Escuela Politécnica Superior de Jaén
Grado en Ingeniería Informática

Universidad de Jaén
Escuela Politécnica Superior
de Jaén

TRABAJO FIN DE GRADO

REBOOT DEL VIDEOJUEGO CRAZY TAXI

Alumno

Víctor Rodríguez Cano

Tutor

Carlos Javier Ogayar Anguita
(Departamento de Informática)

Junio, 2023

Universidad de Jaén

Departamento de Informática

Don Carlos Javier Ogayar Anguita, tutor del Trabajo Fin de Grado titulado: '**Reboot del videojuego Crazy Taxi**', que presenta Don Víctor Rodríguez Cano, otorga el visto bueno para su entrega y defensa en la Escuela Politécnica Superior de Jaén.

Jaén, Junio de 2023

El alumno: El tutor:

Víctor Rodríguez Cano Carlos Javier Ogayar Anguita

Agradecimientos

Quiero transmitir desde aquí mi más profunda gratitud en primer lugar a mis padres y a toda mi familia, que siempre estuvieron ahí y me apoyaron incondicionalmente en todo momento. Sin ellos no habría llegado tan lejos.

A todos mis amigos, tanto los de toda la vida como a aquellos que conocí recientemente, por aguantarme y quereme tal y como soy.

También quiero agradecer a los profesores de la UJA que coincidieron conmigo de alguna manera, haciendo mención especial a Carlos, mi tutor de TFG, que pese a haber estado dado de baja, ha hecho el esfuerzo de tutorizarme y sacar tiempo para responder mis incansables dudas e inquietudes.

FICHA DEL TRABAJO FIN DE TÍTULO	
Titulación	Grado en Ingeniería Informática
Modalidad	Proyecto de Ingeniería
Especialidad (solo TFG)	Tecnologías de la Información
Mención (solo TFG)	Sistemas Gráficos
Idioma	Español
Tipo	Específico
TFT en equipo	No
Autor/a	Víctor Rodríguez Cano
Fecha de asignación	Haga clic aquí o pulse para escribir una fecha.
Descripción corta	De la mano del término de reboot, grandes empresas han creado exitosos juegos bajo el nombre de una franquicia existente, incluyendo en ellos cambios sustanciales que consiguen darle un renacer a la misma. Este proyecto tendrá como objetivo combinar la clásica saga de Crazy Taxi con la esencia de los conocidos juegos Minecraft y Grand Theft Auto, llevando así al mítico arcade a una versión alternativa nunca antes vista. Entre los puntos más destacados, abarcaremos desafíos tales como la generación procedural de ciudades, métodos de navegación exigentes para tiempo real o un sistema inteligente de NPCs para vehículos y personas.

NORMAS APLICADAS EN ESTE DOCUMENTO	
LOCALES	
TFT-UJA:2017	Normativa de Trabajos Fin de Grado, Fin de Máster y otros Trabajos Fin de Título de la Universidad de Jaén (Normativa marco UJA aprobada en Consejo de Gobierno)
TFT-EPSJ:2017	Normativa sobre Trabajos Fin de Grado y Fin de Máster en la Escuela Politécnica Superior de Jaén (Normativa EPSJ aprobada en Junta de Escuela)
TFT-EPSJ	Criterios de evaluación y normas de estilo para TFG y TFM de la Escuela Politécnica Superior de Jaén
NACIONALES E INTERNACIONALES	
ISO 2145:1978	Documentación - Numeración de divisiones y subdivisiones en documentos escritos
UNE 50132:1994	Traducción de la ISO 2145
APA 6ª edición	Estilo de referencias y citas de APA (American Psychological Association)

NORMAS UTILIZADAS COMO BASE O REFERENCIA	
NACIONALES	
UNE 157001:2014	Criterios generales para la elaboración formal de los documentos que constituyen un proyecto técnico
UNE 157801:2007	Criterios generales para la elaboración de proyectos de sistemas de información
*Estas normas se han utilizado **como base o referencia** para la inclusión de algunos contenidos y definiciones sobre elaboración de proyectos, entendiendo como **proyecto** la documentación consensuada entre una empresa y un cliente, que da lugar al perfeccionamiento de un contrato para la elaboración de una obra o la prestación de un servicio. Por consiguiente, no debe esperarse la aplicación de estas normas en cuanto a la completitud de los contenidos ni a la organización de los mismos.*	

Contenido

Índice de ilustraciones

Índice de tablas

1 ESPECIFICACIÓN DEL TRABAJO

En este capítulo se presenta la especificación del trabajo, con una estructura y contenidos **inspirados** en los criterios y recomendaciones que establece la norma UNE 157801:2007 - "*Criterios Generales para la elaboración de proyectos de Sistemas de Información*".

A lo largo del documento se utilizarán términos y acrónimos cuya descripción aparecen en el apartado 8 (Definiciones y abreviaturas).

1.1 Introducción

Aunque los remakes y las remasterizaciones están más de moda que nunca, muchas veces empezar de cero es la opción ideal para revitalizar una serie que se está apagando, aportando nuevas ideas a la marca. En el mundo de los videojuegos para conseguir ese objetivo se toma un juego conocido junto con sus mecánicas, y se empieza desde cero para crear uno nuevo dentro del entorno del juego original. Esta idea es el fundamento de un reboot o reinicio en videojuegos.

A lo largo de esta memoria se describirá el proceso de análisis, diseño y desarrollo llevado a cabo para realizar un reboot que buscará combinar la clásica saga de Crazy Taxi [1] con la esencia de los 2 juegos más vendidos de la historia [2]: Minecraft [3] y Grand Theft Auto (GTA) [4].

Ilustración 1.1: Crazy Taxi

Con el fin de revitalizar la saga y demostrar los conocimientos adquiridos tras cursar el grado de ingeniería informática, este proceso terminará dando a luz un juego nuevo, ubicado dentro del universo del videojuego Crazy Taxi, donde se abarcarán desafíos tales como la generación procedural de ciudades, métodos de navegación exigentes para tiempo real o un sistema inteligente de NPCs constituido por vehículos y personas.

1.2 Objetivos del trabajo

El objetivo principal es diseñar y desarrollar una versión alternativa del videojuego Crazy Taxi (1999), mejorada con elementos y mecánicas encontradas dentro de Minecraft y GTA, a partir de un motor de desarrollo de videojuegos contemporáneo. Para ello destacaremos una serie de objetivos que se tendrán en cuenta para la base del proyecto:

1. Selección de un motor de videojuegos que permita fácil creación de aplicaciones multiplataforma y actividad de aprendizaje requerida para su utilización.
2. Actualización del apartado gráfico del juego con estilo *voxelart* [5] similar al de Minecraft y proceso de aprendizaje requerido para el uso de las herramientas de edición necesarias.
3. Recreación y ampliación de las mecánicas del juego original, incluyendo mecánicas de GTA y nuevos modos de juego.
4. Creación de un sistema de NPCs que esté a la altura del sistema de NPCs original, incluyendo comportamientos inteligentes y novedosos no vistos en el juego original.
5. Búsqueda y aplicación de animaciones y efectos de sonido encontrados dentro de Crazy Taxi.
6. Maximizar la optimización del juego para conseguir un buen rendimiento en la mayoría de ordenadores.
7. Hacer la aplicación de forma que se busque su internacionalización [6].

1.3 Antecedentes y estado del arte

Procedente del término inglés *reboot*, un reinicio se refiere al relanzamiento de una historia presentando una inflexión en la serie donde se ignora la continuidad previa, conservando los elementos más importantes e incluyendo ideas más frescas o reinterpretadas.

Este movimiento es una práctica común en cómics, donde los escritores comienzan una nueva historia con los mismos personajes sin tener en cuenta la historia anterior, permitiendo que nuevos lectores sean atraídos y proporcionando gran libertad al escritor. Frecuentemente en este campo se suele utilizar la idea del multiverso para explicar que la nueva historia alternativa se da en un universo paralelo al original.

Aunque sea complicado determinar cuál fue el primer reboot de la historia, es innegable que la popularidad de éstos ha ido creciendo gracias a su aparición en el ámbito del cine y televisión, reiniciando franquicias con títulos de gran calibre como Godzilla (1954) o The Terminator (1984). Esta tendencia ha seguido firme como una apuesta segura para los estudios cinematográficos y muy alabada por los fans, encontrando éxitos en taquilla para reboots de insignias más actuales como Spider-Man (2002):

- The Amazing Spider-man (2012)
- Spider-Man: Homecoming (2017)

Con los avances en el campo de videojuegos en estos últimos años, el reinicio ha llegado como una nueva opción para los desarrolladores para revivir franquicias que sean capaces de atraer a antiguos y nuevos jugadores, creando además un escenario de seguridad económica para la empresa. A continuación, vamos a hablar de unos ejemplos de reboots conocidos:

- **Castlevania Lords of Shadow**: lanzado en el año 2010 y de la mano de estudios españoles, la saga Castlevania [7] dio un gran giro de guion proporcionando una nueva historia en una línea de tiempo completamente diferente a la que encontrábamos en entregas anteriores. Ahora el objetivo no sería acabar con Drácula, sino que pasaría a ser la búsqueda de

fragmentos de la máscara de Dios y traer a la difunta esposa del protagonista a la vida de entre los muertos. Esto hizo posible un enfoque más oscuro y épico para la entrega junto con un gran cambio de mecánicas y una cámara renovada con la que jugar en tercera persona en lugar de con la cámara lateral de juegos anteriores.

El reboot fue altamente bien recibido por la comunidad, llegando a ser uno de los juegos de la saga con mayor número de ventas y proporcionando aire fresco a la misma.

Ilustración 1.2: Castlevania: Aria of Sorrow

Ilustración 1.3: Castlevania: Lords of Shadow (reboot)

- **Wolfenstein The New Order**: es otro claro ejemplo de reboot sacado en el año 2014 que impulsó nuevamente a la saga Wolfenstein [8] donde abre una nueva línea temporal en la que, a diferencia de la original, los nazis ganan la segunda guerra mundial y tienen el poder total sobre el mundo. En este reinicio se cambia tanto la trama como la jugabilidad y se incluye una narrativa más profunda, elementos que hacen al juego destacar entre las críticas en su lanzamiento.

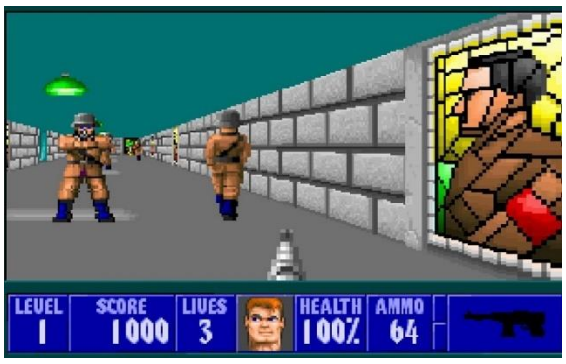

Ilustración 1.4: Wolfenstein 3D

Ilustración 1.5: Wolfenstein: The New Order (reboot)

- **Tomb Raider**: la versión de este juego lanzada en el año 2013 se considera asimismo un reboot de la franquicia de Tomb Raider [9], donde se redefine la identidad de la arqueóloga Lara Croft. Durante el desarrollo del reinicio, vemos a la protagonista con una apariencia más joven y se cuenta una nueva historia, donde se explica un pasado en el que se ven sus primeros pasos como aventurera.

  Con gran influencia proveniente de Uncharted [10], el reboot incorpora un enfoque más orientado a la acción y la exploración, con elementos de juego de sigilo, plataformas y combate que no se daban en las anteriores entregas. Este reboot fue muy importante porque dio pie al comienzo de una nueva visión de la trama, visión que provocó posteriormente una trilogía que continuaba a este juego.

Ilustración 1.6: Tomb Raider

Ilustración 1.7: Tomb Raider (reboot)

Por tanto, tras este breve repaso, al igual que con estos icónicos juegos comentados, en este proyecto se ha apostado por intentar hacer un reboot de uno de los clásicos de SEGA [11], Crazy Taxi. Aun conservando las mismas premisas y objetivos que se encontraban en la entrega original, buscaremos redefinir la franquicia mediante nuevos personajes, vehículos y modos de juego, añadiendo además una mezcla del universo de Crazy Taxi con elementos de otros juegos conocidos y un estilo artístico rompedor con la saga que pueda hacer destacar a nuestro reinicio.

1.4 Requisitos iniciales

En esta sección únicamente se determinan los requisitos iniciales a alto nivel, con objetivo de fijar el aspecto del resultado buscado y utilizarlos como referencia

durante la etapa de validación final. Los requisitos que se han propuesto para el proyecto con intención de cumplir son:

- Debe contener las mecánicas principales del Crazy Taxi original buscando maximizar la experiencia de usuario.
- Debe permitir jugar una partida similar a la del modo arcade del juego original.
- Debe desarrollarse un algoritmo procedural generador de ciudades.
- Debe desarrollarse un conjunto de interfaces, donde proporcionemos al usuario una navegación eficiente, atractiva e intuitiva.
- Debe conseguirse una apariencia atractiva con estilo *voxelart* similar al estilo de Minecraft que deje claro que la estética rompe con la original.
- Debe incluir un sistema policial y de armas como el de GTA.
- Debe tener música y sonidos de fondo que funcionen de manera interactiva y retroalimentativa con el jugador.
- Debería incluir al menos un modo normal de arcade y un par de modos especiales con mecánicas no vistas en las entregas de Crazy Taxi.
- Debería incluir un sistema de NPCs con capacidad de gestionar personas y vehículos en la ciudad.
- Debería hacerse un sistema de GPS que permita la navegación eficaz y eficiente al jugador y a los NPCs.
- Debería alcanzarse un nivel mínimo de internacionalización mediante la posibilidad de traducir el juego a múltiples idiomas.

1.5 Hipótesis y restricciones

Para la realización de este trabajo se cuenta de inicio con una base firme en lo que a uso de motores de videojuegos se refiere, conocimientos básicos de modelado 3D y con mucha soltura dentro de la programación en el lenguaje C#, C++ y Java, que son de los más utilizados en los motores actuales. Es por esta razón que el proyecto ha sido muy ambicioso desde el comienzo, ya que se ha tenido en cuenta la mayor

velocidad de realización que se daría, buscando crear un reboot completo en la mayoría de aspectos y sin quedarnos a mitad de camino con un prototipo de reboot.

El TFT se define como una asignatura de 12 créditos, lo que supone que la duración total del proyecto será de 300 horas. Sin embargo, es importante comprender que el producto de un proyecto de videojuegos es fruto de un trabajo muy extenso, que ni mucho menos se resume en saber utilizar un motor de videojuegos y que, pese al control que se haya adquirido de antemano, el tiempo sí que podría extenderse algo más del propuesto entre otras razones por el tiempo dedicado a la depuración de errores.

Debido al presupuesto que se tiene para la realización del proyecto y el poco tiempo del que se dispone, la gran mayoría de modelos y sprites utilizados para el desarrollo se han obtenido de forma gratuita en páginas especializadas (y los que no, se han creado nuevos), sin embargo, esto no ha supuesto ni mucho menos una bajada en la calidad gráfica. En lo que refiere al sonido no se ha dado ningún tipo de limitación, ya que todos los efectos de sonido y música se han extraído de una versión de Crazy Taxi. Igualmente, la fuente de letra se ha obtenido de manera gratuita en Google Fonts [12].

Cabe mencionar que no se poseen los derechos de ninguno de los assets, modelos, sprites, fuentes de letra, sonidos y efectos de sonido no propios utilizados a lo largo del proyecto, y que el uso que se les está dando a todos es meramente para un propósito académico y en ningún caso comercial.

1.6 Estudio de alternativas y viabilidad

Durante el proyecto se han tenido que discriminar el uso de algunas herramientas a favor de otras, ya sea para el motor gráfico, para el entorno de programación o para las herramientas de modelado y edición. En este apartado vamos a describir las principales alternativas analizadas para su posterior elección.

1.6.1 Motor gráfico

Una de las primeras decisiones que se han tenido que llevar a cabo durante el inicio del proyecto ha sido la elección del motor gráfico que se iba a utilizar. Para ello nos hemos tenido que documentar un poco acerca de los motores más populares y utilizados actualmente. A continuación se muestra un gráfico, Ilustración 1.8, donde aparecen los motores más de moda actualmente en la conocida página de Steam [13].

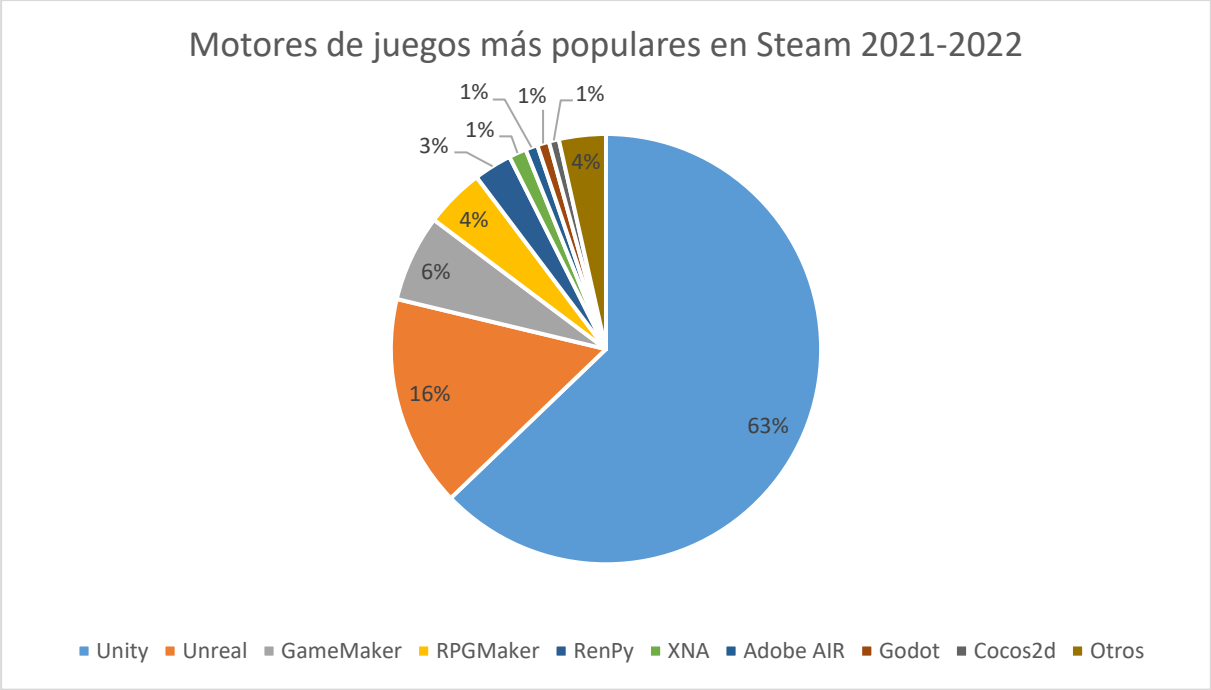

Ilustración 1.8: Popularidad de los principales motores de videojuegos en Steam

Como efectivamente vemos, la tendencia en la actualidad es clara, y es que la mayoría de videojuegos se realizan en el entorno proporcionado por Unity seguido del de Unreal Engine. Es por ello que serán las dos principales alternativas que tomaremos en cuenta.

1.6.1.1 Unity

Aunque para la creación de videojuegos triple A [14] y de mundo abierto es menos frecuente que Unreal Engine, lo cierto es que Unity es capaz de dar unos resultados impresionantes, tratándose del motor de videojuegos más popular en la actualidad debido a su flexibilidad, gran cantidad de funcionalidades y su curva de aprendizaje.

Unity es un motor que permite crear videojuegos 2D y 3D de manera gratuita con la versión Unity Personal, siempre y cuando el juego no genere más de 100.000 dólares anuales. Este motor permite diseño de videojuegos multiplataforma, encontrando desde plataformas como Windows, Linux o Mac hasta incluso plataformas de Realidad Virtual y Realidad Aumentada.

El motor soporta scripting en lenguaje C# y Java, algo que significa una ventaja para los programadores, pues hacer uso de estos lenguajes de más alto nivel aligeran el proceso de aprendizaje y de programación que si se programaran directamente en otros como C o en C++. Añadido a esto, el aprendizaje no solo se ve influenciado por el lenguaje de programación y el entorno, sino que es fuertemente impulsado por la gran comunidad que tiene y la cantidad de información que se encuentra en Internet, algo que ayuda a todo aquel que se inicia en el mundo del desarrollo de videojuegos.

Encontramos juegos muy conocidos realizados sobre este motor, como por ejemplo:

- Asasin's Creed Identity.
- Hollow knight.
- Rust.
- Pokémon Go.

Ilustración 1.9: Asasin's Creed Identity

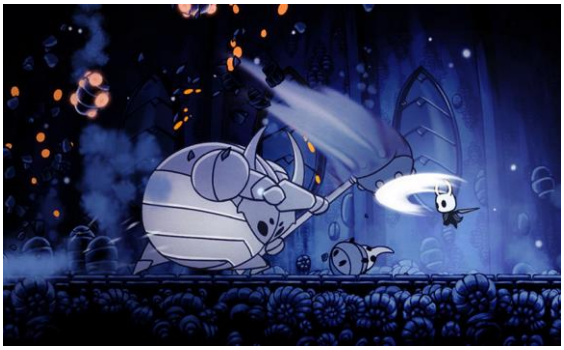

Ilustración 1.10: Hollow knight

En resumidas cuentas, Unity es un gran motor que proporciona accesibilidad a muchos desarrolladores al desarrollo de videojuegos y que es capaz de dar resultados muy buenos con una calidad de hasta triple A.

1.6.1.2 Unreal Engine

Es el motor creado por la compañía Epic Games [15] y se trata de uno de los motores más potentes que existen en la actualidad. Creado en 1998, a día de hoy este motor se encuentra en su versión 5, una versión cuyos resultados gráficos son impresionantes y que mejora mucho la creación de mundos abiertos frente a la versión anterior (Unreal Engine 4).

Aunque se pueden crear juegos en 2 dimensiones, Unreal Engine 5 es fuerte en la creación de videojuegos tridimensionales, pues soporta escenas con mallas de gran cantidad de vértices en tiempo real e incorpora numerosas novedades donde es muy destacable su aspecto de iluminación. Efectivamente, esta quinta versión introduce Lumen [16], el sistema de iluminación global basado en el uso de la técnica de raytracing [17]. Además de las cuestiones de iluminación, Unreal Engine 5 ofrece grandes herramientas como Nanite [18], bibliotecas de objetos 3D y la posibilidad de la fácil migración de la versión 4 a la versión 5.

En cuanto al trabajo en este entorno, éste se lleva a cabo haciendo uso del lenguaje de programación C++. Aunque además de C++, encontramos una alternativa llamada Blueprints [19], que permite hacer el código con pocos conocimientos de programación mediante una metodología basada en nodos visuales e interconexiones entre ellos.

Aun encontrando una muy fuerte comunidad, este motor es considerado un motor donde la curva de aprendizaje para aprender a usarlo es más elevada que la que encontramos en Unity. A esto se le suma la exigencia de recursos que requiere, pues, para su última versión se requiere de un muy potente ordenador para poder trabajar correctamente.

Sea como sea, es una excelente elección para la creación de videojuegos de nivel triple A, viéndose ya ejemplos de juegos que lo utilizan o que están en desarrollo dentro de este motor, tales como:

- The Matrix Awakens.
- S.T.A.L.K.E.R. 2: Heart of Chernobyl.
- The Witcher 4.
- Dragon Quest XII: The Flames of Fate.

Ilustración 1.11: The Matrix Awakens

Ilustración 1.12: S.T.A.L.K.E.R. 2

1.6.2 Entorno de programación

En cuanto al entorno de programación, durante la carrera se han utilizado varios entornos diferentes con los que de alguna manera hemos sido familiarizados. De entre todos los entornos, destacaremos los 2 con los que más soltura se posee.

1.6.2.1 Apache NetBeans

Apache NetBeans es un entorno que proporciona un gran conjunto de herramientas que ayudan de manera inteligente durante la codificación. Este es el preferido por parte del autor ya que es donde más tiempo ha pasado programando. Se trata de un software gratuito y de código abierto que se encuentra disponible para Windows utilizado para el desarrollo de aplicaciones en Java y otros lenguajes conocidos como C o C++.

A parte de la edición de código, el IDE nos ofrece un depurador para corregir errores de código, una interfaz fácil de utilizar y la posibilidad de integrar herramientas externas a parte de las que ya incluye por defecto.

1.6.2.2 Visual Studio

Visual Studio por su parte nos lo ofrece la empresa Microsoft y se trata de un entorno que también soporta gran cantidad de lenguajes. Aunque existen versiones de pago, encontramos la versión Visual Studio Community que es gratuita para proyectos de código abierto y pequeñas empresas.

Al igual que NetBeans, éste incluye muchas características, depurador y sobre todo una gran cantidad de extensiones que han hecho que Visual Studio se vuelva uno de los entornos más populares en la actualidad. Entre las extensiones más destacadas encontramos la llamada *IntelliCode*, que proporciona una recomendación de escritura inteligente al igual que lo hacía NetBeans.

1.6.3 Modelado 3D

1.6.3.1 Blender

Multiplataforma y con licencia GNU GPL, Blender [20] es un software gratuito y de código abierto orientado al modelado, iluminación, renderizado, animación y creación de gráficos tridimensionales. Es muy popular y también cuenta con una gran comunidad e información en internet que ayuda a iniciarse en su entorno.

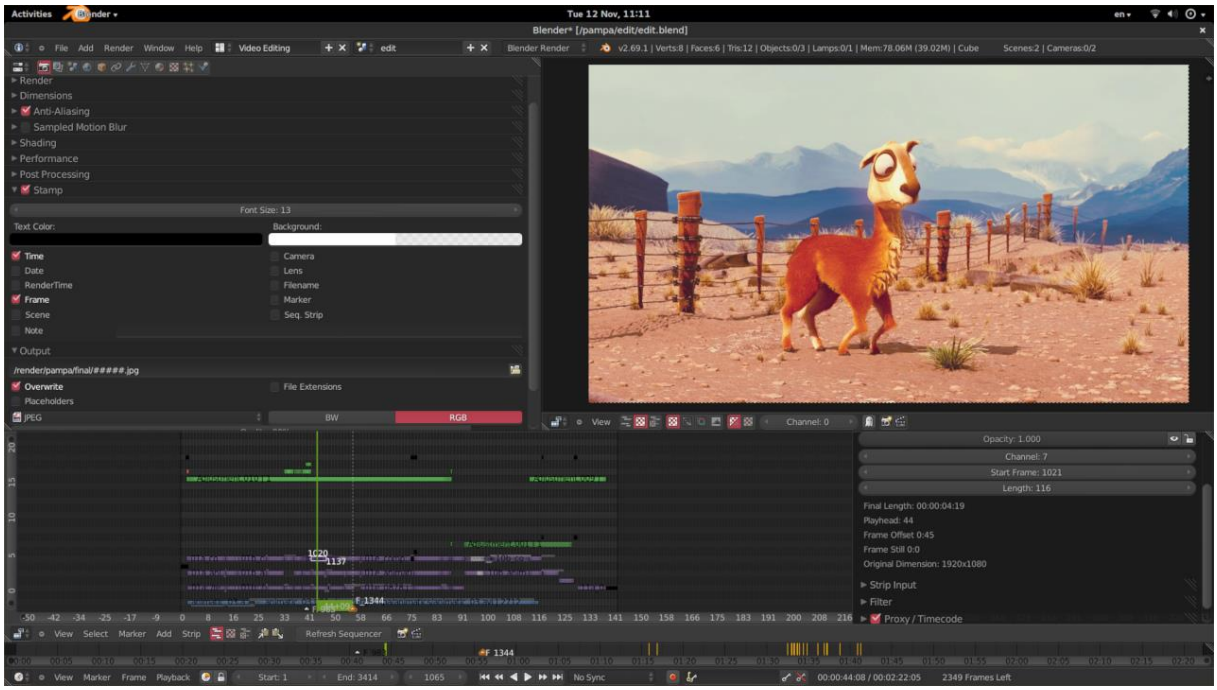

Ilustración 1.13: Entorno de Blender

No obstante, pese a tener una curva de aprendizaje menor que la que encontraremos en la otra opción analizada, Blender es una herramienta que necesita tiempo para aprender a manejarla, y cuya curva de aprendizaje no presume por ser precisamente baja.

1.6.3.2 Houdini 3D

Otro software que permite el modelado, simulación y animación es el conocido Houdini 3D. Houdini a diferencia de Blender, tiene carácter comercial y por tanto no es completamente gratuito, pero sí que contiene una versión gratuita, llamada Houdini Apprentice, que puede ser utilizada por estudiantes y artistas para un uso en proyectos no comerciales.

Es famoso por su utilización principalmente en entornos profesionales para la creación de efectos visuales dentro de la industria. En favor a Houdini 3D, la herramienta goza de unos procedimientos en general más avanzados que los que proporciona Blender, con la desventaja de tener una interfaz más compleja y difícil de usar a la que proporciona Blender.

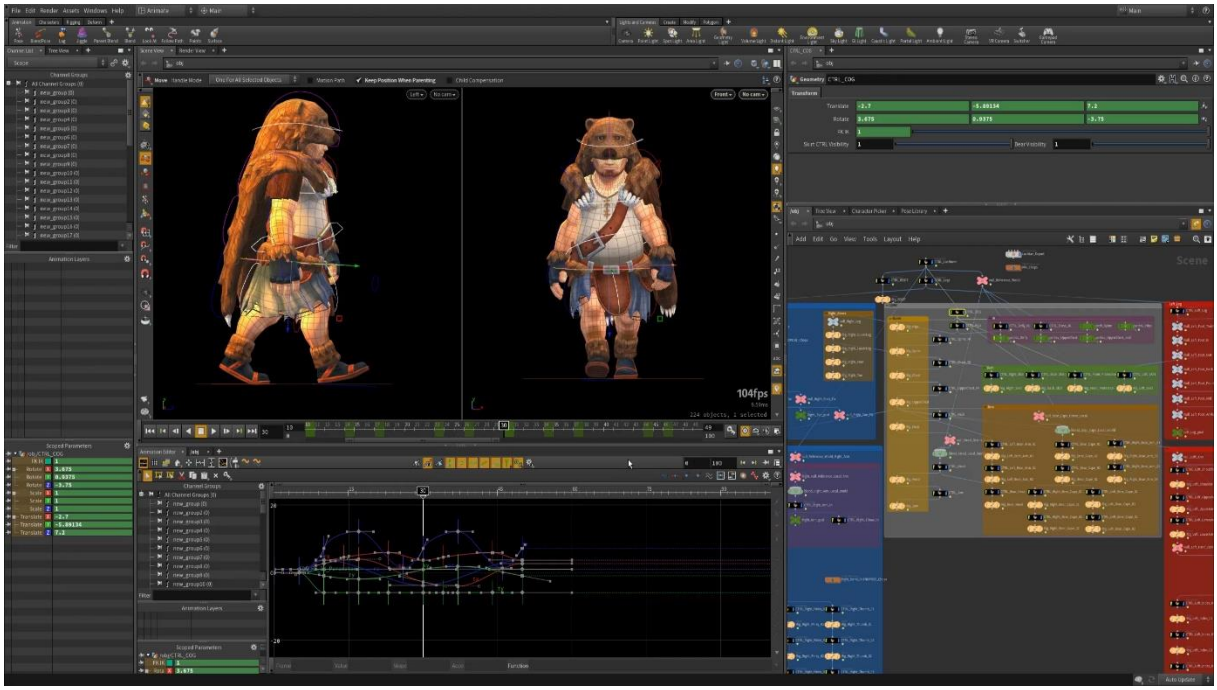

Ilustración 1.14: Entorno de Houdini 3D

1.7 Descripción de la solución propuesta

1.7.1 Solución propuesta para el motor

Para la solución del motor gráfico vamos a analizar los puntos a favor que encontramos entre los motores descritos para así poder determinar una elección fundamentada. Para ello se ha creado la siguiente tabla, Tabla 1.1, donde se muestra una serie de preguntas y respuestas que han ayudado a determinar la elección.

Cuestión	Unity	Unreal Engine
Experiencia inicial	Media	Nula
Curva de aprendizaje	Baja-Media	Media-Alta
Lenguaje de programación	C# y Java	C++ y Blueprints
Exigencia de recursos	Media	Alta
Tienda de recursos	Unity Asset Store [21]	Unreal Engine Marketplace
Resultados gráficos	Buenos	Muy buenos
Precio	Gratis (< 100K$ anual)	Gratis (< 3000$ trimestre)

Tabla 1.1: Balance para propuesta de motor

Analizando los puntos de la tabla anterior, es obvio que si queremos un resultado muy bueno como el que podemos encontrar en grandes y complejos juegos triple A, la opción idónea será hacer uso de Unreal Engine. Sin embargo, planteando la situación, lo más seguro es que en el momento actual del desarrollo y dado los recursos de los que partimos, no seamos capaces de exprimir a fondo las capacidades de Unreal Engine. Por otro lado, programar mediante Blueprint puede ser tedioso cuando el juego crece y usar C++, pese a ser muy buen lenguaje, requiere de un mayor esfuerzo que el que puede darnos C# o Java para la programación en Unity.

Teniendo en cuenta esto, se elegirá Unity frente a Unreal Engine, dado que se trata de un motor que el autor de este proyecto conoce y tiene cierta soltura de inicio. Por otro lado, este motor posee una curva de aprendizaje mucho menor y además dará menos problemas a la hora de trabajar debido a que éste exige menos recursos al ordenador.

1.7.2 Solución propuesta para el entorno de programación

Ciertamente esta decisión fue la más fácil de tomar una vez se determinó que se utilizaría el motor de Unity. Esto es así porque al instalar Unity se da la posibilidad de ser usado con Visual Studio, y de instalarlo automáticamente si no se tiene, permitiendo crear y mantener nuestros archivos de proyecto automáticamente desde dentro de Unity.

Es por esta integración de Visual Studio en Unity que nos decantamos por Visual Studio frente a NetBeans, el cual no permite tan fácilmente este funcionamiento conjunto.

1.7.3 Solución propuesta para el editor 3D

Para la elección del entorno 3D se ha seleccionado finalmente el software de Blender, pues, pese a ser menos potente que Houdini 3D, no se considera que se vaya a notar diferencia en cuanto a las posibilidades ofertadas (se utilizará para cosas muy básicas). Por otro lado, es importante tener en cuenta la curva de aprendizaje grande que se encuentra en ambas posibilidades, por lo que lo más conservador teniendo en cuenta el tiempo de desarrollo, es utilizar aquella opción que tenga una mayor facilidad de uso.

Otra ventaja de usar Blender dentro del proyecto es que el autor adquirió conocimientos básicos de la herramienta antes del comienzo del mismo, por lo que, aunque fuera necesario ampliar los saberes, no se partiría de cero si se elegía esta alternativa.

1.8 Alcance

Una vez definido el marco claro de trabajo, el alcance del proyecto incluirá los siguientes entregables:

1. **Proyecto de Unity**: cuyo contenido es un proyecto de Unity 3D en la versión de Unity 2021.3.15f1. Incluye un total de 4 escenas y una raíz de carpetas donde se encuentran todos los códigos, modelos, imágenes y todo el material utilizado para generar el videojuego. En el interior del proyecto se

localiza además un archivo git, desde el cual se puede revisar a fondo cómo ha sido la evolución del proyecto desde el inicio de éste.

2. **Ejecutable del juego**: cuyo contenido es una carpeta con la aplicación "Crazy Taxi" en forma de archivo .exe que no debe sacarse de la misma ya que hace uso de otras múltiples carpetas que son necesarias para la ejecución y un documento .csv con las traducciones. Dicho archivo es el ejecutable del juego completamente funcional, preparado para su utilización y pruebas. Tras iniciarlo nos llevará al menú principal del juego, donde podremos elegir el modo y personajes mediante su intuitiva interfaz para echar una partida.
3. **Documentación**: cuyo contenido es este documento, tratándose de una memoria donde se desarrollan todos los puntos principales del proyecto, explicados detalladamente.
4. **Manual de juego**: será un manual de controles con la misma información que el apartado Manuales de usuario, pero con más detalles, explicaciones rápidas y consejos a modo de guía, mostrada de manera mucho más visual, con objetivo de simular la apariencia de los antiguos libros de instrucciones que traían las cajas de los videojuegos.
5. **Vídeo**: mostrará brevemente el funcionamiento del juego para que aquellos que no puedan probarlo sí que puedan verlo en funcionamiento.

1.9 Tecnologías utilizadas

De manera agrupada, a lo largo de los siguientes subapartados se enumerarán y describirán brevemente las tecnologías y herramientas utilizadas para la realización de este trabajo de fin de grado (TFG).

1.9.1 Programación

- **Unity**: es el motor gráfico escogido donde se realizará el videojuego 3D.
- **Programación en C#**: es un lenguaje de programación orientada a objetos soportado por Unity. Se utiliza para la programación de los scripts del juego.

- **Visual Studio 2019**: es el IDE que utilizamos para programar en Unity los scripts en lenguaje C#.
- **SmartGit**: es una aplicación que permite gestionar de forma fácil y visual el sistema de control de versiones del proyecto.

1.9.2 Edición 2D y 3D

- **Blender**: es el software de modelado 3D que se utilizará para crear y modificar los objetos necesarios dentro del juego.
- **Photopea** [22]: es una alternativa gratuita y online a Adobe Photoshop (que requiere licencia). Se utiliza para edición de imágenes 2D durante la generación de *sprites*.
- **Paint**: es una herramienta de creación y edición 2D utilizada para creación rápida de elementos simples de la interfaz. Se ha utilizado en conjunto con Photopea debido a su increíble facilidad y rapidez de uso.

1.9.3 Documentación

- **Visual Paradigm**: es una herramienta utilizada para la creación de algunos diagramas de esta memoria. Cuenta también con una versión online.
- **Umlet**: es una aplicación para creación y edición de diagramas totalmente gratuita utilizada para generar algunos diagramas simples del proyecto. La razón principal de su uso es el diseño obtenido de los diagramas para el contexto en el que se incluyen y su facilidad de uso.
- **Creately** [23]: es una página web que se ha utilizado esporádicamente para la creación de algunas máquinas de estado encontradas en este documento. La razón principal de su uso es el diseño obtenido de los diagramas para el contexto en el que se incluyen.
- **Microsoft Word**: es el editor de texto utilizado para la creación de esta memoria.
- **Microsoft Excel** [24]: es la herramienta de hojas de cálculo para la creación de tablas y gráficos de esta memoria. También se utiliza para la creación y modificación del .csv de traducciones.

- **Google Drawings**: es una herramienta online de Google para hacer dibujos y exportarlos fácilmente a formatos de imagen. Se utiliza para hacer algunos esquemas durante el desarrollo de esta memoria.
- **Google Keep**: es una herramienta para anotaciones. Usada para apuntar ideas a lo largo del proyecto.
- **Mendeley Cite**: es una extensión para Microsoft Word (también tiene aplicación y extensión para Google Chrome) que hace de gestor de bibliografía. Nos facilita la inclusión de citaciones en este documento.

1.10 Metodología de desarrollo de software

La metodología es un punto clave dentro del desarrollo del proyecto y lo primero que se ha tenido que plantear es si usar una metodología tradicional o una metodología ágil.

Aunque existe una planificación temporal y unos requisitos iniciales en el proyecto para alcanzar los objetivos del mismo, es importante dejar claro que se llevará a cabo un proceso de experimentación para cada funcionalidad, donde se probarán diferentes ideas y se harán múltiples pruebas para precisar y validar éstas. Las funcionalidades donde destacamos esta primera fase de experimentación son:

- Generación procedural de ciudades.
- Sistema de navegación en ciudad.
- Inteligencia artificial de las personas NPC.
- Inteligencia artificial de los vehículos NPC.

Debido a la incertidumbre de partida de la que se parte en el proyecto, pese a tener una idea general clara de lo que se desarrollará, hay muchas cosas que de inicio están muy claras, pero otras no tanto. Es por ello que lo más indicado es decantarse por utilizar una metodología ágil, que es lo más habitual hoy en día en las empresas, pues ésta nos permitirá reaccionar y cambiar el rumbo del proyecto de forma rápida y eficiente en cualquier momento.

Una vez conocida nuestra situación de partida y la necesidad de una metodología ágil, el siguiente paso fue decidir cuál es la más idónea. Tras un periodo de reflexión se determinó que el proyecto se llevaría a cabo siguiendo la metodología Scrum [25] al ser la que mejor se adaptaba.

1.10.1 Metodología Scrum

El método Scrum es una metodología ágil que parte de la definición de 3 papeles fundamentales dentro del equipo:

- **Product Owner**: representa al cliente dentro del equipo de desarrollo y se responsabiliza de expresar las necesidades del cliente al resto del equipo.
- **Scrum Master**: es el moderador y ayuda al equipo de desarrollo a saber cuáles son las necesidades del cliente manifestadas por el Product Owner.
- **Development Team**: es el equipo de desarrollo.

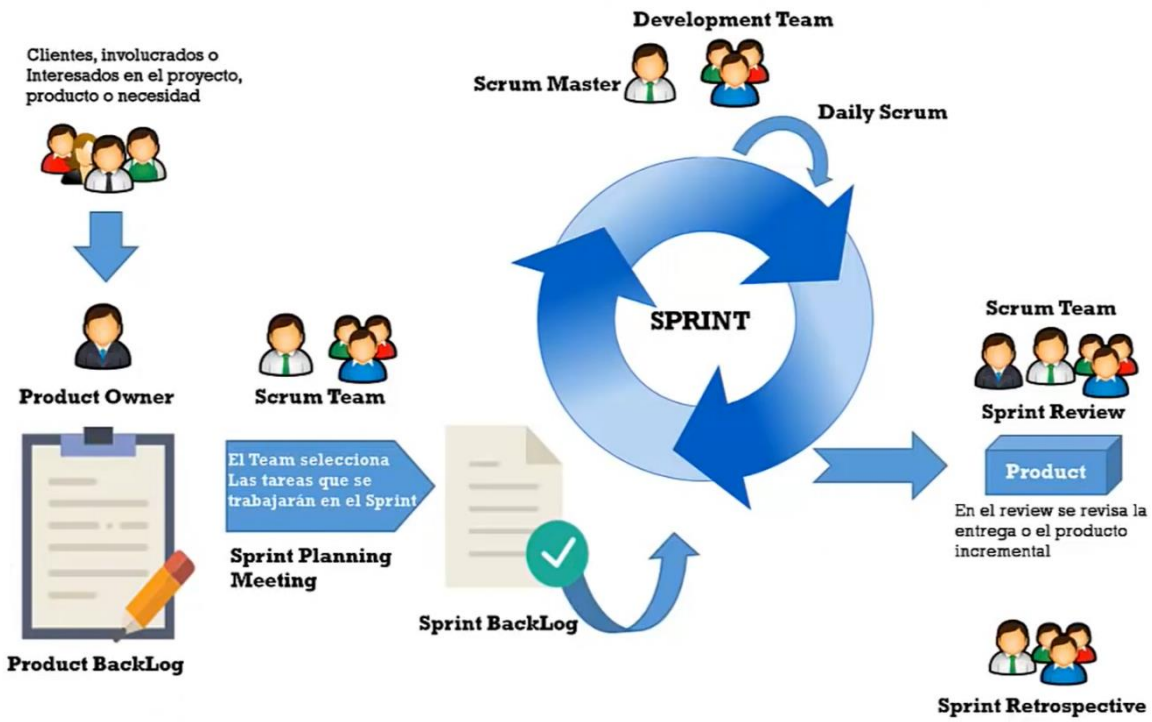

Ilustración 1.15: Ciclo de vida de proyecto Scrum

Adaptando la metodología, el autor del proyecto asumirá todos los roles dentro del equipo Scrum y el cliente del proyecto será también el alumno junto con el tutor del TFG, que realizarán frecuentemente tutorías tras cada iteración o sprint como

ahora veremos. Centrándonos en el desarrollo, éste se resume en las siguientes etapas fundamentales:

1. **Creación del Product BackLog**: es el escrito de un documento por parte del Product Owner donde se indican las tareas o requisitos a cumplir en base a las demandas del cliente. Esto en el proyecto se llevó a cabo durante la primera tutoría del curso con el tutor, día donde se definieron los grandes primeros requisitos del proyecto.
2. **Sprint Planning Meeting**: se manifiesta al equipo de desarrollo todo aquello que se requiere realizar indicado en el Product BackLog y se determina una serie de funcionalidades a realizar en un sprint que durará un periodo de entre 1 y 4 semanas.
3. **Creación del Sprint BackLog**: se crea un documento donde encontramos las funcionalidades que se deben realizar en el sprint y forman parte del conjunto del Product BackLog.
4. **Sprint**: se desarrollan las funcionalidades del Sprint BackLog en el tiempo establecido. Para el seguimiento correcto, se llevan a cabo reuniones breves donde se habla acerca del progreso y los problemas de desarrollo. En el proyecto esto último no se hará porque el autor posee todos los roles, pero sí que se desarrollará cada sprint en forma de iteración dentro del apartado 3 (Ejecución del desarrollo).
5. **Sprint Review**: se verifica en una reunión el cumplimiento de los objetivos del sprint. En el proyecto se realizará mediante una valoración del estado del sprint por parte del autor el día de finalización del sprint.
6. **Sprint Retrospective**: se presenta el producto funcional obtenido al cliente tras la realización del Sprint Review para así evaluarlo. En TFG como el cliente se trata del autor y el tutor, esta acción se realizará en tutoría cada vez que se termina un sprint.

Una vez finalizado el sprint se volverá al paso de Sprint Planning Meeting creando un ciclo que termina con el último sprint donde se finalizarán todas las funcionalidades del Product BackLog.

1.11 Estimación de tamaño y esfuerzo

Ya que el presente proyecto es un TFT, no existen restricciones de tipo económico, sino de tipo temporal (un número aproximado de horas). Por consiguiente, los cálculos de tamaño del proyecto están supeditados el tiempo disponible. En cuanto al esfuerzo, se dispone de tan un solo efectivo (la persona autora del trabajo).

Para llevar a cabo la estimación se hará uso de puntos de historia, es por ello que se ha creado una tabla con las historias de usuario iniciales, a las cuales se les asociarán puntuaciones en función del tiempo estimado que necesiten de desarrollo. Esta tabla primeriza, Tabla 1.2, se irá actualizando a lo largo del desarrollo conforme el proyecto vaya demandando.

Historia	Puntos de historia
Algoritmo procedural de ciudad	10
Vehículo jugador	2
Cámara personalizada	3
GPS con algoritmo A*	4
Flecha guía	1
Vehículos NPC normales	5
Vehículos NPC persecutores	4
Gestor de vehículos	2
Peatones NPC	10
Gestor de peatones	4
Características	1
Armas	3
Proyectiles	1
Interfaz principal	2
Menú principal	3
Menú de selección	4
Menú de pausa	1
Menú de fin de partida	1
Objeto de configuración	7
Modos de juego	3
Audio	2

TOTAL	**73**

Tabla 1.2: Estimación con puntos de historia

Para la estimación de los puntos de historia lo primero que se ha definido es un pivote, que es una historia con una puntuación que se utilizará de referencia para medir la puntuación del resto de historias. El pivote elegido ha sido la historia de "GPS con algoritmo A*" porque se considera que tiene una complejidad intermedia. A partir del pivote, cada una del resto de actividades ha sido evaluada en comparación con la complejidad del pivote y en consecuencia se le ha asignado un número de puntos de historia. Poniendo un ejemplo claro, la actividad de crear el vehículo del jugador se considera que tiene una complejidad de la mitad que la complejidad del pivote, por lo que se le han aplicado la mitad de los puntos del pivote, es decir, 2 puntos. Análogamente pasaría con el caso contrario, si se estima que una actividad es el doble de compleja se le asignarían el doble de puntos de historia.

Como es de esperar, esto es una estimación inicial y es posible que en algunos casos se haya sobreestimado o subestimado alguna historia, sin embargo, sí que nos permite repartir el trabajo durante los sprints de una forma coherente.

Asumiendo que la etapa de desarrollo se llevará a cabo entre los meses de octubre y abril sin tener en cuenta los periodos de exámenes, contamos con 5 meses de desarrollo. Usualmente los sprints son de poca duración y con frecuencia tienen la duración de una semana para equipos Scrum de 4 personas, sin embargo, el contexto de trabajo es un equipo de una única persona (autor del TFG), por lo que se necesitará 4 veces más de tiempo para llevar a cabo el sprint, dándonos aproximadamente la duración de un mes. Entonces, teniendo en cuenta una duración de un mes por sprint, se realizarán un total de 5 sprints y cada uno exigirá aproximadamente el desarrollo de 15 puntos de historia para terminar el proyecto (que cuenta en principio con un total de 73), algo que es asumible para el desarrollador.

1.12 Planificación temporal

Dado que el proyecto comenzó con su planteamiento en el mes de septiembre y se prevé que termine para el mes de mayo con la finalización de la documentación,

la planificación vendrá dada para organizar el tiempo en el periodo de 9 meses, que teniendo en cuenta las fechas de exámenes quedará reducido a unos 7 meses.

Se tendrá en cuenta en el diagrama de Gantt, que vemos en la Ilustración 1.16, los siguientes periodos:

- **Planteamiento**: incluye el tiempo transcurrido para llevar a cabo las primeras tutorías con el cliente (el autor y el tutor del proyecto) donde se especificaron las ideas y requisitos iniciales del proyecto.
- **Desarrollo**: incluye todo el proceso de desarrollo del videojuego, incluyendo tiempos de investigación, aprendizaje, programación, edición y pruebas parciales. Dado que el proyecto utiliza una metodología ágil y esta planificación fue ideada al inicio del proyecto, los detalles de las iteraciones realizadas para llevar a cabo el desarrollo planteado se desarrollarán en las secciones posteriores.
- **Pruebas finales**: es un periodo de varios días que está reservado para hacer pruebas con usuarios reales con objetivo de ajustar elementos dentro del juego para asegurar una buena jugabilidad.
- **Documentación**: es el tiempo estimado para redactar la memoria actual.

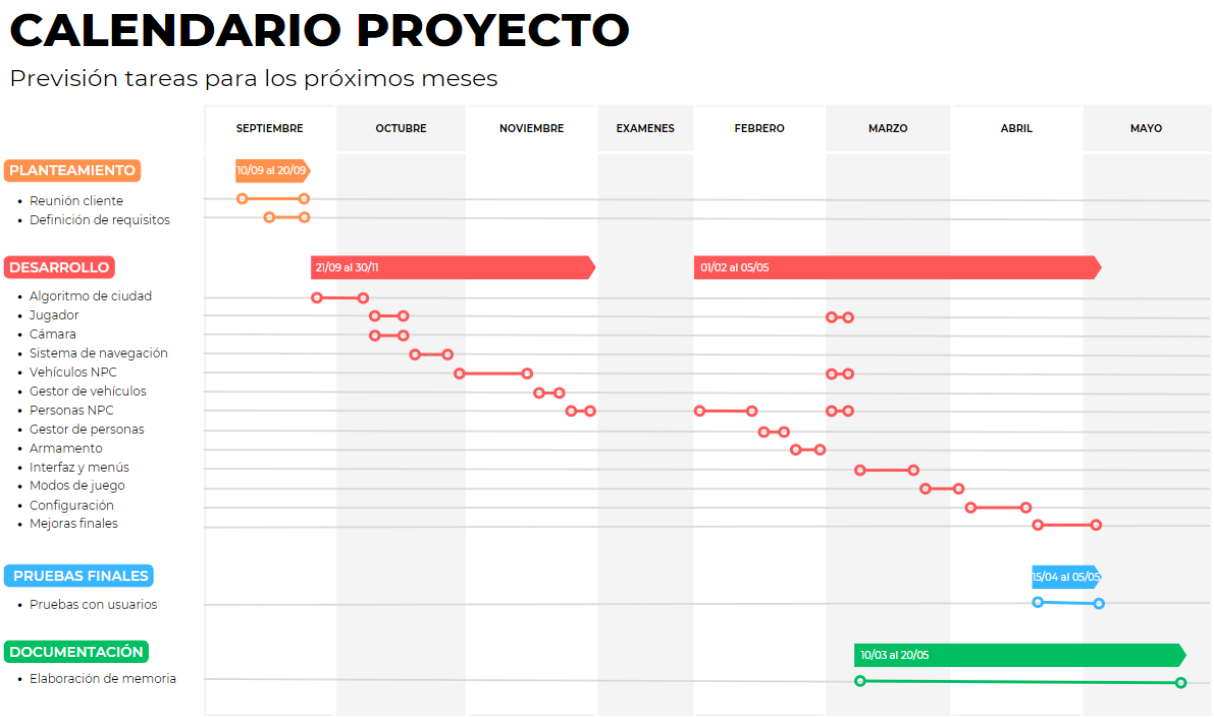

Ilustración 1.16: Planificación con diagrama de Gantt

1.13 Presupuesto

Teniendo en cuenta los resultados obtenidos en el apartado 1.11 (Estimación de tamaño y esfuerzo), procedemos a evaluar el presupuesto, desglosando por un lado el coste del material utilizado y por otro lado el coste del personal.

Material	Precio (€)	Amortización (€/mes)	Tiempo de uso (meses)	Coste (€)
Equipamiento hardware				
Portátil ASUS ROG STRIX G15	1299,99	21,66	7	151,66
Ratón Logitech G402	45,75	0,76	7	5,32
Pen drive SanDisk 64GB	9,65	0,16	7	1,12
Equipamiento software				
Windows 10 Home	144,99	2,42	7	16,94
Unity 3D	0	0	6	0
Visual Studio 2019	0	0	6	0
Microsoft Office	149,99	2,50	3	7,50
Visual Paradigm	0	0	3	0
Blender	0	0	6	0
SmartGit	0	0	6	0
Umlet	0	0	3	0
Paint	0	0	6	0
TOTAL				**182,54**

Tabla 1.3: Costes de material

Como vemos en Tabla 1.3, se ha tenido en cuenta para los cálculos la amortización parcial de los componentes hardware y software, asumiendo un periodo de amortización de 5 años.

Añadido a esto hay que sumar el coste del personal, para lo cual utilizaremos cantidades estimadas de salarios encontradas en la conocida página de InfoJobs [26]. Los cálculos han sido configurados por el número de horas y el rol que lo desempeña, a lo que hay que incluir un porcentaje adicional para estar dado de alta en la seguridad social (alrededor de un 30% del sueldo mensual del trabajador en bruto).

Tarea	Horas	Rol	Salario €/h	Seguridad social (€)	Coste (€)
Análisis y diseño	30	Analista	24,38	219,42	950,82
Investigación	40	Programador	20,32	243,84	1056,64
Programación	180	Programador	20,32	1097,28	4754,88
Documentación	40	Programador	20,32	243,84	1056,64
Testeo	10	Probador	17,83	53,49	231,79
TOTAL					**7099,95**

Tabla 1.4: Costes de personal

Para los cálculos no se han incluido las tareas de animación y modelado 3D porque se utilizarán recursos gratuitos de internet y por tanto no se requerirá de crear nada. Análogamente para el sonido y banda sonora.

Finalmente sumaremos el coste del material y del personal, incorporando un sobrecoste del 10% por los gastos indirectos.

Concepto	Coste (€)
Gastos en material	182,54
Gastos en personal	7099,95
Gastos indirectos	728,25
TOTAL GLOBAL	**8010,74**

Tabla 1.5: Resumen de costes

Por tanto, nuestro proyecto saldría por un precio de **8.010,74€**

2 DISEÑO INICIAL

Dentro de este capítulo se buscará definir el funcionamiento de la aplicación, así como sus requisitos iniciales, forjando mediante la etapa de análisis y diseño unas bases para la misma. El resultado de este proceso es conseguir un diseño preliminar que irá ajustándose en la secuencia de sprints que conforman el desarrollo.

2.1 Especificaciones del sistema

En este capítulo se presentarán la especificación detallada de los requisitos recopilados al inicio del proyecto a partir de las primeras tutorías con el cliente (autor y tutor del proyecto). Para definir los requisitos funcionales elaborados utilizaremos la plantilla dada por la Tabla 2.1:

Identificador: Título	
Título	Nombre descriptivo
Descripción	Detalle del requisito e información adicional
Prioridad	Alta/Media/Baja

Tabla 2.1: Plantilla requisitos

2.1.1 Extracción de requisitos funcionales

Este tipo de requisitos son los que describen como debe comportarse el sistema, indicando que funciones y acciones debe ser capaz de realizar. El proyecto iniciará con los siguientes requisitos funcionales:

RF-01: Jugar partida	
Título	Jugar partida
Descripción	Se podrá iniciar una partida en el modo elegido y llevarla a cabo. Debe haberse indicado anteriormente el modo y el vehículo a utilizar
Prioridad	Alta

Tabla 2.2: RF-01

RF-02: Indicar modo	
Título	Indicar modo
Descripción	El jugador podrá indicar el modo de juego desde un menú. Hay modos normales y modos especiales
Prioridad	Alta

Tabla 2.3: RF-02

RF-03: Seleccionar vehículo y conductor	
Título	Seleccionar vehículo y conductor
Descripción	Se podrá elegir vehículo y conductor desde un menú
Prioridad	Media

Tabla 2.4: RF-03

RF-04: Vehículos diferentes	
Título	Vehículos diferentes
Descripción	Habrá diferentes tipos de vehículos para el jugador con diferentes características, tales como la velocidad, aceleración, peso… Dicha información será visible para el jugador
Prioridad	Baja

Tabla 2.5: RF-04

RF-05: Atribuir características a los objetos	
Título	Atribuir características a los objetos
Descripción	Los objetos tendrán una serie de atributos y cualidades con los que diferenciarse de otros objetos, tales como por ejemplo la cantidad de vida
Prioridad	Media

Tabla 2.6: RF-05

RF-06: Configurar	
Título	Configurar
Descripción	El jugador podrá ajustar variables del juego desde un menú tales como el volumen, idioma y dificultad
Prioridad	Baja

Tabla 2.7: RF-06

RF-07: Pausar	
Título	Pausar
Descripción	Se podrá pausar una partida una vez haya comenzado
Prioridad	Media

Tabla 2.8: RF-07

RF-08: Reanudar	
Título	Reanudar
Descripción	Se podrá continuar la partida pausada
Prioridad	Alta

Tabla 2.9: RF-08

RF-09: Reiniciar	
Título	Reiniciar
Descripción	Se podrá reiniciar la partida desde el menú de pausa
Prioridad	Baja

Tabla 2.10: RF-09

RF-10: Abandonar	
Título	Abandonar
Descripción	Se podrá volver al menú principal desde el menú de pausa
Prioridad	Baja

Tabla 2.11: RF-10

RF-11: Generar ciudad	
Título	Generar ciudad
Descripción	El sistema generará un mapa diferente en cada partida justo antes de iniciarla. Lo hará de manera procedural
Prioridad	Alta

Tabla 2.12: RF-11

RF-12: Conducir	
Título	Conducir
Descripción	Se podrá conducir el vehículo por el mapa una vez la partida haya comenzado y hasta que ésta termine
Prioridad	Alta

Tabla 2.13: RF-12

RF-13: Disparar misiles	
Título	Disparar misiles
Descripción	El vehículo podrá disparar misiles al punto indicado si el modo de juego se lo permite
Prioridad	Baja

Tabla 2.14: RF-13

RF-14: Daño por impacto	
Título	Daño por impacto
Descripción	Los proyectiles bajarán la vida a los objetos que colisionan. Pueden ser explosivos y hacer daño de área
Prioridad	Baja

Tabla 2.15: RF-14

RF-15: Cambiar cámara	
Título	Cambiar cámara
Descripción	El jugador tendrá una cámara y podrá moverla de ubicación si el jugador está en mitad de una partida no pausada. Esto permitirá poder tener cámara de primera y tercera persona al menos
Prioridad	Baja

Tabla 2.16: RF-15

RF-16: Mostrar información de partida	
Título	Mostrar información de partida
Descripción	El jugador deberá poder ver en todo momento elementos como la puntuación, la vida, el tiempo restante de partida y un minimapa
Prioridad	Media

Tabla 2.17: RF-16

RF-17: Vehículos NPC	
Título	Vehículos NPC
Descripción	Habrá vehículos circulando por la ciudad de manera aleatoria (vehículos comunes) o persiguiendo al jugador o a algún otro objetivo (vehículos de policía)
Prioridad	Alta

Tabla 2.18: RF-17

RF-18: Personas NPC	
Título	Personas NPC
Descripción	Habrá personas por las aceras de las calles que realizarán diferentes actividades asociadas a formas de comportamientos concretos. Deben poder cambiar el tipo de comportamiento en tiempo de ejecución
Prioridad	Alta

Tabla 2.19: RF-18

RF-19: Interactuar con NPCs	
Título	Interactuar con NPCs
Descripción	Durante una partida se podrá interactuar con los peatones NPC de la ciudad, haciendo que suban y bajen del vehículo
Prioridad	Alta

Tabla 2.20: RF-19

RF-20: Sumar puntuación	
Título	Sumar puntuación
Descripción	Se dará dinero al jugador por el transporte de peatones dentro de la partida y por bonus. Si no se consigue llevar a tiempo a un peatón éste abandona el vehículo sin pagar
Prioridad	Alta

Tabla 2.21: RF-20

RF-21: Indicar camino	
Título	Indicar camino
Descripción	El sistema indicará al jugador mediante una flecha qué calle debe tomar para ir al destino del pasajero cuando

	suba un cliente al taxi dentro de una partida. Esta mecánica también deben tenerla los vehículos que persigan al jugador
Prioridad	Alta

Tabla 2.22: RF-21

RF-22: Ver resultados	
Título	Ver resultados
Descripción	El jugador podrá ver el resultado de la partida al acabarla, mostrando información como su imagen, nombre, vehículo utilizado, puntuación obtenida…
Prioridad	Media

Tabla 2.23: RF-22

2.1.2 Extracción de requisitos no funcionales

Este tipo de requisitos definen los estándares de calidad software, refiriéndose a temas como la escalabilidad, mantenibilidad, seguridad… En nuestro proyecto los requisitos no funcionales impuestos son:

RNF-01: Rendimiento	
Título	Rendimiento
Descripción	El juego debe tener un rendimiento óptimo que permita una jugabilidad agradable y sin retardos. Para ello se requiere que el cálculo de rutas sea extremadamente rápido y que la gestión de NPCs esté optimizada
Prioridad	Alta

Tabla 2.24: RNF-01

RNF-02: Interfaz intuitiva	
Título	Interfaz intuitiva
Descripción	La interfaz y menús de la aplicación deben ser simples, que no agobien al jugador, y tan intuitivos que no requieran de explicaciones para aprender a usarlos. El usuario no deberá perderse en el flujo de navegación entre menús.
Prioridad	Alta

Tabla 2.25: RNF-02

RNF-03: Consistencia	
Título	Consistencia
Descripción	El producto debe estar altamente depurado para evitar encontrar errores, pues esto puede ser frustrante para el jugador sobre todo cuando le perjudican
Prioridad	Baja

Tabla 2.26: RNF-03

RNF-04: Flexibilidad	
Título	Flexibilidad
Descripción	La aplicación debe ser capaz de ajustarse a diferentes resoluciones de pantalla
Prioridad	Media

Tabla 2.27: RNF-04

RNF-05: Portabilidad	
Título	Portabilidad
Descripción	Se debe poder ejecutar el videojuego en cualquier ordenador que cumpla los requisitos mínimos
Prioridad	Alta

Tabla 2.28: RNF-05

RNF-06: Retroalimentación	
Título	Retroalimentación
Descripción	El juego mostrará una retroalimentación clara mediante colores, sonidos o animaciones para que el jugador sepa con certeza que hubo un cambio
Prioridad	Media

Tabla 2.29: RNF-06

RNF-07: Internacionalización	
Título	Internacionalización
Descripción	El juego debe estar preparado para su uso en varios idiomas
Prioridad	Alta

Tabla 2.30: RNF-07

2.2 Análisis y diseño del sistema

2.2.1 Diagrama de casos de uso

Para la definición del diagrama de casos de uso y para el resto del proyecto tendremos en cuenta 2 actores, dados por las tablas Tabla 2.31 y Tabla 2.32.

A-01: Jugador	
Tipo	Actor principal
Descripción	Persona que juega al videojuego y controla los mandos
Interacción	Directamente sobre el juego

Tabla 2.31: Actor jugador

A-02: Sistema	
Tipo	Actor principal
Descripción	Sistema que gestiona el mundo, NPCs, eventos…
Interacción	Directamente sobre el juego

Tabla 2.32: Actor sistema

En la Ilustración 2.1 se representa el diagrama de casos de uso. Dado que la condición para los casos adicionales es siempre “si el usuario lo desea”, se han omitido los comentarios que indican dicha condición para acceder a las relaciones de extensión <<*Extend*>> con objeto de simplificar el diagrama y evitar redundancia.

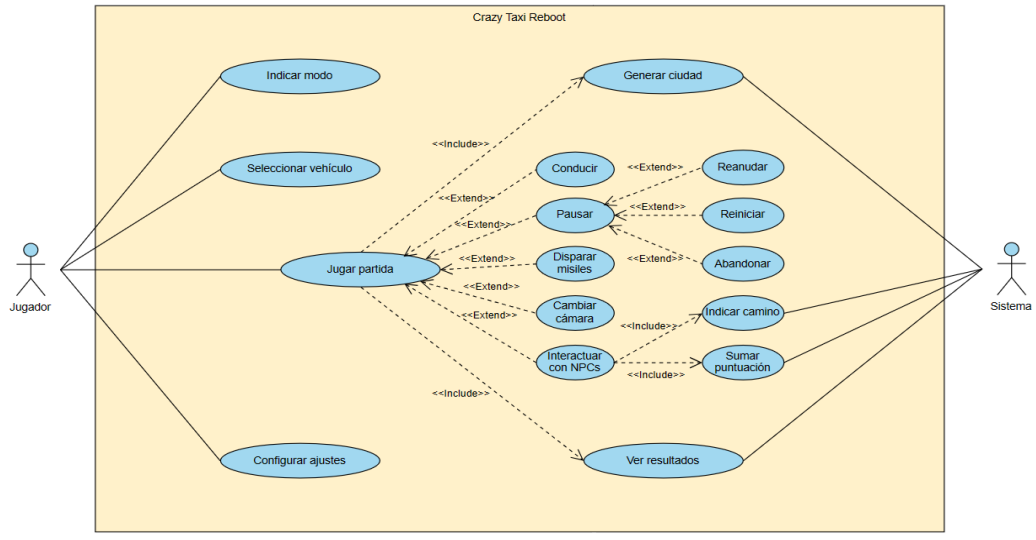

Ilustración 2.1: Diagrama de casos de uso

2.2.2 Casos de uso

Para desarrollar los casos de uso elaborados utilizaremos la plantilla dada en Tabla 2.33:

Identificador: Título	
Título	Nombre descriptivo
Actor primario	Entidad que realiza el caso
Sistema	Entorno donde se realiza la acción
Nivel	Objetivo de jugador u objetivo de sistema
Precondición	Condiciones que deben haberse dado con anterioridad
Flujo normal	Secuencia de pasos común de ejecución del caso
Flujos alternativos	Secuencia de pasos alternativa de ejecución del caso

Tabla 2.33: Plantilla de casos de uso

Teniendo en cuenta el diagrama de casos de uso definido en y conociendo los requisitos iniciales, se han creado los siguientes casos de uso:

CU-01: Jugar partida	
Título	Jugar partida
Actor primario	Jugador
Sistema	Crazy Taxi Reboot
Nivel	Objetivo de jugador
Precondición	Debe haberse seleccionado el modo de juego y el vehículo
Flujo normal	
1	El jugador presiona el botón de “Jugar”
2	El sistema carga la partida con las configuraciones indicadas, el modo de juego, el vehículo y conductor elegidos

Tabla 2.34: CU-01

CU-02: Indicar modo	
Título	Indicar modo
Actor primario	Jugador
Sistema	Crazy Taxi Reboot
Nivel	Objetivo de jugador

Precondición	El jugador debe encontrarse en el menú principal
Flujo normal	
1	El jugador presiona el botón de selección de modo “Arcade”
2	El sistema actualiza la pantalla mediante una animación y muestra los modos disponibles según el botón pulsado
3	El jugador elige un modo de juego
4	El jugador pulsa el botón correspondiente al modo elegido
5	El sistema guarda el modo elegido y configura la siguiente partida con dicho modo
6	El sistema carga el menú de selección de vehículos y conductor actualizando la pantalla mediante una animación
Flujos alternativos	
1.A	El jugador presiona el botón de selección de modo “Crazy Box”

Tabla 2.35: CU-02

CU-03: Seleccionar vehículo	
Título	Seleccionar vehículo
Actor primario	Jugador
Sistema	Crazy Taxi Reboot
Nivel	Objetivo de jugador
Precondición	El jugador debe haber seleccionado un modo de juego
Flujo normal	
1	El sistema carga el menú de selección de personaje actualizando la pantalla mediante una animación
2	El jugador elige el personaje que desea como conductor
3	El sistema carga el menú de selección de vehículo actualizando la pantalla mediante una animación
4	El jugador elige el vehículo
5	El jugador mira las características del vehículo
6	El jugador acepta el vehículo seleccionado
7	El sistema guarda el vehículo y conductor seleccionados y configura la partida siguiente con ese conjunto
Flujos alternativos	
6.A	El jugador puede preferir otro vehículo y volver a 4

Tabla 2.36: CU-03

CU-04: Configurar ajustes	
Título	Configurar ajustes
Actor primario	Jugador
Sistema	Crazy Taxi Reboot
Nivel	Objetivo de jugador
Precondición	El jugador debe estar en el menú principal
Flujo normal	
1	El jugador presiona el botón de selección “Ajustes”
2	El sistema actualiza la pantalla mediante una animación y muestra el menú de configuración
3	El jugador cambia el idioma
4	El jugador acepta los cambios dándole al botón “Aceptar”
5	El sistema guarda los cambios y carga el menú principal mediante la actualización con una animación
Flujos alternativos	
3.A	El jugador cambia el volumen de la aplicación
3.B	El jugador cambia la calidad gráfica
3.C	El jugador cambia los valores de generación procedural
4.A	El jugador hace más cambios, por lo que vuelve a 3

Tabla 2.37: CU-04

CU-05: Pausar	
Título	Pausar
Actor primario	Jugador
Sistema	Crazy Taxi Reboot
Nivel	Objetivo de jugador
Precondición	El jugador tiene que estar en medio de una partida
Flujo normal	
1	El jugador presiona el control específico para activar la pausa
2	El sistema registra la entrada y muestra el menú de pausa deteniendo el tiempo del juego

Tabla 2.38: CU-05

CU-06: Reanudar	
Título	Reanudar

Actor primario	Jugador
Sistema	Crazy Taxi Reboot
Nivel	Objetivo de jugador
Precondición	El jugador tiene que estar en medio de una partida pausada
Flujo normal	
1	El jugador pulsa el botón “Reanudar”
2	El sistema registra la petición y reestablece la partida en el punto en el que se quedó
Flujos alternativos	
1.A	El jugador presiona el control específico

Tabla 2.39: CU-06

CU-07: Reiniciar	
Título	Reiniciar
Actor primario	Jugador
Sistema	Crazy Taxi Reboot
Nivel	Objetivo de jugador
Precondición	El jugador tiene que estar en medio de una partida pausada
Flujo normal	
1	El jugador pulsa el botón “Reiniciar”
2	El sistema registra la petición y recarga la escena de juego con la misma configuración que la actual

Tabla 2.40: CU-07

CU-08: Abandonar	
Título	Abandonar
Actor primario	Jugador
Sistema	Crazy Taxi Reboot
Nivel	Objetivo de jugador
Precondición	El jugador tiene que estar en medio de una partida pausada
Flujo normal	
1	El jugador pulsa el botón “Abandonar”
2	El sistema registra la petición y carga la escena del menú principal

Tabla 2.41: CU-08

CU-09: Generar ciudad	
Título	Generar ciudad
Actor primario	Sistema
Sistema	Crazy Taxi Reboot
Nivel	Objetivo de sistema
Precondición	El jugador tiene que haber iniciado una partida
Flujo normal	
1	El sistema llama al algoritmo de generación procedural de ciudades

Tabla 2.42: CU-09

CU-10: Conducir	
Título	Conducir
Actor primario	Jugador
Sistema	Crazy Taxi Reboot
Nivel	Objetivo de jugador
Precondición	El jugador tiene que estar en mitad de una partida no pausada
Flujo normal	
1	El jugador no indica movimientos
2	El sistema actualiza la posición del jugador en función del movimiento indicado. Volver a 1
Flujos alternativos	
1.A	El jugador indica movimiento de aceleración
1.B	El jugador indica movimiento de frenado
1.C	El jugador indica movimiento de giro
1.D	El jugador indica movimiento de derrape

Tabla 2.43: CU-10

CU-11: Disparar misiles	
Título	Disparar misiles
Actor primario	Jugador
Sistema	Crazy Taxi Reboot
Nivel	Objetivo de jugador
Precondición	El jugador tiene que estar en mitad de una partida no pausada
Flujo normal	

1	El jugador apunta con la mirilla del cursor al objetivo y presiona el control de disparo
2	Sistema detecta la posición de la mirilla y la entrada necesaria para el disparo y genera un misil que envía a dicha posición

Tabla 2.44: CU-11

CU-12: Cambiar cámara	
Título	Cambiar cámara
Actor primario	Jugador
Sistema	Crazy Taxi Reboot
Nivel	Objetivo de jugador
Precondición	El jugador tiene que estar en mitad de una partida no pausada
Flujo normal	
1	El jugador presiona el control para mover la cámara
2	El sistema cambia a cámara de primera persona
Flujos alternativos	
2.A	El sistema cambia a cámara de tercera persona
2.B	El sistema cambia a cámara trasera

Tabla 2.45: CU-12

CU-13: Interactuar con NPCs	
Título	Interactuar con NPCs
Actor primario	Jugador
Sistema	Crazy Taxi Reboot
Nivel	Objetivo de jugador
Precondición	El jugador tiene que estar en mitad de una partida no pausada
Flujo normal	
1	El jugador se acerca a un peatón NPC
2	El sistema activa la retroalimentación correspondiente

Tabla 2.46: CU-13

CU-14: Sumar puntuación	
Título	Sumar puntuación
Actor primario	Sistema
Sistema	Crazy Taxi Reboot

Nivel	Objetivo de sistema
Precondición	El jugador tiene que estar en mitad de una partida no pausada
Flujo normal	
1	El jugador completa un trayecto con pasajero
2	El sistema pondera la calidad del trayecto o bonus y multiplica su valor
3	El sistema suma dinero al marcador de puntuación y realiza un refresco del marcador
Flujos alternativos	
1.A	El jugador completa un bonus

Tabla 2.47: CU-14

CU-15: Indicar camino	
Título	Indicar camino
Actor primario	Sistema
Sistema	Crazy Taxi Reboot
Nivel	Objetivo de sistema
Precondición	El jugador tiene que estar en mitad de una partida no pausada y debe llevar un cliente en el vehículo
Flujo normal	
1	El jugador lleva en el vehículo a un cliente
2	El sistema calcula la ruta más corta y detecta cual es la calle que debe tomar el jugador para llegar al destino del cliente
3	El sistema lleva a cabo una retroalimentación girando la flecha y vuelve a 2

Tabla 2.48: CU-15

CU-16: Ver resultados	
Título	Ver resultados
Actor primario	Jugador
Sistema	Crazy Taxi Reboot
Nivel	Objetivo de sistema
Precondición	El jugador debe haber terminado la partida
Flujo normal	

1	El sistema recopila la puntuación, configuración y elementos de interés y los muestra por pantalla junto con un botón para continuar
2	El jugador mira sus resultados
3	El jugador pulsa el botón de continuar
4	El sistema carga el menú principal

Tabla 2.49: CU-16

2.2.3 Diseño arquitectónico

A continuación en la Ilustración 2.2 se va a representar un UML simplificado (sin atributos ni funciones) donde se esquematiza la forma general del proyecto que se espera desarrollar. El diagrama sufrirá cambios a lo largo del proyecto debido a la incertidumbre de la que se parte en algunos aspectos como por ejemplo el comportamiento de los peatones no clientes. Aun así, proporciona una base muy sólida para comenzar el desarrollo.

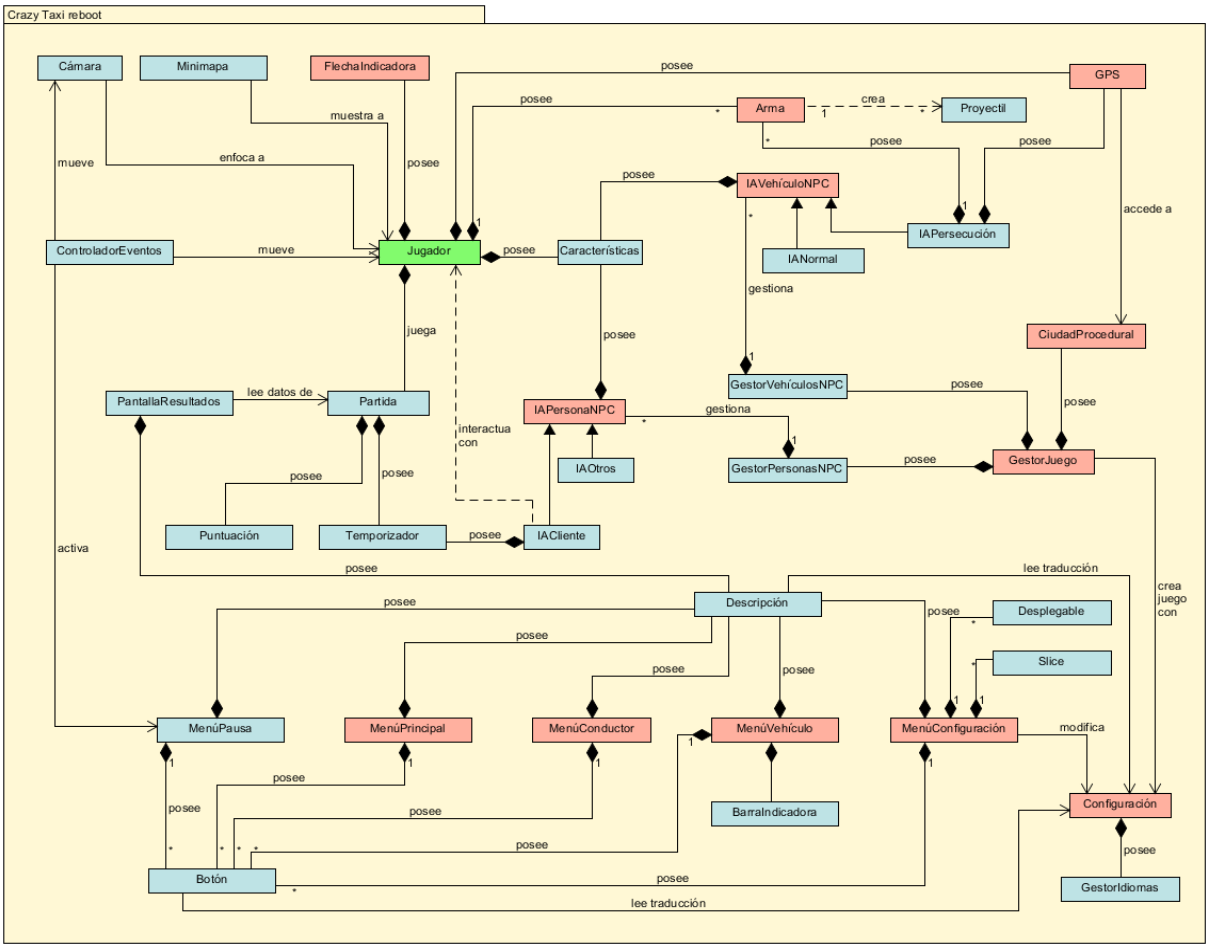

Ilustración 2.2: Visión general de UML simplificado inicial

Como se puede observar es un proyecto muy ambicioso desde el inicio, y para ayudar a la lectura del UML se han destacado en rojo las clases más importantes que se esperan desarrollar, en azul se han dejado aquellas consideradas más a nivel de apoyo y en verde tenemos al jugador que es, como es obvio, el objeto clave del proyecto.

2.3 Diseño de la interfaz

2.3.1 Sketches

La interfaz de nuestro juego la haremos mediante inspiración de múltiples juegos aprovechando que no nos cerramos a hacer un remake, sino que hacemos un reboot. Aun así, sí que respetaremos el flujo de menús del Crazy Taxi original.

El menú principal de nuestro juego estará inspirado en el menú de Crazy Taxi, visible en la Ilustración 2.3.

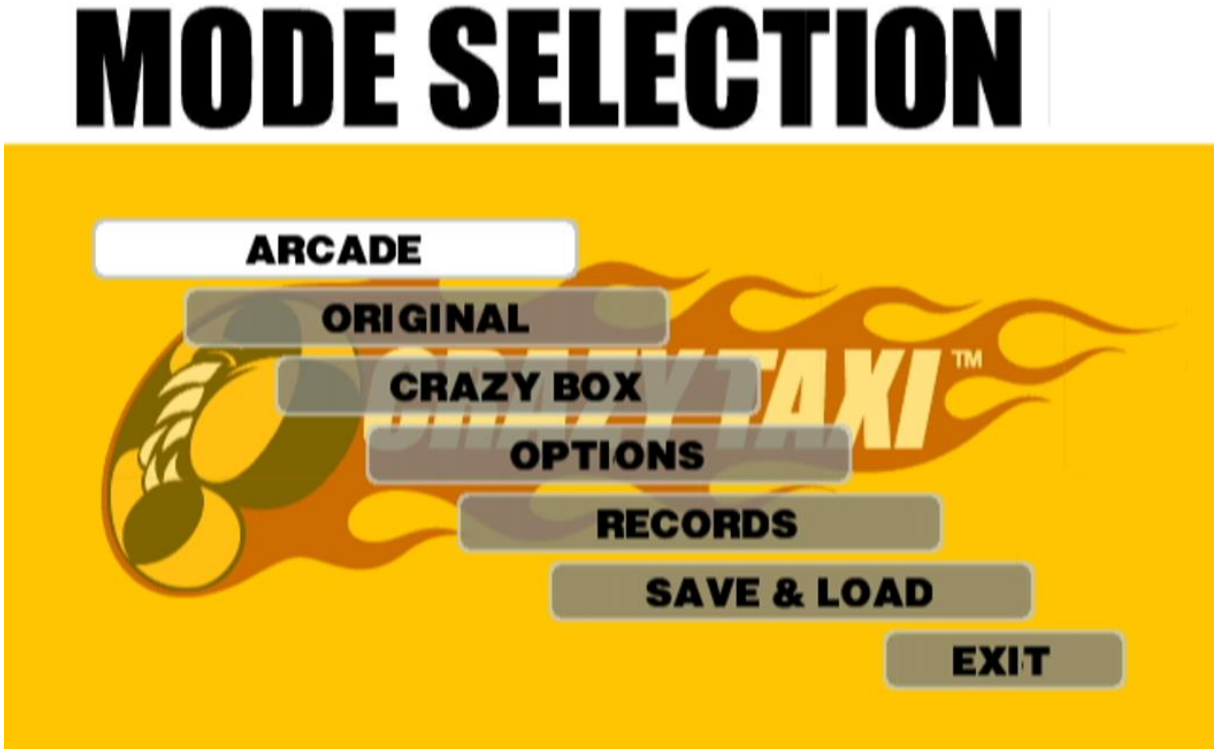

Ilustración 2.3: Menú principal Crazy Taxi original

Para adaptarlo a nuestro juego, que como es de esperar tendrá unas pocas menos de opciones y funcionalidades que el Crazy Taxi de SEGA, haremos un

pequeño sketch donde nos haremos una idea de cómo será el menú, ubicación de botones y animaciones. Concretamente nosotros solo tendremos el botón “Arcade”, el botón “Crazy Box, el botón de “Ajustes” y un último botón para salir. El sketch diseñado es el mostrado en Ilustración 2.4.

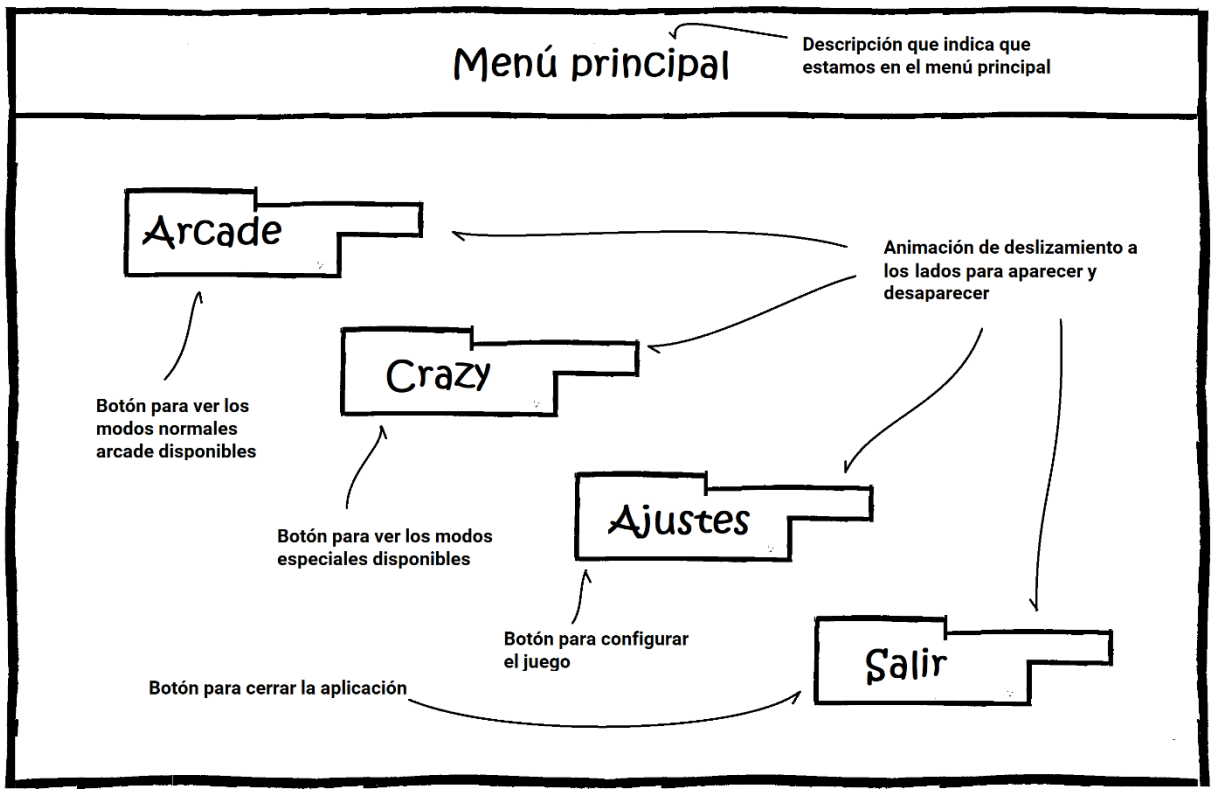

Ilustración 2.4: Sketch de menú principal Crazy Taxi reboot

El diseño de los menús a los que llevan los botones “Arcade” y “Crazy” (menús con modos de juego normales o especiales respectivamente) será el mismo, solo que los botones tendrán textos distintos y nos enviarán a zonas diferentes de la aplicación. En lugar de haber un botón para salir debemos tener un botón para volver. Nótese el detalle de la importancia que se le da y se le dará a la descripción superior en todos los menús para asegurar que el jugador no se pierde con la navegación y sabe en todo momento qué tiene que hacer.

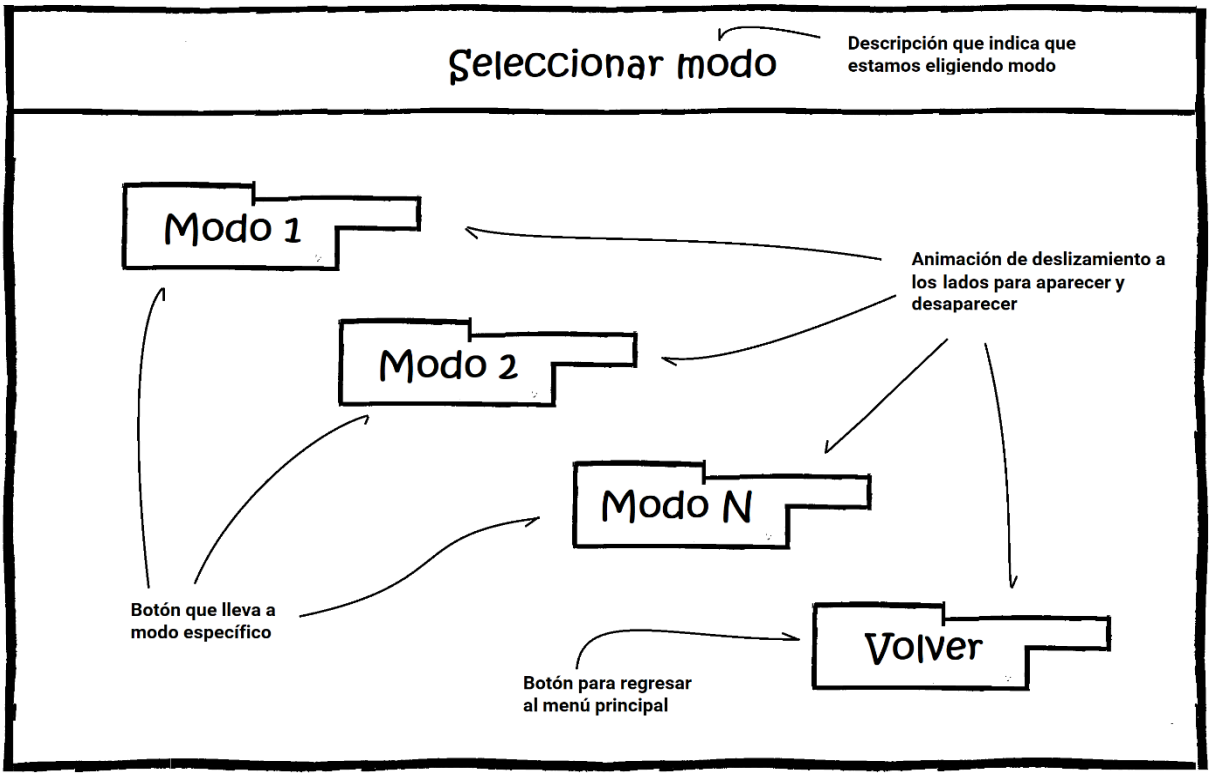

Ilustración 2.5: Sketch de menú selección de modos Crazy Taxi reboot

Una vez tenemos el diseño inicial del menú principal, lo siguiente que nos preocupa es hacer el menú de selección de vehículos y conductores. El menú de Crazy Taxi original es el que se muestra en la Ilustración 2.6.

Ilustración 2.6: Menú de selección de vehículo y jugador de Crazy Taxi original

Aunque si repasamos las ideas que quedan reflejadas en los requisitos, se demanda la posibilidad de seleccionar tanto el vehículo como el conductor de manera separada, cosa que en el juego de Crazy original no es posible porque cada conductor tiene su propio coche. Es por tanto que no utilizaremos esta interfaz, lo que nos deja con la opción de crearla desde cero o inspirar la interfaz en la de otro juego conocido. En este caso nos inspiraremos en otro juego porque conlleva la ventaja de que el jugador tenga que realizar un menor esfuerzo para comprenderla, aprovechando de alguna manera el conocimiento que ya tiene de haber jugado a otros juegos. Para que funcione esta lógica debemos buscar una interfaz que encaje con el estándar de los usuarios. Tras un periodo de debate, nos decantaremos por elegir la interfaz de Mario Kart [27], que es un juego que casi todo el mundo conoce, cuya interfaz es muy aceptada por los jugadores y permite seleccionar vehículos y conductores por separado.

Ilustración 2.7: Menú de selección de personaje de Mario Kart 8

En la imagen Ilustración 2.7 vemos el menú de selección de personajes de Mario Kart 8. Es interesante porque de un vistazo vemos todos los personajes, que son bastantes, y pinchando sobre la cara del que se desea utilizar se configura el conductor. El sketch diseñado para este caso es el visible en Ilustración 2.8.

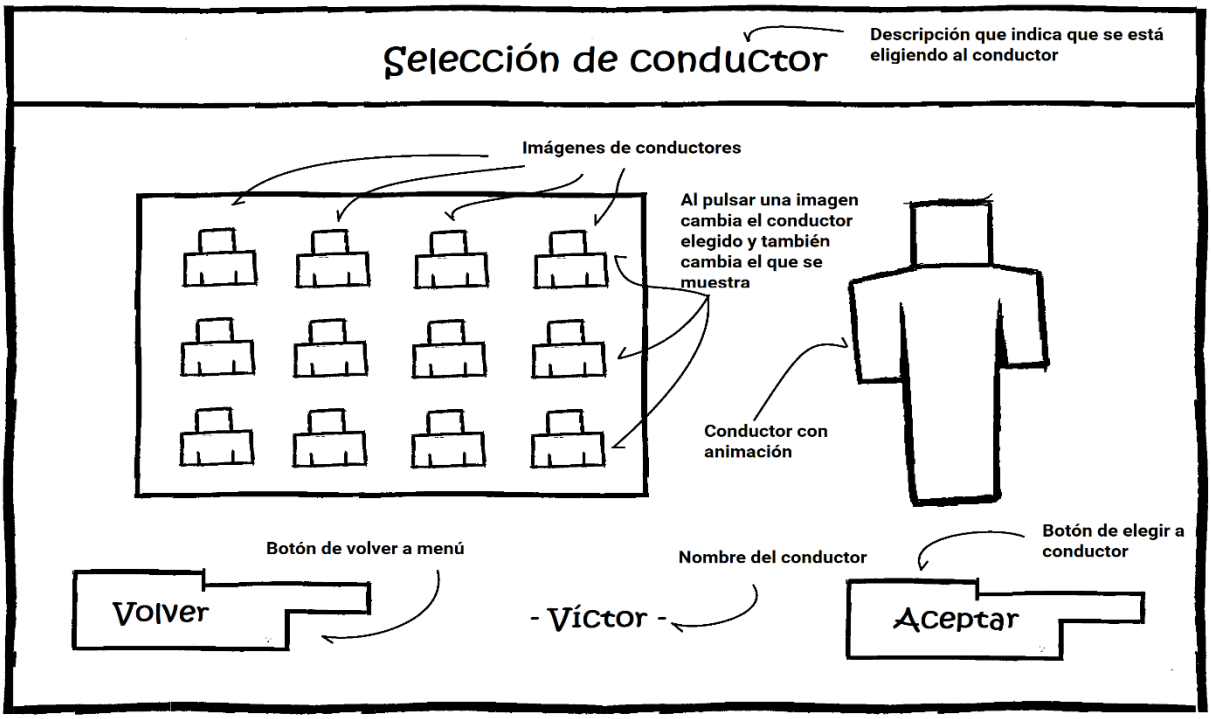

Ilustración 2.8: Sketch de menú de selección de personaje Crazy Taxi reboot

Por otro lado, tenemos la Ilustración 2.9, que muestra el menú de selección de vehículos. El menú de selección de vehículos es interesante porque muestra las características de los vehículos, y teniendo en cuenta que nuestro vehículo también tendrá características y atributos propios, será muy conveniente utilizar este menú a modo de inspiración.

Ilustración 2.9: Menú de selección de vehículo de Mario Kart 8

El sketch para la selección de vehículo es el de la Ilustración 2.10, que como vemos, tiene un panel con las características del vehículo.

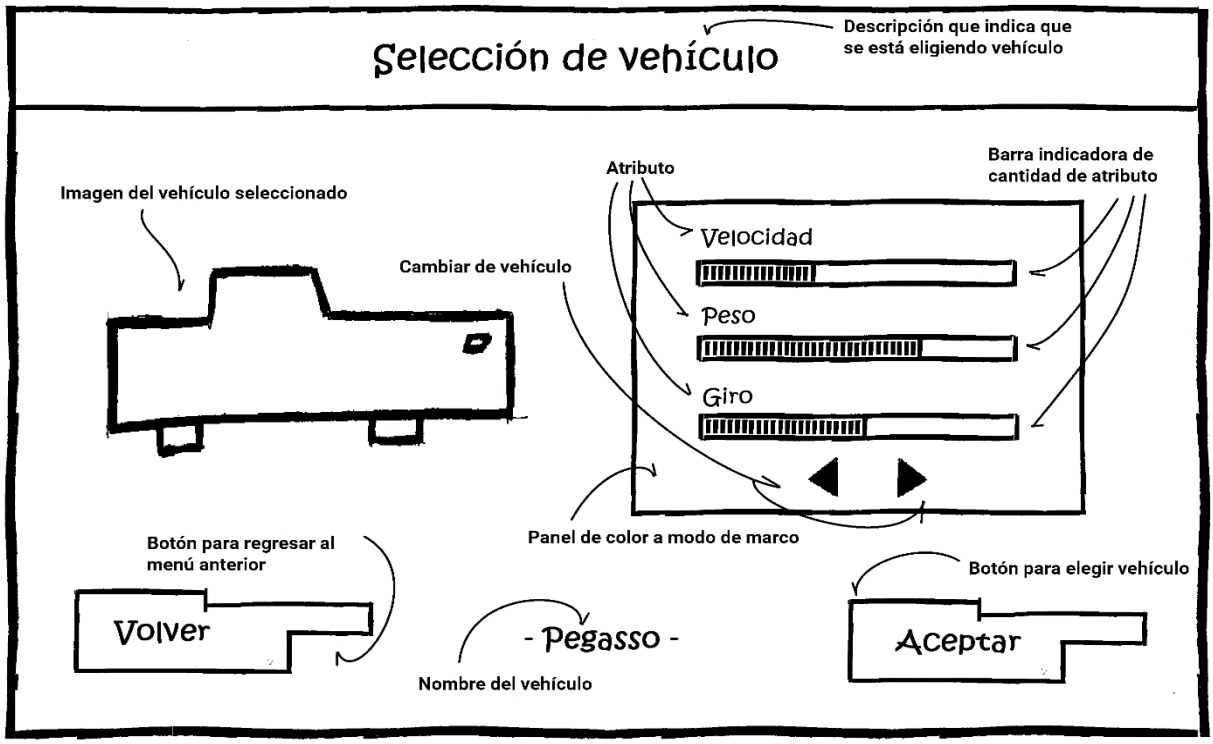

Ilustración 2.10: Sketch de menú de selección de vehículo de Crazy Taxi reboot

Además necesitamos una interfaz para la partida, la cual inspiraremos en el juego de Crazy Taxi y en el juego GTA. Ambas interfaces podemos verlas en la Ilustración 2.11 y en la Ilustración 2.12.

Ilustración 2.11: Interfaz de juego Crazy Taxi original

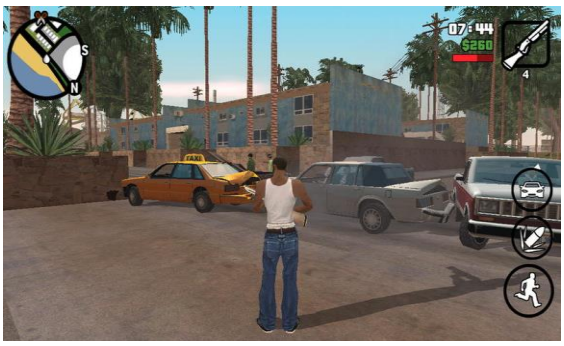

Ilustración 2.12: Interfaz de juego de GTA San Andreas

El sketch resultante de dicha mezcla de interfaces es el siguiente dado en Ilustración 2.13. Puede observarse como nos quedaremos con la idea del minimapa y

la barra de vida del GTA y se mostrará la información del tiempo y puntuación en forma de dinero dentro de paneles al igual que en Crazy Taxi.

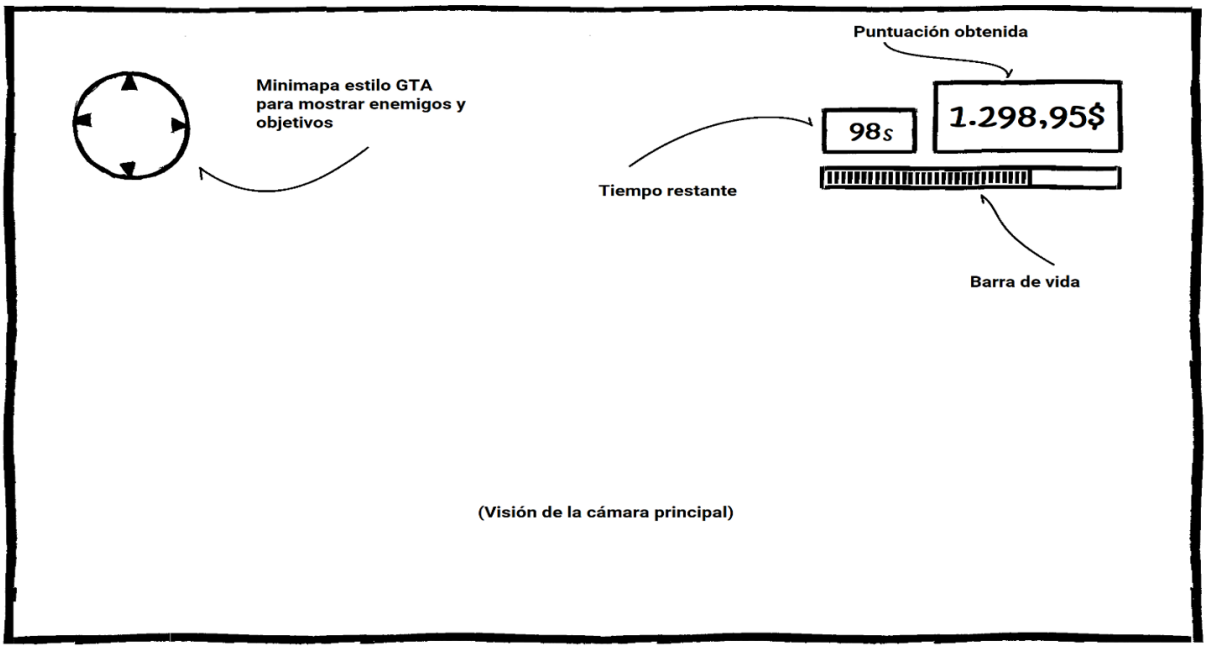

Ilustración 2.13: Sketch interfaz de juego de Crazy Taxi reboot

También necesitaremos un menú de pausa, que tendrá el siguiente diseño dado por el sketch en Ilustración 2.14. Como vemos, sigue una interfaz muy común que no es de Crazy Taxi, pero es usada en gran cantidad de juegos.

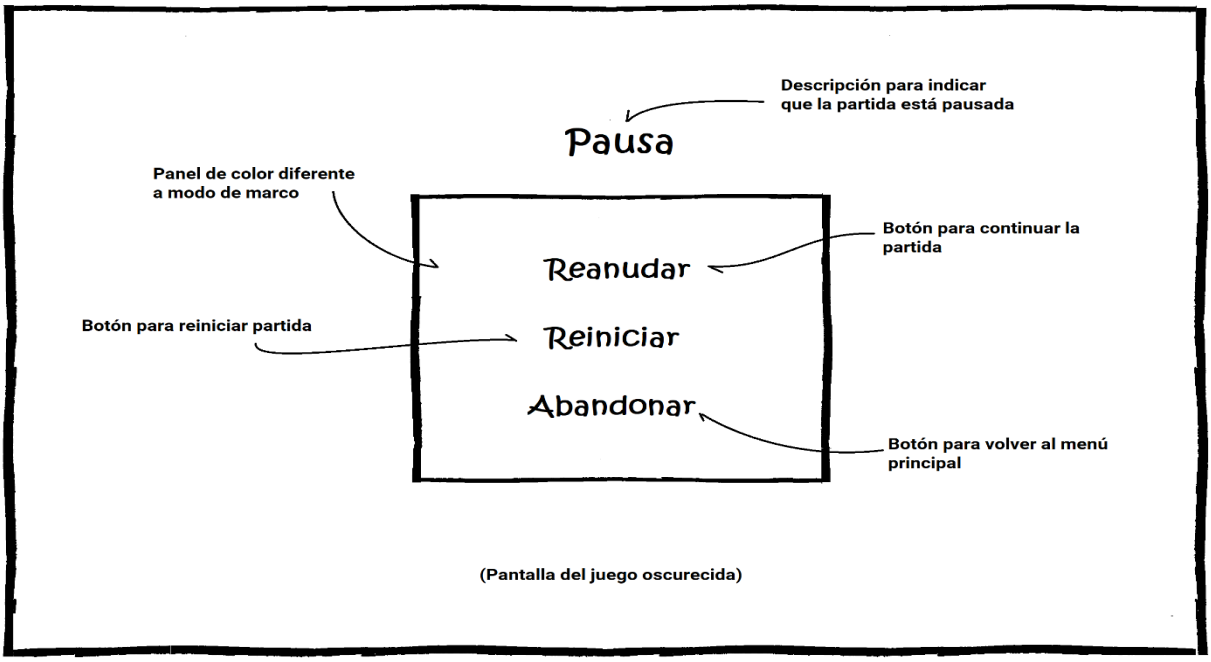

Ilustración 2.14: Sketch menú de pausa de Crazy Taxi reboot

Igualmente necesitamos un tablón de resultados, que crearemos similar al de los resultados de Crazy Taxi original, dado en la Ilustración 2.15.

Ilustración 2.15: Pantalla de resultados de Crazy Taxi original

El sketch para la pantalla de resultados creado es el dado a continuación en la Ilustración 2.16. En ella mostraremos un resumen de la partida, una imagen e información de interés para el jugador.

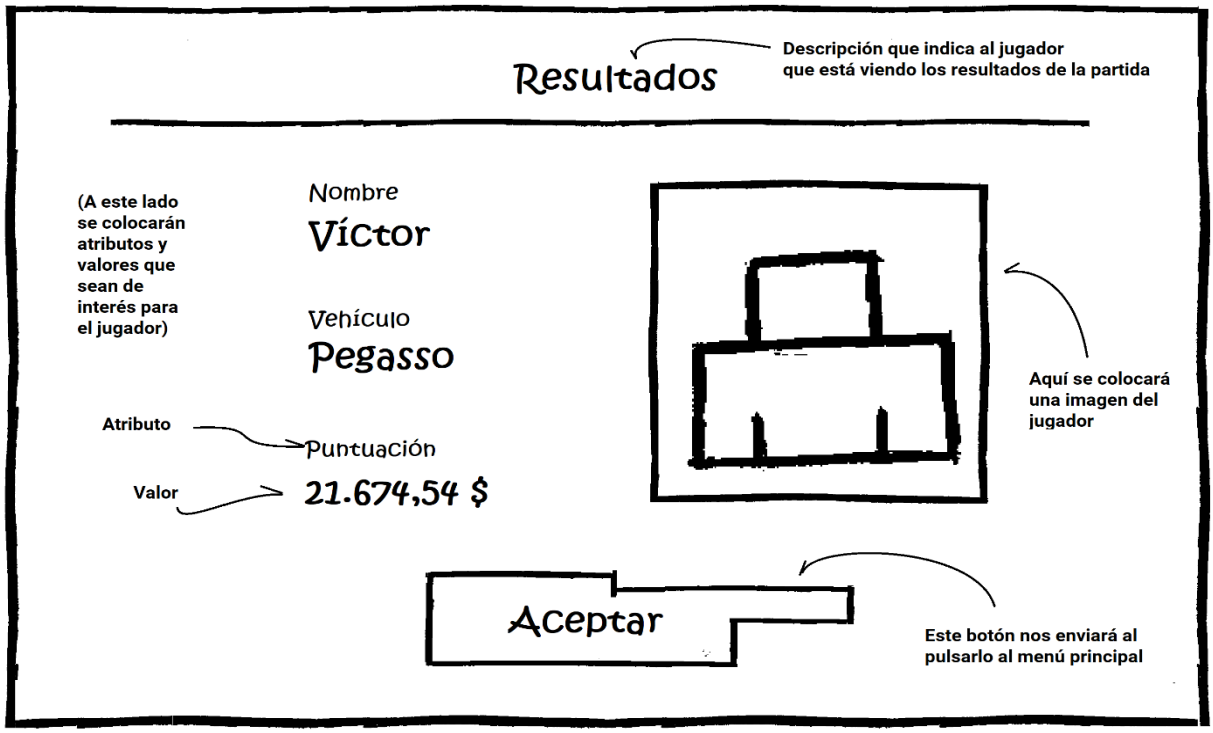

Ilustración 2.16: Sketch de pantalla de resultados de Crazy Taxi reboot

Finalmente se hará también un menú de ajustes que crearemos de cero y para el cual se ha hecho el sketch visto en la Ilustración 2.17.

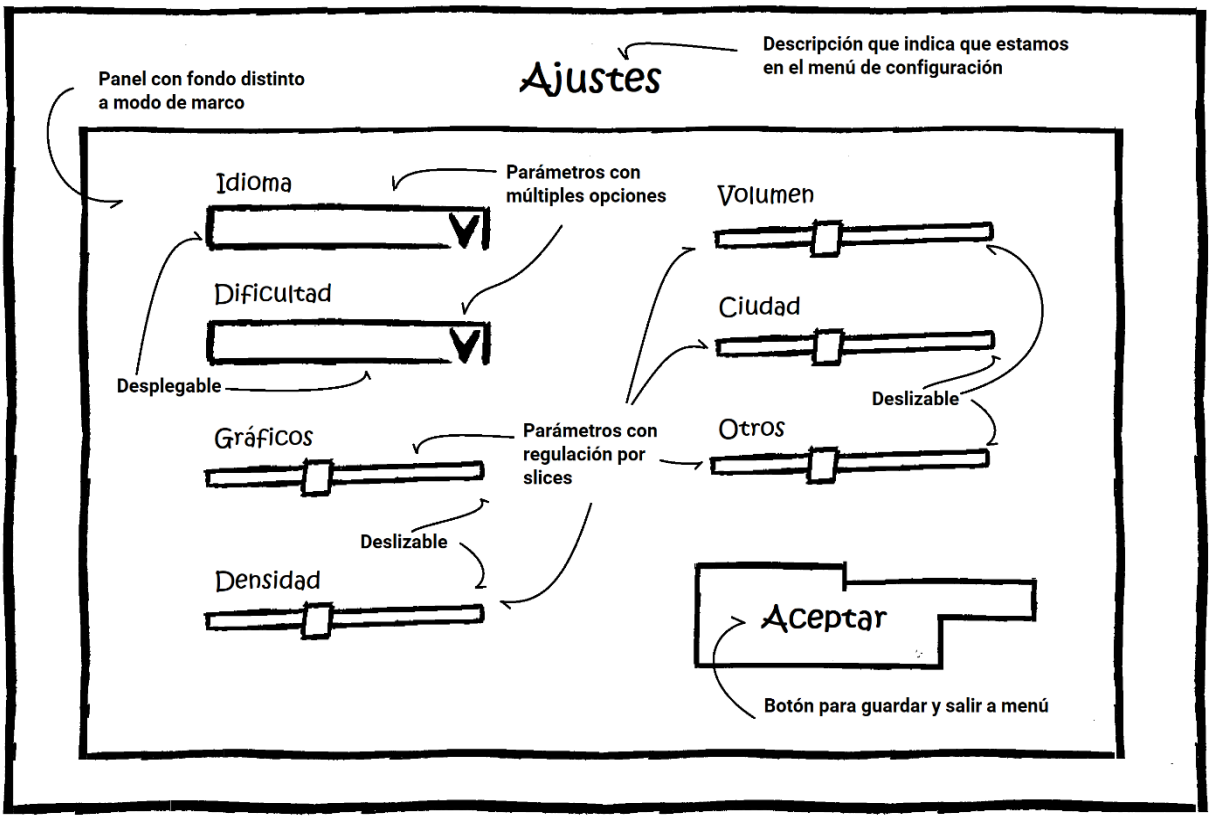

Ilustración 2.17: Sketch de menú de configuración de Crazy Taxi reboot

2.3.2 Storyboard

Podemos interconectar todos los menús diseñados mediante sketches, dando lugar al storyboard de la Ilustración 2.18, que muestra el flujo de navegación entre las diferentes secciones de la aplicación.

Como puede verse están numerados los estados para mayor facilidad de comprensión, los números corresponden a: menú principal (1), menú de selección de modo (2), menú de selección de conductor (3), menú de selección de vehículo (4), interfaz de juego (5), menú de pausa (6), pantalla de resultados (7) y menú de ajustes (8).

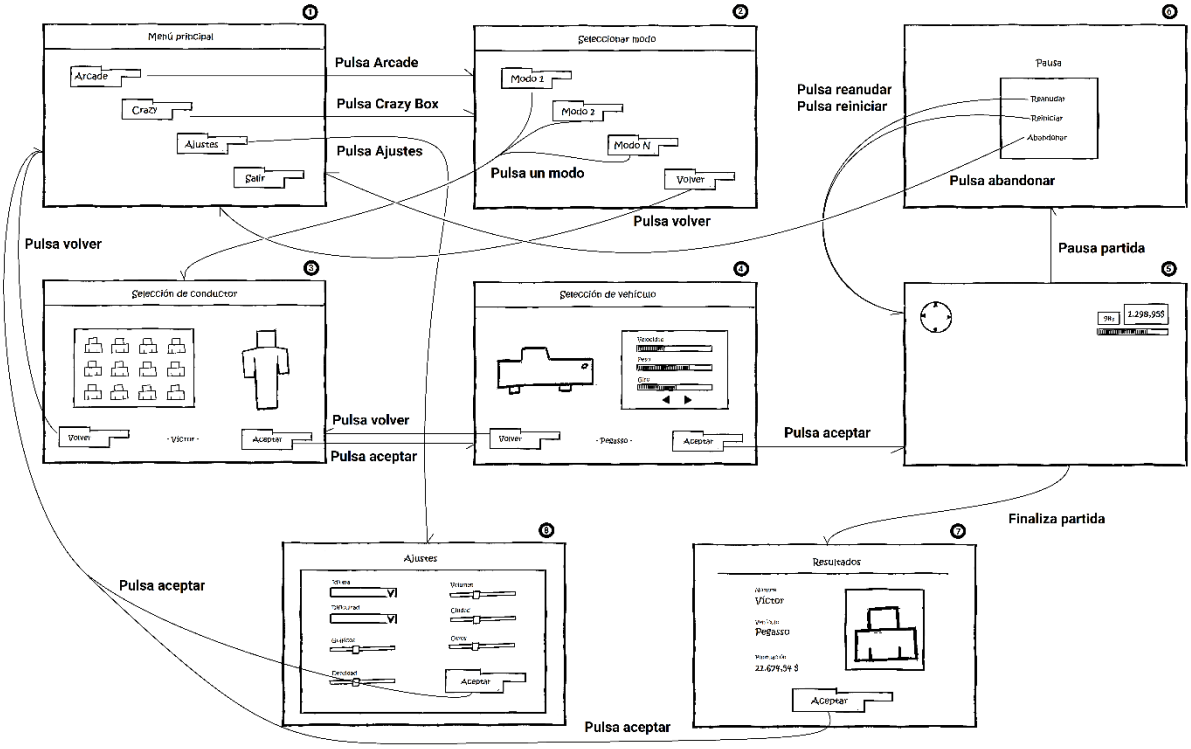

Ilustración 2.18: Storyboard del juego Crazy Taxi reboot

Una aclaración que se debe hacer es que el botón de “Arcade” y el botón de “Crazy Box” no debe llevar al mismo menú como podría dar la sensación en el storyboard, sin embargo, debido a que funcionan y reaccionan igual (con la única diferencia que configuran modos diferentes) se han unificado en el estado 2 del diagrama.

3 EJECUCIÓN DEL DESARROLLO

Como vimos anteriormente, recordamos que el proyecto ha seguido una metodología de desarrollo ágil Scrum. Es por ello que una vez se realizaron las primeras tutorías del curso, donde el alumno y el tutor acordaron una serie de requisitos iniciales, comenzó el desarrollo del proyecto en forma de sprints.

Dadas las dimensiones del proyecto, se ha decidido crear 2 secciones para explicar el desarrollo del mismo. Esta primera sección hablará acerca de las iteraciones reales llevadas a cabo y el contenido que aportan de manera genérica, mientras que, en la segunda sección, apartado 4 (Funcionalidades desarrolladas) hablaremos sobre las funcionalidades construidas de forma ordenada y por temáticas.

Separar el contenido en 2 apartados fue una decisión tomada para facilitar la comprensión general del proyecto, sin dejar de mostrar la cronología real llevada a cabo y permitiendo entender de manera clara y ordenada cada una de las funcionalidades del trabajo.

3.1 Sprint 1

Este sprint inicial es muy importante porque nos servirá para testear el ritmo de trabajo y asegurar que éste es tan viable como se esperaba durante la planificación. Como ya se estudió, este sprint está programado para realizar un total de 15 puntos de historia. Si analizamos la tabla donde tenemos asignados los puntos de historia para cada actividad, vemos que el desarrollo debe completar hasta la tercera actividad, sumando 10 + 2 + 3 = 15 puntos de historia.

En la Tabla 3.1 se ha coloreado en una tonalidad anaranjada todas aquellas tareas que se encuentran programadas para desarrollo en esta iteración.

Historia	Puntos de historia
Algoritmo procedural de ciudad	10
Vehículo jugador	2
Cámara personalizada	3

GPS con algoritmo A*	4
Flecha guía	1
Vehículos NPC normales	5
Vehículos NPC persecutores	4
Gestor de vehículos	2
Peatones NPC	10
Gestor de peatones	4
Características	1
Armas	3
Proyectiles	1
Interfaz principal	2
Menú principal	3
Menú de selección	4
Menú de pausa	1
Menú de fin de partida	1
Objeto de configuración	7
Modos de juego	3
Audio	2

Tabla 3.12: Puntos de historia a desarrollar en sprint 1

Dada la incertidumbre al inicio del proyecto, se planteó hacer un algoritmo de generación procedural para ciudades, pero no se llegó a desmembrar en sus diferentes módulos, es por ello que vamos subdividir ahora esa tarea de gran peso en múltiples tareas, estimadas en tiempo con una puntuación que equivale a la tarea inicial al sumarlas todas:

Historia	Puntos de historia
Algoritmo procedural de ciudad	10
↳ Generación de calles en forma de grafo	(4)
↳ Generación de topología y pendientes	(2)
↳ Creación e instanciación de prefabs	(4)

Tabla 3.22: Especificación de puntos de historia para algoritmo de generación de ciudad

Una vez puestos manos a la obra y como era de esperar, el primer sprint fue en su mayor parte dedicado a la creación de la primera versión funcional básica para la generación de ciudades debido a que es la base de todo el juego. Tal y como estaba previsto, quedaron implementadas las funcionalidades de creación de calles y aceras, el grafo que las resume, la topología pseudoaleatorizada y la instanciación aleatoria de objetos como señales y edificios, quedando únicamente pendiente la creación de barrios, que es una idea que se introdujo en una actualización posterior.

Además, se creó la primera versión funcional del vehículo del jugador con el que se ha podido llevar un gran número de pruebas de depuración. También se incluyó la cámara personalizada del juego, la cual puede acercarse, alejarse y darse la vuelta mediante el uso de los controles respectivos.

Para hacer un control del sprint y asegurar que nos amoldamos a la planificación, se va realizando a lo largo del mismo un diagrama de Burndown, cuyo resultado al final del mismo queda visible en la Ilustración 3.1.

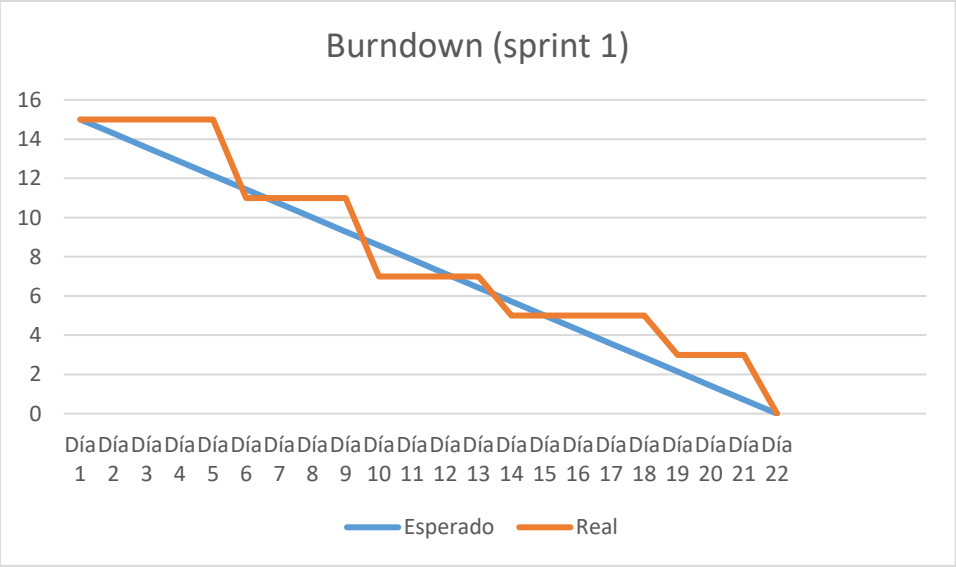

Ilustración 3.1: Diagrama de Burndown (sprint 1)

Como vemos, han sido completadas las tareas en el tiempo dado (duración de un mes promedio sin fines de semana), lo que indica que el ritmo de trabajo es bueno. El hecho de encontrar “escalones” es debido a que se baja la cantidad de puntos de historia restantes únicamente cuando la actividad se termina completamente.

3.2 Sprint 2

Esta segunda etapa tiene un total de 16 puntos de historia, por lo que habrá que ir algo más rápido que en la anterior para no salirnos de la planificación. En concreto los puntos de historia se reparten entre las historias de la creación del GPS, flecha indicadora vehículos y el gestor de vehículos, tareas que aparecen en naranja porque se pondrán en desarrollo, algo que hacíamos igual en el sprint anterior, con la novedad de que ahora en Tabla 3.3 vemos en verde aquellas historias que fueron cerradas.

Historia	Puntos de historia
Ciudad: Generación de calles en forma de grafo	4
Ciudad: Generación de topología y pendientes	2
Ciudad: Creación e instanciación de prefabs	4
Vehículo jugador	2
Cámara personalizada	3
GPS con algoritmo A*	4
Flecha guía	1
Vehículos NPC normales	5
Vehículos NPC persecutores	4
Gestor de vehículos	2
Peatones NPC	10
Gestor de peatones	4
Características	1
Armas	3
Proyectiles	1
Interfaz principal	2
Menú principal	3
Menú de selección	4
Menú de pausa	1
Menú de fin de partida	1
Objeto de configuración	7
Modos de juego	3
Audio	2

Tabla 3.39: Puntos de historia a desarrollar en sprint 2

A lo largo del desarrollo la planificación se consiguió respetar, alcanzando a construir un objeto GPS que hace uso del algoritmo A* [28] para crear rutas, una flecha indicadora para que el jugador conozca la dirección que tiene que seguir (aunque actualmente indica de manera aleatoria porque no se han implementado clientes que le indiquen rutas) y se han creado los vehículos normales y persecutores gestionados por su propio gestor, cuyo objetivo es crear, posicionar y destruir vehículos en el área cercana al jugador. Es importante tener en cuenta que se tiene pensado que algunos vehículos disparen, pero dado que aún no tenemos implementadas las armas, se tendrá que llevar una pequeña actualización en el futuro. Es destacable que este sprint ha requerido de un periodo de investigación para maximizar el rendimiento de la navegación del GPS, algo que se detallará más adelante en el apartado 4.3 (Navegación).

A continuación, se muestra el diagrama de quemado, Ilustración 3.2, que descubre cómo se ha distribuido el tiempo de desarrollo entre los diferentes puntos de historia.

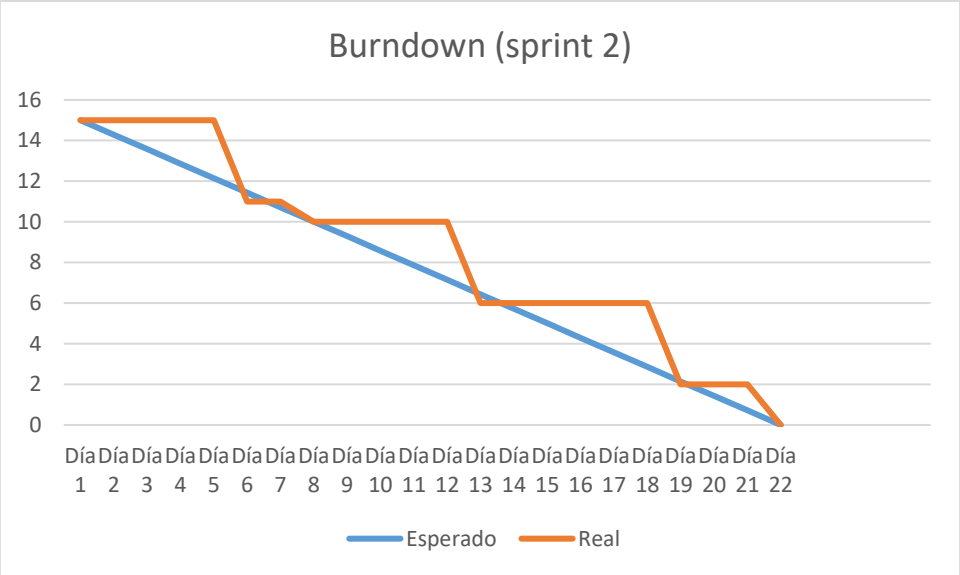

Ilustración 3.2: Diagrama de Burndown (sprint 2)

Al igual que en la anterior iteración conseguimos mantener un buen ritmo de trabajo, de modo que no hace falta hacer modificaciones de momento en las estimaciones del resto de sprints.

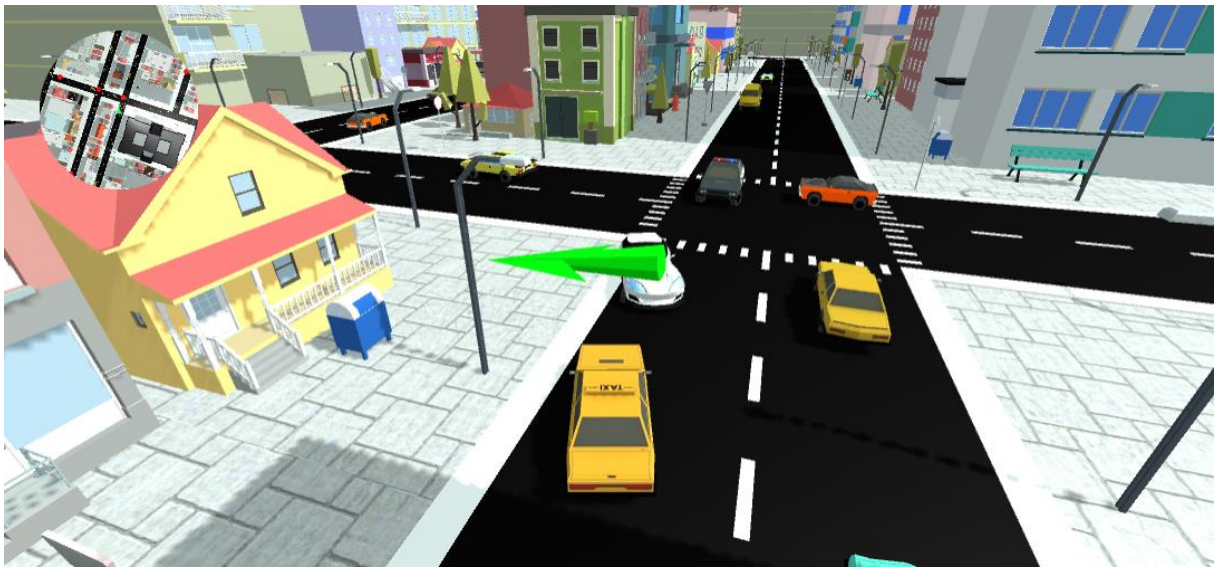

Ilustración 3.3: Apariencia tras finalizar sprint 2

Como puede observarse en la Ilustración 3.3, la apariencia actual está lejos de parecerse a la de Minecraft, tal y como estaba planteado en un inicio, es por ello que tras este sprint se llevó a cabo un periodo de cambio de modelos del juego por unos modelos más parecidos a los de Minecraft, llevando a descartar gran cantidad de modelos bajos en polígonos utilizados hasta ahora a modo de pruebas a favor de modelos de estilo voxelart.

3.3 Sprint 3

Para el tercer sprint de 15 puntos de historia, se han de desarrollar los peatones NPC, un gestor que se ocupe de su organización y de un componente de características que se utilizará para todos los objetos que lo necesiten dentro del proyecto.

Historia	Puntos de historia
Ciudad: Generación de calles en forma de grafo	4
Ciudad: Generación de topología y pendientes	2
Ciudad: Creación e instanciación de prefabs	4
Vehículo jugador	2
Cámara personalizada	3
GPS con algoritmo A*	4
Flecha guía	1

Vehículos NPC normales	5
Vehículos NPC persecutores	4
Gestor de vehículos	2
Peatones NPC	10
Gestor de peatones	4
Características	1
Armas	3
Proyectiles	1
Interfaz principal	2
Menú principal	3
Menú de selección	4
Menú de pausa	1
Menú de fin de partida	1
Objeto de configuración	7
Modos de juego	3
Audio	2

Tabla 3.4: Puntos de historia a desarrollar en sprint 3

De la misma manera que con el algoritmo de la ciudad, al comienzo del proyecto únicamente se tenía una idea superficial de las personas NPC y hasta bastante avanzado el trabajo no se concretó cuáles serían los tipos de comportamientos que éstas tendrían. Llegados a este sprint, ya sí que conocemos los comportamientos que debería haber (individuales, grupales, de cliente y de policía) y se redistribuye la puntuación de la tarea principal de Peatones NPC en varias subtareas de complejidad y duración acorde con los puntos que se les asocian. Así esta tarea genérica inicial de gran peso queda subdividida en múltiples tareas que suman una puntuación equivalente a la de la tarea inicial.

Historia	Puntos de historia
Peatones NPC	10
↳ Personas individuales	(2)
↳ Personas grupales	(2)
↳ Personas clientes	(3)
↳ Personas policías	(3)

Tabla 3.52: Especificación de puntos de historia para peatones

Aunque se tuvieron bastantes errores y problemas de animación en la parte del desarrollo de los NPCs que requirieron de muchas horas de depuración, conseguimos no romper con las fechas establecidas. Podemos ver el resumen temporal en la Ilustración 3.4, el cual muestra que actualmente el ritmo es el correcto.

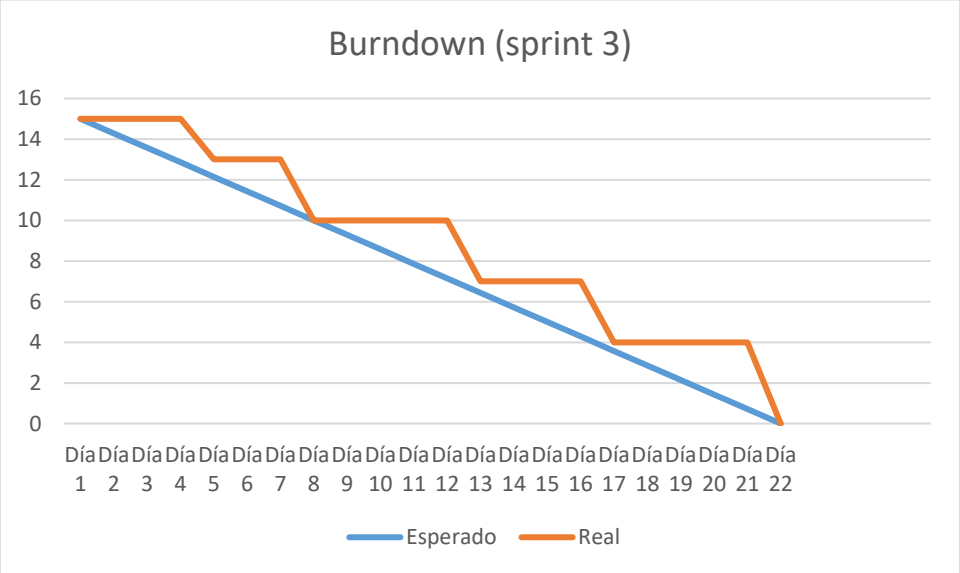

Ilustración 3.4: Diagrama de Burndown (sprint 3)

Nótese en la Ilustración 3.5 como se corrigió el estilo gráfico por uno mucho más parecido al que usa Minecraft y que presume de tener una muy buena apariencia teniendo en cuenta los recursos de los que partía el proyecto.

Ilustración 3.5: Apariencia tras finalizar sprint 3

3.4 Sprint 4

Es un sprint donde vemos claramente sus 15 puntos de historia repartidos en 2 pilares, el primero dedicado para la creación del sistema de armas y proyectiles (que conlleva una pequeña adaptación de los vehículos armados), y el segundo, que para el autor es uno de los más importantes del proyecto, dedicado a la interfaz y menús.

Historia	Puntos de historia
Ciudad: Generación de calles en forma de grafo	4
Ciudad: Generación de topología y pendientes	2
Ciudad: Creación e instanciación de prefabs	4
Vehículo jugador	2
Cámara personalizada	3
GPS con algoritmo A*	4
Flecha guía	1
Vehículos NPC normales	5
Vehículos NPC persecutores	4
Gestor de vehículos	2
Peatones: Personas individuales	2
Peatones: Personas grupales	2
Peatones: Personas clientes	3
Peatones: Personas policías	3

Gestor de peatones	4
Características	1
Armas	3
Proyectiles	1
Interfaz principal	2
Menú principal	3
Menú de selección	4
Menú de pausa	1
Menú de fin de partida	1
Objeto de configuración	7
Modos de juego	3
Audio	2

Tabla 3.6: Puntos de historia a desarrollar en sprint 4

Un punto interesante del proyecto es que lo encontramos completamente preparado para ser traducido a cualquier idioma, algo que no ha sido fortuito, sino que se ha tenido que tener en cuenta desde el primer texto introducido en el juego y para lo que ha requerido de la creación de un pequeño gestor de idiomas de manera paralela a la implementación de menús.

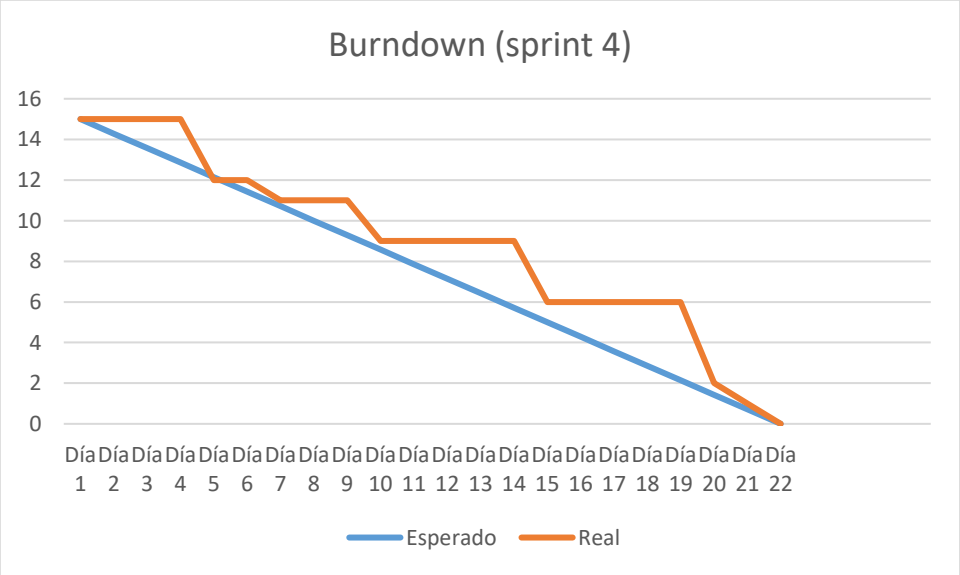

Ilustración 3.6: Diagrama de Burndown (sprint 4)

Sin entrar mucho más en detalle porque ya tendremos un apartado para extendernos en la explicación de las funcionalidades, pasamos a ver la Ilustración 3.6,

donde queda claro que la ejecución del desarrollo se está llevando de manera correcta para pasar a la última etapa.

3.5 Sprint 5

Se trata del último sprint previsto donde se planea dedicar gran parte del tiempo a uno de los objetos clave del proyecto, el objeto de configuración. El objeto de configuración es la raíz de todos los modos de juego, porque básicamente éstos son producto de las instrucciones, reglas y condiciones que marca este objeto al gestor del juego a la hora de generar el nivel. Además, este objeto permite una configuración pública y un sistema de guardado para que el usuario personalice el juego desde el menú correspondiente. Es por esta razón que es un objeto que desde un inicio se sabía que iba a ser complejo y es por tanto que goza de un gran número de puntos de historia reservados.

Una vez creado el objeto de configuración también está planeado incluir el audio y sobre todo los modos de juego, dando pie a los siguientes:

- Modo de reglas arcade: modo normal de arcade con el mismo funcionamiento que el original de Crazy Taxi.
- Modo de trabajo (3 minutos, 5 minutos y 10 minutos): modo normal que traen frecuentemente las entregas de Crazy Taxi.
- Modo de turismo: modo especial orientado para aprender a jugar.
- Modo de persecución: modo especial donde se aprovechan los vehículos persecutores y los policías al máximo.

Pero si contabilizamos todos los puntos de historia, vemos que hay un total de 12 frente a los 15 que llevábamos de media hasta ahora, lo que ha permitido aprovechar los 3 puntos de historia restantes para incluir sobre la marcha otros 2 nuevos componentes de baja complejidad: comportamiento de zombi y objeto bolo. Gracias a esos componentes con poco esfuerzo podremos incluir:

- Modo apocalipsis: modo especial donde se exprimen los zombis al máximo.
- Modo de bolos: modo especial donde se provechan los bolos al máximo.

En la Tabla 3.7 que resume los puntos de historia vemos la actualización y los últimos puntos a desarrollar antes de la finalización del proyecto.

Historia	Puntos de historia
Ciudad: Generación de calles en forma de grafo	4
Ciudad: Generación de topología y pendientes	2
Ciudad: Creación e instanciación de prefabs	4
Vehículo jugador	2
Cámara personalizada	3
GPS con algoritmo A*	4
Flecha guía	1
Vehículos NPC normales	5
Vehículos NPC persecutores	4
Gestor de vehículos	2
Peatones: Personas individuales	2
Peatones: Personas grupales	2
Peatones: Personas clientes	3
Peatones: Personas policías	3
Gestor de peatones	4
Características	1
Armas	3
Proyectiles	1
Interfaz principal	2
Menú principal	3
Menú de selección	4
Menú de pausa	1
Menú de fin de partida	1
Objeto de configuración	7
Incremento al sprint: Personas zombis	2
Incremento al sprint: Bolos	1
Modos de juego	3
Audio	2

Tabla 3.7: Puntos de historia a desarrollar en sprint 5

Véase en el diagrama (Ilustración 3.7) la peculiaridad que encontramos el día 11 donde disminuye la cantidad de puntos de historia realizados. Ello es debido a que ese día se incluyó al sprint los zombis y los bolos, algo que inicialmente no estaba previsto y que incrementa los puntos de historia por hacer.

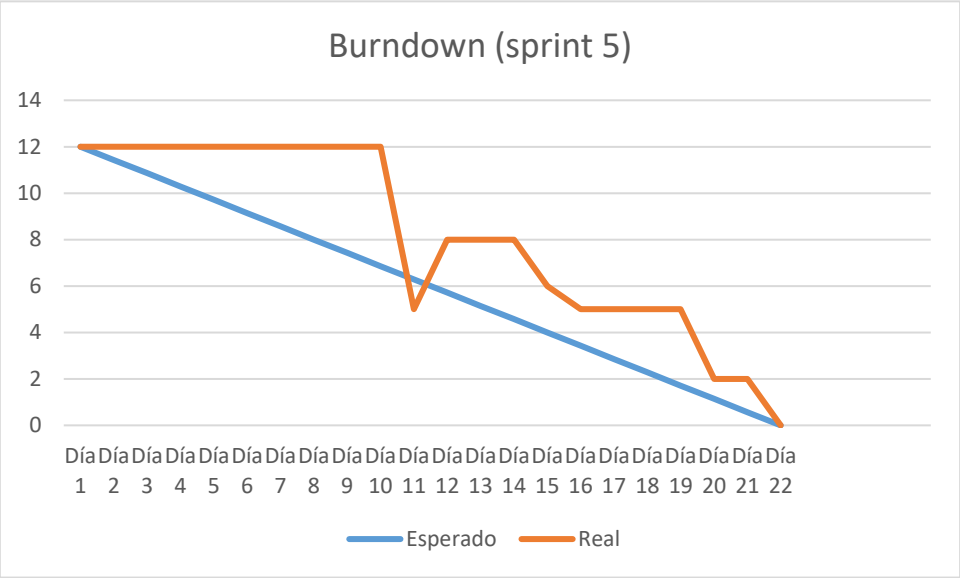

Ilustración 3.7: Diagrama de Burndown (sprint 5)

Tras finalizar la última iteración o sprint del modelo Scrum, el siguiente paso fue una nueva reunión con el cliente donde se presenta el proyecto ahora como un producto final.

4 FUNCIONALIDADES DESARROLLADAS

Aquí se explicará de la forma más amena y ordenada posible todas las funcionalidades que el proyecto incluye, haciendo uso de numerosas imágenes, esquemas, diagramas, tablas y pseudocódigos cuyo objetivo no será otro que transmitir todo el contenido de la manera más clara posible. Se trata del capítulo más extenso del proyecto y es muy importante leerlo detenidamente, porque será el que haga comprender todo lo que se ha nombrado dentro de los sprints.

Para comenzar y antes de entrar en detalle con la cuestión, nuevamente dejaremos claro que durante el diseño y desarrollo se ha buscado el minimalismo y un estilo llamativo similar al que utiliza Minecraft (estilo voxelart). El estilo ha sido un punto muy cuidado durante el proceso de creación del videojuego y es por ello que todos los objetos utilizados dentro del juego, así como la interfaz, textos y el propio icono estarán hechos a base de cubos y píxeles. La Ilustración 4.1 enseña un adelanto del resultado del juego para mostrar el estilo artístico final del juego.

Ilustración 4.1: Estilo voxelart del juego

A continuación, se abrirán múltiples subapartados donde se explicará el funcionamiento organizado por temáticas.

4.1 Jugador

Durante una partida el jugador podrá seleccionar un personaje y un taxi desde el menú de selección, apartados 4.9.5.5 y 4.9.5.6. Dicho personaje será el que posteriormente controle el jugador dentro de la escena de juego con el que podrá acelerar, frenar, dar marcha atrás, girar, derrapar y disparar mediante los controles específicos.

Se dispondrán de 2 posibles coches durante el juego y de 9 conductores como posteriormente veremos. Comenzando con los coches, podemos ver cómo es el modelo del primer vehículo seleccionable, Ilustración 4.2, y del segundo modelo, Ilustración 4.3.

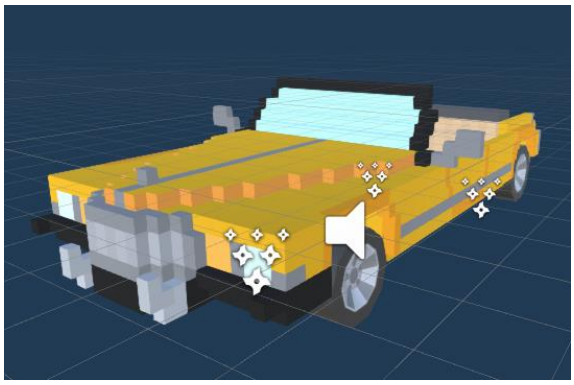

Ilustración 4.2: Chevrolet

Ilustración 4.3: Jeep

Ambos vehículos han sido obtenidos de la página de Sketchfab [29] y eran modelos en formato .obj que tenían las ruedas unidas al chasis, por lo que no estaban preparados para usarse en Unity, sino que estaban más orientados a la creación de voxelart. Para quedarnos solo con el chasis se ha modificado el objeto con Blender, eliminando las ruedas, con objetivo de añadirle unas propias. Además, se modificó la paleta de colores para que ambos modelos tuvieran color amarillo.

4.1.1 Estructura del vehículo y componentes

Si accedemos al interior del prefab [30] vemos que el vehículo del jugador es un objeto complejo, con gran cantidad de *GameObjects* en su jerarquía y componentes que merecen ser comentados.

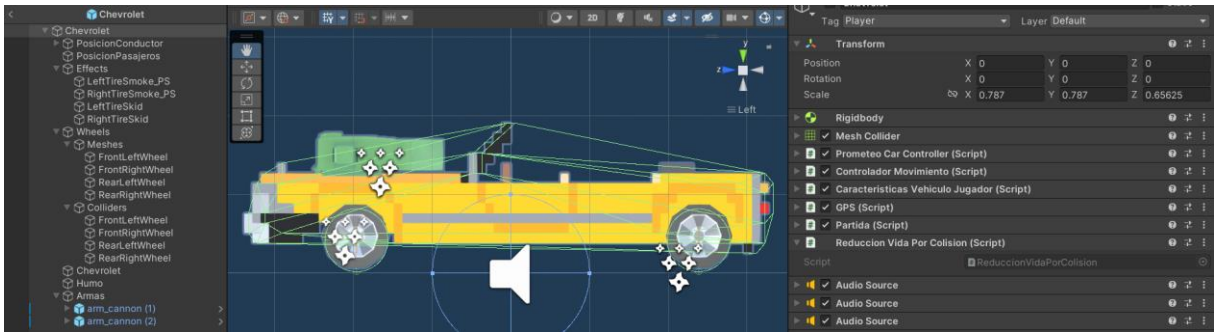

Ilustración 4.4: Estructura del taxi

En la Ilustración 4.4 se muestra la estructura que ambos vehículos siguen, compuesta por un total de 7 objetos principales:

- ***PosicionConductor***: es un objeto vacío ubicado encima del asiento del conductor y que es utilizado por el objeto de *Características* (apartado 4.6.1) del vehículo para situar al conductor.
- ***PosicionPasajeros***: es un objeto vacío que se ubica sobre el asiento de los pasajeros y que igualmente es utilizado por el objeto de características para sentar a los clientes.
- ***Effects***: es un objeto compuesto donde encontramos *GameObjects*, que generan efectos de humo y marcas de derrape en el suelo para las ruedas traseras hechos mediante los componentes *Particle System* [31] y *Trail Renderer* [32]. Dichos efectos muestran un humo proveniente de las ruedas y dejan la marca en el suelo durante unos segundos al derrapar respectivamente.
- ***Wheels***: es un objeto compuesto que contiene las mallas de las 4 ruedas, que es lo que se ve, y sus colisionadores, que es lo que colisiona. Los colisionadores de las ruedas son unos especiales para su uso en ruedas que trae Unity implementados en un componente llamado *WheelCollider* [33], específicos para su uso en vehículos.
- ***Chevrolet***: es la malla que hemos modificado en Blender para quitarle las ruedas (en este caso se llama así por la forma del chasis, en el modelo del Jeep tiene otro nombre).
- ***Humo***: se trata de un sistema de partículas que hace uso del componente *ParticleSystem*. Sirve para asociarlo al componente *Características* y

simular la avería en un vehículo. El sistema de partículas está configurado para que generen partículas con texturas de humo semitransparente de forma cíclica hacia arriba mediante la forma de un tronco de cono.

- ***Armas***: es un *GameObject* que se activará en algunos modos de juego que proporciona a nuestro jugador la capacidad de disparar. Para ello se ha asociado un modelo de cañón voxilizado con un objeto *ArmaJugador* (apartado 4.6.2.2) y su respectivo controlador con proyectiles explosivos y una frecuencia de error del 0% para no perjudicar la experiencia de disparo del jugador, cosa que como veremos no pasa con los NPCs.

Vista la organización de los objetos que componen nuestro vehículo, ahora detallaremos los componentes que encontramos en él y que igualmente pueden verse desde la Ilustración 4.4:

- ***RigidBody*** [34]: es necesario para que nuestro vehículo sea incluido en el sistema de físicas del juego.
- ***MeshCollider*** [35]: se trata de un tipo de caja de colisiones con forma personalizada mediante una malla concreta. La idea es pasarle la malla de vehículo para que los choques sean lo más realistas posible.
- **Script *PrometeoCarController***: hay que pasarle las mallas, colisionadores y efectos de las ruedas, además de varios *AudioSources* y *AudioClips* [36] para el sonido del derrape de las ruedas y del motor.

  Se trata de un código que sirve para controlar el movimiento del vehículo y que fue obtenido de un asset de la Asset Store de Unity y posteriormente adaptado al patrón Modelo-Vista-Controlador (MVC) [37], desarrollado en el apartado 4.10.1.1 (Aplicación de MVC al movimiento del jugador).

  Otro cambio importante ha sido para simular el movimiento del Crazy Taxi original, el cual por cierto no es para nada realista. En el juego original si se intenta girar con el vehículo, este siempre gira sin inercias por lo que no derrapa, pero nosotros al estar usando el componente *WheelCollider* y las físicas de Unity, sí que derrapamos. Para arreglar esto inicialmente se intentó lo más obvio, cambiar el rozamiento de las ruedas para que derraparan menos y, pese a conseguirlo, no era ni de lejos el resultado que

buscamos. Es por tanto que se recurrió a una solución más drástica, en la que tomamos el valor del vector de velocidad, obtenido del componente *RigidBody*, y lo reorganizamos para que siempre tenga la misma dirección que la orientación el vehículo.

- **Script *ControladorMovimiento***: como hemos comentado, el funcionamiento del vehículo del jugador sigue el patrón MVC. Este script precisamente actúa con el papel de controlador, ocupándose de ser el intermediario entre la vista y el modelo.
- **Script *CaracterísticasVehículoJugador***: es un componente que se explica en detalle el apartado 4.6.1.3 (*CaracteristicasJugador*) y que mantiene la información de la vida, velocidad, aceleración, peso... Gracias al script, cada vehículo seleccionable podrá tener características personalizadas que modifican por completo su jugabilidad.
- **Script *GPS***: desarrollado en detalle en el apartado 4.3 (Navegación), se utiliza cuando un cliente sube al taxi para calcular la ruta que hay que seguir para llegar desde la posición actual hasta el destino final.
- **Script *Partida***: es un script que lleva la cuenta del dinero y del tiempo de la partida del jugador. Tiene asociados los paneles de puntuación del *Canvas* que más adelante se explicarán. Añadido a eso tiene un *AudioClip* con sonidos de monedas para reproducirlo en mediante el componente *AudioSource* correspondiente cada vez que el dinero aumente.
- **Script *ReduccionVidaPorColision***: es un script que en el método *OnCollisionEnter()* reduce una cantidad de vida al jugador proporcional a la velocidad y peso del mismo.

4.1.2 Conductores

Como hemos comentado el vehículo del jugador tendrá un conductor ubicado en la posición reservada para el mismo. Dicho conductor, se trata de un modelo igual a los que se utilizarán para el apartado 4.4 (Personajes NPC). Su prefab no trae ningún componente asociado a excepción del componente *Animator* [38], que sirve para asociarle la animación de conducir, pues el conductor es un detalle visual que no modifica la forma de jugar.

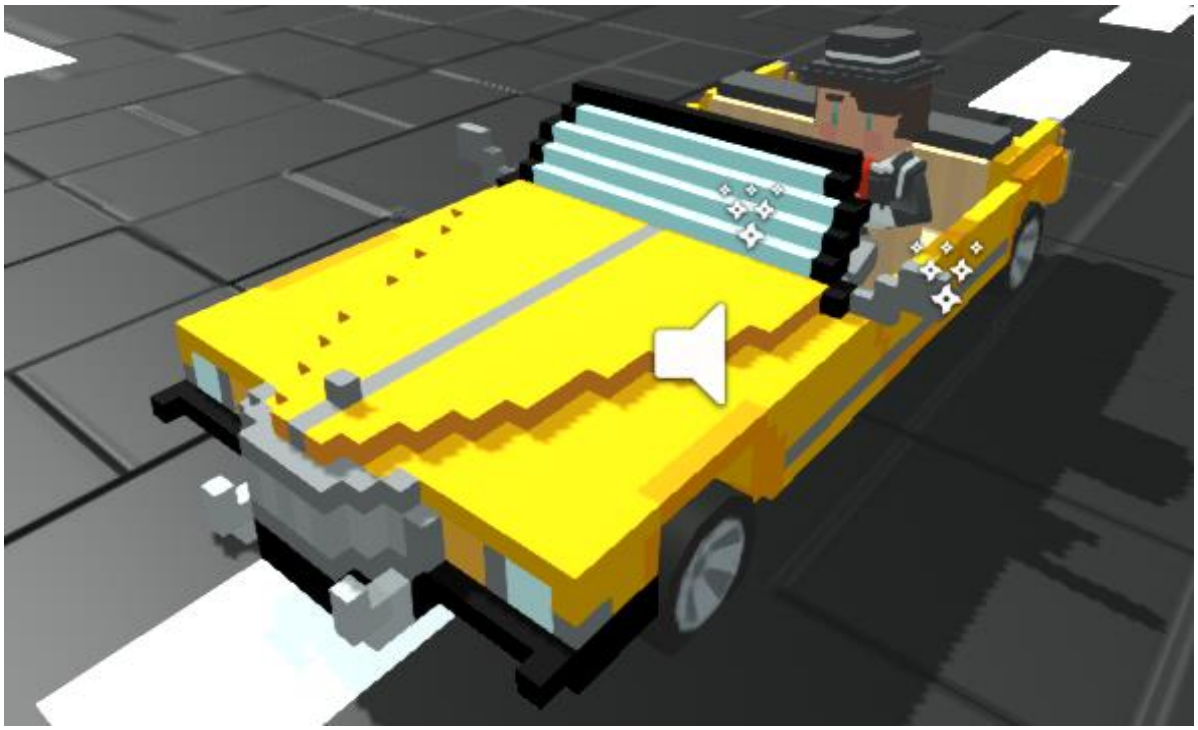

Ilustración 4.5: Taxi con conductor

4.1.3 Pruebas

A continuación, se van a mostrar los resultados de los test realizados para el vehículo del jugador.

T-01 Test de aceleración y deceleración	
Objetivo	El vehículo reaccionará a las entradas dadas por los controles y se permitirá acelerar y frenar
Resultado	El vehículo reacciona correctamente a los controles y puede acelerar y frenar
Observaciones	El frenado es algo bajo y complica un poco su uso

Tabla 4.1: Test de aceleración y deceleración

T-02 Test de giro	
Objetivo	El vehículo reaccionará a las entradas dadas por los controles y se permitirá girar hacia la derecha y hacia la izquierda
Resultado	El vehículo reacciona correctamente a los controles y es capaz de girar fácilmente en las curvas
Observaciones	Aunque el vehículo no sigue una conducción realista, sí que es como la conducción de Crazy Taxi y es muy cómoda

Tabla 4.2: Test de giro

T-03 Test de derrape	
Objetivo	El vehículo reaccionará a las entradas dadas por los controles y se permitirá derrapar
Resultado	El vehículo reacciona correctamente a los controles para derrapar manualmente y además es capaz de derrapar automáticamente cuando el giro es grande y a gran velocidad
Observaciones	

Tabla 4.3: Test de derrape

T-04 Test de disparo	
Objetivo	El vehículo reaccionará a las entradas dadas por los controles y se permitirá disparar donde indica la mirilla
Resultado	El vehículo reacciona correctamente a los controles para disparar y enviar los misiles donde apunta el cursor
Observaciones	A veces al disparar el jugador recibe daño debido al daño de área de los proyectiles porque chocan cerca del mismo

Tabla 4.4: Test de disparo

4.2 Ciudad con generación procedural

El algoritmo *Generacion* persigue la creación de una ciudad, donde el terreno esté generado por hexaedros unidos entre sí con una topología aleatorizada (siguiendo el estilo de Minecraft), sobre el cual se colocan carreteras, edificios y objetos partiendo de posiciones aleatorias a las que llamaremos vértices semilla.

Para comprender mejor el algoritmo vamos a definir en primer lugar 3 niveles de detalle a los que se hará referencia a partir de ahora y son utilizados constantemente dentro del script. La idea es generar la ciudad de forma genérica e ir especificándola con los niveles de profundidad:

- **Nivel 1**: nivel más genérico que indica el número concreto de vértices semilla que deben crearse en cada bloque de profundidad 1 (sin especificar posición). Se decidió que todos los bloques tuvieran el mismo número de vértices semilla (lo que hace que la ciudad tenga una densidad de calles más homogénea).
- **Nivel 2**: nivel intermedio en el que se indica donde se ubica cada vértice semilla en la profundidad 2. Tendrá una representación matricial que se

utilizará, además de para ubicarlos, para darle forma a las calles (calles de cualquier tipo). Cada bloque de profundidad 2 tiene 5x5 bloques de profundidad 3 en su interior.

- **Nivel 3**: nivel de máximo detalle donde se colocan por tanto los objetos que dan la apariencia a la ciudad tales como el bloque base, edificios, aceras, calles rectas, intersecciones de calles, calles curvas…

En el esquema dado por la Ilustración 4.6 se muestra la idea de forma comprensible. Destacaremos además que el tamaño de las matrices de profundidad en esta imagen sería de:

- Matriz de profundidad nivel 1: 3x3
- Matriz de profundidad nivel 2: 9x9
- Matriz de profundidad nivel 3: 45x45

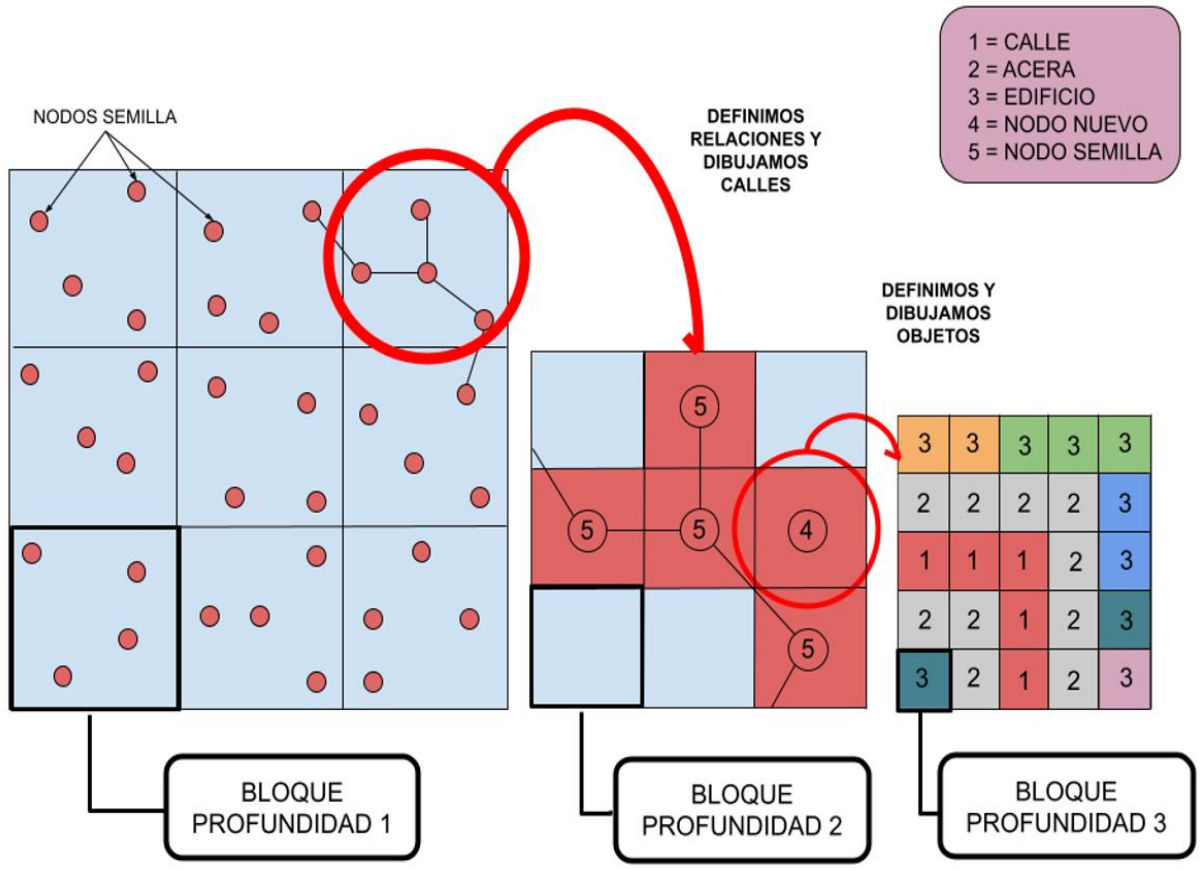

Ilustración 4.6: Esquema de generación

El algoritmo de generación se divide principalmente en:

- Generación aleatoria de barrios
- Generación aleatoria del grafo de calles
- Generación aleatoria de topología
- Instanciación de calles
- Instanciación de aceras
- Instanciación de edificios
- Instanciación de límites

4.2.1 Generación de barrios: algoritmo de relleno multisemilla

Para la creación aleatoria de barrios buscaremos rellenar una matriz *matrizBarrios* que guarde el identificador de barrio que corresponde a cada celda de la matriz de profundidad 3. Sin embargo, asignar de forma completamente aleatoria a cada bloque un barrio provocaría una distribución demasiado caótica de los mismos y muy poco realista, por lo que crearemos un algoritmo que sea capaz de sobrellevar este problema, algoritmo que encontramos en el script *RellenoBarrios*. Para ello nos basaremos en la idea de los algoritmos de relleno por inundación [39], cuya síntesis es un algoritmo recursivo que, dado un píxel semilla, va coloreando en una imagen el área limitada por bordes a la que pertenece.

En nuestro caso como buscamos de alguna forma manchar la matriz con valores que representen al barrio, usaremos varias semillas (tantas como barrios y colocadas aleatoriamente en casillas distintas) y los límites para la expansión de cada semilla ahora no serán bordes, sino que serán los valores enteros que representen un barrio diferente al de dicha semilla. Además, como buscamos que nuestros barrios se generen o crezcan de forma aleatorizada, tenemos que hacer que el relleno se pueda dar por cualquier casilla definida por un barrio hacia cualquier dirección, por lo que llevaremos un listado de casillas abiertas para cada una de las semillas.

En otras palabras, cada semilla inicialmente marcará con el número del identificador de su barrio la celda en la que se ubica y registrará en su lista correspondiente de casillas abiertas a todas las celdas adyacentes que no

pertenezcan a un barrio. En la siguiente iteración se selecciona una celda cualquiera de entre las abiertas por su semilla, la marca con el identificador del barrio y la saca de la lista de casillas abiertas. Este proceso se repite para cada lista hasta que todas queden vacías, dando como resultado una matriz rellena por completo, que representa la división aleatorizada en barrios del área de la ciudad.

En las imágenes Ilustración 4.7, Ilustración 4.8 e Ilustración 4.9 pueden verse múltiples resultados obtenidos con el algoritmo para una ciudad con 4 barrios (aunque podemos enviarle el número de barrios que deseemos):

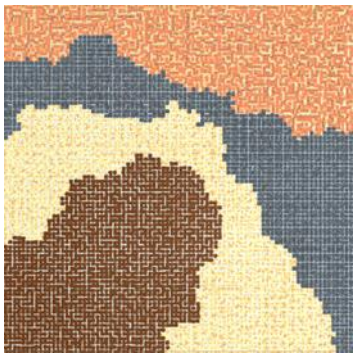

Ilustración 4.7: Barrios 1

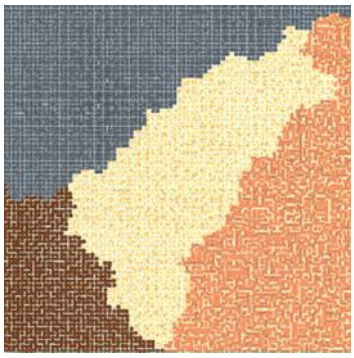

Ilustración 4.8: Barrios 2

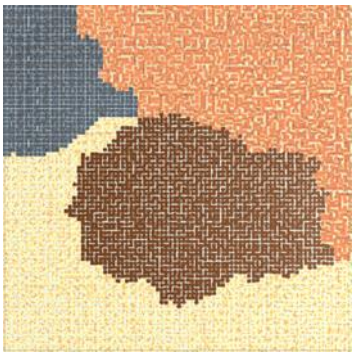

Ilustración 4.9: Barrios 3

De esta manera, y como veremos más adelante, a la hora de colocar un edificio se tendrá en cuenta el valor de la matriz para elegir un edificio de entre el conjunto de edificios de un barrio u otro. Cabe destacar que inicialmente el algoritmo se pensó de forma recursiva, pero por simplicidad y sobre todo por problemas de espacio en la pila, se terminó descartando a favor de la versión sin recursividad. El siguiente pseudocódigo funciona como se ha explicado y aporta una visión fiel del encontrado en el script del proyecto.

```
 1. struct Celda
 2. {
 3.     CoordI : Int
 4.     CoordJ : Int
 5. }
 6.
 7. Int[][] Barrios(dimensionCiudad : Int, numeroBarrios  : Int)
 8. {
 9.     matrizBarrios : Int[dimensionCiudad][dimensionCiudad]
10.     casillasAbiertas : List<Celda>[numeroBarrios]
11.     semillas : List<Celda>
12.     posCelda : List<Celda>
13.     //Inicializamos matriz
14.     matrizBarrios[i -> dimensionCiudad][j -> dimensionCiudad] = VACIO
15.     //Mientras no tengamos el misno numero de semillas que de barrios
```

```
16.     while (semillas.Size < numeroBarrios)
17.         semilla = celdaAleatoria(matrizBarrios)
18.         if(!semillas.Countains(semilla)) semillas.Add(semilla)
19.     //A cada lista de casillas abiertas se le incluye la celda semilla
20.     for(i -> numeroBarrios)
21.         casillasAbiertas[i].Add(semillas[i])
22.         posCelda.Add(casillasAbiertas[0])
23.     do
24.         for (i -> posCelda.Size)
25.             //Marcamos casilla con barrio correspondiente si esta vacia
26.             if (matrizBarrios[posCelda[i].CoordI][posCelda[i].CoordJ] == VACIO)
27.                 matrizBarrios[posCelda[i].CoordI][posCelda[i].CoordJ] = i
28.                 //Se cierra la casilla y se elimina de todas las listas
29.                 for (j -> casillasAbiertas.Size)
30.                     casillasAbiertas[j].Remove(posCelda[i])
31.         //Generamos celdas adyacentes
32.         casillasParaAbrirPorBarrio : List<List<Celda>>
33.         for(i -> posCelda.Size)
34.             casillasParaAbrirPorBarrio.Add(celdasAdyacentes(posCelda[i]))
35.         //Abrimos casillas adyacentes posibles que no esten abiertas
36.         foreach (casilla in casillasParaAbrirPorBarrio)
37.             if (!casillasAbiertas.Countains(casilla)) casillasAbiertas[i].Add(casilla)
38.         if (!casillasAbiertas[i -> casillasAbiertas.Size].Empty)
39.             //Enviamos una semilla aleatoriamente de entre las abiertas para abrir
40.             semillasParaLLamada : List<Celda>
41.             for (i -> casillasParaAbrirPorBarrio.Size)
42.                 if (!casillasAbiertas[i].Empty)
43.                     semillasParaLLamada.Add(celdaAleatoria(casillasAbiertas[i]))]
44.                 else
45.                     //El barrio ha dejado de crecer, se envia una posicion ya cerrada
46.                     semillasParaLLamada.Add(posCelda[i])
47.             posCelda = semillasParaLLamada
48.     while (casillasAbiertas[i -> casillasAbiertas.Size].Empty)
49.
50.     return matrizBarrios
51. }
```

4.2.2 Generación de calles: grafo de la ciudad

Nuestra ciudad fue diseñada desde el primer momento pensando en los problemas de rendimiento que podían aparecer y es por ello que para maximizar la eficiencia lo que se hará es, como en secciones posteriores se desarrollará, trabajar con el grafo que define la ciudad en lugar hacer uso de las mallas navegables que Unity nos proporciona. Para no perdernos, la idea final del grafo es que sus vértices sean curvas, intersecciones y cruces, mientras que sus aristas serán calles rectas.

Para el uso de la matriz de adyacencia en posteriores secciones se usarán funciones que nos permiten trabajar con ella y traducir la información de la misma a coordenadas de mundo, aunque dejando a un lado este tema de momento, vamos a hablar acerca de la elaboración del grafo, operación que constará de 3 fases.

4.2.2.1 Fase 1: definición inicial de la matriz de adyacencia

Primeramente, se crea la matriz de adyacencia donde los vértices serán las posiciones de las semillas generadas inicialmente dentro de la matriz de profundidad de nivel 2, por lo que tendremos una matriz de adyacencia de partida con un tamaño del $\mathrm{numeroSemillas}^2$. Las relaciones del grafo, que será no orientado, vendrán dadas por la distancia entre la ubicación de las semillas, representadas en la matriz (que por definición será simétrica) con un 1.

4.2.2.2 Fase 2: algoritmo para conectividad del grafo

En segundo lugar, vamos a asegurar que nuestro grafo realmente sea conexo [40], y modificarlo en el caso de que no lo sea, pues este requisito es sumamente importante para el correcto funcionamiento de nuestro algoritmo A*, que veremos en el apartado 4.3 (Navegación). Recordemos que un grafo conexo es aquel para el que para cada par de vértices existe un camino que los conecta.

En este caso se ha creado un algoritmo que recorre en preorden [41] el grafo de calles partiendo desde un vértice cualquiera, evitando ciclos e incluyendo todos los vértices que encuentra dentro de la estructura de datos (EEDD) conjunto disjunto [41]. Si el conjunto disjunto tras la lectura completa del grafo contiene a todos los vértices entonces es conexo, y en caso contrario habrá que tomar uno de los vértices no pertenecientes al conjunto y generar un nuevo conjunto disjunto para finalmente conectarlos mediante una relación (que en nuestro caso ha sido la selección de un vértice cualquiera con el más cercano del otro conjunto). Tras conectarlos se debe hacer la unión entre ambos conjuntos disjuntos, pues ya están relacionados, quedándonos nuevamente con solo un conjunto. Este proceso se repite hasta que finalmente solo haya un conjunto disjunto con todos los vértices.

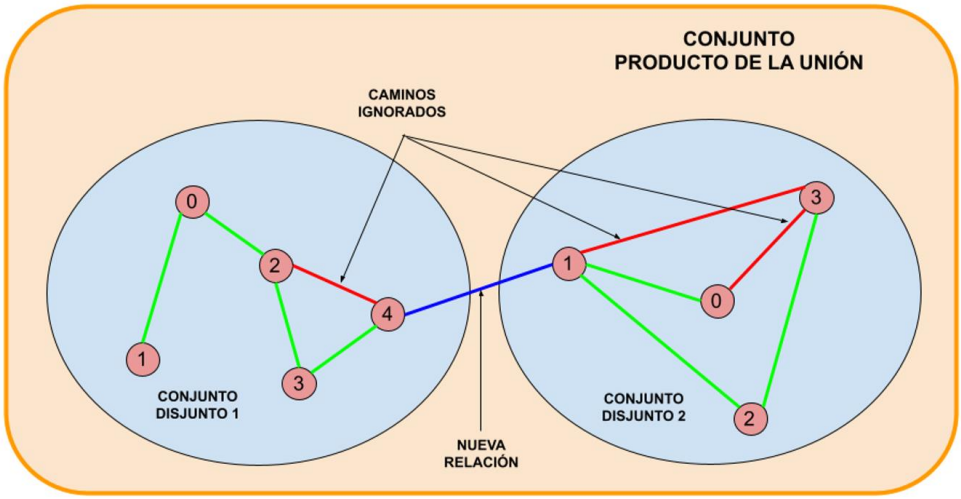

Ilustración 4.10: Algoritmo de conectividad de grafos

En la Ilustración 4.10 se ve representado un grafo en el que se ha realizado el recorrido en preorden siguiendo las aristas de color verde y obviando las rojas porque llevaban a un vértice ya desarrollado anteriormente. Visto que tras dicho recorrido no se pasa por todos los vértices, entendemos que es no conexo y repetimos el proceso con otro vértice sin abrir. Concretamente este grafo dará lugar a 2 conjuntos disjuntos que finalmente se conectan con la arista azul para posteriormente obtener el conjunto naranja como producto de la unión. Dado que en el caso de este esquema ya solo quedaría un conjunto con todos los vértices en su interior, el algoritmo terminaría su ejecución.

El procedimiento puede verse reflejado dentro del proyecto y su pseudocódigo para comprenderlo mejor es el siguiente.

```
1. ConectarGrafo(matrizAdyacencia :Int[][], verticesSemilla :Vertice[], numVerticesSemilla :Int)
2. {
3.     conjuntoTotal : Disjunto<Int>
4.     verticeCreadorGrupo : Int
5.     //Creamos conjuntos parciales para anexionarlos mientras el total no tenga todo
6.     while (conjuntoTotal.Size != numVerticesSemilla)
7.         conjuntoParcial : Disjunto<Int>
8.         //El conjunto parcial tendrá un vértice no perteneciente al total
9.           for (i -> numVerticesSemilla)
10.              if (!conjuntoTotal.Contains(i))
11.                  verticeCreadorGrupo = i
12.                  break
13.        RecorridoPreorden(verticeCreadorGrupo, conjuntoParcial, matrizAdyacencia)
14.        //Unimos vértice del conjunto total con otro del parcial
15.        if (!conjuntoTotal.Empty)
16.             verticeAleatorio = obtenerAleatorio(conjuntoTotal)
17.             //Relación con vértice más cercano del conjunto parcial
18.             cercano = verticeMasCercano(verticesSemilla[verticeAleatorio], conjuntoParcial)
```

```
19.                 matrizAdyacencia[cercano][ verticeAleatorio] = 1
20.                 matrizAdyacencia[verticeAleatorio][cercano] = 1
21.             //Unificamos los conjuntos total y parcial dentro del conjunto total
22.             conjuntoTotal.Union(conjuntoParcial)
23. }
24.
25. RecorridoPreorden(verticeActual :Int, conjuntoParcial :Disjunto<Int>&, matrizAdy :Int[][])
26. {
27.     //Si no se ha explorado lo estudiamos
28.     if (!conjuntoParcial.Contains(verticeActual))
29.         conjuntoParcial.Add(verticeActual)
30.         for (i -> matrizAdy[verticeActual].Size)
31.             //Si hay relación probamos a estudiar el vértice
32.             if(matrizAdy[verticeActual][i] != 0)
33.                 RecorridoPreorden(i, conjuntoParcial, matrizAdy)
34. }
```

4.2.2.3 Fase 3: definición final de la matriz de adyacencia

En este paso tenemos un grafo dado en forma de matriz de adyacencia, que haremos que quede reflejando en la matriz de profundidad 2, la cual simplemente indica el sitio donde hay o no calles.

Como tenemos las posiciones de los vértices semilla dentro de la matriz de profundidad 2, por cada relación con otro vértice marcamos como calle las casillas (en profundidad 2) que usamos para unirlos. Para unirlos necesitamos primero marcar como calle las casillas horizontales partiendo del primer vértice de la relación hasta que se alcance la altura en X del segundo vértice. Posteriormente marcamos como calle las casillas verticales partiendo del segundo vértice hasta alcanzar la altura en Y del primero, que producirá la unión de los vértices mediante calles.

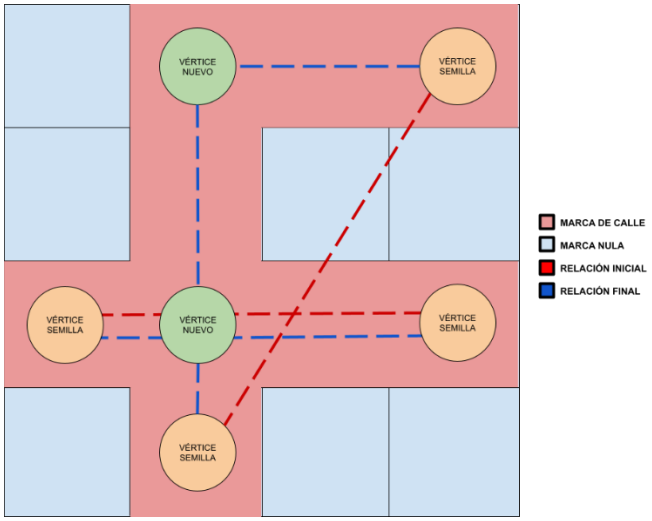

Ilustración 4.11: Generación de nuevos vértices auxiliares

Este proceso provoca nuevas curvas e intersecciones entre calles que no están reflejadas en el grafo tal y como se ve en la Ilustración 4.11 con los nuevos vértices verdes. Si utilizáramos el A* con un vehículo sin tener en cuenta esas nuevas intersecciones, nuestros vehículos no pasarían por ellas incluso si el camino es más corto, por lo que hay que ampliar la matriz de adyacencia. Esa ampliación se basa en sustituir la matriz de adyacencia actual por otra nueva donde se tengan en cuenta los nuevos vértices. Para ello se realiza un proceso inverso donde ahora tomando la matriz de profundidad 2 con la información posicional de calles obtenemos el nuevo grafo.

4.2.2.4 Matriz dispersa

La ampliación de la matriz de adyacencia anterior nos lleva a un punto de mayor incertidumbre acerca del posible tamaño de la matriz en ejecución. Si analizamos la matriz vemos que ésta está repleta de ceros y a lo sumo hay 4 unos en una fila o columna, por lo que es lógico pensar que sería buena idea cambiar de EEDD por otra más eficiente, pues para el A* lo que en verdad nos interesa es encontrar rápidamente los unos. Hasta ahora lo que se estaba usando para la matriz de adyacencia era una matriz de enteros, pero se terminó cambiando por una matriz dispersa [42], basada en listas de listas. En el pseudocódigo vemos de forma resumida los métodos más importantes de la EEDD, estructura que encontramos en el proyecto como *MatrizAdyacenciaSimetrica*.

```
MatrizAdyacenciaSimetrica
{
    matrizDispersa : List<List<Par>>
    length : Int

    struct Par
    {
        posicionJ : Int
        valor : Int
    }

    MatrizAdyacenciaSimetrica(tam : Int)
    {
        inicializa(matrizDispersa)
        length = tam
    }

    Par getParLista(posicionI : Int, iterador : Int)
    {
        return matrizDispersa[posicionI][iterador]
    }

    void insertar(posicionI Int, posicionJ Int, valor Int)
    {
```

```
25.         //Por eficiencia se asume que no se sobrescriben datos
26.         matrizDispersa[posicionI].Add(Par(posicionJ, valor))
27.         matrizDispersa[posicionJ].Add(Par(posicionI, valor))
28.     }
29. }
```

Teóricamente ahora se dan menos pasos para localizar las relaciones (unos) al no tener que iterar cientos de valores nulos (ceros). Por desgracia llevando a la práctica la matriz dispersa no ha dado los resultados deseados. Pese a que se dan menos pasos para encontrar los valores no nulos, es mucho más rápido iterar en arrays donde la memoria se encuentra guardada de forma consecutiva en memoria, que iterar en las listas, donde no lo están. Aunque no empeora el rendimiento, tampoco lo agiliza y es por ello que se ha dejado ya que sí que es posible que se note el aumento de eficiencia cuando la matriz de adyacencia crezca mucho. Por otro lado, hay que decir a favor de la EEDD que ahora se desperdicia menos espacio en memoria porque solo se guardan las posiciones donde hay relación.

4.2.3 Generación e instanciación de topología: ruido Perlin

Para crear el suelo crearemos un prefab que contenga el bloque base. Dicho prefab contendrá un componente *BoxCollider* [43] para que el jugador pueda pisar sobre él, un material pixelado para darle color y además el prefab será marcado como un elemento estático para hacer que Unity optimice la gestión del bloque inamovible. El tamaño de este bloque base corresponde con el de los bloques de profundidad 3 y será etiquetado como “Suelo”.

Ilustración 4.12: Bloque base

Una vez tenemos el bloque hay que estudiar donde se coloca y, al igual que en la generación de barrios, generar una topología de forma completamente aleatoria sería inviable. Es por esto que la generación de nuestra topología se ha llevado a cabo haciendo uso de la conocida técnica llamada Perlin noise [44] o ruido Perlin, utilizada así mismo por la generación procedural de Minecraft (aunque por supuesto en dicho juego la llevan al extremo).

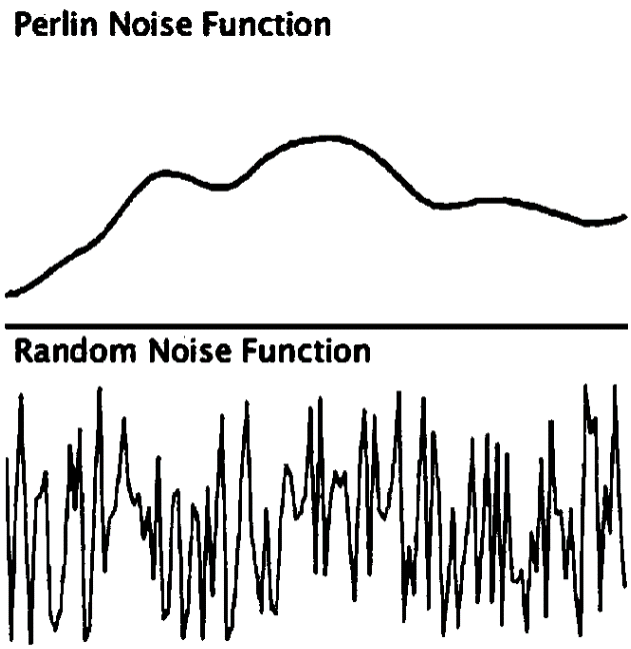

Ilustración 4.13: Ruido Perlin frente a números aleatorios

Este algoritmo fue creado en 1983 por el reconocido profesor en ciencias de la computación Ken Perlin y publicado en su artículo *An Image Synthesizer*. Básicamente esta función genera valores pseudoaleatorios entre 0 y 1 en forma de texturas, cuyos valores son suaves y continuos. Usaremos dichos valores como entradas en el mapa de alturas de nuestra ciudad.

Si instanciamos nuestros bloques base en grupos de 25 (5x5) por cada bloque de profundidad 2 con una altura ponderada por el valor devuelto del ruido Perlin, algo que viene ya implementado en Unity dentro de la biblioteca de funciones matemáticas *Mathf.PerlinNoise*, podemos obtener el siguiente resultado, el cual es bastante prometedor.

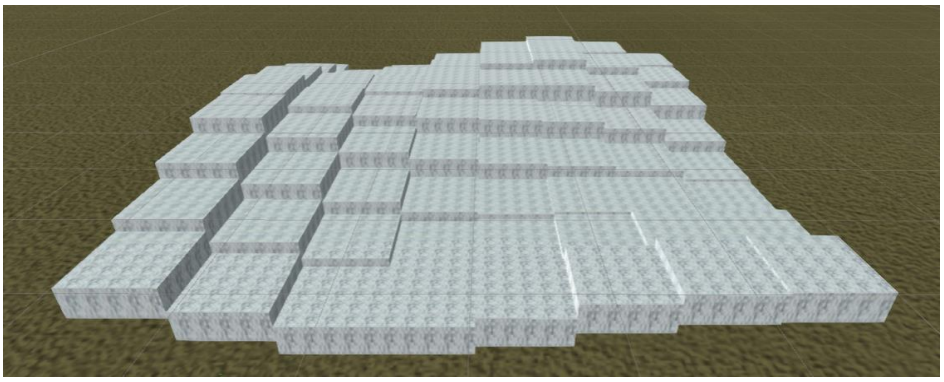

Ilustración 4.14: Topología pseudoaleatoria

Pero como es visible, el suelo está escalonado y los vehículos no son capaces de circular por el mismo. Para solucionar esto se han incluido rampas en los bordes donde hay cambio de alturas y se vaya a situar una calle transitable. Estas rampas serán nuevos bloques de profundidad 3 iguales a los que forman el suelo y etiquetados con el nombre de “Cuesta” que se superpondrán sobre el escalón y que tendrán una translación, una rotación y un escalado. Las cuestas o rampas seguirán el siguiente razonamiento trigonométrico, que aplica el teorema de Pitágoras, semejanza de Tales y trigonometría básica:

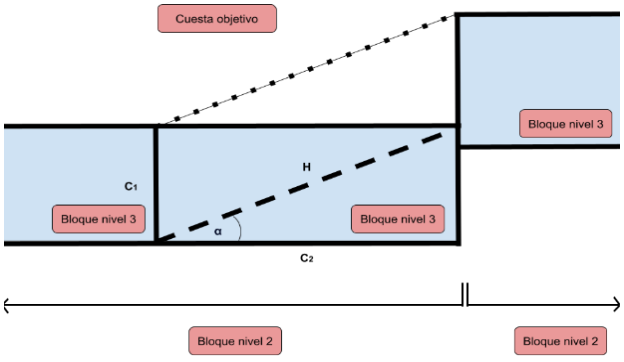

Ilustración 4.15: Cálculo de cuestas

- *Escalado (S, 1, 1) sea S =* $\frac{H}{C_2} = \frac{\sqrt{{C_1}^2+{C_2}^2}}{{C_2}^2}$
- *Rotación (0, 0, α) sea α =* $arcsen(\frac{C_1}{H}) = arcsen(\frac{C_1}{\sqrt{{C_1}^2+{C_2}^2}})$
- *Transformación (0,* $\frac{C_1}{2}$*, 0)*

4.2.4 Instanciación de calles

4.2.4.1 Prefabs utilizados

Para la instanciación de las calles sobre el nivel 3 de profundidad se han usado unos modelos con el estilo de píxeles del juego creados de forma propia usando Blender, pues no se han encontrado por Internet modelos gratuitos que satisficieran el estilo artístico. Éstos tienen un tamaño un poco mayor que el de la anchura de un bloque de nivel 3 de profundidad. La Ilustración 4.16 muestra el resultado de la creación de los 4 prefabs para su uso en Unity.

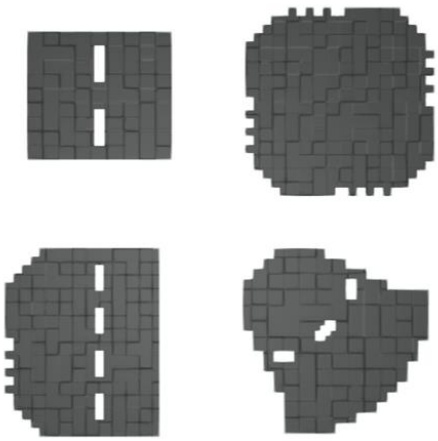

Ilustración 4.16: Modelos de calle

4.2.4.2 Instanciación

Estos prefab mostrados se colocarán sobre los bloques de profundidad 3 en función de las celdas colindantes a la celda en estudio de la matriz profundidad 2 marcada como calle, dándose las siguientes alternativas:

1. Celda derecha y celda izquierda marcadas como calles o celda superior y celda inferior marcadas como calles: se instancia una calle recta colocando 5 prefabs de calle recta alineados y de forma centrada.

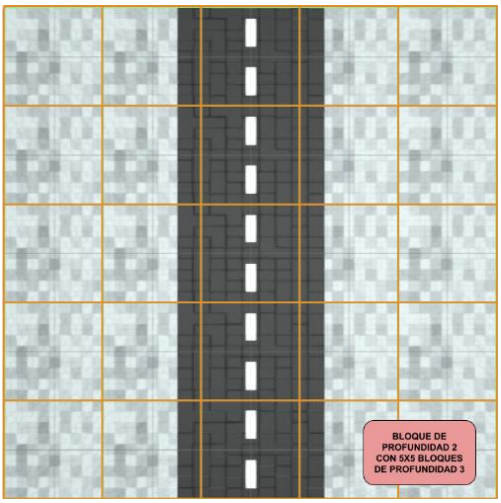

Ilustración 4.17: Calle recta en profundidad 2

2. Hay 2 celdas colindantes marcadas como calle y no son la celda derecha junto con la celda izquierda ni la celda superior junto con la celda inferior: se instancia una curva colocando 2 prefabs de calle recta alineados de forma centrada en una dirección, un prefab de calle curva en el centro y otros 2 prefabs de calle recta alineados de forma centrada en la otra dirección.

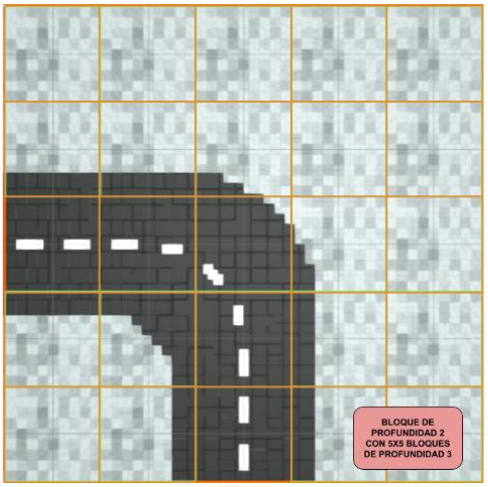

Ilustración 4.18: Curva en profundidad 2

3. Hay 3 celdas colindantes marcadas como calles: se instancia una intersección en forma de T colocando 4 prefabs de calle recta alineados de forma centrada en una dirección dejando el bloque de profundidad 3 central con un prefab de intersección en forma de T y otros 2 prefabs de calle recta alineados de forma centrada en la otra dirección.

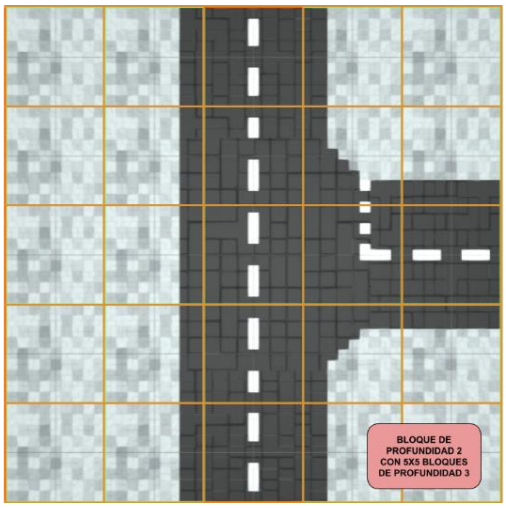

Ilustración 4.19: Intersección en profundidad 2

4. Todas las celdas colindantes están marcadas como calles: se instancia un cruce colocando 4 prefabs de calle recta alineados de forma centrada en ambas direcciones dejando el bloque de profundidad 3 central con un prefab de cruce.

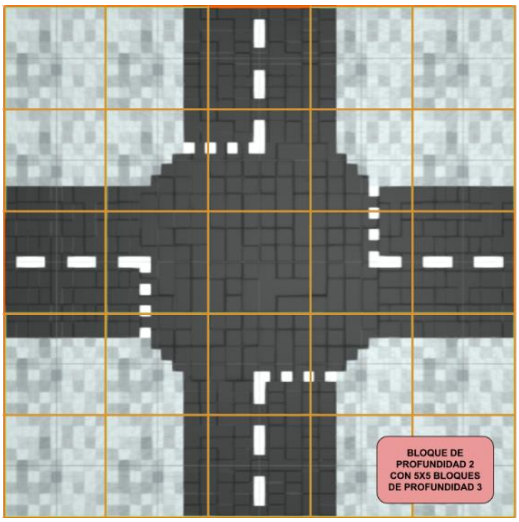

Ilustración 4.20: Cruce en profundidad 2

Estas calles además se marcan en la matriz de profundidad 3, indicando donde se sitúan y de que tipo son.

4.2.5 Instanciación de aceras

4.2.5.1 Prefabs utilizados

4.2.5.1.1 Árboles

Su prefab solo contiene el componente *BoxCollider* y la propia malla del modelo, obtenida de modelos de la web de Sketchfab. Es importante incluir este componente para conseguir que nuestro jugador pueda interaccionar con el árbol. Por otro lado, los *BoxCollider* proporcionan una colisión con una malla de hexaedro, que no corresponde con la forma real del árbol. Bien es cierto que se puede usar un componente *MeshCollider*, al que se le podría indicar que para gestionar las colisiones use la malla que define el objeto, en este caso no se usará. Esta decisión es porque en videojuegos suelen usarse mallas de pocos polígonos para evitar problemas de rendimiento, pues recordemos que trabajamos en tiempo real. En la Ilustración 4.21 pueden verse los 3 modelos de árbol que hay y concretamente en el árbol del centro vemos como es la caja de colisiones aproximada.

Ilustración 4.21: Árboles

4.2.5.1.2 Señales de tráfico

Estos prefabs han sido creados igualmente con modelos obtenidos de la Asset Store de Unity y etiquetados en tiempo de ejecución como “ObjetoAcera”. A estos además se les han añadido el componente de *RigidBody*. Gracias al *RigidBody* nuestro objeto podrá ser golpeado, ser afectado por la gravedad y tener un peso (que pondremos muy bajo para que puedan ser arrastrados con facilidad).

Ilustración 4.22: Señales

Cabe destacar que el componente de cuerpo rígido dio problemas cuando el objeto se situaba sobre una cuesta, porque provocaba la pérdida del equilibrio de los objetos y se caían debido a la pendiente. Para arreglar este problema se ha incluido a estos objetos el script *IgnorarCuestas* que haga que se ignore la colisión recibida por el método *OnCollisionEnter()* cuando el objeto tenga la etiqueta “Cuesta”, para así situarse en el bloque base sin pendiente que hay justo antes del escalón (recordemos que la cuesta se colocaba encima de éste).

4.2.5.1.3 Farolas

Este prefab es igual al de las señales, pero con la diferencia de que se le ha cambiado el centro de gravedad al punto más bajo de la farola, se le ha asignado un peso considerable y se le ha inmovilizado la posición desde los parámetros del cuerpo rígido. Gracias a esa configuración al chocar con una farola ésta se tumbará solo si le damos fuerte, pero sin desplazarse del sitio.

Ilustración 4.23: Farola

4.2.5.1.4 Mobiliario urbano

De nuevo estos prefabs se han creado a partir de los modelos descargados de la Asset Store y de Sketchfab. A estos les hemos aplicado los componentes de *BoxCollider* y *RigidBody* y asignando una etiqueta en tiempo de ejecución llamada "ObjetoAcera". Ninguno de estos dio problemas con el equilibrio.

Ilustración 4.24: Mobiliario urbano

4.2.5.2 Instanciación

Las aceras del juego se definirán en la matriz de profundidad 3 alrededor de todas las celdas marcadas como calle. Un objeto acera como tal es un *GameObject* vacío de Unity que generamos en tiempo de ejecución y al que le incluimos en su interior de forma aleatoria prefabs de señales, farolas, mobiliario urbano y árboles. Finalmente, dicho objeto se ubica junto a la calle y se orienta en la dirección que ésta marca. La siguiente imagen, Ilustración 4.25, muestra cómo queda el mapa tras añadirle tanto las calles como las aceras.

Ilustración 4.25: Ciudad con calles y aceras

4.2.6 Instanciación de edificios y límites

4.2.6.1 Prefabs utilizados

Inicialmente se trabajó con un conjunto de edificios descargados de la Asset Store, sin embargo, con el paso de tiempo se cambiaron por otros edificios obtenidos de Sketchfab que encajan mejor con el estilo artístico del juego. Aun así, ambas versiones son utilizables en el juego como ya se verá en el apartado 4.9.5.4 (Menú de ajustes).

Para ambos conjuntos de edificios se crearon prefabs para cada edificio con un componente *BoxCollider*, se incluyeron en la capa de renderizado con el nombre “Ciudad” y fueron marcados como objetos estáticos al igual que el bloque del suelo para aumentar rendimiento. Además, cada uno de los edificios ha sido clasificado en función de su tamaño en bloques de profundidad 3: 1x1, 2x1, 3x2, 5x3, 7x3 y 12x9. Normalmente no coinciden las escalas y se ha tenido que modificar ligeramente cada edificio para que ocupen el tamaño exacto y así evitar que haya huecos o intersecciones entre edificios a la hora de colocarlos.

4.2.6.1.1 Versión 1 de edificios

Cuando se preparó este conjunto de edificios no estaba implementado el algoritmo de generación de barrios y es por ello que todos los edificios pertenecen al mismo barrio.

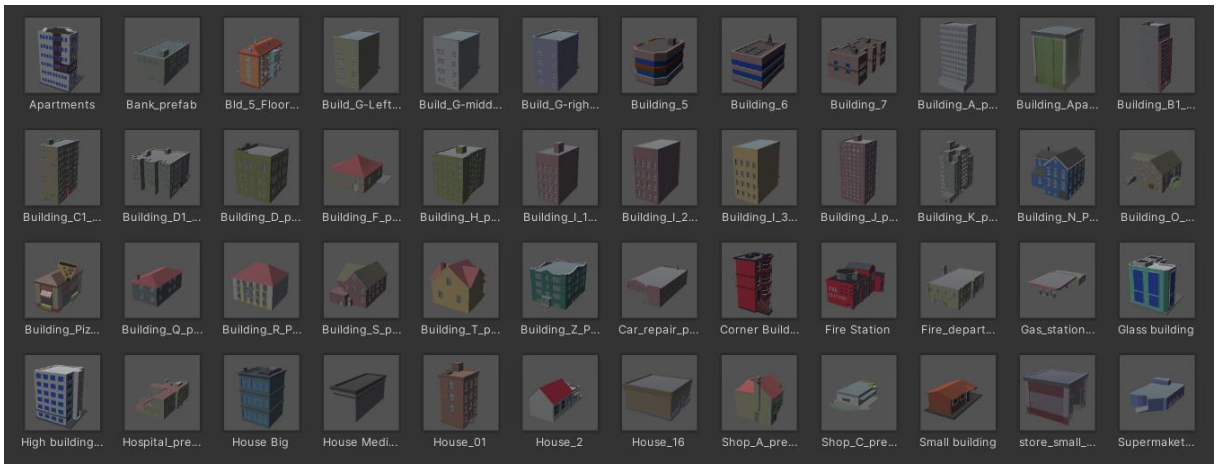

Ilustración 4.26: Versión de edificios low poly

4.2.6.1.2 Versión 2 de edificios

Para esta versión ya estaba implementado el algoritmo de generación de barrios y ya sí que se separaron los edificios en 4 grupos o barrios:

- Barrio egipcio

Ilustración 4.27: Edificios de barrio egipcio

- Barrio japonés

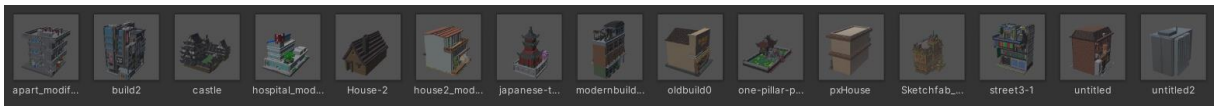

Ilustración 4.28: Edificios de barrio japonés

- Barrio antiguo

Ilustración 4.29: Edificios de barrio antiguo

- Barrio moderno

Ilustración 4.30: Edificios de barrio moderno

4.2.6.2 Instanciación

Para la instanciación de los edificios presentados tenemos que enviar al script *Generacion* un *GameObject* que contenga los edificios separados por barrios. Luego, ubicados en la matriz de profundidad 3, se irá leyendo cada celda, y cuando se encuentre una que sea apta para tener un edificio (no es acera, no es calle y no es otro edificio) se estudia el espacio que tiene disponible para colocar edificio. Una vez

se tiene el tamaño disponible se elige el tamaño del edificio a instanciar, cuya elección se hace aleatoriamente priorizando los tamaños más grandes (siempre que quepan en el espacio disponible). Una vez tenemos seleccionado el tamaño del edificio a elegir, tomamos un edificio de forma aleatoria de los pertenecientes al barrio que le corresponde por la posición. Finalmente se rota 0º, 90º, 180º o 270º para que no estén todos los edificios orientados hacia la misma dirección. A continuación, se muestra un pseudocódigo que resume brevemente el largo proceso.

```
ColocarObjetosEdificio()
{
    cuadrosProf3 = 5
    for (i -> matrizProf2.Size)
    for (j -> matrizProf2.Size)
    for (k1 -> cuadrosProf3)
    for (k2 -> cuadrosProf3)
        //Si la celda está vacía puede usarse para ubicar un edificio
        if (matrizProf3[i * cuadrosProf3 + k1][j * cuadrosProf3 + k2] == VACIO)
            barrio = matrizBarrios[i * cuadrosProf3 + k1][j * cuadrosProf3 + k2]
            //Edificios de 12x9
            if (cabeEdificio(i, j, k1, k2, 12, 9) && Random(probabilidad) == TRUE)
                colocarEdificio(i, j, k1, k2, 6, 12, 9, barrio)
            //Edificios de 7x3
            else if (cabeEdificio(i, j, k1, k2, 7, 3) && Random(probabilidad) == TRUE)
                colocarEdificio(i, j, k1, k2, 5, 7, 3, barrio)
            ...
            //Edificios de 1x1
            else
                colocarEdificio(i, j, k1, k2, 0, 1, 1, barrio)
}

Bool cabeEdificio(i, j, k1, k2, tamMaxSoportadoX, tamMaxSoportadoY : Int)
    cuadrosProf3 = 5
    for (x -> tamMaxSoportadoX)
        for (y -> tamMaxSoportadoY)
            if (i * cuadrosProf3 + k1 + x < matrizProf3.Size
             && j * cuadrosProf3 + k2 + y < matrizProf3.Size)
                if (matrizProf3[i*cuadrosProf3+k1+x][j*cuadrosProf3+k2+y] != VACIO)
                    return false
            else return false
    return true
}
```

Donde la función *colocarEdificio()* por un lado marca en la matriz de profundidad 3 la existencia de un edificio en las celdas que éste ocupará, y por otro lado mide la altura del suelo desde el punto más bajo localizado en el área que ocupa el edificio, para instanciarlo a dicha altura y que no se vean trozos de edificio levitando. Cabe destacar que esto último es un problema complejo, que no abarcaremos en este TFG más allá de lo que se ha hecho, pues habría también que localizar y modificarse aquellas fachadas de edificios que queden tapadas parcialmente por el terreno durante la propia generación para conseguir un resultado realista.

El resultado de añadir los edificios sería casi el resultado final, pero por desgracia para nosotros esto no es suficiente, pues la generación de las calles puede producir una generación con salidas puntuales del mapa trayendo la necesidad de añadir límites para evitar que el jugador pueda salir del mapa. Debajo podemos ver el problema comentado (Ilustración 4.31) y su solución (Ilustración 4.32).

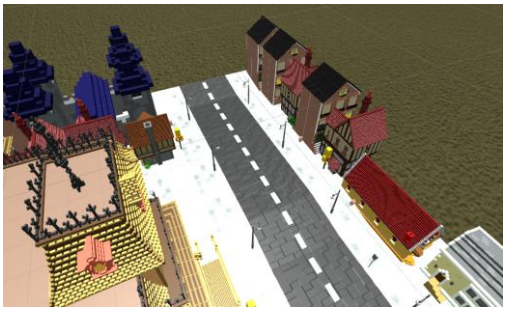

Ilustración 4.31: Ciudad sin límites

Ilustración 4.32: Ciudad con límites

La generación de límites simplemente añade bloques base con árboles encima y con una *BoxCollider* que no permite el paso del vehículo.

De esta forma ya tendríamos completo nuestro algoritmo de generación de ciudades, donde ya el jugador no puede salir de la ciudad y dando resultados como los de la Ilustración 4.33.

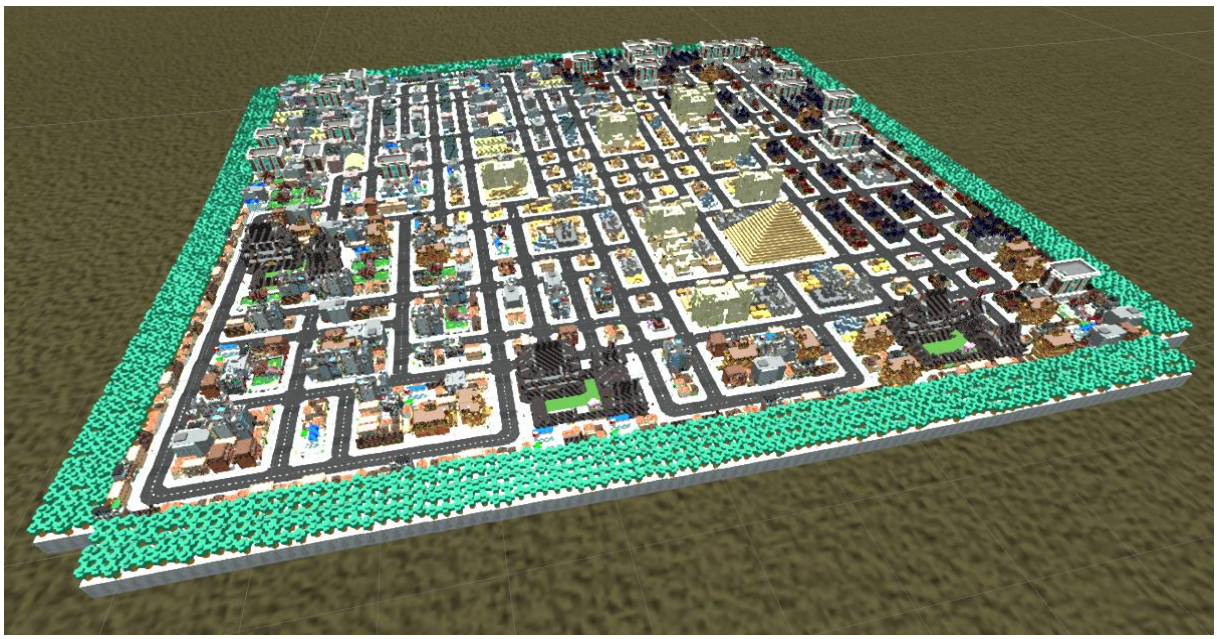

Ilustración 4.33: Resultado del algoritmo de generación procedural de ciudades

4.2.7 Mejoras de la ciudad: normal mapping

Los modelos del bloque base y calles como tal tienen muy poca geometría. El detalle del efecto de pixelado que se muestra en la superficie de éstos (y que ya se ha podido ver en imágenes anteriores) no se da mediante geometría sino mediante la técnica de normal mapping [45], una técnica ampliamente utilizada en el desarrollo de videojuegos para mejorar el rendimiento.

Normal mapping se fundamenta en el uso de un mapa de textura, normal map o mapa de normales, que almacena información sobre la orientación de la normal de la superficie en cada punto en forma de RGB y sirve para calcular cómo la luz interactúa en cada punto del modelo usando las normales del mapa. Esto permite añadir mayor número de detalles sin necesidad de ampliar geometría. El mapa de normales usado para los modelos ha sido creado mediante la herramienta NormalMapOnline [46], que tiene gran número de parámetros configurables.

Ilustración 4.34: Suelo sin normal mapping

Ilustración 4.35: Suelo con normal mapping

4.2.8 Pruebas

A continuación, se van a mostrar los resultados de los test realizados para el algoritmo de generación procedural de ciudades.

T-01 Test de generación de barrios	
Objetivo	El algoritmo debe generar barrios de manera pseudoaleatoria de tantos tipos como se le indique
Resultado	El algoritmo genera barrios correctamente de tantos tipos como se le indique siempre que sea posible por temas de espacio
Observaciones	Los barrios no reparten el área del mapa de manera equitativa

Tabla 4.5: Test de generación de barrios

T-02 Test de generación calles	
Objetivo	El algoritmo debe generar calles conexas y una matriz de adyacencia correcta
Resultado	El algoritmo genera correctamente calles conexas
Observaciones	Algunas calles pueden dar con el final del mapa y la matriz de adyacencia en su primera versión contenía errores que fueron corregidos

Tabla 4.6: Test de generación calles

T-03 Test de topología	
Objetivo	El algoritmo debe generar un terreno con pendientes pseudoaleatorias
Resultado	El algoritmo genera correctamente pendientes
Observaciones	Si se indica una pendiente demasiado elevada, aunque no dé errores, sí que dejaría de ser realista, por lo que es conveniente capar el ángulo máximo de la misma.

Tabla 4.7: Test de topología

T-04 Test de instanciación	
Objetivo	El algoritmo debe generar edificios y objetos sobre la calle de manera lógica
Resultado	El algoritmo genera correctamente los edificios y todo el mobiliario urbano, evitando que objetos delgados como farolas caigan debido a la pendiente donde se ubican
Observaciones	Las señales de tráfico no siguen una lógica y únicamente son objetos decorativos debido al contexto de este proyecto

Tabla 4.8: Test de instanciación

T-05 Test de límites del mapa	
Objetivo	El jugador no podrá salir de la ciudad desde una carretera
Resultado	El algoritmo genera correctamente árboles que actúan de límites del mapa para que el jugador no salga del mismo
Observaciones	

Tabla 4.9: Test de límites del mapa

4.3 Navegación

Unity trae numerosas bibliotecas y componentes que nos ayudan a ahorrar tiempo dentro del desarrollo de un videojuego. Concretamente el motor ya nos proporciona librerías de inteligencia artificial con las que podemos crear mallas de navegación y agentes de navegación que circulen sobre ellas.

4.3.1 Método NavMesh y NavMeshAgent de Unity

Tanto *NavMesh* [47] como *NavMeshAgent* [48] son dos herramientas que desde luego hay que tener en cuenta a la hora de hacer el juego debido a la cantidad de trabajo que nos pueden llegar a ahorrar. En concreto, *NavMesh* se trata de una malla navegable autogenerada por Unity que se adapta al terreno tridimensional de nuestro juego. Por otro lado, *NavMeshNavigator* es un componente que se añade a un objeto para que éste se mueva de forma inteligente encima de la malla navegable.

Para generar esta malla hay que ir a *Windows* > *AI* > *Navigation*. Una vez ahí se abrirá la pestaña *Navigation* que tiene el aspecto siguiente.

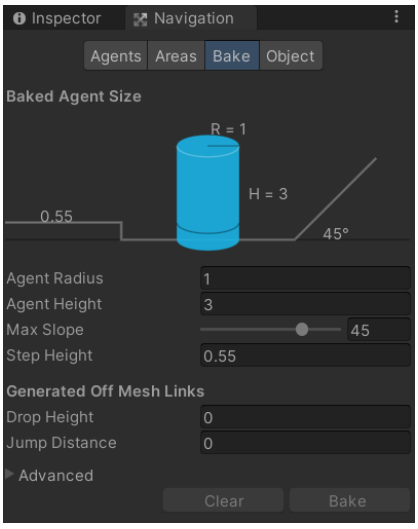

Ilustración 4.36: Creación de malla

Como vemos en la Ilustración 4.36, se nos pide llevar a cabo la configuración de la malla navegable con parámetros como el radio que ocupa nuestro personaje, su altura, el ángulo máximo de pendiente que admite subir, el escalón que permite saltar sin que haya una rampa, etc. Una vez hemos configurado esto le daríamos al botón

Bake y se generaría la malla. Sin embargo, la generación mostrada es para crear mallas fuera del tiempo de ejecución.

Como nuestro juego genera el terreno de forma procedural al arrancar, tenemos que buscar la manera de decirle a Unity que haga el trabajo del horneo de la malla mediante un script, pero por defecto los componentes necesarios no están instalados en Unity. Es por esta razón que debemos descargar una serie de componentes del paquete de Engine.AI. adicionales [49]. Una vez instalados necesitaremos usar el componente llamado *NavMeshSurface* [50] y hacer ejecutar el siguiente script cuando tengamos generado todo el terreno al que queremos añadirle una malla.

```
1. public class GeneradorMalla : MonoBehaviour
2. {
3.     public void generar()
4.     {
5.         // Creamos componente navmesh y construimos la malla en tiempo de ejecución
6.         gameObject.AddComponent<NavMeshSurface>().BuildNavMesh();
7.     }
8. }
```

Gracias a este script podríamos obtener resultados de malla navegable tan buenos como los de la Ilustración 4.37 (donde se puede apreciar la zona navegable con tonos más azulados) para aplicarlos a vehículos con el componente *NavMeshAgent*.

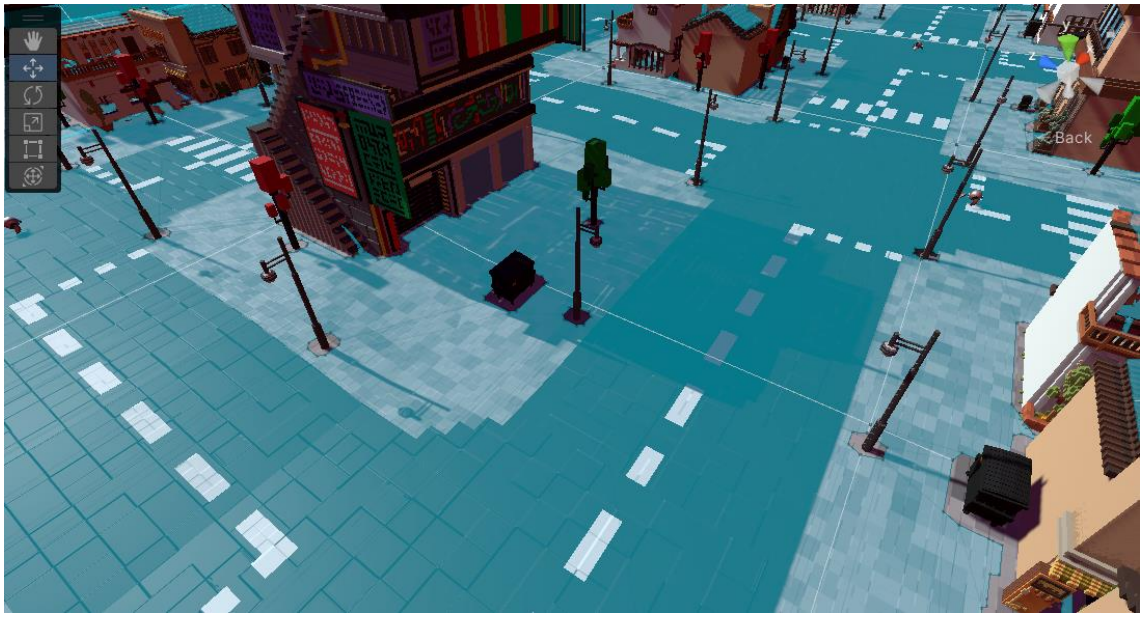

Ilustración 4.37: Malla de navegación en tiempo de ejecución

4.3.2 Método GPS con A* propio

Otra opción es la de programar el algoritmo A* [28] y utilizar para los cálculos el grafo devuelto por el algoritmo de generación procedural de ciudades (que como es conexo no dará problemas). Para ello ya no será necesario de crear una malla, lo cual es una ventaja en cuanto a coste computacional. El algoritmo A* programado podemos resumirlo con el siguiente pseudocódigo:

```
List<Nodo> AEstrella(origen, destino : Vector2, matrizAdyacencia : MatrizAdyacencia)
{
      nodosAbiertos : List<Nodo>
      nodosCerrados : List<Nodo>
      nodosAbiertos.Add(Nodo(origen, coste()))
      while (!nodosAbiertos.Empty)
          //Elegir nodo de menor coste f(n) en el conjunto abiertos
          for (i -> nodosAbiertos.Size)
              if (nodosAbiertos[i].coste < min) nodoActual = nodosAbiertos[i]
          //Si el nodo elegido es el nodo final, se ha encontrado el camino
          if ((nodoActual.position - destino).Magnitude == 0) break
          //Mover el nodo elegido del conjunto de abiertos al de cerrados
          nodosAbiertos.RemoveAt(nodoActual)
          nodosCerrados.Add(nodoActual)
          foreach(nodoAdyacente in matrizAdyacencia.adyacentes(nodoActual))
              adyacente = Nodo(nodoAdyacente, coste(), nodoActual)
              // Si no era nodo abierto, se incluye en lista de abiertos
              if (!nodosAbiertos.ContieneNodoIgual(adyacente)) nodosAbiertos.Add(adyacente)
              //Si si lo era, comprobamos si la ruta necesita ser actualizada
              else
                  temp = nodosAbiertos.NodoIgual(adyacente)
                  if (temp.coste > adyacente.coste)
                      nodosAbiertos.Remove(temp)
                      nodosAbiertos.Add(adyacente)
      //Se rescata y devuelve la ruta
      ruta.Add(nodoActual)
      do
          nodoActual = nodoActual.padre
          ruta.Add(nodoActual)
      while (nodoActual.padre)
      return ruta.Reverse()
}

Nodo{
      posicion : Vector2
      coste : Float
      padre : Nodo
      Nodo(posicion : Vector2, coste : Float, padre = null : Nodo){...}
}
```

Este algoritmo A* será controlado mediante otra clase llamada *GPS*, que será lo que se le añadirá como componente a los vehículos. La clase guarda en sus atributos información relativa a la ciudad y un atributo de ruta que se va actualizando dentro del método repetitivo *Update()* cuando el objeto se mueve lo suficiente como para que haya posibilidad de que la ruta cambie. Entre el funcionamiento más

destacado de la clase y sin entrar mucho en detalles hay que comentar las siguientes funciones:

- *calcularRuta()*: tiene como parámetros la posición de destino al que se quiere ir y se ocupa de llamar al algoritmo A* para que éste calcule una ruta a seguir.
- *getRuta()*: devuelve el listado de objetos *Nodo* del grafo de calles que deben seguirse para llegar al destino. La clase *Nodo* permite conocer la ubicación del mismo en el espacio 3D del mundo.
- *getDirecciónRuta()*: devuelve en *Quaternions* el ángulo de giro que debe realizarse en el momento actual para tomar la calle que sigue la ruta. Este método es la clave del funcionamiento de la flecha indicadora que se ubica sobre el vehículo del jugador, desarrollada dentro del apartado 4.9.5.7.6 (Flecha indicadora).

4.3.3 Ventajas de nuestro sistema frente al de Unity

Una vez puestas ambas posibilidades sobre la mesa, tenemos que determinar si nuestro trabajo extra está justificado con el éxito de la eficiencia.

Definiendo como nuestro sistema aquel que hemos creado con un A* que hace uso del grafo de calles de nuestra ciudad, y el sistema de Unity aquel que usa el componente *NavMeshAgent* que incorpora un A* y que hace uso de la malla navegable generada por un componente *NavMesh*, lo que se hará es realizar varias pruebas de tiempo, en las que se verán los tiempos de carga y la velocidad de cálculo de rutas para ambos sistemas.

En primer lugar, cabe destacar que el proceso de generación de mallas es un proceso costoso, que puede demorar bastante tiempo en función de la complejidad del terreno. A continuación, vamos a ver una gráfica, Ilustración 4.38, que muestra el tiempo de carga (en segundos) que ha habido con 5 configuraciones diferentes, cada una con un tamaño de ciudad superior a la anterior.

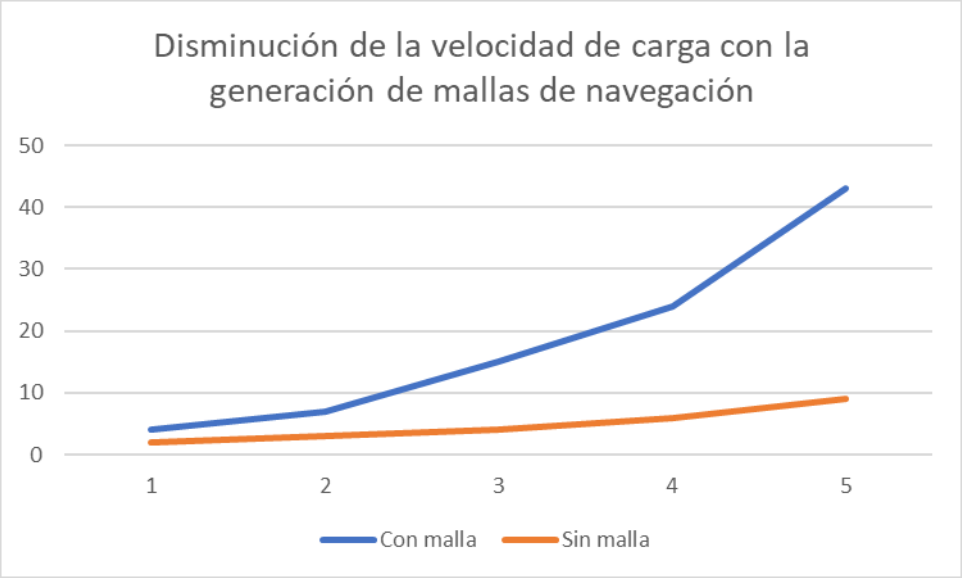

Ilustración 4.38: Crecimiento del tiempo por la generación de malla navegable

Véase cómo la línea azul, que muestra el tiempo de carga cuando se genera malla navegable, crece de forma excesivamente rápida y de forma no lineal frente al crecimiento que lleva la línea naranja, que es la que muestra el tiempo necesario para cargar el juego si no se usa la malla navegable (es decir, nuestro sistema).

Respecto al tiempo de generación de rutas nuestro algoritmo devuelve valores de forma instantánea conforme se le pide, mientras que los *NavMeshAgents* de Unity llegan a tardar hasta un par de segundos. Pese a que ambos métodos usan el algoritmo A*, el nuestro a simple vista busca las rutas en menos tiempo. La diferencia entre tiempos solo puede deberse a una cosa, y es el número de nodos que usan los diferentes sistemas.

- El número de nodos en una *NavMesh* generada por Unity está determinado por la cantidad de triángulos que se utilizan para representar el terreno. Para ver el número de triángulos que tiene la malla en Unity podemos utilizar la siguiente línea de código:

```
1.  NavMesh.CalculateTriangulation().areas.Length;
```

- El número de nodos que genera nuestro algoritmo de generación de ciudades puede comprobarse igualmente desde el código haciendo uso del atributo *numeroNodos* de nuestra clase *Generacion*, desarrollada en el apartado 4.2 (Ciudad con generación procedural).

Ahora vamos a ver la diferencia de nodos que hay cuando hacemos uso de la malla generada por Unity y cuando hacemos uso del grafo devuelto por nuestro algoritmo generador de ciudades. Fijémonos en la Tabla 4.10.

Configuración	Nº nodos en grafo de calles	Nº nodos NavMesh Unity
1	59	152488
2	174	323579
3	443	648863
4	946	1123819
5	1741	1879760

Tabla 4.10: Comparativa de número de nodos

Como puede verse, los resultados son aplastantes a favor de nuestro método y es claro que hay una consecuencia clara en la eficiencia del mismo.

En resumen, utilizar un algoritmo A* personalizado que incluye en sus cálculos el grafo devuelto por el algoritmo de generación de ciudades es mucho más óptimo que utilizar el componente NavMesh y el componente NavMeshAgent proporcionados por Unity. En primer lugar, si utilizamos la versión personalizada nos ahorramos el tiempo de creación de la malla por completo, mientras que si queremos usar el método facilitado por Unity tendremos que esperar a que ésta se calcule. En segundo lugar, la velocidad del algoritmo es superior cuando se hace uso del grafo devuelto por nuestro algoritmo de generación de ciudades que cuando se hace uso de la malla generada por Unity debido a la gran diferencia de nodos.

4.3.4 Pruebas

A continuación, se van a mostrar los resultados de los test realizados para el componente GPS.

T-01 Test de generación rutas	
Objetivo	El GPS debe ser capaz de calcular una ruta de distancia mínima y poder transmitirle las indicaciones necesarias a un vehículo para que éste pueda seguir las rutas
Resultado	El GPS creado permite calcular la ruta e indicar al vehículo que lo usa la dirección correspondiente. Además, si no se sigue la ruta el GPS lo detecta y genera una nueva ruta
Observaciones	Si el grafo de la ciudad no es conexo no es capaz de devolver una ruta

Tabla 4.11: Test de generación rutas

4.4 Personajes NPC

Durante esta sección hablaremos sin duda de uno de los puntos más fuertes del proyecto. Efectivamente nos referimos a las personas NPC dentro de nuestro videojuego y a lo largo de los siguientes apartados desarrollaremos los comportamientos, movimientos, objetivos y habilidades de éstos.

Ilustración 4.39: Personajes del juego

4.4.1 Estructura de los peatones NPC

Estos modelos tienen numerosas animaciones, y hacerlas una a una habría dedicado muchísimo tiempo, por lo que han sido animados mediante animaciones obtenidas de la página de Mixamo [51] desde donde se ha hecho el proceso de rigging [52] en los casos donde no había uno disponible.

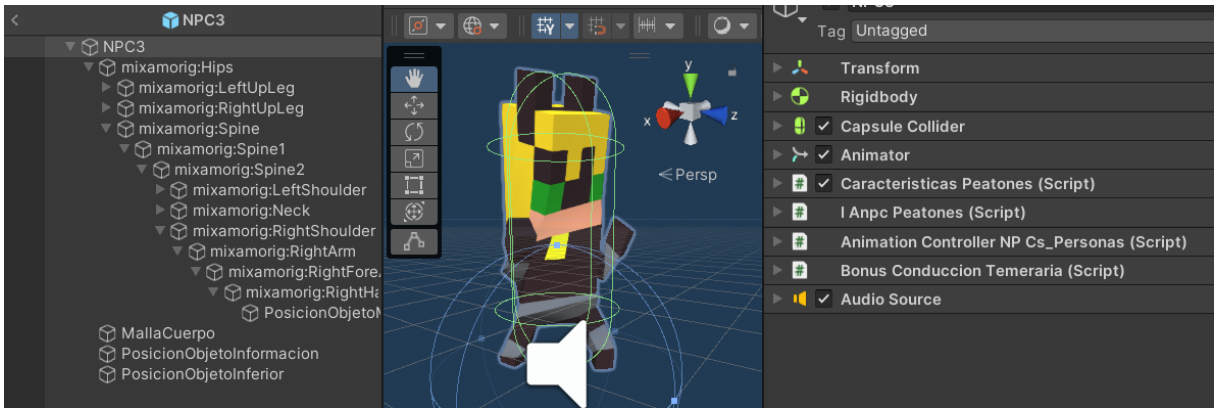

Ilustración 4.40: Estructura de un peatón NPC

4.4.1.1 Partes del humanoide

En Ilustración 4.40 podemos apreciar del prefab de los peatones y al que nos referiremos durante este apartado. Éste se compone de múltiples *GameObjects* que corresponden exactamente con las partes que definen a todo humanoide [52]:

- **Esqueleto o jerarquía de articulaciones:** se trata del primer *GameObject,* con nombre *mixamo:Hips*, del que cuelga una gran jerarquía que no he desplegado. Ese esqueleto se ha conseguido como resultado del proceso de rigging realizado desde la página de Mixamo por lo que no comentaremos demasiado al respecto.
- **Modelo de cuerpo 3D o malla de cuerpo:** es el siguiente *GameObject* que encontramos. Nos permite ver a nuestros NPCs y es en él donde se lleva a cabo la deformación según el modelo de piel y la animación impuesta.
- **Modelo de piel o Skin:** es fundamental para que la malla del cuerpo reaccione a los movimientos realizados por los huesos del esqueleto. Tampoco entraremos mucho en detalle y comentaremos únicamente que también se ha creado desde la página de Mixamo mediante el proceso de skinning [52].

Finalmente, todos los NPCs se han preparado para tener la capacidad de llevar objetos, encontrando 3 posiciones para estos, llamadas sites [53] si nos ponemos técnicos, que entrarán como atributos del componente *CaracteristicasPeatones*:

- **Site para objeto en mano derecha**: cuelga al final de toda la jerarquía del *GameObject* generado por Mixamo y recibe el nombre de

PosicionObjetoMano, siendo la posición donde se situarán los objetos que coja un NPC, como por ejemplo un arma (con esto conseguimos que el objeto se mueva acorde con la animación). Consideramos que los objetos de esta mano existen como tal dentro del mundo del juego.

- **Site para objeto bajo el personaje**: la posición *PosicionObjetoInferior* se utiliza sobre todo para ubicar el aura de los peatones con comportamiento de cliente, sin embargo, puede utilizarse para poner cualquier tipo de objeto en su lugar. Consideramos que los objetos encontrados en esta posición son objetos que solo informan al jugador, y como tal no existen dentro del mundo del juego, en otras palabras, aquello que se coloque aquí debe ser un elemento no diegético.
- **Site para objeto sobre el personaje**: la principal meta de la posición *PosicionObjetoInformacion* es la de ubicar elementos informativos como pueden ser emoticonos, etiquetas o símbolos. Consideramos que los objetos encontrados en esta posición también son objetos que solo informan al jugador, y por tanto tampoco no existen dentro del mundo del juego, en otras palabras, aquello que se coloque aquí debe ser un elemento no diegético.

4.4.1.2 Máquina de estados para la animación

Respecto a los componentes de Unity incluidos en el prefab, regresando a la Ilustración 4.40 vemos que hay un componente *RigidBody* y otro *CapsuleCollider* [54], para que el NPC actúe con físicas y tenga una caja de colisiones, algo que hasta ahora era común, con la diferencia de que ahora encontramos el componente *Animator* [38]. El componente *Animator* se ocupa de dirigir la animación del personaje mediante un *AnimatorController* [55], que se trata de una máquina de estados que interconecta todos los clips de animación del personaje.

Para hacer la máquina de estados de animaciones del *AnimatorController* se ha creado una variable que indica un estado, y en función de ella pasamos de una animación a otra. En un inicio, y en parte por desconocimiento, se empezó haciendo una máquina donde se indicaba para cada animación los posibles cambios hacia otra animación dependiendo del estado, algo que es una buena idea y muy intuitivo cuando

tenemos pocas animaciones que interconectar. Por desgracia cuando crece mucho la cantidad de animaciones, los resultados de esta filosofía son inmanejables tal y como se ve en la Ilustración 4.41, que muestra una de las primeras versiones donde ni siquiera estaban insertadas todas las animaciones.

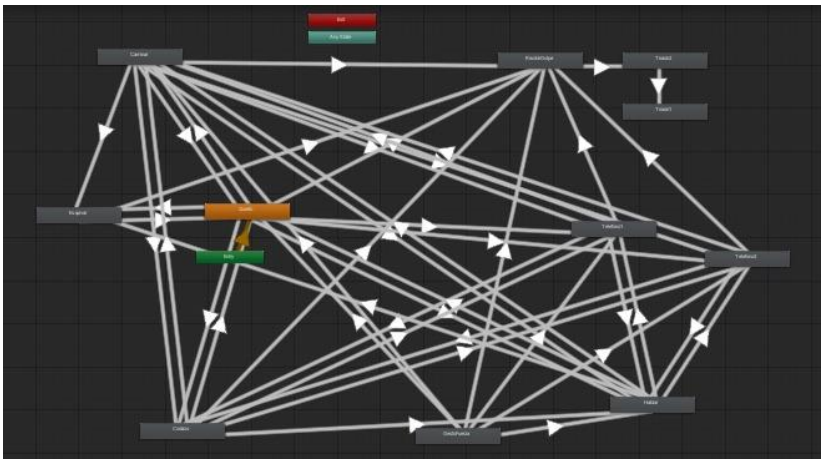

Ilustración 4.41: Primer intento de máquina de estados

La solución fue aprender a hacer uso del estado *AnyState*, incluir 2 variables para conocer el estado anterior (llamada *EstadoAnterior*) y actual (llamada *EstadoActual*) en lugar de una sola como hasta ahora, y utilizar múltiples condicionales a la vez para pasar de una animación a otra. Esta serie de modificaciones han dado una máquina de estados mucho más fácil de manejar y actualizar, aunque conceptualmente más compleja, visible desde la Ilustración 4.42.

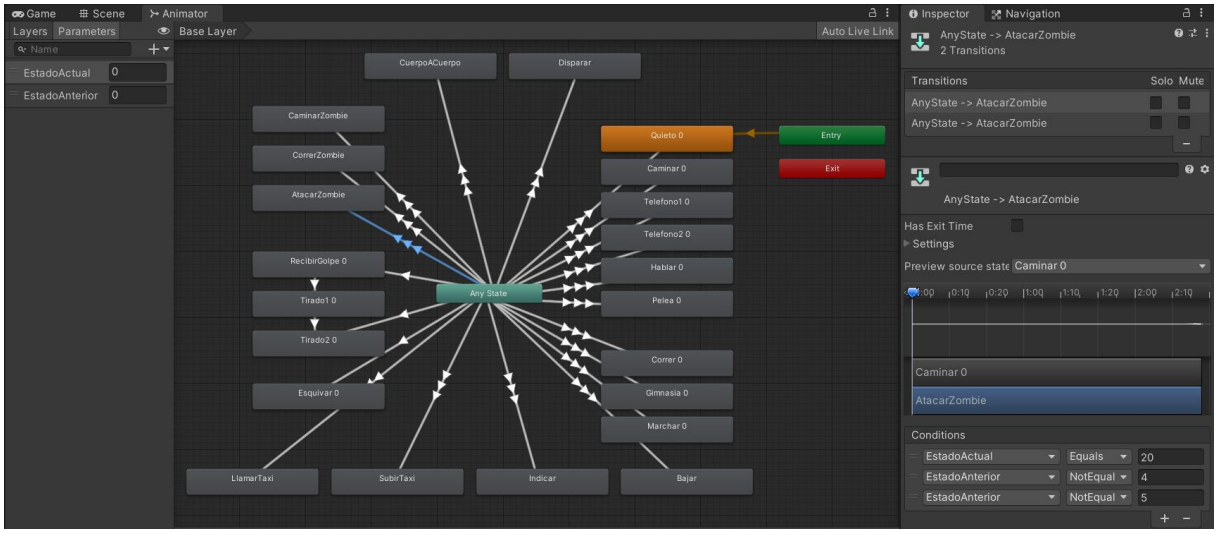

Ilustración 4.42: Máquina de estados final

En la máquina de estados de animaciones final cada relación con múltiples triángulos indica que tiene más de una transición que precede al estado de animación. Cada transición se da siguiendo una serie de condicionales, en el ejemplo marcado en la Ilustración 4.42 la animación de ataque zombi puede alcanzarse desde cualquier estado mediante 2 transiciones, y una de ellas se da cuando el estado anterior no es un esquivo (animación *Esquivar* cuyo valor es 5) o un derribo (animación *RecibirGolpe* cuyo valor es 4) y el estado actual es el de ataque de zombi (animación *AtacarZombie* cuyo valor es 20). La otra transición que no podemos ver en la imagen sería similar, solo que tendría en cuenta si el estado anterior fue un esquivo y el estado actual un ataque zombi. No entraremos mucho más en profundidad para evitar extender demasiado la memoria, pero sí que comentaremos que el funcionamiento para pasar al resto de estados de animación es similar, cada uno con sus respectivas transiciones y condiciones.

El proceso de montar la máquina de estados para el primer personaje ha sido extenso, pero una vez creada, puede reutilizarse para el resto de peatones. Cabe destacar que cada peatón tiene su propia personalidad y por ende sus propios clips de animación, elegidos cuidadosamente y con paciencia. De esta manera se consigue que cada personaje tenga su propia forma de caminar, correr, hablar, etc. Cara a hacer la máquina de estados de cada personaje esto último no es un problema, pues como decíamos podemos reutilizarla duplicando la máquina para el resto de NPCs, indicando para cada estado de animación el clip de animación característico de cada peatón.

4.4.1.3 Scripts del NPC

Mirando nuevamente la Ilustración 4.40 vemos, que al igual que para el jugador con *CaracteristicasJugador*, todos los peatones cuentan con un componente de características llamado *CaracteristicasPeatones* (explicado en apartado 4.6.1.1). Añadido a eso hay un segundo componente, el de *BonusConduccionTemeraria* (explicado en apartado 4.6.3.1) que da bonificaciones al jugador cuando el NPC tiene que esquivar el taxi, algo que se desarrollará cuando lleguemos al apartado 4.4.2.1.4 (Actividad de esquivar).

Relacionado con el comportamiento tenemos los componentes de *IAnpcPeatones*, que debido a la importancia del script se detallará en el siguiente punto, y *AnimationControllerNPCs_Personas*, que cambia las variables *EstadoAnterior* y *EstadoActual* de la máquina de estados del *AnimatorController*, de forma completamente desacoplada, a la orden del script de comportamiento correspondiente.

4.4.2 IA de peatones NPC: patrón estrategia

En función de su comportamiento, un peatón NPC puede comportarse como:

- Peatón con comportamiento individual.
- Peatón con comportamiento de cliente.
- Peatón con comportamiento grupal.
- Peatón con comportamiento de policía.
- Peatón con comportamiento de zombi.

Aunque todo peatón también puede cambiar su comportamiento en tiempo de ejecución, con la finalidad de pasar a tener un comportamiento de cliente a comportamiento individual, pasar a comportamiento de zombi, o cualquiera de sus posibles combinaciones.

Es por esta razón que se han creado múltiples scripts de comportamiento y se aplicado el patrón estrategia [37] para relacionarlos tal y como se explica en el apartado 4.10.2 (Patrón Estrategia aplicado al comportamiento de peatones).

El UML inicial como es de esperar dada la especificación detallada del comportamiento de los peatones y el patrón que se seguirá, se debe actualizar para poder asumir estos cambios tan importantes en lo que se refiere a las personas NPC, siguiendo ahora la idea mostrada en la Ilustración 4.43 para el fragmento de diagrama correspondiente.

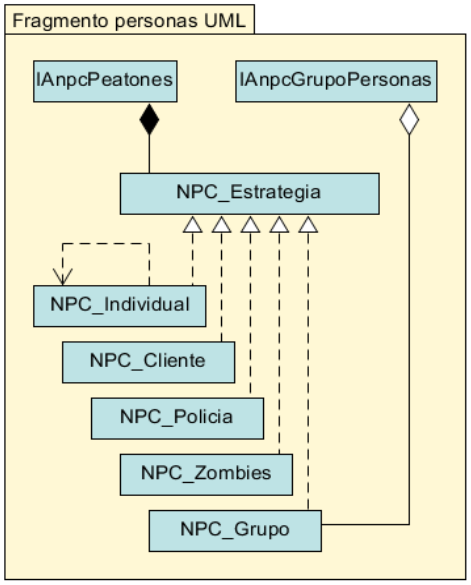

Ilustración 4.43: Especificación de UML para personas NPC

4.4.2.1 Comportamiento compartido: herencia

Aprovechando la interfaz que se requiere para la aplicación del patrón estrategia se ha incluido una serie de métodos para actividades que desde cualquier comportamiento se pueden realizar y que ahora procederemos a explicar.

4.4.2.1.1 Actividad de estar quieto

Es el comportamiento más básico y fácil de realizar, pues lo único que se hace es mantener la posición del NPC y pedir al *AnimationControllerNPCs_Personas* que se active la animación de estar quieto. Ciertamente encontrar un personaje en la ciudad con este estado es poco común ya que solo podemos verlo cuando un grupo de peatones es prácticamente disuelto, quedando una única persona.

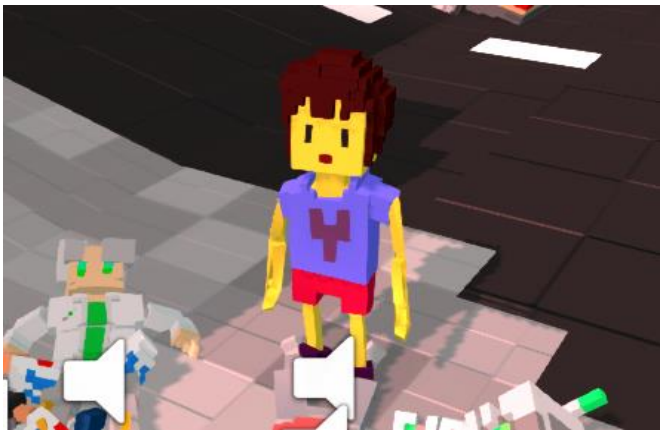

Ilustración 4.44: Actividad de estar quieto

4.4.2.1.2 Actividad de caminar

Es una actividad muy recurrente, cuyo funcionamiento se resume en girar al personaje hacia la posición a la que se dirige, cambiar su posición y pedir que se active la animación de caminar.

El caminar de las personas no está basado en una malla de navegación ni tampoco en el componente *NavMeshAgent*. Es por ello que el desplazamiento se define mediante puntos de destino que los NPCs van asignando sobre la acera con ayuda de funciones del algoritmo de generación, explicado en el apartado 4.2 (Ciudad con generación procedural), y cambiando dichos puntos conforme llegan por otros. Dicho desplazamiento además es implementado gracias a la función *MovePosition()* del componente *RigidBody*, pues de hacer cambios directamente sobre la transformada provocaría fallos sobre las físicas.

Dado que una acera no puede dividirse en diferentes direcciones, no será necesario hacer uso del *GPS*, algo que nos beneficia porque sobrecargamos menos el sistema. Esto es así porque en una acera solo hay una dirección en la que caminar debido a que no aceptamos dar la vuelta e ir hacia atrás.

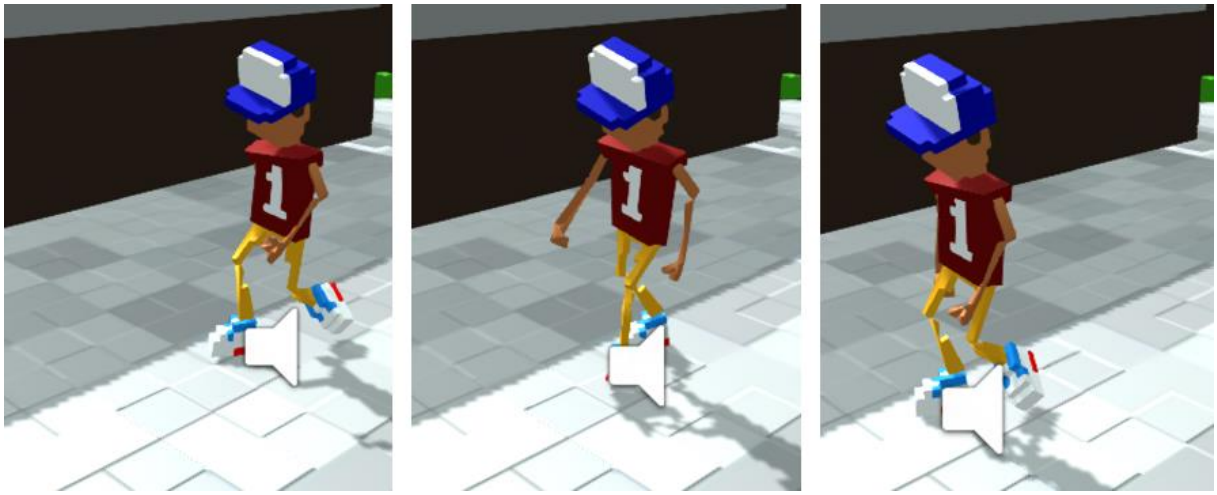

Ilustración 4.45: Actividad de caminar

4.4.2.1.3 Actividad tras ser abatido

Es una actividad que se llama desde la clase de *CaracteristicasPeatones* cuando se detecta que la vida del NPC ha bajado hasta el valor 0, algo que puede suceder al recibir un disparo, explosión, atropello, una derrota en pelea callejera o

cualquier colisión lo suficientemente fuerte. Si la caída se produce por una colisión, el NPC realiza además un estudio de la posición del objeto que activó su método *OnCollisionEnter()* y cae mirando hacia el mismo.

Esta actividad se basa en cambiar la animación a una secuencia de animaciones donde se ve al personaje recibir el golpe, caer y quedar aturdido en el suelo (esta última queda activada de forma cíclica hasta la desaparición del NPC). Luego eliminamos la información que pueda darnos el NPC eliminando todos los objetos informativos no diegéticos que tuviera asociados en las posiciones de objetos vistas en el apartado 4.4.1.1 (Partes del humanoide), de forma que por ejemplo un arma en la mano no sería eliminada, pero sí un emoticono sobre su cabeza. Finalmente se desactiva la caja de colisiones tras un par de segundos para no molestar al jugador mediante la función *IgnoreCollision()* que nos proporcionan las físicas de Unity.

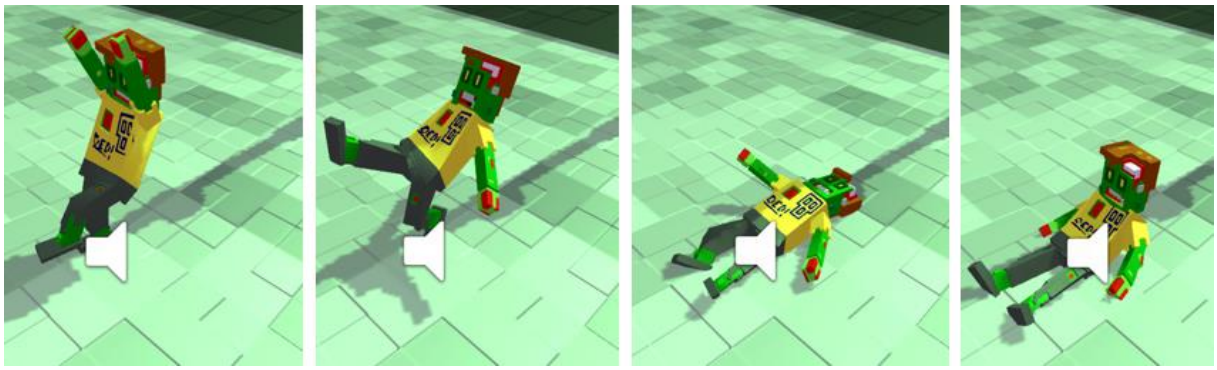

Ilustración 4.46: Actividad de ser abatido

4.4.2.1.4 Actividad de esquivar

No es exactamente una actividad sino más bien un movimiento auxiliar con animación que se da entre la ejecución de otra actividad, detenida temporalmente hasta que termina el movimiento de esquivo.

Un NPC es capaz de esquivar el vehículo del jugador durante cualquier actividad si nota que éste le pasa muy cerca y supera una velocidad específica, algo que es fácil de reconocer accediendo a la transformada y al atributo *velocity* del *RigidBody* del jugador. Una vez se activa el movimiento para esquivar, se reproduce un sonido de grito y se cambia la animación a una en la que el NPC salta. Añadido a

esto se realiza un cálculo para determinar hacia donde salta. Observemos la Ilustración 4.47 y la Ilustración 4.48.

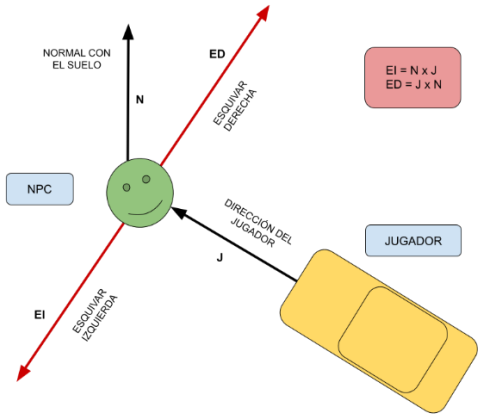

Ilustración 4.47: Direcciones de esquivo

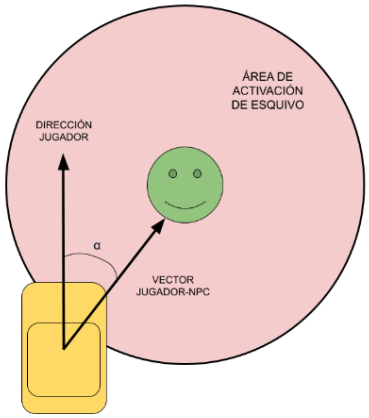

Ilustración 4.48: Selección de dirección

El NPC saltará hacia la derecha o hacia la izquierda en función del ángulo que crean los vectores de la dirección del jugador con el vector que une la posición del jugador con el NPC, por tanto:

- Si alfa es mayor que 0: el NPC saltará hacia la derecha. Para ello hay que calcular el producto vectorial entre la normal del suelo y la dirección del vehículo.
- Si alfa es menor que 0: el NPC saltará hacia la izquierda. Para ello hay que calcular el producto vectorial entre la dirección del vehículo y la normal del suelo.

Cabe mencionar que, sea o no atropellado, el NPC llamará a la policía y dará el *BonusConduccionTemeraria* al jugador cuando tenga que esquivarlo. La llamada a la policía será mediante el método *llamarPolicia()*, perteneciente al *Gestor de vehículos NPC* (que veremos más en detalle en el apartado 4.8.1), el cual, con una baja probabilidad (que aumenta en función de la dificultad), incrementa el número de vehículos policiales o de guerra en la ciudad. Si se instancia por primera vez un vehículo policial a causa de una llamada de peatón se instanciará también un mensaje *PopUps* (apartado 4.9.5.7.5) que avise de ello y se reproducirá un *AudioClip*.

En cualquier caso el resultado es bastante bueno y ciertamente es bastante complicado atropellar un peatón de casualidad. Véase la Ilustración 4.49.

Ilustración 4.49: Peatón esquivando al jugador

4.4.2.2 Comportamiento individual

Localizado en la clase *NPC_Individual*, este comportamiento es para los peatones comunes en la ciudad y tiene un total de 4 actividades además de las que hereda. Su actividad cambia cada varios segundos de forma periódica eligiendo de forma aleatoria pero ponderada otra actividad entre caminar, correr, usar el teléfono o realizar una actividad en pareja.

4.4.2.2.1 Actividad de usar teléfono

Es una actividad que no requiere de movimiento y lo único que hace es cambiar la animación aleatoriamente entre 2 posibles donde se utiliza el teléfono. No es una actividad ni mucho menos compleja de programar, pero tiene el detalle de que estos NPCs mientras utilizan el teléfono no esquivan el vehículo del jugador, simulando que no están pendientes de la calzada.

Ilustración 4.50: Actividad de teléfono 1

Ilustración 4.51: Actividad de teléfono 2

4.4.2.2.2 Actividad de correr

Tiene el mismo funcionamiento que la actividad de caminar, pero con una velocidad mayor y con una animación diferente en la que se ve a la persona correr.

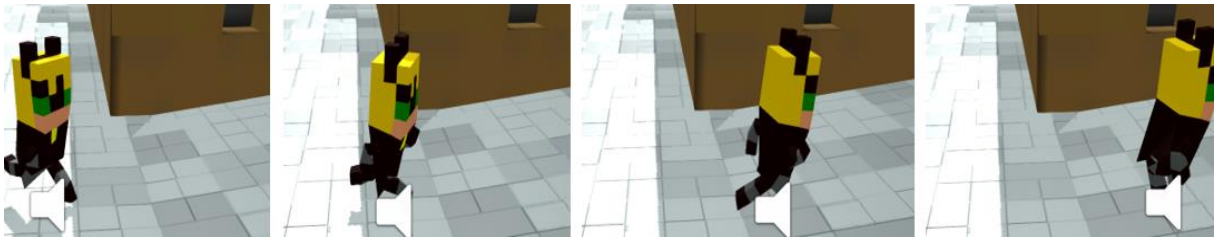

Ilustración 4.52: Actividad de correr

4.4.2.2.3 Actividad en parejas casuales: cooperación entre NPCs

Este tipo de actividad es algo especial ya que se necesitan 2 NPCs individuales para completarla. Una actividad en pareja casual se comienza cuando, de forma casual un NPC encuentra a otro NPC buscando pareja para hacer actividad, y hasta que esto no pase, el NPC realiza la acción de caminar de forma indefinida. Una vez 2 NPCs que buscan pareja se encuentran muy cerca se dirigen al punto central del vector que los une. Cuando están a la distancia correcta llevan a cabo la actividad, y no permiten que otro peatón se una a dicha actividad mientras se lleva a cabo.

Cabe destacar que, aunque la actividad esté limitada por tiempo como en los anteriores casos, el tiempo no empieza a correr hasta que se comienza la actividad en forma conjunta con otro personaje. Además, si en mitad de una actividad, por cualquier causa los NPCs son separados, estos se ponen a caminar hacia el punto central nuevamente, y cuando están lo suficientemente cerca continúan su actividad en pareja. Si uno de esos peatones es abatido durante la actividad, ésta termina y el otro NPC se dedica a hacer otra tarea.

Ahora veremos las 2 posibles actividades en pareja que se pueden dar, actividades que se eligen aleatoriamente con una probabilidad ponderada donde hay un 75% de posibilidades de que se pongan a conversar y un 25% donde comience una pelea callejera.

3.2.4.2.2.1 Actividad de conversar

En esta actividad se pide al *AnimationControllerNPCs_Personas*, igual que en las anteriores, que cambie la animación a la animación de hablar. Es otra actividad sin desplazamiento asociado.

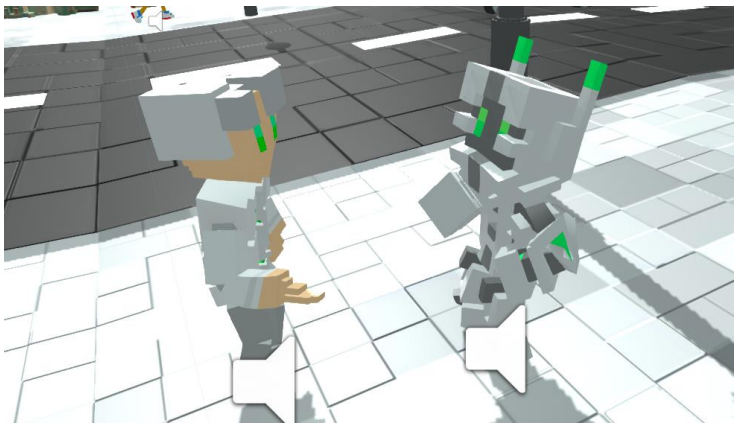

Ilustración 4.53: Actividad de conversar

3.2.4.2.2.2 Actividad de pelea callejera

Todo videojuego tiene sus puntos de macabros y en nuestro caso no será la excepción. Durante esta actividad 2 NPCs se pelearán durante varios segundos, dando como resultado un vencedor, que continuará haciendo otra actividad, y un perdedor, que caerá al suelo y perderá todos sus puntos de vida.

Ilustración 4.54: Actividad de pelear

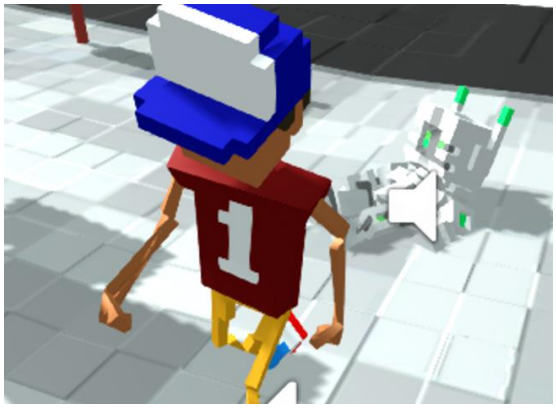

Ilustración 4.55: Resultado de pelea

A continuación, se ilustrará en la Ilustración 4.56 el funcionamiento de este comportamiento resumido en una máquina de estados para facilitar su comprensión (para la actividad de conversar sería prácticamente igual).

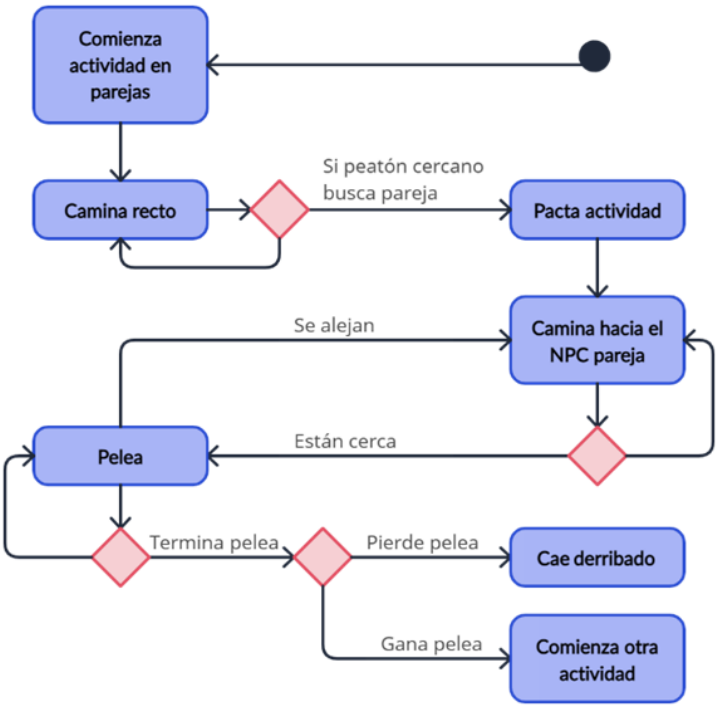

Ilustración 4.56: Máquina de estados de actividad de pelea

4.4.2.3 Comportamiento de cliente

El comportamiento del cliente comienza obteniendo un destino aleatorio del objeto *Generacion* que guarda la ciudad. Dicho punto será un vértice aleatorio del grafo de la ciudad que se encuentre dentro de un rango de distancia establecido. Posteriormente y con un destino asignado, se generan 2 objetos para ayudar al jugador a distinguir a los clientes, uno sobre la cabeza con el símbolo del dólar, el cual gira a velocidad constante sobre su eje Y, y otro bajo sus pies, que será un aura luminosa basada en un sistema de partículas obtenido de un asset de la Asset Store de Unity. Dependiendo de la distancia podemos tener:

- **Destino dentro del rango ubicaciones cercanas**: la cantidad de dinero disponible es baja, el color del aura es verde y asigna poco tiempo al taxi para hacer el transporte. Si se completa en modo arcade suma un poco de tiempo al total de la partida.
- **Destino dentro del rango de distancia medio**: la cantidad de dinero disponible es media, el color del aura es amarillo y asigna una cantidad intermedia de tiempo al taxi para hacer el transporte. Si se completa en modo arcade suma una cantidad moderada de tiempo al total de la partida.

- **Destino dentro del rango de ubicaciones lejanas**: la cantidad de dinero disponible es alta, el color del aura es rojo y asigna mucho tiempo al taxi para hacer el transporte. Si se completa en modo arcade suma mucho tiempo al total de la partida.

Tras iniciar, el cliente comienza la actividad de llamar al taxi, aunque como ya veremos, este comportamiento no es tan aleatorio como el anterior, sino que es más secuencial.

4.4.2.3.1 Actividad de llamar al taxi

En el transcurso de esta actividad lo que se hace es mostrar la animación de llamar al taxi. Es una actividad que no requiere de movimiento y donde en todo momento el NPC mira cara al taxi.

El cliente pide subir al taxi cuando esté cerca mediante la función de *proporcionarAsiento()*. En caso de haber asiento libre se le devolverá la transformación del asiento si el vehículo se mueve a baja velocidad. En el momento en el que ya tiene asociado el asiento se desactivan las colisiones, se activa el atributo de inmunidad que encontramos en *CaracteristicasPeatones* (con el fin de que no pierda vida durante el trayecto), se eliminan los objetos informativos que estaba mostrando y hace un comentario activando un *AudioClip* aleatorio perteneciente al vector de sonidos *subirTaxi* que tiene también dentro de su componente de características. Finalmente da paso a la siguiente actividad, la cual es la de subir al taxi.

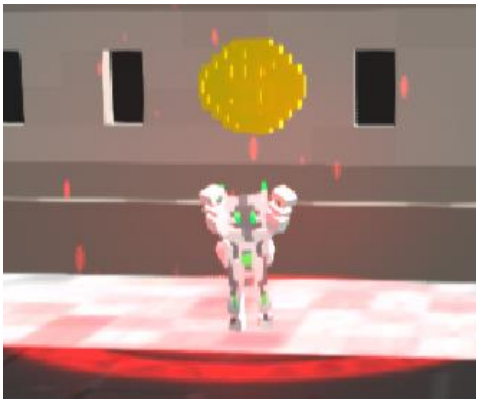

Ilustración 4.57: Cliente llamando al taxi

4.4.2.3.2 Actividad de subir al taxi

Es una actividad que primeramente realiza un movimiento desde la posición del cliente hacia la posición del asiento del pasajero de nuestro taxi con el método *MovePosition()* del componente *RigidBody*. Para llevar a cabo el movimiento el cliente pide la detención del taxi haciendo que éste se detenga automáticamente. Posteriormente cambia su transformación para mirar directamente hacia el asiento asignado a través del método *LookAt()* y realiza el salto hacia dentro del coche con la animación de salto.

Una vez se encuentra lo suficientemente cerca del asiento, el pasajero usa el componente *CaracteristicasJugador* del jugador para añadirse a sí mismo en el listado de pasajeros e indicar cuál es su destino. Seguidamente se genera un aura azul en la zona de destino y crea un cronómetro regresivo a la derecha del NPC donde se muestra el tiempo que tiene el jugador como máximo para realizar el viaje. Dicho cronometro tendrá un objeto *Temporizador*, explicado dentro del apartado 4.9.5.7.2 (Panel de tiempo restante), que se muestra inicialmente en color verde, pasa a amarillo y termina con color rojo. A diferencia del temporizador general de la partida, este temporizador se muestra en segundos y no está colocado directamente sobre la interfaz, sino que se ubica en el mundo y mira siempre a cámara de manera que siempre queda estacionado al lado del NPC, lo que da la impresión de ser un elemento de la interfaz.

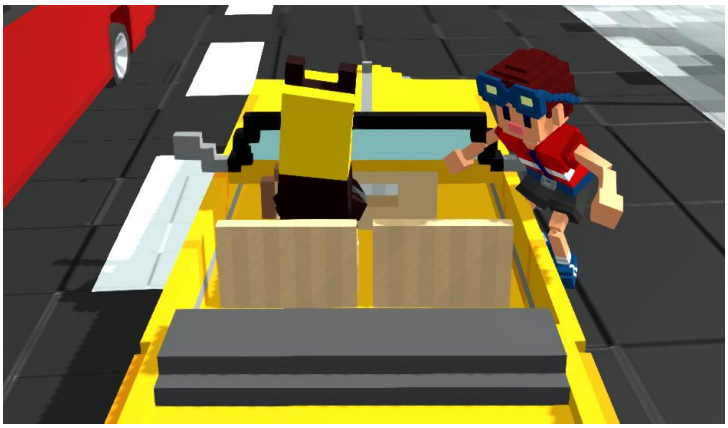

Ilustración 4.58: Cliente subiendo al taxi

Una vez realizado todo este proceso se pasa a la siguiente actividad, donde el peatón indica hacia donde tiene que ir el jugador.

4.4.2.3.3 Actividad de indicar al taxi

Mientras se da esta actividad, el NPC gira hacia la dirección donde se encuentra su destino. Durante el transcurso de esta actividad el cronómetro va bajando y cambiando su color, mientras que de forma paralela el NPC se expresa mediante emoticonos que sitúa en la posición de objetos que tiene sobre su cabeza.

Ilustración 4.59: Actividad de indicar al taxi

3.2.4.2.3.1 Emoticonos: expresividad con pocos recursos

Para ponernos en contexto, en el Crazy Taxi original los clientes hacen una serie de animaciones para conseguir expresarse y mostrar sus emociones, sin embargo, a la hora de recrear este comportamiento no se han encontrado unas animaciones que consigan el efecto buscado. Es por ello que se llevó a cabo la idea de utilizar emoticonos sobre la cabeza del personaje.

Un emoticono en nuestro proyecto se consigue con un plano al que se le cambia la textura del material por una imagen con transparencia de un emoticono. Al igual que el temporizador, le hemos indicado que siempre mire a cámara.

Las imágenes con transparencia de los emoticonos se han obtenido de un asset de la Asset Store de Unity.

4.4.2.3.4 Actividad de bajar

La actividad de indicación dura hasta que se da una de las 2 siguientes posibilidades, con las cuales da comienzo a la actividad de bajar del vehículo:

1. **El jugador agota el tiempo límite dado por el cliente y no ha llegado al destino**: esto provoca al igual que en el juego original un enfado del cliente y hace que salte del vehículo en marcha sin pagar. Además cambia el emoticono sobre su cabeza para expresar el descontento y reproduce un audio localizado en sus características donde se aprecia su enfado.

Ilustración 4.60: Cliente enfadado

2. **El jugador llega al destino dentro del límite de tiempo dado por el cliente**: lo que hace que el cliente baje del vehículo detenido y pague con el dinero que posee, el cual es mayor cuanto más lejos estuviese su destino. Sumado a esto, el cliente paga más o menos en función de lo rápido que haya sido el viaje. El dinero se suma de forma similar a los bonus, mediante el incremento del atributo de dinero del objeto *Partida* asociado al jugador. El NPC termina cambiando su emoticono por uno donde se expresa satisfacción y reproduce un audio felicitando al conductor localizado en *CaracteristicasJugador*.

Ilustración 4.61: Cliente satisfecho

En ambos casos cuando se baja el NPC del taxi desaparece el contador de tiempo que se encontraba junto al mismo y también se destruye el aura que marcaba en color azul la posición de destino.

Una vez el peatón con comportamiento de cliente se baja del vehículo activa las colisiones que estaba ignorando, desactiva su inmortalidad y cambia su comportamiento de cliente a comportamiento de peatón individual mediante la función *modificarEstrategia()* de la clase *IAnpcPeatones*, clase contexto dentro del patrón estrategia.

4.4.2.4 Comportamiento grupal

Los peatones con este comportamiento en el proyecto se crean con la generación de objetos *IA_GrupoPersonas*, que son el grupo al que pertenecen. También pueden conseguir este comportamiento tras un cambio de comportamiento, aunque igualmente el peatón tendría que ser asociado a un grupo. Este comportamiento está llevado a cabo por todos los integrantes del grupo.

Un grupo es un *GameObject* que contiene el componente *IA_GrupoPersonas* y es responsable de instanciar y ubicar tantos NPCs con comportamiento de grupo como se le indique (que son un número aleatorio dentro de un rango). Entre sus atributos más importantes los grupos cuentan con 2 que son esenciales y hay que entender para comprender el funcionamiento de los NPCs con comportamiento grupal:

- *ActividadGrupal*: es la actividad que llevan a cabo todos los NPCs del grupo. Esta no cambia una vez se instancia el grupo y es elegida aleatoriamente de forma ponderada.
- *PosicionObjetivo*: es la posición donde el grupo debe estar ubicado. Veremos que es importante porque será como el punto de quedada de los personajes y el punto hacia donde se dirigen si están en movimiento.

4.4.2.4.1 Actividad de conversación grupal

Es una actividad que en principio no requiere de movimiento y conforme se instancian los peatones y se les da el comportamiento, todos miran hacia el punto de quedada (*posicionObjetivo*) y activan la animación de conversar.

Cabe la posibilidad de que los NPCs se separen mucho, por ejemplo, por un esquivo al jugador, lo que haría que queden lejos del punto de quedada. Para evitar esto, se ha programado éstos caminen hacia el punto de quedada si se encuentran muy alejados, idea que ya se vio dentro de la Actividad en parejas casuales.

Ilustración 4.62: Actividad de conversación grupal

4.4.2.4.2 Actividad de gimnasia grupal

La actividad de gimnasia es igual a la actividad de conversar con solo 2 cambios, el primero es que ya los NPCs no miran hacia el punto de quedada, sino que miran exactamente en la dirección opuesta, y el segundo es que ahora utilizan otra animación para simular la gimnasia.

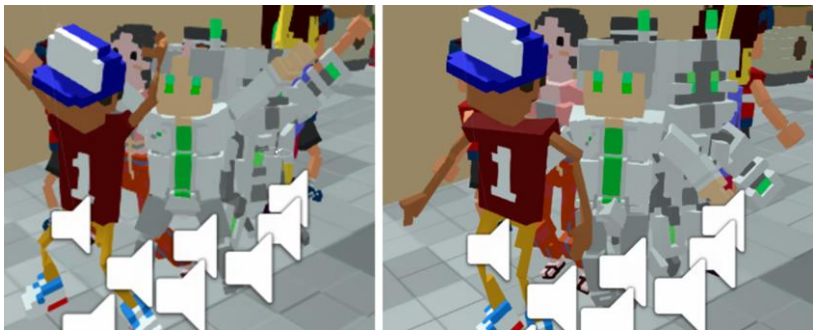

Ilustración 4.63: Actividad de gimnasia grupal

4.4.2.4.3 Actividad de marcha grupal: técnica flocking

Se basa en la idea de la técnica de flocking [52] en el ámbito de videojuegos aplicada de una forma muy básica, pues los NPCs son individuos que buscan recrear una animación colectiva dirigiéndose hacia un punto objetivo cambiante que marca el destino del grupo.

Es la actividad grupal que más ha costado implementar y su comportamiento inicia con la instanciación de todos los peatones del grupo, a los que se les asigna un punto al que dirigirse. De igual forma que se haría en la actividad de correr del comportamiento de peatones individuales, cada peatón corre hacia el punto indicado (*posicionObjetivo*). Una vez que cualquier integrante del grupo llega a ese punto, para todo el grupo cambia el punto *posicionObjetivo* y desencadena que ahora todos se dirijan hacia el nuevo punto.

Un detalle importante es que un NPC grupal puede quedarse atrás respecto al resto, y es por ello que se ha implementado que cuando esto pase, el NPC aumente temporalmente su velocidad para alcanzar al grupo.

Ilustración 4.64: Actividad de marcha grupal

4.4.2.5 Comportamiento de policía

Los peatones con comportamiento de policía son instanciados por coches de policía, pero al igual que pasa con los peatones grupales, pueden ser NPCs que han cambiado su comportamiento y a los que se les ha asignado un coche de policía. Su meta es destruir el objetivo policial marcado por el vehículo policial. Para ello habrá 2 actividades de ataque, y en función del arma que éstos tomen (de las que el vehículo les proporciona) realizarán una u otra. Las armas se ven en detalle en el apartado 4.6.2 (Armas).

Como se comenta en el apartado de IA para el comportamiento de los vehículos policiales, es el propio vehículo policial el que indica al policía cual es la actividad que debe realizar, por lo que en caso de que un vehículo de policía quede destruido también se autodestruirán todos los integrantes que lo utilizaban.

4.4.2.5.1 Actividad de persecución y ataque

En este caso, el personaje corre hacia el objetivo policial (similar a como se ha visto antes) y una vez se encuentra a la distancia idónea, se activa la animación del movimiento del ataque.

Ilustración 4.65: Actividad de ataque policial

3.2.4.2.5.1 Ataque cuerpo a cuerpo

Esta modalidad se da cuando al policía se le ha asignado el arma de bate en su posición de mano derecha. Aprovecha que la posición del objeto de la mano cuelga de la jerarquía de los huesos generados por Mixamo para situar el bate en dicho lugar

y que éste se desplace acorde a la animación, produciendo daño al objetivo policial mediante su colisión.

3.2.4.2.5.2 Ataque con armas a distancia

Esta otra modalidad se da cuando al policía se le ha asignado el arma a distancia en la posición de la mano derecha. Aquí se aplica la animación de disparar y se hace uso de la función *disparar()* del componente *Arma* que contiene el arma a distancia que se está utilizando, lo que hace generar e impulsar proyectiles indicados como no explosivos que deterioran el objeto que tocan. Se ha asociado un pequeño ángulo de error a la hora de disparar para que sea menos predecible el juego. Concretamente al policía se le pueden asignar las 2 armas a distancia de la Ilustración 4.66.

Ilustración 4.66: Armas a distancia

3.2.4.2.5.3 Actividad volver al vehículo policial

Esta actividad se da cuando el vehículo policial detecta muy lejos al objetivo de persecución. Una vez dada la orden, todos los policías integrantes del vehículo policial vuelven corriendo hasta colisionar con el vehículo policial que les pertenece, momento donde éstos son destruidos simulando que suben al coche. Una vez están todos dentro, se continúa con la persecución en coche.

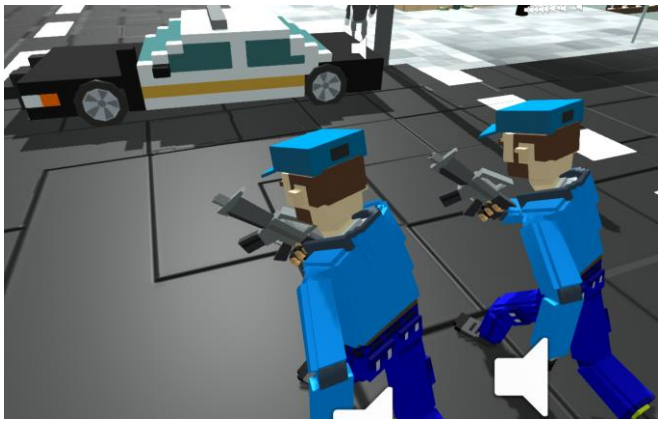

Ilustración 4.67: Actividad de volver al vehículo policial

4.4.2.6 Comportamiento de zombi

El zombi tiene como objetivo expandir la plaga, por lo que su comportamiento se basa en buscar gente para contagiarla.

Si nos centramos en su funcionamiento, lo primero que se hace al activarse el comportamiento es modificar el material de la piel del peatón a un material con una paleta de colores verdes. Posteriormente el zombi comienza a caminar sobre la acera, comportamiento que hereda y que personaliza con su una animación propia, y continúa haciéndolo hasta que se acerca a un peatón no contagiado y no abatido. En ese instante, el zombi lo marca como presa y comienza a correr hacia él de manera similar a como se hacía en anteriores comportamientos, activando la animación de correr para zombis. Su siguiente acción será la de realizar el ataque.

Ilustración 4.68: Zombi corriendo hacia presa

Un detalle de este comportamiento es que los NPC que lo poseen nunca esquivarán el vehículo del jugador, porque se busca recrear la idea de que un zombi ha perdido reflejos y no sería capaz de hacerlo.

4.4.2.6.1 Actividad de ataque zombi: contagio de comportamiento

La única actividad especial nueva incluida efectivamente es la del ataque zombi. Ésta se da cuando el zombi colisiona con el NPC al que persigue, momento en el que se activa una animación de ataque y se envía la orden de cambiar el comportamiento del personaje colisionado a comportamiento de zombi. Una vez convertido en zombi se desmarca al NPC como presa y se busca otra, repitiendo el funcionamiento explicado nuevamente.

Ilustración 4.69: Contagio por un ataque zombi

4.4.3 Pruebas

A continuación, se van a mostrar los resultados de los test realizados para los peatones NPC.

T-01 Test de animación	
Objetivo	Todos los NPCs tienen que poder cambiar de animación a cualquiera de las existentes en función de la variable estado
Resultado	Todos los NPCs cambian correctamente de animación gracias a la variable de estado y múltiples condicionales
Observaciones	Cada NPC tiene sus propias animaciones, lo que les proporciona una personalidad propia Ha habido que corregir errores de asignación de animaciones y bastantes errores de condicionales de la máquina de estados que dirige las animaciones

Tabla 4.12: Test de animación

T-02 Test de utilización de indicadores y objetos	
Objetivo	Todo NPC debe ser capaz de poder colocar un objeto en su posición de cabeza, mano y pies independientemente del funcionamiento de los componentes que se le asocien
Resultado	Todos los NPCs son capaces de usar las posiciones reservadas para objetos independientemente de su comportamiento, con un límite de 1 objeto por posición
Observaciones	Si se intenta incluir 2 objetos en una misma posición se descarta el primer objeto para colocar el segundo

Tabla 4.13: Test de utilización de indicadores y objetos

T-03 Test de cambio de comportamiento	
Objetivo	Todo NPC tiene que poder cambiar en cualquier momento su tipo de comportamiento por otro cualquiera y estar adaptado para ello
Resultado	Todos los NPCs pueden cambiar sin problema a cualquier otro comportamiento en tiempo de ejecución y en cualquier momento
Observaciones	En el juego no se han utilizado todas las combinaciones de cambio de comportamiento, pero sí que están funcionales

Tabla 4.14: Test de cambio de comportamiento

T-04 Test de comportamiento compartido	
Objetivo	Todo NPC debe ser capaz de utilizar las funciones de comportamiento compartido
Resultado	Todos los NPCs son capaces de usar las funciones de comportamiento compartido de forma correcta
Observaciones	

Tabla 4.15: Test de comportamiento compartido

T-05 Test de actividad de esquivar	
Objetivo	Los NPCs que tengan capacidad de esquivar según el funcionamiento de su comportamiento, deben ser capaces de esquivar por el camino más corto
Resultado	Los NPCs son capaces de esquivar por el camino más corto y evitar el atropello en la mayoría de las ocasiones
Observaciones	Los NPCs solo esquivan el vehículo del jugador para evitar sobrecargar al sistema, lo que produce que puedan ser atropellados por los vehículos de policía y tanques con facilidad

Tabla 4.16: Test de actividad de esquivar

T-06 Test de comportamiento individual	
Objetivo	Los NPCs deben realizar todas las actividades del comportamiento individual
Resultado	Los NPCs son capaces de realizar todas las actividades del comportamiento individual, gestionan correctamente el tiempo de cada actividad y permiten las actividades en parejas
Observaciones	

Tabla 4.17: Test de comportamiento individual

T-07 Test de actividades en parejas	
Objetivo	Los NPCs deben organizar actividades en parejas con otros NPCs de la calle de manera espontánea y coordinada
Resultado	Los NPCs son capaces de realizar toda la gestión de la actividad en parejas y además si estos son separados caminan a un punto intermedio para continuar la actividad
Observaciones	Se tuvieron que corregir errores para evitar que se sumaran más de 2 NPCs a la actividad en parejas

Tabla 4.18: Test de actividades en parejas

T-08 Test de comportamiento de clientes	
Objetivo	Los clientes son capaces de subir al vehículo si está vacío y a baja velocidad, indicar su destino y bajarse en el cuando éste se alcanza
Resultado	Estos NPCs hacen correctamente todas sus actividades
Observaciones	A veces al bajar del vehículo el sistema de físicas hace que el NPC quede atrapado debajo del suelo, es por ello que se incluyó un parche con el que se suele evitar el fallo

Tabla 4.19: Test de comportamiento de clientes

T-09 Test de comportamiento de grupos	
Objetivo	Los NPCs pueden entrar y salir de un grupo y participar correctamente en las actividades que estos llevan
Resultado	Los NPCs entran y salen correctamente del grupo y participan en las actividades del mismo a menos que estén lejos del grupo, caso en el que se pondrían a caminar para acercarse (o correr más rápido si la actividad es de marcha)
Observaciones	En la actividad de marcha a veces la orientación de algunos NPCs cambia muy rápidamente de manera poco realista

Tabla 4.20: Test de comportamiento de grupos

T-10 Test de comportamiento de policía	
Objetivo	Los policías persiguen al jugador y utilizan sus armas de manera correcta. Deben también ser capaces de volver al vehículo del que provienen.
Resultado	Los policías realizan sus actividades correctamente
Observaciones	

Tabla 4.21: Test de comportamiento de policía

T-11 Test de comportamiento de zombis	
Objetivo	Los zombis deben ser capaces de hacer sus actividades básicas y de contagiar a otros NPCs con comportamiento diferente al de zombi
Resultado	Los zombis contagian correctamente a los demás NPCs
Observaciones	Se ha tenido que corregir un error para evitar que los zombis se persiguieran entre ellos

Tabla 4.22: Test de comportamiento de zombis

4.5 Vehículos NPC

Los vehículos autocontrolados tienen un papel fundamental dentro del juego y dentro del desarrollo. Podemos destacar 3 tipos de vehículos:

- **Vehículos comunes**: funcionan con el script *IA_NPC_Vehiculos*
- **Vehículos policiales**: funcionan con el script *IA_VehiculoPoliciasNPC*
- **Vehículos de guerra**: funcionan con el script *IA_TanqueNPC*

La IA creada para los vehículos se organiza mediante varios scripts de forma jerarquizada usando herencia, lo que ha permitido ahorrar mucho tiempo de programación. Los scripts llevan a cabo la herencia mediante la estructuración dada por la Ilustración 4.70, la cual actualiza el fragmento reservado para los vehículos del UML inicial y especifica por completo la funcionalidad.

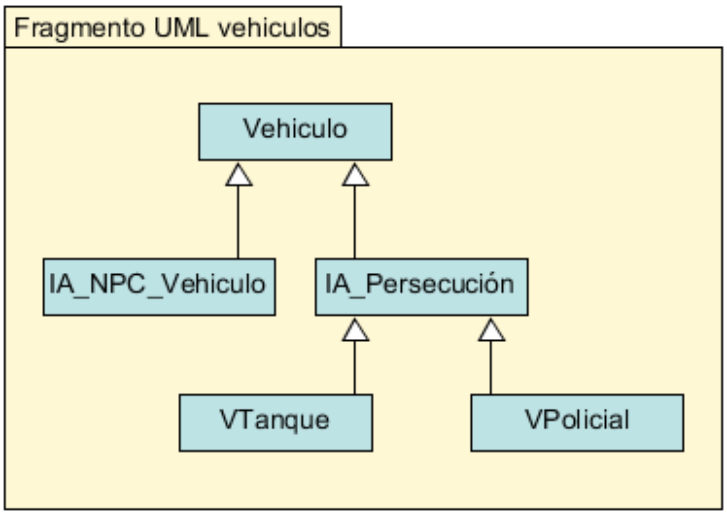

Ilustración 4.70: Especificación de UML para vehículos NPC

4.5.1 Sistema de movimiento: alternativas planteadas

Para desplazar los vehículos existen varias opciones, opciones que se valoraron y probaron, llegando a la conclusión de que no todas son válidas e igual de fáciles de hacer:

- **Mediante translaciones** directamente sobre la matriz de transformación del vehículo. Es la idea más simple y en la que primero se pensó, pero tuvo que ser descartada porque los vehículos son influenciados por las físicas y sus colisiones, algo que repercute generando movimientos extraños e imprevisibles en el desplazamiento de los vehículos.
- **Mediante físicas** con las funciones *addForce()*. Esta opción es viable pero complicada de llevar a cabo. Requiere demasiados ajustes como para plantearlo como una posibilidad lógica en el tiempo disponible para el proyecto.
- **Mediante el componente WheelCollider**, es la opción que se terminó utilizando, y se trata de un colisionador específico de Unity que ya utilizamos en el apartado que habla del Jugador. Este colisionador sirve para crear ruedas y desplazar vehículos, proporcionando detalles tales como la simulación de la amortiguación y la fricción con el suelo, detalles que serían muy complicados de incluir en las anteriores propuestas.

Pese a que el *WheelCollider* sea la mejor opción, este componente es un elemento tan específico que ofrece Unity que muchos no conocen y que, para descubrirlo y aprender a usarlo, se requirió de un periodo investigación, experimentación y adquisición de madurez por parte del autor del proyecto.

4.5.2 Comportamiento compartido: herencia

Este script contiene una serie de métodos que comparten todos los vehículos mediante herencia, tales como:

- **Método que se encarga de hacer girar las ruedas sobre su eje**: hace girar la rueda para simular que está moviendo. Si el vehículo está realizando una curva además del giro sobre su eje, la rueda se rota en Y hacia el lado al que se va el vehículo.

- **Método de acelerar, girar y de frenar**: que hacen uso de los torques de los colisionadores de las ruedas (con los que damos tracción trasera) y del vector de velocidad del *RigidBody* asociado para ajustes la velocidad y dirección del vehículo.
- **Método de detección de choque frontal**: funciona haciendo uso de los algoritmos de rayo que nos proporciona Unity, dentro de su clase *Physics* mediante el método *Raycast* [56], informando si en la dirección de movimiento del vehículo hay un obstáculo.

  Physics.Raycast envía un rayo dada una posición y una dirección, y devuelve un objeto de colisión (*RaycastHit* [57]) asociado al objeto con el que éste ha colisionado.

  Concretamente para este método se utilizan 2 rayos, centrados, separados una pequeña distancia y apuntados hacia la dirección a la que el vehículo se dirige. Es un método sumamente importante porque nos permitirá hacer un sistema de circulación simple pero eficaz.
- **Método de selección de destino aleatorio**: devuelve un nodo de destino aleatorio de entre los que conectan mediante una arista con el último nodo atravesado por el vehículo. Para ello recordemos que los nodos del grafo de la ciudad representan cruces, intersecciones o curvas, mientras que las aristas son calles rectas que los unen.

4.5.3 Vehículos comunes

Los vehículos a los que llamaremos comunes son aquellos que se mueven por el mapa de forma aleatoria y sin objetivos, dando simplemente un ambiente más vivo.

Igual que se hizo para el vehículo del jugador, estos vehículos y los siguientes se descargaron de la página de Sketchfab como modelos no preparados para su uso en Unity, por lo que todos se han tenido que modificar uno a uno desde Blender para eliminar las ruedas que traían acopladas, con la idea de colocar en dicho sitio nuestras propias ruedas, con capacidad de girar y poseedoras del componente colisionador.

Ilustración 4.71: Vehículos comunes

4.5.3.1 Estructura del vehículo y componentes

La estructura si la analizamos es bastante parecida a la que tenía el vehículo del *Jugador* (apartado 4.1), pues seguimos teniendo dentro de la jerarquía unas ruedas con sus respectivas mallas y colisionadores *WheelColliders*, la malla de la carrocería y un sistema de partículas para mostrar humo cuando hay avería. Aunque por el contrario ahora vemos que directamente se ha situado al lado del volante un conductor por defecto en lugar de encontramos una posición para el conductor elegido por jugador.

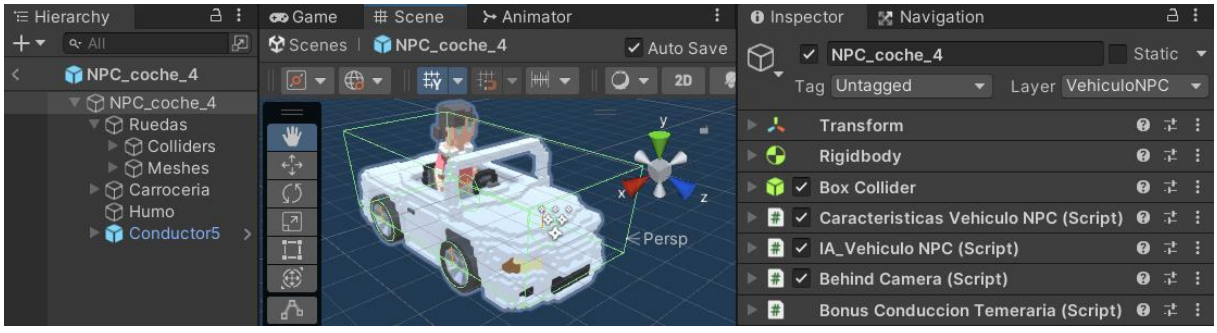

Ilustración 4.72: Estructura vehículo NPC

En cuanto a los componentes sí que vemos que hay más variación. Sin entrar en demasiado detalle tendremos nuevamente los componentes diseñados por Unity *RigidBody* y *BoxCollider* para que el vehículo reaccione a físicas y colisiones.

Como puede verse en Ilustración 4.72, además de los componentes de Unity hay varios propios, tales como el script *CaracteristicasVehiculos*, que dota de vida a nuestros vehículos y guarda toda la información acerca de la aceleración, inactividad, velocidad (que será limitada), peso (que será muy bajo), etc. Otro componente propio desarrollado en su propio apartado es el de *BonusConduccionTemeraria*, el cual hace que el jugador reciba una cantidad de dinero si choca a un vehículo y hay un cliente subido en el taxi. Además, hay 2 nuevos componentes:

- ***BehindCamera***: es un script que se consiguió de Internet y se modificó para adaptarlo. Tiene únicamente una función pública, *lookingCameraTransform()*, que indica si la cámara está viendo o no al vehículo que contiene este script.
- ***IA_VehiculoNPC***: es un script propio y hereda directamente de VehiculoNPC. Proporciona la inteligencia básica para conducir en la ciudad y procedemos a explicarlo en el siguiente subapartado.

4.5.3.2 IA para el comportamiento de los vehículos comunes

Para empezar, es muy importante dejar claro que hacer un sistema de conducción es algo complejo, que daría perfectamente el suficiente trabajo como para otro proyecto a parte. Por tanto y como es de esperar, no se ha recreado por completo un sistema de circulación atendiendo a todas las señales y normas de tráfico, pero sí que se ha hecho el esfuerzo por conseguir unos mínimos.

En este subapartado vamos a comentar el comportamiento de los vehículos conseguido gracias al script *IA_VehiculoNPC* de forma directa y práctica sin entrar en detalles de implementación.

Todo vehículo al ser instanciado e iniciado se sitúa al lado derecho de la vía y se orienta en la dirección de la misma. Una vez hecho, el vehículo se dirige en la dirección que tiene la calle hasta que se encuentra una intersección o una curva (vértices del grafo de ciudad). En este momento el vehículo accede al grafo y elige una arista de entre las posibles de forma aleatoria, que será una que contenga al vértice en el que se ubica el vehículo y no sea la misma arista de la que proviene (no tiene sentido dar una vuelta de 180º). Una vez elegida la arista, se obtiene la posición

del vértice que se encuentra al otro lado y se marca como objetivo para ir, llevando a cabo un giro si es necesario.

Este funcionamiento está bastante bien para un solo coche, sin embargo, llevado a la práctica hay varios problemas:

- Hay una circulación completamente anárquica donde los vehículos chocan entre ellos en las intersecciones, dando un aspecto poco realista y que provoca atascos y vuelcos de vehículos.
- Si un vehículo queda detenido por alguna razón, los de detrás no frenan y lo arrastran.
- Los vehículos no dejan distancia de seguridad, algo que no es realista y provoca una peor jugabilidad.

Para arreglar estos problemas de circulación se utiliza la el método descrito para la detección de choques mediante rayos. Gracias a este método nuestro vehículo baja la velocidad si detecta delante a otro objeto, lo que hace que en las intersecciones siempre pase el primero que llega, cuando un vehículo queda detenido los de detrás se detienen también y además hace que todos guarden una distancia de seguridad.

En la Ilustración 4.73 vemos como se forma un atasco, guardando distancias de seguridad y frenando debido al obstáculo en la vía, que es el coche del jugador.

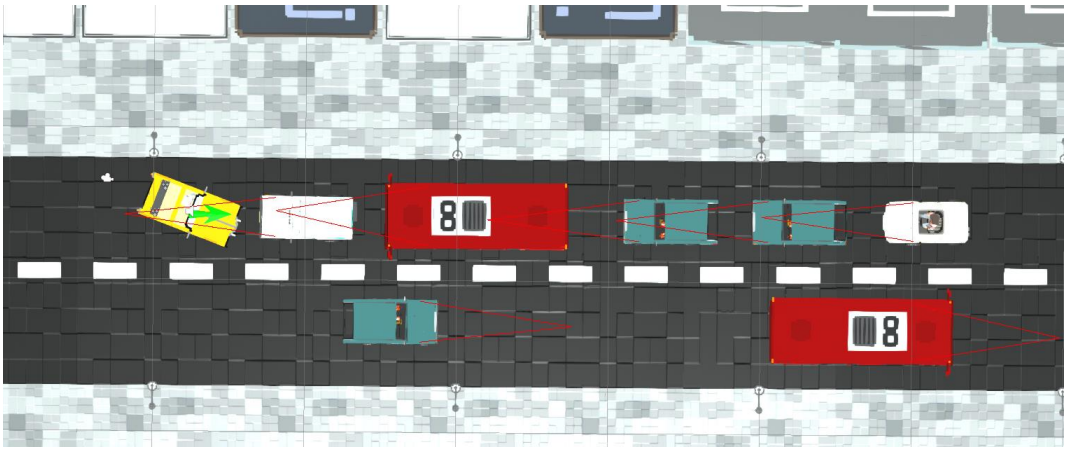

Ilustración 4.73: Atasco

En esta otra Ilustración 4.74 puede apreciarse una secuencia de 9 imágenes tomadas al vehículo de color naranja, el cual espera en la intersección para que el camión pase, porque de no ser así chocaría.

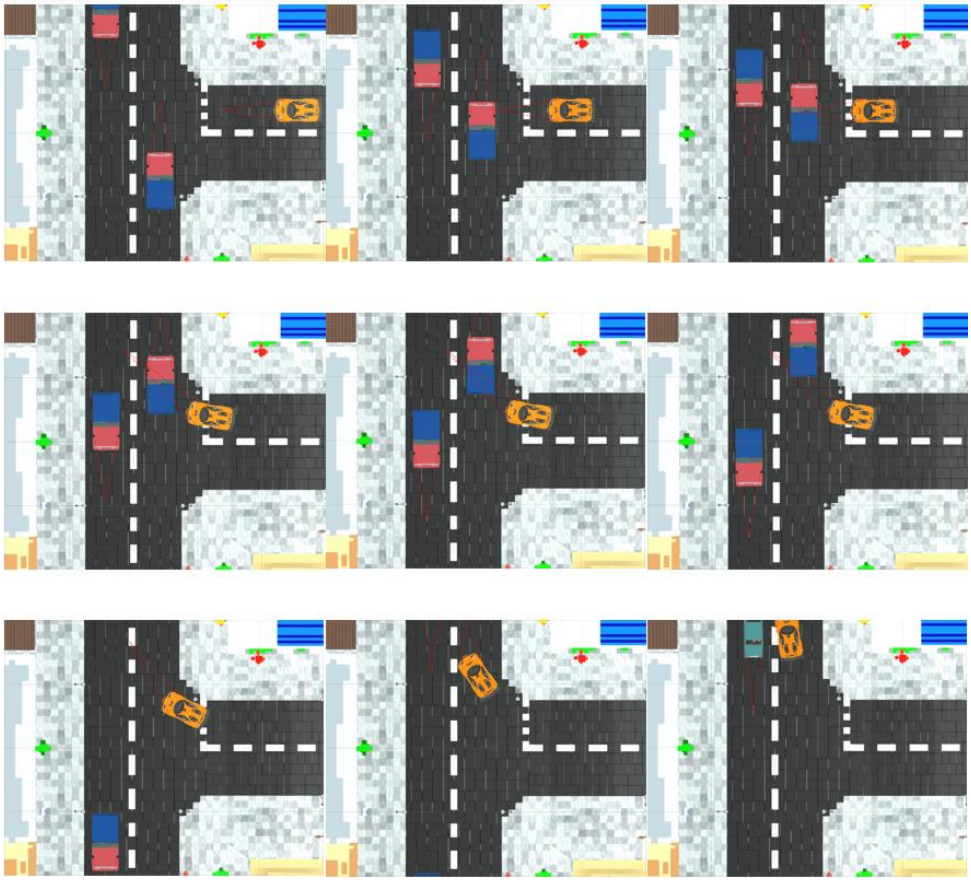

Ilustración 4.74: Vehículo cede el paso

Sin embargo, hay aun un problema, y es que, si un vehículo es desplazado, ya sea por un golpe del jugador o por cualquier otra casuística, el vehículo se dirigirá en línea recta a la posición del vértice objetivo marcado. Esto puede producir que el vehículo circule durante mucho tiempo sobre la acera o en dirección contraria. Para arreglar esto lo que haremos es que ya no se indicará el vértice objetivo al vehículo, sino que en cada frame indicaremos en su lugar un punto intermedio que se encuentre dentro del carril en una posición entre el vehículo y el siguiente vértice. El resultado es bastante bueno y podemos verlo nuevamente con el vehículo naranja en la Ilustración 4.75, el cual se ha sacado de la vía para ver cómo se reincorpora rápidamente.

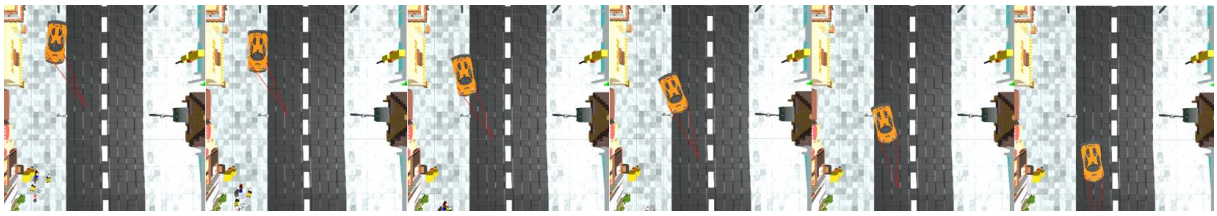

Ilustración 4.75: Reincorporación a la vía

4.5.4 Vehículos policiales: entidades persecutoras I

Todo aquel que haya jugado Crazy Taxi alguna vez sabe que el juego no tiene policía y que esta mecánica junto con la de la inclusión de tanques viene inspirada de la saga de GTA. La idea es hacer que aparezca policía persiguiendo al jugador cuando éste intente atropellar a un cierto número de personas.

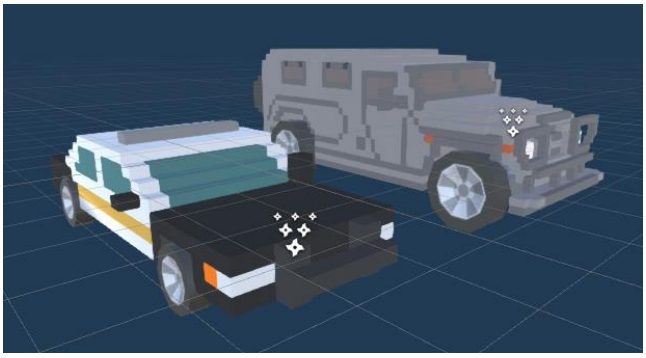

Ilustración 4.76: Vehículos policiales

En lo que a la estructura del vehículo y componentes se refiere, apenas hay novedad frente a los vehículos comunes, pues contienen la misma jerarquía y casi los mismos componentes, aunque estos vehículos son mucho más rápidos y pesados que los vehículos comunes. Al igual que el *Jugador*, los policías también tienen un *GPS* incorporado y es usado dentro del script que le da el comportamiento al vehículo, script que ya no será *IA_VehiculoNPC* sino que será *IA_VehiculoPoliciasNPC*.

4.5.4.1 IA para el comportamiento de los vehículos policiales

IA_VehiculoPoliciasNPC hereda de la clase *IA_PersecucionNPC* que a su vez hereda de *VehiculoNPC*.

IA_PersecucionNPC añade un sistema de movimiento que ya no es aleatorio como se daba en los vehículos comunes, sino que lleva asociado un objetivo de

persecución y la decisión de girar en una dirección u otra llegados a una intersección o cruce se da en función de la ruta devuelta por el GPS, al que se le indica que el destino es un objetivo policial (por ejemplo la posición del vehículo del jugador). Así, los vehículos policiales persiguen por el camino más corto al taxi del jugador hasta que se encuentran cerca del mismo, momento donde deja de preguntar al GPS cual es la ruta y el vehículo comienza a moverse directamente hacia la posición del jugador. Cabe la posibilidad de que la ruta indique la dirección contraria a la que el vehículo se está dirigiendo, ya sea por ejemplo porque el jugador se ha movido y la ruta cambió, en tal caso, como un vehículo no puede girar 180º en mitad de una calle, se sigue un movimiento aleatorio al igual que hacen los vehículos comunes hasta que la ruta pueda retomarse sin dar marcha atrás.

Una vez el vehículo se encuentra lo suficientemente cerca del objetivo policial (en nuestro caso para los vehículos policiales siempre será el jugador) y ambos están detenidos, se instancian múltiples policías, desarrollados en profundidad en el apartado 4.4.2.5 (Comportamiento de policía). En específico se bajan tantos policías como haya dentro de cada vehículo policial. Tras instanciar a los policías se les da un arma de las que el vehículo contiene.

Ilustración 4.77: Policías bajando de un vehículo policial

Siguiendo las órdenes del vehículo policial, los policías persiguen y atacan al objetivo marcado o vuelven al vehículo, algo que el vehículo policial decide en base a la distancia que lo separa del jugador.

El vehículo policial queda detenido tras instanciar a los policías, pero puede volver a arrancarse cuando todos los policías vivos pertenecientes al vehículo hayan subido al mismo, exigiendo un mínimo de un policía para conducir.

El funcionamiento es bastante interesante, y lo explicado puede resumirse en la siguiente máquina de estados, Ilustración 4.78.

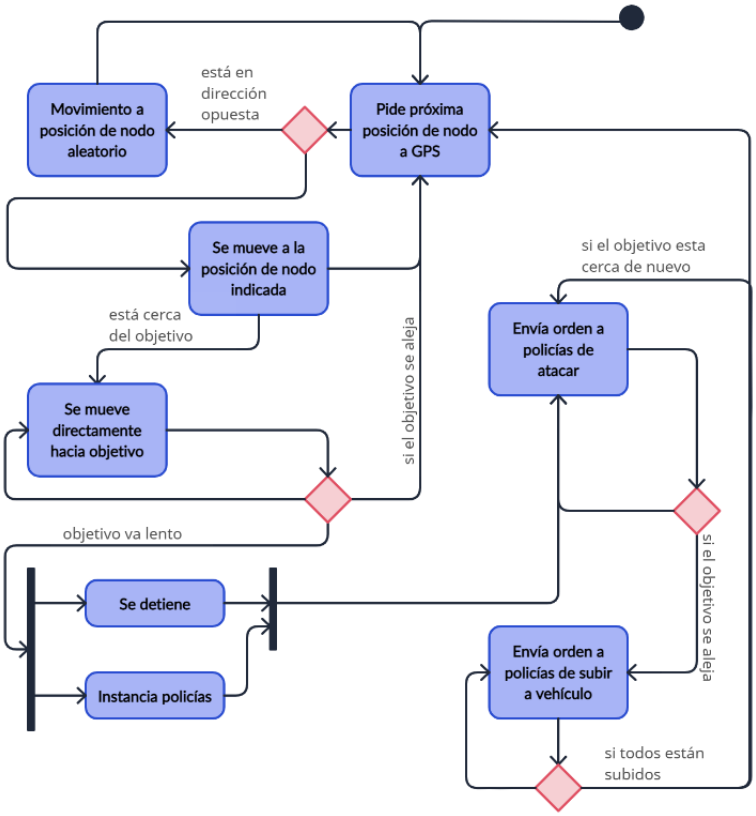

Ilustración 4.78: Máquina de estados de vehículo policial

4.5.5 Vehículos de guerra: entidades persecutoras II

Pese a que hablamos de una estructura prácticamente igual a la que encontramos para los vehículos policiales en lo que a componentes se refiere, diferenciado únicamente por su IA, la estructura tiene un principal añadido que no tienen ningún otro tipo de vehículos.

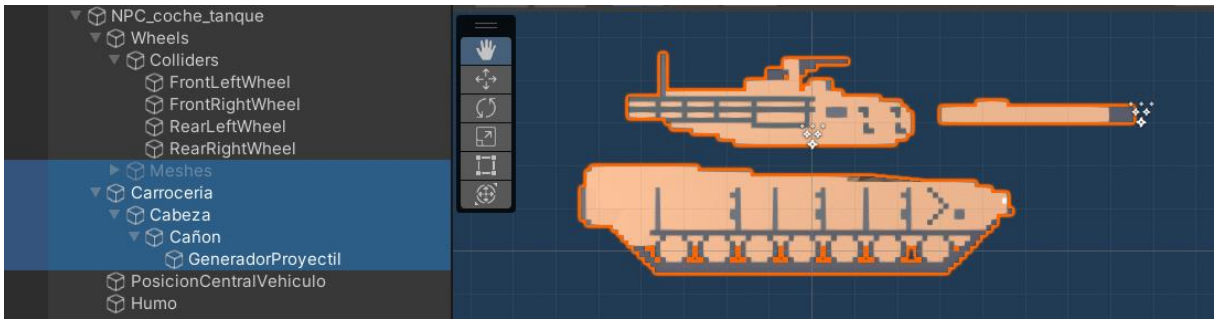

Ilustración 4.79: Estructura de un tanque

En la Ilustración 4.79 puede verse como efectivamente la estructura del vehículo cambia un poco. Por un lado, ahora el tanque tiene una malla de carrocería que se ha dividido en diferentes partes con Blender, diferenciando cuerpo, cabeza y cañón. Estas partes se moverán de forma encadenada gracias a la jerarquía dada. De esta forma, el tanque puede rotar en el eje local Y la cabeza hacia donde desee independientemente de la dirección en la que se dirige todo el vehículo, e igualmente con el cañón, que se mueve en sintonía con la cabeza y que también puede moverse hacia arriba y abajo dentro de un rango establecido. El cañón además tiene incorporado un objeto que es poseedor de un objeto *Arma* (apartado 4.6.2), es decir, puede generar y disparar proyectiles, que además han sido marcados como explosivos. Sumado a eso vemos que ya no mostramos las mallas de las ruedas, ello es porque, aunque sigamos usando la misma metodología para controlar el tanque mediante los colisionadores *WheelColliders*, ahora escondemos la malla de la rueda que realmente usa para que no se vea, pareciendo que el desplazamiento se realiza en verdad mediante las ruedas de oruga del modelo.

4.5.5.1 IA para el comportamiento de los vehículos de guerra

Como se acaba de comentar, ahora usamos para el comportamiento el script *IA_TanqueNPC*, clase que hereda de la clase *IA_PersecucionNPC* de la misma manera que *IA_VehiculoPoliciasNPC*, y que por ende, tampoco tendrá un movimiento aleatorio, sino que perseguirá un objetivo marcado, que irá cambiando conforme éste sea destruido.

Este script se ocupa de añadir un comportamiento de ataque al tanque, sin necesidad de generar policías a su alrededor, gracias a la utilización del objeto con el script *ArmaDistancia* (apartado 4.6.2.1) incorporado en la punta del cañón.

Análogamente a como hacen los vehículos policiales, el tanque persigue y se detiene frente al objetivo cuando lo alcanza, con la diferencia de que puede disparar al jugador estando o no en movimiento si está a una distancia correcta. De esta forma, para realizar un disparo el tanque primero debe apuntar realizando giros de cabeza y cañón de forma inteligente por el camino más corto, y una vez queda prácticamente alineada la dirección de apuntado con el objetivo se dispara.

Es conveniente mencionar que el arma del tanque, al igual que pasa con los policías tiene un error asignado de hasta un par de grados, con objetivo no hacer todos los disparos perfectos.

4.5.6 Pruebas

A continuación, se van a mostrar los resultados de los test realizados para los vehículos NPCs.

T-01 Test de conducción	
Objetivo	Los vehículos siguen el carril derecho sin salirse y son capaces de acelerar, frenar y tomar curvas
Resultado	Los vehículos conducen correctamente por su carril, acelerando frenando y girando siempre que lo necesiten
Observaciones	

Tabla 4.23: Test de conducción

T-02 Test espacio entre vehículos	
Objetivo	Los vehículos normales deben de dejar un espacio mínimo con el vehículo que se encuentra delante
Resultado	Los vehículos frenan en caso de encontrar delante otro vehículo muy cerca dejando un correcto espacio en medio
Observaciones	Los vehículos persecutores no siguen esta regla (se programó así adrede)

Tabla 4.24: Test espacio entre vehículos

T-03 Test de incorporación a la vía	
Objetivo	Un vehículo debe ser capaz de volver a la vía si éste es sacado de la misma
Resultado	Los vehículos vuelven correctamente a la vía

Observaciones	Si encuentran algún elemento entre el vehículo y la vía no llegan a reincorporarse porque frenan para evitar chocar

Tabla 4.25: Test de incorporación a la vía

T-04 Test de persecución	
Objetivo	Los vehículos persecutores deben ser capaces de usar el GPS para perseguir por el camino más corto al objetivo
Resultado	Los vehículos persecutores persiguen correctamente al objetivo, deteniéndose junto a él si éste tiene una baja velocidad y se encuentran muy cerca
Observaciones	Si el objetivo es el jugador y éste se acerca a un vehículo persecutor desde la parte trasera del mismo, le cortará el paso

Tabla 4.26: Test de persecución

T-05 Test de generación de policías	
Objetivo	Los vehículos policiales deben ser capaces de generar policías armados
Resultado	Los vehículos policiales generan policías y les asignan un arma cuando están detenidos y cerca del vehículo del jugador
Observaciones	

Tabla 4.27: Test de generación de policías

T-06 Test de apuntado de tanque	
Objetivo	Los vehículos de guerra deben de ser capaces de orientar el cañón hacia el objetivo antes de disparar
Resultado	Los tanques son capaces de mover la cabeza y el cañón del tanque para alinear el disparo con el objetivo. Posteriormente dispara si se dan las condiciones
Observaciones	

Tabla 4.28: Test de apuntado de tanque

4.6 Elementos básicos

Aquí desarrollaremos todos aquellos scripts que son utilizados por gran número de objetos y que por tanto aparecen repartidos por los diferentes prefabs que se encuentran en el proyecto.

4.6.1 Características

Para gestionar las cualidades propias de todos los peatones y vehículos NPCs del juego se han creado una serie de scripts que están organizados jerárquicamente. Primeramente, se ha creado una clase *Caracteristicas* que tiene los atributos:

- *Vida*: se trata de un valor flotante que marca los puntos máximos de vida de la entidad.
- *Inmune*: es un booleano que ignora la perdida de vida cuando se activa.
- *ID_Nombre*: es un identificador utilizado para buscar el nombre del objeto y la traducción correspondiente.

Además, esta clase va a tener los métodos *get()* y *set()* de los atributos, se han creado 2 funciones muy importantes:

- *RestarVidaPorFuerza*: resta la vida en función de la fuerza que se le aplica.
- *RestarVidaPorPuntos*: resta los puntos de vida indicados.

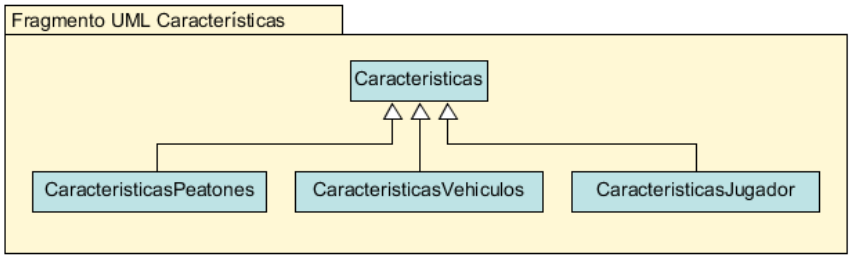

Ilustración 4.80: Espeficiación de UML para características

Actualizamos el trozo referente a las características dentro del UML inicial mediante la nueva organización dada en la Ilustración 4.80. La clase *Características* hará de clase base para las demás, observando la aparición de 3 nuevas relaciones de herencia para especificar las características: las características de los peatones, las características de los vehículos y las características del propio jugador.

4.6.1.1 CaracteristicasPeatones

Esta clase añade un puñado de atributos que particularizan a un peatón (como puede verse en Ilustración 4.81), donde:

- *FuerzaMinimaPeligro*: indica el mínimo de fuerza que se le tiene que aplicar antes de recibir daño. Es importante este parámetro para que la propia colisión con objetos de la calle u otros NPCs no le baje la vida a la entidad.
- *PosiciónObjetoMano, PosiciónObjetoMano, PosiciónObjetoMano*: aquí está registrado el *GameObject* que define la posición de la mano, el de la posición donde se muestra la información y el de la posición donde se muestra el aura del NPC. Dichas posiciones fueron estudiadas en el apartado 4.4.1.1 (Partes del humanoide).
- *SubirTaxi, BajarTaxiContento, BajarTaxiEnfadado y SonidoGrito*: son arrays de *AudioClips* con diálogos y voces diferentes personalizadas para cada peatón que son reproducidos por el *AudioSource* correspondiente cuando se requiere del sonido (por ejemplo al bajarse un peatón del vehículo).

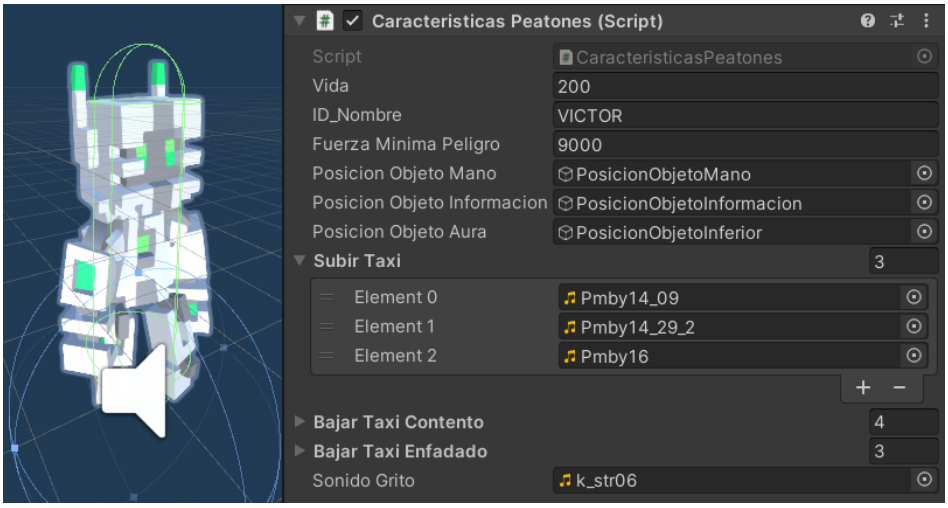

Ilustración 4.81: Parámetros de CaracteristicasPeatones

Esta clase cuenta con una serie de métodos particulares con los que se puede cambiar el objeto que el peatón lleva en la mano, en la zona informativa (sobre la cabeza) y en la zona de aura (en los pies), métodos con los que reproducir los audios del personaje y también cuenta con 2 métodos usados en el modo de zombis con los que se puede cambiar el color del material del personaje (a tonalidades más verdosas mediante el cambio de paleta de colores con un segundo material).

4.6.1.2 CaracteristicasVehiculos

Igualmente, esta clase añade nuevos atributos además de los aportados por su clase padre y es importante entenderlos para cuando veamos el comportamiento de los mismos:

- *VelocidadMaxima*: es la máxima velocidad del vehículo.
- *Peso*: es el peso del vehículo y modifica en tiempo de ejecución el atributo *mass* del componente *RigidBody*.
- *DistanciaVerCurva*: se trata de la distancia a la que un vehículo empieza a girar cuando va a tomar una curva.
- *FuerzaMotor*: es la fuerza que se le aplica al torque de los componentes *WheelCollider* de las ruedas, lo que influye directamente en su aceleración.
- *FuerzaFrenado*: es la fuerza que se aplica al torque de los componentes *WheelCollider* de las ruedas para disminuir su velocidad.
- *AnguloGiroMaximo*: es el máximo angulo de giro del vehículo.
- *Humo*: es un *GameObject* que contiene un sistema de partículas para generar humo cuando se produce una avería. Tiene la emisión del sistema de partículas siempre desactivada a menos que la vida baje del 20% del total, momento en el que se activa para indicar dicha avería.

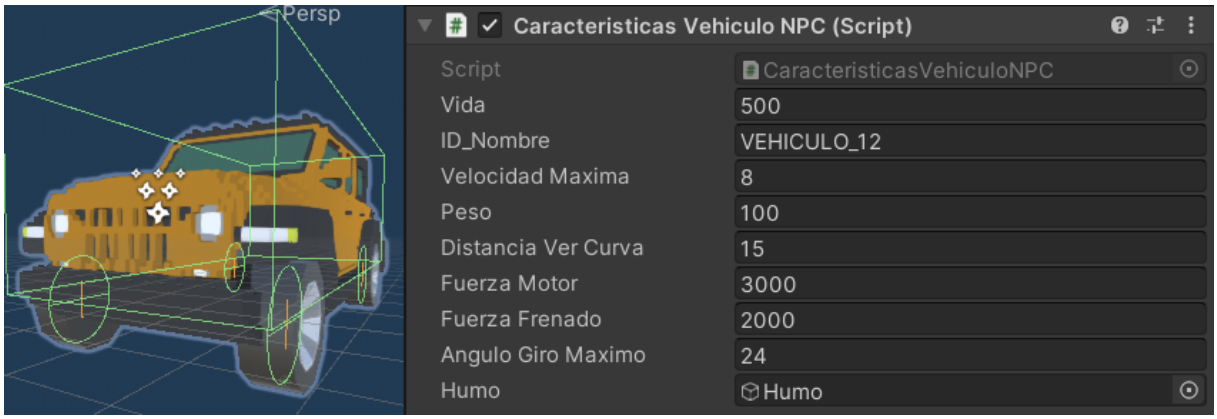

Ilustración 4.82: Parámetros de CaracteristicasVehiculo

Vale la pena destacar que se ha creado una corrutina [58] que se activa cada segundo para comprobar si el vehículo se ha desplazado, con la meta de utilizar dicha información para saber si el vehículo ha quedado inactivo por ejemplo por un choque

con el jugador que le imposibilite la maniobra para volver a la vía o por perder toda la vida. La idea es usar una función llamada *estaInactivo()* que parte de esta información para eliminar vehículos inactivos y evitar posibles atascos, algo que se verá en el apartado 4.8.1 (Gestor de vehículos NPC).

4.6.1.3 CaracteristicasJugador

Finalmente tenemos las características del jugador, que como vemos hay más atributos que deben asignarse, de los cuales vamos a destacar los más novedosos dejando a un lado los que se definen por su propio nombre, tales como la velocidad, aceleración o el peso:

- *MáximaDistanciaParaSubirse*: es la distancia máxima que un cliente puede saltar para entrar al vehículo.
- *MáximaVelocidadParaSubirse*: es aquella velocidad límite en la que se permite subir a un cliente al taxi.
- *AsientoConductor y AsientoPasajeros*: son 2 posiciones respectivamente que indican donde deben sentarse el conductor y los clientes.
- *TieneConductor*: es un booleano que se debe que desactivar si el vehículo no es descapotable para gastar menos recursos.
- *Armas*: es un *GameObject* que debe tener asociado el objeto poseedor del componente *Arma*.

Ilustración 4.83: Parámetros de CaracteristicasJugador

Dentro de la clase tenemos bastantes novedades en cuanto a métodos:

- *DestinoTaxi()*: este método tiene como entrada un punto de destino y lo único que hace es llamar al componente GPS del vehículo para que éste calcule la ruta.
- *ProporcionarAsiento()*: este método está pensado para devolver a un cliente la transformación del asiento de pasajeros asignada. Se le devolverá el asiento y por tanto se dejará subir a un cliente solo si éste no ha sido derribado, si el taxi no está ocupado, se encuentra cerca, y si lleva una velocidad que no supere el máximo establecido.
- *SumarPasajero()* y *quitarPasajero()*: tiene como parámetro de entrada un *GameObject* y sirve para añadir a la lista de pasajeros uno nuevo o para eliminarlo respectivamente.
- *CambiarConductor()*: permite cambiar al conductor por otro que se le pase.
- *BonusConducciónTemeraria()*: es un método que debe ser llamado por todos los objetos que nos dan el bonus *BonusConduccionTemeraria*. En el interior del método encontramos una llamada al método de *incrementarDinero()* del componente *Partida* del jugador, lo que proporcionará al jugador una pequeña cantidad de dinero si lleva un cliente montado en el taxi.

Al igual que los vehículos de los NPCs echan humo cuando averían, el del jugador también lo hace siguiendo la misma metodología.

4.6.1.4 Pruebas

A continuación, se van a mostrar los resultados de los test realizados para el componente de características.

T-01 Test de características	
Objetivo	Los objetos con el componente de características deben utilizar correctamente todos los atributos que éste trae
Resultado	Los objetos utilizan correctamente todos los atributos
Observaciones	

Tabla 4.29: Test de características

4.6.2 Armas

Dentro del juego la existencia de armas está presente en la mayoría de modos de juego, encontrándose en: el coche del jugador, las armas de los policías y en los tanques. Su gestión se ha llevado a cabo con la creación de 4 clases: *ArmaDistancia*, *ArmaJugador*, *ControladorArma* y *ArmaCuerpo*, lo que hace que nuevamente actualicemos una parte de nuestro UML.

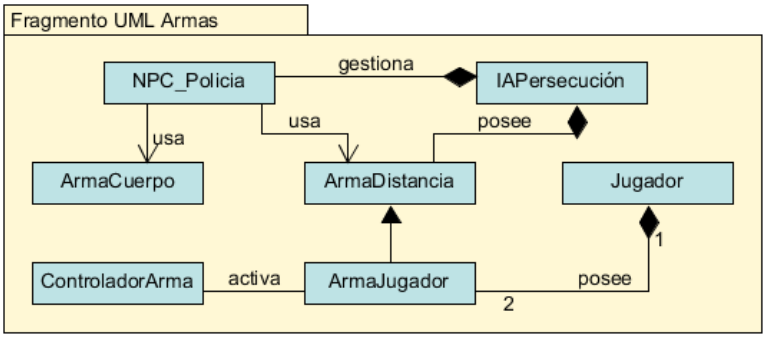

Ilustración 4.84: Fragmento de UML de Armas

4.6.2.1 ArmaDistancia

Esta clase se utilizará para recrear el funcionamiento de las armas a distancia, es decir, aquellas que disparan un proyectil. Existen 2 formas conocidas comunes para hacer el disparo de un arma en Unity:

1. **Mediante rayo**: se lanza un rayo en la dirección a donde se apunta y se estudia donde colisiona mediante *Physics.Raycast*.
2. **Mediante proyectil**: se mueve un proyectil desde el arma en la dirección a la que se apunta y cuando este colisiona gestiona su propia reacción.

Durante el desarrollo se ha seguido la segunda opción, por lo que posteriormente nos veremos en la necesidad de definir el objeto proyectil. Nuestra clase tiene los atributos siguientes:

- *PropietarioDisparo*: es el *GameObject* dueño del arma.
- *ObjetoProyectil*: es el *GameObject* que se dispara.
- *Proyectiles*: es un listado de los proyectiles disparados no destruidos.
- *MáximoProyectiles*: es el límite de proyectiles permitidos disparar a la vez.

- *Fuerza*: de este parámetro depende la velocidad del proyectil.
- *Retardo*: es el tiempo que hay que esperar entre 2 disparos.
- *ProbabilidadDesviacion*: es la probabilidad de que la dirección se altere según el porcentaje de desviación en el disparo.
- *Desviacion*: es el error que se añade como máximo en forma de porcentaje en la dirección de un disparo.

El método más importante de la clase es el método *disparar()*. Éste es llamado desde otra clase y se le pasa por parámetro una dirección de disparo. Si ha pasado el tiempo mínimo desde el último disparo entonces instancia un proyectil y lo añade a la lista de proyectiles indicando quien lo disparó y lo dispara haciendo uso de la función *addForce()* en la dirección indicada más un error aleatorio. Es posible que haya pasado el máximo de proyectiles que puede dispararse a la vez, en dicho caso se elimina en orden FIFO por su creación. Tras el disparo se activa durante unos segundos el sistema de partículas del arma para que parezca que ha echado un poco de humo por el cañón. En la parte inferior puede verse el código descrito.

```
1. //Generar proyectil
2. public void disparar(Vector3 direccion)
3. {
4.     //Normalizamos para que el error no sea desproporcionalmente grande o pequeño
5.     direccion = direccion.normalized;
6.     dir = direccion;
7.     if (puedeDisparar())
8.     {
9.         //Instanciamos el proyectil
10.        Vector3 error = new Vector3();
11.
12.        GameObject proyectil;
13.        //Error en función de la probabilidad
14.        if (((float)Random.Range(0, 100)) / ((float)100) < probabilidadDesviacion)
15.        {
16.            do
17.            {
18.                //Generamos un vector de error
19.                error = new Vector3(Random.Range(-desviacion * 100, desviación * 100) / 100,
20.                                    Random.Range(-desviacion * 100, desviación * 100) / 100,
21.                                    Random.Range(-desviacion * 100, desviación * 100) / 100);
22.                //Nos aseguramos que en verdad no hemos anulado la direccion
23.            } while (direccion + error == new Vector3());
24.        }
25.
26.        direccion += error
27.
28.        //Impulsar proyectil
29.        proyectil.GetComponent<Rigidbody>().AddForce(direccion * fuerza,ForceMode.Impulse);
30.
31.        //Borrar primero que se lanzó si hay demasiados. Funcionamiento de Queue
```

```
32.         if (proyectiles.Count > maximoProyectiles)
33.         {
34.             GameObject primerProyectil = proyectiles[0];
35.             proyectiles.RemoveAt(0);
36.             Destroy(primerProyectil);
37.         }
38.
39.         //Llamamos al sistema de partículas que simula el humo alrededor del cañón
40.         var emission = GetComponent<ParticleSystem>().emission;
41.         emission.enabled = true;
42.
43.         //Lo desactivamos en unos segundos
44.         Invoke(nameof(desactivarParticulas), 0.5f);
45.
46.         //Actualizamos la fecha del último disparo
47.         fechaUltimoDisparo = System.DateTime.Now.Second;
48.     }
49. }
```

Una vez disparado el proyectil, éste puede eliminarse automáticamente al chocar o puede ser eliminado si sobrepasa el máximo de proyectiles disparados a la vez, como hemos comentado. Tras la destrucción se actualiza el listado de proyectiles disparados eliminando el destruido.

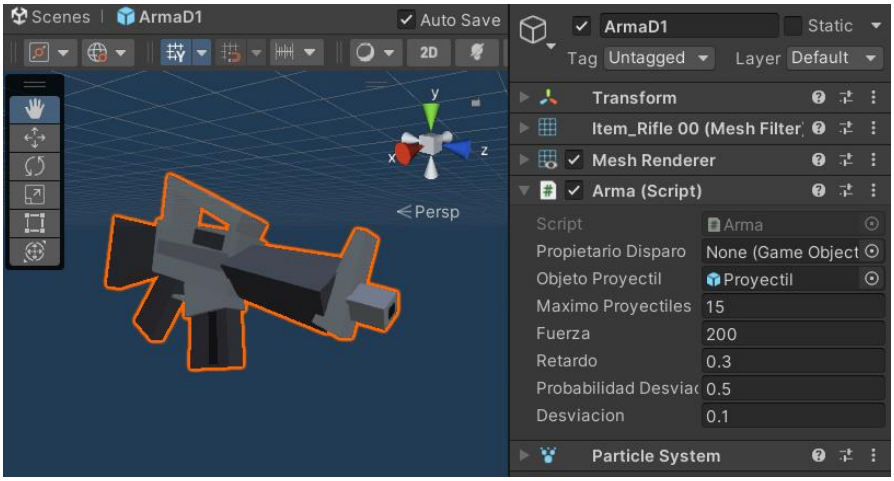

Ilustración 4.85: Parámetros de ArmaDistancia

4.6.2.2 ArmaJugador

Hereda de *ArmaDistancia* y se ha creado para los cañones que lleva el jugador a los lados. Esta clase se diferencia del arma a distancia en que ésta tiene una función llamada *apuntar()* en la que orienta el cañón que contiene el arma en la dirección a la que se apunta, dirección que es dada por el controlador del arma del jugador, pues se sigue el patrón Modelo-Vista-Controlador, desarrollado en profundidad en el apartado 4.10.1.2 (Aplicación de MVC a las armas del jugador). Igualmente, y siguiendo la filosofía del patrón, la función de *disparar()* será activada por la clase controladora. En

la Ilustración 4.86 podemos ver las 2 armas verdes del jugador, cada una con su propio script *ArmaJugador*.

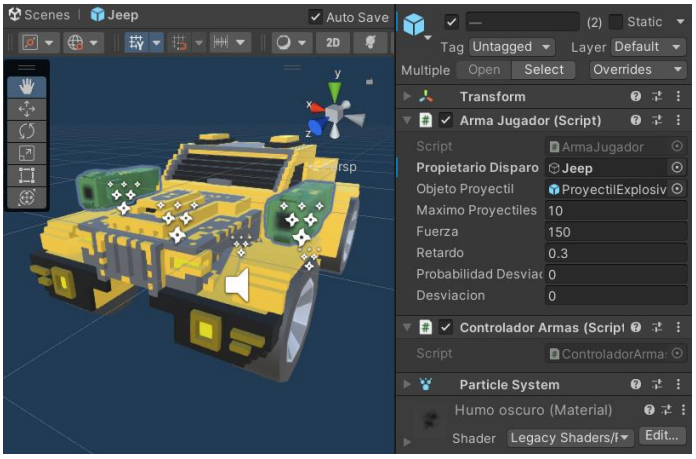

Ilustración 4.86: Parámetros de ArmaJugador

Cuando se da la posibilidad de disparar al jugador, además se modifica el cursor y se convierte en una mirilla. De esta manera el jugador, apuntando con el ratón puede disparar, ya que se pasa la posición del ratón en una posición de mundo y se dispara en dirección a la misma.

4.6.2.3 ArmaCuerpo

Hasta ahora solo hemos hablado de armas a distancia, pero en el juego se ha creado también un arma con forma de bate que funciona por colisión. Es una clase con pocos atributos, destacando los siguientes:

- *Poseedor*: *GameObject* dueño del arma.
- *Daño*: indica el daño que hace por colisión. En este caso sí que es el arma el que produce el daño y no el proyectil como pasaba en los anteriores casos.
- *Retardo*: es el tiempo mínimo entre 2 golpes.
- *EnUso*: si tiene valor true sus colisiones ocasionan daño.

Es una clase sencilla que tiene un método get y set para ver y modificar el valor de la variable *enUso*. En caso de que esta variable esté a valor verdadero, gestionamos la colisión dentro del método *OnColisionEnter()* en caso de haberla. En

concreto lo que se hará es estudiar el objeto con el que colisiona, y si encuentra en él el un componente de *Caracteristícas* llama al método *restarVidaPorPuntos()* indicando el daño recibido. Una vez bajada la vida le aplica una fuerza si se encuentra el componente de *RigidBody* para que el objeto se vea empujado por el golpe mediante la función *AddForce*().

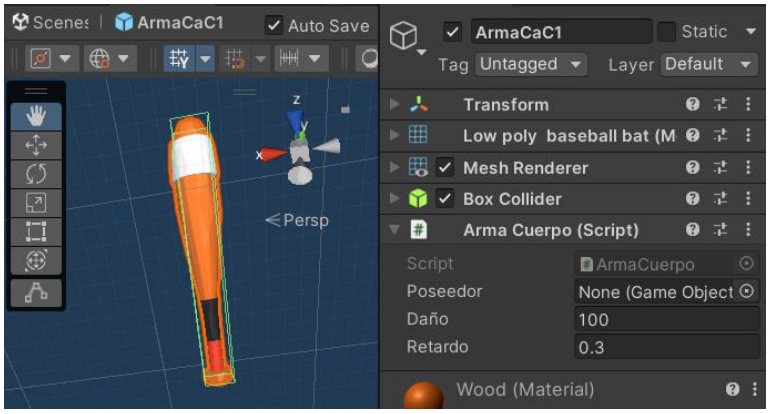

Ilustración 4.87: Parámetros de ArmaCuerpo

4.6.2.4 Proyectil

Un proyectil es instanciado por un arma y realiza daño al objeto con el que colisiona si no es el mismo con el que se disparó.

Se trata de un objeto con un componente *BoxCollider*, un *RigidBody* (al que le hemos quitado la gravedad) y un sistema de partículas que genera partículas en forma de prisma en la parte trasera del proyectil cíclicamente con una textura de humo.

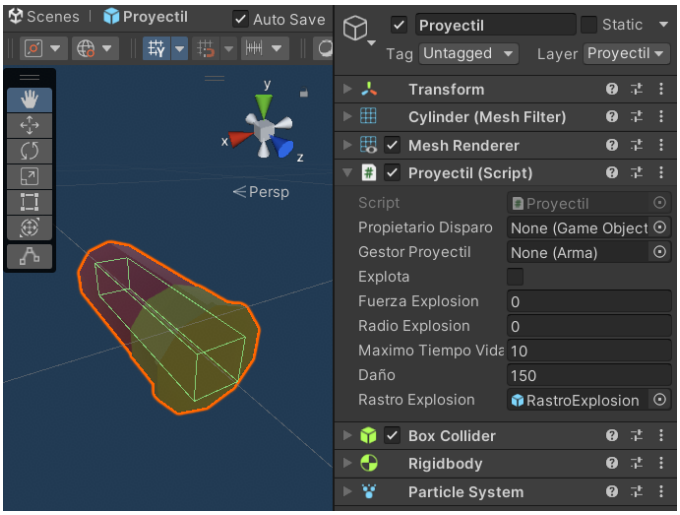

Ilustración 4.88: Parámetros del proyectil

Tiene una serie de atributos de los cuales vamos a recalcar:

- *Daño*: indica el daño que hace por colisión.
- *Explota*: es un booleano que indica si el proyectil realiza una explosión al destruirse y como resultado un daño de área.
- *FuerzaExplosion*: es la fuerza que tiene la detonación de la explosión
- *MaximoTiempoVida*: es el tiempo máximo que tiene el proyectil antes de autodestruirse.
- *RastroExplosion*: es un *GameObject* con un sistema de partículas configurado para que genere un humo en forma cónica hacia arriba durante varios segundos.

Como observamos analizando estos atributos el proyectil puede ser de 2 tipos, explosivo o no explosivo.

- **Si no es explosivo**: cuando detecta una colisión accede a su componente *Características* y pide la reducción de la vida con el método *restarVidaPorPuntos()*, al que se le pasa el daño inflingido.
- **Si es explosivo**: se llamará a una función llamada *explotar()* cuyo funcionamiento se resume en generar un colisionador esférico en tiempo de ejecución en la posición del contacto. Despues de ésto, se estudian los objetos con los que esta colisionando la esfera y a todos los que tengan *Características* se les reduce la vida. Por otro lado si tienen el componente *RigidBody* se les aplica una fuerza explosiva con *AddExplosionForce*(), que simula la explosión.

  Finalmente para mejorar nuestro disparo explosivo se añade en la jerarquía del objeto con el que se colisiona el objeto *rastroExplosion* que generará humo en el lugar del contacto durante varios segundos, dando un efecto más realista. El código inferior muestra la función de la generación de la explosión del proyectil explosivo.

```
1. void explotar()
2. {
3.     Vector3 explosionPos = transform.position;
4.     Collider[] colliders = Physics.OverlapSphere(explosionPos, radioExplosion);
```

```
 5.
 6.      foreach (Collider hit in colliders)
 7.      {
 8.          Caracteristicas recExp = hit.GetComponent<Caracteristicas>();
 9.          if (recExp) recExp.restarVidaPorPuntos(daño);
10.
11.          Rigidbody rigid = hit.GetComponent<Rigidbody>();
12.          if (rigid != null)
13.              rigid.AddExplosionForce(fuerzaExplosion * rigid.mass,
14.                                      explosionPos, radioExplosion, 3.0f);
15.      }
16.
17.      //Dejamos rastro de la explosion en el objeto que colisiona
18.      GameObject gameobj = Instantiate(rastroExplosion,transform.position,transform.rotation);
19.      if (objetoColisionado)
20.      {
21.          //Cambiamos el padre para que el humo quede pegado al objeto
22.          gameobj.transform.parent = objetoColisionado.transform;
23.      }
24. }
```

Una vez hecho el daño, sea explosivo o no, un proyectil se destruye haciendo uso de *Destroy()* pidiendo antes que se nos saque de la lista de proyectiles disparados que recordamos que tienen las *ArmaDistancia*.

Cabe destacar que este método tiene 2 principales inconvenientes frente al método de disparos sin proyectiles (método por rayo):

1. Es importante llevar una correcta gestión de los proyectiles porque son una fuente de pérdidas de recursos y problemas, por lo que para evitar que haya misiles desperdigados por el mapa buscando una colisión que no va a llegar se debe incluir un tiempo de vida máximo, el cual hace que el misil desaparezca como si hubiera chocado tras pasar unos segundos desde su creación.

2. Si la velocidad de la bala es muy alta el sistema de físicas de Unity podría no llegar a detectar la colisión, pues éste no revisa el estado de las colisiones de forma continua.

   Durante el proyecto este problema se ha dado y se ha barajado la posibilidad de aumentar la tasa de refresco del sistema de físicas, pero investigando por Internet se llegó a una conclusión obvia, si aumentamos la tasa de refresco vamos a tener un rendimiento infinitamente menor al actual por intentar que un par de balas funcionen mejor. Dado que buscamos evitar reducir el rendimiento esa opción quedó descartada y la solución que se dio fue la de reducir la velocidad del proyectil.

4.6.2.5 Pruebas

A continuación, se van a mostrar los resultados de los test realizados para las armas y proyectiles.

T-01 Test de arma a distancia	
Objetivo	Estas armas deben ser capaces de enviar un proyectil al frente dada una dirección de disparo
Resultado	Los proyectiles son creados correctamente por el arma, enviados en la dirección especificada. También se eliminan si hay demasiados al mismo tiempo
Observaciones	

Tabla 4.30: Test de arma a distancia

T-02 Test de arma cuerpo a cuerpo	
Objetivo	Estas armas producen daño cuando colisionan con un objeto si están usándose para atacar
Resultado	Las armas reducen correctamente la vida del objeto de características cuando colisionan en mitad de un ataque
Observaciones	

Tabla 4.31: Test de arma cuerpo a cuerpo

T-03 Test de arma del jugador	
Objetivo	El jugador debe ser capaz de utilizar las armas del vehículo y apuntar con el cursor
Resultado	El apuntado con el cursor es correcto y las armas disparan proyectiles explosivos correctamente
Observaciones	El jugador puede dispararse a sí mismo al apuntar al suelo

Tabla 4.32: Test de arma del jugador

T-04 Test de proyectil común	
Objetivo	El proyectil es capaz de seguir la trayectoria indicada, autodestruirse al chocar y producir daño al objeto con el que choca si éste tiene el componente de características
Resultado	Los proyectiles siguen correctamente la dirección establecida, se autodestruyen al chocar y provocan la reducción de vida.
Observaciones	

Tabla 4.33: Test de proyectil común

T-05 Test de proyectil explosivo	
Objetivo	Los misiles deben seguir la dirección indicada, autodestruirse al chocar, realizar una explosión que afecte y reduzca vida a todos los objetos de alrededor además de al objeto colisionado. Deben generar un humo en el lugar del choque que durará varios segundos
Resultado	Los misiles se mueven y realizan la explosión tal y como se exige de manera correcta
Observaciones	

Tabla 4.34: Test de proyectil explosivo

4.6.3 Bonus

En este apartado vamos a hablar de los 2 diferentes bonus que han sido implementados en el juego, que otorgan dinero al jugador cuando se activan. Es un objeto que no estaba en un principio planteado dentro del UML inicial y por tanto se debe incluir como tal.

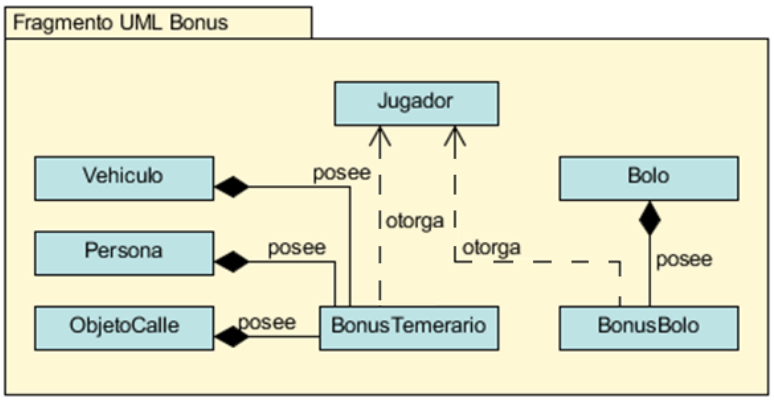

Ilustración 4.89: Expansión de UML para los bonus

4.6.3.1 BonusConduccionTemeraria

Es un script que se puede asociar a objetos de la ciudad, peatones NPC y vehículos NPC que proporciona una cantidad de dinero al jugador si tiene pasajeros a bordo del taxi.

Su funcionamiento está basado en el método *OnColliderEnter()*, el cual cuando se activa por contacto investiga si el objeto que nos ha colisionado es el jugador. En tal caso se usa el método *bonusConduccionTemeraria()* de las *CaracteristicasJugador* para incrementar el dinero del componente *Partida*. El incremento solo se puede hacer

una vez, por lo que una vez otorgado el bonus un booleano se activa impidiendo que se pueda volver a entregar.

Además este script tiene un método público llamado *bonusConduccionTemeraria()* desde el cual también se puede activar el bonus haciendo uso de él (lo usan los NPCs).

4.6.3.2 BonusBolo

Es un script asociado a los bolos del modo de bolos mostrado en el apartado 5.2.6 (Crazy Box: modo de bolos) y se activa cuando un bolo mide su inclinación y ésa ha variado mucho en los ejes X y Z respecto al inicio. Concretamente la variación tiene que ser de 75º, medición llevada con el método *Angle()* de la clase *Vector2*, lo que asegura que el bolo ha sido colisionado y derribado.

Dado que este bonus es para su uso en bolos, cuando se activa reproduce un *AudioClip* con el sonido de un golpe a un bolo.

4.6.3.3 Pruebas

A continuación, se van a mostrar los resultados de los test realizados para los bonus.

T-01 Test de bonus	
Objetivo	Ambos bonus deben proporcionarse de manera correcta una única vez y asignar el dinero correspondiente al jugador
Resultado	El bonus se asigna correctamente cuando la condición que lo activa se llama por primera vez. Además, el bonus de conducción temeraria se puede activar de forma remota por un peatón al esquivar el vehículo
Observaciones	

Tabla 4.35: Test de bonus

4.7 Elementos específicos de modo

4.7.1 Bolo

El objeto bolo es un objeto que aparece generado en grupos de 10 y que se utiliza solamente dentro del modo de bolos. Podemos ver en la Ilustración 4.90 como el bolo es un objeto muy simple. Este es conformado por los componentes de *RigidBody* y *MeshCollider* para que tenga físicas y caja de colisiones, además del script *BonusBolo*.

Ilustración 4.90: Bolo

Nótese el detalle del normal mapping aplicado sobre la textura para dar un efecto de pixelado.

4.7.2 Fuego: técnica billboard

Es un elemento completamente estético programado para su uso en el modo de descrito en el apartado 5.2.5 (Crazy Box: modo apocalipsis). Comencemos viendo la imagen del resultado, Ilustración 4.91, para proceder a comentarlo.

Ilustración 4.91: Fuego

Para hacer el fuego podría haberse usado la opción de generar un sistema de partículas, sin embargo, todo aquel que haya usado Unity sabe que si se abusa de sistemas de partículas al final el rendimiento decae mucho. En nuestro caso, como la idea es generar fuego sobre la ciudad y dejarlo ahí durante toda la partida, acabaríamos notando la bajada de frames por segundo (fps) como consecuencia del fuego.

Es por ello que tenemos que buscar otra solución y es por esto que se ha hecho uso de la técnica de billboarding [52]. Esta técnica busca aumentar el rendimiento de un sistema gráfico cambiando objetos tridimensionales por una textura que siempre mire a cámara. De esta forma, ahorramos una gran cantidad de geometría y no dependemos de los sistemas de partículas de Unity.

La implementación del billboard ha sido simple pues solo se ha tenido que generar un plano con una textura con transparencia aplicada y hacer que éste mire a cámara. Añadido a esto y para darle movimiento (algo que el sistema de partículas sí que daba de forma literal) lo que se hará es cambiar cada varios frames el fotograma que se muestra por uno que continúe el movimiento del fuego. Para conseguir estos fotogramas ha sido tan fácil como buscar un gif y separar sus fotogramas para aplicarlos sobre el plano.

4.7.3 Pruebas

A continuación, se van a mostrar los resultados de los test realizados para los elementos específicos.

T-01 Test de bolo	
Objetivo	El bolo debe ser capaz de mantener su equilibrio hasta que sea colisionado con cierta fuerza. Su inclinación debe ser controlada para poder asignar el bonus por tirarlo.
Resultado	El bolo mantiene su equilibrio y puede tirarse cuando es golpeado por el jugador, lo que proporciona un bonus de manera correcta
Observaciones	Algunos bolos son despedidos al atropellarlos debido al sistema de físicas de Unity

Tabla 4.36: Test de bolo

T-02 Test de fuego	
Objetivo	La llama debe mirar siempre a cámara y situarse sobre el suelo de la ciudad
Resultado	La llama se sitúa correctamente y mira siempre a cámara
Observaciones	Estéticamente mejorable en la zona que interseca con el suelo

Tabla 4.37: Test de fuego

4.8 Gestores

4.8.1 Gestor de vehículos NPC

Por lo general en videojuegos de mundo abierto, los enemigos, animales, NPCs… suelen generarse alrededor del jugador sin que éste lo note para no desperdiciar recursos y además asegurar que los recursos que se dedican sean aprovechados y vistos por el jugador.

El caso de nuestro proyecto no será diferente, pues la gestión de vehículos ha sido un factor clave junto con la gestión de peatones para mejorar el rendimiento del juego y conseguir recrear una zona viva y llena de NPCs alrededor del área donde se encuentra el jugador. Gracias al gestor, solamente tendremos que generar sobre unos 30 vehículos (depende de la configuración elegida) cerca del jugador para conseguir el efecto que sin el gestor requeriría de cientos o miles de vehículos y que por supuesto sería inviable.

El gestor de vehículos se ocupa de la instanciación y destrucción de todos los tipos de vehículos presentados y lo hace mediante 2 corrutinas que son activadas cada cierto tiempo. Es importante llevar a cabo ese proceso con corrutinas porque hacerlo desde el método *Update()* en cada fotograma actuaría de freno para el rendimiento y bajaría mucho el número de frames por segundo. Por otro lado, tampoco está justificado analizar en todo momento si debe añadirse o eliminarse un vehículo, ya que de un fotograma a otro normalmente no hay cambios en lo que a crear y destruir vehículos se refiere.

Centrándonos en el funcionamiento de las corrutinas, básicamente se organiza de la siguiente manera:

- **Corrutina de generación de vehículos**: instancia el número indicado sin pasarse de vehículos de cada tipo, de forma aleatoria sobre la vía, a una distancia de la posición del jugador determinada por un rango y alejados unos vehículos de otros por un espacio mínimo. Cada vehículo instanciado se introduce en una lista de vehículos de su mismo tipo, gestionada por el gestor de vehículos, y se mantiene ahí hasta su destrucción.
- **Corrutina de destrucción de vehículos**: se ocupa de destruir todo aquel vehículo que esté demasiado lejos del jugador o que se encuentre inactivo. La inactividad recordemos que puede verse gracias a las *CaracteristicasVehiculos*, y permitirá que si el vehículo está inactivo se destruya cuando esté fuera del campo de visión de la cámara.

  Tanto el jugador como los vehículos policiales y de guerra cuando realizan sus respectivas actividades pueden chocar y desviar a otros vehículos, que quedan volcados, chocados contra muros o destruidos y por tanto quedan detenidos sobre la calzada obstruyendo la circulación. La destrucción de eficiente y transparente de vehículos inactivos es por tanto un punto crucial para evitar atascos molestos para el jugador (atascos que a veces el jugador no sabría ni por qué se han dado).

En la Ilustración 4.92 se puede ver como el mapa genera vehículos únicamente en un área que rodea al taxi del jugador.

Ilustración 4.92: Area con vehículos

4.8.1.1 Pruebas

A continuación, se van a mostrar los resultados de los test realizados para el gestor de vehículos.

T-01 Test del gestor de vehículos	
Objetivo	El gestor debe ser capaz de colocar y destruir vehículos alrededor del jugador, sin que haya colisiones entre vehículos ni con el jugador e intentando que siempre haya el mismo número de ellos
Resultado	El gestor coloca y destruye vehículos correctamente alrededor del jugador, destruyendo no solo los vehículos lejanos, sino que también los vehículos inactivos
Observaciones	

Tabla 4.38: Test de gestor de vehículos

4.8.2 Gestor de peatones NPC

Es un script que sigue la misma filosofía que el gestor de vehículos, pues gestiona la creación y destrucción de peatones mediante corrutinas, y de igual forma los organiza en diferentes listas, esta vez divididas por su comportamiento en lugar de por el tipo de vehículo.

La dificultad añadida de este gestor reside en la idea de que los peatones puedan cambiar su comportamiento, y por ende tengan que pasar de una lista a otra dentro del mismo gestor (por ejemplo de lista de clientes a lista de zombis). Por esta razón, una vez un peatón se cambia de comportamiento debe avisar del suceso al gestor para que éste lo reubique dentro de las listas, para lo cual:

- Se saca al peatón del grupo, vehículo policial o lista en la que se gestionaba.
- Si el peatón pasa a ser de comportamiento grupal, éste tiene que indicar a que grupo se quiere unir, y el gestor lo integra dentro del nuevo grupo.
- Si el peatón pasa a ser de comportamiento policial, tiene que indicar a que vehículo policial se quiere unir, igualmente se introduce en el vehículo.
- Si el peatón pasa a ser individual, cliente o zombi, el gestor lo reubica automáticamente en su lista correspondiente propia.

Hacer esto tan flexible trae consigo otro problema, y es que, como tenemos marcado un máximo de peatones para cada lista y éstos tienen permitido cambiar su comportamiento, aparece la posibilidad de que la lista de comportamiento que acoge a un peatón que modificó su comportamiento supere el máximo de NPCs que debería tener y termine colapsando poco a poco el juego. Esto es porque hasta ahora, la destrucción de NPCs se hacía igual que en los vehículos, es decir, se eliminan por la lejanía de estos. Para evitar este problema, se tuvo que ampliar la condición de destrucción para que eliminarse también NPCs cuando se superaba el máximo de los aceptados en una lista, pero nuevamente apareció un problema más. Concretamente éste se da de forma pronunciada en el modo visto en el apartado 5.2.5 (Crazy Box: modo apocalipsis) y lo que sucede es que si generamos X número de zombis (donde X es el máximo que se deben crear), y se contagian Y, llegamos a que se supera el número de X zombis máximo y los Y contagiados se eliminan instantáneamente, por lo que hay que introducir un nuevo máximo un poco más alto para que un cambio de comportamiento no repercuta siempre en la eliminación de un NPC, a menos que la adición de NPCs a la lista correspondiente sea desproporcionalmente alta.

4.8.2.1 Pruebas

A continuación, se van a mostrar los resultados de los test realizados para el gestor de peatones.

T-01 Test de gestor de peatones	
Objetivo	El gestor debe ser capaz de colocar y destruir personas NPC alrededor del jugador, sin que haya colisiones entre ellas e intentando que siempre haya el mismo número
Resultado	El gestor coloca y destruye peatones correctamente, mostrando siempre un mínimo de NPCs y permitiendo en casos excepcionales aumentar algo más la cantidad indicada
Observaciones	Se encontraron y corrigieron bugs acerca de las listas de NPCs, que enviaban peatones a listas asociadas a comportamientos que no les correspondían

Tabla 4.39: Test de gestor de peatones

4.8.3 Gestor del juego

Es un script que tiene la misión de preparar la escena para hacer jugable el modo seleccionado. Literalmente este script es el único que conoce a qué se está

jugando, desacoplando el resto de componentes de los diferentes modos de juego y configuraciones que puedan darse.

Conforme inicia la escena este objeto busca al objeto de configuración del juego, *ConfiguracionJuego*, y procede a realizar las siguientes operaciones dentro del método *Awake()*:

- Lee cual es el *AudioClip* que debe reproducirse para la música de fondo y lo pone en bucle.
- Lee las dimensiones, densidad de calles, la magnitud de las cuestas y modelos de edificios para generar la ciudad.
- Instancia el vehículo y el conductor indicados por la configuración. Además activa las armas si así se indica por la configuración.
- Se recrea una ambientación siguiendo los valores indicados en los parámetros de niebla, skybox [59] y de luz de la configuración.

Una vez terminado de ejecutar el *Awake()* y la ciudad está creada pasa a activarse el método *Start()*, donde sigue realizando ajustes para la creación del modo de juego en base a la configuración dada:

- Activa el *GPS* del jugador ahora que hay ciudad disponible.
- Cambia el cursor al cursor correspondiente indicado por la configuración.
- Si la configuración lo pide, genera fuego sobre la ciudad.
- Si la configuración lo pide, puede generar bolos sobre la ciudad, puede desactivar el tiempo máximo de partida, puede desactivar la puntuación e incluso puede activar la inmunidad para el jugador.
- Genera el gestor de peatones NPC y el gestor de vehículos NPC, y luego les indica todos los parámetros necesarios, dados igualmente por la configuración.

Cabe destacar que este script es el que proporciona a los vehículos policiales un objetivo policial en función del modo de juego, dándose la posibilidad de devolver

al jugador o al zombi con vida más cercano (en este segundo caso el zombi sería pedido nuevamente por parte del gestor del juego al gestor de peatones NPC).

4.8.3.1 Pruebas

A continuación, se van a mostrar los resultados de los test realizados para el gestor general del juego.

T-01 Test de gestor principal	
Objetivo	El gestor debe ser capaz leer la configuración del objeto configurador y en base a ello crear el nivel
Resultado	El gestor puede leer la información del objeto configurador y permite crear diferentes modos de juego en consecuencia
Observaciones	

Tabla 4.40: Test de gestor principal

4.9 Interfaz

4.9.1 Cámara

Se ha creado una cámara personalizada que se utiliza durante la escena de juego mediante los controles correspondientes comentados en el manual de usuario. Ésta sigue el patrón modelo vista controlador, desarrollado en el apartado 4.10.1.3 (Aplicación de MVC a la cámara del jugador). Podemos visualizar al jugador de 4 maneras:

- Primera persona
- Tercera persona cerca
- Tercera persona lejos
- Mirar hacia atrás

El script que maneja la cámara es muy personalizable, pues permite ajustar para cada uno de los 4 modos la posición, rotación y velocidad de movimiento de la cámara.

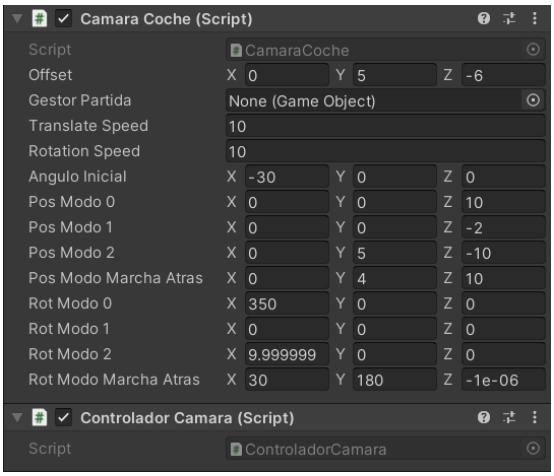

Ilustración 4.93: Parámetros de la cámara

Si analizamos la forma en la que está construido el prefab nos damos cuenta de que es un objeto que contiene a otro, y éste último incluye la flecha indicadora (apartado 4.9.5.7.6):

- El primer objeto se ocupa de seguir al jugador mediante un movimiento interpolado obtenido con la función *Lerp()*.
- El segundo objeto trae incorporado el componente *Camera* [60] y un componente de post-procesamiento con antialiasing FXAA [61]. Éste se ocupa de cambiar la ubicación local de la cámara dependiendo del modo de cámara elegido.

  Si analizamos cómo se mueve este segundo objeto, lo que hace es que cuando recibe la orden de cambio de cámara se dentro del script se modifica el punto hacia donde debe dirigirse y realizamos el movimiento gracias a una función que se llama constantemente dentro del método *Update()* que realiza, con algunos retoques en el script, básicamente el siguiente cálculo:

  $$posicion \mathrel{+}= (posicionDestino - posicion) \bullet multiplicadorVelocidad$$

  Al llamarse constantemente y ser cada vez el valor de *(posicionDestino - posicion)* más pequeño, se consigue un comportamiento similar al de una función logarítmica, de manera que inicialmente se mueva muy rápido y conforme llega a la posición requerida vaya moviéndose más lentamente. Análogamente pasa con la rotación.

4.9.1.1 Pruebas

A continuación, se van a mostrar los resultados de los test realizados para la cámara.

T-01 Test de modos de cámara	
Objetivo	La cámara debe ser capaz de cambiar su ubicación local para permitir diferentes tipos de cámara. Los cambios de posición deberían ser suaves
Resultado	La cámara consigue suavemente moverse entre una posición local y otra para obtener varios puntos de vista, dando pie a cámara de primera persona, tercera persona cerca, tercera persona lejos y cámara trasera
Observaciones	La cámara trasera a veces tarda un poco en volver a mirar al frente

Tabla 4.41: Test de modos de cámara

T-02 Test de seguimiento de cámara	
Objetivo	La cámara tiene que ser capaz de seguir al jugador mediante movimientos suaves
Resultado	La cámara sigue correctamente al jugador mediante movimientos suaves y continuos
Observaciones	Al acelerar mucho la cámara de primera persona asciende un poco debido al movimiento interpolatorio que tiene

Tabla 4.42: Test de seguimiento de cámara

4.9.2 Sonido

A lo largo del desarrollo del proyecto se ha indicado numerosos sitios donde hay audios o eventos asociados a ellos. Sin embargo, en esta sección no vamos a volver a hablar de donde se ubican, sino que nos centraremos en un detalle que se observa en el apartado 4.9.5.4 (Menú de ajustes). Concretamente si nos fijamos, el juego permite cambiar el volumen de la música de fondo y el volumen de los sonidos por separado, es decir, hay 2 canales de audio diferentes. Esto se ha hecho gracias a un mezclador de audio, *AudioMixer* [62] en Unity.

Un *AudioMixer* es un elemento proporcionado por Unity que permite crear diferentes grupos de audio para posteriormente aplicarles efectos por separado.

Nuestro juego en particular usa 2, uno para la música y otro para sonidos. Lo único que debe hacerse a la hora de colocar un audio es indicar a qué grupo pertenece.

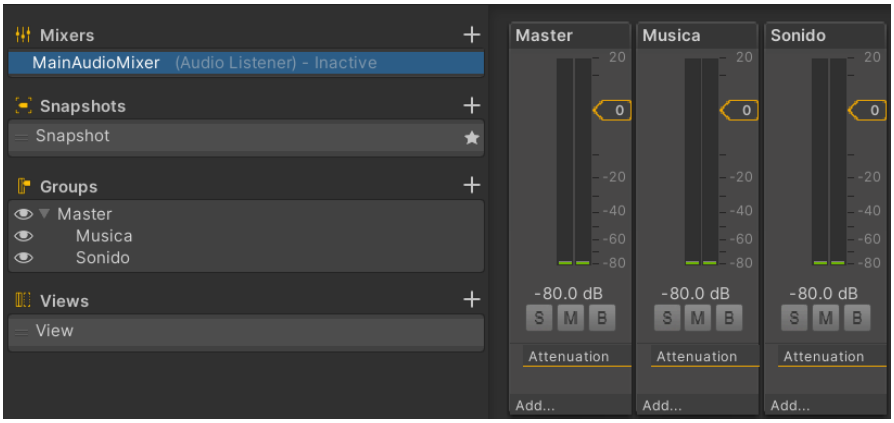

Ilustración 4.94: Mezclador de audio

Nótese en la Ilustración 4.94 que se trabaja en decibelios, por lo que para subir y bajar el volumen dentro del scripting debe realizarse conforme a la siguiente fórmula, pues no se trabaja de forma lineal:

$$a = 20 * Log_{10}(porcentaje)$$

Sea a el valor en decibelios calculado y la variable porcentaje un valor perteneciente al intervalo (0, 1].

4.9.2.1 Pruebas

A continuación, se van a mostrar los resultados de los test realizados para el sonido.

T-01 Test de sonido y mezclador	
Objetivo	El sonido del juego debe estar separado en 2 canales
Resultado	El sonido del juego se ha separado en varios canales y se escucha correctamente
Observaciones	

Tabla 4.43: Test de sonido y mezclador

T-02 Test de volumen	
Objetivo	El sonido del juego debe permitir subir y bajar el volumen de manera constante
Resultado	El sonido del juego se trabaja en decibelios teniendo en cuenta su funcionamiento no lineal, haciendo que se pueda subir y bajar de manera constante de forma correcta
Observaciones	

Tabla 4.44: Test de volumen

4.9.3 Funciones de entrada para controles

La lectura de los controles se hará haciendo uso de los axes [63] de Unity para que dentro del propio menú de Unity se puedan modificar rápidamente los controles desde la dirección: *Edit* > *Proyect Settings* > *Input Manager* > *Axes*.

Los *axes* en Unity se tratan de entradas de control que se utilizan para medir la magnitud y dirección de movimientos en una o varias dimensiones.

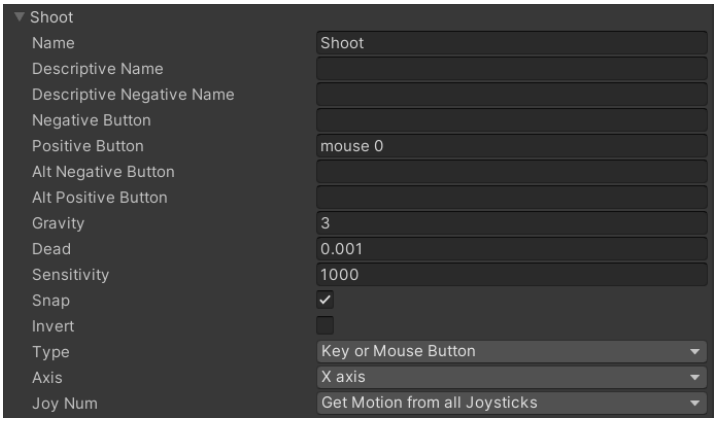

Ilustración 4.95: Axis de disparo

En la Ilustración 4.95 vemos el axis utilizado para leer el control del disparo del arma del jugador.

4.9.4 Textos y traducción: internacionalización

Durante todo el juego se utiliza una fuente de letra pixelada, obtenida del repertorio de fuentes de Google Fonts.

Por otro lado, el juego se ha preparado pensando en la internacionalización del mismo, con objetivo de que todos sus textos puedan ser traducidos al idioma que se desee sin necesidad de modificar la codificación del mismo.

Para ello se ha creado un objeto que se ocupa de gestionar la traducción dada una clave, y éste devuelve el texto traducido en función de la clave y del idioma seleccionado leyendo de un documento guardado en formato .csv muy fácil de modificar desde aplicaciones como Excel. De esta manera, lo único que un trabajador tendría que saber hacer para incluir nuevas traducciones es modificar el documento, por lo que tenemos la ventaja de que no solo no se tiene que modificar el código, sino que además el traductor no necesita tener conocimientos de codificación ni de Unity.

Es importante recalcar que las traducciones mostradas en el juego son traducciones hechas mediante un traductor automático y que no han sido revisadas, cuyo único objetivo es mostrar el correcto funcionamiento del sistema.

4.9.4.1 Pruebas

A continuación, se van a mostrar los resultados de los test realizados para los textos.

T-01 Test de traducción	
Objetivo	Todo el texto debe ser capaz de soportar la traducción a otro idioma
Resultado	El juego soporta en todos los textos que incorpora la traducción a otro idioma de entre los propuestos de ejemplo
Observaciones	Si el idioma contiene caracteres no recogidos en la norma ISO 8859-1 no podrá soportar todos los textos

Tabla 4.45: Test de traducción

4.9.5 Menús e interfaz de partida

4.9.5.1 Menú principal

La creación de este menú ha buscado ser, como vimos en el apartado 2.3 (Diseño de la interfaz), lo más fiel posible al menú del juego original al más puro estilo de Minecraft, dando el resultado de la Ilustración 4.96.

Ilustración 4.96: Menú principal reboot

El menú propio está encabezado por una descripción que sirve al usuario para saber en qué parte del menú se encuentra. En este menú encontramos un total de 4 botones de navegación:

1. Botón de Arcade: nos lleva a los modos de juego originales.
2. Botón de Crazy box: nos lleva a los modos de juego personalizados y nuevos creados durante este proyecto (con la excepción del modo del apartado Crazy Box: modo de bolos, que sí que es original de la saga Crazy Taxi).
3. Botón de Ajustes: que nos lleva al menú de configuración para indicar parámetros como el idioma, variables para la generación de ciudades, dificultad…
4. Botón de Salir: con el que salimos de la aplicación.

Existe un objeto, objeto *Menu*, que gestiona el paso de un menú a otro en base a las pulsaciones de los botones y sigue un funcionamiento similar al funcionamiento del patrón observador [37], aunque ciertamente no se implementó el patrón por completo tal y como vemos en el apartado 4.10.4 (Patrón Observador aplicado a los menús). El funcionamiento es simple, cuando el usuario toca un botón se activa su

método *OnClick()* y éste a su vez avisa al objeto que gestiona los menús de que el botón ha sido pulsado mediante el método correspondiente.

Tiene una música de fondo que se repite en bucle.

4.9.5.2 Menú arcade

Es el menú que se muestra si se pulsa el botón de Arcade en el menú principal. La reacción del objeto *Menu* que gestiona los menús es la siguiente:

1. Activa una animación de los botones y descripción donde éstos se desplazan en el eje X hasta desaparecer. Además, se activa un sonido de motor acelerando.
2. Activa otra animación donde otros botones y una nueva descripción aparecen deslizándose en el eje X. Esta nueva descripción ahora muestra "ARCADE" y tiene los botones que llevan al modo visto en el apartado Arcade: modo reglas arcade y al modo visto en el apartado Arcade: modos de trabajo.

Estas animaciones han sido creadas igual que las que se ven en el Crazy Taxi de SEGA. La apariencia tras la animación es la que muestra la Ilustración 4.97: Menú arcade.

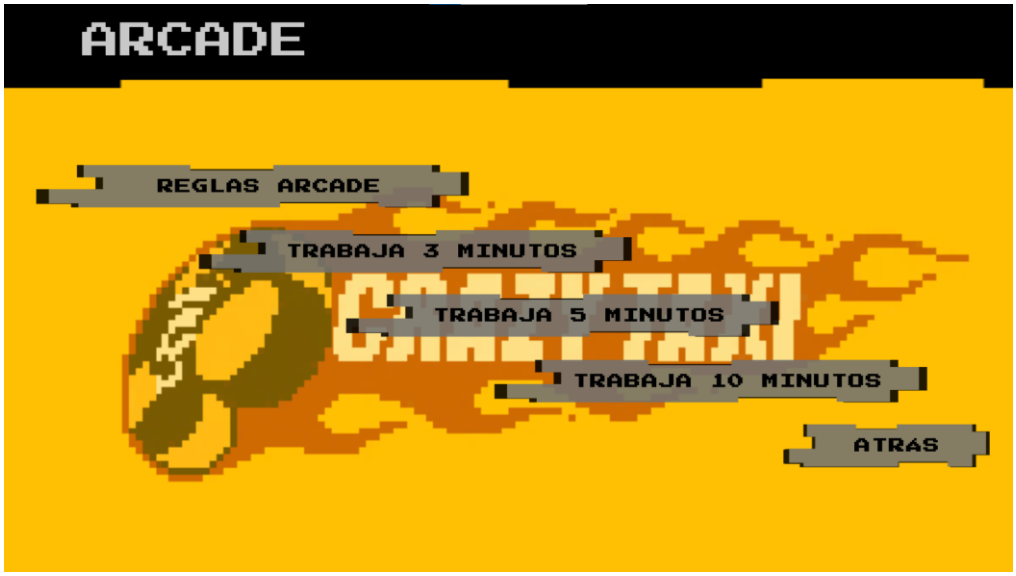

Ilustración 4.97: Menú arcade

Si se pulsa cualquier botón que no sea el de ir hacia atrás se cargará el menú de visto en el apartado Menú de selección de personaje tras una animación en la que la pantalla se oscurece tras el escalado de la barra negra de la descripción. Internamente se indicará el modo de juego que se va a jugar dentro del objeto *ConfiguracionJuego*, modos que detallaremos en el apartado 5 (Resultados).

Un detalle del menú es los botones han sido hechos a mano y el fondo se ha obtenido con la pixelación del fondo original y retoques manuales.

4.9.5.3 Menú crazy box

Tras pulsar el botón de Crazy box, este menú aparece y desaparece de la misma forma que el Menú arcade, con la diferencia de que ahora la configuración interna que hará será con los modos que contiene el menú, visibles desde la Ilustración 4.98, y desarrollados en el apartado 5 (Resultados).

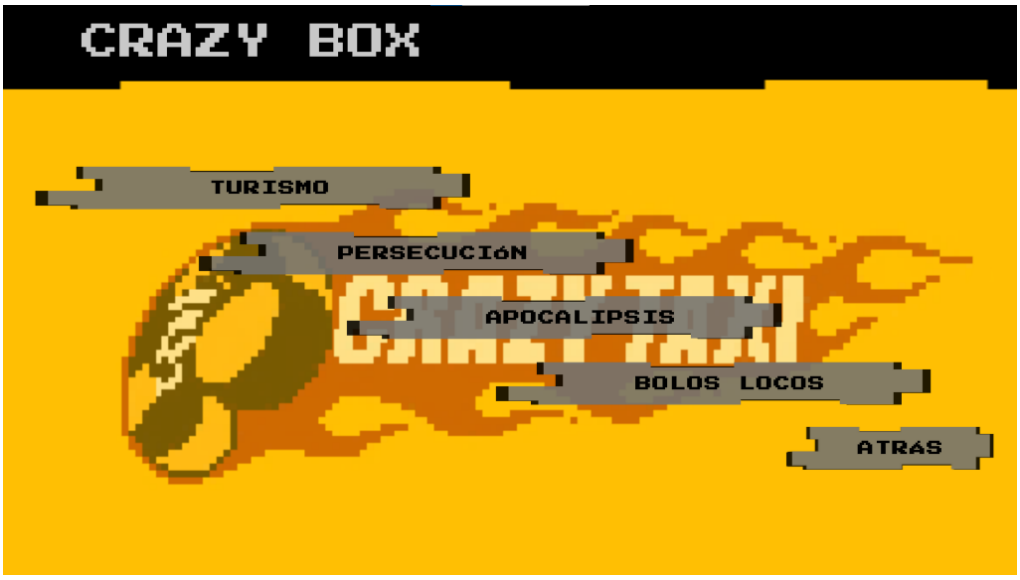

Ilustración 4.98: Menú crazy box

Pulsando cualquiera de los modos propuestos configuraría el modo dentro de *ConfiguracionJuego* y nos lleva a la selección de personaje.

4.9.5.4 Menú de ajustes

Este menú aparece cuando se pulsa el botón de ajustes mediante un oscurecimiento de pantalla y el sonido de aceleración del motor. El aspecto de este menú podemos verlo en la Ilustración 4.99.

Ilustración 4.99: Menú de configuración

En cuanto a apariencia, recalcaremos su fondo de engranajes, el cual está en movimiento. Para recrear el movimiento lo que se ha hecho es tomar un gif y separar sus fotogramas, para cambiar el material de fondo con ellos cada varios frames. Sumado a esto encontramos un detalle implementado, y es que cuando se pulsa el botón aceptar conseguimos el efecto de aceleración de los engranajes porque le reducimos el número de frames que debe esperarse para cambiar el material de fondo con el siguiente fotograma.

En cuanto a sonidos, el menú tiene asociado otra música de fondo que se repite en bucle y emite el sonido de aceleración del motor cuando se pulsa el botón de aceptar.

Si nos centramos más en las funcionalidades del menú, éste nos permite configurar varios parámetros mediante múltiples sliders [64] y dropdowns [65]. Concretamente se puede ajustar:

- ***Idioma***: variable que luego el gestor de idiomas utiliza para traducir y buscar por claves.
- ***Dificultad***: es una variable que influye en el número de vehículos, clientes, porcentaje de vida de jugador y la probabilidad de que la policía persiga al jugador tras intentar atropellar a un NPC.
- ***Gráficos***: aquí se modifica una variable que ajusta la calidad visual del juego mediante la función *QualitySettings.SetQualityLevel()*. Aunque esto se ajuste automáticamente, en el nivel más bajo de gráficos se ha programado para que también cambie los modelos por unos bajos en polígonos, vistos anteriormente en la Ilustración 4.26, aumentando aún más la eficiencia del juego a cambio del aspecto visual.
- ***Música***: indica el porcentaje de volumen de la música de fondo.
- ***Sonido***: indica el porcentaje de volumen de los sonidos.
- ***Elevación del terreno***: cuanto mayor sea este valor, mayores serán las cuestas de la ciudad y por tanto más pronunciada la topología.
- ***Densidad de calles***: provoca una mayor generación de semillas iniciales que repercuten en una mayor generación de calles durante el proceso visto en el apartado 4.2 (Ciudad con generación procedural).
- **Dimensión de ciudad**: cuanto mayor sea el parámetro, mayor será el tamaño de la ciudad (puede conllevar un poco más de tiempo de generación de mapa).

4.9.5.4.1 ConfiguracionJuego

Toda esta configuración se va guardando en un script llamado *ConfiguraciónJuego*, que además de estos parámetros modificables por el usuario, tiene otra serie de variables asociadas y seleccionadas automáticamente de forma privada en función del modo de juego que se elige. Por ejemplo, si jugamos al modo de bolos, activa un parámetro que posteriormente es leído por el *Gestor del juego*, de modo que este último pide al algoritmo de generación de ciudades que incluya en la generación al objeto bolo. También dentro de la configuración se guarda el conductor y el vehículo elegido dentro del menú de selección de personaje (apartado 4.9.5.5) y

del menú de selección de vehículo (apartado 4.9.5.6), para así mismo ser leída dicha información por el gestor del juego al generar la partida.

Un punto interesante es que los valores de la configuración modificados desde el menú de configuración por el propio usuario son guardados dentro de los llamados *PlayerPrefs* [66] de Unity, que de forma resumida son clases de Unity que permiten guardar datos entre diferentes sesiones, por lo que podemos salir del juego cerrando la aplicación y no perder la configuración al volver a arrancar el juego.

El objeto *ConfiguraciónJuego* tiene que ser único y es muy importante que ello sea así, por lo que para evitar incongruencias y asegurar su unicidad se aplicará el patrón Singleton [37], que se desarrollará en detalle en el apartado 4.10.3 (Patrón Singleton aplicado a la configuración).

4.9.5.5 Menú de selección de personaje

Este menú y el Menú de selección de vehículo son un tanto especiales, pues mezclamos en ellos objetos del Canvas [67] bidimensionales y objetos del mundo tridimensional del mundo, coordinados entre sí para dar los resultados de la Ilustración 4.100.

Ilustración 4.100: Menú de selección de personaje

Este menú recordamos que ha sido inspirado en el menú de los míticos juegos de la saga de Mario Kart. En él vemos una pantalla en la que hay un panel con todos los conductores seleccionables y de fondo una ciudad.

En primera instancia, el fondo se trata de una ciudad pequeña, generada con el algoritmo del apartado 4.2 (Ciudad con generación procedural), que ocupa el espacio justo para proporcionar el fondo que vemos.

Centrándonos en el panel, vemos una serie de imágenes de los conductores seleccionables. Dichas imágenes son en verdad botones que tras ser pulsados cambian el conductor tanto internamente como externamente. Internamente se indica al objeto *ConfiguracionJuego* cuál es el nuevo conductor, mientras que externamente y cara al usuario lo que se lleva a cabo es una retroalimentación donde se ve aparecer el personaje que se selecciona. Cabe mencionar que los conductores mostrados durante este menú no son los mismos prefabs que luego se usarán durante el juego, sino que el que se muestra aquí es muy simple y solo tiene una animación (cada uno la suya propia) que repite cíclicamente.

Hablando aun de botones tenemos otros dos botones con los que podremos salir al menú principal o proceder con la selección e ir a ver el listado de vehículos. Éstos cuando son activados, al igual que los botones de los personajes, avisan a un objeto que refresca el estado del menú y que es el encargado de la gestión.

Por otro lado, tenemos una descripción en la zona superior donde el usuario puede ver qué actividad se está llevando a cabo y en la parte inferior si nos fijamos aparece el detalle del nombre que tiene el conductor seleccionado.

Finalmente, si salimos al menú principal pulsando el botón de atrás, la descripción se estirará como ya pasaba en anteriores apartados haciendo una animación que pasa a oscurecer la pantalla (se recomienda probar para ver). Si por el contrario le damos al botón seleccionar, se desplazará hasta esconderse el conductor de muestra, desaparece el panel de personajes, y da paso a la aparición del menú para elegir vehículo.

4.9.5.6 Menú de selección de vehículo

Desde este menú el usuario puede elegir el taxi, ver cómo queda con el conductor seleccionado subido al mismo, y dar inicio a la partida del modo de juego seleccionado.

Como vemos en la Ilustración 4.101 hay muchas similitudes con el menú de selección de personajes como la barra descriptiva superior (que cambia su texto mediante una animación en la que se esconde con un deslizamiento y vuelve a aparecer con el texto cambiado) y en la zona inferior encontramos el nombre del vehículo. También se conserva la ciudad generada, pues no hemos cambiado de escena y 2 botones para volver hacia atrás y para jugar.

Las novedades que encontramos son que ahora aparece un nuevo panel a la derecha donde se muestran las características del vehículo y 2 botones con forma de flecha, para pasar de un vehículo a otro. De la misma manera que con los personajes, cambiar el vehículo implica un cambio interno en la configuración del juego y a nivel externo hay una retroalimentación donde se muestra el vehículo (que también es un vehículo simplificado y no es el que en verdad se usa durante el juego).

Un punto interesante son las barras que indican la vida, velocidad, aceleración y peso. Éstas funcionan igual que las barras vistas en el apartado 4.9.5.7.4 (Barra de vida), igualmente heredando de la clase *ProgressBar* del asset descargado, teniendo como diferencia más notable el diseño.

Ilustración 4.101: Menú de selección de vehículo

Tal y como pasaba con la selección de personajes, si se vuelve hacia atrás, los botones y el panel desplazarán, se cambiará la descripción y aparecerán todos los elementos del menú anterior. Si le damos al botón de jugar iremos a la escena de juego que se configurará según los valores ajustados en el objeto *ConfiguracionJuego*.

4.9.5.7 Interfaz de modo de juego

Esta interfaz es configurada en función de la configuración con la que inicia el modo de juego gracias al *Gestor del juego*. En ella podemos encontrar varios objetos que nos dan información del transcurso de la partida.

Ilustración 4.102: Interfaz durante una partida

4.9.5.7.1 Minimapa

Durante la escena donde se lleva a cabo la partida encontramos como parte de la interfaz un elemento no diegético, el minimapa, siendo otra inspiración de GTA, cuyo objetivo es mostrar al jugador donde se encuentran los enemigos y cuál es la distribución de calles en la zona donde se encuentra.

Ilustración 4.103: Minimapa

Para crearlo se tomó de base un asset de la Asset Store de Unity. Este asset básicamente crea un plano y le asigna una textura que muestra lo mismo que una segunda cámara que graba desde arriba al jugador, luego se le pasa una máscara para hacer transparentes las esquinas y dar la apariencia de círculo.

Ya conocido el funcionamiento que traía el asset, para este proyecto se incluyeron con los siguientes retoques:

- Se ajustó la forma de escalado cuando se cambiaban las dimensiones de la pantalla para que éste no provocara la deformación del mapa con forma circular y lo convirtiera en un óvalo.
- Se colocó una imagen de brújula detrás del círculo relleno con el mapa para mejorar su apariencia.
- Se aplicó un paquete de post-procesamiento de Unity sobre el mapa para conseguir el efecto de deformación de lente y el efecto de viñeta. El efecto de lente simula una lupa y es por ello que vemos agrandar lo que se encuentra en el centro y menguar lo que está en los extremos del minimapa. Por otro lado, el efecto de viñeta lo utilizamos de forma conjunta para oscurecer los bordes del círculo, lo que hace que se vea más volumétrico.

- Se configuraron las máscaras (*Culling Masks* del componente *Camera*) de la cámara del minimapa para que ésta no renderizara los vehículos ni los peatones, pues no aportarían ayuda al jugador y provocan una ralentización del sistema, mostrando únicamente la ciudad y los indicadores de interés.

  Los indicadores de minimapa son objetos con forma triangular o redonda y de colores creados desde Blender que se colocan y acompañan a los vehículos del jugador, vehículos enemigos, triángulos de bolos y destino de clientes. Estos elementos no se pueden ver desde la cámara principal del juego porque se ha restringido en su máscara para que no se vean, siendo visibles por tanto solo desde el minimapa.

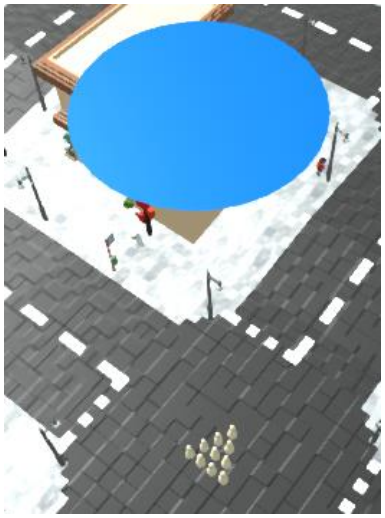

Ilustración 4.104: Indicador de minimapa

4.9.5.7.2 Panel de tiempo restante

El cronometro regresivo que vemos en la partida es otro elemento no diegético que muestra el tiempo restante que queda de ésta, tal y como se hace en Crazy Taxi, pero con nuestro estilo de píxeles y un fondo diferente. El valor del tiempo siempre baja, pero también puede ampliarse si así se le indica.

Ilustración 4.105: Panel de tiempo de partida

Este cronometro es posible gracias a la creación del componente *Temporizador*, el cual tiene como parámetros modificables desde el inspector un objeto con el componente de texto *TextMeshPro* y 3 colores, uno que se muestra cuando queda más de 2/3 del tiempo inicial, otro que aparece cuando queda entre 2/3 y 1/3 del tiempo, y otro cuando queda menos de 1/3 del tiempo. Añadido a esto hay un último parámetro booleano llamado *mostrarEnSegundos*, que sirve para indicar si queremos que se muestre únicamente en segundos, o si queremos que aparezcan los minutos también. En la imagen inferior vemos la configuración que tiene el temporizador general de la partida.

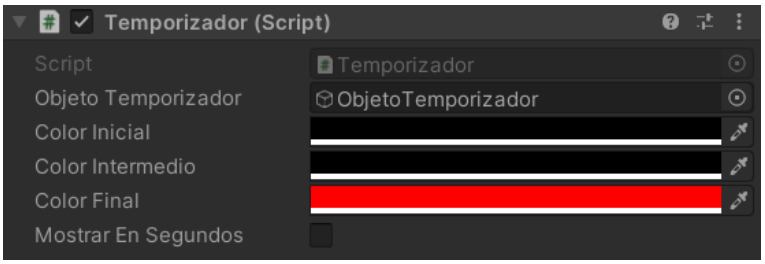

Ilustración 4.106: Parámetros de objeto Temporizador

Este cronometro es observado por el objeto Partida que incorpora el jugador y cuando detecta que el tiempo ha terminado pide la finalización de la partida.

4.9.5.7.3 Panel de dinero conseguido

Se trata de otro objeto no diegético con un texto asociado al que se le ha puesto una imagen de fondo y un script, *MarcadorPuntuacion*. Gracias a este script el componente *Partida* del jugador le indica cuánto dinero tiene que mostrar, y éste se encarga de subir o bajar la cantidad.

Ilustración 4.107: Panel de puntuación

Por su parte, la cantidad de dinero del marcador asciende mediante múltiples incrementos hasta alcanzar el valor demandado por la *Partida*, proceso que dura 1

segundo de animación. La cantidad finalmente se muestra separándose cada 3 cifras por un punto y colocando al final un símbolo de dólar.

4.9.5.7.4 Barra de vida

Este objeto no diegético hace uso de un asset que he descargado de la Asset Store de Unity llamado “ProgressBar Pack” y que he ampliado con herencia para mostrar el valor de vida del jugador, información que se toma del objeto *CaracteristicasJugador* perteneciente al jugador.

En resumidas cuentas, ésta muestra una barra de color que disminuye su relleno conforme menos porcentaje se le envía a su función *UpdateValue()*. Visualmente tiene un fondo similar al del cronómetro y al del panel de dinero, a lo que debemos sumarle el detalle del color interno. Dicho color de relleno cambia cuando el porcentaje es inferior al 20%, mostrando la barra de color rojo en lugar del color verde inicial.

Ilustración 4.108: Barra de vida al 100%

Ilustración 4.109: Barra de vida al 20%

4.9.5.7.5 PopUps

Son objetos (también elementos no diegéticos) con textos informativos que aparecen en pantalla con objetivo de avisar al jugador de algún evento, simulando los mensajes que muestra el juego Crazy Taxi original en pantalla. Es utilizado para mostrar anuncios específicos como “¡VAMOS!" al empezar y “¡YA VIENEN!” cuando comienza a perseguirnos la policía. Se ha conseguido gracias a un script que incorporan, el cual instancia el texto en pantalla con una escala que comienza con valor 0 y termina creciendo hasta el valor 1. Una vez está en la escala 1 permanece un pequeño periodo de tiempo y se desvanece gracias al material, al que se le aplica cada vez más transparencia. Una vez el texto es invisible se destruye el objeto. En la Ilustración 4.110 puede observarse el efecto del mensaje de aviso que aparece

cuando se intenta atropellar a muchos peatones y comienza a perseguir la policía al jugador.

Ilustración 4.110: Popup avisando por policía

4.9.5.7.6 Flecha indicadora

Es un objeto no diegético que indica al jugador la dirección en la que tiene que girar para ir al destino marcado por el cliente haciendo uso del camino más corto devuelto por el GPS. Se ubica junto al objeto cámara del jugador.

Hace uso las funciones del GPS asociado al jugador para controlar el giro de la flecha cuando sube a un cliente al taxi y se mantiene dirigiendo la ruta hasta que se alcanza el destino. La rotación de la flecha se lleva a cabo un poco antes que en el lugar que hay que girar para que el jugador tenga un pequeño tiempo de reacción.

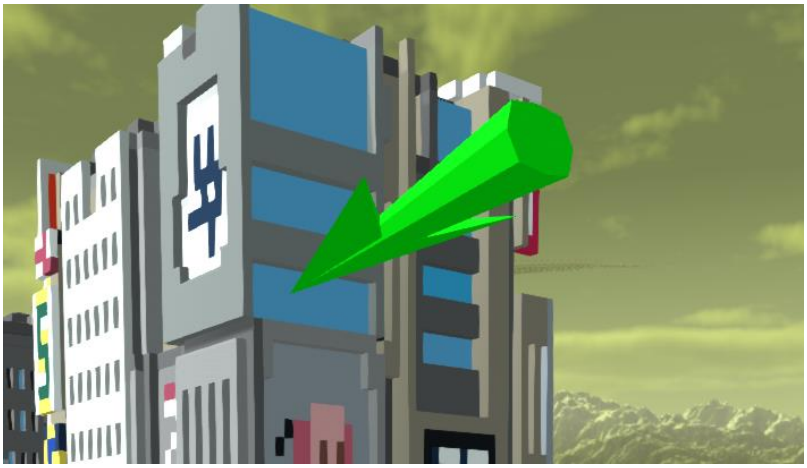

Ilustración 4.111: Flecha indicadora

4.9.5.8 Menú de pausa

Este menú aparece cuando se pulsa el botón asignado para la pausa. Tiene una apariencia simple pero elegante tal y como puede verse en Ilustración 4.112. Nuevamente, para distanciar la lógica de la lectura de entradas por teclado se aplicará

el patrón MVC, desarrollado en el apartado 4.10.1.4 (Aplicación de MVC al evento de pausa).

Para la creación del menú durante la partida lo que se hace es mostrar un plano negro con un porcentaje de transparencia concreto y se detiene el tiempo con el parámetro *Time.scale* para dejar aparecer 3 botones: reanudar, reiniciar y abandonar, que al pulsarlos emiten un sonido.

- Reanudar simplemente quita los botones y el plano semitransparente.
- El botón de reiniciar carga la escena de nuevo.
- El botón de abandonar carga la escena del menú principal.

Además, a los botones de reiniciar y de abandonar les sigue una pequeña transición en la que se oscurece la pantalla y aumenta al mismo tiempo el porcentaje de opacidad tanto del fondo semitransparente como el de las letras.

Ilustración 4.112: Menú de pausa

Siempre que aparece este menú se muestra un cursor para poder seleccionar la opción.

4.9.5.9 Menú de fin del juego

Como último menú desarrollado encontramos el menú de fin del juego. Al acabar el juego se presenta un mensaje con funcionamiento parecido al de los PopUps

donde aparece el mensaje de fin de partida y se desvela menú de fin del juego. En este momento, el gestor del final de la partida manda a mostrar un tablón de resultados con información acerca de la partida jugada. El tablón se muestra con una animación en 2D en la que se ve un taxi pixelado remolcando el cartel hasta que éste cubre toda la pantalla.

Ilustración 4.113: Animación de aparición del menú de fin de partida

La información mostrada por el cartel es por un lado aquella recogida del propio objeto *ConfiguracionJuego*: una imagen del conductor, el nombre del conductor, el modo de juego, la dificultad y nombre de vehículo. Además de esta información se muestra el total de dinero recaudado, información perteneciente al objeto *Partida* que pertenece al jugador.

Ilustración 4.114: Menú de fin de partida

Finalmente tenemos un botón, que al apretarlo nos envía al menú principal del juego. Como es de esperar, junto con la aparición de este menú también debe mostrarse el cursor.

4.9.5.10 Pruebas

A continuación, se van a mostrar los resultados de los test realizados para los menús e interfaz.

T-01 Test de menús intuitivos	
Objetivo	Los menús deben ser intuitivos y no deben requerir de explicaciones para su uso
Resultado	Los menús son intuitivos, tienen una interfaz amigable y que resulta conocida por su parecido con las de otros juegos.
Observaciones	Ningún usuario probador tuvo dudas en ningún momento acerca de lo que hacía cada botón de la interfaz

Tabla 4.46: Test de menús intuitivos

T-02 Test de menús retroalimentados	
Objetivo	Los menús deberían conseguir una retroalimentación que haga saber al usuario que debe realizar una acción
Resultado	Los menús generan sonidos y animaciones cada vez que se interactúa con ellos a través del cursor y pulsaciones de botones
Observaciones	

Tabla 4.47: Test de menús retroalimentados

T-03 Test de menús navegables	
Objetivo	Los menús deben permitir la navegación entre las diferentes acciones que proponen con acciones como la de volver hacia atrás
Resultado	Los menús permiten la navegación de forma cómoda entre unas secciones y otras, mostrando en todo momento qué acción debe realizarse en la zona superior para que el usuario no se pierda
Observaciones	

Tabla 4.48: Test de menús navegables

T-04 Test de interfaz principal	
Objetivo	La interfaz debe ser capaz de dar información acerca de la partida al jugador
Resultado	La interfaz muestra varios datos sin estresar al usuario, tales como la vida, un minimapa, dinero acumulado…
Observaciones	

Tabla 4.49: Test de interfaz principal

4.10 Arquitectura y patrones de diseño

En esta sección se encuentran recopilados todos los patrones de diseño utilizados durante el desarrollo del proyecto.

4.10.1 Patrón Modelo-Vista-Controlador

Se usa para separar los inputs de la aplicación con la lógica del jugador mediante un controlador. Así pues, aunque cambiemos la forma de interacción con el juego (por ejemplo, por un cambio de mandos), los códigos que mantienen la lógica, no se verán afectados. Para los siguientes apartados, entenderemos como la vista todo aquello con lo que el usuario interacciona, como los mandos, el teclado, el ratón, la pantalla, etc.

Se ha aplicado el patrón mediante 4 controladores separados en función de la temática de las entradas que leen.

4.10.1.1 Aplicación de MVC al movimiento del jugador

Separa la lógica del movimiento de las entradas y destacamos:

- **Modelo**: es el script *PrometeoCarController* y lleva a cabo la lógica del movimiento del vehículo del jugador.
- **Controlador**: es el script *ControladorMovimiento* y se ocupa de recibir las entradas y llamar a los métodos del modelo.

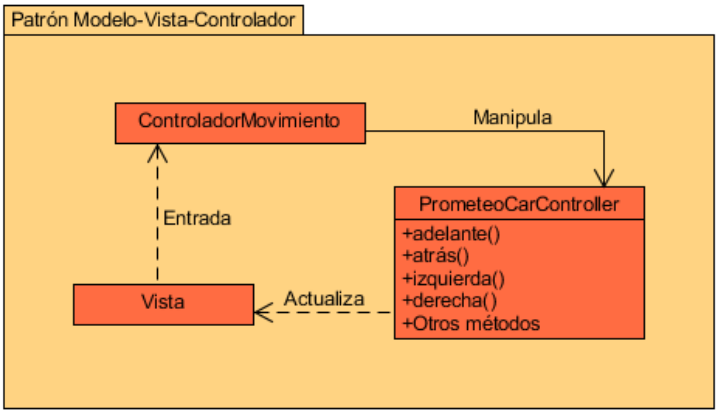

Ilustración 4.115: MVC aplicado al movimiento del jugador

4.10.1.2 Aplicación de MVC a las armas del jugador

Separa la lógica de las armas de las entradas y destacamos:

- **Modelo**: es el script *ArmaJugador* y lleva a cabo la lógica del arma del jugador.
- **Controlador**: es el script *ControladorArmas* y se ocupa de recibir las entradas y llamar a los métodos del modelo.

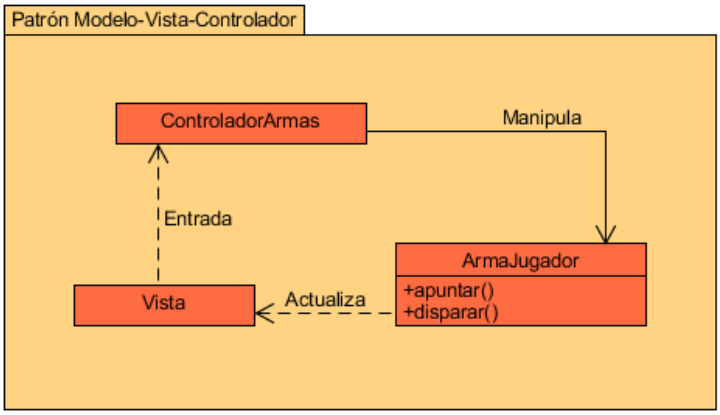

Ilustración 4.116: MVC aplicado a las armas del jugador

4.10.1.3 Aplicación de MVC a la cámara del jugador

Separa la lógica de la cámara de las entradas y destacamos:

- **Modelo**: es el script *CamaraCoche* y lleva a cabo la lógica de la cámara del jugador.
- **Controlador**: es el script *ControladorCamara* y se ocupa de recibir las entradas y llamar a los métodos del modelo.

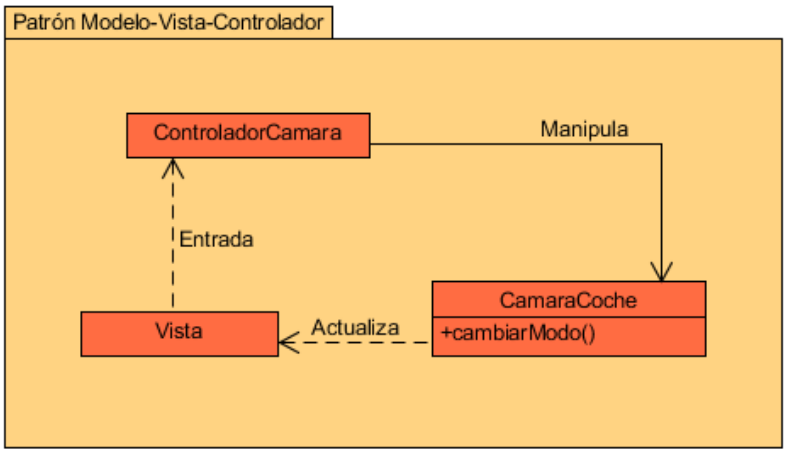

Ilustración 4.117: MVC aplicado a la cámara del jugador

4.10.1.4 Aplicación de MVC al evento de pausa

Separa la lógica de la cámara de las entradas y destacamos:

- **Modelo**: es el script *MenuPausa* y lleva a cabo la lógica del menú de pausa.
- **Controlador**: es el script *ControladorEventos* y se ocupa de recibir las entradas y llamar a los métodos del modelo.

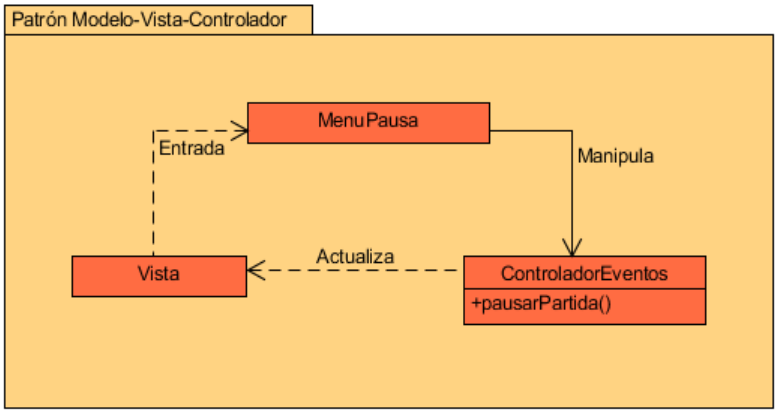

Ilustración 4.118: MVC aplicado al evento de pausa

4.10.2 Patrón Estrategia aplicado al comportamiento de peatones

Los peatones seguirán una estrategia de comportamiento que en tiempo de ejecución se irá alternando. Dicho comportamiento se estructura tal y como se ve en la Ilustración 4.119.

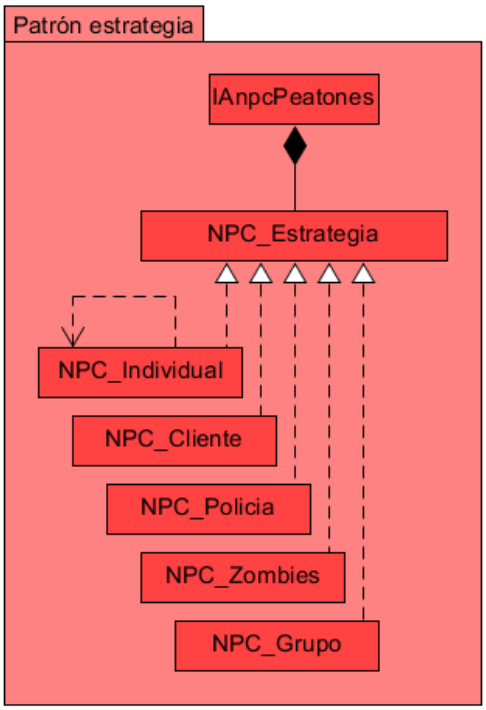

Ilustración 4.119: Patrón estrategia para comportamiento de peatones

Analizándola podemos ver fácilmente las 3 partes principales del patrón, adaptadas de la siguiente manera:

- **Contexto**: será la clase *IAnpcPeatones*, la cual se ocupa de delegar en la jerarquía de estrategias, para configurar un comportamiento concreto. En particular, esta clase añade un componente de comportamiento específico y lo activa a través del método de activación de la interfaz de estrategia.
- **Estrategia**: equivale a la clase *NPC_Estrategia* y por su parte declara una interfaz común para todos los comportamientos comentados, con objetivo de ser usada por *IAnpcPeatones* y así poder activar un comportamiento concreto (estrategia concreta).
- **Estrategia concreta**: en nuestro caso serán 5 posibles, *NPC_Individual*, *NPC_Cliente*, *NPC_Grupo*, *NPC_Policia* y *NPC_Zombies*. Estas clases implementan el comportamiento específico haciendo uso de la interfaz *NPC_Estrategia*. Una vez añadidas al objeto en tiempo de ejecución e iniciadas repiten el comportamiento en bucle hasta que se sustituye el componente por otro siguiendo la estrategia.

4.10.3 Patrón Singleton aplicado a la configuración

La configuración es única y por tanto, buscando el objetivo de tener una única instancia del objeto *ConfiguracionJuego*, acabamos llegando a este patrón, cuyo punto fuerte es su vaga inicialización (Lazy initialitation) que genera una instancia del objeto si se pide y no la hay, y en caso contrario devuelve la única que hay.

Cabe destacar que su uso abusivo en entornos con concurrencia es un error común que debe de evitarse, porque su uso provoca esperas por el acceso al recurso, sin embargo, para usos como el que le daremos es aceptable. El diseño llevado a cabo puede verse en la Ilustración 4.120.

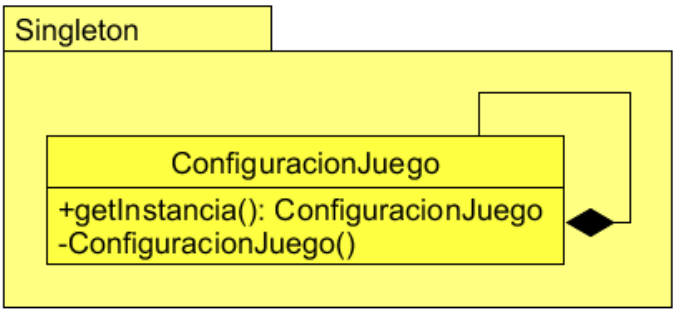

Ilustración 4.120: Patrón singleton para configuración

4.10.4 Patrón Observador aplicado a los menús

El patrón observador se utiliza cuando se necesita que una serie de objetos sean notificados cuando otro objeto cambia su estado. Es un patrón muy interesante que define principalmente los siguientes elementos:

- **Sujeto**: proporciona una interfaz para añadir y eliminar observadores. Estos observadores son guardados en una estructura.
- **Observador**: interfaz que define el método que usa el sujeto para notificar del cambio de estado.
- **Sujeto concreto**: mantiene el estado de interés para los observadores concretos, a los cuales notifica cuando cambia su estado.
- **Observador concreto**: mantiene una referencia al sujeto concreto y son actualizados según el contenido de su método de notificación.

El patrón observador se lleva a cabo de manera no purista dentro de todos los menús del juego, de manera que, aunque no se crean interfaces de sujeto y observador como tal, sí que existen sujetos concretos (botones) y observadores concretos (menús). Es por tanto que, al interactuar con un botón, éste cambia su estado y notifica al menú acerca del suceso, con objetivo de que el menú actualice la interfaz.

4.10.5 Patrón Peso ligero aplicado en los modelos

Se trata de un patrón ya definido por Unity y que se aconseja usar para mejorar el rendimiento del juego. Éste se ocupa de optimizar el uso de memoria reduciendo la redundancia cuando se tiene gran cantidad de objetos con información idéntica.

Se aplica haciendo uso de los llamados prefabs de Unity, elementos que se han utilizado para todos los objetos encontrados dentro del juego.

Ilustración 4.121: Patrón peso ligero mediante prefabs

5 RESULTADOS

5.1 Visión general

Crazy Taxi (reboot) ha llegado a convertirse en un juego casual, basado en el universo de Crazy Taxi, para un único jugador y orientado completamente al ocio. Se puede categorizar como un juego de tipo arcade, pero ciertamente vemos que va más allá y podríamos caracterizarlo incluso como un simulador de taxi, al que podemos añadir unas pinceladas de ideas de videojuegos shooter.

Característica	Clasificación
Título	Crazy Taxi
Género	Arcade/Simulador de taxi
Plataforma	PC
Idioma	Español, inglés, francés y portugués
Clasificación	PEGI 7

Tabla 5.1: Descripción genérica

Teniendo en cuenta el contexto, el juego ha sido creado para la plataforma PC, sin embargo, es importante recordar que el motor utilizado permite exportar el proyecto a otras plataformas, por lo que, aunque la aplicación sea para PC, fácilmente se podría ejecutar en otras plataformas siempre y cuando el rendimiento lo permita.

Como toda aplicación de calidad que busca una correcta internacionalización, nuestro videojuego está preparado completamente para ser traducido a gran cantidad de idiomas originarios de Europa occidental. Inicialmente el juego tiene incorporadas las traducciones en español, inglés, francés y portugués, aunque mediante un .csv modificable desde una hoja de cálculo se pueden sumar nuevos idiomas siempre que sean soportados por la norma de codificación ISO 8859-1 [68].

Finalmente este juego estará dirigido a un público con edad superior a los 7 años, catalogado como tal con un PEGI 7 [69], pues pese a que en el juego no encontremos sonidos ni imágenes que puedan asustar a los niños, sí que se da una suave violencia.

5.2 Modalidades de juego

A continuación vamos a describir cómo son los modos de juego que encontramos al arrancar la aplicación, frutos de todo el desarrollo y la organización llevada a cabo por el gestor del juego a la orden del objeto de configuración. En cada modo repasaremos qué es lo que se puede hacer: objetivos, ambiente, criterios para sumar puntuación y criterios de fracaso... Además, se describirá brevemente la ambientación elegida, algo que es simple de recrear y muy importante a nivel visual, pues una buena ambientación es capaz de transmitir diferentes sensaciones al jugador.

5.2.1 Arcade: modo reglas arcade

Es un modo perteneciente al juego original donde la partida dura 1 minuto inicialmente y ese tiempo es ampliado cuando se consigue llevar a una persona a su destino. El objetivo del juego es conseguir dinero llevando gente de la calle hasta su destino.

La partida termina cuando se pierde toda la vida o todo el tiempo, es por ello que están presentes el panel de tiempo y la barra de vida, además del panel de puntuación, que sube cuando conseguimos un bonus o llevar a un cliente a su destino.

La dificultad seleccionada repercute directamente en el número de vehículos, clientes y cantidad de vida del taxi. La policía no se ha incluido dentro de este modo de juego para así respetar lo máximo posible la esencia del juego arcade del Crazy Taxi original.

Como vemos en la Ilustración 5.1 el nivel tiene una ambientación con luz blanca, ligera niebla también blanca y skybox con montañas.

Ilustración 5.1: Modo de reglas arcade

5.2.2 Arcade: modos de trabajo

Es un conjunto de modos que normalmente aparecen en las entregas de la saga y permite partidas de 3 minutos, 5 minutos o 10 minutos en función del tiempo seleccionado. En este caso el tiempo no es ampliable. El objetivo del juego es conseguir dinero llevando gente de la calle hasta su destino.

Igualmente, la partida termina cuando se pierde toda la vida o todo el tiempo, y el dinero se consigue cuando conseguimos un bonus o llevar a un cliente a su destino.

La dificultad seleccionada repercute directamente en el número de vehículos, clientes, cantidad de vida del taxi y también lo hace con la probabilidad de aparición de policía tras asustar a la gente de la calle (probabilidad que se vuelve nula en dificultad fácil). Si asustando peatones se consigue que la policía aparezca, ésta no desaparecerá y se mantendrá durante el resto de la partida persiguiendo al jugador para bajarle vida.

Encontramos una ambientación con luz blanca, ligera niebla blanca y skybox azulado con planetas.

Ilustración 5.2: Modo de trabajo

5.2.3 Crazy Box: modo de turismo

Es un modo pensado para aprender a jugar o simplemente dar vueltas por la ciudad encontrado en la sección Crazy Box. Como tal no es un modo que encontremos en el juego original y tampoco contiene objetivos para ganar o perder. No hay vida, puntuación ni tiempo y ni siquiera están los paneles que muestran dicha información. Tampoco puede perseguirnos la policía y no hay clientes. Tiene la misma ambientación que encontramos en el modo de arcade.

Ilustración 5.3: Modo de turismo

5.2.4 Crazy Box: modo persecución

No perteneciente a los juegos originales encontramos el modo de persecución dentro de la sección de Crazy Box del menú, este modo es completamente nuevo y en él se aprovecha al máximo los vehículos persecutores tales como los vehículos de policía y tanques.

El juego dura 5 minutos y el tiempo no es ampliable. El objetivo sigue siendo recoger y llevar gente a su destino, solo que ahora conforme la partida comienza ya nos persigue la policía, a lo que se van sumando nuevos vehículos de policía y tanques por cada llamada a la policía por parte de los NPC (hecha cuando son asustados) creciendo muy rápidamente el número de enemigos y convirtiéndolo así en un modo desafiante y complicado.

Dada la situación se han incluido 2 armas al vehículo, una a cada lado para poder disparar misiles explosivos.

En este caso se da una ambientación con luz roja, niebla roja oscura y skybox de cielo rojizo, con objetivo de recordar un ambiente de guerra.

Ilustración 5.4: Modo de persecución

5.2.5 Crazy Box: modo apocalipsis

Es uno de los modos favoritos del proyecto, tratándose de otro modo no perteneciente al juego original con el que se exprimen las funcionalidades de los tanques y el comportamiento de los zombis.

El juego dura 5 minutos y el tiempo no es ampliable. El objetivo es nuevamente recoger gente y llevarla a su destino, aunque esta vez lo haremos en una ciudad apocalíptica llena de zombis que van contagiando a todas las personas que encuentran en su paso.

La dificultad modifica el número de vehículos, de clientes y vida, pero no la probabilidad de que la policía nos persiga, porque en este modo no lo hará. Sin embargo, sí que habrá tanques, pero su objetivo no seremos nosotros, sino los zombis, pues buscan exterminar a la plaga.

Igual que en el Crazy Box: modo persecución el jugador tiene armas en el vehículo. La dificultad aumenta debido a la densa niebla y al entorno lleno de tanques disparando a zombis.

Ilustración 5.5: Modo apocalipsis

Cuenta con una ambientación con luz verde, densa niebla verde oscura y skybox nocturno con cielo estrellado en tonos verdosos con objetivo de recordar un apocalipsis zombi.

5.2.6 Crazy Box: modo de bolos

Este modo sí que es un modo del juego original que encontramos en algunas entregas de la saga. En esta modalidad el jugador tiene como objetivo conseguir dinero tirando bolos que hay desperdigados por la ciudad.

Tiene una duración de 2 minutos, y la ambientación es de niebla leve y blanca, skybox de montañas, sin policía, sin vida y sin vehículos NPC.

Ilustración 5.6: Modo de bolos

6 CONCLUSIONES Y TRABAJOS FUTUROS

A lo largo de este ambicioso proyecto he conseguido hacer un reboot del videojuego Crazy Taxi dentro de un nuevo universo que mezcla los videojuegos Crazy Taxi, Minecraft y GTA. Para ello he tenido que adentrarme en una gran cantidad de campos y temas de investigación, tales como la generación procedural de ciudades mediante bases algebraicas, creación de NPCs con gestión y comportamiento inteligente, entidades de persecución optimizadas para funcionamiento en tiempo real, diseño 3D y animación, diseño de interfaces intuitivas y retroalimentadas, gestión de la traducción dentro de una aplicación internacionalizada, patrones de diseño, etc.

Gracias al TFG podemos asegurar que he adquirido una gran cantidad de conocimientos en el ámbito de videojuegos, desarrollo software y gestión de proyectos. Sumado a esto, no solo he afianzado una gran cantidad de saberes de la carrera y profundizado en muchos otros más novedosos, sino que he sido capaz de orientarlos de manera práctica y creativa, dando un fruto merecido a las horas de investigación dedicadas al trabajo. En resumidas cuentas, he madurado, al menos desde el punto de vista personal, en el ámbito de la ingeniería.

Una vez realizado el proyecto además he aprendido el trabajo que conlleva hacer un videojuego de forma individual, las horas que conlleva y la cantidad de profesionales que se ven involucrados de diferentes ámbitos, tales como analistas, programadores, artistas, músicos, guionistas, traductores, etc.

Para terminar, he de mencionar que este proyecto desde un principio se eligió con muchas ganas e ilusión, y es por ello que el esfuerzo se ha visto multiplicado. Dicho esmero cultivado con años de trabajo y esfuerzo finaliza con unos resultados que, frente a las expectativas iniciales, han sido increíbles y sobrepasan las estimaciones calculadas por el tiempo dado de proyecto. Sin embargo, el trabajo aún se puede mejorar en numerosos aspectos que dejaremos pendientes para una próxima vez, tales como:

- Nuevos controles de juego con idea de alcanzar una interacción multimodal con objetivo de crear una nueva experiencia de juego y hacerlo más

accesible e inclusivo para todos. Esto abriría las puertas a controles mediante gestos, lenguaje natural, expresiones faciales, etc.

- Ampliar la generación procedural de la ciudad implementando otras posibilidades de creación basadas en fundamentos diferentes, como por ejemplo el uso de L-Systems [70].
- Mejoras en el funcionamiento de la técnica flocking de los peatones grupales para un movimiento más realista de las animaciones colectivas que puedan llevarnos a resultados mucho más naturales como lo hacen herramientas contemporáneas conocidas por su uso en cine como por ejemplo MASSIVE [71].
- Diseño propio y creación de todos los objetos y animaciones del juego.
- Creación de un sistema incorporado a la aplicación actual para conseguir que ésta sea rentable y pueda generar riqueza de algún modo. Por supuesto esto sería a nivel teórico, y si se deseara hacer realmente habría que obtener todas las licencias y derechos de autor correspondientes, además de un correspondiente estudio legal.
- Inclusión de nuevos modos de juego didácticos, orientados a la educación vial, ética y moral para que nuestro juego no se quede únicamente dentro del sector del ocio y pueda dar a la sociedad algún otro tipo de servicio.
- Adaptación del juego a otros paradigmas de la informática tales como la realidad virtual.
- Inclusión de un sistema multijugador haciendo uso de assets de Unity como Photon Unity Network (PUN 2) [72] o incluso creando un sistema propio.

Como vemos se puede hacer un listado infinito de fantásticas ideas y seguir mejorando más y más nuestro juego, no obstante, por esta vez nos detendremos aquí, y todas estas mejoras, de momento, tendrán que esperar un poco.

7 APÉNDICES

En esta sección se incluirá la documentación adicional necesaria para comprender correctamente el proyecto.

7.1 Guía original del Trabajo Fin de Título

Este trabajo consiste en la realización de un remake de algún videojuego clásico utilizando un motor de videojuegos actual, como Unreal Engine, CryEngine o Unity. El videojuego original, así como el motor a utilizar, quedan a elección del alumnado.

Hay que describir las mecánicas de juego y la experiencia de usuario perseguida, incluyendo el sistema de recompensas, los mecanismos de progreso del usuario, los recursos gráficos y sonoros empleados, etc. Para la elaboración de escenas, personajes, NPCs, objetos, animaciones y todo tipo de material gráfico y sonoro, se podrán utilizar elementos descargables de la red, incluyendo sprites originales o cualquier otro recurso. Se recomienda elegir un videojuego de la década de los 80s y 90s, buscando la sencillez.

7.1.1 Conocimientos previos

Es necesario tener experiencia con el uso de motores de videojuegos.

7.1.2 Objetivos del TFG

- Diseño del remake del videojuego clásico elegido, especificando las características y el sistema/entorno objetivo.
- Descripción de las dinámicas, mecánicas y componentes de juego, que deberían ser las mismas del juego original. Se permite añadir contenido adicional o variaciones.
- Diseño de la interfaz de usuario y los mecanismos de interacción.
- Diseño del entorno virtual 2D/3D, incluyendo todo el material gráfico y sonoro.

- Desarrollo del juego propuesto, incluyendo la programación de los componentes necesarios.
- Realización de pruebas de evaluación del sistema y análisis de resultados.

7.1.3 Metodología a Desarrollar

- Describir el videojuego clásico elegido para realizar el remake. Describir las dinámicas, mecánicas y componentes de juego, lo que incluye narrativa, objetivos, recompensas, etc.
- Realizar un breve estado del arte sobre motores de videojuegos y seleccionar uno de ellos. Justificar las decisiones tomadas.
- Seleccionar y utilizar una metodología basada en Ingeniería del Software para el análisis de requisitos, diseño, implementación, prueba y documentación. Se recomienda utilizar una metodología incremental.
- Detallar la estimación temporal y costes de desarrollo.
- Realizar el desarrollo del remake, incluyendo el modelado del entorno virtual 2D/3D.
- Realizar pruebas para el prototipo y documentar los resultados.
- Redacción de la documentación final requerida.

7.1.4 Documentos y formatos de entrega

- Documentación generada durante el proceso, de acuerdo a las buenas prácticas de la ingeniería del software. Esto se aplicará tanto en los proyectos de Ingeniería como en los trabajos teóricos o experimentales, es decir, siempre que se genere algún tipo de desarrollo, tanto software como webs, scripts, etc.
- Para el formato de documentación de los Proyectos de Ingeniería se utilizará la norma UNE 157801. Tal y como especifican las normas de estilo de la EPSJ: ‘Los Proyectos de Ingeniería deberán presentar una estructura y contenido adaptados a la norma UNE 157001 - Criterios generales para la elaboración de proyectos - u otra derivada de ésta, en función de la naturaleza del proyecto’. ‘…para el caso de los sistemas de información

informatizados (sistemas informáticos, sistemas de información geográfica, etc.) está la norma derivada de la anterior UNE 157801 - Criterios generales para la elaboración de proyectos de sistemas de información'. Para ello, se puede utilizar también la Norma CCII-N2016-02 (Norma Técnica para la realización de la Documentación de Proyectos en Ingeniería Informática) del Consejo General de Colegios Profesionales de Ingeniería en Informática, que incluye e implementa la norma UNE 157801.

- Memoria del trabajo, incluyendo manuales y anexos, en formato PDF.
- Código fuente completo y ejecutables del prototipo o software desarrollado.
- Documentos y archivos adicionales a los que se haga mención expresa en la memoria.
- Vídeo de demostración de la aplicación o de la experimentación realizada.
- Anexos para la presentación del trabajo:
  - Informe favorable del tutor.
  - Autorización de publicación, si procede.

7.1.5 Modificación del TFG

Se formalizó una modificación del proyecto a la escuela con objeto de pasar del proyecto de carácter general con título "Remake de un videojuego clásico" a proyecto a carácter específico con nombre "Reboot del videojuego Crazy Taxi".

7.2 Instalación y configuración del sistema

El videojuego no requiere de instalación si se hace uso de un sistema operativo Windows. Para utilizar y probar el juego solamente hay que acceder al archivo ejecutable (.exe) con nombre "Crazy Taxi" que encontramos en la carpeta "Ejecutable". Lo único que ha de tenerse en cuenta es que la aplicación se encuentra dentro de una carpeta que contiene otros archivos necesarios para la ejecución, por lo que **el ejecutable no funcionará si se saca de la carpeta**.

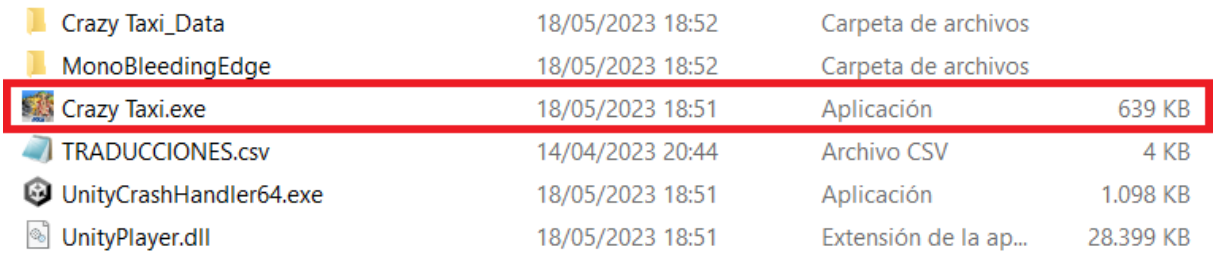

Ilustración 7.1: Ejecutable del juego

Dado que el proyecto fue desarrollado para Windows, si se desea ejecutar en otros sistemas se requerirá del uso de emuladores o máquinas virtuales.

Es importante mencionar que el proyecto fue realizado en la versión de Unity 2021.3.15f1. Por lo que, si se desea acceder y ver el interior del proyecto, habrá que instalar esa versión de Unity para evitar corromperlo. Sea como sea **el proyecto tiene funcionalidades que únicamente se pueden ver funcionar de manera correcta en el ejecutable y no es posible desde dentro de Unity**, por lo que se recomienda que para las pruebas de funcionalidades se utilice el ejecutable.

7.2.1 Configuración recomendada

Al iniciar el juego por defecto la configuración se ajusta con dificultad, gráficos, volumen y parámetros de generación con el valor mínimo. Por experiencia, se recomienda una posible configuración (botón de Ajustes en la aplicación) que para el autor del proyecto es la óptima (recordemos que con gráficos al mínimo no se verá el estilo *voxelart* del juego en los edificios y perderá mucha calidad):

Ilustración 7.2: Configuración recomendada

7.3 Manuales de usuario

En este apartado se verán los controles del juego que también se explican de manera más visual en el manual adjunto (un archivo a parte dentro de la entrega) con el nombre “Manual Visual Crazy Taxi”. En dicho manual también se comentarán, además de los controles, otros aspectos tales como:

- Comparativa de modos de cámara.
- Formas de conseguir dinero.
- Tipos de clientes.
- Presentación de modos de juego.
- Consejos y trucos.

En resumen, aunque no sea necesario el manual visual para comprender el juego una vez leída esta memoria, sí que su lectura facilita el uso de la aplicación. Por otro lado, dado que es un manual que busca simular los libros que traían las cajas de videojuegos, el tono y la forma de expresión variarán comparados con la que encontramos en este documento.

Pasamos a enumerar los controles.

7.3.1 Uso de la interfaz

A continuación, se muestran los controles para el uso de interfaz y menús.

Entrada	Acción
Tecla ESC	Pausar partida
Deslizar puntero del ratón	Mover puntero de juego
Botón izquierdo del ratón	Seleccionar opción

Tabla 7.1: Controles de interfaces y menús

7.3.2 Conducción del vehículo

A continuación, se muestran los controles del vehículo del jugador cuando está dentro de una partida.

Entrada	Acción
Tecla W	Acelerar
Tecla S	Marcha atrás
Tecla A	Girar a la izquierda
Tecla D	Girar a la derecha
Tecla SPACE	Derrapar

Tabla 7.2: Controles del jugador

Para hacer que un cliente se suba o se baje el vehículo debe de estar detenido o a muy baja velocidad.

7.3.3 Manejo de la cámara

A continuación, se muestran los controles de la cámara del jugador cuando éste está dentro de una partida.

Entrada	Acción
Tecla Q	Cambiar cámara
Tecla E (mantener)	Mirar hacia atrás

Tabla 7.3: Controles de cámara

7.3.4 Control de armas

A continuación, se muestran los controles de los cañones laterales del vehículo del jugador cuando éstos se encuentran disponibles en sus modos correspondientes.

Entrada	Acción
Deslizar puntero del ratón	Apuntar
Botón izquierdo del ratón	Disparar

Tabla 7.4: Controles de armas

8 DEFINICIONES Y ABREVIATURAS

Asset: recursos que encontramos en la tienda de Unity y que son utilizados para construir un juego, como modelos 3D, texturas, efectos de sonido, scripts...

Elemento diegético: en el campo de videojuegos, hace referencia a aquellos componentes de la interfaz que se incluyen en el mundo del juego, es decir, pueden ser vistos y escuchados por los personajes del juego.

Elemento no diegético: en el campo de videojuegos, hace referencia a aquellos componentes de la interfaz que se representan fuera del mundo del juego, solo visible y audible para los jugadores en el mundo real.

EEDD: es un término cuyas siglas hacen referencia a Estructuras de Datos.

FIFO: siglas provenientes del término inglés First In First Out, que hace referencia a la gestión de procesos indicando que los primeros que entran al sistema serán también los primeros en ser atendidos.

FPS: es un término que se refiere a los frames por segundo.

GameObjects: son contenedores que guardan las diferentes piezas que son requeridas para hacer un personaje, una luz, un árbol, un sonido… A estos además se les pueden asociar componentes para dotarlos con algún comportamiento.

GTA: es un término cuyas siglas hacen referencia a la saga de videojuegos Grand Theft Auto.

MVC: es un término cuyas siglas hacen referencia al patrón Modelo-Vista-Controlador.

NPC: es un término cuyas siglas vienen de la expresión inglesa *Non Playable Character* y se utiliza para hacer referencia a un personaje no jugador, autocontrolado por el director de juego.

Pixelart: es una forma de arte digital que se caracteriza por utilizar píxeles para crear imágenes de estilo retro o pixelado.

Prefab: en Unity los prefabs son objetos de juego que se pueden crear para luego reutilizar en diferentes lugares del proyecto y que siguen el patrón peso ligero. Es importante su uso porque mejora la eficiencia de los objetos en una aplicación mediante la reducción de su uso de memoria.

Rigging: que es un término utilizado en animación que se refiere a la etapa en la que se ubican huesos en el modelo del personaje para posteriormente animarlo. Es útil porque una vez hecha la animación en base al esqueleto, puede utilizarse la animación en todo modelo que lo contenga.

Voxelart: es una forma de arte digital que utiliza píxeles tridimensionales volumétricos, conocidos como voxels, para crear objetos y escenas. Es similar al arte de píxeles conocido como pixelart.

9 BIBLIOGRAFÍA

[1] "Crazy Taxi - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Crazy_Taxi (accessed May 15, 2023).

[2] "Anexo:Videojuegos más vendidos - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Anexo:Videojuegos_m%C3%A1s_vendidos (accessed May 23, 2023).

[3] "Minecraft - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Minecraft (accessed May 23, 2023).

[4] "Grand Theft Auto - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Grand_Theft_Auto (accessed May 23, 2023).

[5] "¿Qué es el voxel art? | Domestika." https://www.domestika.org/es/blog/5529-que-es-el-voxel-art (accessed May 15, 2023).

[6] J. Lorés, "La interacción persona-ordenador," 2001, Accessed: May 16, 2023. [Online]. Available: http://aipo.griho.net

[7] "Castlevania - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Castlevania (accessed Jun. 15, 2023).

[8] "Wolfenstein - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Wolfenstein (accessed Jun. 15, 2023).

[9] "Tomb Raider (serie) - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Tomb_Raider_(serie) (accessed Jun. 15, 2023).

[10] "Uncharted - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Uncharted (accessed Jun. 15, 2023).

[11] "Sega - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Sega (accessed Jun. 15, 2023).

[12] "Browse Fonts - Google Fonts." https://fonts.google.com/ (accessed May 16, 2023).

[13] "Bienvenidos a Steam." https://store.steampowered.com/?l=spanish (accessed Jun. 15, 2023).

[14] "AAA (industria del videojuego) - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/AAA_(industria_del_videojuego) (accessed May 26, 2023).

[15] “Epic Games - Wikipedia, la enciclopedia libre.” https://es.wikipedia.org/wiki/Epic_Games (accessed Jun. 15, 2023).

[16] “Lumen Global Illumination and Reflections in Unreal Engine | Unreal Engine 5.0 Documentation.” https://docs.unrealengine.com/5.0/en-US/lumen-global-illumination-and-reflections-in-unreal-engine/ (accessed May 29, 2023).

[17] “☑ Ray Tracing: ¿Qué es y para qué sirve? - Definición.” https://www.geeknetic.es/Ray-Tracing/que-es-y-para-que-sirve (accessed May 29, 2023).

[18] “Nanite Virtualized Geometry in Unreal Engine | Unreal Engine 5.0 Documentation.” https://docs.unrealengine.com/5.0/en-US/nanite-virtualized-geometry-in-unreal-engine/ (accessed May 29, 2023).

[19] “Introduction to Blueprints | Unreal Engine 4.27 Documentation.” https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/GettingStarted/ (accessed May 29, 2023).

[20] “blender.org - Home of the Blender project - Free and Open 3D Creation Software.” https://www.blender.org/ (accessed May 15, 2023).

[21] “Unity Asset Store - The Best Assets for Game Making.” https://assetstore.unity.com/ (accessed May 15, 2023).

[22] “Photopea | Online Photo Editor.” https://www.photopea.com/ (accessed May 29, 2023).

[23] “Planes y Precios | Creately.” https://creately.com/es/planes/?ref=home (accessed May 29, 2023).

[24] “Software de hojas de cálculo Microsoft Excel | Microsoft 365.” https://www.microsoft.com/es-es/microsoft-365/excel (accessed May 16, 2023).

[25] K. H. Pries and J. M. Quigley, “Scrum project management,” p. 174, 2011.

[26] “InfoJobs - Bolsa de trabajo, ofertas de empleo.” https://www.infojobs.net/ (accessed Jun. 14, 2023).

[27] “Mario Kart - Wikipedia, la enciclopedia libre.” https://es.wikipedia.org/wiki/Mario_Kart (accessed May 16, 2023).

[28] R. Dechter and J. Pearl, “Generalized best-first search strategies and the optimality of A*,” *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 505–536, Jul. 1985, doi: 10.1145/3828.3830.

[29] “Newsfeed - Sketchfab.” https://sketchfab.com/feed (accessed May 15, 2023).

[30] “Unity - Manual: Prefabs.” https://docs.unity3d.com/es/530/Manual/Prefabs.html (accessed May 15, 2023).

[31] “Unity - Manual: Sistema de Partículas.” https://docs.unity3d.com/es/530/Manual/ParticleSystems.html (accessed May 15, 2023).

[32] “Trail Renderer - Unity Manual.” https://docs.unity3d.com/es/2018.4/Manual/class-TrailRenderer.html (accessed May 15, 2023).

[33] “Wheel Collider - Unity Manual.” https://docs.unity3d.com/es/2018.4/Manual/class-WheelCollider.html (accessed May 15, 2023).

[34] “Rigidbody - Unity Manual.” https://docs.unity3d.com/es/2019.4/Manual/class-Rigidbody.html (accessed May 15, 2023).

[35] “Unity - Scripting API: MeshCollider.” https://docs.unity3d.com/ScriptReference/MeshCollider.html (accessed May 15, 2023).

[36] “Unity - Scripting API: AudioClip.” https://docs.unity3d.com/ScriptReference/AudioClip.html (accessed May 15, 2023).

[37] Robert. Nystrom, “Game programming patterns,” p. 345.

[38] “Unity - Scripting API: Animator.” https://docs.unity3d.com/ScriptReference/Animator.html (accessed May 15, 2023).

[39] K. P. Fishkin and B. A. Barsky, “ANALYSIS AND ALGORITHM FOR FILLING PROPAGATION.,” *Proceedings - Graphics Interface*, pp. 203–212, 1985, doi: 10.1007/978-4-431-68033-8_6/COVER.

[40] J. Francisco. Ruiz Ruiz, “Métodos computacionales en álgebra : matemática discreta: grupos y grafos”.

[41] “Estructuras de datos en Java - Universidad Jaén.” https://buscaenbuja.ujaen.es/discovery/fulldisplay?docid=alma991004201885304994&context=L&vid=34CBUA_UJA:VU1&lang=es&search_scope=CATALOGO&adaptor=Local%20Search%20Engine (accessed May 15, 2023).

[42] M. F. A. Luis Joyanes Aguilar, “Estructura de datos en C,” 2005.

[43] “Box Collider - Unity Manual.” https://docs.unity3d.com/es/2018.4/Manual/class-BoxCollider.html (accessed May 15, 2023).

[44] K. Perlin, “An Image Synthesizer,” vol. 19, no. 3, pp. 22–26, 1985.

[45] “LearnOpenGL - Normal Mapping,” *learnopengl.com*, Accessed: May 15, 2023. [Online]. Available: https://learnopengl.com/Advanced-Lighting/Normal-Mapping

[46] “NormalMap-Online.” http://cpetry.github.io/NormalMap-Online/ (accessed May 15, 2023).

[47] “Construyendo un NavMesh - Unity Manual.” https://docs.unity3d.com/es/2019.4/Manual/nav-BuildingNavMesh.html (accessed May 16, 2023).

[48] “Unity - Scripting API: NavMeshAgent.” https://docs.unity3d.com/ScriptReference/AI.NavMeshAgent.html (accessed May 16, 2023).

[49] “GitHub - Unity-Technologies/NavMeshComponents: High Level API Components for Runtime NavMesh Building.” https://github.com/Unity-Technologies/NavMeshComponents (accessed May 16, 2023).

[50] “Unity - Manual: NavMesh Surface.” https://docs.unity3d.com/560/Documentation/Manual/class-NavMeshSurface.html (accessed May 16, 2023).

[51] “Mixamo.” https://www.mixamo.com/#/ (accessed May 15, 2023).

[52] Rick. Parent, *Computer animation algorithms and techniques*. 2002.

[53] “Humanoid Animation (HAnim) | Web3D Consortium.” https://www.web3d.org/working-groups/humanoid-animation-hanim (accessed May 27, 2023).

[54] “Capsule Collider - Unity Manual.” https://docs.unity3d.com/es/2019.4/Manual/class-CapsuleCollider.html (accessed May 15, 2023).

[55] “Unity - Scripting API: AnimatorController.” https://docs.unity3d.com/ScriptReference/Animations.AnimatorController.html (accessed May 15, 2023).

[56] “Unity - Scripting API: Physics.Raycast.” https://docs.unity3d.com/ScriptReference/Physics.Raycast.html (accessed May 16, 2023).

[57] “Unity - Scripting API: RaycastHit.” https://docs.unity3d.com/ScriptReference/RaycastHit.html (accessed May 16, 2023).

[58] “Unity - Manual: Corrutinas.” https://docs.unity3d.com/es/530/Manual/Coroutines.html (accessed May 16, 2023).

[59] “Skybox - Unity Manual.” https://docs.unity3d.com/es/2018.4/Manual/class-Skybox.html (accessed May 17, 2023).

[60] “Unity - Scripting API: Camera.” https://docs.unity3d.com/ScriptReference/Camera.html (accessed May 16, 2023).

[61] “SMAA vs TAA vs FXAA vs MSAA: explicación de estas tecnologías gráficas.” https://www.guiahardware.es/smaa-vs-taa-vs-fxaa-vs-msaa-explicacion-de-estas-tecnologias-graficas/ (accessed May 16, 2023).

[62] “Unity - Manual: Audio Mixer.” https://docs.unity3d.com/Manual/AudioMixer.html (accessed May 16, 2023).

[63] “Unity - Manual: Input Manager.” https://docs.unity3d.com/Manual/class-InputManager.html (accessed May 15, 2023).

[64] “Unity - Scripting API: UI.Slider.value.” https://docs.unity3d.com/es/530/ScriptReference/UI.Slider-value.html (accessed May 16, 2023).

[65] “Unity - Manual: Dropdown (Desplegable).” https://docs.unity3d.com/es/530/Manual/script-Dropdown.html (accessed May 16, 2023).

[66] “Unity - Scripting API: PlayerPrefs.” https://docs.unity3d.com/ScriptReference/PlayerPrefs.html (accessed Jun. 15, 2023).

[67] “Canvas - Unity Manual.” https://docs.unity3d.com/es/2019.4/Manual/UICanvas.html (accessed May 16, 2023).

[68] “ISO/IEC 8859-1:1998.” https://web.archive.org/web/20060903161324/http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=28245&ICS1=35&ICS2=40&ICS3= (accessed May 25, 2023).

[69] “| Pegi Public Site.” https://pegi.info/es/ (accessed May 15, 2023).

[70] “Sistema-L - Wikipedia, la enciclopedia libre.” https://es.wikipedia.org/wiki/Sistema-L (accessed Jun. 14, 2023).

[71] “MASSIVE (software) - Wikipedia.” https://en.wikipedia.org/wiki/MASSIVE_(software) (accessed Jun. 14, 2023).

[72] “PUN 2 - FREE | Network | Unity Asset Store.” https://assetstore.unity.com/packages/tools/network/pun-2-free-119922 (accessed Jun. 14, 2023).