UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Creación de un videojuego en UNITY 3D utilizando RT-DESK

Proyecto Final de Carrera

Ingeniería Técnica en Informática de Sistemas

Autor: Javier Garrigues Tudela

Director: Ramón Mollá Vayá

Julio 2014

Resumen

Este proyecto tiene como objetivo principalmente dos puntos: por un lado implementar, desde cero, una demo de un videojuego en 3D (uno o dos niveles). Por otro lado acoplar al videojuego la tecnología RT-DESK que han desarrollado profesores de la UPV, y que aporta una gran optimización de los recursos que el software utiliza.

El proyecto servirá para comparar la eficiencia y el ahorro de recursos de la tecnología RT-DESK, frente al enfoque convencional que tiene actualmente la industria de los videojuegos a la hora de desarrollar la mecánica interna del software.

Este proyecto también servirá como introducción al desarrollo de un videojuego completo, de plataformas 3D en tercera persona, cuya temática está inspirada en la famosa película “Toy Story”. Además también pretende dar una visión general de la dificultad que conlleva la tarea de crear un videojuego.

Palabras clave: software, juego, videojuego, 3D, RT-DESK, ingeniería, proyecto.

Tabla de contenidos

1. INTRODUCCIÓN

"El tablero es el mundo; las piezas son los fenómenos del Universo; las reglas del juego constituyen lo que conocemos como leyes de la naturaleza."

Thomas Henry Huxley

Todo Proyecto Final de Carrera debe estar bien definido, bien estructurado, y debe tener unas metas claras. Además debe aportar a la comunidad información relevante en el campo al que se refiere. En estos términos, espero que esta memoria refleje bien el trabajo realizado y que este proyecto cumpla con las expectativas del lector.

1.1 Motivación

La industria de los videojuegos es, entre otras cosas, cambiante, creciente y evolutiva. Su tendencia a la innovación constante la convierte en un ámbito profesional muy interesante. Las posibilidades son casi ilimitadas. Con el paso del tiempo surgen nuevas ideas y nuevas formas de llevar a cabo esas ideas, nuevas tecnologías que facilitan el trabajo a los desarrolladores, logrando resultados inesperados tan sólo unos años atrás.

El 90% de los gamers españoles considera que en la próxima década, jugar a videojuegos será una actividad habitual para todos los sectores poblacionales **[ADESE]**.

Imagen 1: Penetración del videojuego en el futuro

Parece ser que en los próximos años el sector del ocio interactivo promete seguir acarreando un desarrollo al alza. Las empresas que se dedican a la creación y distribución de software interactivo son una buena apuesta profesional y personal.

Además de las buenas expectativas que hay por parte de este sector en cuanto a la inserción al mercado laboral, la motivación para la realización de este proyecto viene dictaminada por varias razones. Por ejemplo la realización de un videojuego requiere la utilización de técnicas muy diversas en el ámbito del trabajo de un informático tales como: gráficos, inteligencia artificial, ingeniería del software, redes de comunicación, etc. Por este motivo este proyecto es una muy buena forma de poner en práctica gran parte de los conocimientos adquiridos durante la carrera, y es además todo un desafío.

1.2 Objetivos

El objetivo principal del proyecto es el diseño e implementación de la demo de un videojuego de plataformas en 3D. En segundo lugar se pretende realizar una portabilidad del juego al paradigma discreto desacoplado. Para ello se utilizará RT-DESK, un núcleo de simulación de aplicaciones gráficas en tiempo real que funciona mediante la gestión de eventos ordenados temporalmente y que ha sido desarrollado en la UPV.

Para el desarrollo del videojuego se utilizarán dos herramientas software muy conocidas en el mundo de los videojuegos y de la animación digital: 3D Studio Max y Unity 3D.

3D Studio Max será el programa utilizado para todo lo relacionado con la parte artística (con excepción de la iluminación):

- Modelado 3D de los escenarios.
- Modelado de los personajes (personaje principal y enemigos).
- Animación de los personajes.

Una vez creados los modelos y las animaciones, estos se exportan a Unity 3D. Unity 3D se encarga de la parte de programación del videojuego:

- Creación de las clases en C# que componen el videojuego.
- Iluminación de los escenarios.

En cuando al acople de la tecnología RT-DESK, este proceso consta de tres partes:

- Migración del código escrito en C# a C++, que es el lenguaje nativo en el que está escrito el software RT-DESK.
- Integración del código en Unity 3D.
- Integración del código resultante en el videojuego.

1.3 Estructura de la obra

Este documento está organizado de manera que sea relativamente fácil encontrar aquello que se busca. En este apartado se pretende aclarar la estructura del mismo, así como resaltar los puntos más relevantes.

El documento consta de cuatro partes esenciales para un proyecto: INTRODUCCIÓN, ANÁLISIS, IMPLEMENTACIÓN Y CONCLUSIONES/RESULTADOS.

- **INTRODUCCIÓN:** En ella se exploran los diferentes objetivos que espero lograr con este proyecto y la motivación que me ha llevado a realizarlo.
- **ANÁLISIS:** En esta fase analizo las diferentes herramientas de trabajo que existen y el porqué de haber elegido dichas herramientas. Se exponen las razones para haber elegido 3D Studio Max para el modelado y mapeado 3D y Unity para la implementación del videojuego. Además encontramos la metodología a seguir para el desarrollo del proyecto, así como la planificación del mismo.
- **IMPLEMENTACIÓN:** Se explica en esta fase el proceso de implementación del videojuego, explorando la herramienta Unity como motor principal de desarrollo.
- **CONCLUSIONES / RESULTADOS:** Estos dos puntos muestran una visión en retrospectiva del desarrollo del proyecto y los resultados obtenidos. Se remarca la diferencia de tiempo que ha habido entre la planificación original y el tiempo total que finalmente ha conllevado la realización del proyecto.

2. ANÁLISIS

"*Sometimes it pays to stay in bed on Monday, rather than spending the rest of the week debugging Monday's code.*"

Christopher Thompson

A la hora de realizar cualquier proyecto, de cualquier envergadura, el proceso de planificación del mismo supone una parte esencial. Un proyecto con una buena planificación es un proyecto con una estructura inicial sólida y precisa.

2.1 La industria de los videojuegos

En los últimos años la industria de los videojuegos ha sabido mantenerse como principal industria de ocio audiovisual e interactivo, con una cuota mercantil muy superior a la de industrias como el cine o la música.

A pesar de la dura situación económica en la que se encuentra la mayor parte de los países del globo, la industria de los videojuegos se mantiene en alza. Esto se debe, en gran parte, a que este sector continúa apostando por la innovación y adaptando sus productos a nuevas tendencias de consumo. Por este motivo los principales estudios de mercado la sitúan como la industria tecnológica con mayor proyección de crecimiento. Los analistas aseguran que el sector será uno de los protagonistas en la denominada "economía de la innovación" **[AEVI]**. Una de las grandes innovaciones que posee este sector es sin duda su apuesta por el modelo "online" de ventas y la multiplicación de las funcionalidades de los videojuegos, que empiezan a separarse ya del puro ocio, para abarcar nuevos campos de vista.

En nuestro país el videojuego se posiciona como la principal opción de ocio para cada vez más segmentos poblacionales. Así, la a penetración social del videojuego continúa aumentando en España, y es que a día de hoy el 62% de los menores de edad y el 24% de los adultos españoles se declaran ya usuarios habituales, según el último estudio realizado por la consultora *Gfk* (*Gesellschaft für Konsumforschung*) **[AEVI]**.

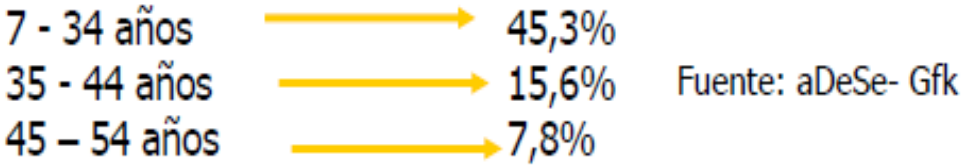


Imagen 2: Usuarios y edades.

España es hoy la cuarta potencia de Europa en consumo de videojuegos, con cifras de ventas que superan los 1.245 millones de euros **[ADESE]**.

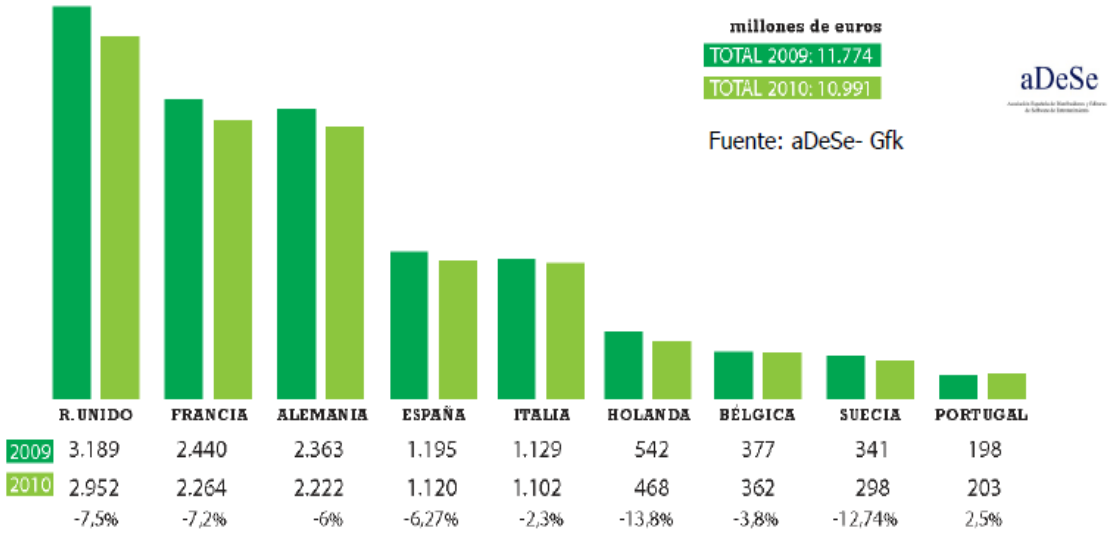


Imagen 3: Consumo de videojuegos en Europa.

2.2 ¿Por qué Unity 3D?

Unity 3D es una herramienta para la creación de videojuegos conocida mundialmente y utilizada por millones de usuarios. Es un motor de desarrollo para la creación de videojuegos interactivos 2D y 3D. Su metodología flexible y su versatilidad han hecho que en los últimos años el número de particulares y empresas que utilizan Unity para el desarrollo de sus aplicaciones crezca de manera exponencial.

Hoy en día Unity cuenta con 2.9 millones de desarrolladores registrados, en parte gracias a que con Unity es posible exportar proyectos a 17 plataformas distintas, siendo Windows, Mac, Android e iOS las más conocidas.

Además de lo ya mencionado, mi decantación por este motor viene de la mano de la experiencia que poseo en el manejo del mismo, llevo ya más de un año manejando este programa a modo de *hobbie* para proyectos pequeños. Durante este tiempo Unity ha probado ser una gran herramienta de trabajo. Su diversidad y fiabilidad la convierten en una gran apuesta de futuro.

Aunque Unity proporciona una gran versatilidad en cuanto a la mecánica del videojuego, las físicas y algunos elementos gráficos como la iluminación, texturas, etc. no es un programa que proporcione funcionalidades de modelado de objetos en 3D. Por ello los modelos que se utilizan en un videojuego creado con Unity, normalmente se importan desde otros programas de modelado y animación 3D. Los más conocidos son: Autodesk 3D Studio Max, Autodesk Maya, Cinema 4D y Blender.

Para este proyecto el programa de creación de modelos 3D y animaciones que he escogido es Autodesk 3D Studio Max.

2.3 ¿Por qué 3D Studio Max?

3D Studio Max es una herramienta de modelado y animación 3D, probablemente la más conocida en el sector. Este programa es obra de la compañía Autodesk, mundialmente conocida por sus aplicaciones en el mundo de la arquitectura y la infografía, con programas como AutoCAD, que utilizan millones de estudiantes de arquitectura y arquitectos titulados.

3D Studio Max ha sido utilizado para la creación de películas de animación, así como para efectos especiales e integración de modelos 3D realistas en películas de carácter realista. Algunas películas para las que se ha utilizado este software son: 2012, Avatar, Iron Man, Minority Report, Planet 51, Spider-Man 3, Matrix Reloaded, X-men, etc.

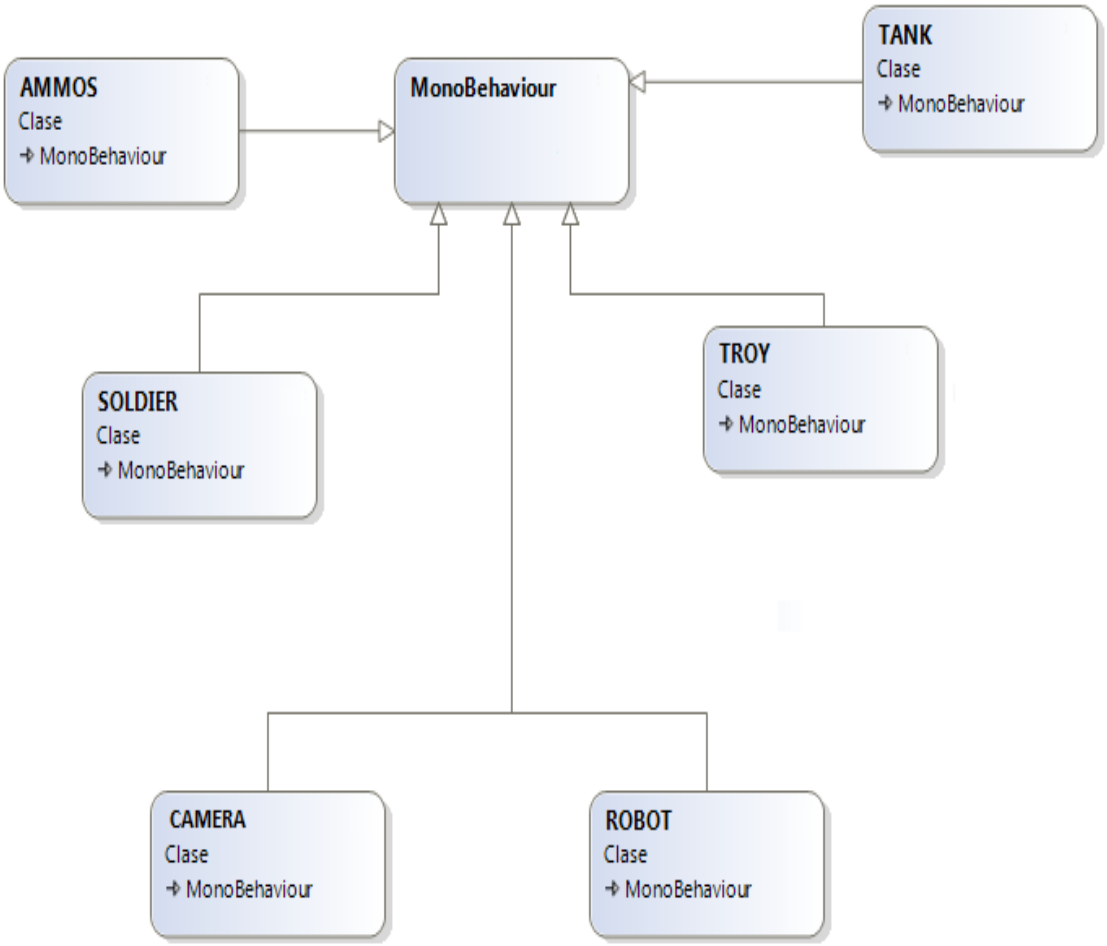
Además de en cine, 3D studio Max es uno de los principales software de modelado 3D para videojuegos. Por ello, y porque ya hace varios años que llevo manejando este programa, me decanté por él para la realización del proyecto.

También existen otras herramientas de modelado y animación 3D, por ejemplo: Maya, Blender, Cinema 4D, etc. La verdad es que todas ellas son muy útiles y complejas, pero en cuanto a producciones profesionales 3D Studio Max lleva la delantera, entre otras cosas porque el abanico de posibilidades que ofrece es mucho mayor que el del resto de programas, y la complejidad de los proyectos realizados en 3D Studio Max sólo se pude comparar con la complejidad del programa en sí, que es muy elevada.

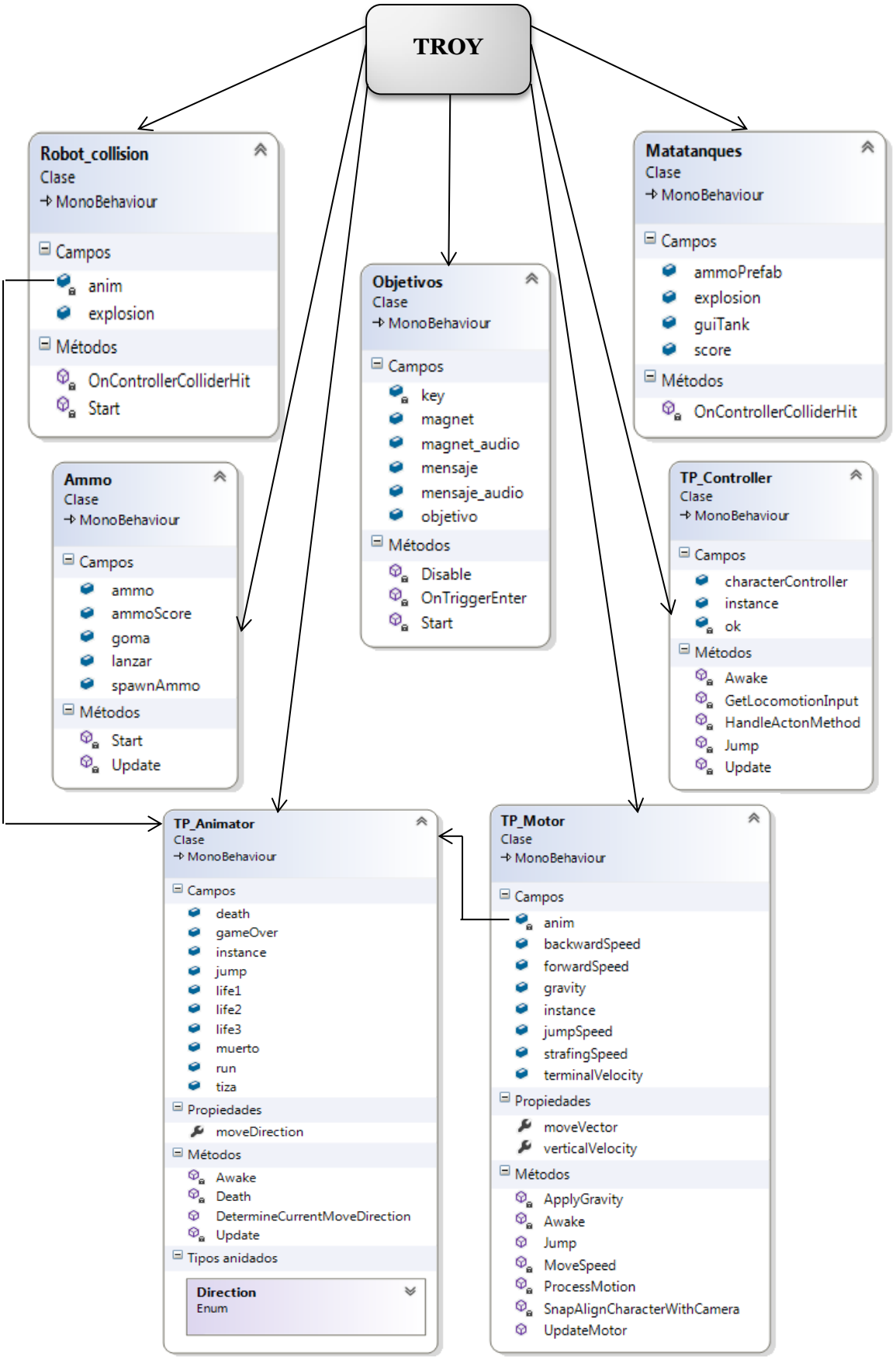
2.4 Diagramas de clase

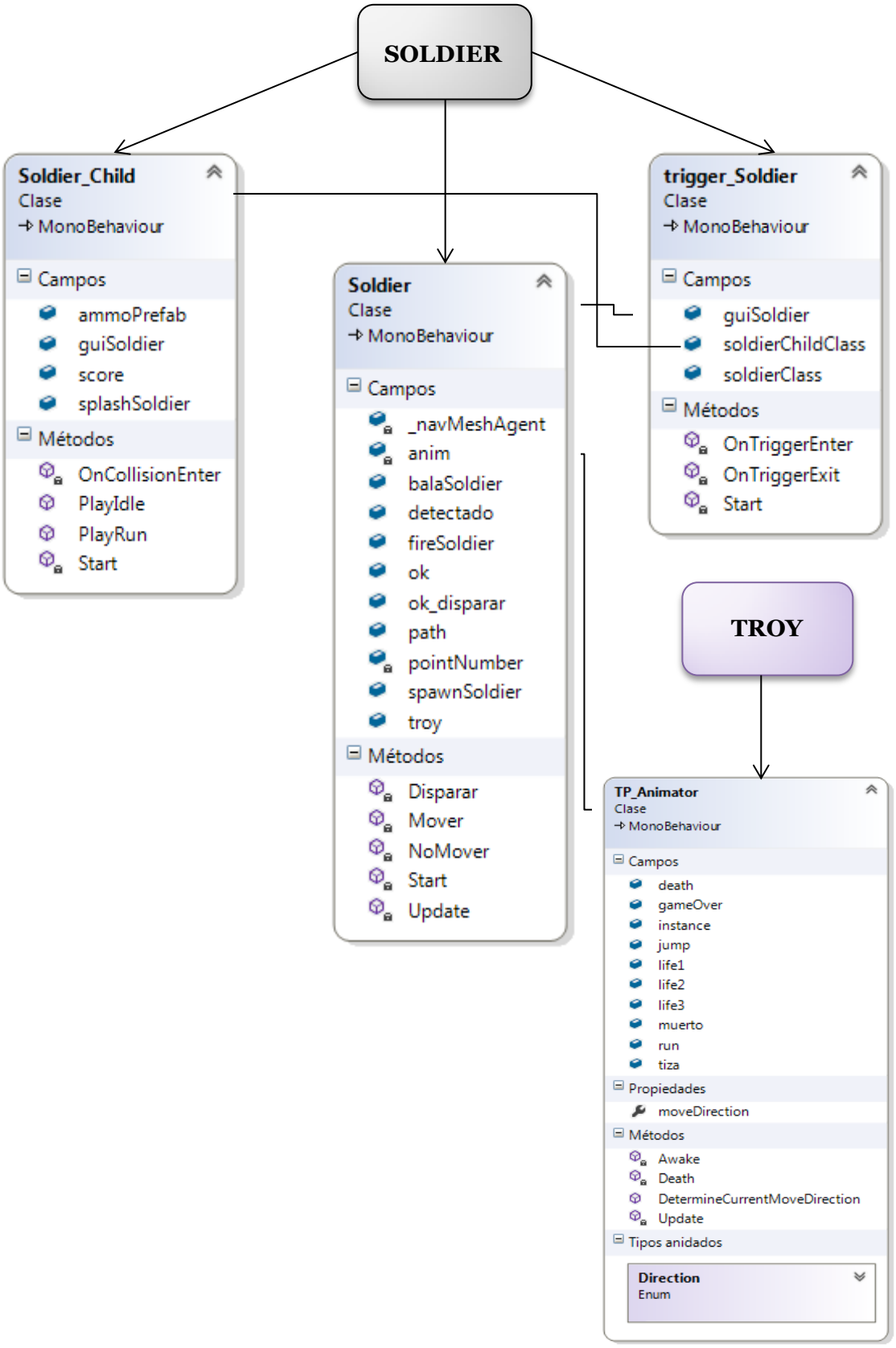
En este apartado exploramos las diferentes clases que componen el juego, así como la interconexión que existe entre ellas. También se muestra el mecanismo de herencia por el que se rige la programación de videojuegos en Unity. En Unity todas las clases que componen el juego heredan de la clase principal: *MonoBehaviour*.

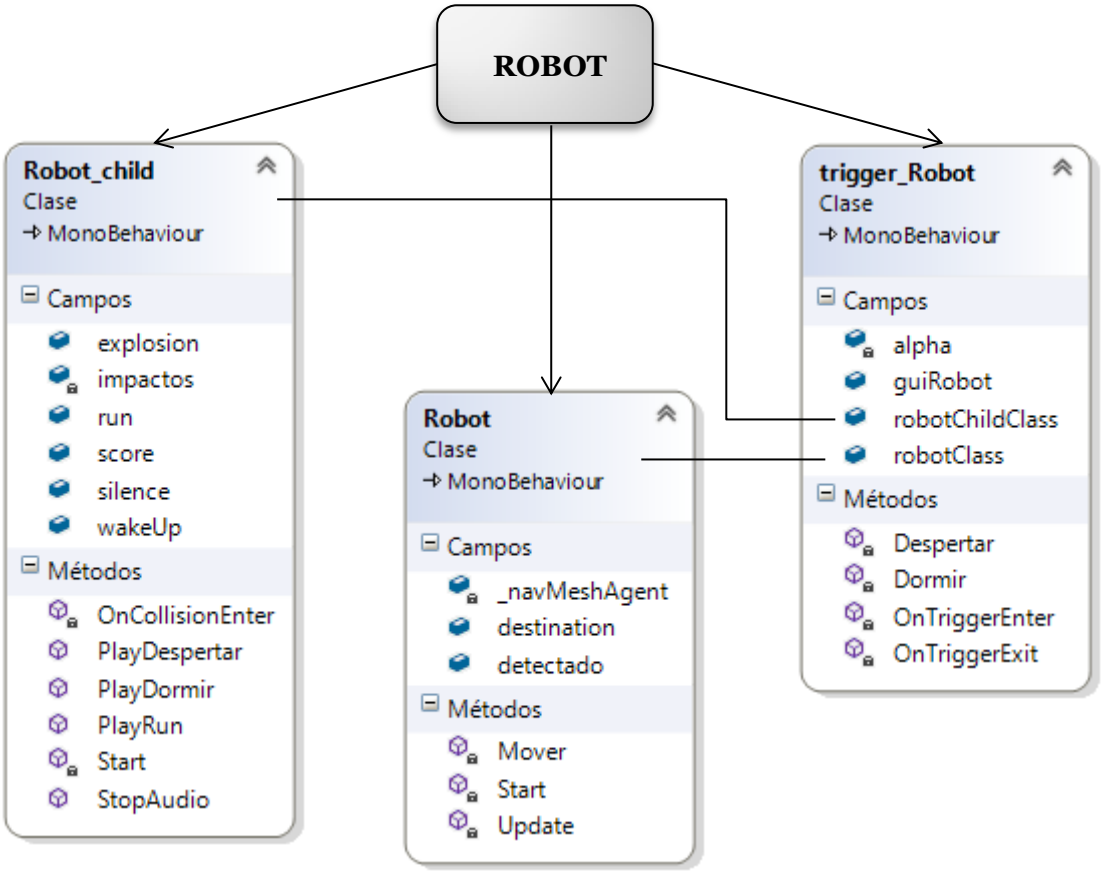
En el siguiente diagrama de herencia encontramos las diferentes agrupaciones de clases que heredan de *MonoBehaviuor*, la clase principal de Unity.

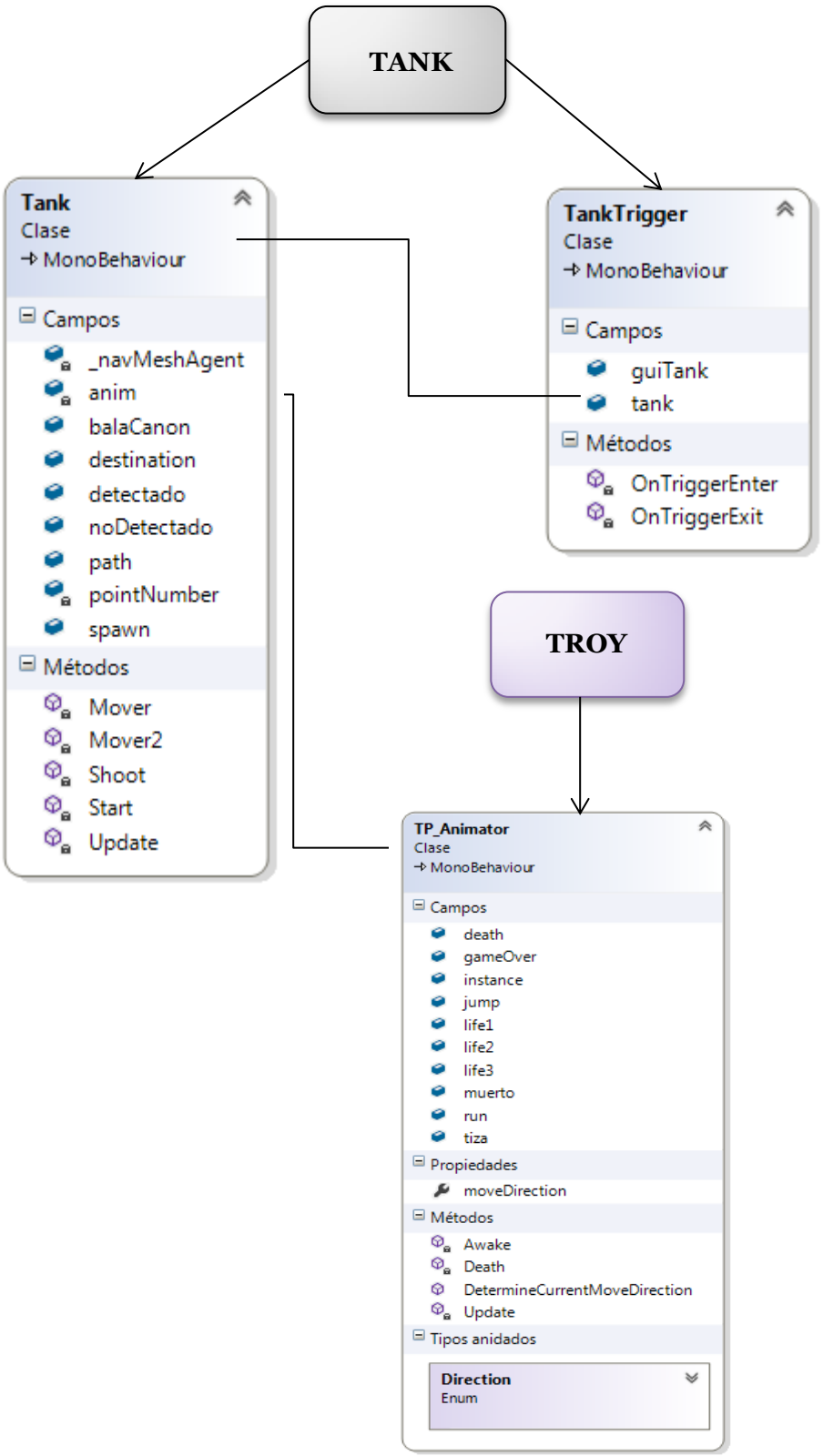


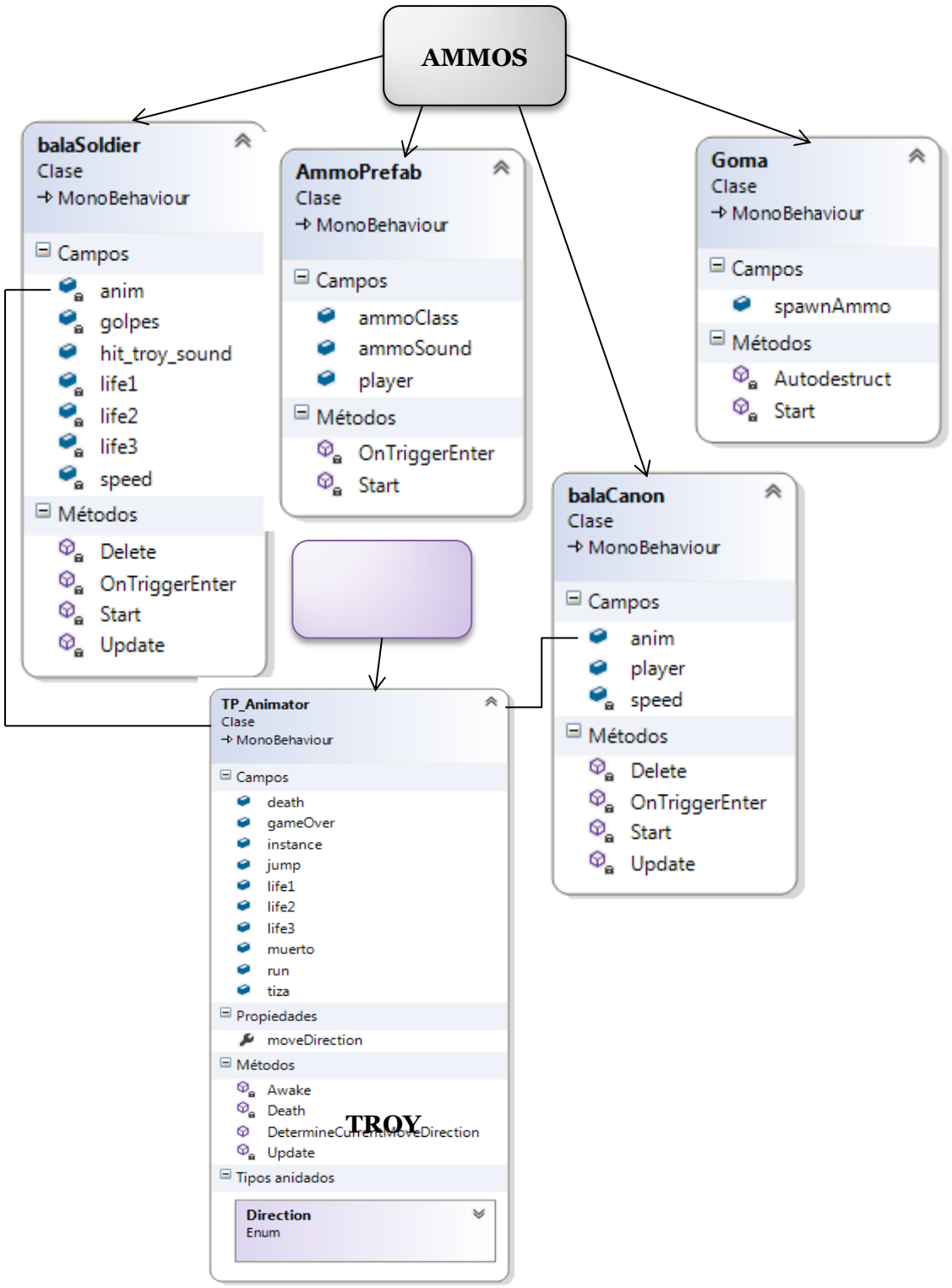
A continuación encontramos el diagrama de clases de cada una de ellas. En ellos se muestra el objeto del juego, las clases que contiene y la comunicación entre dichas clases. Esta comunicación puede darse entre las clases de un mismo objeto o entre clases de objetos diferentes.

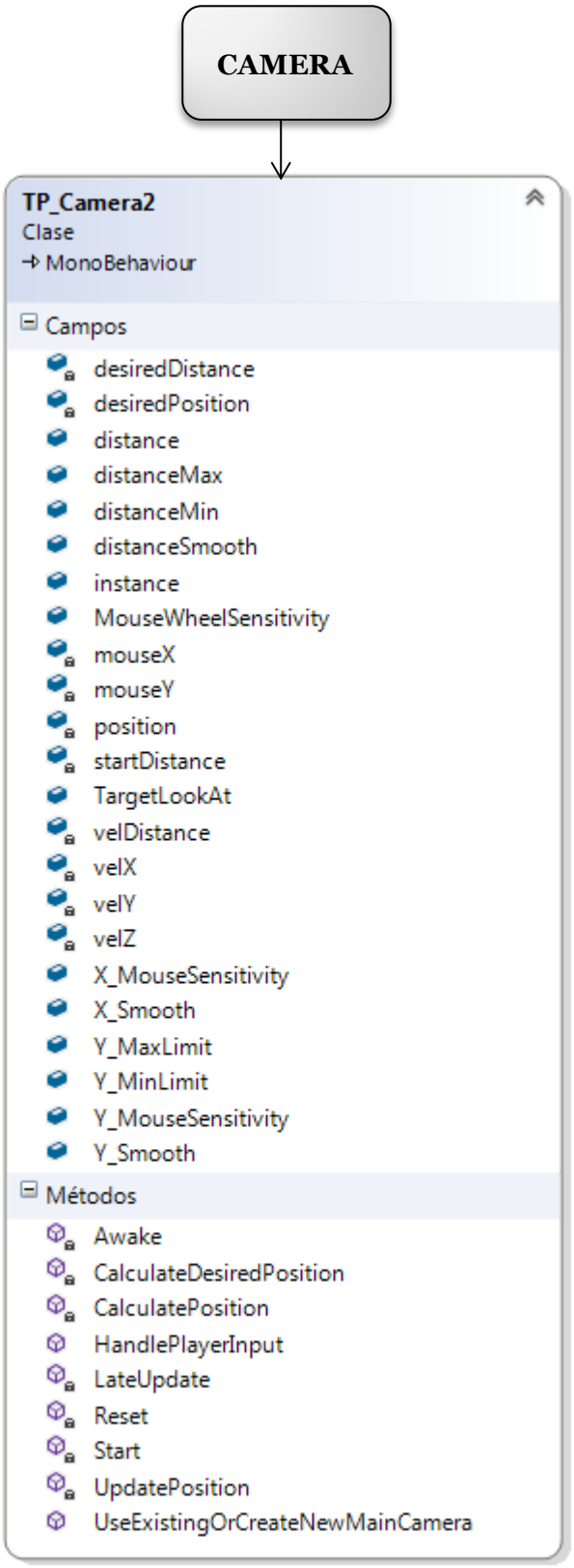

TROY
Robot_collision
Clase
MonoBehaviour
Campos
anim
explosion
Métodos
OnControllerColliderHit
Start
Objetivos
Clase
MonoBehaviour
Campos
key
magnet
magnet_audio
mensaje
mensaje_audio
objetivo
Métodos
Disable
OnTriggerEnter
Start
Matatanques
Clase
MonoBehaviour
Campos
ammoPrefab
explosion
guiTank
score
Métodos
OnControllerColliderHit
Ammo
Clase
MonoBehaviour
Campos
ammo
ammoScore
goma
lanzar
spawnAmmo
Métodos
Start
Update
TP_Controller
Clase
MonoBehaviour
Campos
characterController
instance
ok
Métodos
Awake
GetLocomotionInput
HandleActonMethod
Jump
Update
TP_Animator
Clase
MonoBehaviour
Campos
death
gameOver
instance
jump
life1
life2
life3
muerto
run
tiza
Propiedades
moveDirection
Métodos
Awake
Death
DetermineCurrentMoveDirection
Update
Tipos anidados
Direction
Enum
TP_Motor
Clase
MonoBehaviour
Campos
anim
backwardSpeed
forwardSpeed
gravity
instance
jumpSpeed
strafingSpeed
terminalVelocity
Propiedades
moveVector
verticalVelocity
Métodos
ApplyGravity
Awake
Jump
MoveSpeed
ProcessMotion
SnapAlignCharacterWithCamera
UpdateMotor


SOLDIER
Soldier_Child
Clase
MonoBehaviour
Campos
ammoPrefab
guiSoldier
score
splashSoldier
Métodos
OnCollisionEnter
PlayIdle
PlayRun
Start
Soldier
Clase
MonoBehaviour
Campos
_navMeshAgent
anim
balaSoldier
detectado
fireSoldier
ok
ok_disparar
path
pointNumber
spawnSoldier
troy
Métodos
Disparar
Mover
NoMover
Start
Update
trigger_Soldier
Clase
MonoBehaviour
Campos
guiSoldier
soldierChildClass
soldierClass
Métodos
OnTriggerEnter
OnTriggerExit
Start
TROY
TP_Animator
Clase
MonoBehaviour
Campos
death
gameOver
instance
jump
life1
life2
life3
muerto
run
tiza
Propiedades
moveDirection
Métodos
Awake
Death
DetermineCurrentMoveDirection
Update
Tipos anidados
Direction
Enum


ROBOT
Robot_child
Clase
MonoBehaviour
Campos
explosion
impactos
run
score
silence
wakeUp
Métodos
OnCollisionEnter
PlayDespertar
PlayDormir
PlayRun
Start
StopAudio
Robot
Clase
MonoBehaviour
Campos
_navMeshAgent
destination
detectado
Métodos
Mover
Start
Update
trigger_Robot
Clase
MonoBehaviour
Campos
alpha
guiRobot
robotChildClass
robotClass
Métodos
Despertar
Dormir
OnTriggerEnter
OnTriggerExit


TANK
Tank
Clase
MonoBehaviour
Campos
_navMeshAgent
anim
balaCanon
destination
detectado
noDetectado
path
pointNumber
spawn
Métodos
Mover
Mover2
Shoot
Start
Update
TankTrigger
Clase
MonoBehaviour
Campos
guiTank
tank
Métodos
OnTriggerEnter
OnTriggerExit
TROY
TP_Animator
Clase
MonoBehaviour
Campos
death
gameOver
instance
jump
life1
life2
life3
muerto
run
tiza
Propiedades
moveDirection
Métodos
Awake
Death
DetermineCurrentMoveDirection
Update
Tipos anidados
Direction
Enum


AMMOS
balaSoldier
Clase
MonoBehaviour
Campos
anim
golpes
hit_troy_sound
life1
life2
life3
speed
Métodos
Delete
OnTriggerEnter
Start
Update
AmmoPrefab
Clase
MonoBehaviour
Campos
ammoClass
ammoSound
player
Métodos
OnTriggerEnter
Start
Goma
Clase
MonoBehaviour
Campos
spawnAmmo
Métodos
Autodestruct
Start
balaCanon
Clase
MonoBehaviour
Campos
anim
player
speed
Métodos
Delete
OnTriggerEnter
Start
Update
TP_Animator
Clase
MonoBehaviour
Campos
death
gameOver
instance
jump
life1
life2
life3
muerto
run
tiza
Propiedades
moveDirection
Métodos
Awake
Death
DetermineCurrentMoveDirection
Update
Tipos anidados
Direction
Enum
TROY


CAMERA
TP_Camera2
Clase
MonoBehaviour
Campos
desiredDistance
desiredPosition
distance
distanceMax
distanceMin
distanceSmooth
instance
MouseWheelSensitivity
mouseX
mouseY
position
startDistance
TargetLookAt
velDistance
velX
velY
velZ
X_MouseSensitivity
X_Smooth
Y_MaxLimit
Y_MinLimit
Y_MouseSensitivity
Y_Smooth
Métodos
Awake
CalculateDesiredPosition
CalculatePosition
HandlePlayerInput
LateUpdate
Reset
Start
UpdatePosition
UseExistingOrCreateNewMainCamera

2.5 Metodología

Puesto que un videojuego es principalmente un producto software, se puede utilizar para su desarrollo todas las metodologías típicas de desarrollo de software.

Los diferentes tipos de metodologías se pueden englobar en dos grupos:

- **Clásicas:** Pretenden planificar el desarrollo al inicio, intentando minimizar las modificaciones durante el desarrollo del software. Ejemplo: Waterfall.
- **Ágiles:** Se adaptan a los cambios que puedan surgir durante el desarrollo. Ejemplos: eXtreme Programming (XP) y Scrum.

Cada una de las diferentes metodologías mencionadas se pueden encontrar con mayor detalle en la bibliografía de este proyecto **(METODOLOGÍAS).**

Una vez revisadas las diversas metodologías que se suelen emplear en el desarrollo de videojuegos, clásicas y ágiles, la metodología que elegí seguir es una aproximación al **desarrollo iterativo** o incremental. Esta decisión viene dictaminada por varios motivos. En primer lugar, en vez de un equipo de personas, del desarrollo de este videojuego sólo me voy a encargar yo. Esto hace que la planificación sea verdaderamente inestable, es decir que muy probablemente termine siendo un proyecto en el que los cambios y modificaciones estén a la orden del día, y así fue. Por ese motivo un proceso interactivo es el que mejor se acopla a la posibilidad de incluir dichas modificaciones. En segundo lugar, el hecho de realizar pruebas del producto cada vez que se termina una iteración es un punto que me proporciona la tranquilidad y la confianza necesarias para seguir adelante, sabiendo en todo momento que voy por buen camino.

Así pues, teniendo esta metodología en mente, procedo a explicar mi caso particular. Se parte del diseño completo de juego, especificado en el **documento de diseño,** previamente realizado. Inicialmente se divide el desarrollo del juego en iteraciones. Cada iteración añadirá al proyecto nuevas funcionalidades y características de manera incremental, hasta llegar a la versión definitiva del producto. Para cada una de las iteraciones se concreta qué nuevas cualidades aportará al videojuego, y se estima el coste de tiempo de dicha iteración. Con todo ello se puede estimar el coste temporal total del desarrollo del proyecto. Una vez realizada la división del proyecto en Iteraciones y estimado su coste total, se realiza el diseño del software del juego mediante diagramas UML.

Cuando se ha terminado el diseño se realiza la implementación de las diferentes iteraciones de forma secuencial, de forma que hasta que no se ha terminado completamente una iteración no se comienza con la siguiente. Para cada una de las iteraciones se realiza una fase de codificación y una fase de pruebas, donde se realizan tanto pruebas de software como pruebas de jugabilidad, pudiendo cambiar el diseño del juego.

2.6 Planificación

A la hora de realizar cualquier proyecto, de cualquier envergadura, el proceso de planificación del mismo supone una parte esencial. Un proyecto con una buena planificación es un proyecto con una estructura inicial sólida y precisa.

1ª iteración: escenario principal

Tiempo estimado: Dos semanas.

Comenzamos el desarrollo con la primera iteración. En ella se realizarán los siguientes ítems del videojuego:

- **Modelado y texturizado del escenario principal**, en 3D Studio Max. Dicho escenario es el interior de una habitación, la habitación del personaje principal.
- **Modelado y texturizado el personaje principal**.
- **Inclusión del escenario principal en Unity 3D**.
- **Iluminación básica.** La iluminación del juego no es parte de esta fase de la implementación. No obstante, en un proyecto 3D, antes de ponerse en serio con la iluminación final del escenario conviene tener ya en marcha algunas luces básicas para ir viendo desde el principio cómo reaccionan los materiales a dichas luces y cómo se comportan las sombras, sobre todo las sombras en movimiento.

2ª iteración: movimiento

Tiempo estimado: Dos semanas.

Teniendo ya modelado el personaje principal, es momento de dotarlo de movimiento. Esta iteración aporta lo siguiente:

- **Animaciones del personaje principal (Troy).** Animaciones realizadas:
  - *Idle.*
  - *Correr.*
  - *Salto.*
  - *Andar hacia detrás.*
  - *Lanzar proyectil.*
  - *Muerte.*
- **Inclusión del personaje principal en el escenario** (Unity 3D).
- **Sincronización de las animaciones de movimiento de Troy**.
- **Control de colisiones** de Troy con el escenario.
- **Movimiento de la cámara.** Rotación de la cámara alrededor del personaje principal y control de la distancia usando la rueda del ratón.

3ª iteración: enemigos

Tiempo estimado: Una semana.

Una vez incluido y testeado el personaje principal en el proyecto, el paso siguiente es crear los enemigos a los que nos enfrentaremos. Al igual que el personaje principal, los enemigos están modelados y texturizados en 3D Studio Max.
Tareas a realizar:

- **Modelado y texturizado del enemigo: Robot.**
- **Modelado y texturizado del enemigo: Soldado.**
- **Modelado y texturizado del enemigo: Tanque.**

4ª iteración: movimiento de los enemigos

Tiempo estimado: Dos semanas.

Antes de incluir a los enemigos en el proyecto de Unity, deben tener implementadas las animaciones que van a realizar en el juego:

- **Animaciones del enemigo: Robot.** Animaciones realizadas:
  - *Despertar* - El Robot despierta cada vez que Troy se acerca lo suficiente.
  - *Dormir* - Cuando Troy se aleja a una distancia prudente del Robot, éste se queda en un estado de descanso.
  - *Correr*.
- **Animaciones del enemigo: Soldado.** Animaciones realizadas:
  - *Desplazamiento* - Los soldados se desplazan a saltos, puesto que son soldaditos de plástico que están sujetos a una base.
- **Animaciones del enemigo: Tanque.** Animaciones realizadas:
  - *Desplazamiento*.

5ª iteración: inclusión de los enemigos en el proyecto

Tiempo estimado: Una semana.

Las características que aporta esta iteración al juego son:

- **Inclusión en el proyecto del enemigo: Robot.**
- **Inclusión en el proyecto del enemigo: Soldado.**
- **Inclusión en el proyecto del enemigo: Tanque.**
- **Configuración de las animaciones de los enemigos.** Definición de los fotogramas clave de las distintas animaciones de cada personaje, para dejarlas lo más preparadas posible de cara a manejar dichas animaciones desde código.
- **Diseño e implementación del objetivo principal.**

6ª iteración: Inteligencia Artificial

Tiempo estimado: Tres semanas.

Esta es una iteración crucial en el desarrollo del videojuego. En ella se integra la IA de los enemigos y su interacción con Troy, lo que constituye el núcleo principal del juego.

- **Inteligencia Artificial del enemigo: Robot.** PathFinding y detección del personaje principal.
- **Inteligencia Artificial del enemigo: Soldado.** PathFinding y detección del personaje principal.
- **Inteligencia Artificial del enemigo: Tanque.** PathFinding y detección del personaje principal.
- **Modelado y texturizado de los proyectiles,** con los que nos disparan algunos de los enemigos.
- **Sistema de disparos del enemigo: Soldado.**
- **Sistema de disparos del enemigo: Tanque.**
- **Control de colisiones de los proyectiles sobre Troy.** Decremento de la variable "Vida" del personaje al impactar un proyectil sobre él.
- **Control de colisiones del enemigo Robot sobre Troy.**
- **Modelado y texturizado de la munición,** de la que disponemos para lanzar a los enemigos.
- **Control de colisiones de la munición sobre los enemigos.**
- **Muerte de los enemigos.** Eliminación en tiempo real del enemigo o enemigos a los que alcanzamos con nuestra munición.
- **Muerte de Troy.** Aplicación de la animación "Muerte" cuando la variable "Vida" llega a cero.

7ª iteración: Heads-Up Display (HUD) y menú principal

Tiempo estimado: Dos semanas.

En esta iteración se incluye en el juego el menú principal y el panel de información del jugador.

- **Diseño del Heads-Up Display (HUD)**. Diseño de cada HUD de manera individual.
  - **Estado de la Vida de Troy**. Barra de vida que cambia de color conforme decrece.
  - **Munición restante**.
  - **Cantidad de enemigos restantes**. Para cada enemigo se muestra su icono y el número existente.
  - **Sistema de detección**. Si un enemigo detecta a Troy el icono del enemigo se vuelve de color rojo.
- **Posicionamiento del HUD**.
- **Diseño del menú principal.**
- **Implementación de la interacción del usuario con el menú principal.**

8ª iteración: iluminación y audio

Tiempo estimado: Una semana.

Esta iteración aporta gran parte del realismo del juego. Las luces y el audio ayudan a sumergir al jugador en el ambiente del juego.

- **Creación y ajuste de las luces que iluminarán el escenario.**
- **Creación de los sonidos necesarios para el juego.**
- **Coordinación de los elementos auditivos con la partida en tiempo real.**

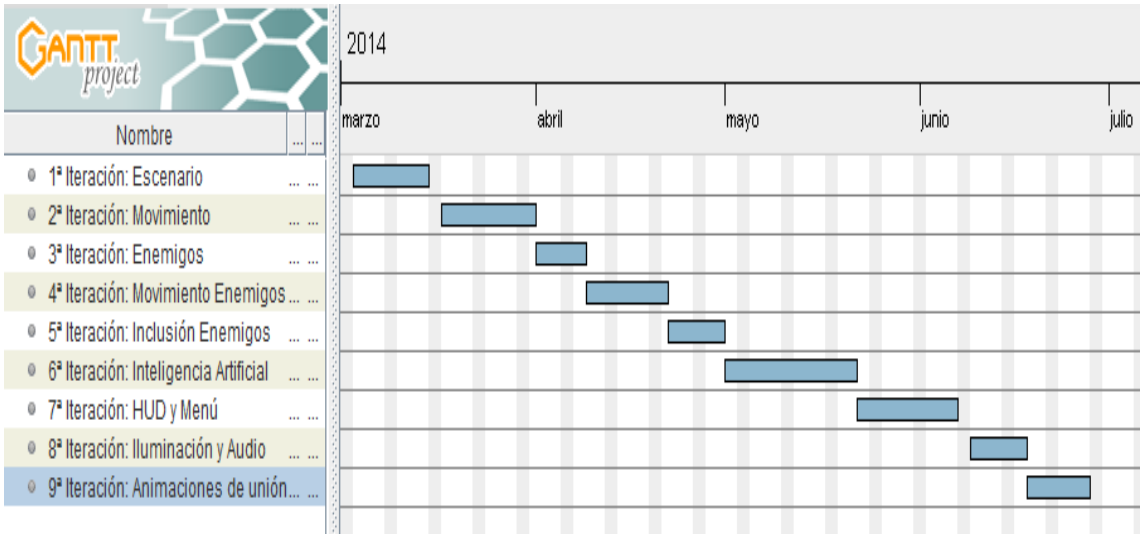
9ª iteración: animaciones de unión

Tiempo estimado: Una semana.

Últimos retoques necesarios para la fluidez del juego. Las animaciones de principio y fin funcionan como un nexo entre el inicio y el fin de la partida.

- **Animaciones de inicio de juego.**
- **Animaciones de fin de juego.**
- **Animaciones de pantalla completada.**

Diagrama de Gantt: Planificación



3. IMPLEMENTACIÓN

"*Perfection [in design] is achieved, not when there is nothing more to add, but when there is nothing left to take away*."

Antoine de Saint-Exupéry

3.1 Entendiendo Unity

Debido a la complejidad del programa, desarrollar con Unity a nivel experto es bastante complicado de conseguir. No obstante es un programa bastante intuitivo, por lo que empezar como desarrollador con proyectos sencillos es un trabajo agradecido e interesante. Además en la web de Unity hay una sección dedicada exclusivamente al aprendizaje, donde se puede encontrar tutoriales y todo lo relacionado con la documentación a utilizar, dependiendo del lenguaje elegido para programar un videojuego.

Unity trabaja principalmente con proyectos y escenas. Un proyecto es un videojuego o aplicación, y cada proyecto puede tener una o varias escenas. En un videojuego una escena sería un nivel del juego. En cada escena se puede crear objetos o importarlos desde otros programas de modelado 3D. Cada objeto de una escena es un "Objeto del Juego" (*GameObject*), siendo tratadas las luces y las cámaras también como *GameObjects*, por lo que se les puede aplicar código para hacer que se comporten a gusto del desarrollador.

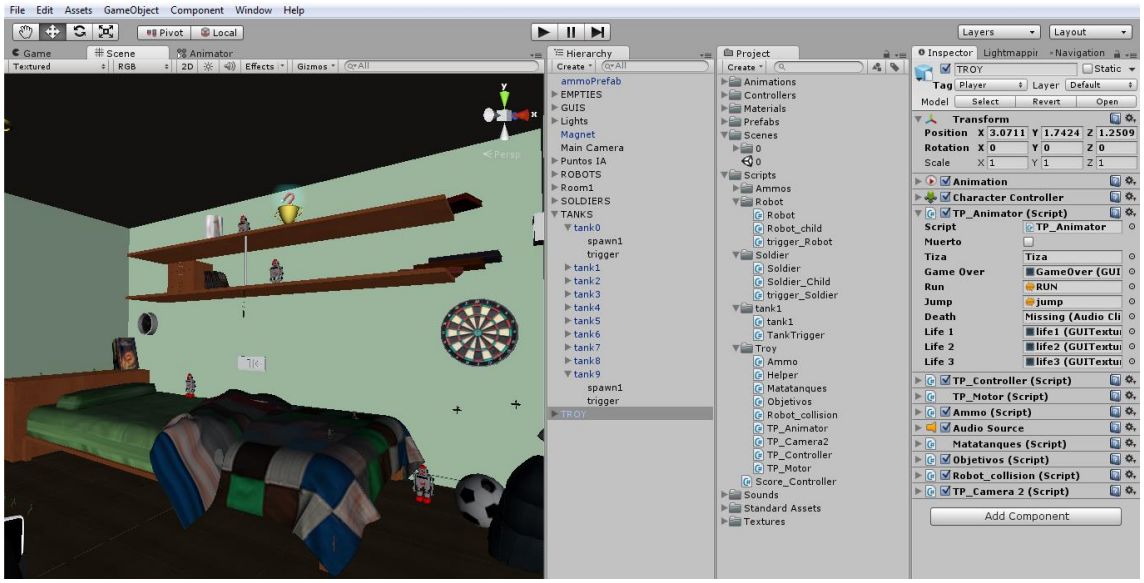


Imagen 4: Ejemplo interfaz de un proyecto en Unity

Cada *GameObject* consta de una serie de componentes, que se pueden añadir o quitar para caracterizar el objeto. Estos componentes se pueden modificar desde la interfaz de Unity, pero también se puede acceder a ellos desde código, para cambiar sus

características en tiempo de ejecución si así se requiere. Algunos de los componentes más importantes, que muchos de los *GameObjects* de un juego suelen tener son:

- **Transform**: Es el componente más básico, que todo *GameObject* debe tener. Este componente dota al *GameObject* de posición en la escena, rotación y tamaño.
- **MeshRenderer**: Este componente se encarga de renderizar un objeto en tiempo de ejecución. Por ello todo objeto que queremos que se vea en una escena debe contar con este componente.
- **Colliders** (Colisionadores): Son áreas que envuelven al *GameObject,* que son utilizadas para comprobar las colisiones entre los distintos objetos del juego. Hay varios tipos de colisionadores:
    - **BoxCollider**: Colisionador de caja. Se crea un área con forma de cubo alrededor del objeto.
    - **SphereCollider**: Colisionador de esfera. Se crea un área con forma de esfera alrededor del objeto.
    - **CapsuleCollider**: Colisionador de cápsula. Se crea un área con forma de cápsula alrededor del objeto.
    - **MeshCollider**: Colisionador de malla. El colisionador se acopla a la malla del objeto al que se le asigna. Es una forma de obtener colisiones de manera muy precisa, pero consume muchos recursos.

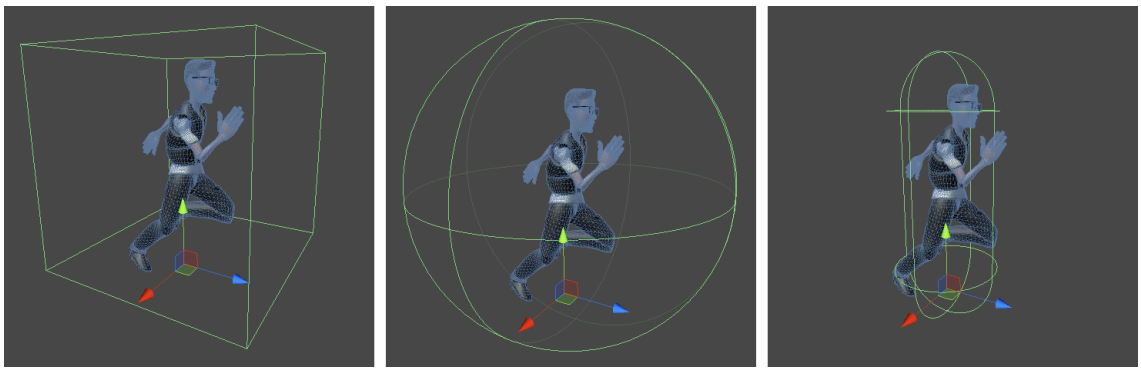


Imagen 5: *BoxCollider* *SphereCollider* *CapsuleCollider*

- **RigidBody**: Adjudica al *GameObject* algunas características que tendría un cuerpo en la realidad: masa, gravedad, fricción, etc. Es muy útil para hacer que objetos de la escena interaccionen entre sí.
- **CharacterController**: Normalmente se le añade este componente tan sólo al personaje principal, que será controlado por el jugador. *CharacterController* contiene propiedades de interacción con el entorno.
- **Animation**: Todo *GameObject* que contenga animaciones debe tener este componente añadido. En él se especifican el número de animaciones y el rango de fotogramas que ocupa cada animación. Además sirve de referencia a la hora de controlar dichas animaciones desde código.
- **AudioSource**: Cualquier *GameObject* que emita sonidos debe tener este componente asociado. Funciona como un almacén de sonidos y proporciona opciones para la reproducción de dichos sonidos.

- **AudioListener**: Este componente actúa como un oído, capta los sonidos emitidos por los objetos que tienen asociado un componente de tipo *AudioSource* y los reproduce por el altavoz del dispositivo. En una escena sólo puede haber un único objeto que contenga este componente, normalmente es la cámara principal la que lo contiene.
- **Script:** Un *Script* es una pieza de código añadida como componente a un *GameObject*. Están explicados a continuación.

La manera que tiene Unity 3D de coordinar el código del videojuego con los modelos es mediante componentes de tipo *Script*. Un Script es una pieza de código que puede asignársele a uno o varios *GameObjects* como un componente más. Dichos *Scripts* pueden estar escritos en: C#, JavaScript o Boo. Depende del desarrollador elegir un lenguaje de programación u otro. También es posible combinar varios lenguajes de programación, es decir, se puede crear dos Scripts en lenguajes distintos y asignárselos a un mismo *GameObject*, los dos funcionarán.

Unity 3D proporciona un amplio conjunto de librerías de clases específicas y funciones principales para el desarrollo de juegos por lo que muchas de las tareas comunes en el desarrollo de videojuegos ya están implementadas. Estas funciones o métodos heredan de la clase principal de Unity: ***MonoBehaviour***. La herencia de *MonoBehaviuor* en *JavaScript* está implícita, mientras que en *C#* y *BOO* se tiene que especificar para cada clase.

Estas son algunas de los métodos más importantes presentes en las clases que heredan de *MonoBehaviour*:

- **Start()**: A este método se le llama sólo al inicio del juego. Es muy útil para inicializar variables, posiciones, estados, etc.
- **Awake()**: Este método es llamado justo un fotograma antes que Start(), por lo que también es práctico para inicializar variables, sobretodo variables que puedan tener conflictos de tiempo con Start().
- **Update()**: Este método es llamado una vez por cada fotograma del juego, en tiempo de ejecución. Dentro de él se debe incluir todo el código que necesita actualizarse una vez por fotograma, como el movimiento o la detección de cambios. Por ello es el más importante a la hora de desarrollar el videojuego.
- **OnCollisionEnter()**: Es quizá uno de los métodos predefinidos más útiles. A este método se le llama cada vez que el objeto que lleva este *Script* colisiona con otro objeto de la escena.
- **OnTriggerEnter()**: Un componente de tipo *Collider* puede ser marcado como *Trigger*. Esto significa que los objetos que colisionen con él no reaccionarán como si de una colisión real se tratase, sino que pasarán de largo. No obstante quedará registrado que un objeto ha entrado en ese área definida por el colisionador. Este método detecta esa intrusión en dicha área. Es muy útil para crear sistemas de detección.

Otra funcionalidad de Unity (*MonoBehaviour*) a tener en cuenta son las **Corrutinas.** Éstas permiten reorientar el flujo de código, generalmente para retrasar la ejecución de una parte del código durante un periodo determinado de tiempo. Esta característica es muy práctica para aquellas acciones del juego en las que debe haber un lapso de tiempo

entre ellas, por ejemplo en disparos de los enemigos, autodestrucción de proyectiles, etc. Un ejemplo de autodestrucción, utilizado en el proyecto, sería el siguiente:

```
1       ...
2       void Start ()
3       {
4          StartCoroutine (Autodestruct ());
5       }
6
7       IEnumerator Autodestruct()
8       {
9         yield return new WaitForSeconds(2);
10        Destroy (this.gameObject);
11      }
12      ...
```

El método *Start()* llama a la corrutina *Autodestruct* (linea 4). La corrutina detiene la ejecución del código por dos segundos (línea 9) y destruye el objeto (línea 10).

3.2 De 3D Studio Max a Unity 3D

Exportar modelos desde un programa para utilizarlos en otro es algo muy común en el mundo del diseño 3D. La mayoría de programas tienen su propia extensión para los archivos que se exportan Aunque suele haber mucha flexibilidad a la hora de que un programa acepte formatos de otros programas, también existen formatos estándar de exportación 3D. En mi caso, al exportar desde 3D Studio Max, para importar luego a Unity3D, he utilizado un formato estándar conocido como "***Filmbox***" (.fbx), que soporta exportaciones de modelos animados y modelos sin animación.

Al importar un modelo 3D de un programa a otro es bastante común que surjan errores, sobre todo si los modelos están animados. Estos errores de deben a las diferencias entre las configuraciones generales de los distintos programas que intervienen en el proceso de exportación-importación. Por ejemplo pueden surgir errores de ejes invertidos. La mayoría de programas utilizan la coordenada "Y" para el eje vertical, mientras que 3D Studio Max utiliza la "Z". También pueden surgir errores en los fotogramas clave de los modelos con animaciones. Todos estos pequeños detalles se deben tener en cuenta en las opciones de exportación y, si no se pueden solventar con eso, toca aplicar un poco de ingenio para que todas las piezas caigan en el lugar adecuado.

3.3 Animaciones

Las animaciones de los personajes son esenciales para un videojuego. Aportan un grado de realismo necesario para la credibilidad del mismo.

Dotar de movimiento a un modelo 3D no es nada fácil. Esta tarea suele ser muy parecida para todos los programas de modelado 3D. Primero se crea un esqueleto, que es un conjunto de modelos 3D conectados entre sí y que siguen una jerarquía de movimiento. Luego, con mucha precisión, hay que superponer dicho esqueleto al modelo del personaje, haciendo que todos los huesos caigan en la posición correcta, como si de huesos de verdad se tratase. Esta tarea es la más costosa y requiere mucho tiempo y concentración. Por último se realiza lo que se denomina el **pesado de vértices**: mediante varias herramientas se le asigna a cada vértice de la malla del personaje un porcentaje proporcional por cada hueso que queremos que "tire" de ese vértice. Por ejemplo, un mismo hueso puede mover vértices del brazo y del antebrazo de manera proporcional, justo como ocurriría con el brazo de una persona real.

A todo este proceso se le llama comúnmente "Rigg". Una vez tenemos al personaje "Riggeado" es hora de animarlo. Para ello se animan los huesos del esqueleto y, si hemos realizado bien el "Rigg", la malla del personaje debería seguir la animación de los huesos de manera suave y precisa.

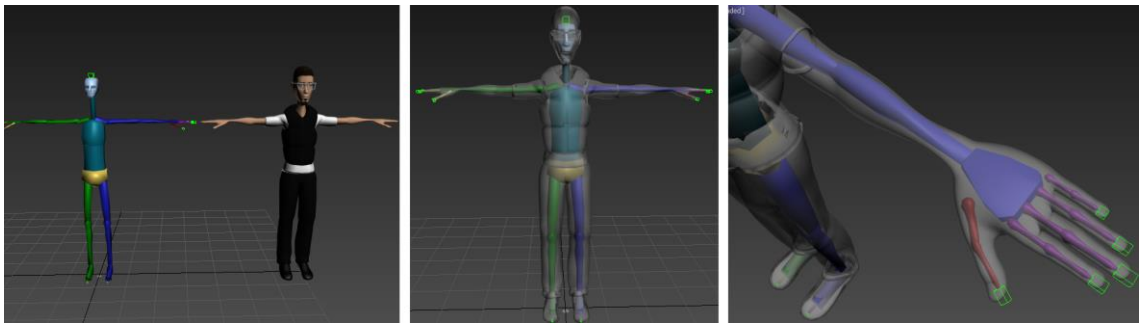


Imagen 6: Rigg de Troy

De las animaciones de los personajes sólo se realiza un "Loop", utilizando para ello los fotogramas clave. Cuando animamos el personaje debemos hacer que en el último fotograma, el personaje esté exactamente en la misma posición que en el primero. De esta forma, en el juego, se pueden repetir una y otra vez estas animaciones para crear una sensación de movimiento fluido.

3.4 Inteligencia Artificial (IA)

La Inteligencia Artificial de los distintos elementos de un videojuego es siempre lo más difícil de programar por razones obvias. Dotar a un personaje virtual de un comportamiento que pueda parecer inteligente es un trabajo laborioso y requiere unos conocimientos de programación y matemáticas bastante avanzados.

Por suerte Unity 3D incorpora una herramienta que facilita bastante este trabajo, el ***NavMesh.*** NavMesh es un componente que se le puede asignar a cada objeto del juego. Una vez asignado, se le pasa al objeto (en nuestro caso, el enemigo) una lista de posiciones ordenadas a gusto del programador. El *GameObject* empezará a dirigirse de

manera ordenada a cada una de las posiciones indicadas, rodeando los obstáculos que encuentre, y que hayamos marcado previamente como obstáculos.

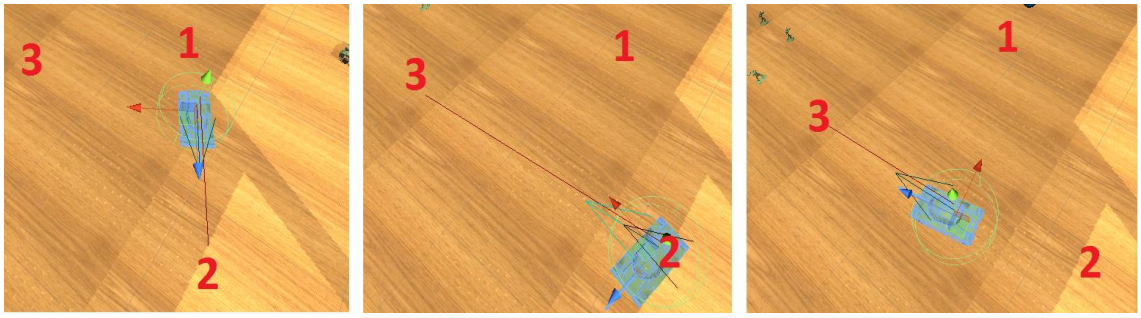


Imagen 7: NavMesh, redirección hacia la siguiente posición de la lista

NavMesh contiene una serie de parámetros que debemos ajustar para lograr el comportamiento deseado. Por ejemplo cada *GameObject* que contiene un componente *NavMesh* ha de tener asignado un número de prioridad. De esa manera si dos o más de esos objetos se cruzan decidirán, basándose en ese parámetro de prioridad, cuál de ellos evita primero la colisión y rodea a los demás.

En mi caso todos los enemigos contienen un componente *NavMesh,* que es utilizado para recorrer un camino predeterminado mientras no detecten al jugador. Una vez el jugador se acerca lo suficiente a un enemigo, éste lo detecta y se realiza un cambio de paradigma en el código. El enemigo entonces deja de realizar su camino predeterminado y pasa a comportarse de una de las siguientes maneras, dependiendo de qué tipo de enemigo se trate:

- **Tanque**: En el caso del tanque se realiza un cambio en el código que hace que el tanque siga al personaje principal, disparándole un proyectil cada segundo. Esto se consigue mediante la sustitución de la lista de puntos a los que dirigirse por una única posición, la del personaje. Puesto que el código correspondiente al *NavMesh* se incluye dentro del *Update(),* a cada fotograma se realiza una actualización de la posición actual del personaje, por lo que el tanque siempre sigue a Troy vaya donde vaya. Si troy se sale del rango de alcance se vuelve al paradigma original y el tanque vuelve a recorrer su camino predeterminado.
- **Soldado**: Si es un soldado el que nos detecta, éste deja de recorrer su camino para quedarse quieto, disparando desde la lejanía. En este caso lo único que se actualiza a cada fotograma es la orientación del soldado, que siempre enfocará al personaje principal. Si Troy sale del rango establecido el soldado vuelve a recorrer su camino.
- **Robot**: Idéntico al tanque, pero sin disparos.

A continuación adjunto la parte del código del tanque que se encarga del cambio de comportamiento del mismo dependiendo de si se detecta o no al personaje principal:

```
1       ...
2       void Update ()                  // se ejecuta a cada fotograma
3       {
```

```
4          if (detectado == false)              // si no se ha detectado a Troy
5          {
6            noDetectado = true;
7            if (_navMeshAgent.remainingDistance == 0)      //  LISTA DE POSICIONES
8            {
9                    pointNumber++;                          // LISTA DE POSICIONES
10                   if (pointNumber >= path.Length)         // LISTA DE POSICIONES
11                   {
12                     pointNumber = 0;                      // LISTA DE POSICIONES
13                   }
14                 Mover ();                  // se ejecuta Mover()
15           }
16         }
17
18         if (detectado)                     // si Troy es detectado
19         {
20           Mover2 ();                       // se ejecuta Mover2()
21         }
22       }//end Update
23
24       void Mover() // Mover()  fija un nuevo destino de la lista de posiciones
25       {
26          _navMeshAgent.SetDestination(path[pointNumber].position);
27       }
28
29       void Mover2() //Mover2() fija como destino la posición de Troy, que se actualiza a cada fotograma
30       {
31          _navMeshAgent.SetDestination(destination.position);
32       }
33       ...
```

3.5 GameObjects

GameObject: Troy

El *GameObject* más importante del juego es Troy. Es el personaje principal, y contiene un número considerable de componentes y clases (*Scripts*) que definen su comportamiento. Sus componentes más importantes son:

- **Transform**: Posición, rotación y tamaño de Troy.
- **MeshRenderer**: Se encarga de renderizar la malla.
- **Animation**: Contiene las animaciones del personaje.
- **Character Controller**: Necesario para el personaje principal.
- **Audio Source**: Control de los sonidos emitidos por el personaje.

Troy contiene un total de 7 clases para su manejo e interacción con el entorno:

- **Ammo()**: Controla la munición disponible. Se encarga de decrementar la variable munición cada vez que se dispara, así como de incrementarla cuando el personaje colisiona con el *GameObject* que representa la munición.
- **Matatanques()**: Se encarga de destruir un tanque si Troy colisiona con él desde arriba, es decir, si saltamos sobre un tanque.
- **Objetivos()**: Gestiona la interacción con el objetivo principal del escenario.

- **RobotCollision()**: Si colisionamos con un Robot Troy muere. Esta clase se encarga de poner a *true* la variable *muerte* de Troy si éste colisiona con el Robot.
- **TP_Animator()**: Es la clase encargada de gestionar las animaciones de Troy. Cada vez que corremos, saltamos, aterrizamos, disparamos o morimos se llama a esta clase y ella elige la animación que se debe ejecutar.
- **TP_Controller():** Controla la entrada por teclado (*Input*) y envía señales a TP_Motor().
- **TP_Motor():** Recibe los mensajes de TP_Controller() y aplica el movimiento necesario al personaje dependiendo de dichos mensajes.

GameObject: Tank

El escenario principal consta de diez tanques. Cuando un tanque detecta a Troy comienza a perseguirlo a la vez que dispara. Si un proyectil alcanza a Troy éste muere y se detiene el juego.

El *GameObject Tank* contiene los siguientes componentes:

- **Transform**: Posición, rotación y tamaño del tanque.
- **MeshRenderer**: Se encarga de renderizar la malla.
- **Animation**: Contiene sus animaciones.
- **BoxCollider**: Necesario para realizar las colisiones con el resto de objetos.
- **NavMeshAgent**: Control de la IA del tanque.
- **AudioSource:** Control de los sonidos emitidos por el tanque.

Tank contiene un total de 2 clases para su interacción con el entorno:

- **Tank()**: Es la clase encargada de la IA del tanque. Gestiona su movimiento autónomo y los ataques que realiza sobre Troy.
- **Tank_Trigger()**: Controla el área dentro de la cual Troy es detectado. Se comunica con la clase Tank() para cambiar el paradigma del comportamiento del tanque según se detecte al personaje o no.

GameObject: Soldier

En el escenario principal hay un total de doce soldados. Cuando un soldado nos detecta se detiene y comienza a disparar. Si un proyectil nos alcanza la variable vida se decrementa, hasta llegar a cero, en cuyo caso Troy muere.

El *GameObject Soldier* contiene los siguientes componentes:

- **Transform**: Posición, rotación y tamaño del soldado.
- **MeshRenderer**: Se encarga de renderizar la malla.
- **Animation**: Contiene sus animaciones.
- **BoxCollider**: Necesario para realizar las colisiones con el resto de objetos.

- **NavMeshAgent**: Control de la IA del soldado.
- **Audio Source**: Control de los sonidos emitidos por el soldado.

Soldier contiene un total de 3 clases para su interacción con el entorno:

- **Soldier()**: Es la clase encargada de la IA del soldado. Gestiona su movimiento y los ataques que realiza sobre Troy.
- **Soldier_Child**: Se encarga de gestionar las animaciones de los soldados.
- **Trigger_Soldier()**: Controla el área dentro de la cual Troy es detectado. Se comunica con la clase Soldier() para cambiar el comportamiento del soldado.

GameObject: Robot

Hay un total de seis robots en el escenario principal. Cada robot es capaz de detectarnos y seguirnos a una gran velocidad. Si un robot colisiona con el personaje principal éste muere y termina la partida.

El *GameObject Robot* contiene los siguientes componentes:

- **Transform**: Posición, rotación y tamaño del robot.
- **MeshRenderer**: Se encarga de renderizar la malla.
- **Animation**: Contiene sus animaciones.
- **BoxCollider**: Necesario para realizar las colisiones con el resto de objetos.
- **NavMeshAgent**: Control de la IA del robot.
- **Audio Source**: Control de los sonidos emitidos por el robot.

Robot contiene un total de 3 clases para su interacción con el entorno:

- **Robot()**: Es la clase encargada de la IA del robot. Gestiona su movimiento y los ataques que realiza sobre el personaje.
- **Robot_Child**: Se encarga de gestionar las animaciones de los robots.
- **Trigger_Robot()**: Controla el área dentro de la cual Troy es detectado. Se comunica con la clase Robot() para cambiar el comportamiento del robot según se detecte al personaje o no.

GameObject: Habitación (Room)

El escenario principal es la habitación del personaje Troy. La habitación contiene muchos objetos que definen el nivel y marcan el camino a seguir para completarlo. Estos objetos: ordenador, pantallas, mesas, sillas, cajones, estanterías, libros, cama, balón de fútbol, monopatín, diana, dardos, reloj de pared, enchufes, papelera, etc., permiten a nuestro personaje moverse, saltar, esquivar ataques de los enemigos y contraatacar.

Imagen 8: Habitación

El objetivo principal del nivel es escapar de la habitación. Para ello debemos llegar a la puerta principal, no sin antes haber recogido un imán que nos permitirá atraer el pomo de la puerta para abrirla. Dicho imán se encuentra en lo alto de la más alta estantería. Para acceder a esta estantería debemos cruzar la habitación varias veces, subiendo cada vez más y enfrentándonos a innumerables enemigos.

4. RT-DESK

4.1 Introducción a RT-DESK

RT-DESK (Real Time Discrete Event Simulation Kernel) es un núcleo de simulación de aplicaciones gráficas en tiempo real que sigue las directrices del paradigma discreto desacoplado. Es una librería desarrollada en C++ diseñada para la gestión temporal de eventos discretos en tiempo real.
Es una herramienta enfocada tanto para el desarrollo de aplicaciones en tiempo real como pueden ser videojuegos, simuladores de entrenamiento, realidad virtual o realidad aumentada, como para modelar sistemas que estén descritos como una colección masiva de objetos simples que interactúen unos con otros como pueden ser la simulación de partículas o los autómatas celulares.
RT-DESK compone el núcleo principal de un determinado software gráfico, y maneja su funcionamiento desde su misma estructura interna, siguiendo las bases del paradigma discreto desacoplado. Esto quiere decir que RT-DESK no es un software que funcione como una aplicación independiente, sino que RT-DESK se integra en una determinada aplicación gráfica y toma el control de la gestión de sus recursos.
RT-DESK gestiona los recursos de una aplicación mediante eventos discretos, es decir, maneja los eventos de una aplicación de manera que los recursos el tiempo de CPU sólo se otorga a aquellos procesos que realmente lo necesitan en ese preciso instante. Cada aspecto de cada objeto de la aplicación solo se ejecuta cuando es necesario. De esta manera se logra una gran optimización y una distribución eficaz de los recursos de la aplicación.
RT-DESK se proporciona como un API desarrollado en C++ en forma de una colección de ficheros cabecera y un fichero .dll que se pueden incorporar a cualquier aplicación gráfica en tiempo real o de simulación. El código fuente no puede ser accedido por parte de los programadores externos, aunque si pueden acceder a su funcionalidad.

4.2 Funcionamiento

RT-DESK utiliza para su funcionamiento dos entidades básicas: los objetos y los mensajes. Los objetos se envían mensajes, que pueden ser en tiempo real o con un tiempo asociado de espera. RT-DESK mantiene los eventos ordenados dependiendo del tiempo en que deben ser entregados. De esta forma RT-DESK controla a los objetos de la aplicación mediante el paso de mensajes.

El núcleo de RT-DESK es el Dispatcher, que se encarga de gestionar el envío y la recepción de mensajes. También mantiene actualizada la cola de mensajes pendientes y mantiene su orden temporal. Todos los elementos de la aplicación objetos independientes que se ejecutan según sus propias necesidades. Esto incluye también al

proceso de renderizado, a todos los elementos del mundo virtual y a la entrada de usuario.

RT-DESK ofrece todas las ventajas de la simulación discreta pero además sincroniza la simulación con el tiempo real durante su ejecución. Todos los elementos de la aplicación se consideran objetos y se tratan por igual, incluidos los elementos especiales como el render visual, eventos de usuario, audio etc. Cada aspecto de cada objeto de la aplicación solo se ejecuta cuando es necesario. De esta forma no se necesitan bucles que estén evaluando constantemente y por ello se evitan sobremuestreos y submuestreos. Además cada aspecto de cada objeto de la aplicación puede ajustar su frecuencia de ejecución dinámicamente para cumplir con su calidad de servicio, por lo tanto la potencia de cálculo se reparte de forma muy efectiva entre los diferentes aspectos de la aplicación.

Debido a la simulación discreta todos los eventos se ejecutan en base al tiempo de simulación, por lo que se puede alcanzar una alta precisión en el instante de ejecución de cada uno de ellos sin perder la sincronización con el tiempo real.

Cada vez que el Dispatcher termina de enviar todos los eventos necesarios finaliza su ejecución y devuelve el tiempo que debe transcurrir hasta su próxima llamada, pudiéndose realizar cualquier actividad de la que se conozca su tiempo de ejecución y liberando a la CPU de ciclos de ejecución que no son necesarios.

4.3 Acople RT-DESK al PFC

RT-DESK funciona sustituyendo al bucle principal del videojuego para realizar una gestión discreta de los eventos del mismo. Así pues la primera tarea debería ser sustituir este bucle por las clases de RT-DESK. Sin embargo aquí empiezan a surgir los problemas. Por un lado Unity proporciona este bucle principal a través de herencia de su librería base, *MonoBehaviour*, que proporciona los métodos necesarios para crear el videojuego (*Start(), Update(),* etc). Esto significa que este bucle principal que gestiona el videojuego está oculto al desarrollador, lo que complica bastante la tarea. Por otro lado Unity tiene una manera bastante peculiar de trabajar, como ya he comentado Unity trabaja mediante *Scripts,* o piezas de código, asignadas a cada *GameObject* en cada escenario del juego. Esto quiere decir que, a diferencia de otras herramientas (XNA, etc), Unity no tiene un punto de entrada claro. Por ejemplo, para un programa en C#, XNA cuenta con la clase Program.cs que se encarga de crear la clase del programa y el método main, que es el punto de entrada al videojuego. Sin embargo, en Unity cada *Script* hereda de *MonoBehaviour*, por lo que no se puede sustituir toda la clase principal de juego por RT-DESK. Ello plantea un serio problema, ya que para acoplar RT-DESK a un proyecto en Unity se debe hacer por partes. Primero es necesario crear una librería parecida a *MonoBehaviour (*en mi caso esta librería se llama *MyBehaviour)* que sobrescriba todos aquellos métodos que vayan a ser utilizados por RT-DESK, dejando intactos los restantes. Luego debemos hacer que cada clase que vaya a ser controlada por RT-DESK herede de *MyBehaviour* en vez de heredar de *Monobehaviour*.

El primer paso ya trae consigo una serie de problemas. Para empezar, *MonoBehaviour* debería incluir entre sus líneas de código los métodos que se deben sobrescribir para crear *MyBehaviour*. Esto no ocurre así, *Monobehaviour* hereda de Behaviour, que a su vez hereda de *Component*, que a su vez hereda de *Object*. Ninguna de estas clases contiene los métodos a sobrescribir (*Awake(), Start(), Update(),* etc.).

Lo que Unity hace es comprobar si nuestros *Scripts* contienen alguno de estos métodos y los llama desde el motor principal de Unity cuando lo necesite. Unity utiliza aquí lo que en programación se conoce como Reflexión (*Reflection),* de modo que se salta el proceso normal de herencia que permite dar o denegar el acceso a estos métodos a las clases del videojuego. Esto no complica demasiado las cosas si recordamos en todo momento que el motor de Unity es el único que va a llamar a estas funciones de manera inapropiada, por lo que podemos establecer nosotras dichas llamadas de manera manual sobrescribiendo aquellos métodos declarados como virtuales en nuestra clase puente, *MyBehaviour*. De esa forma deberíamos ser capaces de tomar el control del hilo de ejecución. He aquí un ejemplo de sobrescritura de la clase *Update()* de *MonoBehaviour*:

```
1	public class MyBehaviour : MonoBehaviour
2	{
3		public virtual void Update()
4		{
5
6		}
7	}
```

```
1	public class Player : MyBehaviour
2	{
3		public override  void Update()
4		{
5
6		}
7	}
```

En este ejemplo la clase *Player* sobrescribe el método *Update()* de *MyBehaviour*, que cuenta con la funcionalidad proporcionada por *MonoBehaviour*.

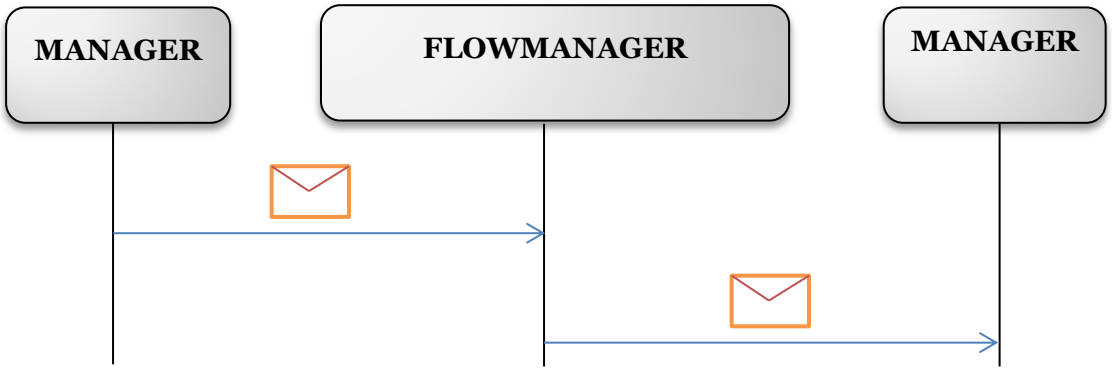
Con todo esto claro procedí a implementar dichas clases. En principio la idea era relativamente sencilla: sobrescribir las funciones principales (principalmente *Update()* y *Render()*) en la clase *MyBehaviour* para luego utilizarlas en las demás clases que fueran a ser controladas por RT-DESK. Sin embargo este procedimiento me llevó a toparme con un problema mucho mayor que los anteriores, que a su vez me condujo hacia otro más. Estos dos problemas juntos, explicados en el siguiente apartado, son la causa de la conclusión final: RT-DESK y Unity tienen serios problemas de compatibilidad que convierten la tarea de realizar un juego en Unity utilizando RT-DESK en una tarea, de momento, imposible.

4.4 Los dos grandes problemas

La línea de acción del apartado anterior sería la correcta si queremos sustituir todo el núcleo principal del juego por RT-DESK que, a priori, sería lo adecuado. Sin embargo aquí nos encontramos con el primero de los dos grandes problemas:

La función *Render()* de Unity está oculta al desarrollador. En Unity, a diferencia de otros motores, no se permite el acceso al *Render* del juego. El hecho de que el juego se *renderice* a cada fotograma es una característica intrínseca del motor a la que no se puede acceder por código. Por este motivo abandoné la idea de gestionar la totalidad del código del videojuego mediante RT-DESK, sino que ahora RT-DESK se encargaría de controlar tan solo algunas de las clases del juego, pudiendo así realizar las pruebas pertinentes con algunos de los métodos de dichas clases, por ejemplo el método *Update()*. El método *Update()* sería el candidato idóneo para las pruebas puesto que se ejecuta a cada fotograma del videojuego y por lo tanto consume muchos recursos.

Para la implementación mediante RT-DESK del método *Update()* de una clase seguí el siguiente esquema:



- **Clase Manager**

La clase *Manager* es un ejemplo de clase que envía y recibe mensajes. Puesto que esta clase hereda de MonoBehavior, no puede heredar de otras clases, por lo que implementa una clase interna (*Receiver*) que hereda de *ManagedEntity*. *ManagedEntity* es una clase abstracta que contiene el método *ReceiveMessage*, el cual es el que recibe los mensajes. Así pues la clase *Receiver* implementa la clase abstracta *ManagedEntity* y sobrescribe o da implementación al método *ReceiveMessage* del siguiente modo:

```
public override void ReceiveMessage(ManagedMsg Msg);
```

La clase Manager envía y recibe los mensajes, de esta manera implementamos el método *Update()*, haciendo que el tiempo de envío y recepción de mensajes sea el adecuado para el correcto desarrollo del juego en tiempo de ejecución.

- **Clase FlowManager:**

La clase *FlowManager* se encarga de distribuir los mensajes. En ella está implementada la inicialización de RT-DESK, que ocurre nada más ejecutar el videojuego. Esta clase es un objeto de tipo *singleton*, el cual tiene la característica de que se tiene una instancia estática a sí mismo y se inicializa una única vez. Dicha instancia es la que se devuelve, por lo que siempre se devolverá la misma instancia a todos aquellos objetos que la precisen. Como se inicializa una única vez, el *ManagedEngine* se inicializa en el constructor del *FlowManager*.

A partir del método *getInstance()* (método estático) podemos obtener la instancia del *FlowManager*. Para llamar a dicho método simplemente se ejecuta el comando FlowManager.getInstance(). Este método, para cumplir con la condición de que se inicialice la instancia una única vez, se implementa del siguiente modo:

```
public static FlowManager getInstance()
{
    if(instance == null)
    instance = new FlowManager();
    return instance;
}
```

Del mismo modo, con el método getEngine() obtenemos una referencia al *ManagedEngine*. Por ejemplo, si necesitamos obtener una referencia a ManagedEngine desde otro objeto, simplemente indicaremos la línea:

```
FlowManager.getInstance().getEngine().
```

Con estas dos clases implementadas sólo quedaba añadirlas a dos objetos del juego y probarlas. Para ello creé una escena aparte con tres objetos: un plano, un cubo y un objeto vacío. El cubo sería el personaje principal que se movería por el plano. Para ello tan sólo había que incluir al cubo el script que contiene la clase *Manager,* que simularía el método *Update()* con RT-DESK, con las líneas de código necesarias para el movimiento del cubo con entrada por teclado. A su vez el objeto vacío se encargaría de albergar la clase *FlowManager,* para la inicialización de RT-DESK.

Teniendo la escena lista para compilar, con todos los elementos en su sitio, es aquí donde aparece el segundo gran problema:

Cuando compilamos el proyecto aparece el siguiente error:

Unhandled Exception: System.TypeLoadException: Could not load type 'System.Security.SecurityRulesAttribute' from assembly 'RTDESK_NET'.

Al parecer la librería RT-DESK hace referencia a alguna librería no compatible con Unity. Tras investigar sobre el tema parece ser un error de compatibilidad con el framework de .NET. Según el siguiente enlace:

http://answers.unity3d.com/questions/23450/mono-and-missing-types-using-external-library.html

se debe probar diferentes configuraciones para el framework pero no se asegura la compatibilidad de Unity con dichas librerías. De hecho para el caso que nos concierne he probado todas las opciones que tenía a mi disposición, sin ningún éxito (Mono / .NET 4.5, Mono / .NET 4.0, .NET Framework 4 Client Profile, Mono / .NET 3.5, .NET Framework 3.5 Client Profile, Mono / .NET 3.0 y Mono / .NET 2.0). En teoría, con la última versión de Mono debería ser compatible, sin embargo, tras ejecutar varias pruebas seguía obteniendo el mismo resultado.

5. CONCLUSIONES

A lo largo de este proyecto se ha implementado desde cero la demo de un videojuego de plataformas en 3D. Para ello se ha ido pasando por las diferentes fases de desarrollo del software. Este proyecto ha supuesto una gran experiencia de aprendizaje en el desarrollo de un software tan complejo como puede ser un videojuego, y supone también un paso al frente de cara a trabajar en la industria de los videojuegos y el desarrollo de software. El desarrollo de este proyecto ha traído consigo una serie de inconvenientes u obstáculos que vale la pena remarcar. El primero y más obvio es el nivel de dificultad. Realizar un videojuego no es tarea fácil, requiere un grado de abstracción muy serio.

Otro aspecto a resaltar es la dificultad añadida de que todo el trabajo en la realización del videojuego recaiga sobre una sola persona. Dicha persona debe cubrir todos los campos que intervienen en el proceso, los cuales son muy variados. Además cualquier error que surja debe ser subsanado por una sola persona, y a veces puede resultar una tarea muy tediosa.

5.1 Conocimientos adquiridos en la carrera

Crear todas las partes que compondrán el videojuego y luego organizarlas de manera que funcionen como una máquina bien engrasada requiere unos conocimientos muy precisos de las herramientas utilizadas, así como conocimientos generales sobre desarrollo software, programación y otras ramas de la informática. Es aquí donde toman importancia los conocimientos adquiridos en la UPV. Asignaturas como:

- **MTP e ISS** (Metodología y Tecnología de la Programación) e (Ingeniería del Software de Sistemas): Aportan conocimientos cruciales sobre las fases de desarrollo y los ciclos de vida del software, es por ello que han sido de gran ayuda a la hora de planificar y llevar a cabo el proyecto.
- **SO, PR y EDA** (Sistemas Operativos), (Programación) y (Estructura de Datos y Algoritmos): Me han aportado conocimientos de programación sin los cuales me hubiera sido imposible implementar este software.
- Claro está también que todas las demás asignaturas aportan su granito de arena a este proyecto, pues han ayudado a formar las ideas y a cohesionar dichos conocimientos.

5.2 Problemática

Durante el desarrollo del videojuego pueden surgir muchos problemas. Algunos de los más importantes quedarían englobados en los siguientes puntos:

- Al no contar con compositor el proyecto tendrá que adaptarse a los efectos de sonido libres que pueda encontrar.

- Poca experiencia en el desarrollo de videojuegos.
- Problemas a la hora de desarrollar la IA de los enemigos. La IA es uno de los puntos más difíciles a la hora de desarrollar un videojuego. A un programador le consume mucho tiempo el tener que dotar de una cierta inteligencia a los personajes de un juego.
- Problemas para cuadrar las animaciones de los personajes.
- El tiempo necesario para desarrollar un videojuego por una sola persona puede ser extremadamente cuantioso.
- Puesto que disponía de tiempo limitado y nada de ayuda externa, al final sólo he podido implementar uno de los tres niveles pensados para la demo.
- El acople de RT-DESK a Unity parece tener serios problemas de compatibilidad.

5.3 Conclusiones RT-DESK

También se ha intentado el acople de la tecnología RT-DESK a dicho videojuego. No obstante se ha llegado a la conclusión de que dicho acople no es posible si el motor utilizado para el desarrollo del videojuego es Unity, por las razones expuestas en los apartados 6.3 y 6.4. Este descubrimiento puede resultar tremendamente útil por dos razones. Por un lado puede servir como precedente a un futuro desarrollador que pretenda realizar el mismo acople y ahorrarle mucho tiempo. Por otro lado puede servir como motivación para la realización de una versión de RT-DESK que sea compatible con Unity.

Estos errores de compatibilidad son una verdadera lástima. RT-DESK ha demostrado en otros proyectos ser capaz de realizar grandes mejoras de rendimiento en las aplicaciones de tiempo real y, dado que el nivel de mejora depende de cada aplicación, hubiese sido muy interesante hacer pruebas con RT-DESK para un videojuego en 3D. Esta opción queda abierta a proyectos futuros, siempre y cuando se llegue a solucionar los problemas de compatibilidad entre RT-DESK y Unity.

6. RESULTADOS

Planificar un proyecto, aunque no sea un proyecto de gran envergadura, nunca es fácil. En mi caso empecé con el proyecto en marzo de 2014. Antes de realizar ninguna planificación seria pensé, iluso de mí, que podría tener el proyecto terminado en un par de meses. Después de realizar la planificación del proyecto me di cuenta de que iba a ser bastante más complejo de lo que me imaginaba. En retrospectiva veo el fallo, y he de decir que la realización de un videojuego es muchísimo más complicada de lo que alguien ajeno a ella pueda pensar.

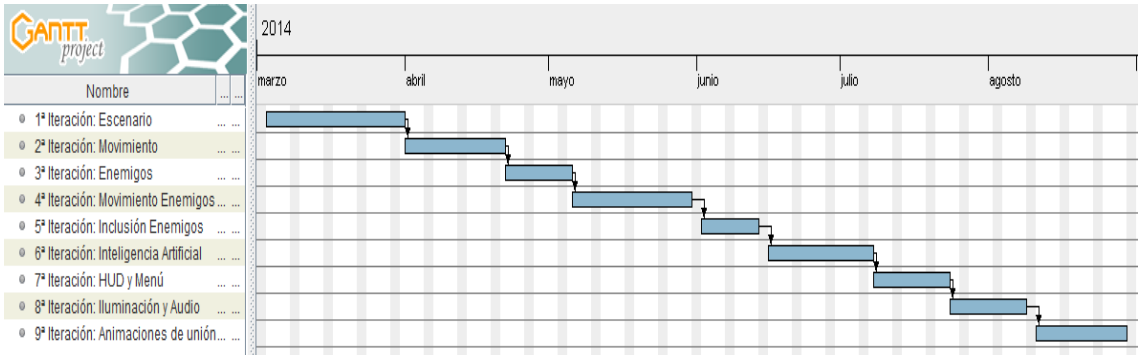
Una vez abandonada la idea de terminar el proyecto en un tiempo relativamente breve, empecé a planificarlo desde un punto de vista más realista e intentando tener en cuenta cada aspecto del proyecto, la estimación del tiempo necesario para llevarlo a cabo y un margen de tiempo extra para subsanar los errores que, con toda seguridad, surgirían. Con todo ello decidí no ponerme límites de tiempo y dedicar al proyecto todos los recursos que fueran necesarios, con la idea de que trabajando duro era posible realizar un buen proyecto dentro de unos márgenes de tiempo razonables.

Así pues la siguiente tabla muestra la diferencia entre el tiempo estimado de cada iteración y el tiempo que finalmente me llevó implementarlas:

ITERACIONES	**TIEMPO ESTIMADO**	**TIEMPO REAL**
1ª iteración: escenario principal	Dos semanas	Cuatro semanas
2ª iteración: movimiento	Dos semanas	Tres semanas
3ª iteración: enemigos	Una semana	Dos semanas
4ª iteración: movimiento de los enemigos	Dos semanas	Tres semanas
5ª iteración: inclusión de los enemigos en el proyecto	Una semana	Dos semanas
6ª iteración: Inteligencia Artificial	Tres semanas	Tres semanas
7ª iteración: HUD y menú principal	Dos semanas	Dos semanas
8ª iteración: Iluminación y audio	Una semana	Dos semanas
9ª iteración: Animaciones de unión	Una semana	Dos semanas

Así la estimación del tiempo total era de unos tres meses y medio, cuando en realidad me llevó finalmente casi seis meses realizarlo.

Diagrama de Gantt: Tiempo Real



La complejidad del proyecto también ha afectado, aparte de al tiempo requerido, al número de niveles finalmente implementados. El videojuego completo está pensado para que tenga un total de niveles no inferior a diez, de los cuales para esta demo pensaba implementar tres. Al final, por motivos de tiempo y complejidad, la demo ha quedado reducida a un único nivel.

7. BIBLIOGRAFÍA

(ESA) The Entertainment Software Association. http://www.theesa.com

(ADESE) / (AEVI) Asociación Española de Distribuidores y Editores de Software de Entretenimiento. http://www.adese.es http://www.aevi.org.es

(METODOLOGÍAS)

(XP) Extreme Programming Examined. Autor: Giancarlo Succi, Michele Marchesi.
Qué incluye:
33 documentos sobre XP sacados de la conferencia de Junio del 2000 sobre programación extrema, abordando diversos puntos de vista.

(SCRUM)
Título**:** Manual imprescindible de métodos Ágiles y Scrum.
Autor: Alonso Álvarez García, Rafael de las Heras del Dedo, Carmen Lasa Gómez
Editorial: Anaya Multimedia
Identificador**:** ISBN 9788441531048

(RT-DESK)
D.E.S.K., Discrete Events Simulation Kernel. I. Garcia, R. Molla, E. Ramos y M.Fernandez. European Congress on Computational Methods in Applied Sciences and Engineering ECCOMAS. Barcelona 11-14 Septiembre 2000.

Game Discrete Event Simulation Kernel. I. Garcia, R. Molla y T. Barella. Journal of WSCG, Vol.12, No.1-3, ISSN 1213-6972, February 2-6, 2004, Plzen, Czech Republic.

8. ANEXO I: DOCUMENTO DE DISEÑO

VERSIÓN DEL DOCUMENTO DE DISEÑO Y FECHA DE LA ÚLTIMA REVISIÓN

- 6 (25/07/2014), Javier Garrigues Tudela

PLATAFORMAS

- PC
- Posibilidad de portabilidad a Android / IOS

TÍTULO

- TROY STORY.

GÉNERO

- Plataformas 3D.

TEMÁTICA

- Ciencia Ficción.

TARGET

- Dirigido a cualquier tipo de jugador, tanto adultos como menores.

DESCRIPCIÓN GENERAL

Troy Story es un juego de plataformas 3D en tercera persona, inspirado en la mítica película infantil “Toy Story”, en el que el usuario controla al personaje principal, Troy. Troy Story sigue la historia de Troy, un chico normal aficionado al cine y a los videojuegos, que se ve atrapado en una trepidante aventura en la que tendrá que vérselas con sus juguetes de la infancia para poder escapar de su propia casa convertida en una trampa mortal, tras haber quedado reducido (Troy) a un tamaño similar al de uno de sus juguetes.

Troy Story sigue un modo historia. El usuario irá descubriendo con el tiempo el porqué de la situación en la que se encuentra nuestro protagonista, e irá dilucidando poco a poco cómo apañárselas para salir con vida de semejante controversia.

Troy Story tiene un componente caótico, el jugador tiene que correr, saltar, disparar, esquivar balas, esquivar enemigos, etc. En cada nivel el usuario tendrá que reaccionar rápidamente ante los ataques de cada uno de los enemigos. El juego pone a prueba los reflejos del jugador, su capacidad para manejar al personaje, su concentración y su habilidad a la hora de reconocer a qué tipo de enemigos nos enfrentamos, y cómo derrotarlos. El videojuego completamente desarrollado tendrá un final de la historia un tanto inesperado, propio de una película de cine.

ASPECTOS CLAVE

- El juego sigue una historia, que se va desarrollando conforme el jugador avanza.
- Los enemigos son controlados por IA.
- Cada uno de los enemigos ataca de manera distinta.
- La partida se puede guardar y reanudar.
- Diferentes escenarios, que se descubren a medida que avanza el juego.
- La velocidad de reacción y los reflejos son un componente clave a la hora de evitar los ataques de los enemigos.
- Dispondremos de armas arrojadizas para luchar.
- El juego tiene un componente de exploración. Exploraremos la casa desde los ojos de un juguete.
- El camino que debe seguir el personaje principal no está completamente marcado, el ingenio es imprescindible para llegar a los sitios más altos de cada escenario.

HISTORIA

Troy es un joven bastante normal, aficionado a los videojuegos y al cine. Pasa mucho de su tiempo libre en el escritorio de su habitación, jugando con el ordenador. Una tarde, tras haberse pasado algunas horas jugando a su videojuego favorito, Troy sufre una especie de shock y despierta encima del teclado de su ordenador. Después de recomponerse se queda perplejo al observar que su tamaño ha sido reducido al de un juguete convencional.

Sin tiempo para poder asimilar lo ocurrido, Troy descubre que sus juguetes le están atacando. Es entonces cuando decide que tiene que salir de su casa en busca de ayuda, aunque ello suponga arriesgar su vida enfrentándose a los juguetes de su infancia.

Troy deberá ingeniárselas para sobrepasar a todos y cada uno de sus juguetes, y descubrirá los peligros que acechan en su anteriormente tranquila casa.

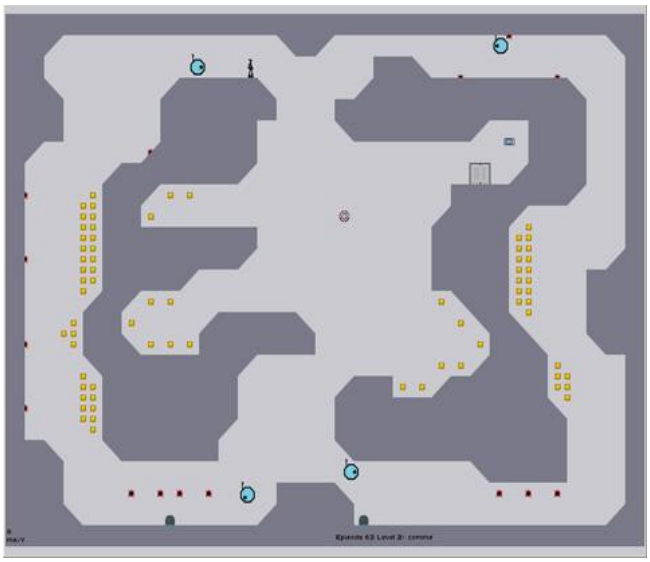
REFERENCIAS

En este proyecto se pueden observar pinceladas de otros videojuegos:

Super Mario 64. Seguramente el primer juego de plataformas en 3 dimensiones. De este videojuego he cogido como referente el estilo del movimiento y los saltos del personaje principal.

Toy Story. Adaptación al formato videojuego de la película de animación 3D del mismo nombre. De esta maravillosa saga he cogido como referente la estética y la temática del videojuego.

Ninja n+. Videojuego de plataformas 2D. El ritmo caótico de las partidas de este videojuego me cautivó y decidí seguir la misma línea de interacción con el entorno y los enemigos.



ESTÉTICA

El juego tendrá una estética realista y un tanto infantil. Los personajes y los objetos están bien proporcionados, pero el colorido y las luces proporcionan el toque infantil en que se basa el juego.

JUGABILIDAD

Troy Story presenta una jugabilidad basada en reflejos y acciones tomadas rápidamente por el jugador. La rapidez y la destreza son aspectos clave en el desarrollo de la partida, así como el ingenio y la observación del entorno.

El jugador puede:

- Moverse hacia delante y hacia atrás.
- Desplazarse lateralmente a izquierda y derecha.
- Saltar.
- Recoger objetos arrojadizos (gomas de borrar) para lanzárselos a los enemigos.
- Dirigir con el ratón al personaje en tiempo real.
- Combinar cada una de las opciones anteriores.

CONTROLES

Los controles del videojuego son los siguientes:

Puesto que la cámara principal de la escena orbita alrededor del personaje principal, el usuario controla la orientación del personaje principal mediante el ratón, pudiendo además realizar *zoom-in* o *zoom-out* con la rueda del ratón.

Si se realizara portabilidad a otro sistema operativo, como por ejemplo Android habría que definir nuevos sistemas de control y botonería.

PERSONAJES

TROY. Es un joven chico normal, de composición atlética. A lo largo del juego pondremos a prueba su agilidad y sus reflejos.

Clasificación: Personaje principal.

Comportamiento: Dirigido por el jugador.

Munición: La munición del personaje principal consta de objetos arrojadizos que vamos recogiendo a lo largo de la partida. Con ellos podemos atacar a los diferentes enemigos que encontremos.

Animaciones:

- **Idle:** Animación del personaje en estado de reposo, cuando no apretamos ningún botón.
- **Correr**: *Loop* simple de carrera.

- **Andar hacia detrás:** *Loop* de caminar de espaldas.
- **Desplazamiento lateral (izquierda y derecha):** *Loops* de desplazamiento lateral.
- **Saltar:** Animación simple de salto con voltereta.
- **Aterrizar:** Cada vez que realizamos un salto, cuando Troy toca el suelo se renderiza esta animación que sirve de puente entre el estado de salto y los estados de Idle o correr.
- **Disparo:** Animación de Troy lanzando una munición con la mano.
- **Muerte:** Si morimos, Troy cae al suelo realizando una animación de muerte.



SOLDADITOS DE PLÁSTICO.

Clasificación: Enemigo.

Comportamiento: Patrullan en grupo, siempre vigilantes. Si te acercas demasiado te detectarán. Si te detectan comenzarán a dispararte, inmóviles, hasta que te alejes a una cierta distancia. La única manera de acabar con ellos es mediante objetos arrojadizos.

Munición: Poseen munición ilimitada. Dicha munición consta de balas de plástico que en caso de impacto decrementan la vida de Troy. Con tres impactos directos Troy pierde toda su vida.

Animaciones:

- **Desplazamiento**: Tan sólo poseen una animación puesto que se desplazan a saltos, debido a que tienen los pies pegados a su base.

ROBOT.

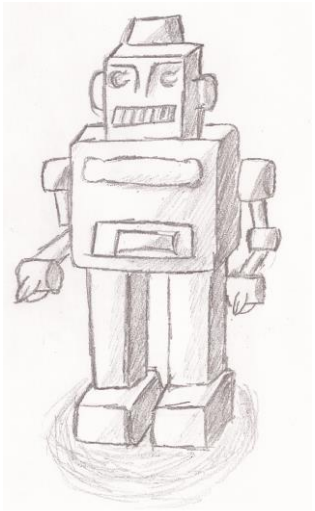
Clasificación: Enemigo.

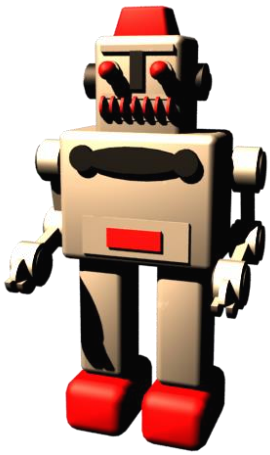
Comportamiento: Patrullan en solitario. Si te detectan te perseguirán. Su movimiento es más rápido que el del resto de enemigos. Se puede acabar con ellos, pero ello requiere el impacto directo de tres proyectiles.

Munición: No posee munición. El mismo impacto directo del cuerpo del robot contra Troy destruye a ambos.

Animaciones:

- **Idle**: Estado de reposo del robot. Esta animación ocurre mientras no se detecte la presencia del personaje principal.
- **Despertar**: Cada vez que el robot detecta a Troy se despierta de su estado de Idle, justo antes de perseguirlo.
- **Correr**: *Loop* de correr para el robot.
- **Dormir**: Cuando Troy se sale del alcance de la detección del robot, éste se vuelve a su posición de reposo (Idle). Esta animación funciona como puente entre la persecución y el estado de reposo del robot.





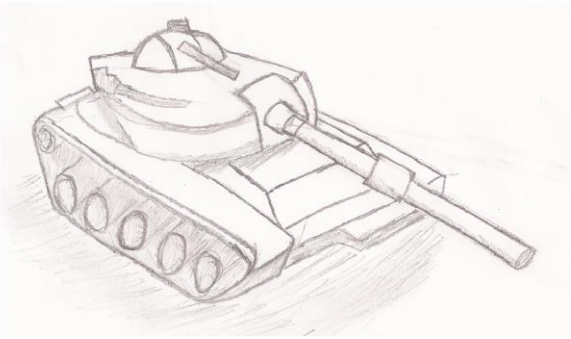
TANQUE DE GUERRA.

Clasificación: Enemigo.

Comportamiento: Hacen un recorrido fijo. Te detectarán a más larga distancia que cualquier otro enemigo. Su disparo es también más largo y más dañino que los demás enemigos. Para destruirlos debes saltar sobre ellos.

Munición: Poseen munición ilimitada, que consta de balas de cañón. El impacto de una sola bala de cañón es letal para Troy.

Animaciones: No poseen animaciones. El desplazamiento y la rotación del tanque se realizan tratando al objeto tanque como un "todo", siendo el cañón la parte delantera del objeto.



INTERACCIÓN DE TROY CON EL ENTORNO

Nuestro personaje principal interacciona los siguientes componentes del nivel: **enemigos** (Robot, Soldadito, Tanque), **escenario**, **proyectiles**, **munición**, **objetivo** e **imán**.

- **Enemigos**: La interacción de Troy con los enemigos está explicada en el apartado PERSONAJES.
- **Escenario**: Cada objeto del escenario es una plataforma sólida donde nuestro personaje puede apoyarse para realizar sus movimientos. Si Troy salta de una altura considerablemente alta (unas cinco veces su tamaño) resultará en la muerte de Troy, con excepción de si éste aterriza en la cama.
- **Proyectiles**: Los proyectiles de los Soldaditos hieren a Troy, pero es necesario tres de ellos para acabar con él. Sin embargo los proyectiles de los Tanques matan a Troy de un solo golpe.
- **Munición**: Sólo hay un tipo de objeto munición, la munición de Troy. Si tocamos este objeto incrementamos la munición de Troy en cinco unidades.
- **Objetivo**: El objetivo de la pantalla está englobado en un haz luminoso azul, situado justo debajo de la puerta de la habitación. Si llegamos a él sin el imán

necesario para abrir la puerta nos aparece un mensaje que nos indica que debemos recoger el imán.

- **Imán**: Debemos tocar este objeto para recogerlo. Una vez recogido podemos completar el nivel llegando al objetivo.

HUD (Heads-Up Display)

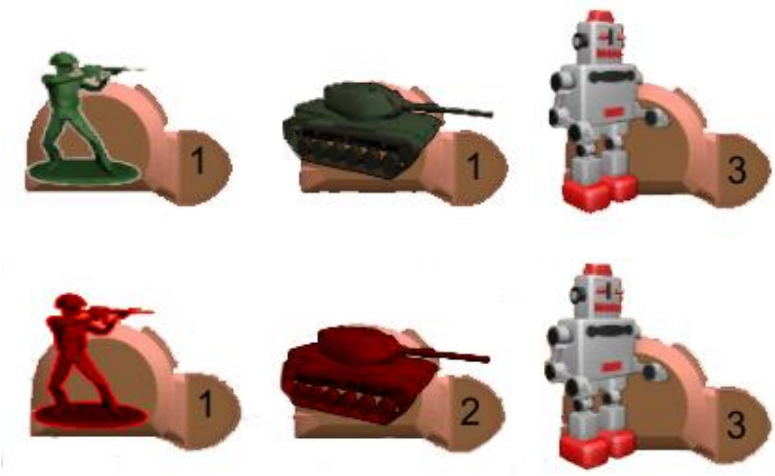
La interfaz gráfica debe proporcionar al usuario información esencial para el desarrollo de la partida. Normalmente aparecen en ella elementos como: la vida restante, mapa de la pantalla, número de enemigos, munición disponible, etc.

En este caso la interfaz que se muestra para el personaje principal está formada por:

Barra de vida: Muestra la vida restante del jugador respecto de la vida total con la que se empieza la partida.

Munición: En principio la munición consta de diferentes objetos arrojadizos que encontramos a lo largo de cada nivel. En la demo sólo se implementará el objeto Goma. Empezamos con una munición de 10 unidades, cada vez que eliminamos a un enemigo de tipo Soldadito, éste nos deja 5 unidades de munición disponibles para recoger en el lugar de su muerte.

Enemigos restantes: Muestra información de los enemigos que quedan, así como del estado de alerta en el que están. Si un enemigo detecta a Troy, su icono se vuelve de color rojo.



ESCENARIOS

El juego totalmente desarrollado contará con un número elevado de niveles. La demo del juego está pensada para tener una extensión total de tres niveles, con posibilidad de aumentarse:

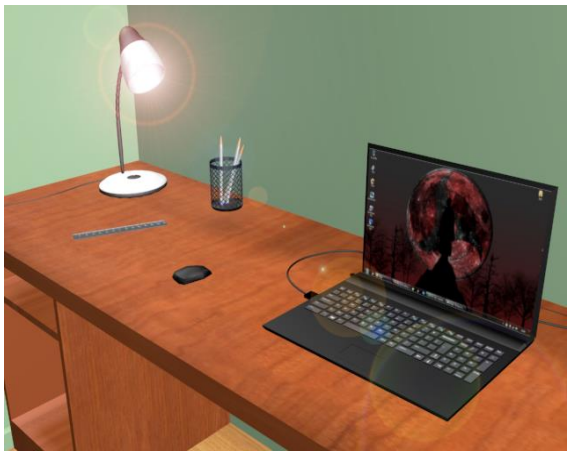
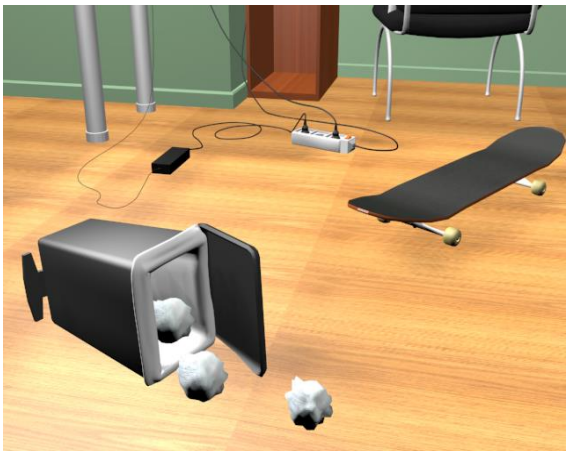
- Habitación.
- Pasillo.
- Entrada.

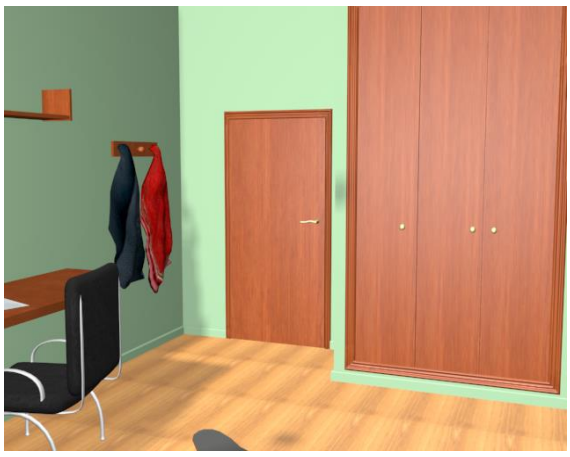
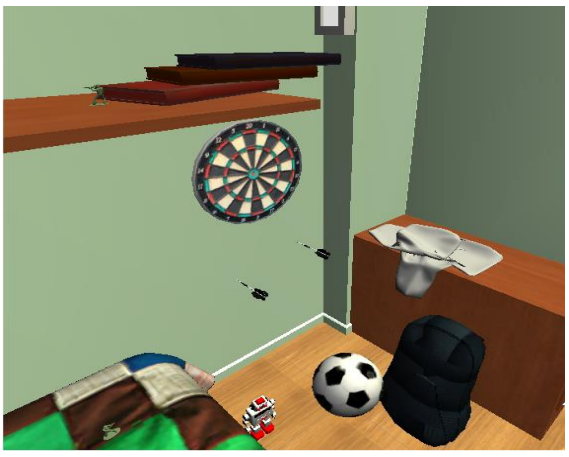
Por motivos de tiempo finalmente sólo se implementará el escenario: Habitación (Room).

En este nivel encontramos todo tipo de objetos con los que podemos interaccionar: paredes, suelo, techo, luces, dos mesas de escritorio, dos sillas de escritorio, cinco estanterías, un perchero, ordenador portátil, flexo, papelera, libros, cama, botes de escritorio, lápices, bolígrafos, un cuadro, un reloj de pared, cableado, ventana,

persianas, diana con dardos, enchufes, botes para CD-ROM, monopatín, pelota de fútbol, mochila, copa, puerta principal y puertas de armario.

Todos estos elementos nos ayudan a evadir a nuestros enemigos y a la vez nos guían por la habitación marcando el camino a seguir para completar el nivel. Imágenes de muestra:

OBJETIVO

El objetivo del nivel es conseguir cruzar toda la habitación para apoderarnos de un imán situado a lo más alto de la más alta estantería, justo encima de la copa. Con dicho imán conseguiremos atraer el pomo de la puerta para abrirla y salir de la habitación, pasando así al siguiente nivel.

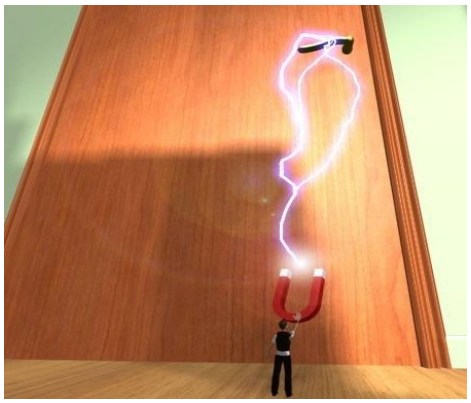
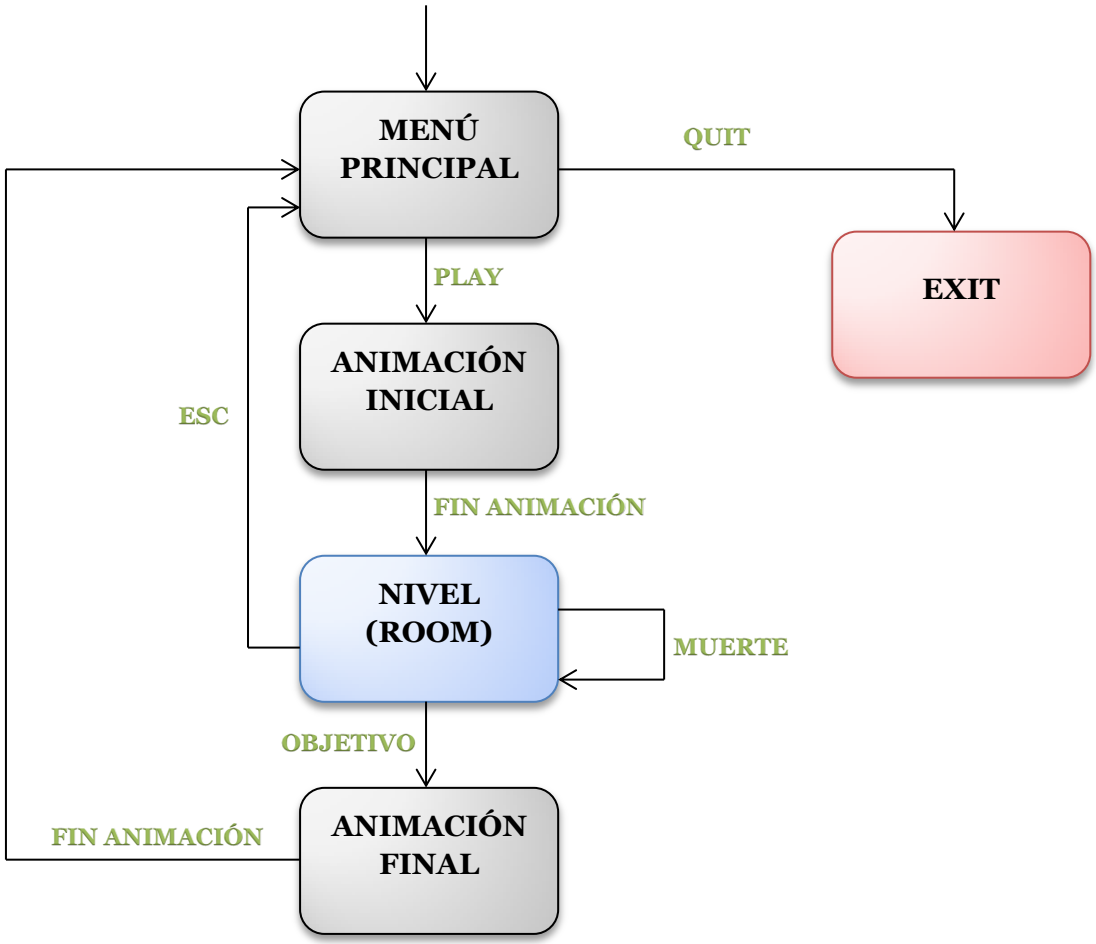


DIAGRAMA DE FLUJO DEL JUEGO



ILUMINACIÓN

La iluminación en el videojuego consiste en una serie de luces dinámicas creadas en el entorno de Unity. Las luces dinámicas se renderizan en tiempo de ejecución, por lo que son más pesadas y consumen más recursos que las luces pre-renderizadas.

La iluminación en la escena de la habitación consta de un total de siete luces:

- **Iluminación exterior 1 (Sol):** Es la luz principal de la escena. Ésta ilumina toda la escena creando la sensación de luz solar. Produce sombras.
- **Iluminación exterior 2:** Es una luz similar a la del Sol, con la diferencia de que ilumina en sentido contrario y con menos intensidad. Además no produce sombras. Esta luz tiene el objetivo de aclarar las sombras que provoca la luz principal.
- **Luces internas:** Son cuatro. Simulan los ojos de buey que iluminan el interior de la habitación. Producen sobras leves.
- **Luz de flexo:** Es la luz que produce el flexo del escritorio. Produce sombras leves.

Todas estas luces producen el ambiente realista que caracteriza al videojuego y ayudan al usuario a sumirse en él.

TIEMPO DE JUEGO

Se trata de un videojuego exclusivamente en modo historia, por lo que el tiempo de juego se puede, en cierta medida, estimar a priori. Una vez que el jugador se ha acostumbrado a los controles, no debería tardar más de 30 minutos en completar cada nivel. El tiempo de juego total dependerá del número de niveles que finalmente se implementen.

9. ANEXO II: CONSIDERACIONES LEGALES

Debido a que este proyecto está pensado para que permanezca accesible por cualquier persona interesada, y sin ánimo de lucro, algunos de los elementos que lo componen pueden estar sujetos a licencias comerciales que no se han tenido en cuenta. Entre dichos elementos se encuentran:

- **Imágenes.** Algunas de las imágenes utilizadas como mapa para elementos de la habitación o para partes del cuerpo de los personajes.

- **Sonidos.** Algunos de los sonidos del videojuego, entre los que se encuentra la música principal (*The Great Escape, de Elmer Bernstein*).

Siendo este proyecto sin ánimo de lucro no debería haber problemas con las licencias, pero es recomendable tener esto en cuenta a la hora de que alguna entidad, de ámbito público o privado, se interesara por el proyecto y decidiera retomarlo.