

- *RemoveAssetsFromModel*: este método podría considerarse virtual puro (aunque no existe este concepto en C#), ya que en esta clase está vacío, y debe de ser implementado en las clases hijas. Se invoca cuando la unidad ha muerto y va a comenzar la animación de ascenso a los cielos, y sirve para destruir los accesorios del modelo de la unidad, como armas, pico, mochila, gorro, etc.

d. Máquina de estados

Debido a las diferentes características y comportamientos, alguno de bastante complejidad, de las unidades, se opta por diseñar sus comportamientos en base a máquinas de estado en diferentes niveles. De esta manera, la clase padre (*UnitController*) contiene el nivel más básico de la máquina de estados de cada tipo de unidad, con los estados que pueden alcanzar todas las unidades independientemente de su tipo (recolectora, artillería, etc.), y después, en las clases hijas (las que representan el comportamiento de un tipo de unidad en concreto) una máquina de estados con los estados propios de cada clase. En cada ciclo de juego se ejecuta, como se ve en el Código 5.11, cada método *Update* concreto dependiendo del estado en el que se encuentre; en algunos casos estos métodos, que son virtuales, los sobrescriben las clases hijas para concretar más alguno de ellos.

En la Figura 5.8 puede verse el diagrama de estados de esta máquina.

Descripción de los estados:

- *Idle*: la unidad se encuentra en reposo, esperando a órdenes del jugador, o a otros eventos.
- *GoingTo*: dirigiéndose a, se dirige al destino marcado por el atributo *destiny*.
- *GoingToAnEnemy*: dirigiéndose a un enemigo.
- *Attacking*: atacando a un enemigo.
- *Flying*: la unidad ha recibido un ataque con explosión y a causa de esta ha sido lanzado por los aires.
- *Dying*: este estado indica que la unidad está muriendo, su vida ha llegado a 0 y se está ejecutando la animación de muerte.
- *AscendingToHeaven*: estado previo a la desaparición de la unidad del juego, asciende mientras desaparece.

Estos estados permiten definir el comportamiendo de la unidad a través de varios métodos virtuales que pueden ser sobreescritos en las clases hijas. De esta manera se consigue el funcionamiento a través de las dos máquinas de estados (la propia de la clase *UnitController*) y la de la clase hija en concreto.

```
public override void Update ()
{
    base.Update();

    switch (currentState)
    {
        case State.Idle:              UpdateIdle();              break;
        case State.GoingTo:           UpdateGoingTo();           break;
        case State.GoingToAnEnemy:    UpdateGoingToAnEnemy();    break;
        case State.Attacking:         UpdateAttacking();         break;
        case State.Flying:            UpdateFlying();            break;
        case State.Dying:             UpdateDying();             break;
        case State.AscendingToHeaven: UpdateAscendingToHeaven(); break;
    }
}
```

Código 5.11: Método *Update* de la clase *UnitController*.

La unidad cuando se encuentra en reposo está en el estado *Idle* y en ese momento dependiendo de las acciones del juego y eventos empezará a comportarse como una máquina de estados. Se han simplificado por espacio varias transiciones y estados, los estados son:

- *LastState*: en el momento en el que una unidad es lanzada por los aires se guarda el estado en el que se encontraba, para que cuando deje de volar pueda volver al estado en el que se encontraba.
- *Idle or Going or GoingToAnEnemy or Flying or Attacking*: desde estos estados se podrá llegar al estado *Dying.*

Las transiciones que se han simplicaco son:

- *A: cambia de estado *Attacking* a *GoingToAnEnemy* cuando la distancia desde la unidad hasta el enemigo es mayor que la distancia máxima de ataque, pero menor que la de visión.
- *B: cambia de estado *GoingToAnEnemy* a *Attacking* cuando la distancia desde la unidad hasta el enemigo es menor que la distancia máxima de ataque.
- *C: cambia de estado *Flying* al estado anterior al cuando la unidad deja de volar y está en el suelo.

2. Unidades Recolectoras

A continuación se describe el diseño e implementación de las unidades recolectoras del RTS, centrándose en las funcionalidades específicas de este tipo de unidades. El comportamiento general de todas las unidades se describe en el apartado anterior.

El principal cometido de estas unidades es la recolección de recursos por el escenario para el ejército. Además, estas unidades también son capaces de realizar tareas de curación, tanto a otras unidades del ejército, como al héroe del equipo.

a. La clase UnitHarvester

El componente *UnitHarvester* es un *script* donde está implementada la clase con el comportamiento y representa el principal componente.

Esta clase hereda de la clase *UnitController* y contiene una nueva máquina de estados más específica, extendiendo a la de la clase padre y sobrescribiendo varios métodos.

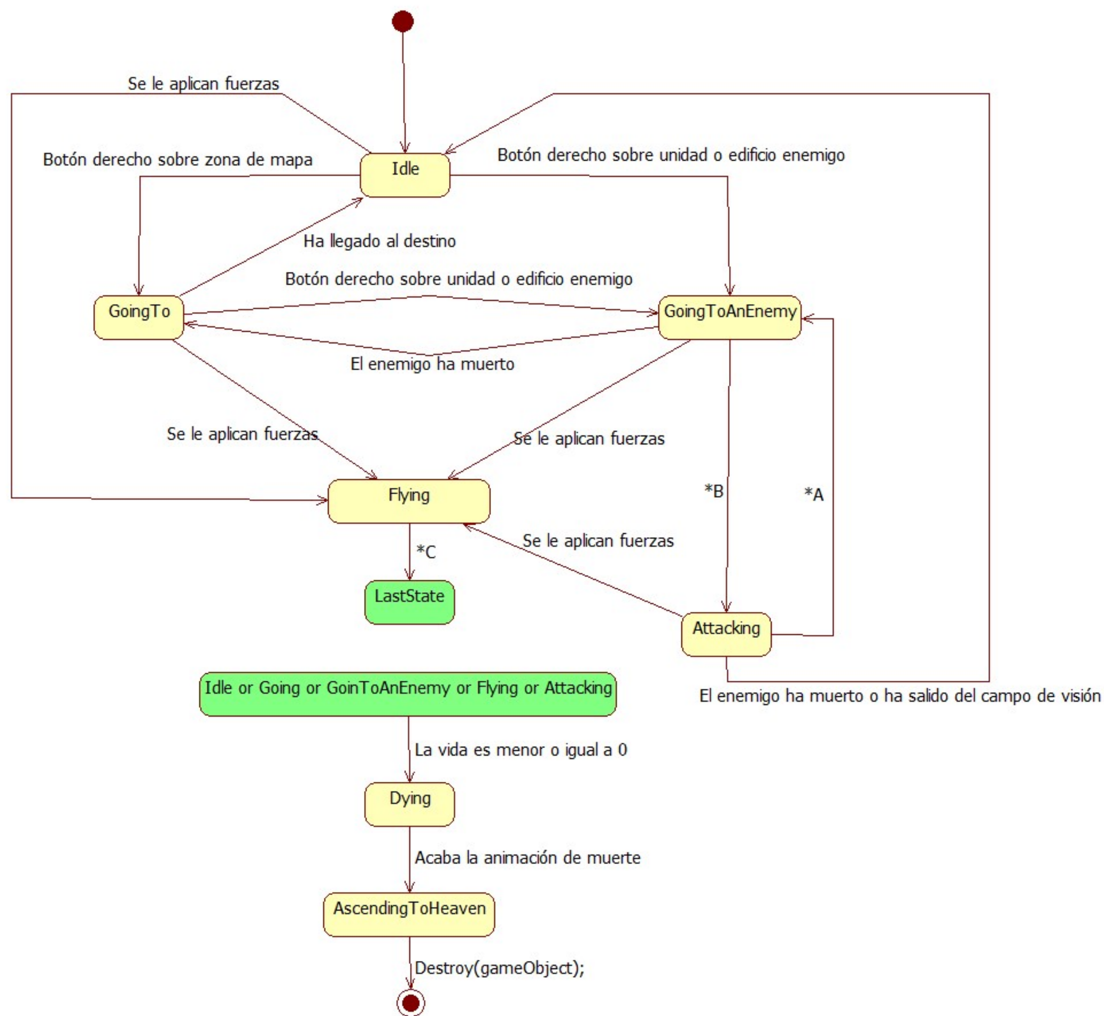


Figura 5.8: Diagrama de estados de la clase UnitController.

Principales atributos:

- *HarvestCapacity*: capacidad de recursos que puede llevar consigo una unidad.
- *ResourcesLoaded*: indica la cantidad actual de recursos que la unidad lleva cargados.
- *HarvestTime*: tiempo (en segundos) que la unidad tarda en realizar una recolección de recursos.
- *AmountOfResourcesPerHarvest*: especifica la cantidad de recursos que la unidad es capaz de recolectar cada fracción de tiempo indicada por el atributo *HarvestTime*.
- *TotalHarvest*: variable que acumula la cantidad total de recursos cosechados por la unidad.
- *MineralPack*: referencia al objeto que contiene la apariencia de un pack de minerales instanciados, para cuando la unidad regrese a la base del ejército cargada de recursos.
- *Dummys varios*: referencias a las posiciones dentro de la propia jerarquía del modelo de la unidad donde se pueden instanciar varios accesorios como gafas, casco, mochila, etc.

- *CurrentMine*: referencia a la última mina donde se ha cosechado.
- *LastHarvestPosition* y *LastHarvestIndex*: indican la posición y el índice respectivamente de la propia mina referenciada en *CurrentMine* donde la unidad está recolectando (como si fuera la posición de la cola de espera de esta).
- *CurrentCharacterHealed*: referencia a la última unidad o héroe que se ha curado.
- *AmountOfLifePerHeal*: cantidad de puntos de curación que la unidad es capaz de sanar por unidad de tiempo (*HealTime*).
- *TotalHealed*: cantidad de puntos de sanación que la unidad ha producido en total.
- *CurrentHarvestState* y *NextHarvestState*: indican el estado actual y posterior esperado respectivamente de la máquina de estados que define el comportamiento de las unidades recolectoras.

Principales métodos:

- *RightClickOnSelected*: este método se hereda de la clase *UnitController* y se ejecuta cuando la unidad está seleccionada por el jugador y se hace clic con el botón derecho del ratón. Es uno de los métodos más importantes, ya que puede alterar de manera abrupta el comportamiento de la máquina de estados de la unidad (al ser un evento asíncrono de la máquina, pues se puede ejecutar en cualquier instante de tiempo independientemente del estado en el que se encuentre).

b. Máquina de estados

Para que se puedan especificar bien las acciones de recolección de estas unidades, se ha creado una máquina de estados propia, usando también la de la clase padre (*UnitController*), para poder efectuar estas acciones de manera efectiva. Estos estados tienen que recoger las acciones de recolección y de curación de unidades aliadas.

Descripción de los estados:

- *None*: en reposo, sin ninguna acción.
- *GoingToMine*: la unidad se está dirigiendo a una mina para comenzar a recolectar minerales.
- *Waiting*: la unidad se encuentra esperando a que la mina tenga lugares de extracción libres.
- *GoingToChopPosition*: la mina tiene un lugar de extracción libre y la unidad se está dirigiendo a este.
- *Choping*: *picando*, se están extrayendo minerales.
- *ReturningToBase*: la unidad tiene recursos cargados y se dirige a la base (o a un almacén de recursos) para “descargarlos”.
- *GoingToHealUnit*: se ha seleccionado a un aliado para su curación y la unidad se dirige hacia su posición para sanarlo.
- *Healing*: sanando a un aliado.

Transiciones:

- *None -> GoingToMine*: la unidad se encuentra en reposo y se ha seleccionado una mina para su recolección y además la cantidad de recursos con la que carga la unidad es menor a la cantidad máxima ("le caben más recursos").
- *None -> ReturningToBase*: la unidad se encuentra en reposo y se ha seleccionado una mina para su recolección pero la cantidad de recursos con la que carga es igual a la cantidad máxima con la que puede cargar ("está totalmente cargada de recursos"). Vuelve a la base (o al almacén de recursos más cercano a la mina seleccionada) para dejar los recursos con los que carga y a continuación irá a la mina seleccionada para comenzar otra extracción (el valor de *NextHarvestPosition* será de *GoingToMine*).

Eventos asíncronos: eventos que se producen cuando el jugador hace clic con el botón derecho del ratón sobre algún objeto de juego con la unidad seleccionada, de manera asíncrona a la máquina de estados de la unidad, lo que altera su funcionamiento normal.

- Se hace clic en el suelo, fuera:
  - Si se está dirigiendo a la posición de extracción o ya se está extrayendo: se abandona la posición de recolección (comunicándoselo a la mina).
  - Si no, si estaba esperando en la mina por huecos de extracción libres: se abandona la cola de espera (comunicándoselo a la mina).
  - Si no, si estaba curando a un aliado: se vuelve a pintar el pico (que desaparece mientras se realizan labores de curación).
  - Se actualiza el estado actual a *None*.
- Se hace *clic* en una mina de recursos:
  - Es posible que la mina seleccionada sea nueva y no se haya descubierto (seleccionado) antes, por lo tanto hay que actualizar la lista de minas descubiertas del *ArmyController* del ejército (método *UpdateMines*).
  - Se actualiza la referencia de la última mina seleccionada (atributo *currentMine*).
  - Se actualiza la posición de la base (o del almacén de recursos) donde se dejarán los recursos cuando termine la extracción.
  - Si la unidad estaba a espera (estado = *None*):
    - Si el número de recursos con los que carga la unidad es igual a la capacidad máxima de esta, vuelve a la base para descargarlos, y después irá a la mina seleccionada para comenzar otra extracción:
      - estado actual = *ReturningToBase*.
      - estado siguiente = *GoingToMine.*
    - Si no, si el número de recursos cargados es menor a su capacidad, la unidad se dirigirá a la mina:
      - estado actual = *GoingToMine*.
      - estado siguiente = *Choping*.
  - Si no, si estaba dirigiéndose a la base:
    - Si la unidad está llena de recursos, vuelve a la base para dejarlos y después se dirigirá a la mina:
      - estado siguiente = *GoingToMine.*

        - Si no, si el número de recursos cargados es menor a su capacidad, la unidad se dirigirá a la mina:
            - estado actual = *GoingToMine*.
            - estado siguiente = *Choping*.
    - Si no, si estaba curando a un aliado:
        - Se vuelve a pintar el pico.
- Se hace clic en la base del ejército o en un almacén de recursos:
    - La unidad se dirige a la base, si tiene recursos los descargará:
        - estado actual = *ReturningToBase*.
        - estado siguiente = *None*.
    - Si no tiene recursos, simplemente se dirigirá a la base.
- Se hace clic en una unidad *ControllableCharacter*, es decir, en otra unidad del ejército, o en un héroe del mismo equipo:
    - Se actualiza la referencia al último compañero al que se va a sanar.
    - estado actual = *GoingToHealUnit*.
    - estado siguiente = *Healing*.

Figura 5.9: Diagrama de estados de las unidades recolectoras del RTS.

3. Unidades Ingenieras

A continuación se describe el diseño e implementación de las unidades ingenieras del *RTS*, centrándose en las funcionalidades específicas de este tipo de unidades. El comportamiento general de todas las unidades se describe en el apartado anterior.

Las funciones principales de estas unidades son las de construcción de torres y almacenes de recursos de su raza, conquista de torres neutrales y reparación de cualquier tipo de edificio. Además tiene un tipo de ataque característico, lanzando una granada, explota al pasar un tiempo y daña a las unidades enemigas en un radio determinado.

a. La clase UnitEngineer

El componente *UnitEngineer* es un *script* donde está implementada la clase con el comportamiento y representa el principal componente.

Esta clase hereda de la clase *UnitController* y contiene una nueva máquina de estados más específica, extendiendo a la de la clase padre y sobrescribiendo varios métodos.

Principales atributos:

- *AmountPerAction*: capacidad de construcción, conquista y/o reparación.
- *EngineerTime*: ratio en que la unidad tarda en realizar una construcción, conquista y/o reparación.
- *ActualEngineerTime*: tiempo que ha pasado desde el último engineerTime hasta el actual.
- *CurrentItem*: Referencia al edificio que se está construyendo, conquistando o reparando.
- *TowerGoblinPrefab y warehousePrefab:* Referencia al *prefab* de la torre goblin y almacén de recursos respectivamente, para poder instanciarlas.
- *TowerGoblin* y *warehouse:* los objetos que han sido instanciados (torre goblin y almacén de recursos respectivamente).
- *NewTGConstruct* y *newWConstruct*: booleanas para saber si se está construyendo en ese momento una torre goblin o un almacén de recursos.
- *ConstructDestiny:* el destino de construcción de una torre goblin o un almacén de recursos.
- *Construct* y *Conquest:* booleana para saber si se ha acabado la construcción o conquista respectivamente.
- *DummysLaptop* y *dummyHammer*: referencias a las posiciones dentro de la propia jerarquía del modelo de la unidad donde se puede instanciar un ordenador portátil y un martillo respectivamente..
- *Laptop* y *hammer*: referencia al *prefab* del ordenador portátil y del martillo respectivamente, para poder instanciarlas.
- *lastEngineerPos*y *lastEngineerIndex*: indican la posición y el índice respectivamente del propio edificio referenciado en *currentItem* donde la unidad está actuando.

- *Grenade*: referencia al prefab de la bola de fuego que lanza la unidad para poder instanciarla.
- *NewGrenade*: Donde se va a guardar la nueva bola de fuego.
- *currentEngineerState*: indica el estado actual de la máquina de estados que define el comportamiento de las unidades ingenieras.

Principales métodos:

- *RightClickOnSelected*: este método se comenta anteriormente en las unidades recolectoras.

b. Máquina de estados

Estas unidades son las que tienen el comportamiento más complejo de todas, ya que pueden realizar muchas acciones (conquista, reparación, construcción y ataque) y se especifican como una máquina de estados para facilitar su implementación. Si no tuvieran esa máquina de estados la clase sería mucho más compleja, además de ser mucho más proclive a errores.

Descripción de los estados:

- *None*: en reposo, sin ninguna acción.
- *GoingToRepairItem*: la unidad se está dirigiendo a un edificio para repararlo.
- *GoingToConstructItem*: la unidad se está dirigiendo a un zona/edificio para construir.
- *GoingToConquerableItem*: la unidad se está dirigiendo a un edificio para conquistarlo.
- *Waiting*: la unidad se encuentra esperando a que el edificio al que va a realizar alguna de las tres acciones posibles tenga lugares libres.
- *GoingToConquestPosition*: el edificio tiene un lugar de conquista libre y la unidad se está dirigiendo a este.
- *GoingToRepairPosition*: el edificio tiene un lugar de reparación libre y la unidad se está dirigiendo a este.
- *GoingToConstructPosition*: el edificio tiene un lugar de construcción libre y la unidad se está dirigiendo a este.
- *Repairing*: la unidad se encuentra reparando un edificio.
- *Conquering*: la unidad se encuentra conquistando un edificio.
- *Constructing:* la unidad se encuentra construyendo un edificio.

Diagrama de estados:

El diagrama se ha simplificado considerablemente para poder explicar de manera fácil el funcionamiento de estas unidades. Hay estados que se muestran que no están en la máquina de estados, pero se han hecho para poder agrupar varios estados similares, estos son:

- *GoingToActionItem*: Este estado será *GoingToConstructItem*, *GoingToConquerableItem* o *GoingToRepairItem*, dependiendo de la acción que esté llevando a cabo la unidad.
- *GoingToActionPosition*: Este estado será *GoingToConstructPosition, GoingToConquerablePosition* o *GoingToRepairPosition*, dependiendo de la acción que esté llevando a cabo la unidad.
- *Action*: Este estado será *constructing*, *conquering* o *repairing*, dependiendo de la acción que esté llevando a cabo la unidad.

Se va a explicar más detalladamente la transición del estado *None* al estado *GoingToActionItem* y la transición del estado *GoingToActionPosition* al estado *Action*:

- **A:* Estando en el estado *None*, si se hace clic en un edificio, va a ir o no al estado *GoingToActionItem* dependiendo del tipo de edificio seleccionado y del estado de esta estructura.
    - Si es en un edificio conquistable:
        - Si el edificio no está conquistado o está siendo conquistado, se pasa al estado *GoingToActionItem* que en este caso será *GoingToConquerableItem*.
    - Si es en el suelo del mapa y se ha decidido construir un nuevo edificio, se pasa al estado *GoingToActionItem* que en este caso será *GoingToConstructItem.*
    - Si es en un edificio en construcción y es del mismo equipo que el de la unidad, se pasa al estado *GoingToActionItem* que en este caso será *GoingToConstructItem.*
    - Si es en un edificio que tiene menos del 100% de vida, hay que ver diferentes casos para poder pasar al estado *GoingToActionItem* que en este caso será *GoingToRepairItem:*
        - Si es una torre neutral y es de la posesión del equipo de la unidad.
        - Si es en algún edificio construible y está construido.
        - Si es en algún otro edificio y es del mismo equipo.
- *B: Para pasar del estado *action* al estado *None* hay que diferenciar las acciones que la unidad esté llevando a cabo en ese momento, por lo que puede estar en el estado *conquering*, *constructing* y *repairing:*
    - *Conquering*: cambia de estado si se ha conquistado el edificio (ya sea del mismo equipo o no).
    - *Constructing*: cambia de estado si ha acabado de construir o ha sido destruido el edificio.
    - *Repairing:* cambia de estado si se ha reparado o se ha destruido el edificio.

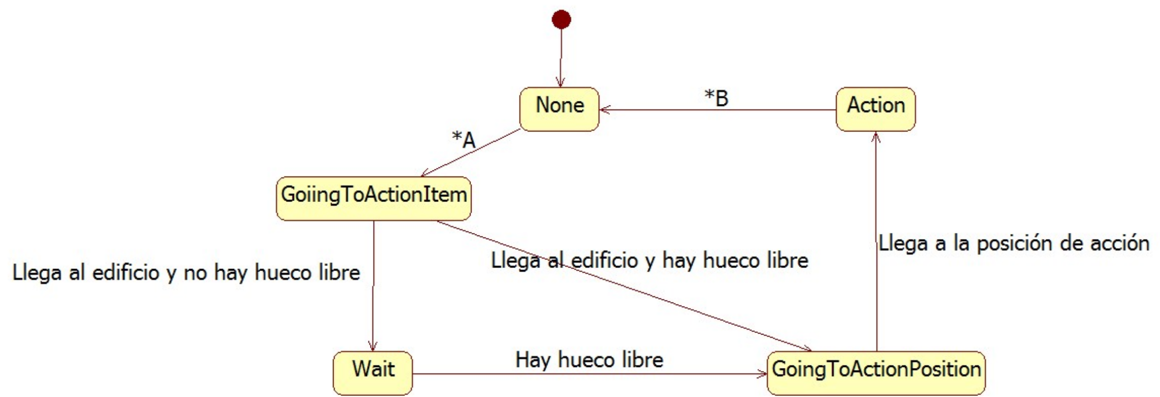


Figura 5.10: Diagrama de estados de las unidades ingenieras del RTS.

Eventos asíncronos:

- Se hace clic en el suelo, fuera:
  - Si está en alguno de los estados de *GoingToActionItem*, *GoingToAcionPosition* o *Action*:
    - Se abandona esa posición (comunicándoselo al edificio) y va a la posición indicada.
  - Si no, si estaba en el estado *Wait*:
    - Se abandona la cola de espera (comunicándoselo al edificio) y va a la posición indicada.
  - Se actualiza el estado actual a *None*.
- Se hace clic en un edificio:
  - Si es un edificio del mismo equipo:
    - Si es un edificio en construcción:
      - Se va a construirlo, se cambia de estado a *GoingToConstructItem.*
    - Si es un edificio con la vida menor del 100%:
      - Se va a repararlo, se cambia de estado a *GoingToRepairItem.*
  - Si es un edificio de otro equipo y tiene una vida mayor del 0%:
    - Se va a atacarlo y se cambia el estado de la clase padre a *GoingToAnEnemy*.
  - Si es un edificio neutral:
    - Se va a conquistarlo, se cambia de estado a GoingToConquerableItem.
  - Sea el edificio neutral, del equipo o enemigo, si tiene que abandonar la posición:
    - Si está en alguno de los estados de *GoingToActionItem*, *GoingToAcionPosition* o *Action*:
      - Se abandona esa posición (comunicándoselo al edificio)
    - Si no, si estaba en el estado *Wait*:
      - Se abandona la cola de espera (comunicándoselo al edificio).

4. Unidades Artilleras

A continuación se describe el diseño e implementación de las unidades artilleras del RTS, centrándose en las funcionalidades específicas de este tipo de unidades.

La función principal de estas unidades es atacar a todos los enemigos. Poseen dos tipos de ataque: las unidades básicas tienen un ataque a distancia y otro ataque más rápido a corta distancia; las unidades pesadas tiene un ataque a distancia y otro ataque de artillería con una bomba que explota al chocar con el suelo y daña a las unidades enemigas con una explosión en un radio determinado.

a. La clase UnitArtillery

El componente *UnitArtillery* es un script donde está implementada la clase con el comportamiento y representa el principal componente de estas unidades. De esta clase heredan las que implementan las unidades de artillería básica y artillería pesada: *UnitBasicArtillery* y *UnitHeavyArtillery*.

Esta clase hereda de la clase *UnitController* extendiendo a la de la clase padre y sobreescribiendo varios métodos.

Principales Atributos

- *currentArtilleryState*: guarda el estado actual de la unidad de artillería.
- *enemiesInside*: lista donde se almacenan las referencias a los enemigos que están dentro del área de visión de la unidad.
- *movingAttacking*: indica si la unidad está ejecutando el movimiento de ataque, es decir, que va a atacar a las unidades enemigas mientras se mueve de forma automática.
- *dummyLeftWeapon*: referencia a la posición dentro de la propia jerarquía del modelo de la unidad para posicionar el arma en la mano izquierda.
- *dummyRightWeapon*: igual que el anterior pero para la mano derecha.
- *dummyLeftWeaponGunBarrel*: referencia a la posición dentro de la propia jerarquía del modelo de la unidad para posicionar el cañón de la pistola en la mano izquierda.
- *dummyRightWeaponGunBarrel*: igual que el anterior pero para la mano derecha.
- *numberOfWeapons*: número de armas con las que carga el modelo.
- *shotParticles*: referencia a las particulas que se van a emitir al disparar.
- *alertHitTimer*: determina el tiempo que tarda en lanzar un rayo para detectar un enemigo.
- *eyesPosition*: indica la posición desde donde lanza rayos la unidad para comprobar si “ve” a un enemigo en concreto.
- *primaryAttackCadence*: determina la cadencia de disparo del ataque primaria.
- *secondaryAttackCadence*: determina la cadencia de disparo del ataque secundario.
- *attack2Selected*: determina qué tipo de ataque está usando.
- *maxAttackDistance1*: máxima distancia del ataque primario.
- *maxAttackDistance2*: máxima distancia del ataque secundario.

Principales Métodos

- *SearchForAnEnemy*: este método es llamado cuando se selecciona un enemigo para atacarle. Lanza un rayo para determinar si está a la vista (ver Código 5.12).
- *SearchForEnemies*: recorre la lista *enemiesInside* y lanza un rayo por cada uno de los enemigos para saber si están a la vista.
- *AttackMovement*: se llama cuando el jugador envía una unidad en alerta de ataque hacia una posición.
- *EnemyEntersInVisionSphere*: este método es llamado desde la clase *DistanceMeasurerTool* (ver capítulo 5.a.viii) cuando una unidad enemiga entra en el área de visión.
- *EnemyLeavesVisionSphere*: se llama cuando una unidad enemiga abandona el área de visión.

```
protected void SearchForAnEnemy ()
{
    Vector3 fwd = enemySelected.transform.position - this.transform.position;
    fwd.Normalize();
    RaycastHit myHit;
    if ( Physics.Raycast(transform.position + eyesPosition, fwd, out myHit,
         maxAttackDistance) )
    {
        //Debug.Log(myHit.transform.name);
        // the ray has hit something
        CTeam enemy = myHit.transform.GetComponent<CTeam>();
        if ( enemy && (enemy == enemySelected) )
        {
            // this "something" is the enemy we are looking for...
            // rotate the unit in the enemy direction
            transform.LookAt(enemy.transform.position);
            lastEnemyAttacked = enemy;
            alertHitTimerAux = alertHitTimer;
            // the unit stops moving
            StopMoving();

            currentState = State.Attacking;
            cState.currentState = currentState;
        }
    }
}
```

Código 5.12: función *SearchForAnEnemy* de la clase *UnitArtillery*.

b. Artillería Ligera

Su comportamiento se define por el componente *UnitBasciArtillery*, el cual es un script donde está implementada la clase con el comportamiento específico de esta unidad.

Principales Atributos:

- *attackPower1* y *attackPower2*: guarda la fuerza de ataque del ataque principal y segundario.
- *attack1Cadence* y *attack2Cadence*: cadencia de disparo del ataque principal y secundario.

- *leftWeapon, rightWeapon, baseballBat*: referencias a los objetos que representan sus armas de ataque.
- *dummyBat*: referencia a la posición dentro de la propia jerarquía del modelo de la unidad para posicionar el bate.

Principales métodos:

- *UpdateGoingToAnEnemy*: método sobreescrito de la clase padre que comprueba si el enemigo al que se ha seleccionado para atacar está a la vista.
- *UpdateAttacking*: comprueba si esta unidad puede atacar al enemigo y, si puede, le quita vida.
- *ChangeAttack*: cambia el tipo de ataque de esta unidad (de ataque a distancia ataque cuerpo a cuerpo y viceversa).

c. Artillería Pesada

Su comportamiento se define por el componente *UnitHeavyArtillery*, el cual es un script donde está implementada la clase con el comportamiento específico de esta unidad. Esta unidad tiene la peculiaridad de poder desplegarse para aumentar la distancia y el poder de ataque, a costa de no poder moverse. Para esto, esta clase, implementa una nueva máquina de estados que guarda el estado de "despliegue".

Principales Atributos

Los atributos de la clase *UnitHeavyArtillery* son similares a los de la clase *UnitBasicArtillery* descritos anteriormente, con la salvedad de la introducción de los enumerados necesarios para guardar el estado de "despliegue" de la máquina de estados de esta unidad y la referencia al *GameObject* que representa el misil que se dispara (`public GameObject missile;`).

Principales Métodos

De forma análoga a los atributos, los principales métodos son equivalentes con los de la clase *UnitBasicArtillery* con la salvedad de los métodos referentes a la máquina de estados de esta clase que permiten cambiar el estado del modo de despliegue.

Esta máquina de estados tiene cuatro estados y empieza en modo sin desplegar; en el momento que se pulsa la tecla 'D' cambia a desplegándose y cuando termina la animación correspondiente cambia desplegado de forma automática; si se vuelve a pulsar se vuelve al modo no desplegado (pasando por el estado intermedio de des-desplegándose). Este cambio automático al finalizar una animación introduce un nuevo concepto en la implementación como es el uso de corrutinas.

Como se ha comentado, este uso de corrutinas es necesario porque cuando la unidad pasa de estado no-desplegado a desplegado, pasa primero por un estado-transición mientras el modelo está ejecutando la animación de despliegue, de manera que se necesita conocer el instante de tiempo en el que termina la animación para poder realizar el cambio al estado final de desplegado. Esto se consigue comenzando una nueva corrutina y congelando el nuevo hilo de ejecución tanto tiempo como dure la animación de transición, a continuación se invoca a un

método llamado *AnimationFinished* que establece el nuevo estado de la unidad (ver Código 5.13).

```
public enum DeployState
{
    Undeployed,
    Deploying
    Deployed,
    Undeploying
}
public DeployState currentDeployState = DeployState.Undeployed;

public override void Update ()
{
    base.Update();

    if ( isSelected && Input.GetKeyDown(KeyCode.D) )
    {
        switch (currentDeployState)
        {
            case DeployState.Undeployed:
                StopMoving();
                animation.CrossFade("Deployment-prepare");
                StartCoroutine(WaitAndCallback(
                      animation["Deployment-prepare"].length));

                currentDeployState = DeployState.Deploying;
                break;

            case DeployState.Deployed:
                if (currentState != State.Attacking)
                {
                    animation.CrossFade("Deployment-Up");
                    StartCoroutine(WaitAndCallback(
                                animation["Deployment-Up"].length));

                    currentDeployState = DeployState.Undeploying;
                }
                break;
        }
    }
} // Update

private IEnumerator WaitAndCallback (float waitTime)
{
    yield return new WaitForSeconds(waitTime);
    AnimationFinished();
}
```

Código 5.13: Fragmento de la clase *UnitHeavyArtillery* encargado del cambio de estado por medio de corrutinas.

d. Máquina de Estados

La clase *UnitBasicArtillery* y *UnitHeavyArtillery* tienen la misma máquina de estados, ya que las dos son las únicas unidades que pueden atacar de forma automática. Aparte la unidad de artillería pesada tiene una máquina de estados independiente para el tipo de ataque de despliegue.

Descripción de los estados de la máquina de estados común (Figura 5.11)

- *None*: en reposo, sin ninguna acción.

- *Alert*: este estado significa que hay enemigos dentro del área de visión de la unidad. Cada cierto tiempo (valor guardado en el atributo *alertHitTimer*) la unidad busca enemigos dentro de su radio de visión.

En la Figura 5.11 el estado *Attacking* está en verde porque es de la clase padre *UnitController*. La unidad al principio se encuentra en reposo y cuando una unidad entra en su esfera de visión (atributo *visionSphereRadius*) cambia al estado *alert*, de tal forma que cuando el enemigo que está en la esfera de visión se encuentra a una distancia de ataque (atributo *maxAttackDistance*) cambia el estado de la clase padre de *Idle* a *Attacking*. Las clases padres (*UnitController* y *UnitArtillery*) tienen el método virtual *UpdateAttacking* que las clases hijas sobrescriben, por lo que el ataque se realiza desde las hijas.

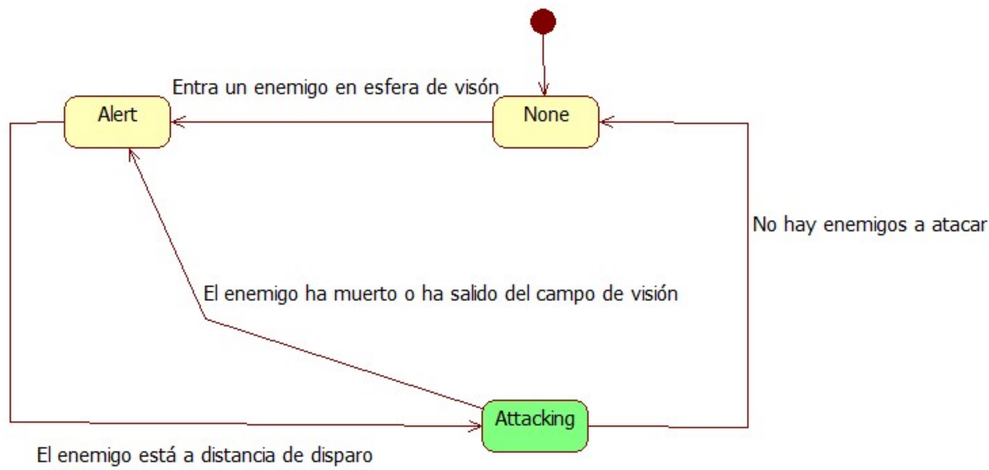


Figura 5.11: Diagrama de estados de las clases *UnitBasicArtillery* y *UnitHeavyArtillery*.

5. Unidades Exploradoras

A continuación se describe el diseño e implementación de las unidades exploradores del *RTS*, centrándose en las funcionalidades específicas de este tipo de unidades, el comportamiento general de todas las unidades se describe en el apartado anterior.

La labor principal de estas unidades es el descubrimiento del mapa gracias a su mejorada capacidad para desplazarse por el escenario. Son las únicas unidades capaces de atravesar ciertos puntos del mapa mediante un salto y se mueven más con una mayor agilidad. Además tienen la posibilidad de patrullar una zona de terreno.

a. La clase UnitScout

El componente *UnitScout* es un *script* donde está implementada la clase con el comportamiento de las unidades recolectoras de los ejércitos y representa el principal componente de estas.

Esta clase hereda de la clase *UnitController* y contiene una nueva máquina de estados más específica, extendiendo a la de la clase padre y sobreescribiendo varios métodos.

Principales atributos:

- *patrolPositionsList*: lista que contiene los puntos a patrullar.
- *nextPositionIndex*: siguiente posición de patrulla a la que debe ir la unidad.

- *Mount*: modelo del asset de la máquina cortacésped.
- *FireMount y FireMountInst*: referencian a las partículas de fuego de la máquina cortacésped.
- *Afire*: indica si la máquina cortacésped arde o no.
- *StartAfire*: indica el porcentaje de vida al que empieza a salir fuego de la máquina cortacésped.
- *StopAfire*: indica el porcentaje de vida al que, si hay fuego, desaparece.
- *CurrentScoutState*: indica el estado actual correspondiente a la máquina de estados que define el comportamiento de las unidades exploradoras.

Principales métodos:

- *RightClickOnSelected*: este método se comenta anteriormente en las unidades recolectoras.
- *StartPatrol*: este método controla y establece la secuencia de los puntos de patrulla seleccionados. Anteriormente, con la unidad exploradora seleccionada y la tecla 'P' pulsada se van definiendo dichos puntos. Dicha lista es recorrida y actualizada cada vez que la unidad llega a un punto de patrulla.

b. Máquina de estados

El conjunto de estados que define el comportamiento de las unidades exploradoras es el siguiente:

Descripción de los estados:

- *None*: en reposo, sin ninguna acción.
- *Patrolling*: patrullando entre los puntos de la lista *patrolPositionsList*.

Transiciones:

- *None -> Patrolling*: la unidad se encuentra en reposo y se ha seleccionado una secuencia de puntos a patrullar con la tecla 'P' pulsada al mismo tiempo para su recolección y además la cantidad de recursos con la que carga la unidad es menor a la cantidad máxima ("le caben más recursos").

Eventos asíncronos:

- Se hace clic en el suelo, fuera:
  - Si se está dirigiendo a la siguiente posición de patrulla o está patrullando: se abandona la misión de patrulla.
  - Se actualiza el estado actual a *None*.

iii. Los edificios

Los edificios son una parte fundamental de los videojuegos RTS y, evidentemente, también del videojuego de este proyecto. Existen diversos edificios y los más importantes son: la base del ejército (única en toda la partida), las torretas de defensa y los almacenes de recursos (ver Figura 5.12). Todos pueden ser destruidos por las unidades enemigas, o el héroe del equipo enemigo, y pueden ser reparadas por las unidades ingenieras aliadas. Además unos pueden ser construidos y otros pueden ser conquistados (como las torretas neutrales, ver Figura 5.13).

Figura 5.12: Aspecto gráfico de los 3 tipos de edificios de cada ejército (almacén de recursos, torretas y base).

Figura 5.13: Aspecto gráfico de una torreta neutral conquistable (izquierda), y del proceso de conquista por unidades ingenieras (derecha).

Estos edificios están clasificados según las funciones que comparten en una estructura o jerarquía de herencia. En esta arquitectura hay una clase padre llamada *BuildingController* que a su vez hereda de *Photon.MonoBehaviour*. Las clases hijas de *BuildingController* son *Tower*, de la que heredan todas las clases de las torres, y *CResourceBuilding*, de la que heredan todas las

clases que manejan recursos. Hasta ahora las clases mencionadas son abstractas, pero las clases hijas de *Tower* y de *CResourceBuilding* ya serán las que implementen los edificios.

Los scripts *TowerNeutral* y *TowerArmy* heredan de *Tower* y serán los que se añadan en Unity a las torres neutrales y torres goblin, y torres robot respectivamente. Por su parte, *BaseController* y *Warehouse* heredan de *CResourceBuilding* y serán los scripts que se añadan a las bases y almacenes de recursos, respectivamente.

El diagrama de clases se ha dividido en dos para mejorar su visibilidad: los edificios que manejan recursos (ver Figura 5.14) y las torres (ver Figura 5.15).

En la Figura 5.14 puede verse que la clase *CResourceBuilding* es la que se encarga de incrementar y decrementar recursos, además mediante *numEngineerPositions*, *engineerPositions* y *engineerPosTaken* gestiona las colas para construir, reparar y conquistar; también guarda una referencia a *ArmyController*. De ahí se ha dicho que heredan *Warehouse* y *BaseController:* la primera es para los almacenes que pueden ser construidos por las unidades ingenieras; la segunda es para las bases y son las que van a añadir nuevas unidades al jugador cuando se posean los suficientes recursos para ello.

En la Figura 5.15 puede verse que la clase *Tower* es la que se encarga de todos los ataques de la torre que va a instanciar, además tiene un atributo que dice si la torre puede o no ser conquistada. De ahí se ha dicho que heredan *TowerNeutral* y *TowerArmy*: la primera es para las torres neutrales, que debe ser conquistada por un ejército u otro y tiene un contador por cada equipo que la está conquistando hasta que pase a la posesión del primer equipo que llegue a *finalCont*; la segunda se usa para las torres goblin y la torres robot, que pueden ser construidas por las unidades ingenieras.

1. Construcción de edificios

Las unidades ingenieras son las que pueden construir estructuras, y son las encargadas de crear las instancias, ya sea una torre goblin, una torre robot o un almacén de recursos. Para ello, después del evento del teclado donde se le dice al ingeniero qué edificio hay que construir se llama a la función `SetCanConstruct (int item)` (ver Código 5.14) donde dependiendo del tipo de construcción, se instancia uno u otro elemento. En ambos casos se tiene que primero hacer *Instanciate*, después rotarlo en la posición óptima, pasarle el *teamNumber*, el *teamColorIndex* y la *BaseController* del ingeniero. En este momento se habrá instanciado un edificio nuevo pero no se estará construyendo, estará flotando con el *Material* transparente en la posición que apunta el ratón, para decidir con en qué zona construir;

Para decidir si se puede o no construir en un área determinada el *prefab* del edificio tiene un *collider* de construcción a la altura del suelo con *trigger* activado, de esta forma se sabe cuándo un objeto entra en este *collider* y en se sabe que no se puede construir (por ejemplo si hay un edificio dentro del *collider* no dejará construir ya que no se va a construir encima una de otra); además, el *prefab*, tiene una luz de construcción que cambiará el tono de color del edificio a rojizo si la posición en la que está actualmente es una zona donde no se puede construir y verdoso en caso de que sí se pueda.

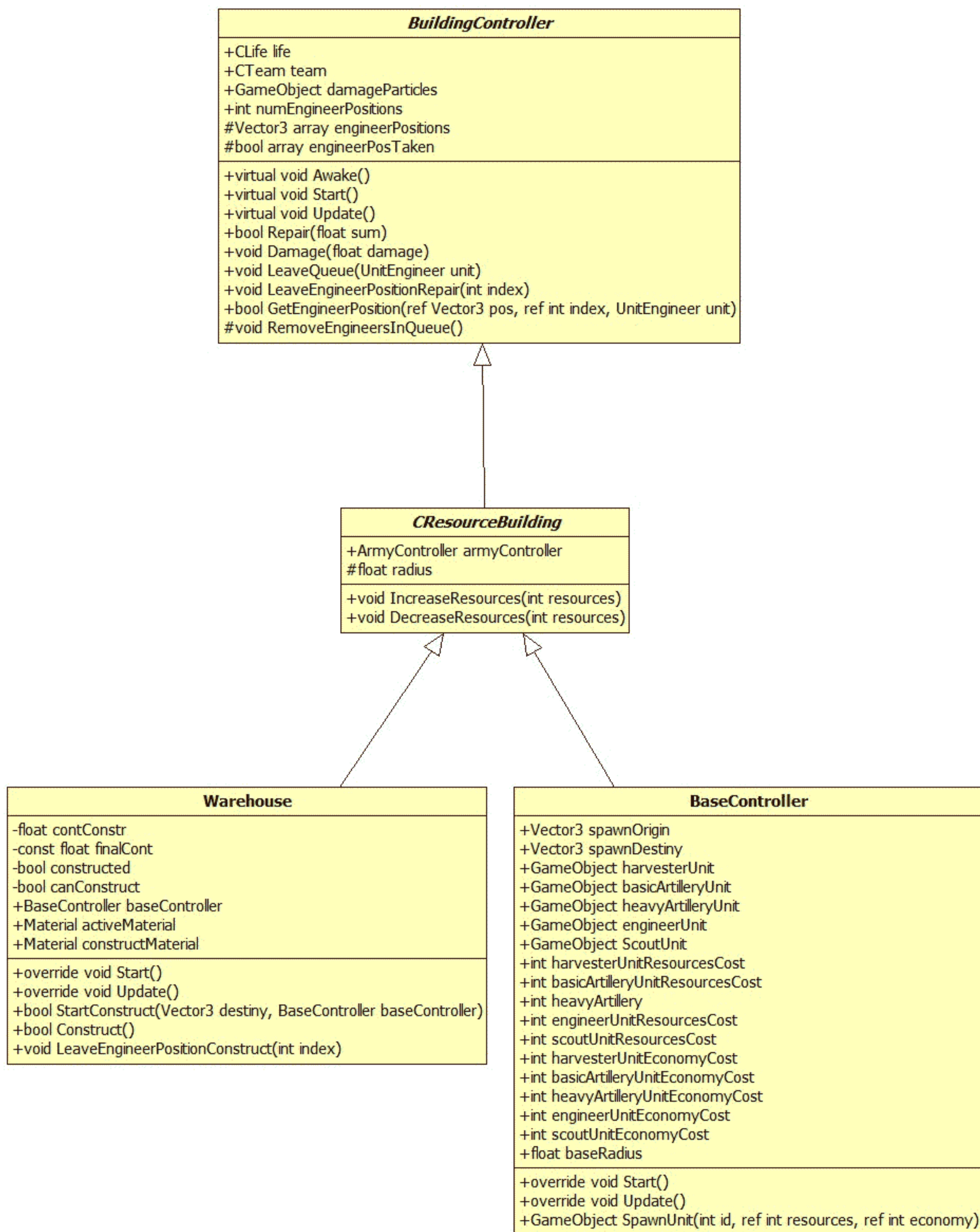


Figura 5.14: Diagrama de clases simplificado de los edificios de recursos del RTS.

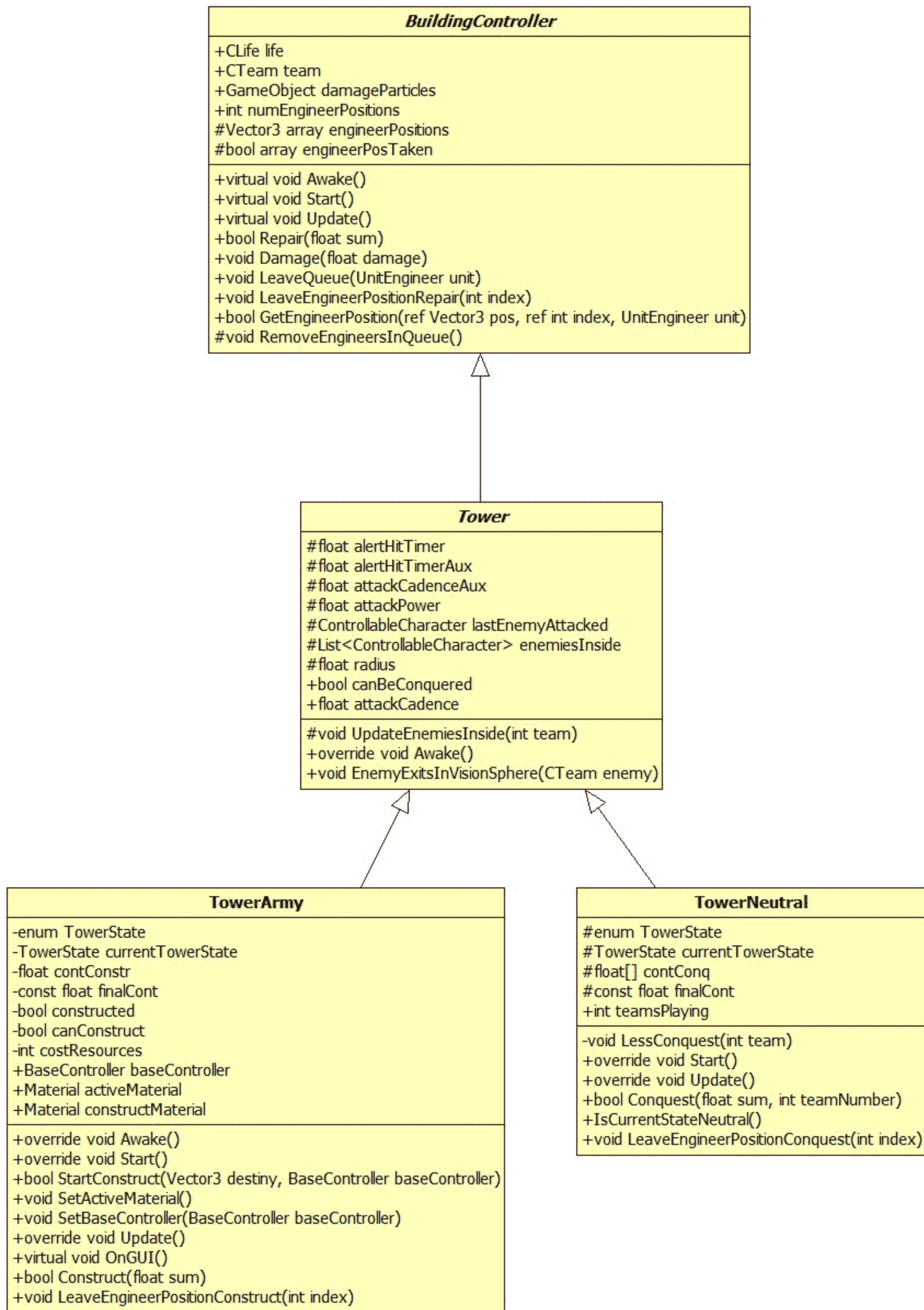


Figura 5.15: Diagrama de clases simplificado de las torres.

En el momento que se le dice al ingeniero que construya, se llamará a la función `StartConstruct (constructDestiny, baseController)` (ver Código 5.15) del edificio en cuestión. Si se puede construir, tanto porque la zona sea correcta como que se tienen los suficientes recursos para hacerlo, se realizan varias acciones: Se decrementan los recursos del equipo, se activa su componente *NavMeshObstacle*, se calculan los puntos de construcción/reparación del edificio y se destruyen los *GameObjects* innecesarios, que son el *collider* de construcción y la luz de construicción. En este momento estaría en construcción, ahora solo falta que los ingenieros acaben la construcción para que pueda estar activo el edificio.

Los edificios que pueden ser construidos tienen dos atributos de tipo *Material* que son *activeMaterial* y *constructMaterial*: el primero es un *Material* que define un *shader* con transparencia para usarlo en los edificios cuando se instancian y antes de ser construidos; el segundo es el *Material* que tendrá una vez construido.

```
public override void SetCanConstruct (int item)
{
    switch (item)
    {
        case 0:
            if (towerArmy)
                lastTowerArmy = towerArmy;
            if (warehouse)
                lastWarehouse = warehouse;
            towerArmy = Instantiate
            (
                towerArmyPrefab,
                transform.position + new Vector3(30,-30,30),
                new Quaternion(0, 0, 0, 0)
            ) as GameObject;
            towerArmy.transform.Rotate(270.0f, 0.0f, 0.0f);
            towerArmy.name = towerArmy.name.Replace("(Clone)", "");
            towerArmy.GetComponent<TowerArmy>().SetTeamNumber(this.teamNumber,
                                                      team.teamColorIndex);
          towerArmy.GetComponent<TowerArmy>().SetBaseController(baseController);
            newTGConstruct = true;
            break;
        case 1:
            if (towerArmy)
                lastTowerArmy = towerArmy;
            if (warehouse)
                lastWarehouse = warehouse;
            warehouse = Instantiate
            (
                warehousePrefab,
                transform.position + new Vector3(30, -30, 30),
                new Quaternion(0, 0, 0, 0)
            ) as GameObject;
            warehouse.transform.Rotate(0.0f, 180.0f, 0.0f);
            warehouse.name = warehouse.name.Replace("(Clone)", "");
            warehouse.GetComponent<Warehouse>().SetTeamNumber(this.teamNumber,
                                                      team.teamColorIndex);
          warehouse.GetComponent<Warehouse>().SetBaseController(baseController);
            newWConstruct = true;
            break;
    }
}
```

Código 5.14: función *SetCanConstruct* de los ingenieros para establecer la posibilidad de la construcción.

```
public bool StartConstruct (Vector3 destiny, BaseController baseController)
{
    if (canConstruct && baseController.GetResources() >= costResources)
    {
        canConstruct = false;
        this.baseController = baseController;
        baseController.DecreaseResources(costResources);
        Vector3 posN = transform.position;
        posN.y = 0;
        transform.position = posN;
        Vector3 vecCOll = GetComponent<CapsuleCollider>().transform.position;
        Vector3 vecnav = GetComponent<NavMeshObstacle>().transform.position;
        GetComponent<CSelectable>().enabled = true;
        this.GetComponent<NavMeshObstacle>().enabled = true;
        isActive = true;

        float twoPi = Mathf.PI * 2;
        Vector3 center = transform.position;
        for (int i = 0; i < numEngineerPositions; i++)
        {
            Vector3 pos = new Vector3
            (
                center.x +
                (radius + despPosition) * Mathf.Sin(i * (twoPi /
                                                   numEngineerPositions)),
                0,
                center.z +
                (radius + despPosition) * Mathf.Cos(i * (twoPi /
                                                   numEngineerPositions))
            );
            engineerPositions[i] = pos;
            engineerPosTaken[i] = false;

            cubes[i] = GameObject.CreatePrimitive(PrimitiveType.Cube);
            cubes[i].transform.position = pos;
            Destroy(cubes[i].GetComponent<BoxCollider>());
            cubes[i].renderer.material.color = new Color(0.196f, 0.804f,
0.196f);
            cubes[i].transform.parent = this.transform;

            cubes[i].SetActive(createCubes);
        }
        DestroyUnnecessaryGameobjects();
        return true;
    }
    return false;
}
```

Código 5.15: función *StartConstruct* de los ingenieros para establecer el comienzo de la construcción de edificios.

2. Conquista de Torres Neutrales

Las unidades ingenieras son también las que pueden conquistar estructuras, en concreto torres neutrales que se encuentran esparcidas por el escenario. Para realizar esta conquista, tiene que estar seleccionada una unidad ingeniera y al pulsar sobre una de estas torres en la función *RightClickOnSelected* del ingeniero tiene que comprobar que la torre no esté conquistada. En el momento que un ingeniero comienza a conquistarla llama a la función *Conquest* (ver Código 5.16). Para llevar la cuenta de la conquista de cada equipo sobre una torre hay un array de *floats* llamado *contConq*. En el momento que uno de los equipos deja de conquistarla, este contador va decreciendo poco a poco hasta que llega a cero o llega una

unidad ingeniera de ese equipo para continuar con la conquista; para manejar esto se creó un array de float llamado *actualTimeConquering* donde cada posición es la de un equipo y si llega a un número determinado por la constante *conqueringTime* empieza a decrementar el valor en su posición en *contConq*. Cuando uno de los equipos llegan al tope de conquista, dato que está almacenado en la constante *finalCont*, la torre pasa a ser posesión del equipo que la conquistó.

```
// Conquest is called by the engineers
public bool Conquest (float sum, int teamNumber)
{
    actualTimeConquering[teamNumber] = 0;
    contConq[teamNumber] += sum;
    if (contConq[teamNumber] >= finalCont)
    {
        for (int i = 0; i < teamsPlaying; i++)
            contConq[i] = 0;
        life.currentLife = 80.0f;

        // set the team values
        team.teamNumber = teamNumber;

        // insert the tower in the DistanceMeasurerTool
        DistanceMeasurerTool.InsertUnit(team);

        // change the state to Idle
        currentTowerState = TowerState.Idle;

        UpdateEnemiesInside(teamNumber);
        RemoveEngineersInQueue();
        for (int i = 0; i < numEngineerPositions; i++)
            cubes[i].renderer.material.color = new Color(0.196f, 0.804f,
                                                         0.196f);
        return true;
    }
    return false;
}
```

Código 5.16: función *Conquest* de la clase *NeutralTower*.

3. Reparación de Edificios

Todos los edificios pueden ser atacados, por lo tanto destruidos, y reparados. Las unidades ingenieras, son también las encargadas de reparar todas las estructuras. Cuando un ingeniero está seleccionado y se selecciona un edificio, tiene que ver, entre otras cosas, si el edificio es del mismo equipo y si no tiene la vida completa. En este momento el ingeniero va al edificio y comienza a repararla, llamando al método *Repair* de *BuildingController* (ver Código 5.17).

```
// Repair is called by the engineers
public bool Repair (float sum)
{
    // increasement of the towers life
    if (life.currentLife < life.maximunLife)
    {
        life.currentLife += sum;
        if (life.maximunLife < life.currentLife)
            life.currentLife = life.maximunLife;
    }
    if (life.currentLife == life.maximunLife)
    {
        RemoveEngineersInQueue();
```

```
        for (int i = 0; i < numEngineerPositions; i++)
            cubes[i].renderer.material.color = new Color(0.196f, 0.804f,
0.196f);
        return true;
    }
    else
        return false;
}
```

Código 5.17: función *Repair* de la clase *BuildingController*.

4. Almacenes de Recursos

Los almacenes de recursos pueden ser construidos por las unidades ingenieras y sirven para, una vez recolectados los recursos en las minas por las unidades recolectoras, poder llevarlos a estos edificios en vez de a la base. Al principio las unidades recolectoras llevan los recursos a la base, pero según se van construyendo almacenes de recursos se van calculando las distancias entre las minas y estos para saber cuál es la menor distancia desde las minas a los diferentes almacenes de recursos. La lógica se realiza en *ArmyController*.

5. Máquinas de Estados de las Torres

Las torres se comportan de una manera diferente al resto de edificios y es necesaria una máquina de estados para implementarla correctamente ya que pueden atacar, las torres neutrales se pueden conquistar y las torres goblin y robot pueden construirse.

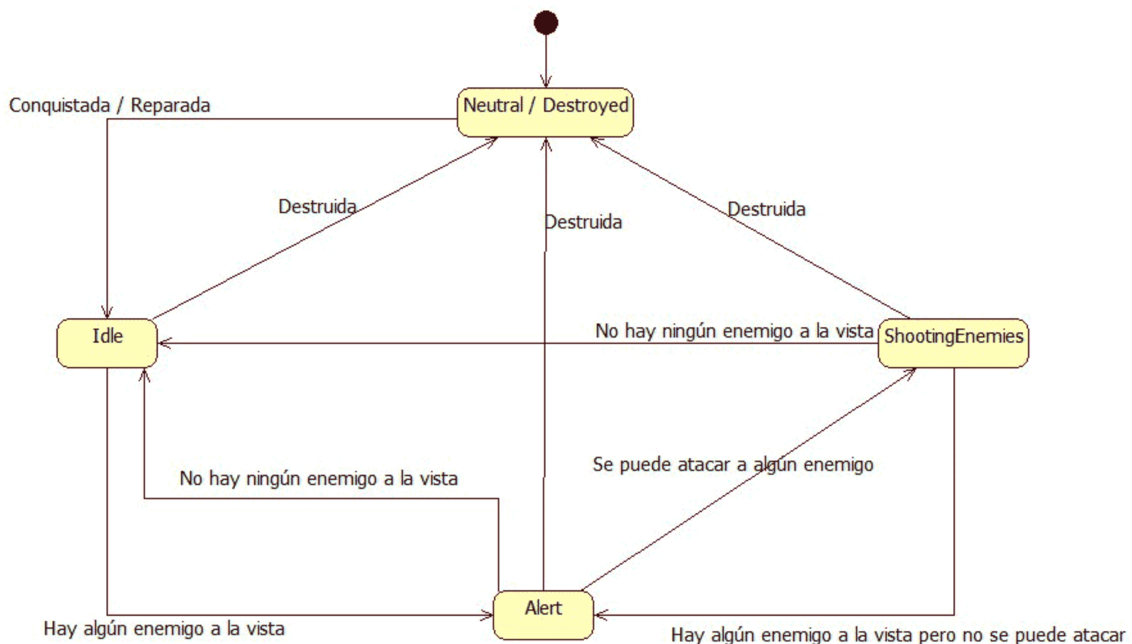


Figura 5.16: Diagrama de estados de las Torres.

iv. ArmyController

Es un componente que contiene el script *ArmyController*.cs, es uno de los scripts más importantes del juego, ya que se encarga de manejar todo lo relacionado a un equipo, y por lo tanto, habrá uno por cada equipo.

Es la encargada de llevar el *teamNumber* del equipo y el *teamColorIndex* para ver con qué color se tienen que colorear las unidades cuando se seleccionan. Mediante la lista *unitList* guarda todas las unidades del equipo en cuestión y mediante la lista *unitSelectedList* gestiona todas las unidades seleccionadas en un instante determinado, por lo que maneja todo lo relacionado con la selección de unidades del equipo (explicada en el apartado 6.a.i). Cada vez que se añade o elimina una unidad hay que actualizar, además de la lista *unitList*, el minimapa.

Lleva el control y la cuenta tanto de los recursos, como de la economía del equipo. También hace todo lo relacionado con la gestión de los almacenes de recursos (explicado en el apartado 5.a.iii.4).

También tiene como objetivo la gestión de las minas, los almacenes de recursos y la elección del mejor camino de retorno en la recogida de recursos. Para hacer esto se han creado una lista de *CResourceBuilding* para guardar los almacenes de recursos consruidos llamada *ResourceBuildingList*, una lista de *CResources* para guardar las minas encontradas llamado ResourceMineList y una referencia a la base; también se ha creado una clase privada llamada *ResourcesLinkStruct* que tiene como atributos un índice a un *CResourceBuiding*, otro a una mina y la distancia entre estos. Esta clase privada se ha creado para hacer una lista de elementos de este tipo llamada *BuildingMineLink*.

En el momento que a una unidad recolectora se le dice que vaya a una mina se llama al método *UpdateMines* de *ArmyController* (Código 5.18: Inserción de una nueva mina de recursos en la lista de minas de *ArmyController*), primero mira que la mina no esté ya en la lista *ResourceMineList* y si es así la añade primero y luego mira la distancia más corta entre esta mina y los almacenes de recursos, además de la base; encuentra la distancia mínima y crea un objeto de tipo *ResourceLinkStruct* con los datos calculados y lo inserta en la lista BuildingMineLink.

```
public void UpdateMines (Transform mineTransform)
{
    CResources mine = mineTransform.GetComponent<CResources>();
    if (!resourceMineList.Contains(mine))
    {
        resourceMineList.Add(mine);
        // Update Links
        Vector3 posMine = new Vector3(0, 0, 0);
        float dist = -1;
        float min = -1;
        int index = -1;
        int i = 0;
        posMine.x = mineTransform.position.x;
        posMine.z = mineTransform.position.z;
        foreach (CResourceBuilding c in resourceBuildingList) // foreach warehouses
        {
            Vector3 posBR = new Vector3(0, 0, 0);
            posBR.x = c.transform.position.x;
```

```
            posBR.z = c.transform.position.z;

            dist = Vector3.Distance(posMine, posBR);
            if (min < 0)
            {
                min = dist;
                index = i;
            }
            else if (dist < min)
            {
                min = dist;
                index = i;
            }
            i++;
        }
        if (min == -1)
            Debug.Log("Link NO realizado");
        else
        {
            // We have to create the link
            buildingMineLink.Add(new ResourcesLinkStruct(index, resourceMineList.Count
- 1, min));
            Debug.Log("Link realizado");
        }
    }
}
```

Código 5.18: Inserción de una nueva mina de recursos en la lista de minas de *ArmyController*.

Cuando se construye un almacén se llama a la función de *ArmyController* llamada *AddWarehouse* (Código 5.19), en esta función primero se mete el nuevo almacén en la lista *ResourceBuildingList* y se miran las minas y se calcula la distancia a ellas, guardándose la que tiene el camino más corto, de forma que se guarda en la posición donde se encuentra la mina de la lista ResourceMineLink el almacén y la distancia.

```
public void AddWarehouse (CResourceBuilding w)
{
    if (!resourceBuildingList.Contains(w))
    {
        resourceBuildingList.Add(w);
        Debug.Log("Warehouse agregada");
        // Link the nearest mine to this warehouse
        float dist;
        Vector3 posWare = new Vector3(w.transform.position.x, 0,
w.transform.position.z);
        Vector3 posMine = new Vector3(0, 0, 0);
        float currentDist = -1;
        int i = 0;
        foreach (CResources c in resourceMineList) // Foreach mines
        {
            // The distance to the current mine/CResourceBuilding
            bool found = false;
            int maxCount = buildingMineLink.Count;
            int j = 0;
            while ((j < maxCount) && !found)
            {
                if (buildingMineLink[j].mineIndex == i)// If the mine is found
                {
                    currentDist = buildingMineLink[j].distance;
                    found = true;
```

```
                }
                j++;
            }
            posMine.x = c.transform.position.x;
            posMine.z = c.transform.position.z;
            dist = Vector3.Distance(c.transform.position, posWare);

            if ((currentDist != -1) && (dist < currentDist)) // Update the new
warehouse to the mine
            {
                buildingMineLink[j - 1].distance = dist;
                buildingMineLink[j - 1].buildingIndex = resourceBuildingList.Count - 1;
            }
            i++;
        }

    }
    else Debug.Log("The warehouse has NOT been added");
}
```

Código 5.19: Inserción de un nuevo almacén de recursos en la lista de *ArmyController*.

La economía del equipo se va sumando con el paso del tiempo, definido con el atributo *timeToIncreaseEconomyAux*, se suma la cantidad definida con el otro atributo *increasedEconomy*.

Por último tiene como misión fundamental la gestión de los eventos del teclado y del ratón sobre las unidades y edificios del equipo:

- Con la tecla *A* pulsada se consigue cambiar el tipo de movimiento de las tropas seleccionadas a *attackMovement* (Código 5.20), de esta forma las unidades artilleras (ya sean *basic* o *heavy*) mientras van a una posición del mapa estarán alerta por si hay enemigos en su zona de visión para pararse a atacarles. Primero se lanza un rayo y se mira donde cae; si hay más de una unidad se calculan sus posiciones de llegada mediante el algoritmo de bandada (explicado en el apartado 5.a.v). Para acabar se llama al método de cada unidad seleccionada `AttackMovement(destinyList[i])`.

```
if (Input.GetKey(KeyCode.A))
{
    // attack movement mode
    // we launch a ray and receive where it hits
    myRay = Camera.main.ScreenPointToRay(Input.mousePosition);
    if (Physics.Raycast(myRay, out myHit, 1000f, layerMask))
    {
        Vector3 destiny = myHit.point;
        int count = unitSelectedList.Count;
        if (count > 1)
        {
            //The calculation depending on the number of different points of arrival
            List<Vector3> destinyList = SwarmAlgorithm(destiny);

            for (int i=0; i<count; i++)

unitSelectedList[i].GetComponent<UnitController>().AttackMovement(destinyList[i]);
        }
        else if (count == 1)
```

```
            unitSelectedList[0].GetComponent<UnitController>().AttackMovement(destiny);
    }
}
```

Código 5.20: Ejecución del movimiento con ataque automático en la clase *ArmyController*.

- Si se tiene al menos una unidad de tipo *Scout* seleccionada y se pulsa la tecla *P* entonces la unidad cambia su estado a *Patrolling* y se presupone que se comenzará la elección de puntos de patrulla de estas unidades. Ahora con el botón izquierdo del ratón y la tecla *P* pulsada se eligen los puntos de patrulla (ver Código 5.21): primero se mira si se ha alcanzado el máximo número de puntos posible, luego se mira si se ha pulsado en el suelo mediante el lanzamiento de un rayo desde la cámara hasta la posición del ratón, y si es el suelo entonces se agrega a la lista de puntos de patrulla llamada *patrolPosList*. Para finalizar con la elección de los puntos de patrulla basta con dejar de pulsar la tecla *P* (ver Código 5.21), en ese momento lo primero que se hace es mirar si todas las unidades que estaban seleccionadas eran *UnitScout* y si es así por cada una de ellas se llama al método `StartPatrol(patrolPosList)` para añadirles la lista de patrulla a todas y que comiencen a realizar la patrulla.

```
else if (Input.GetMouseButtonUp(0))
{
    // left click done
    if (keyPPressed)
    {
        // Selection of the patroll points of the scout Units
        if (patrolPosList.Count < maxPatrolPoints)
        {
            // we launch a ray and receive where it hits
            myRay = Camera.main.ScreenPointToRay(Input.mousePosition);
            if (
                 Physics.Raycast(myRay, out myHit, 1000f, layerMask) &&
                 (myHit.transform.name == "WorldFloor" ||
                  myHit.transform.name == "Terrain")
               )
            {
                //Debug.Log("New point");
                Vector3 destiny = myHit.point;
                patrolPosList.Add(destiny);
            }
        }
    }
    …
}
```

Código 5.21: Elección de puntos de patrulla para los exploradores en la clase *ArmyController*.

```
if (keyPPressed && Input.GetKeyUp(KeyCode.P))
{
    //Debug.Log("Finalize points");
    keyPPressed = false;
    bool allExplorers = true;
    foreach (GameObject u in unitSelectedList)
    {
        UnitScout unit = u.GetComponent<UnitScout>();
```

```
        if (!unit)
        {
            allExplorers = false;
            return;
        }
    }

    if (allExplorers)
    {
        foreach (GameObject u in unitSelectedList)
            u.GetComponent<UnitScout>().StartPatrol(patrolPosList);
        patrolPosList.Clear();
    }
}
```

Código 5.22: Finalización de elección de puntos de patrulla en la clase *ArmyController*.

- Con la tecla *T* y la tecla *W* se le ordena construir a un ingeniero que esté seleccionado una nueva Torre o un nuevo Almacén de Recursos respectivamente (ver Código 5.23). Primero se mira que sólo haya una unidad seleccionada y luego la tecla pulsada (*T* o *W*), después se comprueba que la unidad seleccionada sea una unidad ingeniera y, si es así, se manda construir al ingeniero el tipo de edificio deseado.

```
// If "T" or "W" is pulsed and there are only one engineer selected, it can construct
if (unitSelectedList.Count == 1)
{
    if (Input.GetKeyDown(KeyCode.T))
    {
        UnitEngineer unit = unitSelectedList[0].GetComponent<UnitEngineer>();
        if (unit != null && !unit.IsNewConstructing())
        {
            unit.SetCanConstruct(0);
        }
    }
    else if (Input.GetKeyDown(KeyCode.W))
    {
        UnitEngineer unit = unitSelectedList[0].GetComponent<UnitEngineer>();
        if (unit != null && !unit.IsNewConstructing())
        {
            unit.SetCanConstruct(1);
        }
    }
}
```

Código 5.23: Nueva Torre o nuevo Almacén de Recursos

- Aparición de nuevas unidades mediante las teclas 1, 2, 3, 4 y 5. Si está seleccionada la base del ejército, se pueden crear nuevas unidades con estas teclas. Se llama al método `SpawnUnit (int unit)`, que es encargada de comunicarle con el componente *baseController* de la base*,* ya que hay una referencia a este, para que cree así una nueva unidad del tipo elegido.
- Se cambia el modo de ataque de las unidades de tipo *heavyArtillery* mediante la tecla *D*.

v. Algoritmo de bandada

Los algoritmos de bandada resultan imprescindibles para un videojuego *RTS*, ya que se tratan de algoritmos que son útiles cuando se controla o se simula el comportamiento de múltiples unidades que forman un enjambre y se quieren, por ejemplo, enviar a la vez a todos los agentes del enjambre a un mismo destino, de manera que cada unidad se coloca en una posición diferente al resto en función del resultado del algoritmo.

El funcionamiento de las unidades de este proyecto resulta similar al enfoque de Lagrange[24], que es un modelo basado en agentes individuales que forman parte de un enjambre (al contrario que el enfoque Euleriano que trata el enjambre como un único campo[25]). Esto es así gracias al propio componente *Navemesh Agent* de Unity descrito en el capítulo 4.a.vii que permite la interacción entre las diferentes unidades que forman parte de un enjambre, incluidas el roce entre estas, empujones o la posibilidad de esquivarse mientras se están desplazando.

Se ha desarrollado una pequeña parte de lo que podría considerarse un algoritmo de bandada para la selección de múltiples destinos dado uno solo marcado por el jugador, cuando hay varias unidades seleccionadas de un mismo enjambre. Si no se usara este algoritmo, todas las unidades de las tropas seleccionadas intentarían ir a la misma posición (la marcada por el jugador), empujándose unas contra otras para intentar llegar al mismo punto, y no llegando ninguna al destino deseado.

Explicación del algoritmo

Este algoritmo (ver Código 5.24) es llamado cuando se han seleccionado múltiples unidades y se les ha indicado que se desplacen a una posición del mapa. Recibe un objeto de la clase *Vector3* llamado *destiny*, que es la posición de destino de las tropas (marcada por el jugador). Con esta posición, el número de tropas seleccionadas y la distancia por tropa (*alpha*) se intenta hacer un cuadrado perfecto y si no es así, lo más parecido a uno. Con estas variables se calcula el lado del cuadrado (solo se hace una raíz cuadrada ya que consume demasiado al ser una operación software) y se llega al bucle principal.

Hay dos casos especiales que son tratados aparte: cuando hay únicamente 2 o 3 unidades; si hay más unidades entra en el bucle principal. En este bucle **`foreach (GameObject u in unitSelectedList)`** se mira:

1. Si está construyendo el cuadrado
   a. Si no se está en el final de una fila: añade su posición con la última más *alpha*.
   b. Si es otra fila: se calcula la nueva fila y se añade la posición nueva.
2. Si ha acabado el cuadrado, tiene que ir rellenando fila por fila, a los lados, empezando por la fila 2. Pueden pasar tres cosas:
   a. Se añade a la izquierda de la fila:
      i. Si no es el último elemento del *unitSelectedList* se añade a la izquierda solo.

[24] http://ieeexplore.ieee.org/xpl/login.jsp?reload=true&tp=&arnumber=1489390&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D1489390

[25] http://mate.unipv.it/~toscani/publi/swarming.pdf

ii. Si es el último elemento de después de añadirse, se centra la fila.

b. Se añade a la derecha de la fila:

i. Se añade a la derecha y se centra la fila

c. Se añade una última fila: esto pasa cuando se han añadido dos unidades a cada fila (menos a la primera), entonces se crea una nueva fila al final y se van añadiendo.

3. Rotación del cuadrado-formación de las tropas en función de su dirección de origen y de la posición de destino: a continuación hay que rotar la formación al completo para que quede de la manera más coherente posible:

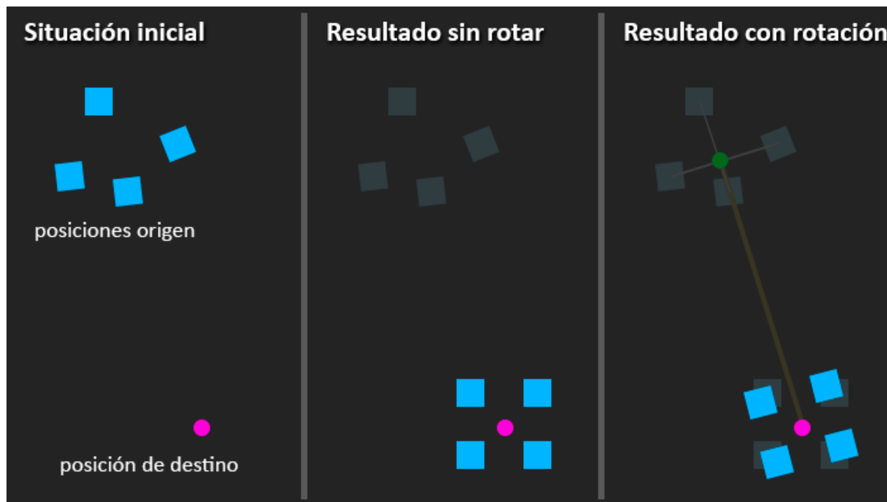


Figura 5.17: Ejemplo visual del algoritmo de rotación de posiciones de destino.

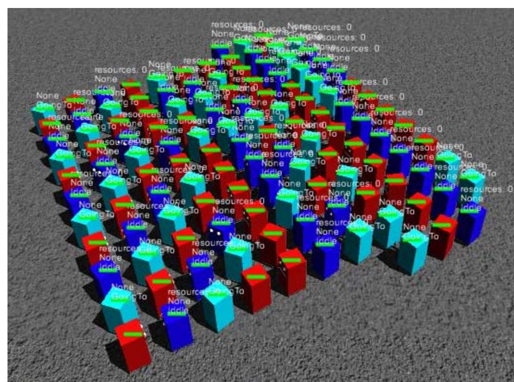
a. Se calcula la posición de origen media de las unidades.

b. Se calcula el ángulo del vector que une la posición media con la posición de destino.

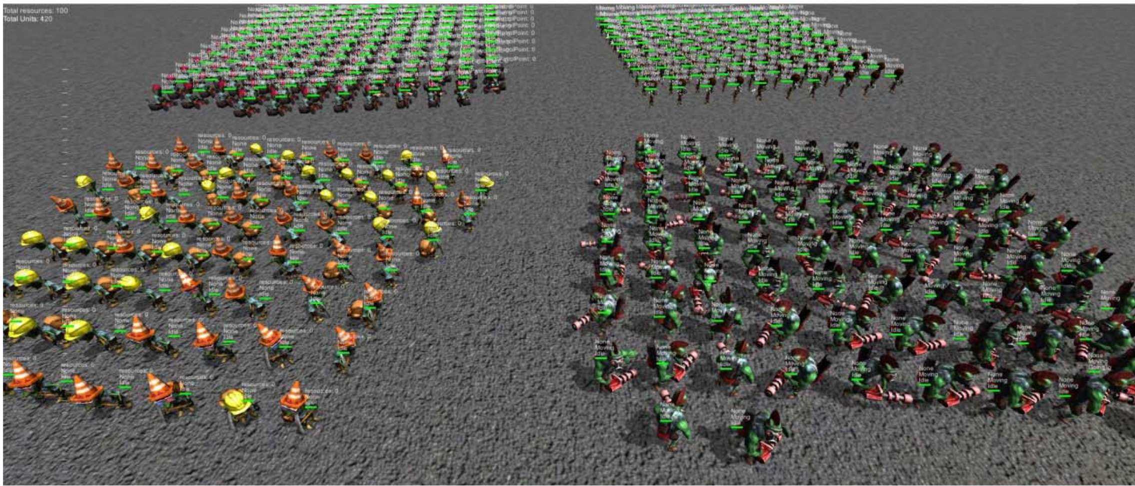
c. Se toma cada una de las posiciones calculadas previamente y se rotan la cantidad de grados calculada en b. con respecto a la posición de destino marcada por el jugador (cómo rotar un vector alrededor de un punto dado[26]).

Ejemplos de resultados:

En las siguientes imágenes se puede apreciar el resultado de este algormo en diversos ejemplos con distintas rotaciones generales de la formación y con un gran número de unidades seleccionadas.



[26] http://answers.unity3d.com/questions/532297/rotate-a-vector-around-a-certain-point.html



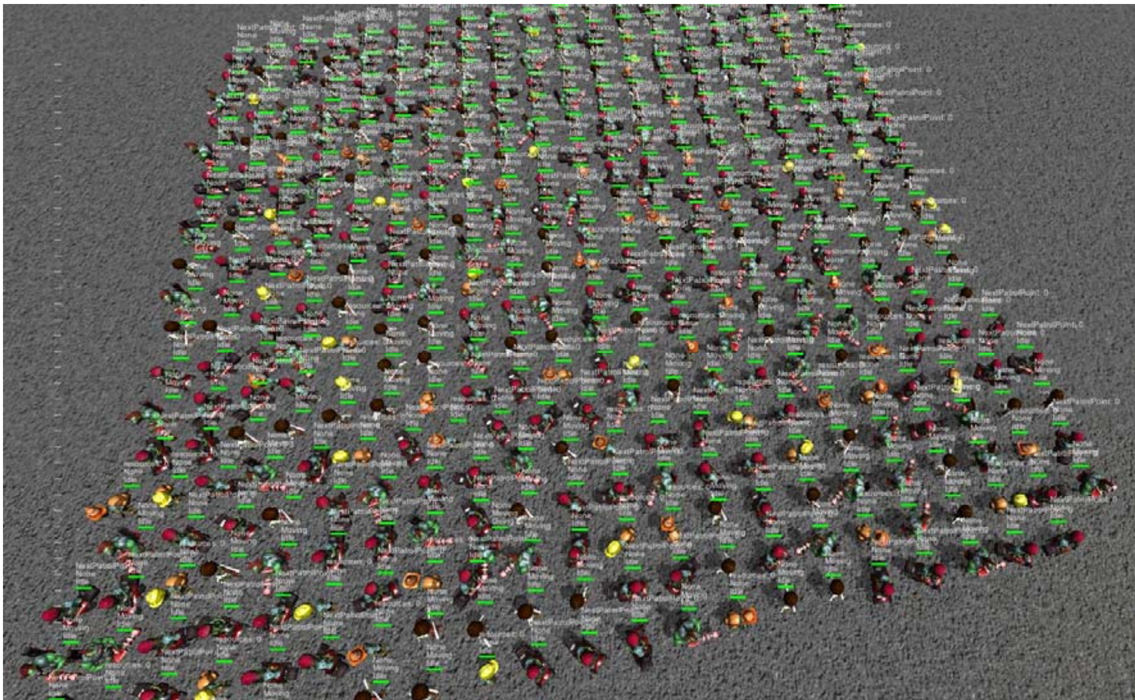


Figura 5.18: Ejemplos visuales del resultado final del algoritmo de cálculo de múltiples destinos.

Código fuente:

```
private List<Vector3> SwarmAlgorithm (Vector3 destiny)
{
    List<Vector3> destinyList = new List<Vector3>();
    double radious = System.Math.Sqrt(unitSelectedList.Count);
    int truncateRadious = (int)System.Math.Truncate(radious);
    Vector3 destinyAux = destiny;
    Vector3 origDestiny = destiny;
    //sup-left corner
    int squareNum = (truncateRadious * truncateRadious);
    int alpha = 2;
    float beta = 0.5f;
    int cont = 0;
    int row = 0;
    int contSqr = 0;
    bool first = true;
    int numSelected = unitSelectedList.Count;

    // to know how to center the trops
    if (numSelected % 2 != 0)
    {
        beta = 1;
    }
    origDestiny.x = destinyAux.x = destiny.x - (float)(truncateRadious/2) -
beta;
    if (numSelected == 2) // special case NumSelected = 2
    {
        foreach (GameObject u in unitSelectedList)
        {
            destinyList.Add(destinyAux);
            destinyAux.x += alpha;
        }
    }
    else if (numSelected == 3) // special case NumSelected = 2
    {
        foreach (GameObject u in unitSelectedList)
        {
            destinyList.Add(destinyAux);
            destinyAux.x += alpha;
        }
    }
    else
    {
        origDestiny.z = destinyAux.z = destiny.z + (float)(truncateRadious / 2)
                                        + beta;
        foreach (GameObject u in unitSelectedList)
        {
            cont++;
            contSqr++;
            if (squareNum + 1 > contSqr) // still doing the square
            {
                destinyList.Add(destinyAux);
                // next unit
                if (cont == truncateRadious) // there is still one row to made
                {
                    cont = 0;
                    destinyAux.z -= alpha;
                    destinyAux.x = origDestiny.x;
                }
                else // this is the last row
                {
```

```
                    destinyAux.x += alpha;
                }
            }
            else if ((squareNum + (2 * (truncateRadious - 1))) >= contSqr)
            // have to add to the rows left
            {
                // initialize in the left-sup corner
                if (squareNum + 1 == contSqr)
                {
                    destinyAux = origDestiny;
                }
                // is the first to going down, left to the orders
                if (first)
                {
                    row++;
                    destinyAux.z -= alpha; // down row
                    destinyAux.x = origDestiny.x - alpha; // going left
                    destinyList.Add(destinyAux);
                    if (contSqr == numSelected)
                    // if its the last -> center the row
                    {
                        Vector3 unity = new Vector3();
                        int min = truncateRadious * row;
                        int max = min + truncateRadious;
                        for (int i = min; i < max; i++)
                        {
                            unity = destinyList[i];
                            unity.x += alpha / 2;
                            destinyList[i] = unity;
                        }
                        unity = destinyList[destinyList.Count - 1];
                        unity.x += alpha / 2;
                        destinyList[destinyList.Count - 1] = unity;
                    }
                    first = false;
                }
                else // if its the second, it goes to the right,
                     // then center the row
                {
                    destinyAux.x += ((alpha) * truncateRadious) + 2; //Se
posiciona al final
                    destinyList.Add(destinyAux);
                    first = true;
                }
            }
            else // it is the last -> have to make a new row
            {
                destinyAux = origDestiny;
                destinyAux.z -= alpha * truncateRadious;
                for (int i = contSqr; i < numSelected + 1; i++)
                {
                    destinyList.Add(destinyAux);
                    destinyAux.x += alpha;
                }
            }
        }
    }

    // rotaion of the destinys in function of the middle of all the origins
    Vector3 origCenter;
    float minX = float.MaxValue,
          maxX = float.MinValue,
```

```
            minZ = minX, maxZ = maxX;
    for (int i = 0; i < numSelected; i++)
    {
        if (unitSelectedList[i].transform.position.x > maxX)
            maxX = unitSelectedList[i].transform.position.x;
        if (unitSelectedList[i].transform.position.z > maxZ)
            maxZ = unitSelectedList[i].transform.position.z;
        if (unitSelectedList[i].transform.position.x < minX)
            minX = unitSelectedList[i].transform.position.x;
        if (unitSelectedList[i].transform.position.z < minZ)
            minZ = unitSelectedList[i].transform.position.z;
    }
    origCenter = new Vector3( (maxX + minX) / 2.0f, 0.0F, (maxZ + minZ) / 2.0f);
    float angleDeg = Mathf.Atan2(destiny.z - origCenter.z, destiny.x -
origCenter.x) * Mathf.Rad2Deg;
    angleDeg = (angleDeg + lastCrowdAngle) % 90;
    Quaternion rot = Quaternion.Euler(0, -angleDeg + 90, 0);
    for (int i = 0; i < destinyList.Count; i++)
    {
        // get point direction relative to pivot
        Vector3 dir = destinyList[i] - destiny;
        dir = rot * dir; // rotate it
        destinyList[i] = dir + destiny; // calculate rotated point
    }
    return destinyList;
}
```

Código 5.24: Algoritmo de selección de múltimpes destinos.

vi. Navegación y Pathfinding

Como se ha comentado en el apartado de tecnologías dedicado a estudiar la navegación en Unity (4.a.vii), el modo de juego RTS necesita un sistema de navegación y pathfinding, por lo que se ha creado una malla de navegación o *navmesh* para que diferentes *Navmesh Agent* logren desplazarse por el mapa por caminos previamente buscados, y también se han creado en diferentes puntos del mapa diversos *Off-Mesh Links* para unidades especiales que pueden saltar por accidentes del terreno como ríos.

1. Navmesh

La malla de navegación se ha creado a partir de un terreno, el cual no es completamente plano. Por este terreno hay diferentes objetos a tener en cuenta a la hora de crear la malla de navegación: hay dos puentes en el escenario que cruzan el río y tienen que tener su malla de navegación propia, integrada y unida a la del terreno, para que las unidades y los héroes puedan cruzar el río; además hay dos canchas de baloncesto, un parking, dos parques con columpios y dos gasolineras, que están sobre el terreno y que, como los puentes, necesitan un *navmesh* propio y solapado con el terreno para que los personajes puedan moverse sin problema entre unos lugares y otros. Hay diferentes *assets* que no tienen que estar en el *navmesh* para que los personajes no puedan subirse a ellos y calculen rutas sobre ellos, como por ejemplo: coches, vallas, canastas de baloncesto, árboles, etc. Además, las bases, almacenes de recursos, minas y torretas, por el mismo motivo, no tienen *navmesh*.

Una vez creado el terreno, para crear la malla en Unity, antes de pulsar *bake* (apartado 4.a.vii.1) se ha tenido que seleccionar, en la jerarquía, todos los *GameObjects* que tienen que formar parte del *navmesh*.

2. Navmesh Agent

Todas las unidades locales, al contrario de los héroes (ya sean locales o remotos), necesitan el componente *Navmesh Agent* para poder desplazarse por la malla de navegación y poder ser manejadas por el jugador; las unidades remotas, como se explica en el apartado de red (5.c.i), no tienen este componente activado.

Se han usado diferentes funciones definidas en el apartado 4.a.vii.2:

- `bool FindClosestEdge(NavMeshHit hit);`

  Función usada para que las unidades, cuando se vayan a chocar con un muro, calculen el borde más próximo a éste para no hacerlo. Por ejemplo, si está yendo de frente hacia el muro, calculará su punto más cercano al muro e irá a esa posición:

  ```
  if (this.FindClosestEdge(out hit))
      this.GoTo(hit.position);
  ```

- `bool Raycast(Vector3 targetPosition, NavMeshHit hit);`

  Función usada para saber si hay un obstáculo entre una unidad y un punto determinado, que suele ser el destino de un camino:

  ```
  NavMeshHit hit;
  if (!agent.Raycast(target.position, out hit))
      this.GoTo(hit.position);
  ```

La función `GoTo (Vector3 destiny)` de *UnitController.cs*, da valor al atributo *destination* del componente *NavMeshAgent* para que una unidad comience a calcular las rutas posibles para poder elegir la más corta y ventajosa. Se resume en:

```
public void GoTo (Vector3 destiny)
{
    this.destiny = destiny;
    GetComponent<NavMeshAgent>().destination = destiny;
    currentState = State.GoingTo;
}
```

Código 5.25: función *GoTo* de la clase *UnitController* que establece el destino al componente *NavMeshAgent*.

3. Navmesh Obstacle

Los héroes no tienen que tener un sistema de navegación, pero sí tienen que influir en los caminos de los que sí lo tienen y, por eso, tienen un componente *Navmesh Obstacle.* Por ejemplo, si un héroe está en mitad del puente y no deja espacio para que una unidad pase, cuando se le diga a una unidad que cruce el puente tiene que saber que hay un obstáculo: el propio héroe.

4. OffMesh Link

Como se ha sido comentado anteriormente, todas las unidades del modo de juego RTS tienen un sistema de navegación. Sólo unas podrán tanto saltar el río como charcos: las unidades exploradoras. Para que puedan realizar esta acción, en la ventana *Navigation*, pestaña *Objects,* al atributo *Navigation Layer* se da el valor *jump*. De esta forma los enlaces que se crean estarán en la capa jump. Todas las unidades, excepto la exploradora, en el componente *Navmesh Agent,* el valor del atributo *NavMesh Walkable* es *Default*, el cual no incluye la capa jump, algo que sí hace el valor *Everithing*, que es el que se le da a las unidades exploradoras para que puedan atravesar, tanto la capa del *navmesh Default*, como la *jump*.

A la hora de crear la malla, se ha tenido que meter un plano como obstáculo justo encima del fondo del río para crear los enlaces. Si no se hiciera, cuando se crea el *navmesh* se generarían enlaces desde el borde del río al fondo de este (ver Figura 5.19).

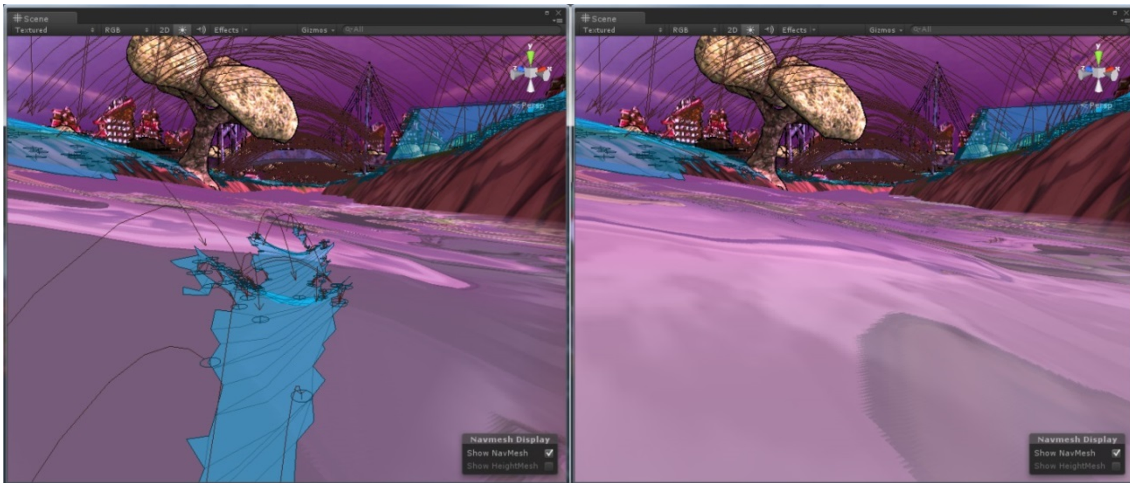


Figura 5.19: Creación del *OffMeshLink* en uno de los escenarios del juego.

vii. Cámara RTS

Para el control de la cámara del modo de juego RTS se hace uso del script *CameraRTSController.cs*. Esta cámara se caracteriza por:

- Se mueve libremente por el escenario.
- Utiliza un modo de vista casi cenital, aunque no llega a ser completamente perpendicular al suelo, de modo que el escenario se ve ligeramente inclinado.
- Se permite hacer zoom, aunque la vista es más paralela al escenario cuando está situada más cerca del mismo, de modo que se ve más el horizonte.

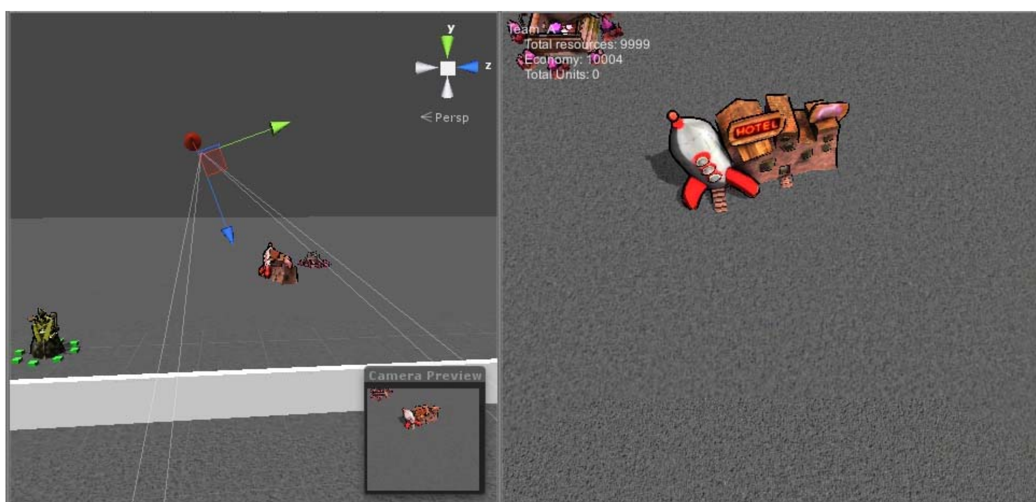


Figura 5.20: Cámara RTS sin zoom.

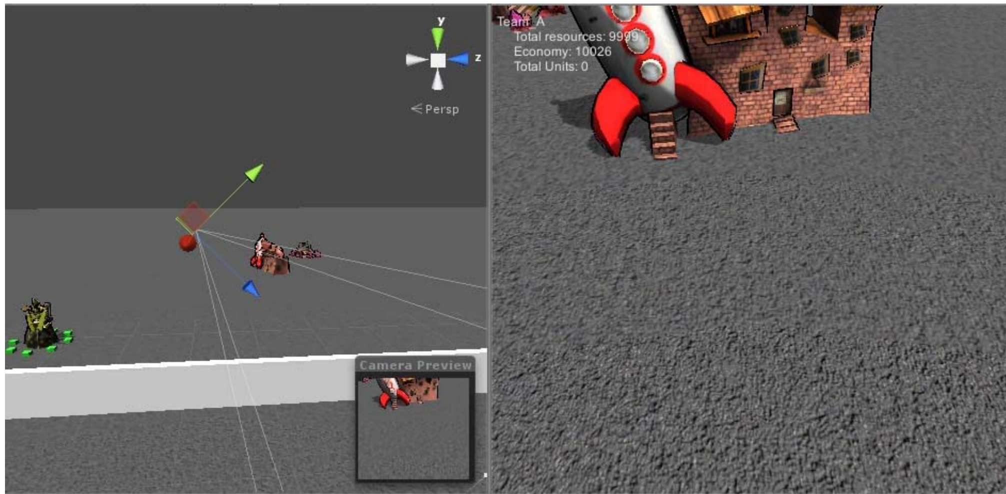


Figura 5.21: Cámara RTS con zoom.

CameraRTSController

El script *CameraRTSController.cs* está parametrizado de manera que desde Unity se puede configurar de forma sencilla (Figura 5.22). Los atributos que cabe destacar son:

- **XMinMax**: esta variable limita el movimiento de la cámara en el eje x.
- **ZMinMax**: esta variable limita el movimiento de la cámara en el eje z.
- **Scroll Speed**: esta variable indica la velocidad de movimiento de la cámara cuando se mueve con el teclado.
- **Scroll Edge**: esta variable se utiliza en el desplazamiento de la cámara mediante el desplazamiento del ratón, y su función es indicar a partir de la zona en que se empieza a desplazar con respecto al borde.
- **Pan Speed**: esta variable indica la velocidad de movimiento de la cámara cuando se mueve manteniendo pulsada la rueda del ratón.
- **Zoom Range**: esta variable limita el zoom de la cámara, es decir, lo que se puede alejar y acercar al escenario.
- **Zoom Speed**: esta variable indica la velocidad de movimiento de la cámara a la hora de hacer zoom, es decir, la velocidad a la que se acerca o se aleja.

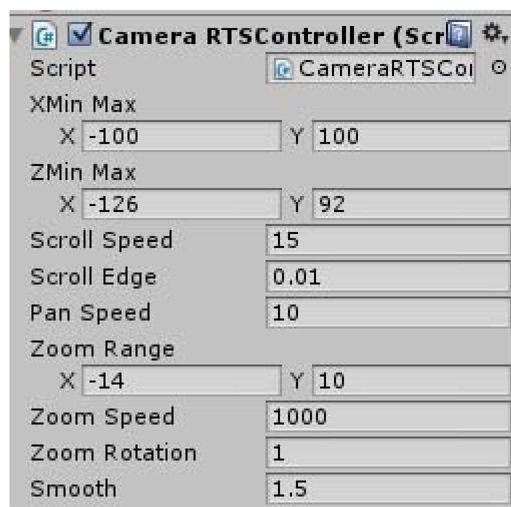


Figura 5.22: Variables del script *CameraRTSController.*

Además, este script permite realizar las siguientes acciones con la cámara:

- Presionando la tecla *Left Shift* la cámara se mueve al doble de velocidad (ver Código 5.26).
- Presionando la rueda del ratón la cámara se mueve por el escenario (ver Código 5.27).
- Mediante el teclado se puede mover la cámara por el escenario (ver Código 5.28).
- Desplazando la rueda del ratón se modifica el zoom de la cámara (ver Código 5.29).

```
if (Input.GetKey(KeyCode.LeftShift))
    scrollSpeedMult = scrollSpeed * 2.0f;
else
    scrollSpeedMult = scrollSpeed;
```

Código 5.26: Velocidad aumentada del desplazamiento en los ejes X y Z de la cámara.

```
if (Input.GetMouseButton(2))
{
    transform.Translate
    (
        Vector3.right * Time.deltaTime * panSpeed *
        (Input.mousePosition.x - Screen.width * 0.5f) / (Screen.width *
            0.5f),
        Space.World
    );
    transform.Translate
    (
        Vector3.forward * Time.deltaTime * panSpeed *
        (Input.mousePosition.y - Screen.height * 0.5f) / (Screen.height *
            0.5f),
        Space.World
    );
}
```

Código 5.27: Movimiento de la cámara por medio del botón central (rueda) del ratón.

```
else
{
    if ((Input.GetKey(KeyCode.UpArrow) ||
        (Input.mousePosition.y >= Screen.height * (1 - scrollEdge)))
        && (transform.position.z < zMinMax.y))
    {
        transform.Translate(Vector3.forward * scrollSpeedMult *
            Time.deltaTime, Space.World);
    }
    else if ((Input.GetKey(KeyCode.DownArrow) ||
        (Input.mousePosition.y <= Screen.height * scrollEdge))
        && (transform.position.z > zMinMax.x))
    {
        transform.Translate(Vector3.forward * -scrollSpeedMult *
            Time.deltaTime, Space.World);
    }

    if ((Input.GetKey(KeyCode.RightArrow) ||
        (Input.mousePosition.x >= Screen.width * (1 - scrollEdge)))
```

```
        && (transform.position.x < xMinMax.y))
    {
        transform.Translate(Vector3.right * scrollSpeedMult *
            Time.deltaTime, Space.World);
    }
    else if (( Input.GetKey(KeyCode.LeftArrow) ||
        (Input.mousePosition.x <= Screen.width * scrollEdge))
        && (transform.position.x > xMinMax.x))
    {
        transform.Translate(Vector3.right * -scrollSpeedMult *
            Time.deltaTime, Space.World);
    }
}
```

Código 5.28: Movimiento de la cámara por medio del teclado.

```
// Zoom in-out
currentZoom -= Input.GetAxis("Mouse ScrollWheel") * Time.deltaTime * zoomSpeed;
currentZoom = Mathf.Clamp(currentZoom, zoomRange.x, zoomRange.y);

transform.position = new Vector3
 (
    transform.position.x,
    transform.position.y - (transform.position.y - (initPosition.y +
        currentZoom)) * 0.1f,
    transform.position.z
);
transform.eulerAngles = new Vector3
(
    transform.eulerAngles.x - (transform.eulerAngles.x - (initRotation.x +
       currentZoom * zoomRotation)) * 0.1f,
    transform.eulerAngles.y,
    transform.eulerAngles.z
);
```

Código 5.29: Zoom de la cámara.

El diagrama UML del *script* se muestra a continuación:

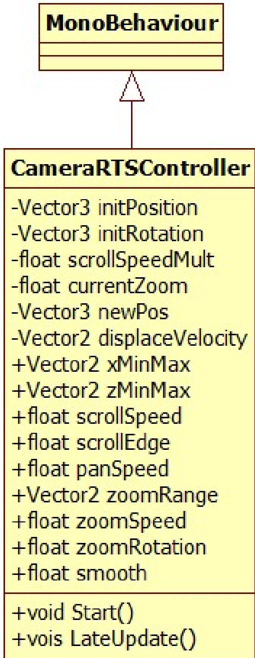


Figura 5.23: UML del *script CameraRTSController*.

viii. Herramienta para la medición de distancias

Uno de los atributos de las unidades de los ejércitos es un *float* que determina el área de visión de la unidad, es decir, cuanto mayor es este atributo, la unidad tendrá una capacidad de detectar, por ejemplo, enemigos que estén colocados a una mayor distancia.

En un principio, esta funcionalidad de detección de enemigos se desarrolló usando componentes físicos de Unity (ver apartado de físicas en Unity 4.a.iv), tales como una burbuja de visión (con un *Capsule Collider*) que, unido a un componente *RigidBody* en el propio modelo 3D de la unidad (ver Figura 5.24), permiten detectar colisiones con unos simples métodos (*OnTriggerEnter*, para cuando otro colisionador/*RigidBody* entra en el propio; *OnTriggerExit*, para cuando sale; *OnTriggerStay*, que se llamará en cada *frame* que esté dentro).

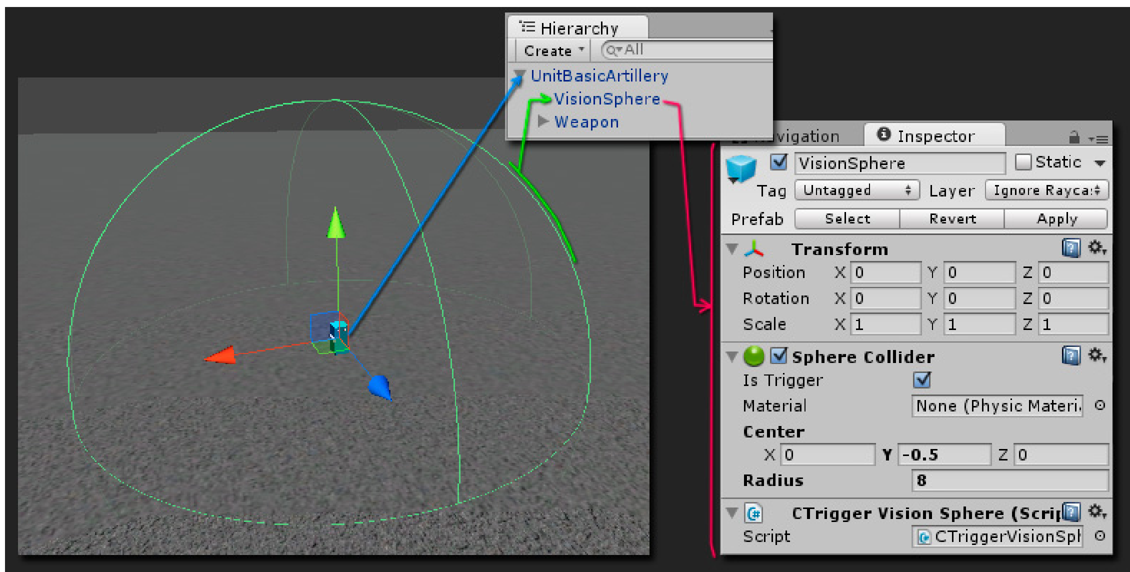


Figura 5.24: Aspecto de la burbuja de visión en una unidad en el editor de Unity.

La burbuja de visión lleva adjunto el *script* (llamado *CTriggerVisionSphere*) que se encarga de recoger el disparador que se produce en la colisión:

```
public class CTriggerVisionSphere : MonoBehaviour
{
    void OnTriggerEnter (Collider other)
    {
        CTeam unit = other.transform.GetComponent<CTeam>();
        if (unit != null)
        {
            UnitArtillery selfUnit =
                            transform.parent.GetComponent<UnitArtillery>();
            if ( (selfUnit != null) &&
                 (selfUnit.GetTeamNumber() != unit.teamNumber) )
            {
                selfUnit.EnemyEntersInVisionSphere(unit);
            }
        }
    }

    void OnTriggerExit (Collider other)
    {
        CTeam unit = other.transform.GetComponent<CTeam>();
        if (unit != null)
```

```
        {
            UnitArtillery selfUnit =
                          transform.parent.GetComponent<UnitArtillery>();
            if ( (selfUnit != null) &&
                 (selfUnit.GetTeamNumber() != unit.teamNumber) )
            {
                selfUnit.EnemyLeavesVisionSphere(unit);
            }
        }
    }
}
```
Código 5.30: Clase *CTriggerVisionSphere* que detecta las colisiones entre unidades.

Estos métodos únicamente se encargan de comprobar que el Objeto de Juego que ha entrado en la burbuja sea un enemigo, y de ser así, se lo comunica al *Script* que contiene el comportamiento de la propia unidad.

Durante el desarrollo de pruebas de carga surgió un importante problema de rendimiento de la CPU debido a la detección de múltiples colisiones. Supongamos, por ejemplo, dos ejércitos de 100 unidades cada uno. Si las unidades están suficientemente cerca, se puede dar el caso de que todas las áreas de visión de las unidades de una facción estén en colisión con todos los componentes *RigidBody* de las unidades del ejército enemigo, lo que supone 20.000 colisiones (100 unidades x 100 unidades x 2 ejércitos). Además, este método de comprobar las colisiones tiene una complejidad de orden $O(n^2)$, creciendo de forma cuadrática hasta n^2 x m colisiones (siendo n el número máximo de unidades que puede tener un ejército y m el número de ejércitos presentes en una partida, normalmente 2).

Esto hacía que la tasa de refresco del juego cayera de forma muy abrupta, por lo que se opta por establecer otra solución que no incluya el uso de componentes físicos, quizás menos cómoda y evidente, pero mucho más liviana en términos de carga de CPU.

Resumen del algoritmo

La idea principal es mantener una matriz de distancias entre todas las unidades que haya en una partida de manera que las filas de la matriz representan a las unidades de un ejército y las columnas las de otro. Así, *Matriz[i][j]* contendrá la distancia de separación entre la unidad *i* del ejército de un equipo y la unidad *j* del ejército contrario.

Calcular la distancia entre dos unidades resulta sencillo, ya que se dispone en todo momento de una variable de tipo *Vector3* que representa la posición de un *GameObject* en el mundo. Sin embargo, rellenar esta matriz constantemente midiendo las distancias euclídeas resulta demasiado costoso, puesto que es necesario hacer una raíz cuadrada y varias multiplicaciones en coma flotante por cada distancia, y la matriz puede llegar a tener un tamaño bastante considerable (hasta un máximo de n x n, siendo n el número máximo de unidades que puede tener un ejército: si n = 200, entonces la matriz podrá tener un máximo de 40.000 casillas). Por lo tanto podrían llegar a ser necesarias n x n raíces cuadradas en cada ciclo de juego (40.000 según el caso en el que n sea 200, lo que supondría demasiada carga para la CPU y se comprometería la fluidez del juego).

De esta manera, se opta por varias alternativas simultáneas para relajar la actualización de los valores de la Matriz:

- Limitar el número de casillas que se actualizan en cada ciclo del juego.
- Cuando se mira una casilla se descartan los casos más fáciles de comprobar (distancias en las coordenadas del plano del suelo *x* y *z*) con dos simples restas, y si éstas son menores al área de visión de las unidades, ya se comprueba la distancia euclídea.

Además, la matriz se recorre de forma bidimensional y alternando casillas pares e impares.

La clase DistanceMeasurerTool

Esta clase se compone, básicamente, de varias estructuras para guardar las referencias a todas las unidades presentes en el nivel del juego y poder así controlar sus posiciones para detectar las distancias entre todas. De esta manera, se las puede "avisar" cuando tienen otras unidades "a vista". Además de estas listas, tiene una matriz de bidimensional de *floats* con las distancias que va calculando entre todas las unidades, es decir, esta matriz tendrá un tamaño de NxM dónde N es el número de unidades de un equipo y M el del otro.

Aparte de esto, el aspecto más importante de esta clase se encuentra en la forma de rellenar esta matriz de distancias, que se tiene que hacer de forma lo menos costoso posible sin comprometer la jugabilidad (cosa que ocurriría si por ejemplo, una unidad tarda mucho tiempo en detectar a unidades enemigas que tenga en su área de visión y el jugador se percate de este retardo).

Estructuras:

- *Army0*: lista de referencias a las unidades del equipo 0.
- *Army1*: lista de referencias a las unidades del equipo 1.
- *distancesMatrix*: matriz de números reales que indican la distancia de separación entre dos unidades.
- *maxCalculationsPerUpdate*: el número máximo de cálculos de distancias que se harán en cada ciclo de juego. Es un número que representa la raiz cuadrada del número real de cálculos que se realizan ya que al recorrerse la matriz de forma bidimensional este valor se utiliza tanto para el número de filas a calcular como para el número de columnas.
- *calculationsPerUpdate*: valor auxiliar para cuando el número de unidades en el juego, y por lo tanto el número de casillas de la matriz *distancesMatrix,* es menor al atributo *maxCalculationsPerUpdate*.
- *searchMode*: modo de búsqueda actual. Se alterna entre par-par, par-impar, impar-par, impar-impar, o *no_search* cuando la búsqueda no es necesaria, por ejemplo, cuando en el juego solo hay unidades de un ejército.
- *prevIndexI* y *prevIndexj*: indica el índice de la fila y la columna de la matriz de distancias donde ha empezado la última búsqueda.

```
// List of references at all the units of the Team 0
public static List< CTeam > Army0 = new List< CTeam >();
// List of references al all the units of the Team 1
public static List< CTeam > Army1 = new List< CTeam >();

// rows = Team 0, cols = Team 1
```

```
// Example distancesMatrix[2][3] is the distances between the units Army0[2]
// and Army1[3]
public static List< List<float> > distancesMatrix = new List<List<float>>();

private static int maxCalculationsPerUpdate = 2;
private static int calculationsPerUpdate = 1;

private enum SearchMode
{
    no_search,
    pair_pair,
    pair_odd,
    odd_pair,
    odd_odd
};
private static SearchMode searchMode = SearchMode.no_search;

private static int prevIndexi = 0,
                   prevIndexj = 0;
```

Código 5.31: Atributos de la clase *DistanceMeasurerTool*.

Búsqueda

Cada etapa de búsqueda se realiza en el método *LateUpdate* (véase el capítulo sobre el ciclo de juego en Unity 4.a.iii) para asegurarse de que se han actualizado las posiciones de todas las unidades antes de calcular las distancias. Aquí se van alternando los modos de búsqueda (atributo *searchMode*), se incrementan los valores de los índices (atributos *prevIndexi* y *prevIndexj*) y se llama al método que realiza el paso de actualización de distancias.

```
private void SearchStep ()
{
    int list0Count = Army0.Count,
        list1Count = Army1.Count;

    for (int i = prevIndexi; i <= calculationsPerUpdate + prevIndexi;
         i += 2)
    {
        for (int j = prevIndexj; j <= calculationsPerUpdate + prevIndexj;
             j += 2)
        {
            //Debug.Log("Búsqueda: i:" + i + ", j:" + j + ".");
            int iAux = i % list0Count,
                jAux = j % list1Count;

            CTeam unit0 = Army0[iAux],
                  unit1 = Army1[jAux];

            if (unit0 && unit1)
            {
                // discard cases for distances x and z
                float auxDist = Mathf.Abs(unit0.transform.position.x -
                                          unit1.transform.position.x);
                float prevDist = distancesMatrix[iAux][jAux];
                if (auxDist < unit0.visionSphereRadius ||
                    auxDist < unit1.visionSphereRadius)
                {
                    auxDist = Mathf.Abs(unit0.transform.position.z -
                                        unit1.transform.position.z);
                    if (auxDist < unit0.visionSphereRadius ||
                        auxDist < unit1.visionSphereRadius)
                    {
```

```
                        // the new distance is calculated
                        float newDist = Vector3.Distance
                        (
                            unit0.transform.position,
                            unit1.transform.position
                        );
                        distancesMatrix[iAux][jAux] = newDist;
```

Código 5.32: Primera parte del algoritmo de búsqueda de la clase *DistanceMeasurerTool*.

Esta búsqueda recorre tantas filas y columnas de la matriz como indique el atributo *calculationsPerUpdate* alternando dos posiciones (el índice de los bucles se incrementa en 2 en cada paso). En cada paso del doble bucle se recogen las referencias de las unidades correspondientes y se calcula su distancia de la siguiente forma:

1. Se comprueba la distancia en la coordenada x. Si es menor que el área de visión de una de las unidades se pasa al paso 2, si no, ir al paso 7.
2. Se comprueba la distancia en la coordenada z. Si es menor que el área de visión de una de las unidades se pasa al paso 3, si no, ir al paso 7.
3. Se calcula la nueva distancia entre ambas unidades con el método *Vector3.Distance*[27] de Unity. Este método retorna la distancia euclídea entre dos estructuras *Vector3*.
4. Se actualiza la casilla de la matriz de distancias con el nuevo valor calculado en 3.

```
                        if (prevDist >= unit0.visionSphereRadius &&
                                newDist <= unit0.visionSphereRadius)
                            unit0.EnemyEntersInVisionSphere(unit1);
                        else if (prevDist < unit0.visionSphereRadius &&
                                     newDist > unit0.visionSphereRadius)
                            unit0.EnemyLeavesVisionSphere(unit1);

                        if (prevDist >= unit1.visionSphereRadius &&
                                newDist <= unit1.visionSphereRadius)
                            unit1.EnemyEntersInVisionSphere(unit0);
                        else if (prevDist < unit1.visionSphereRadius &&
                                     newDist > unit1.visionSphereRadius)
                            unit1.EnemyLeavesVisionSphere(unit0);
                    }
                    else
                    {
                        distancesMatrix[iAux][jAux] = float.MaxValue;
                    }
                }
                else
                {
                    if (prevDist <= unit0.visionSphereRadius)
                        unit0.EnemyLeavesVisionSphere(unit1);
                    if (prevDist <= unit1.visionSphereRadius)
                        unit1.EnemyLeavesVisionSphere(unit0);

                    distancesMatrix[iAux][jAux] = float.MaxValue;
                }
            }
        } // for j
    } // for i
} // SearchStep
```

Código 5.33: Segunda parte del algoritmo de búsqueda de la clase *DistanceMeasurerTool*.

[27] http://docs.unity3d.com/ScriptReference/Vector3.Distance.html

5. Si la distancia previa de la matriz era mayor al área de visión de una unidad y la nueva distancia calculada es menor, significa que la unidad enemiga antes no estaba dentro del área de visión y que ahora sí, y por lo tanto, hay que comunicárselo con el método *EnemyEntersInVisionSphere*.
6. Sin embargo, si la distancia previa era menor al área de visión y la nueva es mayor, significa que antes el enemigo sí que estaba "a la vista" y que ha salido del área de visión. Se le comunica con el método *EnemyLeavesVisionSphere*. **Fin del paso del bucle**.
7. Se comprueba si la distancia previa era menor al área de visión, lo que, de nuevo, indicaría que el enemigo estaba "a la vista" pero ahora ya no. Se le comunica con el método *EnemyLeavesVisionSphere*.
8. Si se va al paso 7 significa que no ha sido necesario calcular la distancia euclídea. Puesto que el caso se ha descartado previamente, actualizamos el valor de la matriz con infinito (*float.MaxValue*).

Además, existen dos métodos auxiliares para cuando se insertan nuevas unidades en el juego: *InsertUnit* (que inserta la unidad en la lista correspondiente además de incrementar en uno el número de filas o columnas de la matriz de distancias) y para cuando se eliminan unidades del juego: *DeleteUnit*.

Ejemplo

Se dispone de un escenario en el que hay dos ejércitos compuestos por 20 unidades de artillería básica cada uno. Estas unidades tienen un área de visión de 10 unidades métricas (unos 10 metros), las filas de unidades más cercanas sí que se verán entre ellas, pero no las 2 últimas filas de cada ejército que quedarán al margen de sus enemigos (la pared dispuesta entre los ejércitos se coloca para que las unidades no empiecen a dispararse):

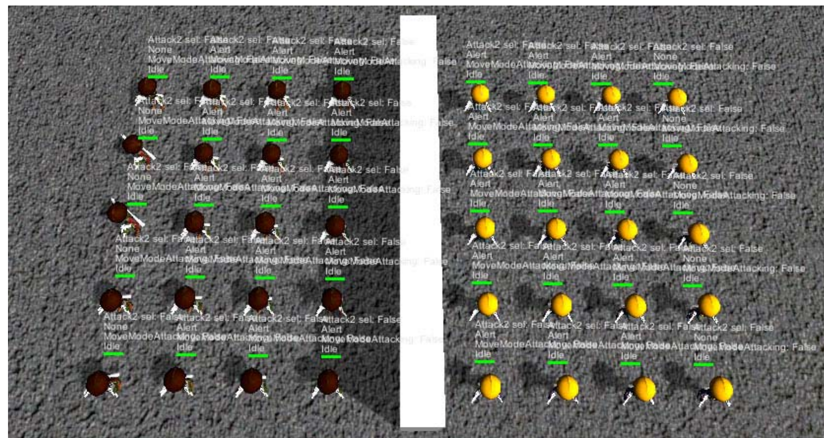
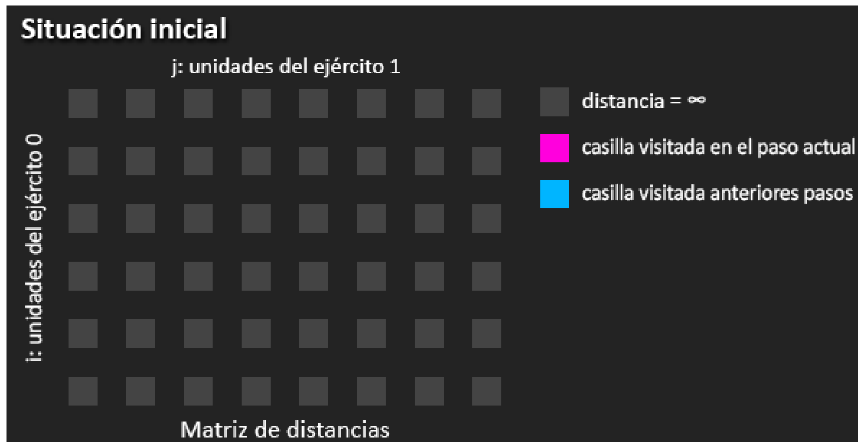


Figura 5.25: Ejemplo de las posiciones de unidades de dos ejércitos.

Así, al iniciarse el algoritmo de medición de distancias, siendo el valor del atributo *maxCalculationsPerUpdate* de 2, la matriz se irá rellenando de la siguiente forma:



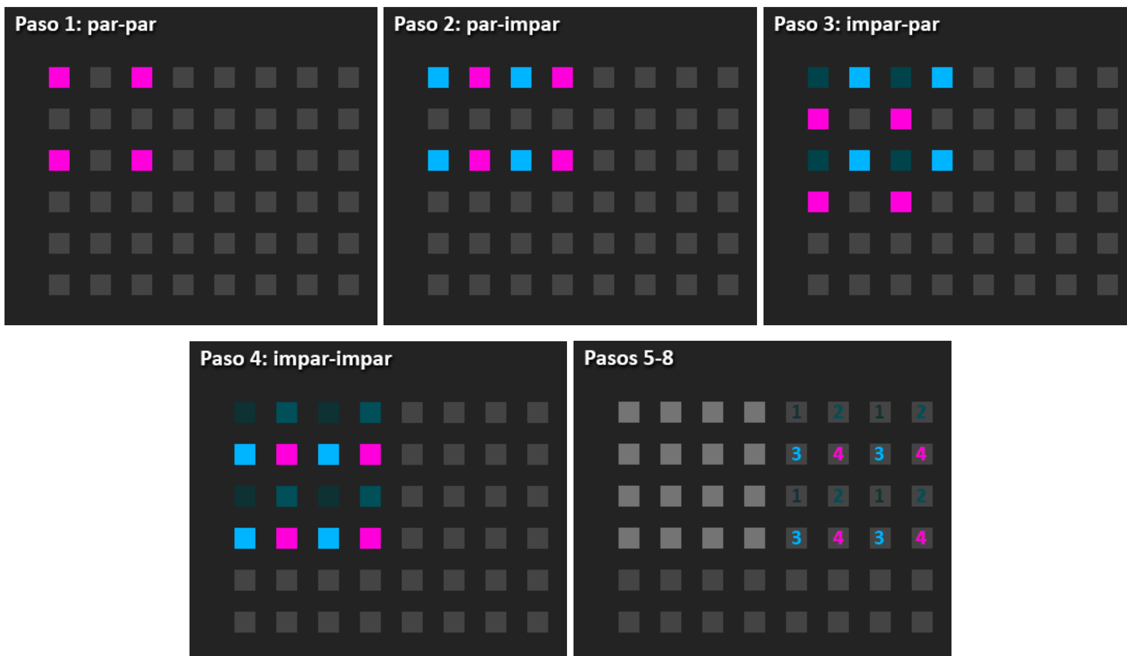


Figura 5.26: Muestra visual del recubrimiento de la matriz de distancias de la clase *DistanceMeasurerTool*.

Después de completarse la matriz de distancias este es su aspecto:

8.456447	—	7.436563	—	7.372967	6.526125	8.4839	—	—	—	—	6.561648	—	8.230502	—	—	—	6.230567	9.177361	9.125889
—	—	11.51881	—	8.230502	8.483563	—	—	—	—	—	10.21938	—	—	—	—	—	9.176083	—	—
—	—	8.483974	—	10.1517	9.126148	—	—	—	—	—	8.230505	—	—	—	—	—	8.456729	—	—
6.56122	8.230502	9.10168	9.176083	4.230692	4.704085	8.689919	—	—	—	11.51884	7.38873	—	7.435003	8.483562	—	10.21939	5.861577	10.18637	6.230566
8.611207	11.43873	4.230694	9.12709	9.000113	7.296005	6.526524	—	—	—	8.230505	4.655583	—	7.374453	10.15254	—	8.456749	5.784579	6.230568	10.0957
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
—	—	8.230505	—	11.4387	10.15252	—	—	—	—	—	8.45675	—	—	—	—	—	9.12706	—	—
8.483562	—	10.18635	—	6.230567	6.561221	10.21939	—	—	—	—	8.689911	—	9.176082	—	—	—	7.435004	11.51882	8.230502
6.230566	8.456449	7.389921	8.483562	4.655007	4.230692	7.43539	—	—	—	10.22027	5.862054	—	6.56122	8.230502	—	9.176404	4.704085	8.690938	6.526124
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
10.15253	—	6.230569	—	10.09569	8.611192	8.456748	—	—	—	—	6.526525	—	9.127074	—	—	—	7.374434	8.230504	11.43871
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
9.126155	—	6.561741	—	8.610217	7.373297	8.230504	—	—	—	—	6.230569	—	8.456735	—	—	—	6.526497	8.483973	10.1517
—	—	10.22024	—	8.456449	8.230502	—	—	—	—	—	9.176388	—	—	—	—	—	8.483563	—	—
—	—	9.177346	—	9.12589	8.456449	—	—	—	—	—	8.483894	—	—	—	—	—	8.230502	—	—
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
8.230502	—	8.690923	—	6.526125	6.230567	9.176394	—	—	—	—	7.435381	—	8.483562	—	—	—	6.561221	10.22025	8.456447
7.373305	10.15171	4.704793	8.456743	7.294845	5.783115	6.230568	—	—	—	8.483974	4.230694	—	6.526506	9.126163	—	8.230505	4.655543	6.56174	8.610225
6.526124	9.12589	5.863554	8.230502	5.782684	4.655007	6.561658	—	—	—	9.177378	4.70468	—	6.230566	8.456448	—	8.48391	4.230692	7.436581	7.372966

Figura 5.27: Muestra de la matriz de distancias de la clase *DistanceMeasurerTool*.

En esta matriz de ejemplo se aprecia cómo hay 4 filas y 4 columnas sin rellenar. Estas pertenecen a las 2 filas de unidades más alejadas del centro. Además, se reparte el rellenado

completo de la matriz a lo largo de 100 ciclos de juego, que transcurren en aproximadamente 1,67 segundos.

Al rellenarse las casillas en las que el resultado es menor al área de visión de las unidades (10 metros), se llama al método *EnemyEntersInVisionSphere* de la unidad correspondiente, pasando además por parámetro la referencia a la otra unidad.

b. MOBA

Al igual que el desarrollo de la parte RTS del juego, el desarrollo de la parte MOBA también ha supuesto uno de los procesos de mayor atención y dedicación, tanto por las funcionalidades de este género como por la obtención de un sistema de juego atractivo y variado.

A diferencia de la parte RTS, el jugador tiene un control directo sobre las acciones de su personaje (héroe), en las cuales, se produce una reacción automática a las órdenes efectuadas por el jugador. Estas reacciones instantáneas son las que otorgan esa versatilidad a este género dotándolo de una gran jugabilidad.

En lo referente al diseño de estos personajes, se ha llevado a cabo con carácter específico y minucioso, ya que existen varios héroes, cada uno dotado con sus propias características y habilidades.

Para finalizar, cabe destacar el tiempo dedicado al sistema de control del personaje, ya que la complejidad en su manejo, por el número de acciones posibles, es mayor de lo que puede parecer, y por lo tanto, se ha escogido un sistema simple de movimiento y acceso a habilidades.

i. Diseño software del héroe

En esta sección se cuenta cómo está diseñado cada uno de los héroes del juego a nivel software, así como todos los componentes que necesita cada héroe para su funcionamiento en la escena. Aquí se detallan aquellos componentes que están relacionados con el control de los héroes, con sus atributos y con su evolución.

1. Diseño general

Todos los héroes poseen los siguientes componentes

- *Transform:* este componente se encarga de la posición, rotación y escalación del héroe.
- *CStateUnit:* este componente se encarga de ejecutar un cambio de animación, o una animación en la que ya esté.
- *ThirdPersonCamera:* este componente se encarga del comportamiento de la cámara, que básicamente sigue al héroe como una cámara en tercera persona.
- *CharacterController*: este componente es necesario para el manejo del héroe en tercera persona y proporciona su colisionador.

- *HeroeController:* este componente es común a todos los héroes y se encarga de almacenar y manejar sus atributos activos (ataque físico, ataque mágico, velocidad de ataque…), y también de actualizar sus estados (caminar, correr, atacar…). Se tiene en cuenta que cada héroe posee un componente de control específico que es una clase que hereda de ésta (OrcController, RobotController).
- *CTeam:* este componente se encarga de relacionar al héroe con alguno de los bandos existentes en el juego (también es usado por cualquiera de las unidades del juego).
- *CBasicAttributesHero:* este componente se encarga de almacenar los valores de maná, adrenalina, nivel actual, defensa mágica y física del héroe, y además de pintar su GUI.
- *NavMeshObstacle:* este componente se encarga de considerar al héroe como un obstáculo a la hora de calcular el recorrido en las unidades RTS.
- *CLife:* este componente se encarga de manejar la vida del héroe (también es usado por cualquiera de las unidades del juego).
- *HeroNetwork:* este componente se encarga de enviar la información de nuestro héroe a través de la red.
- *FogOfWarUnit:* este componente se encarga de manejar la niebla de guerra para el héroe.
- *PhotonView:* este componente es necesario para la conexión a través de Photon.
- *Animation:* este componente contiene las animaciones del héroe.

El diagrama de componentes se muestra en la Figura 5.28.

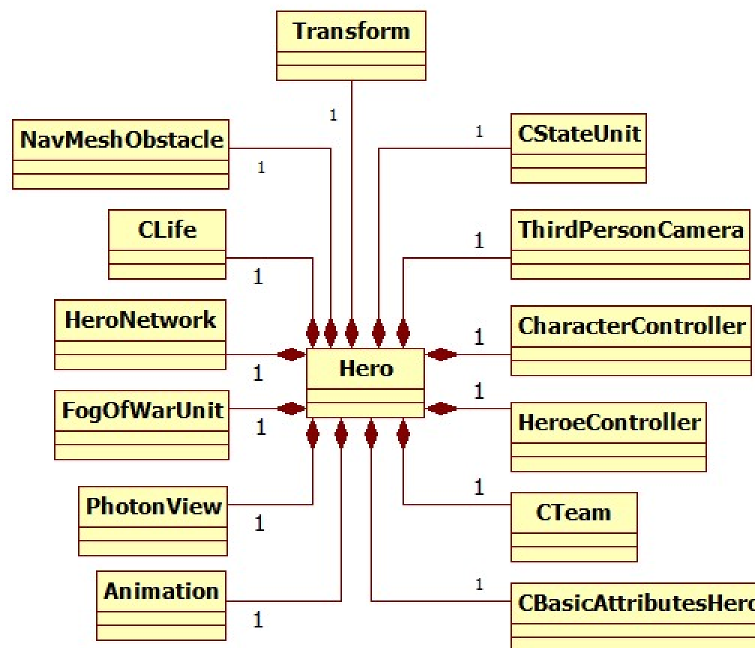
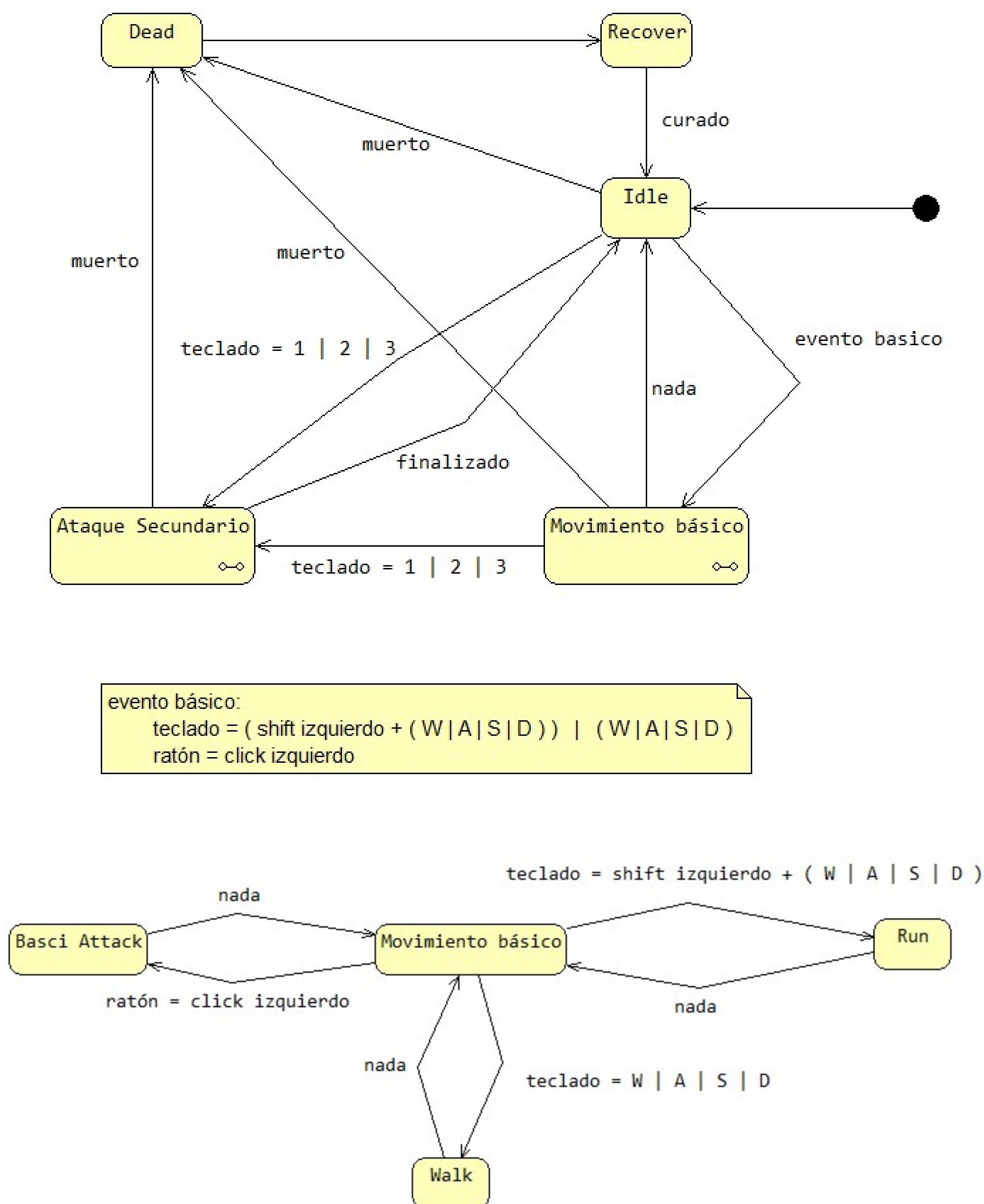


Figura 5.28: Diagrama de componentes del héroe

2. HeroeController

Este componente es una clase que hereda de la clase *ControllableCharacter.cs*. Se encarga de almecenar algunos atributos del héroe (ataque físico y mágico, velocidad de ataque, etc), y de reaccionar frente a los posibles eventos de teclado y ratón.

Una de sus partes importantes es el control que lleva sobre el estado de un héroe para poder animarlo después, haciendo uso del componente *CStateUnit.cs*. Todos los héroes del juego poseen una máquina de estados común en función de la acción que el jugador quiera realizar. Las posibles acciones son la de reposo (*idle*), la de caminar (*walk*), la de correr (*run*), la de realizar un ataque básico (*basic attack*), la de realizar uno de los ataques secundarios (*secondary attack*), la de muerte (*dead*), y la de revivir (*recover*). El funcionamiento de la máquina de estados se muestra en la Figura 5.29 y su implementación se muestra en el Código 5.34.



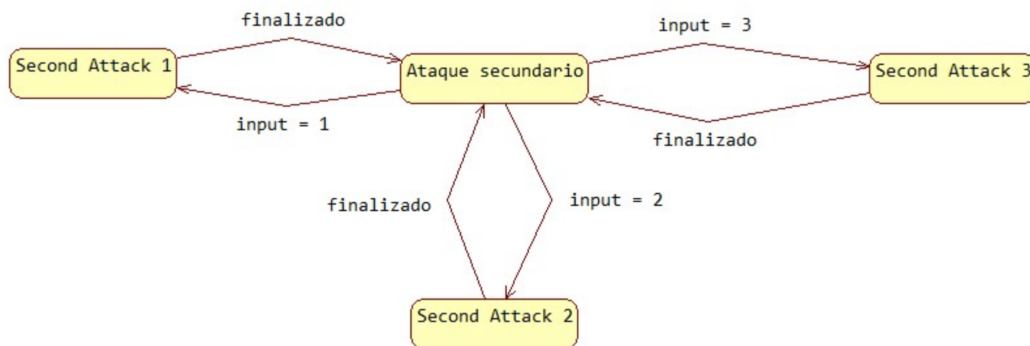


Figura 5.29: Máquina de estados del héroe.

```
// Secondary attack
if (
    (Input.GetKey(KeyCode.Alpha1) || useSkill1) && ability1 && cooldown1 ==
cooldown1total &&
    ((manaSkill1 != -1 && cBasicAttributes.consumeMana(manaSkill1)) ||
(adrenSkill1 != -1 && cBasicAttributes.consumeAdren(adrenSkill1)))
    )
{
    state = StateHeroe.AttackSecond;
    stateAttackSecond = AttackSecond.Attack1;
}
else if (
    (Input.GetKey(KeyCode.Alpha2) || useSkill2) && ability2 && cooldown2 ==
cooldown2total &&
    ((manaSkill2 != -1 && cBasicAttributes.consumeMana(manaSkill2)) ||
(adrenSkill2 != -1 && cBasicAttributes.consumeAdren(adrenSkill2)))
    )
{
    state = StateHeroe.AttackSecond;
    stateAttackSecond = AttackSecond.Attack2;
}
else if (
    (Input.GetKey(KeyCode.Alpha3) || useSkill3) && ability3 && cooldown3 ==
cooldown3total &&
    ((manaSkill3 != -1 && cBasicAttributes.consumeMana(manaSkill3)) ||
(adrenSkill3 != -1 && cBasicAttributes.consumeAdren(adrenSkill3)))
    )
{
    state = StateHeroe.AttackSecond;
    stateAttackSecond = AttackSecond.Attack3;
}
// Basic attack
else if (Input.GetButton("Fire1"))
{
    state = StateHeroe.AttackBasic;
    stateAttackSecond = AttackSecond.None;
}
// Movement
else if (Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.S) ||
Input.GetKey(KeyCode.D) || Input.GetKey(KeyCode.W))
{
    if (Input.GetKey(KeyCode.LeftShift))
    {
        state = StateHeroe.Run;
```

```
    }
    else
    {
        state = StateHeroe.Walk;
    }
    stateAttackSecond = AttackSecond.None;
}
// Heroe dead
else if (this.life.currentLife <= 0 && this.state != StateHeroe.Dead &&
this.state != StateHeroe.Recover)
{
    this.state = StateHeroe.Dead;
}
// Recover heroe
else if (this.state == StateHeroe.Dead)
{
    this.state = StateHeroe.Recover;
}
// Idle
else
{
    state = StateHeroe.Idle;
    stateAttackSecond = AttackSecond.None;
}
```

Código 5.34: Máquina de estados del héroe

Otro de los aspectos reseñables en esta clase es el suavizado que se le aplica al movimiento cuando nuestro héroe cambia de dirección. Para ello lo que se hace es aplicar la siguiente función que proporciona la clase *Vector3*, que es la siguiente:

```
RotateTowards(current: Vector3, target: Vector3, maxRadiansDelta: float,
maxMagnitudeDelta: float): Vector3;
```

Código 5.35: Función *RotateTowars* de Unity.

Donde *current* es la dirección actual a la cual está apuntando el héroe, *target* es la dirección destino a la que quiere apuntar el héroe, y *maxRadiansDelta* es el ángulo que gira el héroe en cada paso hasta alcanzar su dirección destino.

El vector actual se rota hacia la dirección destino con un ángulo *maxRadiansDelta*, aunque no va más allá de su destino. Si las magnitudes de la posición actual y destino son distintas, entonces la magnitud del resultado es linealmente interpolada durante la rotación. Si se utiliza un valor negativo para *maxRadiansDelta* el vector rota lejos del objetivo hasta que esté apuntando exactamente a la direccion opuesta, y entonces para.

El fragmento de código donde se usa esta función se muestra en el Código 5.36.

```
moveDirection = Vector3.RotateTowards(moveDirection, targetDirection,
rotateSpeed * Mathf.Deg2Rad * Time.deltaTime, 1000);
```

Código 5.36: Rotación suave del héroe

Por último se destaca el uso de dos flags, *canRotate* y *canMove*, que permiten girar y moverse respectivamente al héroe. Estos flags se actualizan desde los componentes controladores específicos de cada héroe (*OrcController*, *RobotController*). Por ejemplo, si se está realizando una habilidad que no permite el movimiento o la rotación del héroe se desactiva el flag correspondiente.

El diagrama UML de esta clase se muestra en la Figura 5.30.

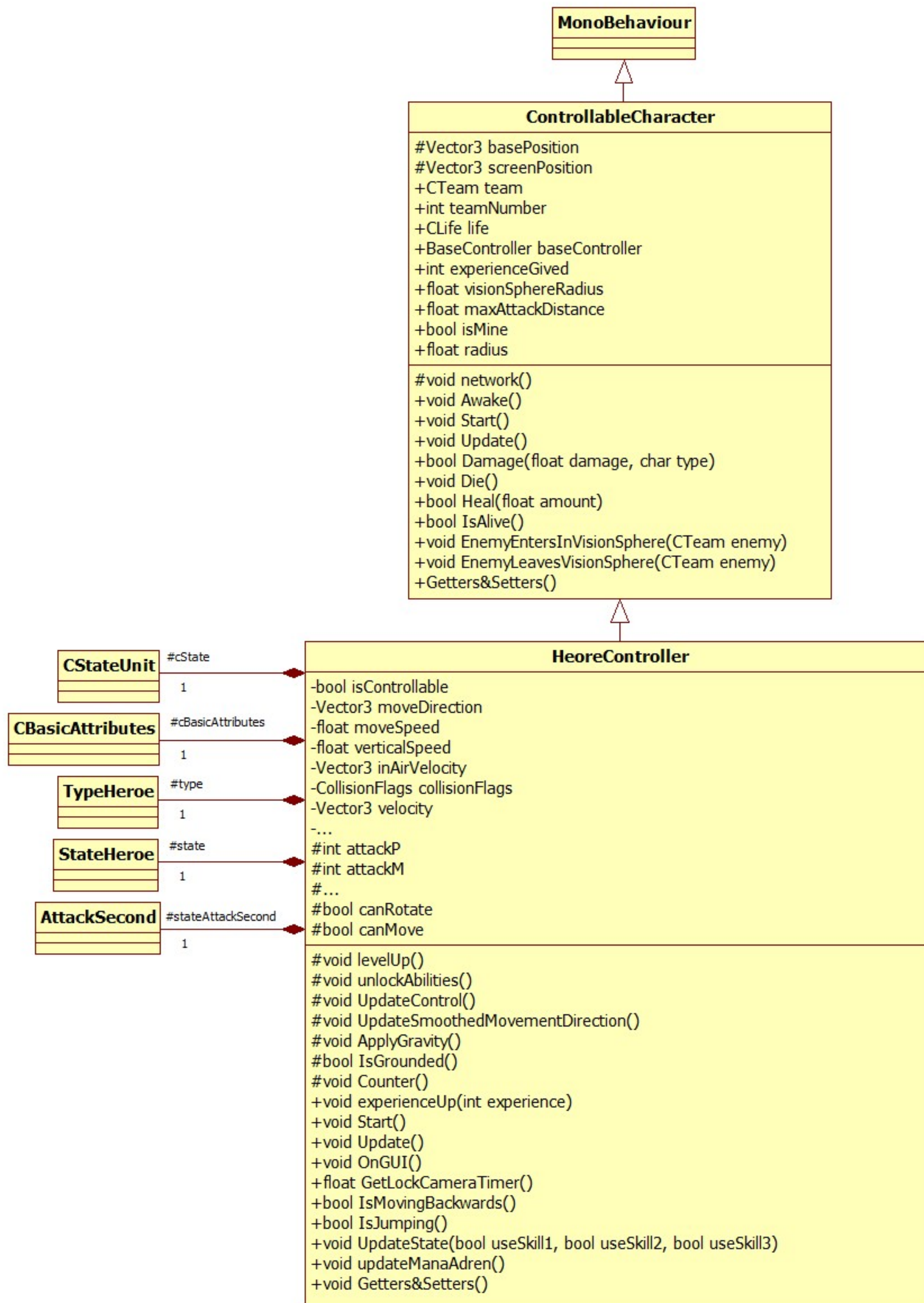


Figura 5.30: UML del script *HeroeController*

3. BasicAttack

Para manejar el daño del ataque básico de ambos héroes se les incluye un colisionador en sus armas de ataque. Cada uno de los colisionadores tiene los siguientes componentes:

- *Transform*: esta componente almacena la posición, rotación y escalación.
- *BoxCollider*: esta componente es el colisionador en sí.

- *RigidBody*: esta componente sirve para que se detecte las colisiones contra los objetos.
- *BasicAttak*: esta componente es un script que se encarga de activar / desactivar el colisionador cuando sea necesario.
- *PhotonView*: esta componente es necesaria para instanciar el daño por red.
- *BasicAttackNetwork*: esta componente es necesario para activar los componentes necesarios, dependiendo de si la instancia es la nuestra o no.

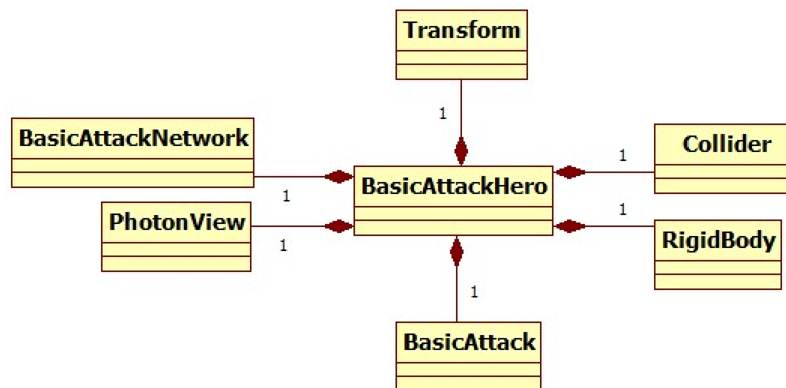


Figura 5.31: Componentes del ataque básico

En cuanto al script *BasicAttack.cs* (ver Código 5.37) se utilizan dos funciones principales:

- *Damage*: se encarga de quitar una cierta cantidad de vida a un objetivo.
- *OnTriggerEnter*: se encarga de detectar la colisión del ataque contra un enemigo para después aplicarle el daño correspondiente mediante la llamada a la función *Damage*.

```
[RPC]
public void Damage(string sEnemy, int damage)
{
    Debug.Log("Ataco con " + damage + " de fuerza");
    GameObject enemy = GameObject.Find(sEnemy);
    enemy.GetComponent<CLife>().Damage(damage, 'P');
}

void OnTriggerEnter (Collider collisionInfo){
    GameObject go = collisionInfo.gameObject;
    if (go.name != this.owner.name)
    {
        CLife goCLife = go.GetComponent<CLife>();
        if (goCLife == null) return;

        photonView.RPC("Damage", PhotonTargets.All, go.name,
owner.GetComponent<HeroeController>().getAttackP());
    }
}
```

Código 5.37: Detección de colisiones y daño en el ataque básico

El diagrama UML de este script se muestra en la Figura 5.32.

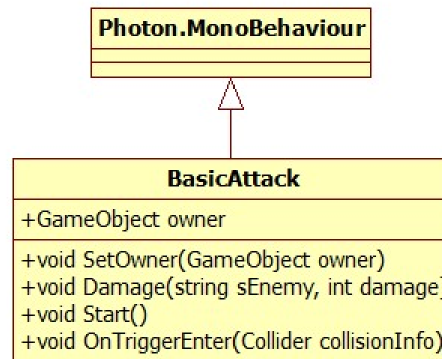


Figura 5.32: UML del script *BasicAttack.cs*

4. Skills: SkillAttack y SkillDefense

Para realizar las habilidades de los héroes se distinguen 2 tipos, habilidades activas y habilidades pasivas. Las habilidades activas son aquellas que realizan daño, y las pasivas son aquellas que modifican los atributos del héroe.

En las habilidades activas, a la hora de realizar el daño, lo primero que se hace es detectar la colisión contra un enemigo, y luego se le aplica el daño correspondiente. Para detectar la colisión contra un enemigo se hace de dos formas distintas:

- *OnTriggerEnter*: esta detección se hace cuando las partículas no tienen activadas la opción de colisión, y entonces se les añade un componente *collider*.
- *OnParticleCollision*: esta detección se hace cuando las partículas tienen activadas la opción de colisión.

Las componentes que se utilizan tanto en las habilidades activas como en las pasivas son:

- *Transform*: este componente lleva la posición, rotación y escalación de las instancias de las habilidades.
- *ParticleSystem*: este componente es el que da forma a las partículas.
- *Collider*: este componente es el colisionador. Solo esta presente en las habilidades activas que no usan las partículas como colisión.
- *RigidBody*: este componente es necesario para detectar la colisión, y va ligado a *Collider*.
- *SkillAttack / SkillDefense*: este componente es el script que maneja la lógica de cada habilidad.
- *PhotonView*: este componente es necesario para instancias las habilidades por red.
- *SkillAttackNetwork / SkillDefenseNetwork*: este componente es el script que se encarga de decidir los componentes que están activos en cada instancia.

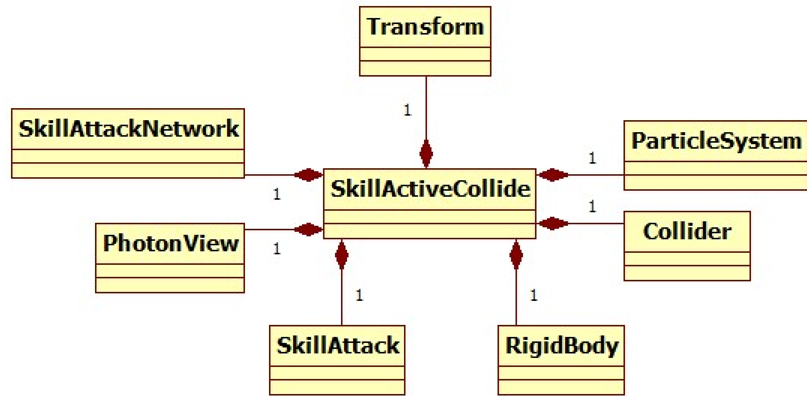


Figura 5.33: Componentes de las habilidades activas con colisionador

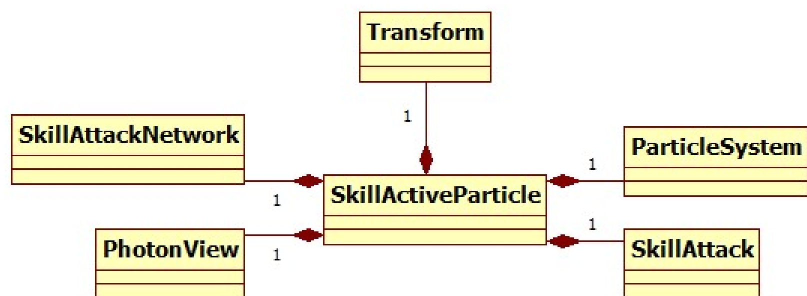


Figura 5.34: Componentes de las habilidades activas con colisionador de partículas

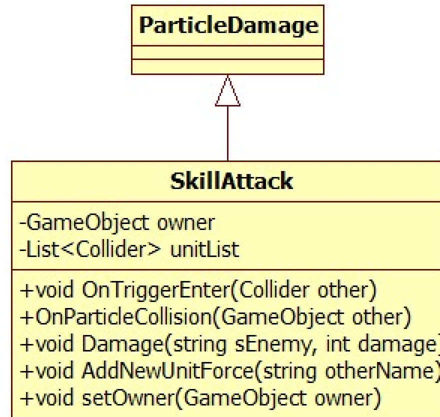


Figura 5.35: UML del script *SkillAttack*

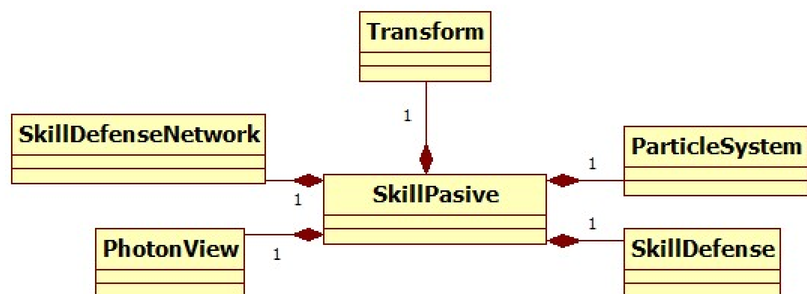


Figura 5.36: Componentes de las habilidades pasivas

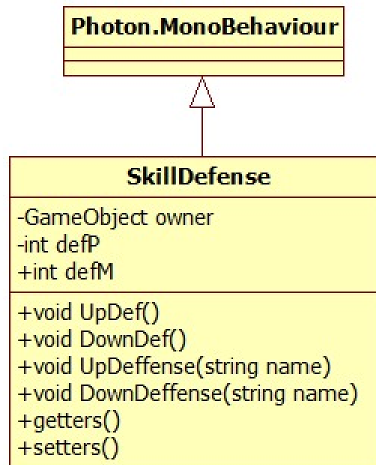


Figura 5.37: UML del script *SkillDefense*

El código del script *SkillAttack* se muestra en el Código 5.38 y el código del script *SkillDefense* se muestra en el Código 5.39.

```
void OnTriggerEnter(Collider other)
{
    Debug.Log(other.tag);
    if (other.gameObject.name != owner.name)
    {
        if (other.tag == "Minion")
        {
            if (!unitList.Contains(other))
            {
                // For damage
                UnitController otherUC = other.GetComponent<UnitController>();

                photonView.RPC("Damage", PhotonTargets.All,
other.gameObject.name, totalDamage);
                photonView.RPC("AddNewUnitForce", PhotonTargets.All,
other.gameObject.name);

                unitList.Add(other);
            }
        }
        else if (other.tag == "Player")
        {
            photonView.RPC("Damage", PhotonTargets.All, other.gameObject.name,
totalDamage);
        }
    }
}

void OnParticleCollision(GameObject other)
{
    if (owner.name != other.name)
    {
        CLife goCLife = other.GetComponent<CLife>();
        if (goCLife == null) return;

        photonView.RPC("Damage", PhotonTargets.All, other.name, totalDamage);
    }
}
```

```
[RPC]
public void Damage(string sEnemy, int damage)
{
    GameObject enemy = GameObject.Find(sEnemy);
    enemy.GetComponent<CLife>().Damage(damage, 'M');
}

[RPC]
public void AddNewUnitForce (string otherName)
{
    GameObject other = GameObject.Find(otherName);
    if (other.GetComponent<PhotonView>().isMine)
    {
        // For damage
        UnitController otherUC = other.GetComponent<UnitController>();
        float enemyDist = Vector3.Distance(transform.position,
other.transform.position);
        otherUC.GetComponent<CLife>().Damage(GetDamage() / enemyDist, 'P');

        // For add a force to the minions so they can fly
        if (!other.rigidbody)
            other.gameObject.AddComponent<Rigidbody>();
        other.rigidbody.isKinematic = false;
        other.rigidbody.useGravity = true;

        if (other.GetComponent<NavMeshAgent>() &&
other.GetComponent<NavMeshAgent>().enabled)
            other.GetComponent<NavMeshAgent>().Stop(true);
        Vector3 dir = other.transform.position - transform.position;
        dir = dir.normalized;

        other.rigidbody.AddForce(new Vector3(dir.x * 2f,
                                            5f,
                                            dir.z * 2f),
                                ForceMode.Impulse);
        otherUC.Fly();
    }
}
```

Código 5.38: Código del script *SkillAttack.cs*

```
public void UpDef()
{
    photonView.RPC("UpDeffense", PhotonTargets.All, owner.name);
}

public void DownDef()
{
    photonView.RPC("DownDeffense", PhotonTargets.All, owner.name);
}

[RPC]
public void UpDeffense(string name)
{
    GameObject robot = GameObject.Find(name);
    CBasicAttributesHero cbah = robot.GetComponent<CBasicAttributesHero>();
    cbah.setDeffenseMagic(cbah.getDeffenseMagic() + 50);
    cbah.setDeffensePhysic(cbah.getDeffensePhysic() + 50);
}

[RPC]
```

```
public void DownDeffense(string name)
{
    GameObject robot = GameObject.Find(name);
    CBasicAttributesHero cbah = robot.GetComponent<CBasicAttributesHero>();
    cbah.setDeffenseMagic(cbah.getDeffenseMagic() - 50);
    cbah.setDeffensePhysic(cbah.getDeffensePhysic() - 50);

}
```

Código 5.39: Código del script *SkillDefense.cs*

5. ParticleDamage

Todas las habilidades activas heredan del script *ParticleDamage.cs*, que es el encargado de almacenar la cantidad de daño que se realiza. El diagrama UML se muestra en la Figura 5.38.

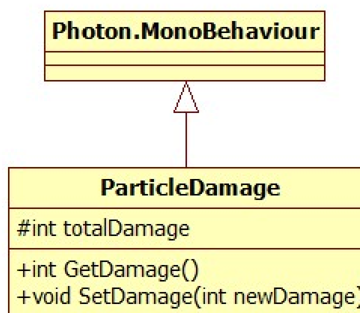


Figura 5.38: UML del script *ParticleDamage.cs*

6. Diseño del orco

Se hace uso del script *OrcController.cs* para manejar sus animaciones. Para ello se comprueba el valor actual de su estado y, en función de éste, se anima haciendo uso de *CStateUnit.cs*. El código de las animaciones se muestra en el Código 5.40.

```
// Secondary attack
if (state == StateHeroe.AttackSecond)
{
    if (stateAttackSecond == AttackSecond.Attack1 && cooldown1 ==
cooldown1total)
    {
        cState.animationName = "Burp";
        cState.animationChanged = true;
        //--------------------------
        StartCoroutine(FirstSkill(animation["Burp"].length * 0.1f));
        //--------------------------
        extraSpeed = false;
    }
    else if (stateAttackSecond == AttackSecond.Attack2 && cooldown2 ==
cooldown2total)
    {
        cState.animationName = "FloorHit";
        cState.animationChanged = true;
        //------------------------------
        StartCoroutine(SecondSkill(animation["FloorHit"].length * 0.25f));
        //--------------------------
        extraSpeed = false;
```

```
    }
    else if (stateAttackSecond == AttackSecond.Attack3 && cooldown3 ==
cooldown3total)
    {
        cState.animationName = "BullStrike";
        cState.animationChanged = true;
        //--------------------------------
        //Smoke
        StartCoroutine(SmokeParticles(0));
        //--------------------------------
        //Shpere
        StartCoroutine(ThirdSkill(animation["BullStrike"].length * 0.2f));
        //-------------------------
        extraSpeed = true;
    }
    //------------------------------------
    canMove = false;
    canRotate = false;
    doingSecondaryAnim= true;
}
// Basic attack
else if (state == StateHeroe.AttackBasic)
{
    if (!animation.IsPlaying("Attack01") && !animation.IsPlaying("Attack02") &&
!animation.IsPlaying("Attack03"))
    {
        cState.animationName = "Attack01";
        cState.animationNameQueued = "Attack02";
        cState.animationNameQueued2 = "Attack03";
        cState.animationChanged = cState.animationChangeQueued =
cState.animationChangeQueued2 = true;
    }
    //-------------------------------------
    canRotate = false;
    canMove = false;
    extraSpeed = false;
}
// Movement
else if (state == StateHeroe.Run)
{
    cState.animationName = "Run";
    cState.animationChanged = true;
    //-------------------------------------
    canRotate = true;
    canMove = true;
    extraSpeed = false;
}
else if (state == StateHeroe.Walk)
{
    cState.animationName = "Walk";
    cState.animationChanged = true;
    //-------------------------------------
    canRotate = true;
    canMove = true;
    extraSpeed = false;
}
else if (state == StateHeroe.Dead)
{
    this.life.currentLife = 0;
    this.transform.position = this.initialPosition;
    isMine = false;
```

```
    //-----------------------------------
    canRotate = false;
    canMove = false;
    extraSpeed = false;
}
else if (this.state == StateHeroe.Recover)
{
    if (this.timeCountLife < 1) this.timeCountLife += Time.deltaTime;
    else
    {
        this.timeCountLife = 0;
        this.life.currentLife += 20;
        if (this.life.currentLife >= this.life.maximunLife)
        {
            this.life.currentLife = this.life.maximunLife;
            this.state = StateHeroe.Idle;
            isMine = true;
        }
    }
    //-----------------------------------
    canRotate = false;
    canMove = false;
    extraSpeed = false;
}
// Idle
else
{
    if (!animation.IsPlaying("Iddle01") && !animation.IsPlaying("Iddle02"))
    {
        cState.animationName = "Iddle01";
        cState.animationNameQueued = "Iddle02";
        cState.animationChanged = cState.animationChangeQueued = true;
    }
    //-----------------------------------
    canRotate = false;
    canMove = false;
    extraSpeed = false;
}
```

Código 5.40: Animaciones del orco

A la hora de instanciar las habilidades se hace mediante la llamada a funciones utilizando corrutinas (ver Código 5.41).

```
private IEnumerator FirstSkill(float time)
{
    yield return new WaitForSeconds(time);
    GameObject snt = (GameObject) PhotonNetwork.Instantiate(snot.name,
transform.localPosition + transform.forward * 2 + Vector3.up,
transform.rotation, 0);
    SkillAttack sa = snt.GetComponent<SkillAttack>();
    sa.SetDamage(1);
    sa.setOwner(gameObject);

    yield return new WaitForSeconds(3f);
    PhotonNetwork.Destroy(snt);
}

private IEnumerator SecondSkill(float time)
{
    yield return new WaitForSeconds(time);
```

```
    GameObject spl = (GameObject) PhotonNetwork.Instantiate(splash.name,
transform.position + new Vector3(0, -1.3f, 0), Quaternion.identity, 0);
    SkillAttack sa = spl.GetComponent<SkillAttack>();
    sa.SetDamage(attackM + 40);
    sa.setOwner(gameObject);

    yield return new WaitForSeconds(1.5f);
    PhotonNetwork.Destroy(spl);
}

private IEnumerator ThirdSkill(float time)
{
    yield return new WaitForSeconds(time);

    GameObject sphereThirdSkillInst = (GameObject)
PhotonNetwork.Instantiate(sphereThirdSkill.name, head.position,
transform.rotation, 0);
    SkillAttack sa = sphereThirdSkillInst.GetComponent<SkillAttack>();
    sa.setOwner(gameObject);
    sa.SetDamage(attackP + 100);
    sphereThirdSkillInst.transform.parent = pelvis;

    yield return new WaitForSeconds(animation["BullStrike"].length * 0.8f);
    PhotonNetwork.Destroy(sphereThirdSkillInst);
}

private IEnumerator SmokeParticles(float time)
{
    yield return new WaitForSeconds(time);

    GameObject smokeInst = (GameObject)PhotonNetwork.Instantiate(smoke.name,
transform.localPosition + Vector3.down*2, transform.rotation, 0);
    smokeInst.transform.parent = pelvis;

    yield return new WaitForSeconds(5f);
    PhotonNetwork.Destroy(smokeInst);
}
```

Código 5.41: Instanciación de las habilidades del orco

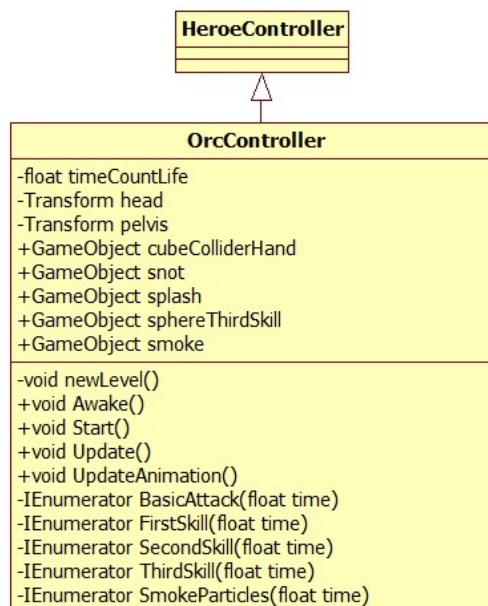


Figura 5.39: UML del script *OrcController*

7. Diseño del robot

Se hace uso del script *RobotController.cs* para manejar sus animaciones. Para ello se comprueba el valor actual de su estado y, en función de éste, se anima haciendo uso de *CStateUnit.cs*, como se hace en el orco. El código de las animaciones se muestra en el Código 5.42.

```
// Secondary attack
if (state == StateHeroe.AttackSecond)
{
    if (stateAttackSecond == AttackSecond.Attack1 && cooldown1 == cooldown1total
&& !doneFirstSkill)
    {
        StartCoroutine(FirstSkill(0));
        //-----------------------------------
        canRotate = true;
        canMove = true;
        doingSecondaryAnim = false;
        doneFirstSkill = true;
    }
    else if (stateAttackSecond == AttackSecond.Attack2 && cooldown2 ==
cooldown2total)
    {
        cState.animationName = "Attack2";
        cState.animationChanged = true;
        //---------------------------
        StartCoroutine(SecondSkill(0));
        //-----------------------------------
        canRotate = false;
        canMove = false;
        doingSecondaryAnim = true;
    }
    else if (stateAttackSecond == AttackSecond.Attack3 && cooldown3 ==
cooldown3total)
    {
        cState.animationName = "Attack3";
        cState.animationChanged = true;
        //---------------------------
        StartCoroutine(ThirdSkill(0));
        //-----------------------------------
        canRotate = false;
        canMove = false;
        doingSecondaryAnim = true;
    }
    //-----------------------------------
    extraSpeed = false;
}
// Basic attack
else if (state == StateHeroe.AttackBasic)
{
    // Do the animation
    if (!animation.IsPlaying("Attack1") && !animation.IsPlaying("Attack2") &&
!animation.IsPlaying("Attack3"))
    {
        cState.animationName = "Attack1";
        cState.animationNameQueued = "Attack2";
        cState.animationNameQueued2 = "Attack3";
        cState.animationChanged = true;
        cState.animationChangeQueued = true;
        cState.animationChangeQueued2 = true;
```

```
    }
    //-------------------------------------
    canRotate = false;
    canMove = false;
    extraSpeed = false;
}
// Movement
else if (state == StateHeroe.Run)
{
    cState.animationName = "Run";
    cState.animationChanged = true;
    //-------------------------------------
    canRotate = true;
    canMove = true;
    extraSpeed = false;
}
else if (state == StateHeroe.Walk)
{
    cState.animationName = "Walk";
    cState.animationChanged = true;
    //-------------------------------------
    canRotate = true;
    canMove = true;
    extraSpeed = false;
}
else if (state == StateHeroe.Dead)
{
    cState.animationName = "Die";
    cState.animationChanged = true;
    //-------------------------------------
    this.life.currentLife = 0;
    this.transform.position = this.initialPosition;
    isMine = false;
    //-------------------------------------
    canRotate = false;
    canMove = false;
    extraSpeed = false;
}
else if (this.state == StateHeroe.Recover)
{
    if (this.timeCountLife < 1) this.timeCountLife += Time.deltaTime;
    else
    {
        this.timeCountLife = 0;
        this.life.currentLife += 20;
        if (this.life.currentLife >= this.life.maximunLife)
        {
            this.life.currentLife = this.life.maximunLife;
            this.state = StateHeroe.Idle;
            isMine = true;
        }
    }
    //-------------------------------------
    canRotate = false;
    canMove = false;
    extraSpeed = false;
}
// Idle
else
{
    if (!animation.IsPlaying("Idle01"))
```

```
    {
        cState.animationName = "Idle01";
        cState.animationChanged = true;
    }
    //------------------------------------
    canRotate = false;
    canMove = false;
    extraSpeed = false;
}
```

Código 5.42: Animaciones del robot

A la hora de instanciar las habilidades se hace mediante la llamada a funciones utilizando corrutinas (ver Código 5.43).

```
private IEnumerator FirstSkill(float time)
{
    yield return new WaitForSeconds(time);

    fireCircleInst = (GameObject)PhotonNetwork.Instantiate(fireball.name,
head.position + Vector3.down * 0.5f, transform.rotation, 0);
    SkillDefense sd = fireCircleInst.GetComponent<SkillDefense>();
    sd.setOwner(gameObject);
    sd.UpDef();
    fireCircleInst.transform.parent = head;

    yield return new WaitForSeconds(1.7f);

    sd.DownDef();
    PhotonNetwork.Destroy(fireCircleInst);

    doneFirstSkill = false;
}

private IEnumerator SecondSkill(float time)
{
    yield return new WaitForSeconds(time);

    turnInst = (GameObject)PhotonNetwork.Instantiate(skelterTurn.name,
transform.position + Vector3.up, transform.rotation, 0);
    SkillAttack sa = turnInst.GetComponent<SkillAttack>();
    sa.SetDamage(75);
    sa.setOwner(gameObject);

    yield return new WaitForSeconds(5f);

    PhotonNetwork.Destroy(turnInst);
}

private IEnumerator ThirdSkill(float time)
{
    yield return new WaitForSeconds(time);

    if (fireShotInst != null)
        PhotonNetwork.Destroy(fireShotInst);
    skelterShot.GetComponent<MeshRenderer>().enabled = false;
    fireShotInst = (GameObject)PhotonNetwork.Instantiate(skelterShot.name,
gun.position, transform.rotation, 0);
    SkillAttack sa = fireShotInst.GetComponent<SkillAttack>();
    sa.setOwner(gameObject);
    sa.SetDamage(75);
```

```
    fireShotInst.transform.parent = gun;

    yield return new WaitForSeconds(2.5f);

    PhotonNetwork.Destroy(fireShotInst);
}

private IEnumerator ThirdBasicAttack(float time)
{
    yield return new WaitForSeconds(time);

    if (fireShotInst != null)
        PhotonNetwork.Destroy(fireShotInst);
    skelterShot.GetComponent<MeshRenderer>().enabled = false;
    fireShotInst = (GameObject)PhotonNetwork.Instantiate(skelterShot.name,
gun.position, transform.rotation, 0);
    SkillAttack sa = turnInst.GetComponent<SkillAttack>();
    sa.SetDamage(75);
    sa.setOwner(gameObject);
    fireShotInst.transform.parent = gun;

    yield return new WaitForSeconds(animation["Attack3"].length);

    PhotonNetwork.Destroy(fireShotInst);

    isShot = false;
}
```

Código 5.43: Instanciación de las habilidades del robot

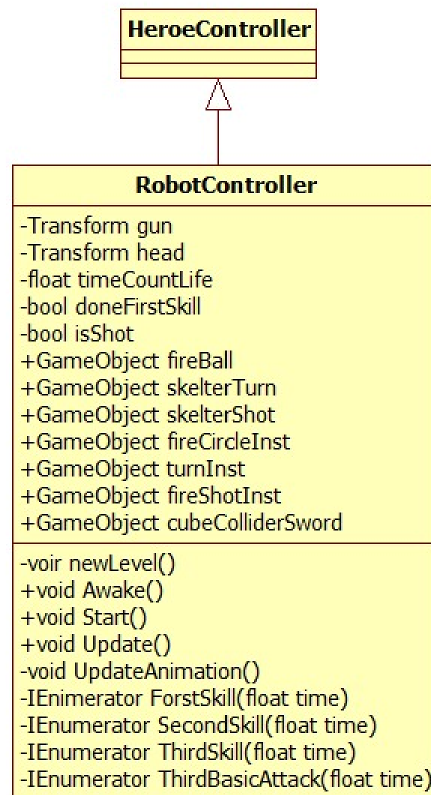


Figura 5.40: UML del script *RobotController*

ii. La cámara en tercera persona

Para el control de la cámara se hace uso del script *ThirdPersonCamera.cs* que se proporciona en la tienda de Unity, aunque se ha modificado ligeramente para adaptarlo al proyecto con la ayuda del libro *Unity 4.x Cookbook* (Smith & Queiroz, 2013), y para mostrar la GUI de la parte MOBA se utiliza el script *CameraMOBAController.cs* que se detalla más adelante.

En el modo de juego MOBA se utiliza una cámara distinta a la del RTS. Esta cámara se coloca detrás del héroe, a una cierta altura, y le persigue todo el rato según avanza por el escenario procurando estar detrás, aunque cuando se hace un giro no se coloca inmediatamente en dicha posición (salvo que se le obligue a ello mediante el botón derecho del ratón), sino que lo hace con un cierto retardo y lentamente. El objetivo de este retardo es no perder el foco anterior del escenario repentinamente.

1. ThirdPersonCamera

El script *ThirdPersonCamera.cs* se puede parametrizar desde Unity. Los atributos que se han modificado en el proyecto son:

- *Camera transform*: es la cámara principal.
- *Distance*: es la distancia de la cámara con respecto al héroe.
- *Height*: es la altura de la cámara con respecto al héroe.
- *Angular smooth lag / Angular max speed*: es la velocidad de la cámara para posicionarse detrás del héroe.
- *Snap smooth lag / Snap max speed*: es la velocidad de la cámara para posicionarse detrás del héroe cuando se presiona el botón derecho del ratón o *alt* del teclado.
- *Lock camera timeout*: es el tiempo que tarda en empezar a girar la cámara una vez se ha girado al héroe.

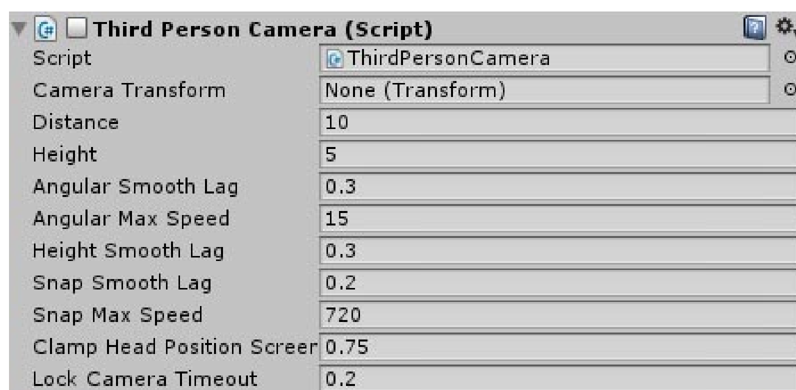


Figura 5.41: Parámetros del script *ThirdPersonCamera.cs.*

En el script se ha hecho una pequeña modificación en donde se sustituye el atributo *controller* de tipo *ThirdPersonController* que había anteriormente por un nuevo atributo *controller* de tipo *HeroeController*.

```
//private ThirdPersonController controller;
private HeroeController controller;
```

Código 5.44: Modificación del tipo de *controller*.

La función *OnEnable* asigna la cámara y asigna el controlador del jugador objetivo para poder hallar las desviaciones entre el héroe objetivo y su colisionador (*centerOffset* y *headOffset*). Estas desviaciones son útiles para posicionar la cámara detrás del héroe.

```
void OnEnable()
{
    if (!cameraTransform && Camera.main)
        cameraTransform = Camera.main.transform;
    if (!cameraTransform)
    {
        Debug.Log("Please assign a camera to the ThirdPersonCamera script.");
        enabled = false;
    }


    _target = transform;
    if (_target)
    {
        //controller = _target.GetComponent<ThirdPersonController>();
        controller = _target.GetComponent<HeroeController>();
    }

    if (controller)
    {
        CharacterController characterController =
(CharacterController)_target.collider;
        centerOffset = characterController.bounds.center - _target.position;
        headOffset = centerOffset;
        headOffset.y = characterController.bounds.max.y - _target.position.y;
    }
    else
        Debug.Log("Please assign a target to the camera that has a
ThirdPersonController script attached.");


    Cut(_target, centerOffset);
}
```

Código 5.45: Código *OnEnable*.

La función *Apply* posiciona la cámara detrás del héroe. Para ello:

- Calcula las posiciones central y superior del personaje o héroe (Código 5.46).
- Calcula la rotación con respecto al eje y del héroe y de la cámara (Código 5.47).
- Se calcula el nuevo ángulo de la cámara para que se posicione detrás del héroe (Código 5.48).
- Se calcula la posición de la cámara en altura (Código 5.49).
- Se halla la rotación haciendo uso del tipo *Quaternion* (Código 5.50).
- Actualiza la posición de la cámara en el plano XOZ teniendo en cuenta la distancia (Código 5.51).
- Actualiza la altura de la cámara (Código 5.52).
- Mira al objetivo (Código 5.53).

```
Vector3 targetCenter = _target.position + centerOffset;
Vector3 targetHead = _target.position + headOffset;
```

Código 5.46: *OnEnable* posiciones central y superior.

```
// Calculate the current & target rotation angles
float originalTargetAngle = _target.eulerAngles.y;
float currentAngle = cameraTransform.eulerAngles.y;

// Adjust real target angle when camera is locked
float targetAngle = originalTargetAngle;
```
Código 5.47: *OnEnable* rotaciones en el eje y del héroe y de la cámara.

```
// When pressing Fire2 (alt) the camera will snap to the target direction real
quick.
// It will stop snapping when it reaches the target
if (Input.GetButton("Fire2"))
    snap = true;

if (snap)
{
    // We are close to the target, so we can stop snapping now!
    if (AngleDistance(currentAngle, originalTargetAngle) < 3.0f)
        snap = false;

    currentAngle = Mathf.SmoothDampAngle(currentAngle, targetAngle, ref
angleVelocity, snapSmoothLag, snapMaxSpeed);
}
// Normal camera motion
else
{
    if (controller.GetLockCameraTimer() < lockCameraTimeout)
    {
        targetAngle = currentAngle;
    }

    // Lock the camera when moving backwards!
    // * It is really confusing to do 180 degree spins when turning around.
    if (AngleDistance(currentAngle, targetAngle) > 160 &&
controller.IsMovingBackwards())
        targetAngle += 180;

    currentAngle = Mathf.SmoothDampAngle(currentAngle, targetAngle, ref
angleVelocity, angularSmoothLag, angularMaxSpeed);
}
```
Código 5.48: *OnEnable* nuevo ángulo de la cámara.

```
// Damp the height
float currentHeight = cameraTransform.position.y;
currentHeight = Mathf.SmoothDamp(currentHeight, targetHeight, ref
heightVelocity, heightSmoothLag);
```
Código 5.49: *OnEnable* altura de la cámara.

```
// Convert the angle into a rotation, by which we then reposition the camera
Quaternion currentRotation = Quaternion.Euler(0, currentAngle, 0);
```
Código 5.50: *OnEnable* halla la rotación de tipo *Quaternion* con recpecto al eje y.

```
// Set the position of the camera on the x-z plane to:
// distance meters behind the target
cameraTransform.position = targetCenter;
cameraTransform.position += currentRotation * Vector3.back * distance;
```
Código 5.51: *OnEnable* posición de la cámara con respeto al plano XOZ.

```
// Set the height of the camera
cameraTransform.position = new Vector3(cameraTransform.position.x,
currentHeight, cameraTransform.position.z);
```
Código 5.52: *OnEnable* actualiza la altura de la cámara.

```
// Always look at the target
SetUpRotation(targetCenter, targetHead);
```

Código 5.53: *OnEnable* mira hacia el héroe.

La función *SetUpRotation* hace que la cámara mire hacia el personaje. Para ello:

- Se consigue la desviación entre la posición del héroe y la posición de la cámara (Código 5.54).
- Se calcula la rotación de la cámara teniendo en cuenta la desviación con respecto al eje y entre la cámara y el objetivo, y teniendo en cuenta la distancia y altura a la que se tiene que situar (Código 5.55).

```
Vector3 cameraPos = cameraTransform.position;
Vector3 offsetToCenter = centerPos - cameraPos;
```

Código 5.54: *SetUpRotation* desviación entre la posición de la cámara y el centro del héroe.

```
// Generate base rotation only around y-axis
        Quaternion yRotation = Quaternion.LookRotation(new
Vector3(offsetToCenter.x, 0, offsetToCenter.z));

        Vector3 relativeOffset = Vector3.forward * distance + Vector3.down *
height;
        cameraTransform.rotation = yRotation *
Quaternion.LookRotation(relativeOffset);
```

Código 5.55: *SetUpRotation* rotación de la cámara.

El diagrama UML de este script se muestra en la Figura 5.42.

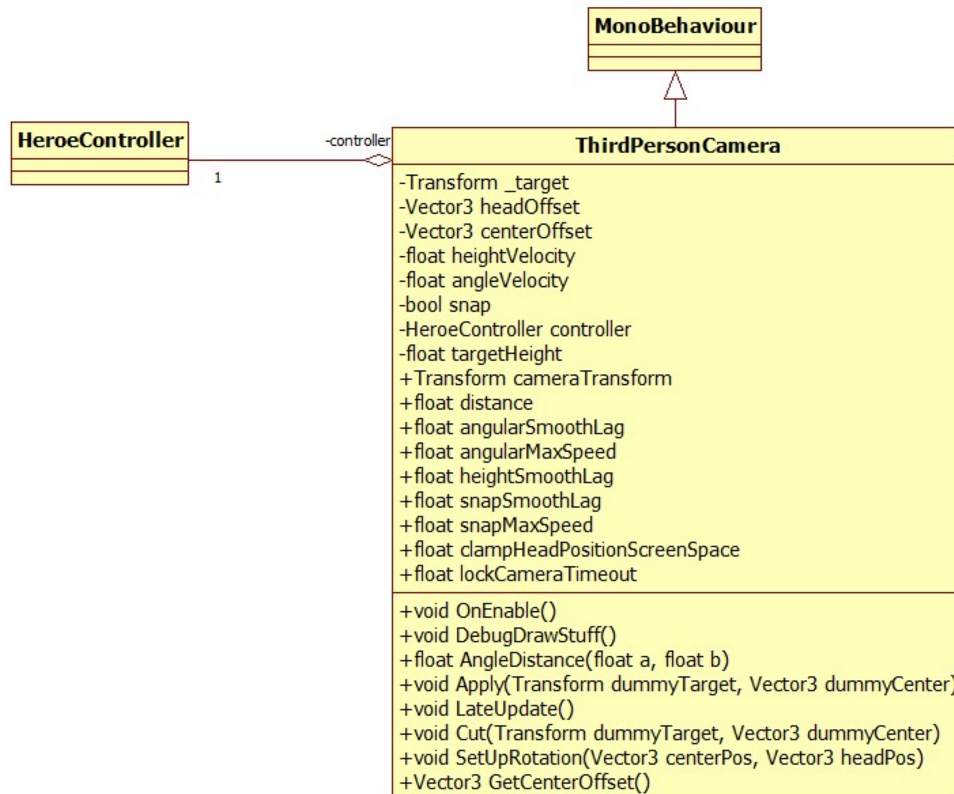


Figura 5.42: UML del script *ThirdPersonCamera*.

2. CameraMOBAController

El script *CameraMOBAController.cs* es el encargado de mostrar la interfaz en pantalla cuando se está jugando con un héroe. En esta interfaz se pueden ver:

- Los valores de los atributos del héroe (poder de daño físico y mágico, defensa física y mágica, etc) en la parte derecha-inferior de la pantalla.
- La vida, la adrenalina y el maná del héroe en la parte en la parte inferior de la pantalla.
- Los estados de cada habilidad especial del héroe (si está bloqueado o no), así como poder presionar encima de cada habilidad para que el héroe las ejecute y ver su *cooldown* (tiempo que debe transcurrir desde que se usa una habilidad hasta poder usarla de nuevo) en la parte inferior de la pantalla.

Además, cuando se va a desbloquear una habilidad, se puede presionar sobre una que esté bloqueada para desbloquearla (Figura 5.45).

Figura 5.43: Aspecto visual del script *CameraMOBAController* (1).

Figura 5.44: Aspecto visual del script *CameraMOBAController* (2).

Figura 5.45: Aspecto visual del script *CameraMOBAController* (3).

El diagrama UML de este script se muestra a continuación.

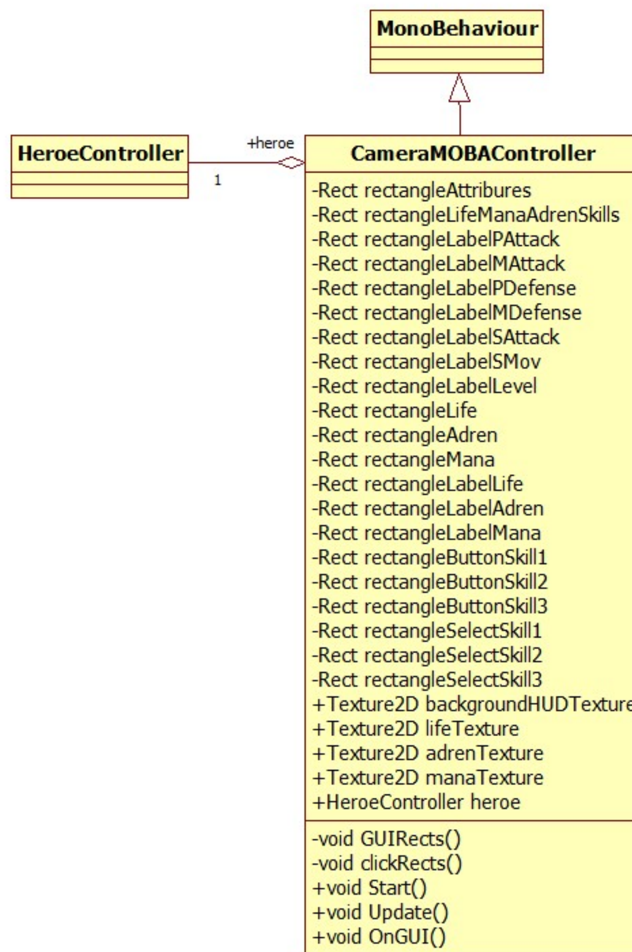


Figura 5.46: UML del script *CameraMOBAController.*

c. Común

i. El sistema multijugador online

En esta sección se detalla cómo funciona el videojuego en red a través de Photon. En la escena del menú principal, hay un script como componente del segundo panel (ver Figura 5.47) que gestiona la sincronización y conexión con *PhotonNetwork*. Para ello lo que hace es:

1. Sincronizar la escena del juego: se actualiza la escena del jugador que se acaba de conectar para poder cargar la escena que le llevará a la partida.
2. Establecer la conexión: se comprueba si existe una conexión establecida con el servidor, y en caso de no existir, se establece.
3. Generar el nombre del cliente: se genera el nombre de la instancia del jugador de la siguiente forma: Guest + random(1, 9999), si el jugador no establece un nombre se deja este por defecto.

```
// this makes sure we can use PhotonNetwork.LoadLevel() on the master client and
all clients in the same room sync their level automatically
PhotonNetwork.automaticallySyncScene = true;

// the following line checks if this client was just created (and not yet
online). if so, we connect
if (PhotonNetwork.connectionStateDetailed == PeerState.PeerCreated)
{
    // Connect to the photon master-server. We use the settings saved in
PhotonServerSettings (a .asset file in this project)
    PhotonNetwork.ConnectUsingSettings("1.0");
}

// generate a name for this player, if none is assigned yet
if (string.IsNullOrEmpty(PhotonNetwork.playerName))
{
    PhotonNetwork.playerName = "Guest" + Random.Range(1, 9999);
}

// if you wanted more debug out, turn this on:
// PhotonNetwork.logLevel = NetworkLogLevel.Full;
```

Código 5.56: Configuración de Photon

A continuación se explica el funcionamiento de dicho script en el menú y los dos modos de juego.

1. Menú

Una vez creada la conexión con el servidor, el jugador se encuentra ante un menú en el que puede elegir crear una nueva sala o entrar en una sala ya existente. Esto se puede hacer mediante los botones que aparecen en la interfaz (*CreateRoom, JoinRoom*) junto con los campos para rellenar con texto (ver Figura 5.47). Se pueden configurar de esta manera tanto el nombre del jugador como el nombre de la sala que se quiere crear. En el centro se ve un panel donde veremos las salas de los jugadores que se han unido a una sala si pulsamos el botón *UpdateRooms* y, o bien están jugando, o bien están esperando a otros jugadores para unirse. Así pues, si se selecciona una sala que no esté completa con todos los jugadores, el jugador

podrá unirse mediante el botón *JoinRoom.* Estos botones desencadenan una serie de eventos (ver Código 5.57).

```
public void CreateRoom()
{
    UILabel roomName = labelRoomName.GetComponent<UILabel>();
    UILabel playerName = labelPlayerName.GetComponent<UILabel>();
    RoomInfo[] roomsInfo = PhotonNetwork.GetRoomList();
    int i = 0; bool enc = false;
    while (i < roomsInfo.Length && !enc)
    {
        if (roomsInfo[i].name.Equals(roomName))
            enc = true;
    }
    if (!enc) // If there's no other roome with the same name
    {
        // If the room has no name we create a random name
        if (roomName.text.Equals(""))
        {
            if (!playerName.Equals(""))
                PhotonNetwork.playerName = playerName.text;
        PhotonNetwork.CreateRoom("Room" + Random.Range(1, 9999), true, true, 4);
            info.text = "Room created.";
        }
        else
        {
            if (!playerName.text.Equals(""))
                PhotonNetwork.playerName = playerName.text;
            PhotonNetwork.CreateRoom(roomName.text, true, true, 4);
            info.text = "Room " + roomName.text + " created.";
        }
    }
    else
    {
        info.text = "already exist a room with that name";
    }
}

public void JoinRoom()
{
    if (selected == -1)
    {
        info.text = "Select one room first";
    }
    else
    {
        RoomInfo[] roomsInfo = PhotonNetwork.GetRoomList();
        // Check if the room is full before connecting
        if (roomsInfo[selected].playerCount == roomsInfo[selected].maxPlayers)
        {
            info.text = "This room is full";
        }
        else
        {
            UILabel playerName = labelPlayerName.GetComponent<UILabel>();
            if (!playerName.text.Equals(""))
                PhotonNetwork.playerName = playerName.text;
            PhotonNetwork.JoinRoom(roomName);
        }
    }
}
```

```
public void UpdateRooms()
{
    RoomInfo[] roomsInfo = PhotonNetwork.GetRoomList();
    Debug.Log(roomsInfo.Length);
    // Update info of the number of rooms
    info.text = roomsInfo.Length + " rooms";
    UILabel roomScript = labelRooms.GetComponent<UILabel>();
    UILabel playerScript = labelPlayers.GetComponent<UILabel>();
    roomScript.text = playerScript.text = "";
    originalButton.SetActive(true);
    originalButton.transform.GetChild(0).gameObject.SetActive(true);
    // Destroy the buttons for the selections
    for (int i = 0; i < layerButons.Count; i++)
    {
        Destroy((Object)layerButons[i]);
        layerButons.RemoveAt(i);
    }
    // Create a button for every room created
    for (int i = 0; i < roomsInfo.Length; i++)
    {
        GameObject button = (GameObject)Instantiate(originalButton);
        button.name = "labelButton" + i;
        button.transform.parent = labelRooms.transform;
        button.transform.position = originalButton.transform.position;
        button.transform.localScale = originalButton.transform.localScale;

        button.transform.localPosition += new Vector3(0, -12 * i, 0);
        layerButons.Add(button);

        roomScript.text = roomScript.text + roomsInfo[i].name + "\n";
        playerScript.text = playerScript.text + roomsInfo[i].playerCount + "/" +
roomsInfo[i].maxPlayers + "\n";
    }

    originalButton.SetActive(false);
    originalButton.transform.GetChild(0).gameObject.SetActive(false);
}
```

Código 5.57: Menú de creación o selección de sala

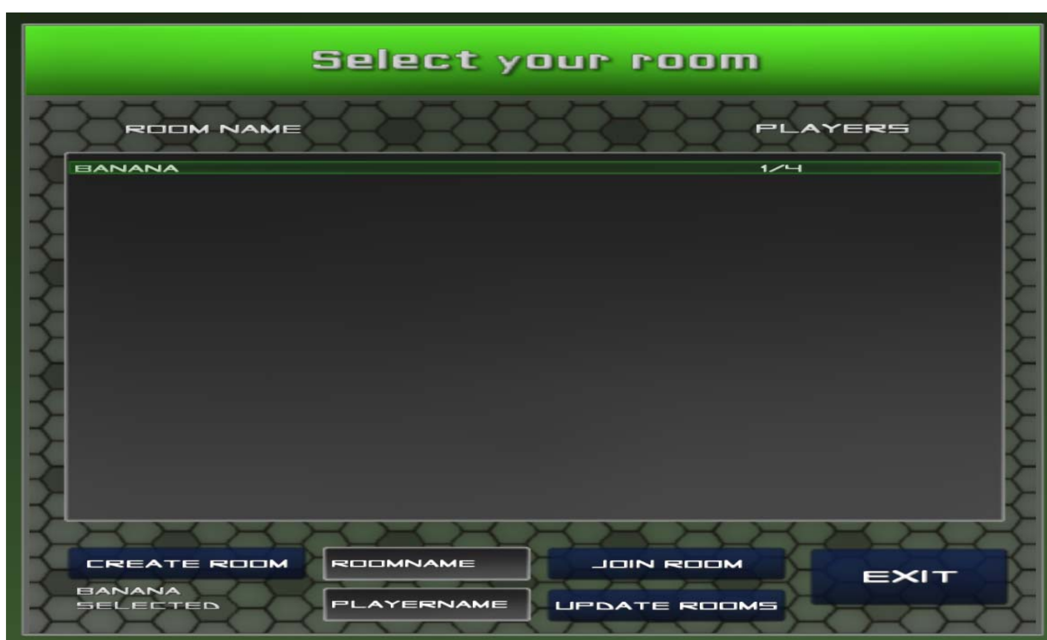


Figura 5.47: Menú de creación o selección de una sala

A continuación, aparece el último menú en el que se da la posibilidad de elegir entre manejar a uno de los dos héroes o al ejército de una de las dos facciones. A la hora de elegir una de las modalidades se activan/desactivan una serie de scripts en la escena en la que se jugará la partida.

```
// Instantiate an army
BlueBase.GetComponent<CSelectable>().enabled = false;
BlueBase.GetComponent<BaseController>().enabled = false;
Destroy(army1);

Camera.main.GetComponent<CameraRTSController>().enabled = true;
Camera.main.GetComponent<CameraMOBAController>().enabled = false;
//-----------------------------------------------
// Instantiate a heroe
GameObject heroInst = PhotonNetwork.Instantiate("Skelterbot",
Camera.main.transform.position + Vector3.down * 24f, new Quaternion(), 0);
heroInst.GetComponent<ThirdPersonCamera>().cameraTransform =
Camera.main.transform;

Camera.main.GetComponent<CameraRTSController>().enabled = false;
Camera.main.GetComponent<CameraMOBAController>().enabled = true;
Camera.main.GetComponent<CameraMOBAController>().heroe =
heroInst.GetComponent<HeroeController>();
```

Código 5.58: Menú de selección de equipo y modo de partida

Figura 5.48: Menú de selección de equipo y modo de partida.

Figura 5.49: Menú de selección de equipo y modo de partida.

2. Unidades

En esta sección se explica como funcionan las unidades (tanto RTS como héroes) a través de Photon. Para que cualquier componente funcione a través de red debe tener lo siguientes componentes:

- *scriptNetwork:* este componente es necesario para poder enviar información desde el objeto local y recibir información desde el remoto.
- *photonView:* este componente es necesario para instanciar el objeto en red a través de Photon.

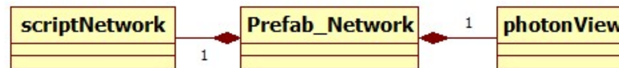


Figura 5.50: Componentes de un objeto en red.

Se destaca que la información que se envía desde un objeto local a través de Photon solo la recibe el mismo objeto en una instancia remota.

Para actualizar las unidades y sus componentes en todas las instancias se realizan las siguientes acciones a través de Photon: instanciar algunos objetos, actualizar su posición y rotación, aplicar daño y fuerza a una unidad y modificar sus atributos.

Instanciar los objetos

Para instanciar objetos a través de Photon y que todos los jugadores puedan verlos desde sus instancias se hace mediante la siguiente función:

```
PhotonNetwork.Instantiate(namePrefab: string, position: Vector3, rotation:
Quaternion, group: int)
```

Código 5.59: Instanciación de objetos en Photon.

Actualizar posición y rotación

Para actualizar la posición y la rotación de una unidad se envía la información desde la local a las remotas a través del script *BasicNetwork.cs*, donde para enviar mensajes se hace mediante la instrucción *SendNext* y para recibirlos se hace mediante la instrucción *ReceiveNext*. Los datos se envían o reciben en la función *OnPhotonSerializeView* (ver Código 5.60).

```
public virtual void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo
info)
{
    if (stream.isWriting)
    {
        //We own this player: send the others our data
        stream.SendNext(transform.position);
        stream.SendNext(transform.rotation);
    }
    else
    {
        //Network player, receive data
        correctPlayerPos = (Vector3)stream.ReceiveNext();
        correctPlayerRot = (Quaternion)stream.ReceiveNext();
```

```
    }
}

private Vector3 correctPlayerPos = new Vector3(0, -10, 0); //We lerp towards
this
private Quaternion correctPlayerRot = Quaternion.identity; //We lerp towards
this

public virtual void Update()
{
    if (!photonView.isMine && correctPlayerPos.y != -10)
    {
        //Update remote player (smooth this, this looks good, at the cost of
some accuracy)
        transform.position = Vector3.Lerp(transform.position, correctPlayerPos,
Time.deltaTime * 5);
        transform.rotation = Quaternion.Lerp(transform.rotation,
correctPlayerRot, Time.deltaTime * 5);
    }
}
```

Código 5.60: Modificación de la posición y rotación a través de Photon.

Aplicar daño y fuerza a una unidad

Para que una unidad reciba daño se hace mediante la función *Kick* o *Damage*, y lo que hacen estas funciones es buscar la unidad a la que se ha golpeado para quitarle vida. Para que una unidad reciba una fuerza se hace mediante la función *AddNewUnitForce*, y el procedimiento es similar al anterior, pero en vez de quitar vida aplica una fuerza (ver Código 5.61).

```
[RPC]
public void Kick (string otherName, float damage)
{
    GameObject other = GameObject.Find(otherName);
    CLife otherCL = other.GetComponent<CLife>();
    otherCL.Damage(damage);
}
[RPC]
public void Damage(string sEnemy, int damage)
{
    GameObject enemy = GameObject.Find(sEnemy);
    enemy.GetComponent<CLife>().Damage(damage, 'M');
}

[RPC]
public void AddNewUnitForce (string otherName)
{
    GameObject other = GameObject.Find(otherName);
    if (other.GetComponent<PhotonView>().isMine)
    {
        // For damage
        UnitController otherUC = other.GetComponent<UnitController>();
        float enemyDist = Vector3.Distance(transform.position,
other.transform.position);
        otherUC.GetComponent<CLife>().Damage(GetDamage() / enemyDist, 'P');

        // For add a force to the minions so they can fly
        if (!other.rigidbody)
            other.gameObject.AddComponent<Rigidbody>();
```

```
        other.rigidbody.isKinematic = false;
        other.rigidbody.useGravity = true;

        if (other.GetComponent<NavMeshAgent>() &&
other.GetComponent<NavMeshAgent>().enabled)
            other.GetComponent<NavMeshAgent>().Stop(true);
        Vector3 dir = other.transform.position - transform.position;
        dir = dir.normalized;

        other.rigidbody.AddForce(new Vector3(dir.x * 2f,
                                             5f,
                                             dir.z * 2f),
                                 ForceMode.Impulse);
        otherUC.Fly();
    }
}
```

Código 5.61: Aplicación de Fuerza y daño a una unidad a través de red.

Las funciones expuestas anteriormente son RPC, y se llaman por medio de la función *photonView.RPC* (ver Código 5.62).

```
photonView.RPC(nameFunc: string, targetPlayer: PhotonPlayer, params[]: Object);
```

Código 5.62: Llamada a una función RPC.

Modificar los atributos de una unidad

En este caso solo se modifican dos atributos: defensa física y defensa mágica. Para ello se usan las funciones *UpDeffense* y *DownDeffense*. Si se quisiera modificar otros atributos se haría de manera similar (ver Código 5.63).

```
[RPC]
public void UpDeffense(string name)
{
    GameObject robot = GameObject.Find(name);
    CBasicAttributesHero cbah = robot.GetComponent<CBasicAttributesHero>();
    cbah.setDeffenseMagic(cbah.getDeffenseMagic() + 50);
    cbah.setDeffensePhysic(cbah.getDeffensePhysic() + 50);
}

[RPC]
public void DownDeffense(string name)
{
    GameObject robot = GameObject.Find(name);
    CBasicAttributesHero cbah = robot.GetComponent<CBasicAttributesHero>();
    cbah.setDeffenseMagic(cbah.getDeffenseMagic() - 50);
    cbah.setDeffensePhysic(cbah.getDeffensePhysic() - 50);

}
```

Código 5.63: Modificación de los atributos de defensa a través de red.

Como en la sección anterior, las funciones expuestas anteriormente son RPC, y se deben llamar de la misma manera.

ii. Niebla de guerra

Para hacer la niebla de guerra se han usado dos scripts, *FogOfWarPlane.cs* y *FogOfWarUnit.cs*:

- *FogOfWarUnit.cs*: este script se utiliza sobre las unidades que tendrán efecto sobre la niebla de guerra, es decir, las del RTS.
- *FogOfWarPlane.cs*: este script se utiliza en el plano que hará la función de niebla de guerra.

a. La clase FogOfWarUnit

El objetivo de este script es hallar las posiciones de la unidad en el plano de la niebla de guerra para poder guardarlas en el atributo *positions* del script *FogOfWarPlane.cs*. Para conseguirlo, cada cierto tiempo se lanza un rayo hacia el plano de la niebla de guerra que devuelve dichas posiciones.

Atributos

- *positions [ ]:* en este atributo se guardan las posiciones halladas al lanzar el rayo hacia el plano de la niebla de guerra.
- *time:* en este atributo se guarda el tiempo que ha pasado desde la última vez que se actualizó *positions*.

El núcleo principal de este script se compone de un bucle que recorre los puntos hallados por el rayo lanzado al plano de la niebla, que se guardan en el atributo *positions*, y también se actualiza el atributo *positions* del script *FogOfWarPlane.cs*. Este fragmento de código se lanza una vez cada cierto tiempo. Cuando no se lanza, se utilizan los puntos guardados anteriormente para actualizar el atributo correspondiente del script *FogOfWarPlane.cs*.

```
// Update is called once per frame
void Update () {
    // Accumulated time since last update of collision points.
    time += Time.deltaTime;
    // Check the collision every second.
    if (time >= 1)
    {
        // Reset time and points for updating them.
        time = 0;
        positions.Clear ();
        // Update the collision with the fog plane.
        RaycastHit[] hits;
        hits = Physics.RaycastAll (transform.position, Vector3.up);
        for (int i = 0; i < hits.Length; i ++)
        {
            RaycastHit hit = hits[i];
            MeshFilter filter = hit.collider.GetComponent<MeshFilter>();
            Vector3 relativePoint;
            if (filter)
            {
                relativePoint =
filter.transform.InverseTransformPoint(hit.point);
                FogOfWarPlane.positions.Add (relativePoint);
                // Save the collision points
                positions.Add(relativePoint);
```

```
            }
        }
    }
    else
    {
        for (int i = 0; i < positions.Count; i ++)
        {
            FogOfWarPlane.positions.Add (positions[i]);
        }
    }
}
```

Código 5.64: Implementación del script *FogOfWarUnit.*

b. La clase FogOfWarPlane

El objetivo de este script es hacer transparente el plano en la posición donde se sitúan las unidades del RTS, teniendo en cuenta el rango de visión de éstas, por lo que un radio alrededor de las unidades también será transparente.

Además, las posiciones por las que pasa una unidad se marcan como *visitadas* con el objetivo de que no vuelvan a oscurecerse del todo cuando la unidad abandone dicha posición.

Atributos

- *positions [ ]:* cuando una unidad está en una posición del escenario, lanza un rayo hacia el plano de la niebla, y ese rayo devuelve una lista de posiciones. En este atributo se guardan esas posiciones.
- *visited [ ]:* en este atributo se guardan las posiciones (vértices) del escenario (es decir, las del plano de la niebla de guerra) que han sido visitadas para no volver a oscurecerlo completamente.
- *isVisiting [ ]:* en este atributo se guardan las posiciones (vértices) del escenario (es decir, las del plano de la niebla de guerra) que se están visitando para esclarecer completamente la zona.
- *inRadius:* en este atributo se guarda el rango de visión de las unidades.

El núcleo principal de este script se compone de un bucle que recorre *positions* anidado a otro bucle que recorre todos los vértices del plano de la niebla, de tal forma que si el vértice se encuentra dentro del rango de visión de la unidad se esclarece la zona, y si no se oscurece.

```
private void FullMesh(float inRadius)
{
    Mesh mesh = GetComponent<MeshFilter> ().mesh;
    Vector3[] vertices =  mesh.vertices;
    float sqrRadius = inRadius * inRadius;
    Color[] colours = mesh.colors;
    //draw the fog for each unit.
    for (int num = 0; num < positions.Count; num ++)
    {
        Vector3 position = positions[num];
        for (int i = 0; i < vertices.Length; i ++)
        {
            float sqrMagnitude = (vertices [i] - position).sqrMagnitude;
            if (sqrMagnitude <= sqrRadius)
            {
                colours [i].a = 0;
                visited [i] = true;
                isVisiting [i] = true;
```

```
                }
                else
                {
                    if (!isVisiting [i] && visited [i])
                        colours [i].a = 0.3f;
                    else if (!isVisiting [i])
                        colours [i].a = 1;
                }
            }
        }
        mesh.colors = colours;
        //set isVisiting false to the next loop.
        for (int i = 0; i < vertices.Length; i++)
        {
            isVisiting [i] = false;
        }
        //clear the list of positions to the next loop.
        positions.Clear ();
}
```

Código 5.65: Implementación del script *ForOfWarPlane.*

c. Ejemplo de resultado

La siguiente figura (Figura 5.51) sirve de muestra del aspecto final de la niebla de guerra en el juego, se aprecia cómo existe un plano negro superior en el escenario y cómo se han aclarado ciertos polígonos de este para perminir la visión de una unidad.

Figura 5.51: Aspecto de la niebla de guerra.

iii. El Minimapa del HUD

En todo juego RTS o MOBA es necesario un minimapa a un lado de la pantalla para poder tomar decisiones estratégicas, como por ejemplo: qué intentar conquistar, a quién atacar, qué defender, etc.

En el juego hay una niebla, la cual cubre el resto de extensión del mapa y se va aclarando según el jugador va descubriendo áreas nuevas (ver apartado 0). Esta niebla tiene tres tipos de densidad: transparente (se ve todo lo que hay en esa zona), semi-transparente (se ve el mapa y los edificios del jugador y los descubiertos del enemigo) y opaca (no se ve nada, esa zona es negra). La niebla está sobre el mapa y se ha representado en el minimapa.

Cada jugador tiene su propio minimapa y en éste se muestra:

- Tanto el héroe y las unidades vivas que posee, como los edificios: torres, almacenes y la base.
- Si el héroe o alguna unidad del jugador comienza a descubrir zonas nuevas del mapa, cuando encuentre héroes o unidades enemigas, éstas se mostrarán en el minimapa; si las tropas que están descubriendo el nuevo área se van de la zona enemiga, dejando de ver las tropas de los otros jugadores, pasará un pequeño lapso de tiempo y después las tropas enemigas desaparecerán del minimapa. (Los siguientes puntos hacen referencia, a modo de ejemplo, a las imágenes que continúan a esta explicación).
    1. Se ven la base del jugador y ocho unidades. La parte de abajo no es visible ya que hay niebla (Figura 5.52).
    2. Cuando una unidad va a la zona de niebla, se despeja y al encontrarse con tropas, éstas se muestran en el minimapa (Figura 5.53).
    3. La unidad vuelve y deja de ver las tropas enemigas, con lo que éstas desaparecen del minimapa (Figura 5.54).

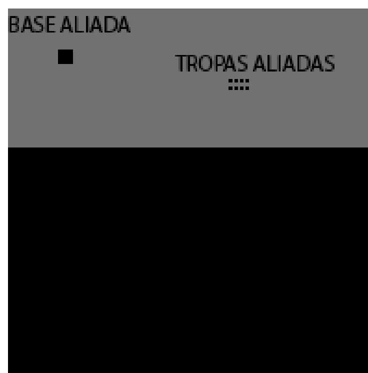


Figura 5.52: Minimapa con tropas enemigas (1).

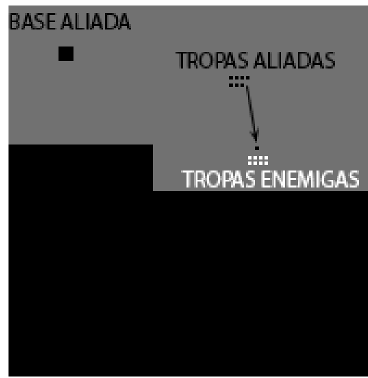


Figura 5.53: Minimapara con tropas enemigas (2).

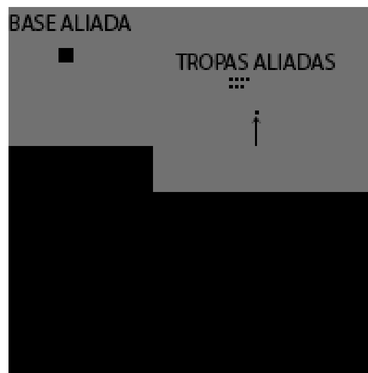


Figura 5.54: Minimapa con tropas enemigas (3).

- En cambio si lo que descubre el héroe o tropas aliadas son edificios enemigos, estas construcciones siempre se verán en el minimapa, ya han sido descubiertos.
    1. Se ven la base del jugador y ocho unidades. La parte de abajo no es visible ya que hay niebla (Figura 5.55).
    2. Cuando una tropa va a la zona de niebla, se despeja y al encontrarse con una base enemiga, ésta se muestra en el minimapa (Figura 5.56).
    3. La unidad vuelve, y la base enemiga, como es un edificio, se sigue mostrando en el minimapa (Figura 5.57).

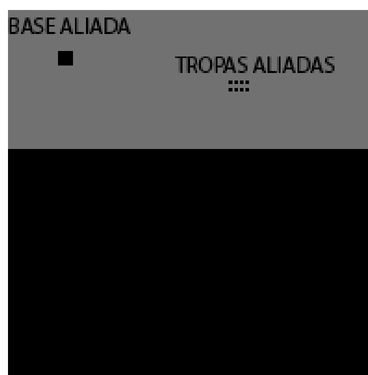


Figura 5.55: Minimapa con edificios enemigos (1).

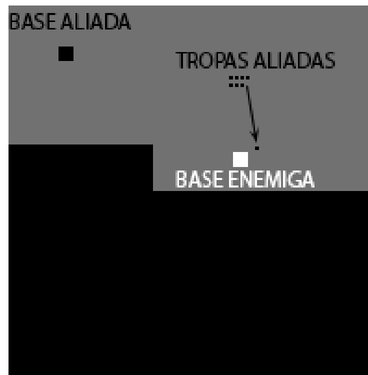


Figura 5.56: Minimapara con edificios enemigos (2).

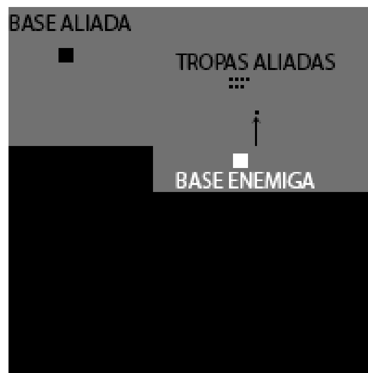


Figura 5.57: Minimapa con edificios enemigos (3).

Funcionamiento

Para poder representar el minimapa son necesarias las posiciones de todas las unidades, bases, torres (de todos los tipos) y almacenes de ambos jugadores.

Se ha creado una matriz de 20x20 llamada *fogTypeMatrix* (el tamaño puede variar para cambiar la precisión) que va a dividir el minimapa en filas y columnas, donde cada elemento puede ser opaco, semi-transparente o transparente; también contiene un contador para cambiar de transparente a semi-transparente cuando no haya una unidad dentro. De esta forma, se mostrará todo lo que haya en las posiciones transparentes y parte de lo que haya en las semi-transparentes (como se comentó en el apartado anterior); en las que son opacas no se mostrará nada, solo niebla. Estas casillas se irán actualizando en función de las unidades aliadas, ya que en la casilla que haya una unidad del jugador se pondrá transparente. Además hay que ir actualizando las posiciones de las unidades, ya que durante el juego van a ir desplazándose por el mapa. Idealmente, la actualización de las posiciones se debería hacer en cada ciclo de juego, pero si se actualizaran todas las posiciones de todas las unidades en cada ciclo de ejecución podría ser muy costoso si el número de éstas fuera muy elevado. Dado que el tiempo de actualización de las posiciones de las unidades del minimapa no es crucial y pueden esperar varias décimas de segundo desde que una unidad se mueve hasta que este cambio se representa en el minimapa, se puede retrasar la actualización:

- Actualización de las unidades aliadas: Como se ha dicho, no se debe actualizar en cada ciclo de ejecución la posición de todas las unidades porque podría ralentizar el juego y no es algo primordial. Por esto, en cada ciclo se actualizará la posición de una unidad por cada cuatro que haya, de manera que después de cuatro ciclos de ejecución se habrán actualizado todas. Dentro se calculará la posición dentro del

minimapa y la casilla en la matriz de la unidad; así se actualiza su posición y se pone la casilla como transparente; además se pone el contador de la casilla a cero.

```
// myArmy
for (int i = contUpdate0; i < max0; i += 3 )
{
    // Update the unit position
    float posx = margin + (myArmy[i].transform.position.x +
        (sizeWorldFloor / 2)) * sizeProportion;
    float posy = posHeight + ((sizeWorldFloor / 2) -
        myArmy[i].transform.position.z) * sizeProportion;
    myUnitList[i] = new Vector2(posx, posy);

    // Update the matrix
    int tileX = (int)((posx - margin) / tileSize);
    int tileY = (int)((posy - posHeight) / tileSize);
    fogTypeMatrix[tileX, tileY].SetFogType(2);
    if (fogTypeMatrix[tileX, tileY].GetCont() != 0)
        fogTypeMatrix[tileX, tileY].ResetCont();
}
contUpdate0 = (contUpdate0 + 1) % 4;
```

Código 5.66: Actualización de las unidades aliadas en el minimapa.

- Actualización de las unidades enemigas: Se actualiza de la misma forma que las aliadas, una de cada cuatro en cada ciclo del juego. En este caso solo se actualiza la posición de la unidad y se cambia su tipo de transparencia al que tenga la casilla en la que se encuentre.

```
// enemyArmy
for (int i = contUpdate1; i < max1; i += 3)
{
   // update the unit position
   float posx = margin + (enemyArmy[i].transform.position.x +
      (sizeWorldFloor / 2)) * sizeProportion;
   float posy = posHeight + ((sizeWorldFloor / 2) -
      enemyArmy[i].transform.position.z) * sizeProportion;
   enemyUnitList[i].SetPosition(new Vector2(posx, posy));

   // update the unit fog type
   int tileX = (int)((posx - margin) / tileSize);
   int tileY = (int)((posy - posHeight) / tileSize);
   enemyUnitList[i].SetFogType(fogTypeMatrix[tileX, tileY].GetFogType());
}
contUpdate1 = (contUpdate1 + 1) % 4;
```

Código 5.67: Actualización de las unidades enemigas en el minimapa.

- Actualización de las casillas de la matriz: En este caso las casillas que se van a actualizar son una de cada tres filas y en cada fila una de cada tres columnas.

```
// The transparent positions of the matrix to semitransparent
for (int i = contY; i < MATRIXSIZE; i += 3)
{
    for (int j = contX; j < MATRIXSIZE; j += 3)
    {
        i = i % MATRIXSIZE;
        j = j % MATRIXSIZE;
        // If the tile is transparent and has to change to
semitransparent
        if ( fogTypeMatrix[i, j].IsMaxCont() &&
             fogTypeMatrix[i, j].GetFogType() == 2 )
```

```
            {
                fogTypeMatrix[i, j].SetFogType(1);
                fogTypeMatrix[i, j].ResetCont();
            }
            // else if the tile is transparent we increase its cont
            else if (fogTypeMatrix[i, j].GetFogType() == 2)
                fogTypeMatrix[i, j].IncreaseCont();
        }
    }
    contX = (contX + 1) % 4;
    if (contX == 0)
        contY = (contY + 1) % 4;
```
Código 5.68: Actualización de las casillas de la matriz en el minimapa.

Teniendo la matriz inicial de la figura X, las casillas que se van actualizando en cada ciclo de juego son:

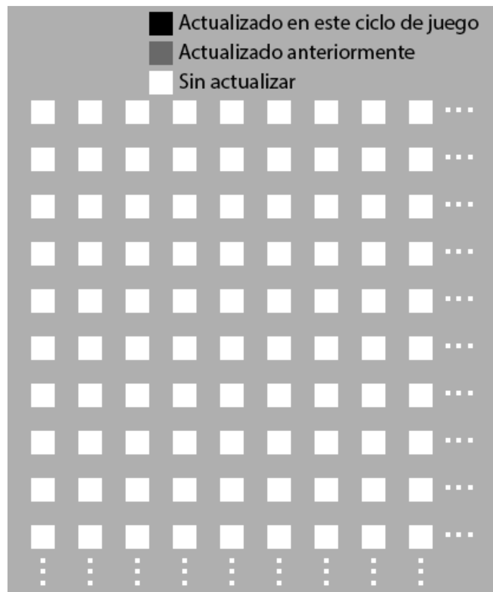


Figura 5.58: Matriz en el minimapa (1).

1. contY= 0 y contX = 0
2. contY = 0 y contX = 1

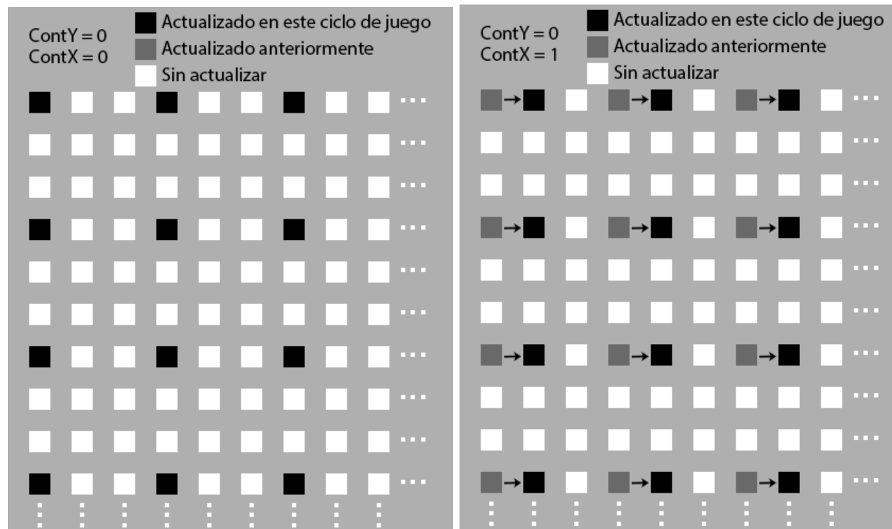


Figura 5.59: Matriz en el minimapa (2).

3. contY = 0 y contX = 2
4. contY = 1 y contX = 0

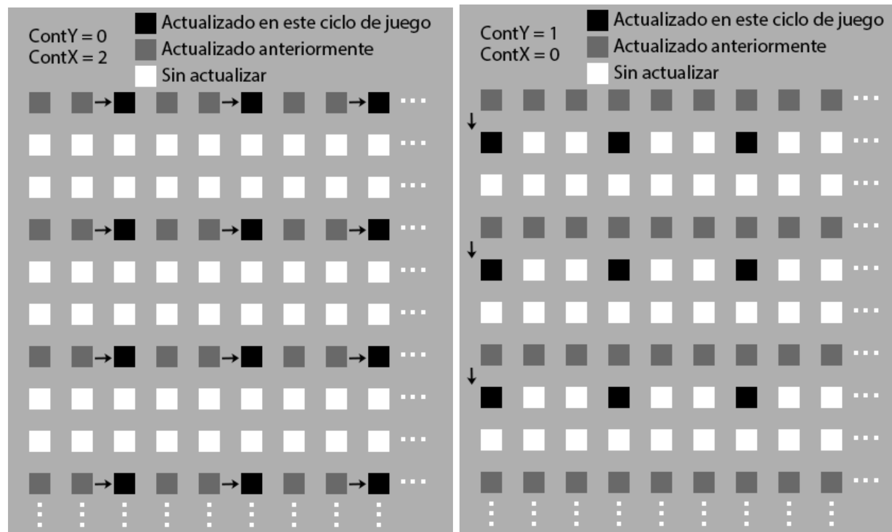


Figura 5.60: Matriz en el minimapa (3).

5. contY = 1 y contX = 1
6. contY = 1 y contX = 2

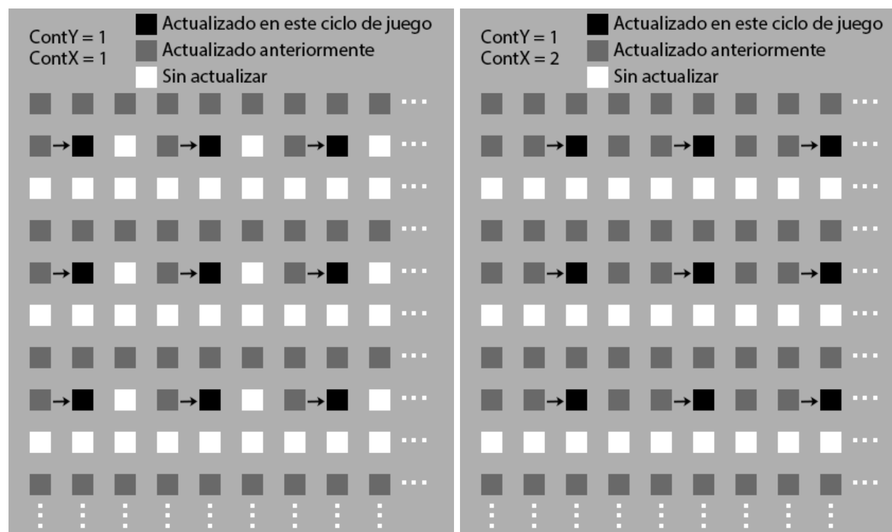


Figura 5.61: Matriz en el minimapa (4).

Faltaría para `contY = 2`, las tres columnas (`contX = 0, contX = 1 y contX = 2`). De esta forma se habrá actualizado toda la matriz.

iv. Shaders implementados

Cuando se comenzó a desarrollar el proyecto en Unity, ninguno de los integrantes del grupo tenía conocimiento sobre programación de *shaders*, por lo que en principio para los materiales (ver apartado sobre materiales en Unity 4.a.v) del proyecto se utilizaron *shaders* que incluye Unity por defecto. Dentro de estos *shaders* hay una gama amplia que incluye desde shaders muy básicos con solo un color difuso hasta *shaders* más complejos con *Normal Mapping* o componentes autolimunidados.

Antes de continuar con la lectura de de este apartado se recomienda la visualización del apartado sobre *Shaders* en Unity de la presente memoria (4.a.v.2).

Llegados a un punto avanzado del proceso de desarrollo del proyecto, se descubrió la necesidad de preparar *shaders* propios para elementos concretos, como por ejemplo, establecer una línea de contorno en las unidades del RTS. Debido al nulo conocimiento en programación de shaders con Cg/HLSL se realizó una primera aproximación a este lenguaje gracias a varios ejemplos extraidos del libro *Unity Shaders and Effect Cookbook* (Lammers, 2013) junto con el uso de un plugin para Unity llamado *ShaderForge*[28] que premite realizar *shaders* utilizando un modelo de cajas muy sencillo y rápido. A continuación, se han estudiado los ficheros de código que se generaba con esta herramienta y se han modificado parámetros y funciones hasta conseguir los efectos deseados.

A continuación se describen ligeramente algunos *shaders* creados, se han elegido los que se consideran más interesantes o atractivos de entre los más de 20 shaders distintos creados.

1. ***Bumped Diffuse Color Texture Specular Color*** (Figura 5.62)

Este *shader* cuenta con un componente difuso (de color) que se establece con una textura de color, un componente especular (de brillo) que se crea en base al componente del canal alfa de la textura de color, varios parámetros (ver Código 5.69) ajustables para conseguir el efecto especular deseado y una textura de *Normal* para el efecto de relieve. Los parámetros más importantes de este *shader* son:

- Texturas:
    - *Diffuse Texture* (o *Color Texture*): textura que contiene los datos de color.
    - *Specular Texture*: textura que contiene los datos de brillo (los puntos claros son los brillantes y los oscuros los no brillantes), este mapa se puede tomar del canal alfa de la textura de color.
    - *Normal Texture*: textura de normales para el relieve.
- Parámetros:
    - *Diffuse Color*: parámetro de tipo color RGB que permite sumar el color que aporta la textura de color, si este es blanco, el color permanecerá tal cual se muestra en esta textura.
    - *Specular Intensity*: parámetro numérico que define la intensidad del canal especular del material.
    - *Specular Color*: parámetro de tipo color RGB que se multiplica con el color aportado por la textura especular, así se consigue que los brillos del material tengan un color en concreto.
    - *Glossines* (brillo): define el tamaño del área que brillará en el objeto, cuanto mayor sea su valor, mayor será el área de brillo del objeto.

[28] http://acegikmo.com/shaderforge/

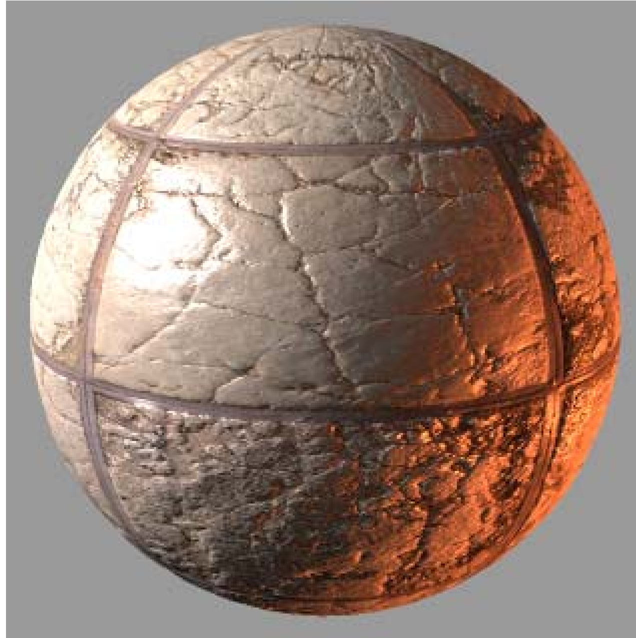


Figura 5.62: Ejemplo del aspecto que produce el *shader BumpedDiffuseColorTextureSpecularColor*.

```
Properties {
    diffuse ("Diffuse", 2D) = "white" {}
    diffuseColor ("Diffuse Color", Color) = (1,1,1,1)
    normal ("Normal", 2D) = "bump" {}
    specularIntensity ("Specular Intensity", Float ) = 1
    gloss ("Gloss", Range(0, 1)) = 0.5
    specularColor ("Specular Color", Color) = (1,1,1,1)
}
```

Código 5.69: Propiedades del *shader BumpedDiffuseColorTextureSpecularColor*.

2. ***Bumped Diffuse Independent Texture Specular Outlined View Position*** (Figura 5.63)

Este *shader* es una modificación del anteriormente descrito (*Bumped Diffuse Color Texture Specular Color*) con la característica principal de mostrar una línea de contorno alrededor del modelo, esta línea hace uso de dos parámetros para ajustar su aspecto, que se modifican en función a la distancia del modelo a la cámara del juego para que cuanto más alejado esté, menor sea el ancho del contorno, sino fuera así, cuando la cámara estuviera muy alejada de las unidades, estas mostrarían un contorno mayor al propio modelo, lo que desvirtuaría mucho su aspecto en el juego. Estos parámetros son (se muestran solo los particulares de este *shader*):

- Parámetros:
    - *Outline Color*: color del contorno del modelo, este parámetro se modifica dentro de los *scripts* del juego y es utilizado para marcar las unidades como seleccionadas (ver apartado Componente *CSelectable* 5.a.i.1).
    - *Outline Width*: permite ajustar el ancho del contorno del modelo aunque no es el único parámetro determinante para esta variable, ya que se multiplica por el valor de la distancia entre la posición del objeto y la posición del punto de vista (Cámara) (ver Código 5.70).

BumpedDiffuseIndepTextureSpecularOutlinedViewPos

Figura 5.63: Ejemplo del aspecto que produce el *shader BumpedDiffuseIndepTextureSpecularOutlinedViewPos*.

```
// uniform float _OutlineWidth;

VertexOutput vert (VertexInput v)
{
    VertexOutput o;
    float4 objPos = mul ( _Object2World, float4(0,0,0,1) );
    float dist = distance(objPos.rgb, _WorldSpaceCameraPos.rgb) * 0.5;
    o.pos = mul
    (
        UNITY_MATRIX_MVP,
        float4
        (
            v.vertex.xyz + v.normal * ( dist * outlineWidth ),
            1
        )
    );
    return o;
}
```

Código 5.70: Cálculo del ancho de la línea de contorno en el *shader BumpedDiffuseIndepTextureSpecularOutlinedViewPos*.

3. ***Bumped Independent Texture Specular Refaction Outlined View Position*** (Figura 5.64)

Este *shader* es una variación del descrito anteriormente (*Bumped Diffuse Independent Texture Specular Outlined View Position*) con la peculiaridad que se incluyen componentes de Alfa y refracción que combinados permiten conseguir una transparencia en el objeto y simular así un efecto de cristal. Los parámetros necesarios para conseguir este efecto son los siguientes (se omiten los elementos explicados en los anteriores *shaders*):

- *Alpha*: parámetro númerico que indica la transparencia del material, si el valor es 0, el material será totalmente transparente, si en cambio es 1, el material será totalmente opaco.
- *Refraction*: deformación que sufre la luz al pasar por el material, gracias a esto se pueden simular rugosidades en el material (como si el cristal no fuese totalmente

liso). Este parámetro recibe el resultado de la multiplicación de los dos siguientes parámetros.

- *Refraction Map*: textura que se crea como resultado de tomar los canales de color Rojo y Verde del *Normal Map*. Sirve para remarcar las rugosidades que pueda tener el material y variar así el aspecto de cristal totalmente pulido a cristal rugoso.
- *Refraction intensity*: número que multiplica al *refraction map* para potenciar el valor de esta textura en el resultado final.

Figura 5.64: Ejemplo del aspecto que produce el *shader BumpedIndepTextureSpecularRefactionOutlinedViewPos*.

4. *Alpha Color Noise Beat Color Diffuse* (Figura 5.65)

Este *shader* fue realizado para añadir un aspect previo a la construcción de los edificios cuando el jugador está seleccionando el lugar donde se construirá este (ver Figura 5.66). El material posee una transparencia sumada a un color base (de tipo *uniform*, por lo que es modificable desde código) que parpadea en base a una función senoidal. Este color, cambia de rojo a verde en función de si el sitio marcado por el jugador es válido para la construcción o no. Además, este *shader* tiene otra textura aplicada a un componente de recorte alfa, para conseguir un efecto de huecos con formas aleatorias en el material. Los parámetros necesarios para conseguir estos efectos son los siguientes:

- *Alpha Color*: color que se suma a la textura de color para alterar el color final del material, este parámetro es de tipo *uniform* y cambia en el código del juego entre rojo y verde.
- *Alpha*: parámetro númerico que indica la transparencia del material (comentado en el *shader* anterior). En este *shader* el parámetro *alpha* varía en función a una señal senoidal en tiempo agregando así un aspecto de "latido" en el modelo.
- *AlphaClip*: este parámetro determina las áreas del material que se recortarán y no aparecerán en el juego, recibe una textura de ruido (ver Figura 5.67) que determina estas áreas en base al color negro o blanco. Además, las coordenadas de esta textura se desplazan a lo largo del tiempo, haciendo que estos huecos que se producen en el modelo se desplacen a una velocidad constante.

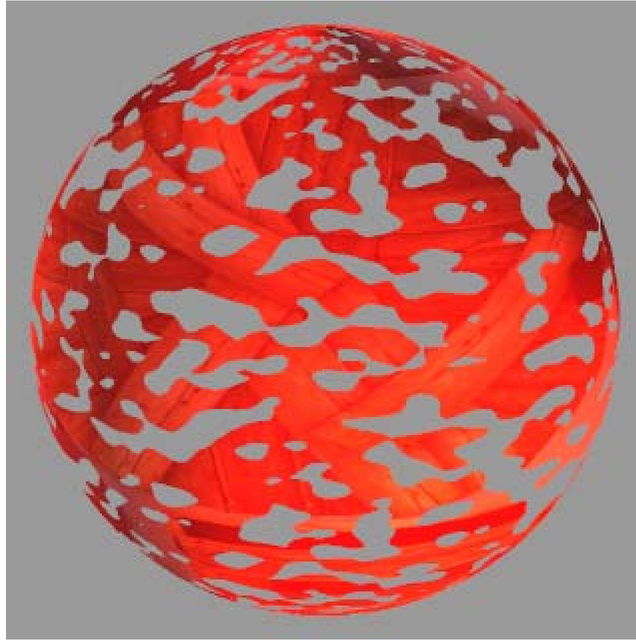


Figura 5.65: Ejemplo del aspecto que produce *el shader Alpha Color Noise Beat Color Diffuse*.

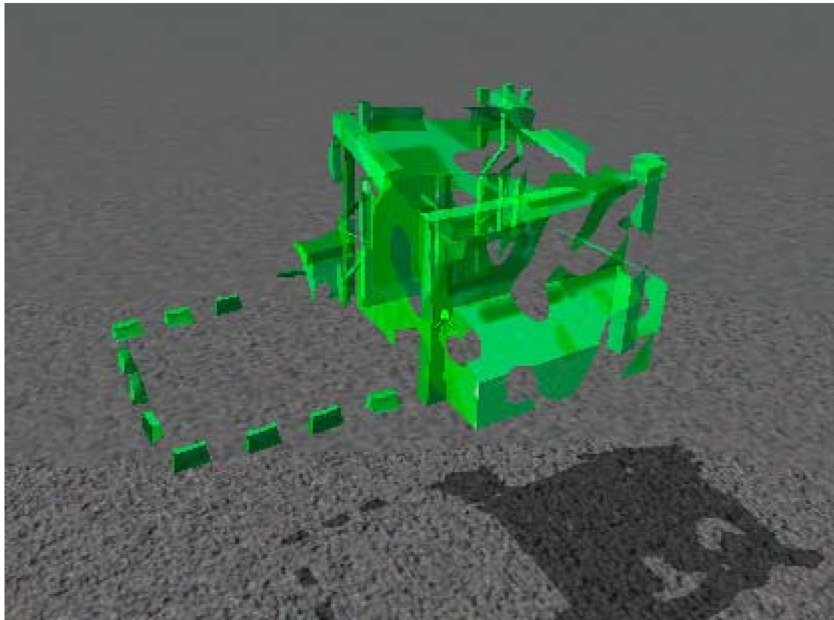


Figura 5.66: Ejemplo del aspecto del *shader Alpha Color Noise Beat Color Diffuse* en un edificio del juego.

Figura 5.67: Textura de ruido *PerlinNoise* para el *Alpha Clip* del *shader Alpha Color Noise Beat Color Diffuse*.

5. UnitDie Timer (Figura 5.68)

Este *shader* fue creado para simular la desaparición gradual de las unidades y consiste en una combinación de los parámetros utilizados en los anteriores *shaders*, con la peculiaridad de que el *Alpha Clip* aumenta con el paso del tiempo, esto hace que los huecos transparentes del material al princpio sean inexistentes y que con el paso del tiempo aumenten hasta que el modelo es totalmente transparente. Este efecto se consigue modificando una variable uniforme que multiplica el valor de la textura de ruido cuando la unidad cae muerta, y este efecto, sumado a la animación que hace el modelo de elevarse al cielo, ofrece un vistoso efecto de la unidad ascendiendo a los cielos mientras se desvanece en el aire.

En este *shader* intervienen principalmente el parámetro de *Alpha Clip*, que recibe una textura de ruido especial (Figura 5.69) y se multiplica por un valor que se incremente con el paso del tiempo (ver Código 5.71), este efecto hace que la textura resultante de la multiplicación, pase de ser blanca completamente (por lo tanto no se recorta nada en el material) a tomar valores grises en zonas uniformemente hasta que es totalmente negra (y por lo tanto se recorta toda la superficie y el material aparece transparente).

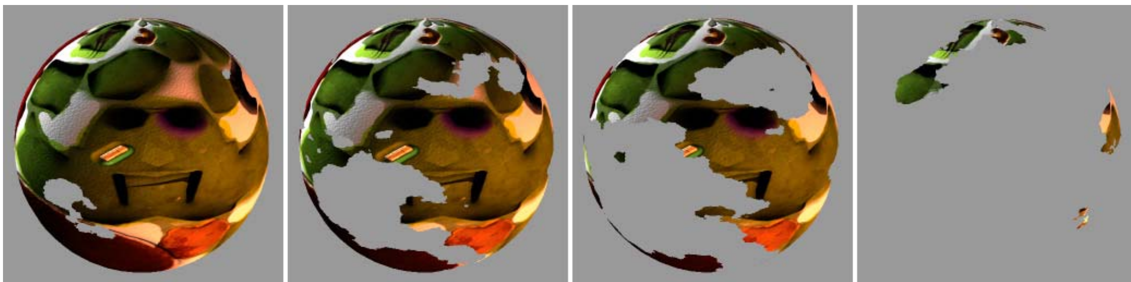


Figura 5.68: Secuencia ejemplo del aspecto que produce el *shader UnitDie Timer*.

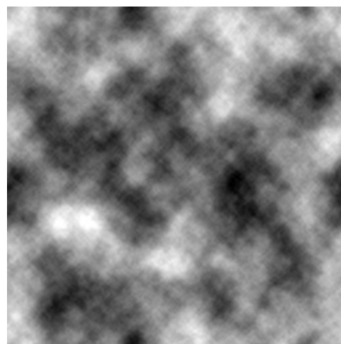


Figura 5.69: Textura de ruido *PerlinNoise* para el *Alpha Clip* del *shader UnitDie Timer*.

```
fixed4 frag (VertexOutput i) : COLOR
{
    float2 aux1 = i.uv0;
    float4 auxTime = _Time + _TimeEditor; // time variable
    text2D text2dAux = tex2D
    (
        alphaNoiseTexture, // noise texture
        TRANSFORM_TEX(aux1.rg, alphaNoiseTexture)).r *
            (1.0 / fmod((auxTime.g * alphaVelocity), 5.0)) // time * alpha
    );
    clip (text2dAux - 0.5);
    SHADOW_COLLECTOR_FRAGMENT(i)
}
```

Código 5.71: fragmento del shader *UnitDie Time* donde se realiza la transformación del *Alpha Clip* en base al tiempo.

v. Sistemas de partículas en NewDetroit

Como se mencionó en el apartado de Sistemas de Partículas en Unity (ver apartado 4.a.vi) existen dos formas de crear sistemas de partículas: una primera forma, añadiendo los componentes por separado, cada uno con su configuración; una segunda forma, creando un componente único de sistemas de partículas. Esta segunda forma ha sido la utilizada en el proyecto.

A continuación, se exponen las partículas creadas para el proyecto:

- Humo de la explosión del cortacésped de las unidades exploradoras del ejército orco: escenifica el humo de la explosión, emitiendo un número establecido de partículas de tamaño variable en un radio esférico de emisión. Posee una textura de humo y la velocidad de las partículas se reduce en función del tiempo.
- Piezas de la explosión del cortacésped de las unidades exploradoras del ejército orco: escenifica las piezas destruidas del cortacésped emitiendo un número fijo de partículas de forma cúbica, emitiéndose aleatoriamente en un radio cónico. La fuerza, velocidad y rotación de las partículas se reduce en función del tiempo. Las partículas colisionan y rebotan con otros elementos del escenario.
- Llamas del cortacésped de las unidades exploradoras del ejército orco: escenifican las llamas provenientes del cortacésped el momento antes de la explosión. Estas llamas aparecen cuando la unidad está por debajo del 25% de vida. Las partículas emitidas tienen un tiempo de vida, tamaño, rotación y velocidad variables en el comienzo de su vida. Se emiten de forma continuada mientras la unidad no pierda toda su vida y su forma de emisión está limitada a un cono. Su velocidad, tamaño, rotación y color varían en función del tiempo de vida.
- Explosión encontrada en el *asset store* de Unity, de uso libre.

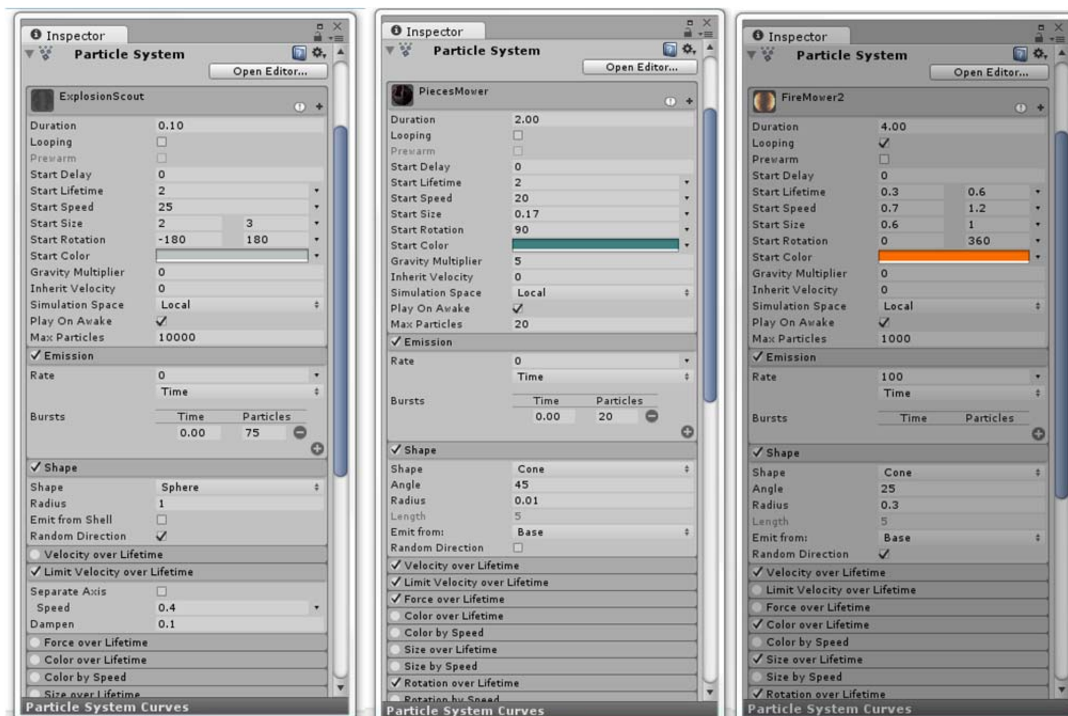


Figura 5.70: Aspecto de los 3 sistemas de partículas expuestos en el ejemplo anterior en el inspector de Unity.

El conjunto de las partículas mencionadas hasta ahora crean la animación de explosión ocasionado por la muerte de las unidades exploradoras del ejército goblin (Figura 5.71).

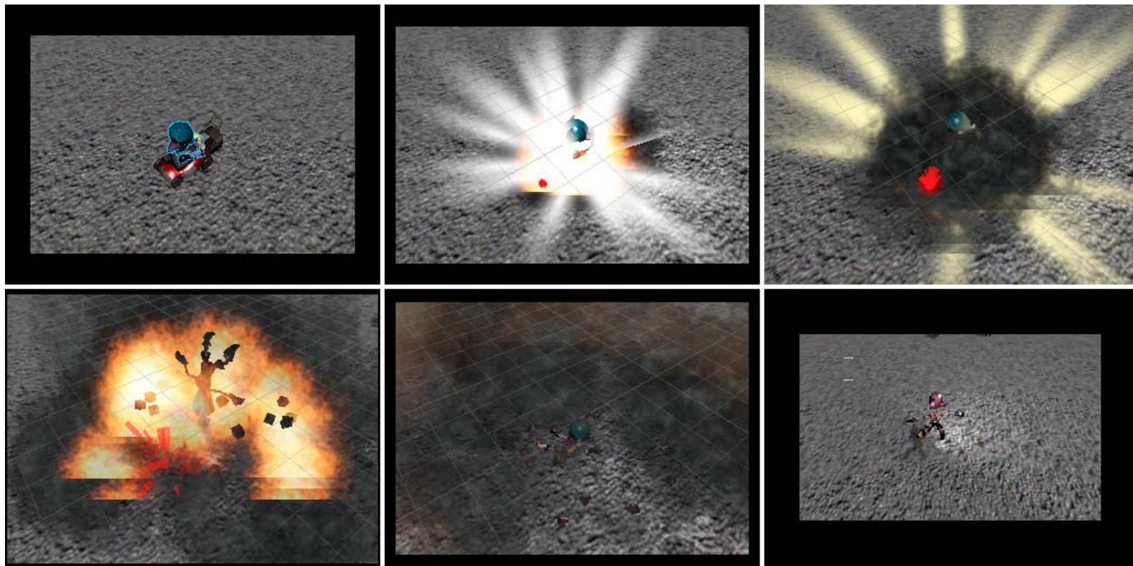


Figura 5.71: Aspecto del sistema de partículas de la explosión del explorador goblin.

- Explosión de los cohetes de los goblin de artillería pesada y explosión de las bolas de fuego de los goblin ingenieros: representa una bomba circular con una onda expansiva que va aumentando su radio. En el caso de artillería pesada la onda expansiva es mayor y en ambos finaliza a los 0.60 segundos.
- Tiene además un *sphere collider* con radio 0.1 y que en cada ciclo del juego va aumentando a la misma velocidad que aumenta el radio de la esfera de las partículas. Este aumento del radio del *collider* se realiza en un *script* que se ha incorporado a las partículas. Además, este *script* va analizando si una unidad enemiga ha entrado en la esfera y si ya ha sido dañada por este. En caso de que una unidad enemiga esté dentro del *collider* y no haya sido dañada por este hay que, primero, hacerla daño (con un valor inversamente proporcional a la distancia de la unidad con respecto al centro de la explosión) y, segundo, aplicar una fuerza a la unidad con un vector *unidad - centroExplosion* como se explica en el apartado de físicas en Unity (4.a.iv).
- Lanzamiento de partículas del héroe orco (Figura 5.74): escenifica el disparo de residuos en línea recta tras activar la primera habilidad del orco. Las partículas emitidas tienen un número fijo y se emiten de forma aleatoria en un rango en forma de cono. En el nacimiento tienen cierto retardo de emisión, tamaño y rotación fija y son afectadas por la gravedad. El tamaño y la rotación varía en función del tiempo de vida de las partículas. Colisionan con cualquier objeto o medio y si se da el caso de que sean unidades quita su porcentaje de vida, usando el script *ParticleDamage*.

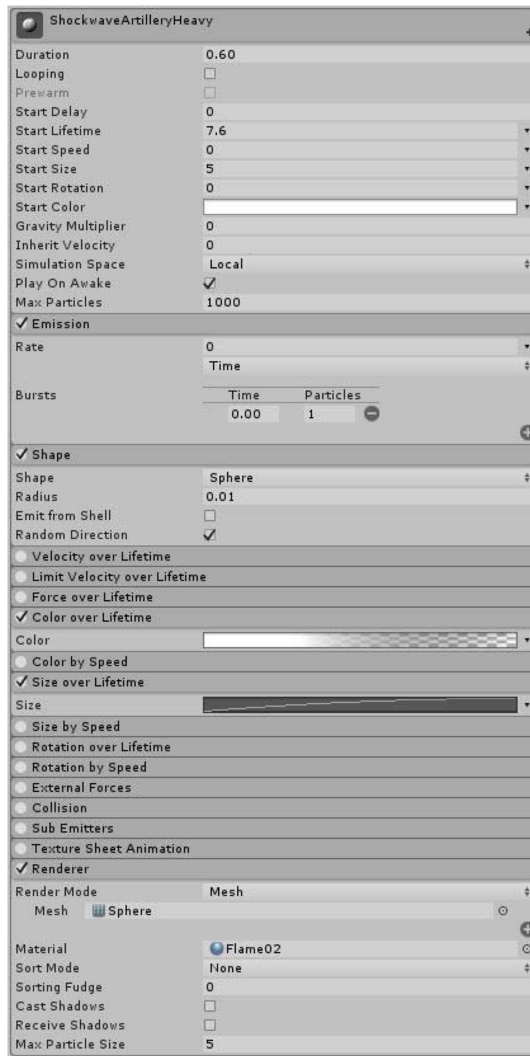


Figura 5.72: Aspecto de la explosión en el inspector de Unity.

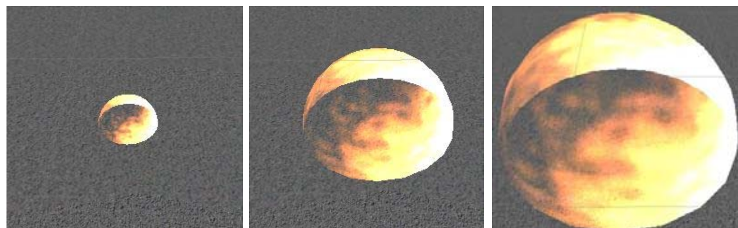


Figura 5.73: Secuencia de imágenes que ilustra el aspecto de esta explosión en el juego.

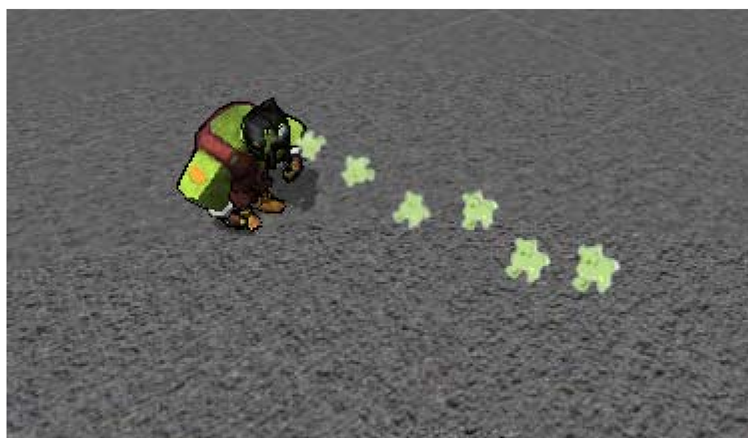


Figura 5.74: Sistema de partículas de la habilidad 1 de Rob Render.

- Partículas emitidas tras el golpe en el suelo del héroe orco (Figura 5.75): escenifica la onda expansiva producida tras el aporreo del suelo por parte del héroe orco tras activarse su segunda habilidad. Se emite una única partícula en el momento de la activación en forma de esfera. El tamaño y el color varían en función del tiempo de vida. Tiene agregadas un *sphereCollider* (esfera de colisión) que aumenta con el paso del tiempo y que emite fuerzas si colisiona con alguna unidad, empujandolas en la dirección oportuna, además de su daño aplicado para reducir la vida.

Figura 5.75: Sistema de partículas de la habilidad 2 de Rob Render.

- Humo producido en el embestida del héroe orco (Figura 5.76): en la tercera habilidad del orco, corre de forma desenfrenada hacia delante dejando un rastro de humo en la carrera. Las partículas tienen cierto retardo en su nacimiento y se emite un número fijo de partículas en forma de cono, A lo largo de su tiempo de vida se aplica una fuerza vertical ascendente para escenificar la ascensión del humo.
- Circulo de fuego alrededor del héroe orco en su embestida (Figura 5.76): tras activarse la tercera habilidad, en la carrera producida por la embestida el orco se recubre de un círculo de fuego. Tiene un tamaño fijo y se emite una única partícula que tiene una rotación variable en función del tiempo de vida de la partícula.
- En la embestida del orco las unidades que son golpeadas salen despedidas tras colisionar y aplicar fuerzas, aplicandoles también su daño en forma de reducción de vida. La unión de los dos últimos sistemas de partículas dan lugar a la escenificación de la tercera habilidad del héroe orco Rob Render.

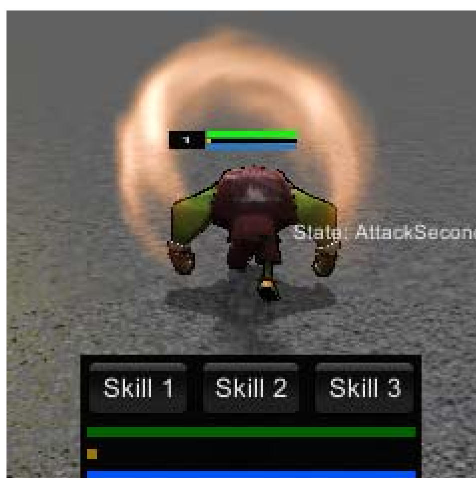


Figura 5.76: Aspecto de los dos sistemas de partículas de la habilidad 3 de Rob Render.

- Círculo protector alrededor del héroe robot (Figura 5.77): tras activarse la primera habilidad del Skelterbot, se recubre de una esfera que escenifica la mejora de

resistencia física y mágica durante un breve periodo de tiempo. Se emite una única partícula con forma de esfera y rota en función del tiempo de vida de la partícula.

Figura 5.77: Sistema de partículas de la habilidad 1 de Skelterbot.

- Viento producido en la segunda habilidad del héroe robot (Figura 5.78): tras activarse, Skelterbot gira blandiendo la espada y junto al sistema de partículas escenifica el viento producido por la misma. Se emite una única partícula en forma de esfera. Tiene un tamaño fijo y un breve retardo. La partícula rota y aumenta de tamaño en función del tiempo de vida. En la duración del movimiento de la espada se crea un sphereCollider que aumenta progresivamente desde el comienzo de la animación, hasta que termina, colisionando con los enemigos cercanos, aplicandoles daño y reduciéndose la vida.

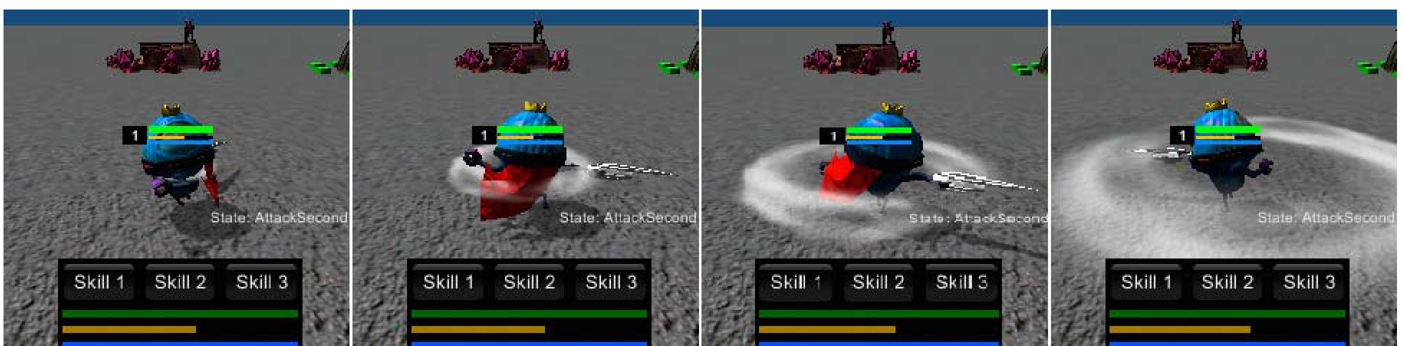


Figura 5.78: Sistema de partículas de la habilidad 2 de Skelterbot.

- Disparo con la pistola en la tercera habilidad del héroe robot (Figura 5.79): escenifica la bola de fuego disparada tras activarse dicha habilidad. Se emite una única partícula con un tamaño fijo al nacer pero que aumenta con el paso del tiempo, también tiene una rotación que varía en función del tiempo. Cuando se dispara, la partícula sigue unida al disparo y su rotación continúa. El disparo posee un rigidBody que controla si el disparo colisiona con algún enemigo, produciendo daño y su correspondiente disminución de vida.

Figura 5.79: Sistema de partículas de la habilidad 3 de Skelterbot.

vi. Menús e Interfaz de Usuario

La interfaz principal del juego se ha desarrollado con la herramienta NGUI (ver capítulo 4.d) y cuenta con 3 paneles que se intercambian según la interacción del usuario. En el primer panel (Figura 5.80) se han empleado dos botones con iluminación propia que se activa cuando se pasa por encima con el cursor y un texto que se muestra letra por letra cuando se inicia. El fondo de la ventana que aparece contiene un objeto de NGUI *texture* que se ve afectado por la iluminación de la escena.

El segundo panel (Figura 5.81) del juego sirve para la gestión de las partidas. Los jugadores tienen que ver las salas disponibles y tener la opción de unirse a una o crear una ellos mismos. Para este diseño se han añadido *sprites* y *labels* mediante el menú de NGUI ya que en esta pantalla se verán las salas disponibles en red y los jugadores que están en ellas. Para el texto se emplean distintas fuentes definidas en el objeto *Atlas* implícito en la escena. El paso entre paneles es animado por eventos de selección con el cursor en los botones de la escena.

En el último panel (Figura 5.83) el jugador está dentro de una sala esperando a los demás jugadores para unirse a la partida y además tiene que seleccionar el modo de juego en el que va a jugar. Para mostrar las opciones se ha usado un boton de selección en el que se muestra los modos de juego y los componentes *widegt* para controlar la lógica de selección de los roles posibles. También se han añadido los personajes en 3D para representar el rol que se quiere y reaccionan a la interacción del cursor mediante el *widget*.

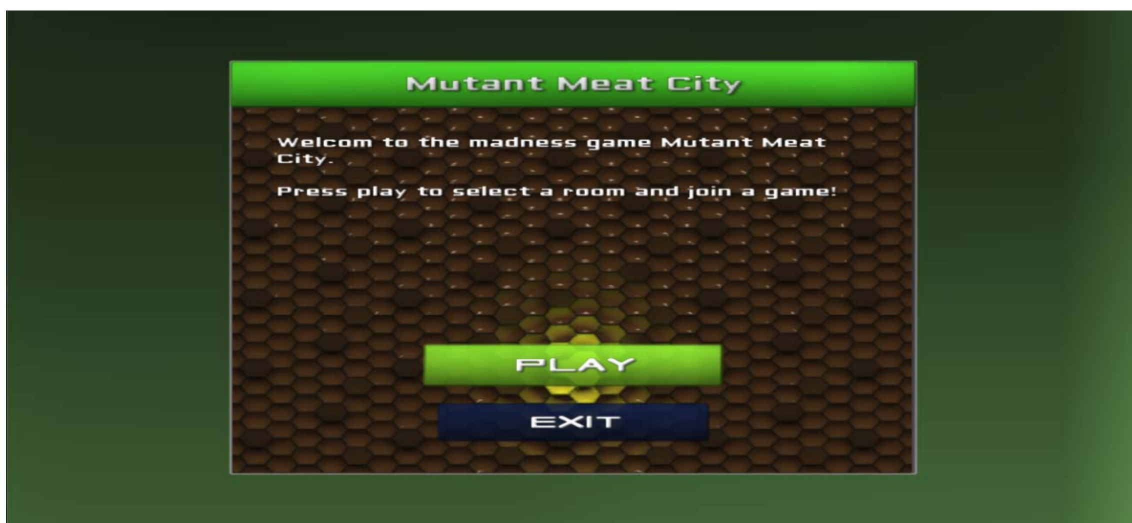


Figura 5.80: Interfaz, panel principal del juego.

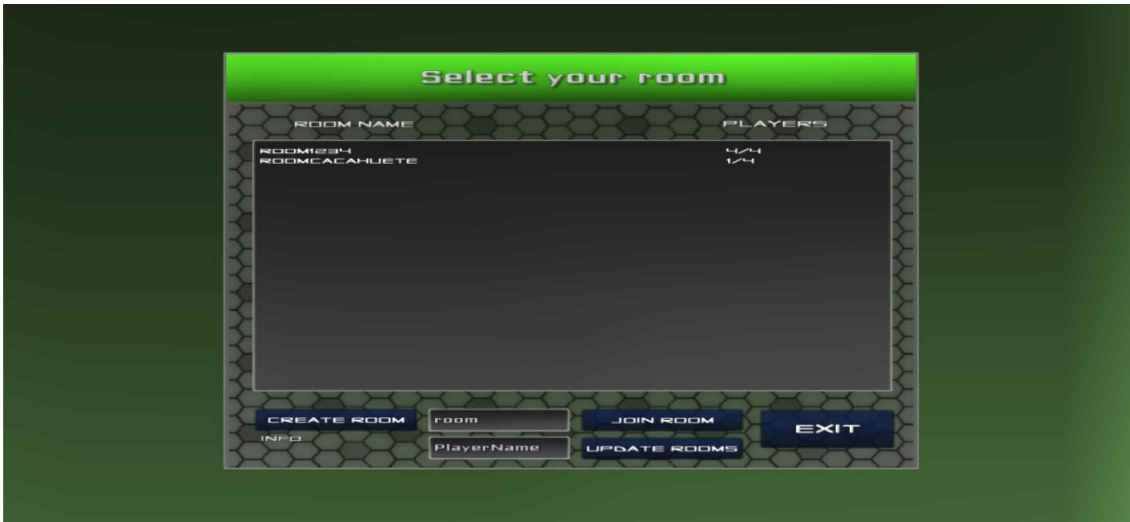


Figura 5.81: Interfaz, panel de selección de sala.

Figura 5.82: Interfaz, panel de selección de rol - boton desplegado.

Figura 5.83: Interfaz, panel de selección de rol - personajes de ejercito.

6. Discusión

En general, el resultado final del proyecto ha sido bueno y las herramientas utilizadas han sido las correctas. La elección más relevante ha sido la del entorno de desarrollo (Unity), que ha sido muy satisfactoria por diversas razones: el elevado volumen de información y documentación que hay, sobre todo, en Internet y del que carecen otros entornos como UDK; el importador de *assets* es muy simple y cómodo; a diferencia que otros entornos posee un editor de materiales y la posibilidad de realizar una GUI vistosa; da la posibilidad de usar un lenguaje de programación sencillo y muy documentado: C#. De todas formas tiene pequeños inconvenientes: el motor gráfico es menos potente que el de otros entornos, como UDK. En resumen, se ha elegido Unity, principalmente, por su sencillez de uso.

Se investigó acerca de la posibilidad de crear una aplicación de servidor y de buscar alguna gratuita. Nos decantamos por Photon porque cuenta con una interfaz de gestión de partidas online por salas, con chat incluidos y un simulador de latencia; todo ello gratuito, pudiendo realizar cien conexiones.

Lo más satisfactorio ha sido el aprendizaje de un entorno de desarrollo 3D muy utilizado en la industria, que además de dar la posibilidad de realizar videojuegos para diversas plataformas, ofrece la oportunidad de realizar otro tipo de aplicaciones de ingeniería del software. En cuanto a la implementación, lo mejor ha sido la integración y sincronización de ambos modos de juego en una misma partida.

Inicialmente, se pensó en la utilización del repositorio *Bitbucket* y del control de versiones *Mercurial*, pero debido a que resultó poco intuitiva, se desechó la idea en favor del repositorio *GitHub* y del control de versiones *Git + Extensions*, que ya se había utilizado anteriormente. Además *GitHub* es un repositorio de código abierto, idea muy acorde con la universidad pública.

Algo de lo que no se está satisfecho es que en algunas ocasiones los *FPS* (tasa de refresco de los gráficos en pantalla) son inferiores a lo deseado, sobre todo, cuando se renderizan grandes cantidades de objetos y unidades. Esto ocurre debido al poco tiempo que se ha dedicado a mejorar la eficiencia en términos de carga gráfica. Tampoco se ha dedicado el tiempo necesario para desarrollar una interfaz de usuario *ingame* completa y satisfactoria. Todo esto forma parte del futuro trabajo que se piensa realizar tras la presentación de este proyecto.

Dejando a un lado la dificultad que implica coger un proyecto avanzado e intentar extenderlo o continuarlo, se considera que está bien estructurado y comentado, y, por lo tanto, no debería suponer una dificultad añadida este aspecto.

En principio, las expectativas de oportunidad de mercado son altas, ya que un juego que mezcle los géneros MOBA y RTS de forma cooperativa no existe y, por lo tanto, en el mercado se valoraría la novedad. La principal dificultad sería la creación de una aplicación de servidor para poder dar una prestación eficiente a los jugadores. Además, se requiere una gran inversión para poder comercializarlo con garantías. Al principio, se intentaría contar, tanto con las subvenciones del ministerio de Industria, Energía y Turismo en forma de ayudas al sector

de videojuego, como con la búsqueda de eventos de videojuegos para aumentar las posibilidades de que esta idea novedosa pueda resultar atractiva a inversores externos.

Teniendo en cuenta este último aspecto, se ha participado en dos eventos:

- Zerouno: se trata de un evento de inversores, jugadores aficionados y desarrolladores de videojuegos. Una de las partes del evento consistía en un concurso de presentación[29] de videojuegos con un video y una breve explicación, todo de un minuto y medio. Se asistió y se mostró el video[30] del juego al público del evento, nuestro juego resultó ganador y, posteriormente, los representantes del evento hicieron una video entrevista[31] a un miembro de los desarrolladores y a otro del grupo de artistas.
- Gamelab: es una de las ferias mundiales más importantes de desarrolladores de videojuegos, se celebra en Barcelona y nuestro proyecto participó con un vídeo[32] y una demo en el concurso a mejor proyecto de videojuegos universitarios[33].

Por otro lado, se consiguió la colaboración, gracias a la mediación de ESNE, de un conocido artista *Youtuber* llamado Rush Smith, que ha realizado un videoclip con imágenes y animaciones del videojuego realizado en este proyecto. Este video[34], en tan solo 4 días llevaba más de 35.000 visitas en YouTube y más de 4.000 *me gusta*.

[29] https://www.youtube.com/watch?v=m97gaXmV1kQ
[30] https://www.youtube.com/watch?v=vMgNFLCOVHA
[31] https://www.youtube.com/watch?v=f3OCu8mJXPk
[32] https://www.youtube.com/watch?v=f0HNy2u1nhQ
[33] http://gamelab.es/premios2014/candidates/view-student/?id=36
[34] http://youtu.be/a2j-IivbSCc

7. Conclusiones

El objetivo principal era tener un nivel online jugable por cuatro jugadores controlando dos ejércitos y dos héroes, de al menos dos razas distintas, y se ha llevado a cabo.

A pesar del tiempo empleado en el aprendizaje del entorno de desarrollo Unity, una vez llevado a cabo, es muy intuitivo y sencillo de utilizar; y, efectivamente, hay un gran volumen útil de documentación e información.

La herramienta utilizada para la implementación de la red del juego (Photon) ha sido de gran ayuda, ya que implementar un sistema online es muy complejo y conlleva bastante tiempo. Gracias a esta tecnología, gran parte de este trabajo ya estaba realizado y sólo se ha tenido que integrar en el proyecto, algo que aun así no ha resultado trivial.

El proyecto es de código libre y está alojado en *GitHub*[35]. La utilización de este repositorio no ha supuesto una dificultad añadida dado que ya se había utilizado con anterioridad.

Uno de los primeros trabajos futuros sería la creación de distintos héroes y ejércitos que puedan otorgar mayor versatilidad al juego. Otro trabajo futuro sería la mejora *ingame*, que se trata de la obtención de mejoras por tesoros y obtención de recompensas durante partidas. Además, la mejora *outgame* consistiría en la selección de paquetes de cartas que doten de ciertas ventajas o cualidades al ejército o héroe durante la partida. Además de mejoras estáticas que afectarían tanto al héroe como al ejército.

Somos conscientes de que para mejorar la eficiencia del juego se necesitaría programar un servidor propio y, por esta razón, es un trabajo futuro a realizar muy importante.

Algo que visualmente atraería más al público sería el diseño de menús. También habría que otorgar variedad al mundo de juego en el que se desenvuelven los jugadores, ya que daría un toque distinto a cada partida, con lo que la variedad de mapas también se encuentra entre lo que se puede seguir desarrollando.

La asistencia a eventos, ya sea como participante o como simple visitante, es fundamental para dar conocer el videojuego. En estos eventos también es posible encontrar *publishers*, inversores, etc. Y finalmente, una idea futura de financiación es la de *crowdfunding*, que consiste en la cooperación colectiva por parte de personas externas a la causa con el fin de conseguir dinero u otros recursos. En nuestro caso sería una financiación colectiva o en masa que aportaría, al menos al principio, la tranquilidad económica para poder dedicarse al desarrollo del videojuego.

[35] https://github.com/maxi-jp/Project_NewDetroit

8. Referencias Bibliográficas

Blackman, S. (2013). *Beginning 3D Game Development with Unity 4* (2nd Edition ed.). Apress.

Carrillo, J., Fornasier, M., Toscani, G., & Vecil, F. (2009). *Particle, Kinetic, and Hydrodynamic Models of Swarming.*

Cuesta Boluda, D., Cuesta Boluda, G., & Rodríguez-Osorio Jiménez, J. (2014). *Producción de un Videojuego Multijugador en Unity Combinando los Géneros MOBA y RTS - Ingeniería del Software y Diseño del Videojuego.*

Ferguson, J., Patterson, B., Beres, J., Boutquin, P., & Gupta, M. (2003). *La Biblia de C#.* Anaya Multimedia.

J. Deitel, P., & M. Deitel, H. (2010). *C# 2010 for Programmers* (4th Edition ed.). Deitel Developer Series.

Lammers, K. (2013). *Unity Shaders and Effects Cookbook.* Birmingham: Packt Publishing.

Norton, T. (2013). *Learning C# by Developing Games with Unity 3D Beginner's Guide.* Birmingham: Packt Publishing Ltd.

R. Stagner, A. (2013). *Unity Multiplayer Games.* Birmingham: Packt Publishing.

Rollings, A., & Morris, D. (1999). *Game Architecture and Design.* Coriolis Group Books.

Sithu Kyaw, A., Peters, C., & Naing Swe, T. (2013). *Unity 4.x Game AI Programming.* Birmingham: Packt Publishing.

Smith, M., & Queiroz, C. (2013). *Unity 4.x Cookbook.* Birmingham: Packt Publishing.

Sriyanyong, P. (2005). *Unit commitment using particle swarm optimization combined with Lagrange relaxation.* Brunel Univ, Brunel Inst. of Power Syst. Power Engineering Society General Meeting.

Unity. (n.d.). *Unity Scripting API*. Retrieved Junio 2014, from http://docs.unity3d.com/ScriptReference/index.html

9. Apéndices

a. Métricas

En esta sección se exponen algunas de las métricas que se han extraido del proyecto a la finalización del mismo, las cuales pueden aportar conocimiento sobre su dimensió, la cantidad de trabajo que se ha realizado o algunas características del videojuego. Se distinguen dos tipos de métricas, a nivel de implementación de código y producción de *assets* y a nivel del propio juego.

i. Métricas a nivel de implementación

Estas métricas son las que se refieren a la implementación del juego. Algunas de estas métricas son:

- Número de tareas en las que se ha dividido el proyecto: 249
- Número de proyectos de prototipos: 4
- Índice de mantenimiento: 80
- Complejidad ciclomática: 14.084
- Máxima profundidad de herencia: 10
- Acoplamiento de clases: 493
- Líneas de código reales: 32.100
- Número de *assets*: 889, de los cuales:
    - Audio: 23, de los cuales:
        - Efectos *SFX*: 21
        - Música: 2
    - Escenas: 25
    - Materiales: 116
    - Modelos 3D: 87, de los cuales:
        - Modelos de personajes con animaciones: 13
        - Modelos de otros *props*: 74
    - Animaciones: 94
    - Sistemas de Partículas: 18
    - Prefabs: 145
    - Shaders: 36
        - De los cuales propios: 20
    - Scripts: 283
        - De los cuales propios: 110
    - Terrenos: 1

    - Texturas: 155, de las cuales
        - Texturas solo con canal difuso (de color): 55
        - Texturas solo con canal specular: 4
        - Texturas con canales difuso y especular: 48
        - Texturas de normales: 48
- Número de horas estimadas: 2650
- Número de horas consumidas: 2391
- Porcentaje completado del juego en horas (según la estimación inicial): 90.2%

ii. Métricas a nivel de juego

Estas métricas son las que se refieren a las propiedades del juego:

- Número de facciones: 2
- Número de héroes por facción: 1
- Número de unidades RTS por facción: 5
- Número de habilidades por héroe: 3
- Número de niveles de los héroes: 4
- Número de ataques básicos encadenados en el héroe: 3

b. Arte

Se entiende que el trabajo realizado en el apartado artístico del juego no corresponde a un proyecto de Sistemas Informáticos, sin embargo, se antoja interesante esbozar parte del gran trabajo realizado en este aspecto, no solo por la importancia que recibe el arte en los videojuegos en general, sino también por el gran trabajo que se ha realizado en este apartado a lo largo de todo el proceso de desarrollo con mucho esfuerzo.

El juego comparte un estilo artístico muy característico, heredado en gran parte de videojuegos de estética desenfadada como *Banjo – Kazooie* (Rareware, 1998) y de videojuegos ambientados en el Miami de los ’80 como Grand Theft Auto: Vice City (Rockstar North, 2002).

Se ha conseguido gracias a la unión de ambos estilos un apartado visual que mezcla esa temática del Miami de los años ’80 con el estilo desenfadado que predominaba en los videojuegos de Rareware. Además, para el diseño de los personajes se ha tomado como referencia una estética futurista distópica como la vista en películas de los 80-90 como *1997: rescate en Nueva York* (*Escape from New York*, John Carpenter, 1981).

A continuación se expone una breve muestra del diseño artístico de los personajes del juego.

i. Ejército Orco

La banda de los Red Lobsters, liderada por Rob Render (ver Figura 9.1), poseen un estilo característico del Miami de los ochenta, la raza posee rasgos asiáticos.

Figura 9.1: Presentación de la banda de los Reb Lobster.

En cuanto al héroe Rob Render, caben destacar detalles como el chaleco, los pantalones, los guantes, el parche pirata, etc, que como hemos mencionado dan ese carácter chocante a la estética. Su gama de animaciones se caracteriza por darle un estilo brusco, teniendo animaciones para el triple ataque básico, andar, correr y sus tres habilidades.

En referencia a las unidades de los Red Lobsters, cabe destacar:

- Unidades recolectoras: se caracteriza por la generación aleatoria de ítems que llevan, de modo que a cada una le dan un aspecto diferente, por ejemplo pueden llevar en la cabeza un cono, un casco obrero, o un sombrero de papel de aluminio. Además cabe destacar su camisa hawaiana y los guantes como su líder Rob Render. Poseen sus propias animaciones de recolección de recursos, ataque y curación.
- Unidades de artillería básica: su estética viene definida por el peinado al estilo afro, las gafas de sol y el tatuaje en el brazo derecho. Poseen su propia animación de movimiento y ataque.
- Unidades de artillería pesada: destacan por un tamaño mayor al resto de unidades del ejército, una máscara de gas, una camiseta sin mangas y una cresta verde. Poseen su propia animación de movimiento, ataque básico y lanzamiento de misiles en despliegue.
- Unidades exploradoras: comparten el mismo modelo que las unidades recolectoras, carecen de la generación aleatoria del casco, pero van montados en un cortacésped. Poseen sus propias animaciones de movimiento y de ataque.
- Unidades ingenieras: comparten el mismo modelo que las unidades de artillería básica, pero llevan un portátil en la espalda. Poseen su propia animación de movimiento, lanzamiento de granadas (ataque básico) y construcción o conquista de torretas.

Edificios (Figura 9.2)

- The Stinky Squid: es el cuartel general de Rob Render y sus secuaces. Está inspirado en un bar de carretera americano.
- Almacén de recursos: típico almacén de recursos del oeste americano.
- Torreta: basada en las torres de vigilancia de las cárceles

Figura 9.2: Aspecto de los edificios del ejército goblin.

ii. Ejército Robot

La banda de los Skelters (Figura 9.3), liderada por el Skelter Bot, son unos robots manejados por gusanos alienígenas. Su estética es variada y pintoresca, con un estilo Andy Warhol.

Figura 9.3: Presentación de la banda de los Skelters.

El líder de este ejército, SkelterBot, es un gusano alienígena con muy mala leche que controla un robot mecánico con piernas y brazos, caracterizado por llevar una capa y un sable-pistola como arma. En cuanto a sus animaciones, se caracteriza por darle un estilo irrisorio para andar y correr. Además tiene animaciones del triple ataque básico, de las que dos de ellas corresponden con dos de sus habilidades.

Las unidades de la banda de los Skelters no manejan todas el mismo robot, de modo que es fácil distinguirlas entre sí:

- Unidades recolectoras: se caracterizan por un estilo tipo traje de buzo antiguo. Al igual que su líder, es un robot manejado por un gusano. Puede llevar un gorro vikingo, un gorro de fiesta, o un trozo de metal generados de forma aleatoria como casco. Llevan colgado un collar con el símbolo del dólar. Poseen sus propias animaciones de recolección, ataque y movimiento.
- Unidades de artillería básica: tienen por cabeza un poliedro con tres ojos en vertical. En este caso es un robot que maneja una escopeta como arma. Poseen sus propias animaciones de movimiento y ataque.
- Unidades de artillería pesada: el modelo se basa en un ovni que tiene un arma pesada, y es controlado por una gelatina verde alienígena con brazos y tres ojos en vertical. Se caracteriza por descubrir al alienígena cuando se despliega. Poseen sus propias animaciones de movimiento, ataque básico y ataque en despliegue.

- Unidades exploradoras: tiene el mismo modelo de la artillería básica, sin su escopeta pero van montados sobre un cohete. Poseen sus propias animaciones de movimiento y ataque.
- Unidades ingenieras: se usa el mismo modelo que la unidad recolectora. No poseen la generación aleatoria de cascos, y tiene sus propias animaciones para moverse, atacar y conquistar o construir torretas.

Edificios (Figura 9.4)

- Skelter Hotel: base principal de los Skelters, han construido un hotel alrededor de su nave espacial.
- Almacén de recursos: el cuartel general de Skelterbot y su equipo. Inspirado en una base del Área 51 del desierto de Nevada.
- Torreta: se asemeja a una atracción de un parque infantil

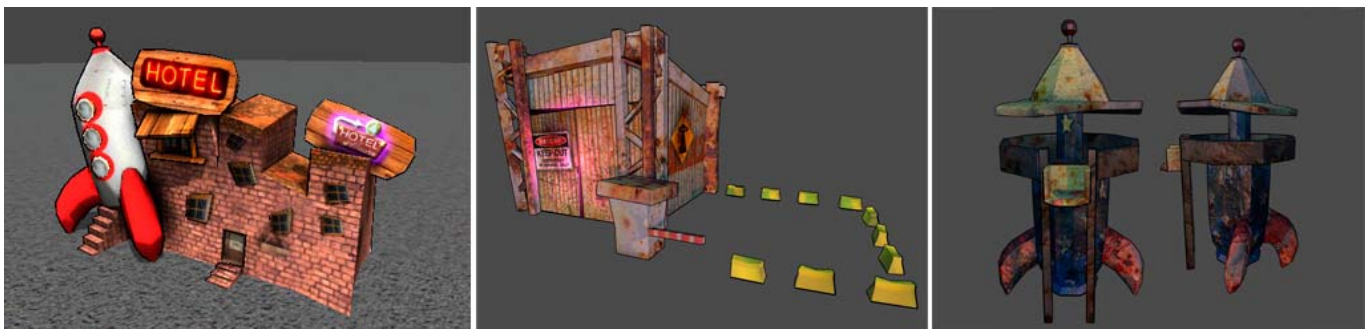


Figura 9.4: Aspecto de los edificios del ejército robot.

iii. Assets del Escenario

A continuación, se expone una breve muestra de varios assets creados para el juego.

Torretas neutrales (Figura 9.5): Creado a partir de la chatarra que encuentran en los vertederos de residuos. Al comienzo de la partida

Figura 9.5: Aspecto de las torretas neutrales.

Gas Station (Figura 9.6): Gasolinera que podremos encontrar en el escenario.

Figura 9.6: Gasolinera del escenario.

Goblin District Building (Figura 9.7): edificios que conforman el barrio de los Goblins, cuentan con un estilo a los bloques de apartamentos que se pueden encontrar en los barrios chinos de las grandes ciudades americanas.

Figura 9.7: Edificio del distrito goblin.

Poor District Building (Figura 9.8): edificio semi en ruinas de un barrio pobre de NewDetroit.

Figura 9.8: edificio del barrio pobre de NewDetroit.

NewDetroit Cathedral (Figura 9.9): catedral de estilo gótico moderno de NewDetroit.

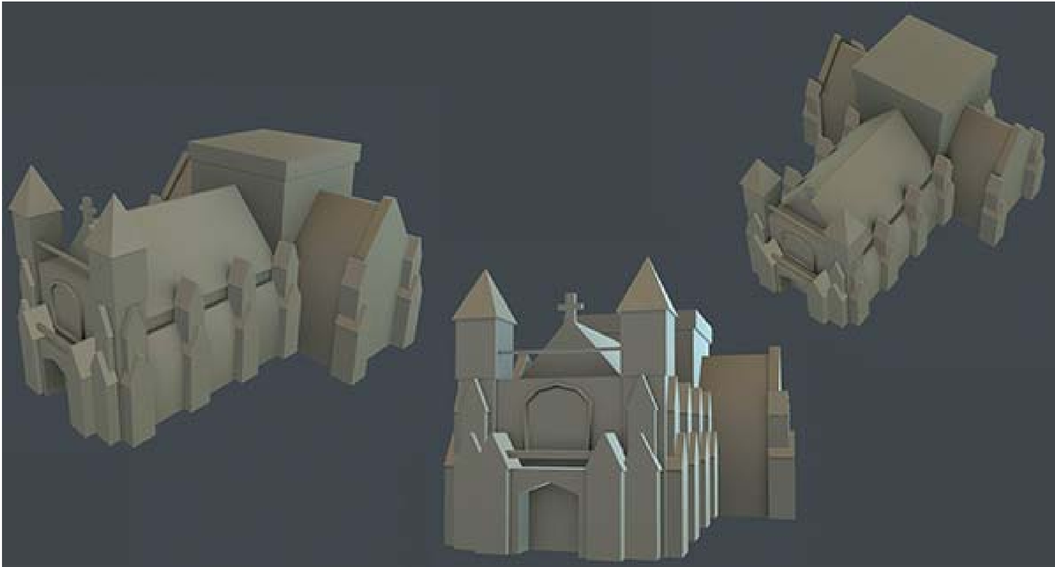


Figura 9.9: Catedral neo-gótica de NewDetroit.

NewDetroit City Bridge (Figura 9.10): puente principal que une los dos principales barrios de NewDetroit, esta basado en el puente real de la ciudad de Detroit en EEUU. Sirve de nexo entre los 2 territorios enfrentados.

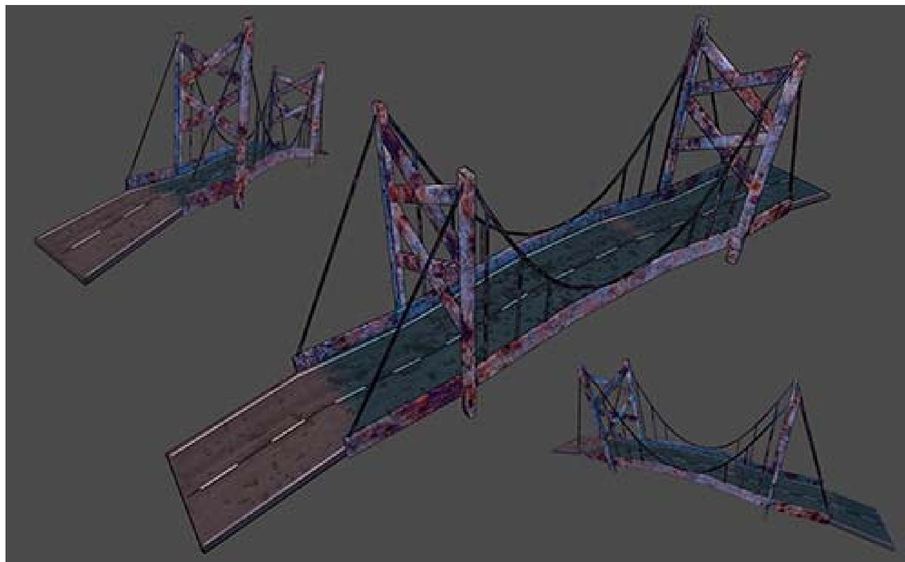


Figura 9.10: Puente principal de la ciudad de NewDetroit.

NewDetroit Trees (Figura 9.11): árboles que pueblan las pocas zonas verdes de la ciudad.

Figura 9.11: Set de árboles de NewDetroit.

Parada de Metro "Glitter Caverns" (Figura 9.12): estación del metro de NewDetroit, de donde manan minerales, codiciados por las pandillas que pueblan la ciudad.

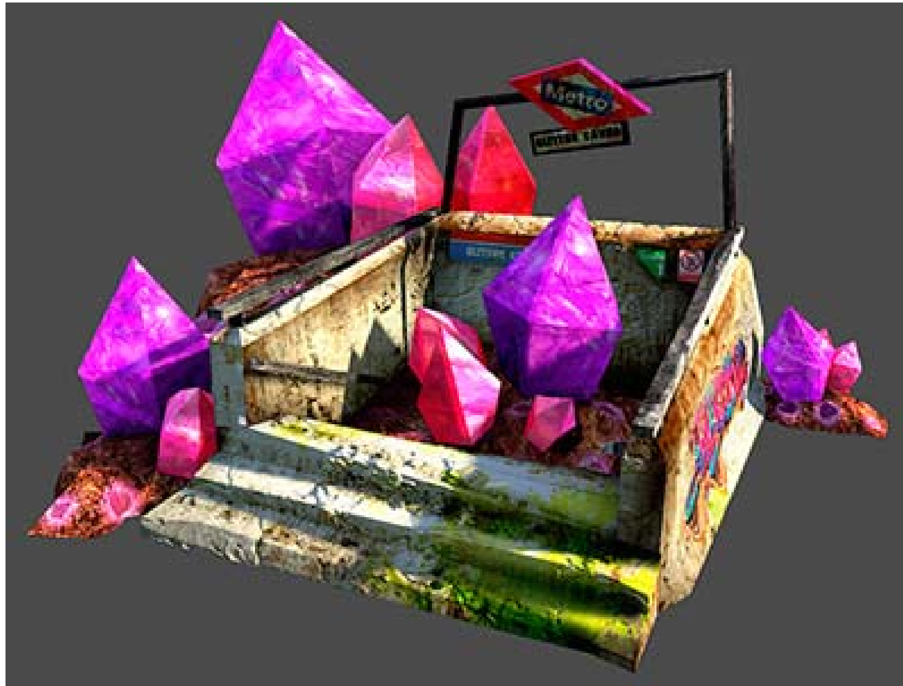


Figura 9.12: Parada de Metro de NewDetroit.

Dispensadores de periódicos de NewDetroit (Figura 9.13): muebles urbanos que sirven como dispensadores de los dos principales periódicos de la ciudad de NewDetroit, "The NewDetroit Post" y "The Unreason".

Figura 9.13: Dispensadores de periódicos de NewDetroit.

Set de Assets1 (Figura 9.15): set de objetos de escenario, formado por un semáforo, una señal de Stop y una boca de incendios.

Set de Assets2 (Figura 9.14): cuenta con cubos de basura, contenedores y un poste de alta tensión.

Figura 9.14: Set de assets 2 de NewDetroit.

Figura 9.15: Set de assets 1 de NewDetroit.

Chopper Motorbike (Figura 9.16): motocicleta *Chopper* que decora los exteriores de la base de los Red Lobster.

Figura 9.16: Motos Choper de los Red Lobster.

c. Manual de Instrucciones Básicas

Cuando se ha iniciado el juego y elegido una sala para empezar a jugar, se presenta un menú en el que se puede elegir entre dos modos de juego:

MOBA mode (*Hero*)

En este modo de juego se controla a uno de los dos héroes Rob Render o, en el otro bando, al héroe Skelterbot. Para ello se controla su movimiento mediante las teclas WASD para moverlo hacia adelante, atrás y girar. Para combatir a sus enemigos vale con mantener pulsado el botón izquierdo del ratón para golpear a los que se tengan justo delante. Con el clic derecho del ratón se puede posicionar justo detrás la cámara desde la que se observa al héroe. También cuenta con tres habilidades para abatir a los enemigos más duros, para activarlos hace falta tener suficiente nivel y pulsar uno de las teclas 1,2 o 3.

RTS mode (*Army*)

En este modo se controla al resto del ejército de Rob, o de Skelterbot. Se puede seleccionar la base haciendo clic con el botón izquierdo del ratón poniendo previamente el cursor encima de esta. Después de seleccionarla se tornará del color del equipo y se podrán desplegar unidades al punto de inicio (éste se puede cambiar con el clic derecho del ratón mientras la base esté seleccionada). Mientras esté seleccionada la base se crean unidades con las teclas 1, 2, 3, 4 y 5. Para mandar las órdenes a las unidades desplegadas primero hemos de seleccionarlas (más adelante se ve cómo hacerlo). Después se selecciona el enemigo o el lugar con el que queremos que interactúe posicionando el cursor encima del sitio y pulsando el clic derecho del ratón.

A continuación se explican las características de las unidades que se pueden desplegar, para usar sus habilidades recordamos que han de estar seleccionadas antes:

1. **Unidad recolectora**: esta unidad es capaz de recolectar cristal de las minas y llevarlo a la base para conseguir recursos que utilizaremos en el despliegue de unidades. También pueden curar a las unidades amigas.
2. **Unidad artillera ligera**: esta unidad es capaz de disparar a las otras unidades a distancia.
3. **Unidad artillera pesada**: esta unidad es más potente que la anterior y además puede desplegarse en un sitio para hacer más daño a sus oponentes y aumentar su rango. Para hacerlo hay que tener esta unidad seleccionada y pulsar la tecla “D”.
4. **Unidad ingeniera**: esta unidad es capaz de construir torres que defienden a tus unidades y centros de recolección de recursos para que los recolectores vayan a depositar los recursos a estos. Para ello debemos tener el ingeniero seleccionado y pulsar las teclas “T” y “W” respectivamente, y después seleccionar el lugar donde quieras que se construya. Esta unidad también puede reparar edificios dañados, ayudar a completar los edificios en construcción y conquistar las torres neutrales.
5. **Unidad exploradora**: esta unidad es más rápida y más barata que los demás, pero más débil por lo que nos sirve para explorar el terreno en busca de enemigos. Además tiene la habilidad de recorrer rutas marcadas por el jugador. Esto se hace

marcando puntos con el clic izquierdo del ratón mientras tenemos pulsada la tecla "P". Estas unidades también pueden saltar ríos.

Se puede eliminar a tus propias unidades si pulsamos la tecla "suprimir" con la unidad seleccionada.

Existe un modo de ataque automático con las unidades de artillería (ligera y pesada). Este se activa con la tecla "A" presionada mientras hacemos clic izquierdo en un lugar del mapa, de esta manera los artilleros que tuviéramos seleccionados se desplazaran hasta ese punto aniquilando a todo enemigo que se encuentren a su paso.

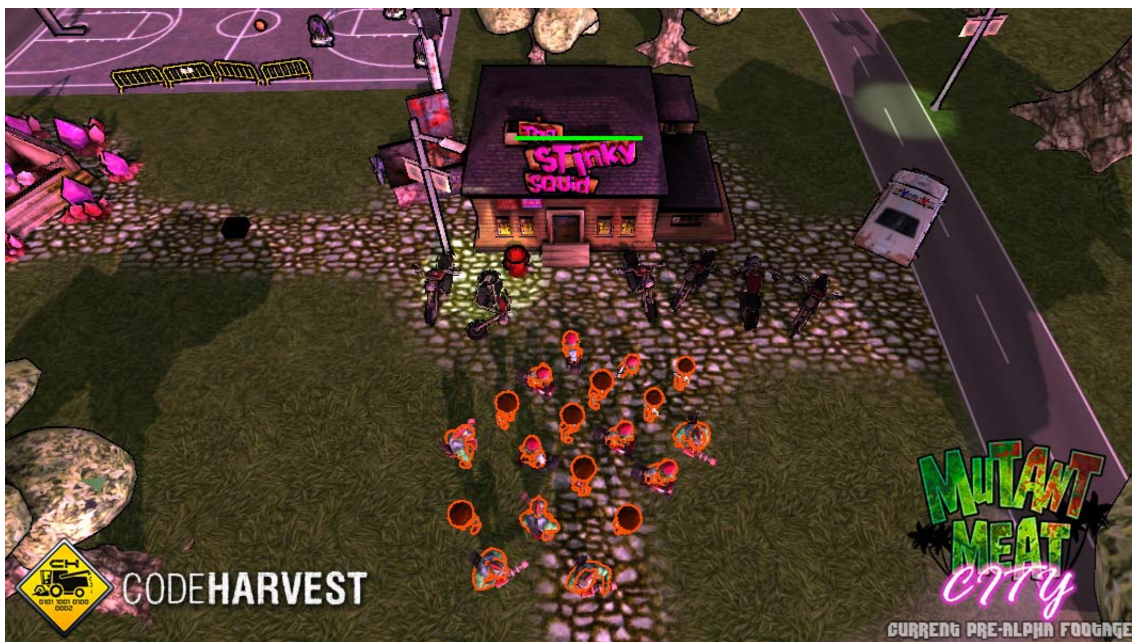
En cuanto a la selección de unidades:

- Se puede realizar selección múltiple dejando pulsado el botón izquierdo del ratón y desplazando el puntero.
- Se puede realizar selección múltiple haciendo doble clic con el botón izquierdo del ratón, y así se seleccionan todas las unidades del mismo tipo.
- Se puede realizar selección múltiple mediante la tecla Ctrl del teclado y seleccionando la unidad que se quiera, de manera que se mantienen seleccionadas las unidades que se tenían y se añade esta última.

El objetivo de la partida es destruir la base rival sin que el enemigo destruya la tuya antes.

e. Capturas del juego

A continuación se muestran varias capturas que sirven para ilustrar el estado del videojuego a la finalización de este proyecto.



Team_A
Total resources: 172598
Economy: 195813
Total Units: 21
Gas Station
CODEHARVEST
MUTANT
MEAT
CITY
CURRENT PRE-ALPHA FOOTAGE

CODEHARVEST
MUTANT
MEAT
CITY
CURRENT PRE-ALPHA FOOTAGE

CODEHARVEST
MUTANT
MEAT
CITY
CURRENT PRE-ALPHA FOOTAGE

CODEHARVEST
MUTANT
MEAT
CITY
CURRENT PRE-ALPHA FOOTAGE

CODEHARVEST
0101 1001 0100 0002
MUTANT MEAT CITY
CURRENT PRE-ALPHA FOOTAGE

CODEHARVEST
0101 1001 0100 0002
MUTANT MEAT CITY
CURRENT PRE-ALPHA FOOTAGE

Skill 1
Skill 2
Skill 3
375 / 375
80 / 150
175 / 175
CODEHARVEST
MUTANT MEAT CITY
CURRENT PRE-ALPHA FOOTAGE

Skill 1
Skill 2
Skill 3
375 / 375
5 / 150
175 / 175
CODEHARVEST
MUTANT MEAT CITY
CURRENT PRE-ALPHA FOOTAGE

State: AttackSecond
Skill 1
Skill 2
Skill 3
375 / 375
150 / 150
130 / 175
CODEHARVEST
MUTANT MEAT CITY
CURRENT PRE-ALPHA FOOTAGE

State: Idle
Skill 1
Skill 2
Skill 3
375 / 375
150 / 150
25 / 175
CODEHARVEST
MUTANT MEAT CITY
CURRENT PRE-ALPHA FOOTAGE

Team 1 conquered
Team 0 conquered
Team 1 conquered
State: AttackSecond
Skill 1
Skill 2
Skill 3
375 / 375
80 / 150
175 / 175
CODEHARVEST
MUTANT MEAT CITY
CURRENT PRE-ALPHA FOOTAGE

CODEHARVEST
MUTANT MEAT CITY
CURRENT PRE-ALPHA FOOTAGE

g. Glosario de términos

- **Aliasing**: es el efecto visual que se produce al intentar representar una imagen con curvas y líneas inclinadas en una pantalla. Como la resolución es finita, se ve un efecto visual en el que las curvas se muestran dentadas debido a que están compuestas por pixeles.
- **As a service** (*Software as a service*): es un modelo de distribución de software. El soporte lógico y los datos se alojan en servidores a los que se accede por un navegador web a través de Internet. La empresa contenedora de todo esto es una compañía de tecnología de información y comunicación (TIC), y ofrece un servicio de mantenimiento y del soporte de software usado por el cliente, así como actividades desde ubicaciones centrales en lugar de la sede de cada cliente, la distribución según el modelo uno-a-muchos (una instancia con múltiples usuarios) y actualizaciones centralizadas, lo cual elimina la necesidad de descargar paquetes por parte de los usuarios finales.
- **Asset**: cualquier archivo del proyecto, ya sean ficheros de material gráfico (como modelos 3D, texturas, materiales, etc.), ficheros de audio, scripts de código fuente o shaders.
- **Atlas**: textura de gran tamaño que se usa para proyectar las texturas que contiene **.**
- **DeltaTime**: tiempo que ha transcurrido desde el último *frame*.
- **Drawcall**: llamada a la función que tiene un objeto que está presente en la escena para que el motor gráfico lo pinte y pueda ser visible para el usuario.
- **Dummy**: marca una posición de un modelo 3D dentro de su jerarquía interna (p.e. un dummy en la mano para marcar la posición donde tiene que ir un martillo, de este modo el martillo seguiría la mano al moverse).
- **FixedDeltaTime**: tiempo (en segundos) que tarda en realizarse la física.
- **FPS** (*frames per second*): imagenes por segundo, es la frecuencia a la que un reproductor gráfico genera distintos fotogramas.
- **Gameplay**: *ver jugabilidad*.
- **GameObject:** clase base de la que heredan todos los elementos de una escena.
- **Gizmo**: se usa para depurar visualmente o como ayudas de configuración en la escena visual.
- **Idle**: estado que representa reposo.
- **Jugabilidad:** *aquello que hace el jugador*. En diseño y análisis de juegos es un término que describe la calidad del juego en términos de sus reglas de funcionamiento y de su diseño como juego. El famoso diseñador de videojuegos Sid Meier definió la *jugabilidad* como "Una serie de decisiones interesantes" (Rollings & Morris, 1999).
- **Layout**: en el ciclo de juego, es el evento que se envía antes que cualquier otra cosa.
- **Malla**: estructura de datos que contiene los datos necesarios para representar un objeto tridimensional como son los vértices, índices, normales y coordenadas de textura. En

videojuegos se suele utilizar este término para referirse a esta representación tridimensional de un objeto.

- **Minijuego:** un juego con una mecánica sencilla.
- **Minions:** personajes menos poderosos que los héroes que se controlan en la parte RTS del videojuego (*unidad de artillería, unidad exploradora, etc.*).
- **Mipmapping**: efecto básico en programación gráfica que se encarga de difuminar las texturas para que no "pixelen", suavizándola, usando copias en menor resolución de la textura original.
- **MOBA**: (siglas en inglés de *Multiplayer Online Battle Arena*) videojuego de estrategia de acción en tiempo real, es un sub-género de los RTS en el que el jugador no gestiona un ejército sino un único héroe.
- **Prefab (Prefabricado)**[36]: es una copia de un *GameObject* convertido a *Asset* reusable.
- **Prop**: son objetos del juego formados por un conjunto de *Assets* (p.e. un personaje controlable con su modelo 3D y sus scripts asociados).
- **Repaint**: en el ciclo de juego, es el evento que se envía una vez por frame. Primero se envían los otros eventos y después se envía este.
- **Rigidbody:** componente físico de *Unity* que toma el control sobre la posición del objeto que lo tenga agregado y lo afecta de forma que simula la influencia de gravedad y calcula cómo responde a colisiones.
- **RTS**: (siglas en inglés de *Real-Time Strategy*) videojuego de estrategia en tiempo real en el que el jugador, entre otras cosas, gestiona un ejército en base a una economía.
- **Rush**: ataque rápido y por sorpresa.
- **Shader**: pequeños programas que se ejecutan en la GPU. Para más información consultar el capítulo de *Shaders*.
- **Shooter:** género o jugabilidad que consiste en disparar a otros personajes para poder conseguir un objetivo.
- **Skybox**: textura que envuelve a la escena para recrear el cielo y todo lo que hay en la lejanía.
- **Spawn:** en videojuegos, se utiliza el término *engendrar* para referirse a la aparición de un objeto en el juego en un momento determinado.
- **Sprite**: un tipo de mapa de bits que se dibujan sin generar cálculos adicionales en la CPU.
- **Tower defense:** género basado en defender una base de oleadas de enemigos que pretenden destruirla.
- **Update**: en programación de videojuegos es el método que se encarga de actualizar el estado y atributos de los objetos del juego. Es invocado con una determinada frecuencia (normalmente entre 30 y 60 veces por segundo).

[36] http://docs.unity3d.com/Manual/Prefabs.html

- **WASD:** esquema de control en videojuegos de ordenador, se refiere al uso de las teclas “W”, “A”, “S”, “D” para mover al avatar del jugador hacia adelante, izquierda, atrás y derecha respectivamente.
- **Yield**: suspende la ejecución de una corrutina para reanudarla en el siguiente frame desde el punto en que se suspendió, manteniendo los valores como estaban.