Análisis, diseño y desarrollo de un videojuego en 2D con perspectiva top-down

Universidad de Las Palmas de Gran Canaria
Escuela de Ingeniería Informática

Trabajo de Fin de Grado

Yraya Bolaños Frasquet

Tutores: Agustín Trujillo Pino y Adrián Rivero Pérez

Las Palmas de Gran Canaria, Julio 2015

Índice

1. Objetivos y aportaciones del proyecto

El objetivo de este proyecto es crear un prototipo jugable de un videojuego en 2D. Mediante la realización del mismo, se pretende familiarizarse con todas las etapas en las que se desglosa el desarrollo de videojuegos, familiarizarse con algunas de las herramientas más empleadas y demandadas en la actualidad y, en general, adquirir destreza a la hora de llevar a cabo un proyecto software (cumplir los plazos establecidos, mantener buenas prácticas como el “good naming”, poca presencia de comentarios, etc.).

1.1. Aportaciones

La realización de este trabajo ha supuesto la adquisición de una gran variedad de nuevos enseñanzas para la alumna así como el afianzamiento de otras ya adquiridas durante el curso de la carrera. En relación a todo el proceso que supone un proyecto, se han mejorado los conocimientos relativos a

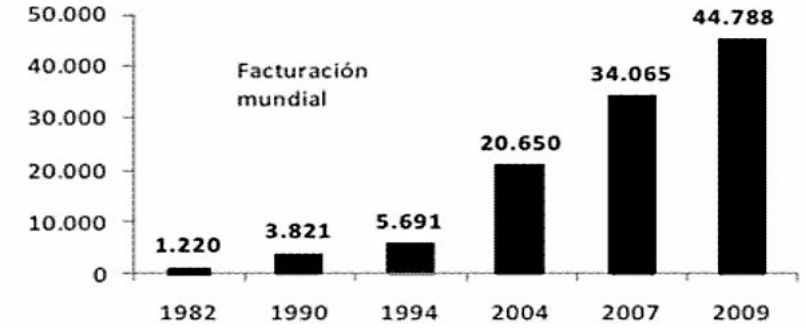
- la planificación
- el análisis
- el diseño
- el desarrollo
- la documentación necesaria

así como aquellos relacionados con proyectos más centrados en el ámbito informático, como los conocimientos relativos a lenguajes de programación.

También se ha ampliado la información que se poseía respecto al mundo de los videojuegos y se ha adquirido nueva relacionada con el análisis, diseño y desarrollo de los proyecto enmarcados en este ámbito en particular. Entre las nuevas capacidades adquiridas destaca el aprendizaje del manejo de nuevas herramientas software útiles para el desarrollo de videojuegos.

En cuanto al entorno social, los videojuegos, con apenas 30 años de historia, tienen un importante potencial cultural y comercial. Podemos considerarla una de las industrias más importantes de entretenimiento. A pesar de que, Inicialmente, los videojuegos eran considerados como una forma de ocio dirigida a niños o para el mundo informático, con el tiempo, y gracias a su diversidad, se han hecho atractivos para personas de todas las edades.

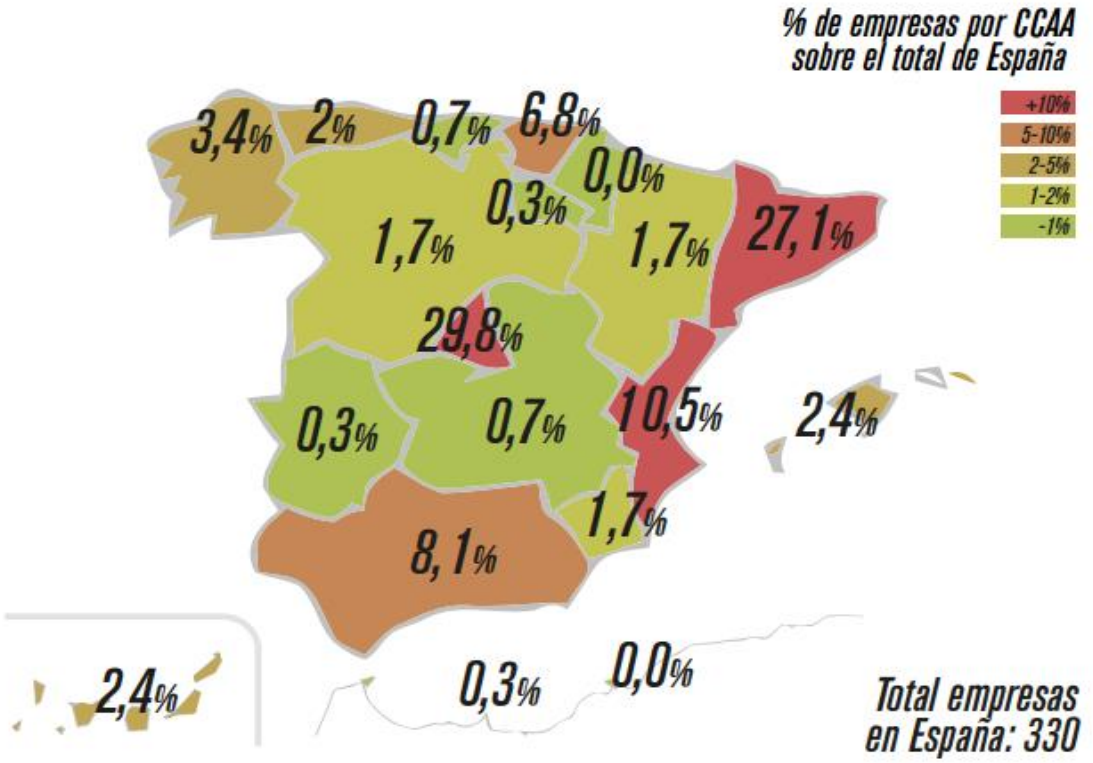
La industria de los videojuegos ha sido un sector en auge desde su aparición y factura millones de euros cada año.



Crecimiento del sector del videojuego a nivel global en millones de euros

Este hecho supone miles de puestos de trabajos relacionados con esta industria y que integran a distintos tipos de profesionales (diseñadores gráficos, diseñadores de sonido, desarrolladores, publicistas, etc.),

Es una industria muy dinámica y que se encuentra en constante proceso de evolución. Este crecimiento se produce a un ritmo mucho mayor que el del resto de sectores y, en lo que al videojuego respecta, España no es una excepción.



Porcentaje empresas de videojuegos por CCAA sobre total de España. Fuente: Libro Blanco del DEV (Asociación Española de Empresas Desarrolladoras de Videojuegos y Software de Entretenimiento)

La industria española del videojuego ofrece excelentes oportunidades de negocio, amplias posibilidades de creación de empleo y está consolidada en cuanto a su condición de industria exportadora. Los videojuegos se mantienen como sector líder en consumo frente a otras industrias de ocio tradicionales como la música o el audiovisual.

Al margen de las aportaciones económicas, los videojuegos pueden suponer importantes beneficios en distintas áreas de la sociedad. Aunque lo más común sea asociarlos con el mundo del ocio, los videojuegos también pueden ser herramientas que faciliten el aprendizaje de distintas materias, las relaciones sociales, capacidades mentales como la lógica y habilidades físicas como los reflejos.

Captura del juego de habilidad Super Hexagon

Cada vez son más los educadores que integran videojuegos didácticos como parte del proceso formativo de los niños. El área de enseñanza en la que más se emplean este tipo de videojuegos es la de las matemáticas.

Los videojuegos favorecen las interacciones entre las personas, incluso entre aquellas que no se conocían antes de utilizar un vídeo juego (videojuegos online). En torno al mundo de los videojuegos se crean blogs, debates en redes sociales, etc. que favorecen la comunicación y la discusión de distintas opiniones.

Así pues, el desarrollo del presente Trabajo de Fin de Grado se enmarca en un entorno con grandes posibilidades laborales, sociales y educativas.

2. Competencias

La realización del presente Trabajo de Fin de Grado implica cubrir las competencias CII01, CII02, CII018 y TFG01, las cuales se detallan a continuación junto con una explicación acerca de cómo han sido cubiertas.

CII01 - Capacidad para diseñar, desarrollar, seleccionar y evaluar aplicaciones y sistemas informáticos, asegurando su fiabilidad, seguridad y calidad, conforme a principios éticos y a la legislación y normativa vigente.

Esta competencia queda cubierta durante las fases de análisis, diseño y desarrollo del proyecto expuestas en los apartados “Análisis y diseño de propuestas”, “Prototipado” e “Implementación de la demo final” de este documento. La alumna también ha tenido que tomar decisiones acerca de la utilización o no de las distintas herramientas software disponibles tal y como se muestra en los subapartados “Elección del motor” y “Elección del lenguaje de programación”. Durante todo el proceso relativo a la realización del proyecto, se respetaron los principios éticos y la legislación y normativa vigente.

CII02 - Capacidad para planificar, concebir, desplegar y dirigir proyectos, servicios y sistemas informáticos en todos los ámbitos, liderando su puesta en marcha y su mejora continua y valorando su impacto económico y social.

Para llevar a cabo este proyecto, se ha seguido una planificación constante en la que se asignaba a las distintas tareas una fecha límite para su realización y se evaluaban constantemente los resultados y posibles mejoras a fin de ir añadiendo o modificando las tareas futuras. En los apartados de “Aportaciones” y en el de “Conclusión y mejoras futuras” se recoge el impacto social del producto desarrollado.

CII018 - Conocimiento de la normativa y la regulación de la informática en los ámbitos nacional, europeo e internacional.

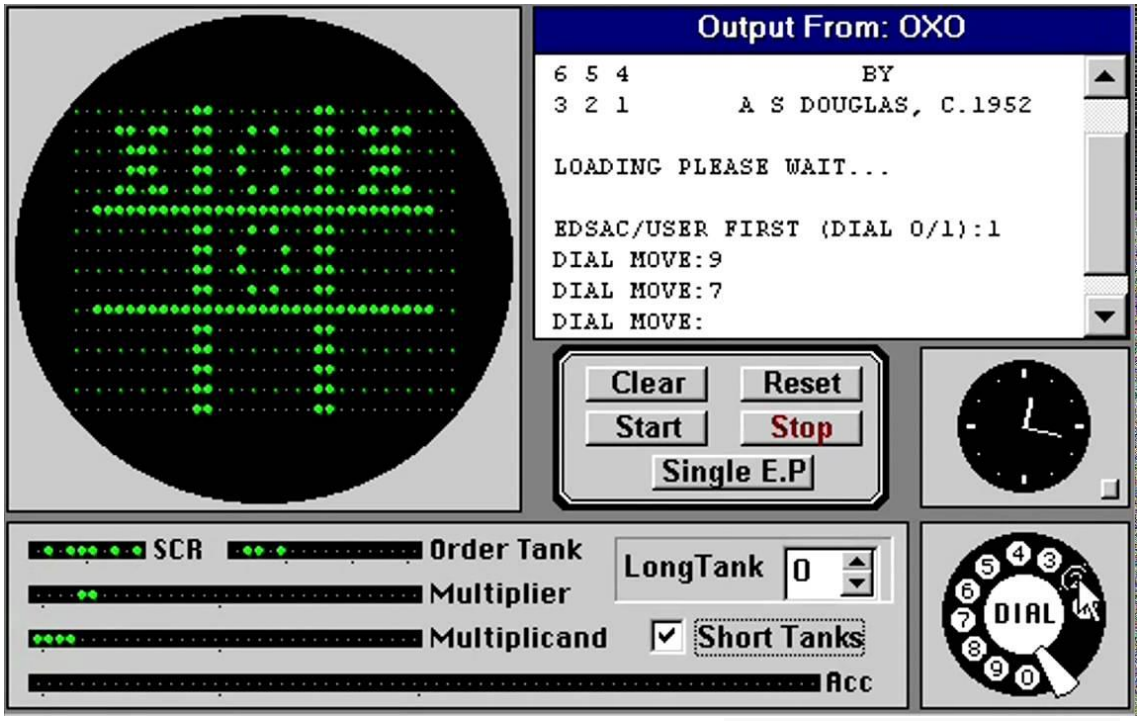
El conocimiento de la normativa y la regulación de la informática en los tres ámbitos que se indican en esta competencia se muestra en el apartado “Normativa y legislación” de este documento.

TFG01 - Ejercicio original a realizar individualmente y presentar y defender ante un tribunal universitario, consistente en un proyecto en el ámbito de las tecnologías específicas de la Ingeniería en Informática de naturaleza profesional en el que se sinteticen e integren las competencias adquiridas en las enseñanzas.

Esta competencia quedará cubierta cuando concluya la presentación del presente trabajo de fin de grado.

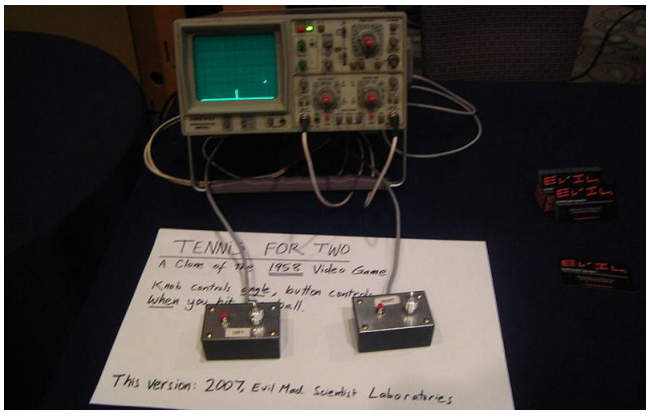
3. Historia de los videojuegos

Para encontrar el primer videojuego de la historia hay que remontarse al año 1947, cuando Thomas T. Goldsmith y Estley Ray Mann patentan un sistema electrónico que simulaba un lanzamiento de misiles contra un objetivo, aunque este proyecto no se considera un videojuego al no haber movimiento en la pantalla. Cuatro años después, se presentó la computadora NIMROD que utilizaba un panel de luces para su pantalla y fue diseñada exclusivamente para jugar el juego NIM, primera computadora diseñada específicamente para jugar un juego. En 1952, Alexander S. Douglas creó el primer juego de computadora en usar una pantalla gráfica digital, OXO, una versión del tres en raya.



OXO, de Alexander Douglas (1952)

A pesar de todos los avances, en 1958 el concepto de videojuego aún resultaba elusivo. Este año, William Higinbotham crea un juego de computadora interactivo llamado *Tennis for Two*, un juego de tenis que había construido con la ayuda del ingeniero Robert Dvorak usando la pantalla de un osciloscopio y circuitería de transistores. El juego, que recreaba una partida de tenis presentando una visión lateral de la pista con una red en el medio y líneas que representaban las raquetas de los jugadores, se manejaba con sendos controladores que se habían construido a tal efecto.



Clon de Tennis for Two en un osciloscopio moderno. La versión original tenía la pantalla redonda.

En 1961 llega el Spacewar! creado por Russell en la computadora PDP-1. Este juego que consiste en dos naves que se enfrentan en el espacio exterior. Es considerado como el primer juego de ordenador de la historia, ya que el Tennis for Two solo usaba circuitería.

Un PDP-1 ejecutando Spacewar!, de Steve Russell (1962).

En 1966 Ralph Baer reconsideró una idea que había abandonado unos años antes: un dispositivo que, conectado a un simple televisor, permitiese jugar al espectador con su aparato. Junto a Bill Harrisony y Bill Rusch siguieron trabajando en el proyecto hasta que en marzo de 1967 finalizaron un primer prototipo que incorporaba ya una serie de juegos, entre los que se encontraban el ping-pong y un juego para dos jugadores en el que ambos debían acorralar al contrario. Baer y sus colaboradores también diseñaron un rifle que, conectado al dispositivo, permitía disparar a una serie de objetivos, y con el prototipo y varios juegos terminados decidió presentar su máquina a Herbert Campman, director de investigación y desarrollo de Sanders Associates. Interesado por la propuesta Campman ofreció 2000 dólares y cinco meses a Baer para que éste completara su proyecto, una cantidad insuficiente pero que "oficializaba" el trabajo de Baer. Su Brown Box, como la llamaba Baer, tuvo un efecto mucho menor en otros altos cargos de la empresa, pero a finales de 1967 el proyecto estaba casi completado y atrajo la atención de TelePrompter Corporation, una compañía de televisión por cable. Tras unas negociaciones que duraron dos meses no se llegó a ningún acuerdo, y las ideas de Baer fueron relegadas por segunda vez al olvido.

En la década de los 70, con el descenso de los costes de fabricación, aparecieron las primeras máquinas y los primeros videojuegos dirigidos al gran público.

A finales de la década de los 60 Bill Pits, un estudiante de la Universidad de Stanford fascinado por Spacewar! tuvo la idea de hacer una versión del juego que funcionase con monedas para su explotación en los salones recreativos. Los elevados costes del hardware requerido fueron un impedimento hasta la aparición en el mercado del PDP-11, más económico. En 1971, Pits y su amigo Hugh Tuck formaron Computer Recreations, Inc., con el propósito de construir una versión operada con monedas de Spacewar!; Pitts se hizo cargo de la programación y Tuck construyó la cabina. Tras tres meses y medio de trabajo habían finalizado la máquina, pero decidieron cambiar el título del programa a Galaxy Game. El invento obtuvo cierta resonancia, pero no resultaba rentable, de modo que construyeron una segunda versión de la máquina que permitía a un sólo computador PDP-11 hacerse cargo de hasta ocho consolas simultáneamente, amortizando así los gastos. La máquina fue instalada en junio de 1972 en el Coffe House de Tresidder Union y allí permaneció con bastante éxito hasta 1979, cuando fue desensamblada y almacenada en una oficina.

Galaxy Game, de Pitts y Tuck (1971).

En 1969, Nolan Bushnell quiso aprovechar la salida al mercado de Data General Nova, un computador con coste bastante inferior a la media de la época, para introducir Spacewar! en los salones recreativos del país. Junto a Ted Dabney construyó un primer prototipo, pero el computador resultó demasiado lento y en 1970 el proyecto fue abandonado. Bushnell y Dabney decidieron construir una nueva máquina dedicada exclusivamente a ejecutar el programa. El nuevo aparato no poseía CPU, sino que usaba componentes discretos fabricados por ellos mismos así como un aparato estándar de televisión en blanco y negro como monitor. En el verano de 1971 Bushnell y Dabney se habían asociado bajo el nombre de Syzygy Engineering para presentar su prototipo a Nutting Associates, una empresa que comercializaba un aparato electrónico de preguntas y respuestas, entre otros productos. Nutting mostró interés hacia el proyecto y se encargó de la fabricación de un primer modelo, usando una carcasa de diseño futurista. En noviembre el primer Computer Space se instaló en el bar Dutch Goose, cerca del campus de la Universidad de Stanford y obtuvo un éxito inmediato entre los estudiantes, animando a Nutting a la fabricación en serie del aparato. Sin embargo, cuando las primeras unidades de Computer Space se pusieron en circulación en bares y salas de juego no consiguieron el éxito esperado. El sistema de control y el objetivo de las partidas resultaban muy complicados para el público no universitario. Bushnell y Dabney finalizaron su contrato con Nutting Associates y el 27 de junio de 1972, por problemas de derechos de autor cambiaron el nombre de su empresa por el de Atari.

Computer Space, de Nolan Bushnell (1971).

En julio de 1968, Bill Enders, trabajador de Magnavox, convence a otros directivos de la empresa para dar una oportunidad a la “Brown Box”, que había seguido siendo presentada por Ralph Baer. Tras una segunda demostración Gerry Martin, jefe de marketing de la división de televisión de la firma, quedó convencido y se hizo cargo del proyecto. Magnavox firmó un acuerdo con Sanders Associates -la empresa para la que trabajaban Baer y sus colaboradores- y en marzo de 1971 se aprobó definitivamente la fabricación del producto. Tras algunos cambios que afectaban sobre todo al número de juegos incluidos, comenzó la fabricación de la recién bautizada Magnavox Odysseyen y en abril de 1972 la firma presentó la nueva máquina a la prensa y a sus distribuidores. Al mismo tiempo se presentó el primer accesorio de la máquina, un rifle de plástico de buena apariencia, y diez juegos adicionales, todos ellos vendidos por separado. Los flyers de la época mostraban ya una consola de videojuegos exactamente tal y como la conocemos hoy, pero Magnavox cometió una serie de errores de marketing que jugaron en su contra: por un lado el anuncio de televisión daba la impresión de que la consola sólo se podría usar con un aparato

Magnavox Odyssey, de Ralph Baer (1972)

de televisión de la misma marca, lo que no era cierto; por otro lado, la distribución se limitó a las franquicias de Magnavox, lo que limitaba considerablemente el número de clientes potenciales. Aun así fue un éxito que atrajo la atención de numerosos emprendedores, entre ellos Nolan Bushnell.

Pong, de Atari (1972)

Tras jugar al Ping-Pong, uno de los juegos que incluía la Magnavox Odysseyen, Nolan Bushnell contrató a Alan Alcorn y lo puso a trabajar en una versión Arcade del juego al que llamó “Pong”. El juego, que se convirtió en el primer título de la recién creada Atari, no suponía grandes innovaciones respecto al título de Baer, pero sí contaba con mejoras (rutina de movimientos mejorada, puntuación en pantalla, efectos de sonido, entre otras). Brushnell y Dabney decidieron probar la máquina creada por Alcorn en un local de Sunnyville, California. A pesar del éxito del prototipo, Bushnell no logró que ninguna empresa lo fabricara en serie, así que decidió que la misma Atari se hiciera cargo de la fabricación y distribución de las máquinas. El resultado fue un éxito y de repente el país se encontraba inundado de máquinas Pong, así como de copias manufacturadas por compañías de la competencia. La japonesa Taito lanzó al mercado oriental su propia versión del juego, y lo mismo ocurrió en países como Francia o Italia. El enorme éxito de la máquina de Atari impulsó las ventas de la Odyssey, la consola que le había dado origen, y a finales de 1974 había cerca de 100 000 máquinas arcade solamente en Estados Unidos que generaban más de 250 millones de dólares anualmente. La industria de los videojuegos había nacido definitivamente.

En 1977, aparece la exitosa Atari 2600 o VCS con su sistema de cartuchos intercambiables. Las maquinas arcade empezaban a hacerse comunes en bares y salones recreativos, una expansión favorecida por el juego Space Invaders creado por Tomohiro Nishikado para Taito, juego que inició el género Shoot 'em up o "matamarcianos". Otros juegos que marcaron esta primera época fueron el Galaxian (1979) de Namco, que supuso la irrupción del color en el mundo de los videojuegos; el Asteroids(1979) o el Pac-man (1980) de Iwatani que, a la inversa de la tendencia de la época, había sido diseñado para ser jugado relajadamente, y evitar cualquier signo de ansiedad en el jugador y que en contra de lo que cabía esperar resultó un éxito inmediato y sin precedentes.


Apple II, uno de los primeros ordenadores personales (1977).

Atari 2600, la segunda consola de la compañía (1977).

En los años 80, la norteamericana Atari hubo de compartir su dominio en la industria del videojuego con dos compañías llegadas de Japón: Nintendo (con su famosa consola NES) y SEGA. Paralelamente, surge una generación de ordenadores personales asequibles y con capacidades gráficas que llegaron a los hogares de millones de familias, como fueron el Spectrum, el Amstrad CPC, el Commodore 64 o el MSX. Los videojuegos empiezan a convertirse en una poderosa industria. Fue además una época muy creativa para los desarrolladores de videojuegos; muchos de los principales géneros que existen hoy en día (conducción, lucha, plataformas, estrategia, aventura...) tomaron forma en esta década.

Commodore 64 (1982).

La crisis llega cuando el mercado se encuentra saturado de juegos, unos clónicos de otros, y decenas de consolas. Los comercios se encuentran con una gran cantidad de material que no pueden vender y tienen que rebajar los precios de manera drástica para conseguir algún beneficio. A consecuencia de esto, en un año los videojuegos pasan de ser la industria con mayor crecimiento a tener la crisis más absoluta.

Al año siguiente seguiría la crisis y es en 1985 cuando la industria se empieza a recuperar. Nintendo lanza el Super Mario Bros., que creó un antes y un después en el mundo de los videojuegos y aparecen míticos juegos como el Tetris. Mientras Sega lanzaría la Master System en Japón y un año más tarde en Estados Unidos, donde no tendría muy buenas ventas.

Sega Master System (1985).

La industria sigue recuperándose y aparecen grandes juegos como el Legend of Zelda, Metroid, Arkanoid, Castlevania, Maniac Mansion,MegaMan, Metal Gear, etc.

Commodore Amiga 500 (1985).

Nintendo Game Boy (1989).

A finales de esta década aparecerían las Amiga 500 y Amiga 2000 y la Mega Drive saldría a la venta en Japón, lo que hizo que Sega le fuera ganando terreno a Nintendo poco a poco.

Aparecieron también las primeras consolas de bolsillo que aunque hasta la llegada de la Gameboy de Nintendo (1989) solo ejecutaban un juego cada una, alcanzaron gran popularidad entre los más jóvenes.

Algunos de los nuevos juegos de esta época trajeron consigo nuevos géneros. Populous (1989), de Molyneux, tomando prestadas algunas de las ideas de SimCity (Apple, 1989) fue el primer representante de un nuevo género, el de simulación de dioses. Sid Meier, uno de los fundadores de Microprose, publicó en 1991 su Civilization, un juego que sentaba las bases definitivas de género de estrategia por turnos.

1992 y 1993 también están plagados de juegos que hoy en día se han convertido en clásicos como Mortal Kombat, Wolfenstein 3D, Alone in the Dark, Doom y FIFA.pong

Sega Megadrive (1990).

Los años 90 traen el salto a la tecnología de 16-bit, lo que significa importantes mejoras gráficas. Entra en escena el gigante Sony con su primera Playstation (1994), mientras Nintendo y Sega actualizan sus máquinas (Nintendo 64 y Sega Saturn). En cuanto a las computadoras, el progreso de los PC termina por barrer del mapa a los demás sistemas salvo el de Apple. Aparecen juegos cada vez más avanzados tecnológicamente, como los shooters en 3D. En el año 2002 entra Microsoft en el sector de las videoconsolas con su Xbox. En 2004 salió al mercado la Nintendo DS, primer producto de la nueva estrategia de una compañía que había renunciado al mercado de las videoconsolas clásicas, y poco después apareció la Sony PSP, una consola similar que no llegó a alcanzar a la primera en cifras de ventas. En 2005 Microsoft lanzó su Xbox 360, un modelo mejorado de su primera consola diseñado para competir con la Playstation 2, pocos meses Sony después lanzó su Playstation 3 y en el 2006 Nintendo lanza su innovadora Wii. En los PC, gracias a la expansión de internet, cobran protagonismo los juegos en línea y multijugador, triunfan los videojuegos de estrategia en tiempo real (Warcraft, Age of Empires) y los juegos de acción en línea (Call of Duty,Battlefield 1942).

Para muchos aficionados, la profesionalización de la industria trajo consigo un cierto estancamiento de la originalidad que había caracterizado el trabajo de los desarrolladores de décadas anteriores, aún así aparecen títulos como Guitar Hero (2005), con un original sistema de control que abrió una lucrativa franquicia. The Sims (2000) puso de moda los juegos de simulación social, un género relacionado con los videojuegos de rol en línea cuyo representante más importante fue Ultima Online, que a su vez tenían su origen en los MUD de mediados de la década de 1980.112 Por su parte, Grand Theft Auto III (Rockstar Games 2001) iniciaba una serie de videojuegos que mezclaban dos de las tendencias más características de las nuevas generaciones -argumentos cada vez más complejos y libertad de movimientos y de acción- y que se convertiría en una de las series más exitosas de todos los tiempos.

Otro fenómeno que ha caracterizado las prácticas de las compañías desarrolladoras en los últimos años ha sido la explotación de franquicias, series de programas basados en los personajes de un videojuego original; títulos como Halo: Combat Evolved (2001), Resident Evil (1996), Silent Hill (1999),Prince of Persia (2003), The King of Fighters (1994), Final Fantasy (1987), Street Fighter (1987), Call of Duty (2003), Metal Slug (1996), Medal of Honor (2001), o las ya citadas Guitar Hero (2005) o Grand Theft Auto III (2001).

En la década de 2010 emergen como plataforma de juegos los dispositivos táctiles portátiles, como los smartphones y las tabletas, llegando a un público muy amplio. Por otro lado, varias empresas tecnológicas empiezan a desarrollar cascos de realidad virtual que prometen traer nuevas experiencias al mundo del entretenimiento electrónico.

4. Situación actual

La industria de los videojuegos es el sector económico involucrado en el desarrollo, la distribución, la mercadotecnia y la venta de videojuegos y del hardware asociado a ellos. Desde la década del 2000, los videojuegos han pasado a generar más dinero que las industrias del cine y la música juntas.

4.1. Los motores de desarrollo de videojuegos

Entre 1973 y 1990 comienzan a emplearse los motores gráficos también conocidos como motores de desarrollo de videojuegos o simplemente motores de videojuegos, sistemas diseñados para la creación y desarrollo de los mismos. En un principio los escenarios eran muy sencillos y las implementaciones eran costosas, pero esta situación cambia a medida que aparecen las texturas, los modelos 3D, mejora el hardware gráfico, etc. hasta llegar al foto realismo de nuestros días.

Uno de los detalles más importantes de algunos de los motores de videojuegos actuales, es que permiten que prácticamente cualquier persona pueda crear su propio videojuego desde cero. Esto ha favorecido la aparición de los videojuegos independientes o indie games, videojuegos creados por individuos o pequeños grupos, sin apoyo financiero de distribuidores. Eso es precisamente lo que se pretende con el presente proyecto: crear un videojuego independiente, o al menos una demo del mismo, con el apoyo de uno de los motores de videojuegos actuales.

A continuación veremos algunos de esos motores de videojuegos. Son muchas las posibilidades, por ello en este documento se resaltarán aquellas que más llamaron la atención de la alumna.

4.1.1. Unreal Engine 4

Unreal Engine, creado por Epic Games en 1998, ha sido siempre uno de los motores gráficos más sonados dentro de la industria del videojuego y su última versión, Unreal Engine 4, no podía faltar entre los destacados. Es el sucesor de Unreal Development Kit (UDK), la versión gratuita de Unreal Engine 3. Puede emplearse de forma gratuita, aunque a cambio debe pagarse un 5% en royalties, siempre y cuando la empresa genere más de 3.000 dólares por producto y trimestre.

Unreal Engine 4 tiene unas capacidades gráficas impresionantes, incluida la iluminación dinámica y un sistema de partículas que permite manejar un millón de partículas en una misma escena. Un sueño hecho realidad para los artistas 3D. Permite portar los videojuegos a PC, Mac, iOS, Android, Xbox One y PlayStation 4.

No es uno de los motores gráficos más fáciles de utilizar, aunque aparentemente es más sencillo que su predecesor el UDK. El lenguaje en el que se programa es C++.

4.1.2. Unity 5

Unity3D es posiblemente uno de los motores gráficos más conocidos a día de hoy. Robusto, fácil de usar, potente, versátil tanto si eres artista como si eres programador, compatible con un montón de plataformas, innovador en el modo que afronta el desarrollo de un videojuego y sobre todo y lo que es más importante… con una comunidad de usuarios increíble detrás. Se trata un entorno de desarrollo muy potente, pero se potencia mucho más con los aportes de terceros a través de su Asset Store. Cientos de desarrolladores venden sus plugins, recursos y mejoras a través de esta tienda (algunos se ofrecen gratuitamente). Unity permite programar en C#, JavaScript y Boo.

El precio de la licencia de Unity Pro es de 1.500 $ por persona más impuestos. Permite el uso de todas las prestaciones de Unity Pro en hasta 2 ordenadores (de la misma persona). Las principales mejoras se encuentran en efectos, texturas y rendimiento 3D. Juegos sencillos, y en especial, juegos sencillos 2D, no deberían necesitar de estas prestaciones.

Unity 5 es la última revisión, entre sus novedades más importantes destacan la iluminación en tiempo real y un nuevo tipo de shaders que aumenta enormemente la calidad de renderizado.

En el sistema de sonido, se nos permite por ejemplo realizar mezclas en tiempo real. Unity 5 da el salto a los 64 bits, un paso importante que allana el camino para la llegada de tablets y smartphones con procesadores basados en dicha arquitectura. También soporta PhysX 3.3, la herramienta de física de NVIDIA, y físicas específicas para juegos en dos dimensiones.

Unity es un motor gráfico históricamente asociado a los juegos para dispositivos móviles, pero el lanzamiento de Unity 5 y su nuevo sistema de renderizado suponen nuevas y potentes capacidades para crear juegos con gráficos realistas que ofrezcan una gran inmersión al jugador para dispositivos cada vez más potentes.

4.1.3. CryEngine

CryEngineE es un motor gráfico extremadamente potente, desarrollado por Crytek e introducido con su primer Far Cry. El motor está diseñado para usarse en juegos de PC y consolas, incluyendo PlayStation 4 y Xbox One. Las capacidades y potencia de este motor sobrepasan a motores como Unity, tan solo Unreal Engine 4 puede competir en aspectos como las físicas, los sistemas de animación de modelos y la capacidad de iluminar en tiempo real las escenas. No es un motor gratuito a diferencia de Unreal Engine 4 o Unity 5, sino que tiene una cuota de $9.90 al mes.

El principal atractivo de Cryengine 3, la última versión, es la facilidad con la que se pueden crear escenarios de una calidad asombrosa. Sin duda alguna es un motor

gráfico creado para impresionar por su belleza y potencia, pero no cuenta con demasiadas facilidades a la hora de crear otra cosa que no sea un shooter. Esto no significa que no sea un motor versátil, pero no se puede considerar la mejor opción para aprender a crear juegos propios. Eso sí, a la hora de crear escenarios de junglas y bosques no encontraremos nada que dé unos resultados tan buenos.

CryEngine es un motor gráfico fantástico, pero tiene una gran curva de aprendizaje, por lo es necesario invertir bastantes horas ante de comenzar a ser productivo con él. Se programa en C++ y Lua.

4.1.4. Otros

Entre otros motores y frameworks destacados para el desarrollo de videojuegos se encuentran

- Source 2
- Torque
- Cocos2D (Cocos2DX, Cocos2DJS, Cocos2D-XNA, Cocos2DSwift, Cocos2D(Python))
- UbiArt
- Stencyl
- GameMaker/GameMarker: Studio

4.2. Elección del motor

De entre los motores de desarrollo anteriormente citados, se escogió Unity para la realización de este proyecto. Algunos de los motivos para tomar esta decisión fueron los siguientes:

- **Facilidad de aprendizaje**. Si bien es cierto que se necesita bastante tiempo para hacerse con las técnicas más avanzadas de Unity y convertirse en un verdadero experto, dominar los elementos más básicos de esta herramienta y poder crear con ellas un videojuego desde cero se puede conseguir con relativa rapidez.
- **Abundantes prestaciones**. Unity permite crear videojuegos muy diversos y de gran calidad gracias a las numerosas opciones que posee, incluso en su versión gratuita.
- **Abundante documentación.** La documentación de Unity es abundante. Sus tutoriales en vídeo o texto son claros y algunos están traducidos al español. El soporte es relativamente rápido y conciso.
- **Gran comunidad.** Gracias a la comunidad es fácil encontrar soporte en los foros oficiales de Unity o en otros foros externos cuando ocurre cualquier problema, ya que muy probablemente a otra persona le ha sucedido lo mismo con anterioridad y ya se ha resuelto.
- **Versión gratuita.** Al disponer de una versión gratuita, este motor está completamente al alcance de la alumna.

Podría decirse que el mayor inconveniente de Unity es el elevado coste de su licencia de Unity Pro, pero las prestaciones que ofrece la versión gratuita en general son suficientes para los desarrolladores freelance o grupos de desarrollo pequeños como es el caso de la alumna.

4.3. Elección del lenguaje de programación

De los tres posibles lenguajes de programación que permite emplear Unity (JavaScript, C# y Boo), se decidió utilizar C# en primer lugar porque ya se tenían conocimientos previos en programación y este lenguaje permite manipular muchos más datos que JavaScript y emplear características más llamativas. JavaScript es más recomendable para los que comienzan en el mundo de la programación por ser más sencillo de entender y manejar. En cuanto a Boo, al ser un lenguaje más reciente existe menos soporte y además tiene una sintaxis tipo Python con la que la alumna no está familiarizada. Por añadidura, la alumna se siente más cómoda empleando lenguajes más estrictos y exactos, con menos probabilidad de error.

5. Estudio y aprendizaje de las herramientas

Durante las primeras semanas de este proyecto, se dedicaron muchas horas a familiarizarse con las herramientas que se iban a utilizar. Puesto que Unity sería la herramienta principal y la que más frecuentemente se emplearía, el funcionamiento de su interfaz de usuario fue lo primero que se aprendió.

5.1. La interfaz de usuario de Unity

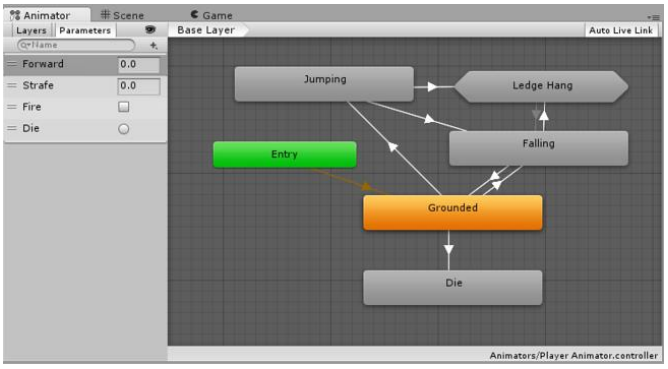
La interfaz de usuario de Unity contiene la barra de herramientas y una serie de "vistas", de las cuales las más importantes son

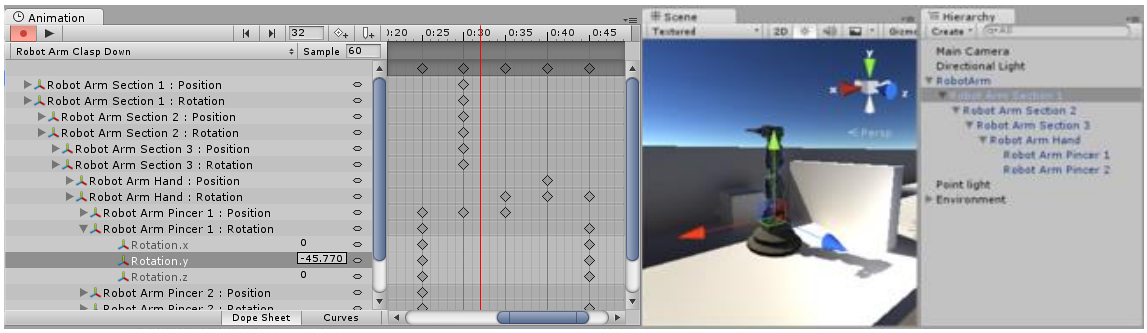
- La vista de escena. Es el área de construcción, donde se construye cada escena del juego.
- La vista del juego. Contiene una vista previa del juego, desde esta vista se puede reproducir el juego y jugarlo.
- La vista del proyecto. Contiene la librería de assets del proyecto. Se pueden importar objetos 3D, texturas, imágenes,... y también generar otros en la propia plataforma Unity, como en el caso de los prefabs. Los assets importados permanecerán en el proyecto al que fueron añadidos aunque se cierre Unity.
- La vista de jerarquía. Contiene todos los elementos de la escena actual.
- La vista del inspector. La vista de inspector sirve para varias cosas. Si se seleccionan objetos mostrará las propiedades de ese objeto pudiendo personalizar las características del mismo. También contiene la configuración para ciertas herramientas como la herramienta de terrenos el terreno está seleccionado.

Interfaz de Usuario de Unity. Imagen obtenida de la página oficial de Unity. Apartado de documentación. Manual

Otras vistas de Unity son

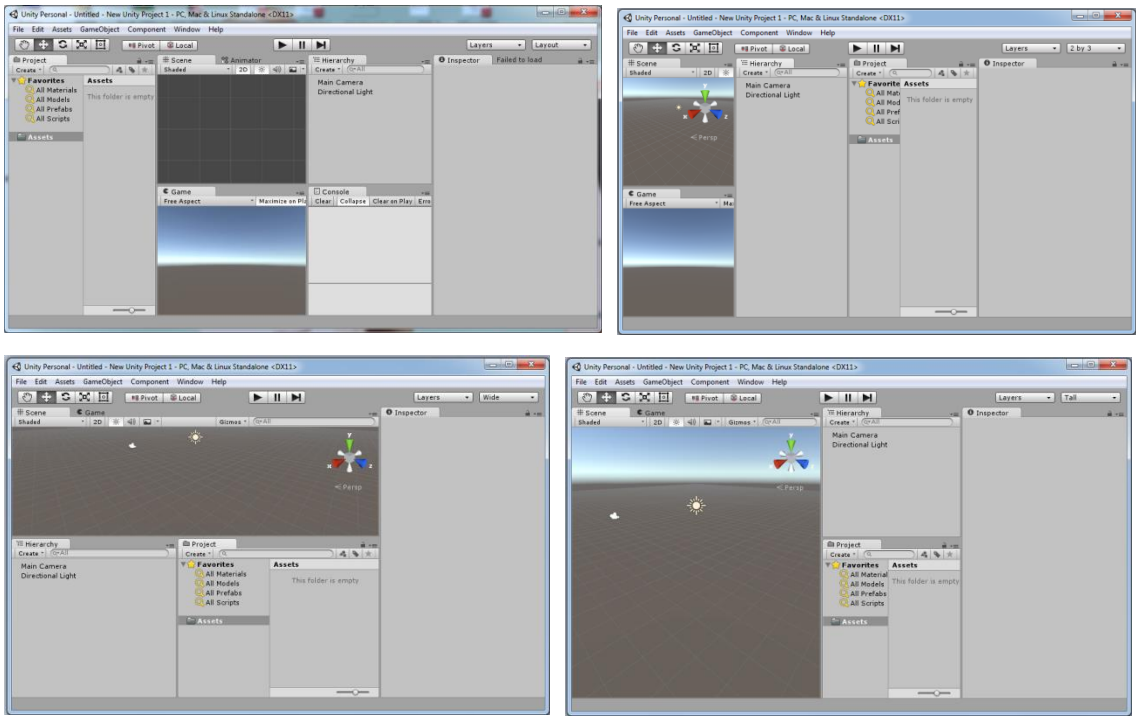
- La vista de animación. Puede ser utilizada para pre-visualizar y editar las animaciones de los objetos animados.
- La vista del animador (animator). Permite gestionar las animaciones asociadas a un objeto. Por ejemplo, si tenemos un personaje andando e iniciamos la acción de correr, con el *Animator* se cambiará la animación de "andar" por la de "correr".





Arriba: Vista del animator. Abajo: Vista de animación. Ambas imágenes han sido obtenidas de la página oficial de Unity. Apartado de documentación. Manual

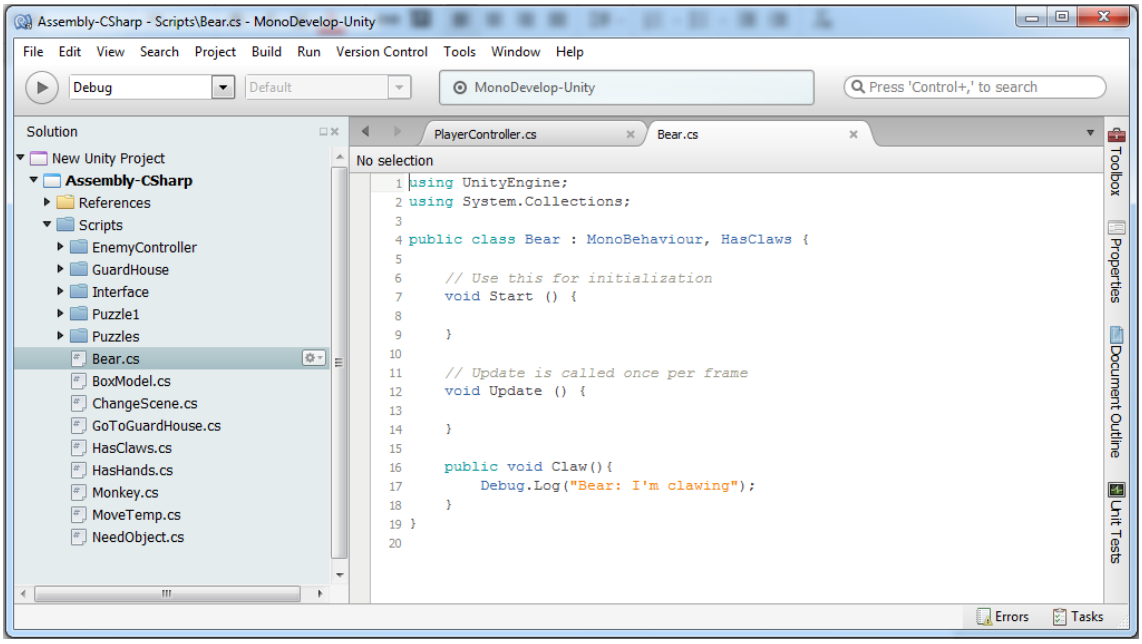
La interfaz por defecto de Unity se puede modificar para personalizarla a gusto del usuario. Las vistas son ventanas que pueden estar flotando de forma independiente, agrupadas en pestañas u ocultas, y se pueden mover y soltar en casi toda la interfaz.



Ejemplos de distintas personalizaciones de la interfaz de usuario de Unity

5.2. MonoDevelop

El MonoDevelop es el editor predeterminado de los scripts en Unity y se instala por defecto con este motor, aunque es posible excluirlo de la instalación si se desea. Cuando creamos un nuevo script desde Unity, por defecto se añaden las líneas que importan las librerías UnityEngine y System.Collections. La primera nos permite emplear clases y métodos propios de Unity como las clases GameObject, Camera, Sprite,… o los métodos Start(), Update(), OnTriggerEnter(Collider other),.. La segunda, también conocida como Espacio de nombres, contiene interfaces y clases que definen varias colecciones de objetos, como listas, colas, matrices de bits, tablas hash y diccionarios.



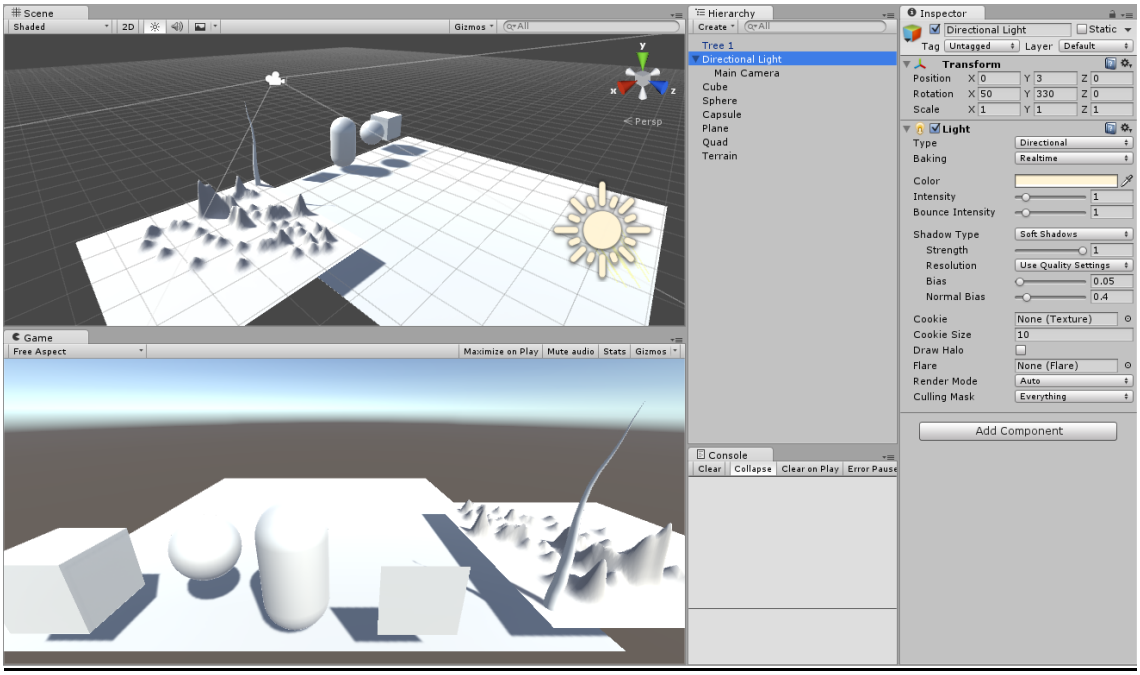
Captura de pantalla del MonoDevelop.

5.3. Algunos elementos de Unity y primeros experimentos

Gracias a los manuales, la API y los tutoriales presentes en la propia página de Unity, así como a los artículos, foros y tutoriales de otros usuarios, la alumna fue adquiriendo conocimientos sobre elementos de Unity como

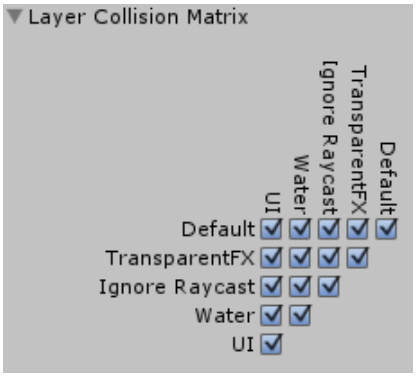
- Los objetos 3D simples y el terreno. Los elementos 3D se pueden añadir fácilmente a la escena mediante la pestaña GameObject > 3D Object y posteriormente editar su posición, rotación, tamaño,… y/o añadirles componentes como colliders que los hagan "sólidos", scripts que controlen algunos comportamientos, animaciones,… Un elemento especial de los objetos 3D es el terreno.
- Las luces. Las luces se emplean para iluminar la escena, existen tres tipos:
    - Spot Light. La luz parte de un punto y se expande cónicamente en una dirección, de forma similar al haz de luz de una linterna.
    - Directional Light. Es una luz de ambiente con dirección, su efecto en la escena sería similar al del sol en la vida real.
    - Point Light. La luz parte de un punto en todas las direcciones de forma similar a la de una bombilla.

- Las cámaras. Se encargan de “capturar” lo que ve en la escena y esta captura será lo que se verá al ejecutar nuestro juego. Las capturas de las distintas cámaras se superponen en la vista del juego según indica el atributo “Depth” que poseen, de esta forma, aquella cuyo valor en Depth sea más grande, se verá por encima de las demás, a continuación la siguiente con el valor mayor y así sucesivamente. La captura de una cámara puede cubrir parte de la vista del juego o su totalidad.
- Los objetos padres/hijos (jerarquía). Los objetos se pueden agrupar “dentro” de otro objeto (padre) en la vista de jerarquía, los cambios de posición, rotación y tamaño realizados sobre el padre afectan de igual forma a sus hijos.



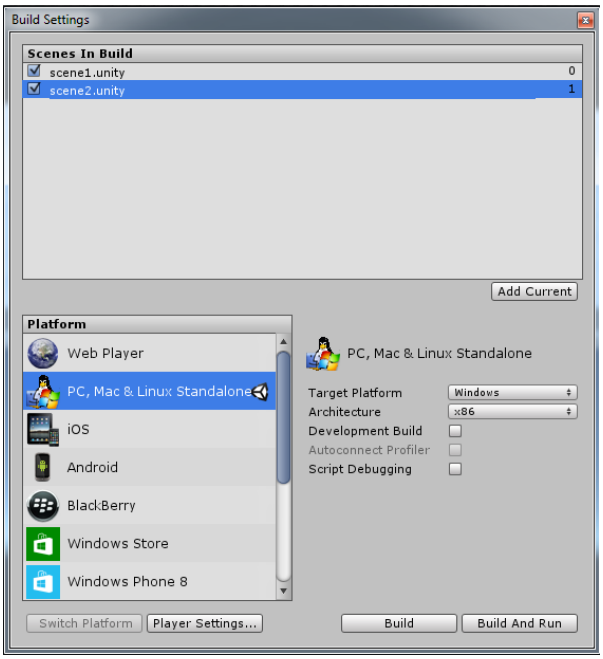
Ejemplo de objetos 3D en Unity (cubo, esfera, terreno,…), directional light y jerarquía

- Los colliders. Los colliders son elementos que pueden añadirse a los objetos para que sean sólidos y/o puedan disparar eventos en caso de colisión. Un collider con el atributo “IsTrigger” verdadero son traspasables, los colliders tirgger se emplean para controlar cuando otro collider los atraviesa.
- Las funciones más comunes de los scripts (Awake, Start, Update, FixedUpdate, Debug.Log, Input.GetKeyDown/Up, OnTriggerEnter/Exit/Stay, etc.) cuyo funcionamiento podría intuirse en algunos casos por el nombre, pero que se verá mejor en apartados posteriores.
- Las tags. Etiquetas para diferenciar los objetos de la escena
- Los layer y la matriz de colisiones. Los layers son capas que permiten crear comportamientos diferentes para cada tipo de objeto conjuntamente con la matriz de colisiones. Por ejemplo, si tenemos varios enemigos cada uno con su collider y queremos que choquen con todos los elementos de la escena, pero no entre ellos (que se atraviesen), podemos ponerles un layer “Enemy” e indicar en la matriz de colisiones que los Enemy no interactúan entre ellos.



- Los raycast. Son rayos invisibles que parten desde un origen concreto, con una dirección y que se propagan en línea recta hasta que chocan con un objeto de la escena. Se emplean frecuentemente en los *shooters* para emular el disparo de un arma sin que el rendimiento del juego se vea afectado por los costes que supondría disparar varias balas a gran velocidad y controlar su posterior eliminación para no llenar la memoria de objetos.

- Las escenas (niveles) y el Build Settings. Unity permite crear distintas escenas en un mismo proyecto, estas escenas se corresponderán con los niveles del juego. Las escenas del juego se añaden al apartado de "Build Settings" para que aparezcan en dicho juego y se pueda pasar de una a otra mediante scripting empleando el orden que tienen asignado.



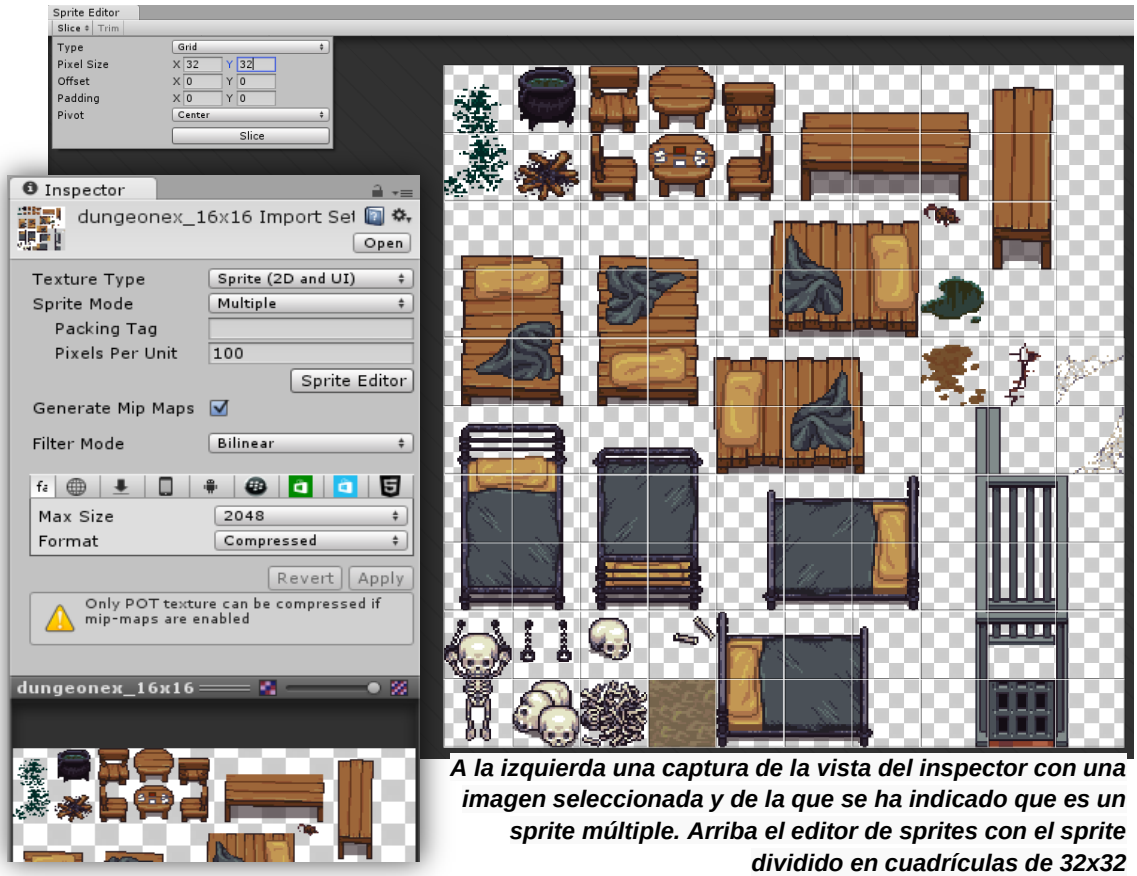
- El sistema de partículas. Se emplea para representar entidades de los juegos que son fluidas e intangibles por naturaleza tales como líquidos en movimiento, humo, nubes, llamas y hechizos mágicos.

- Los sprites. Los sprites son imágenes que normalmente se emplean en juegos 2D para crear los gráficos de los personajes. Mediante una secuencia de sprites se puede generar una animación que simule, por ejemplo, un personaje andando.

  Unity permite importar sprites fácilmente y respeta las transparencias de las imágenes.

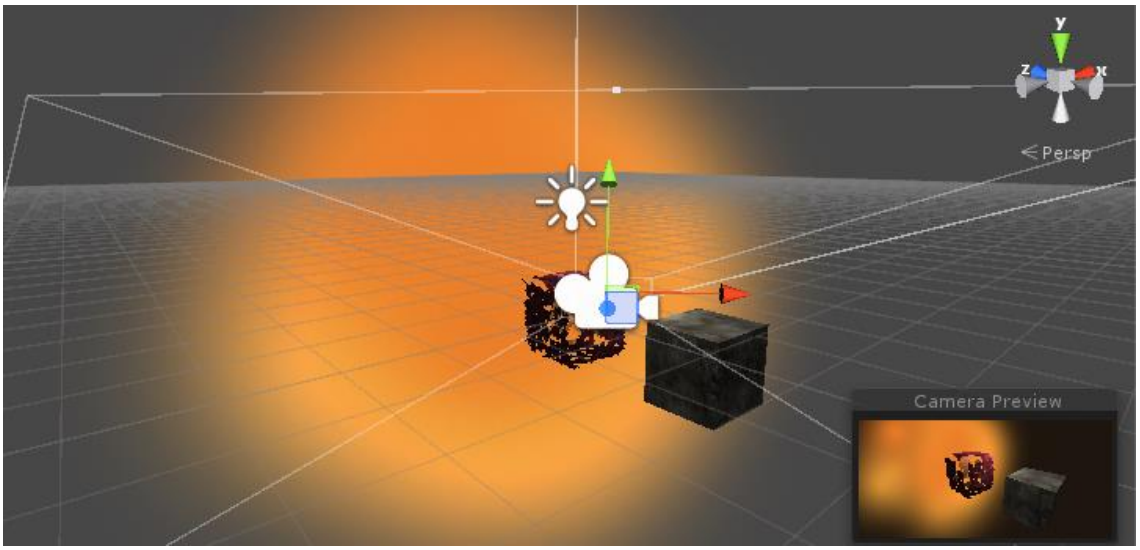
  Cuando importamos una imagen con múltiples sprites como la que se muestra a la derecha, tenemos que indicar a Unity que la imagen es efectivamente un sprite y además que no es único (single) sino múltiple. Hecho esto, se deberá emplear el editor de sprites para dividir la imagen entre los distintos sprites. Podemos hacer esta separación de forma automática, en cuyo caso el editor se basará en las transparencias para separar los sprites ajustándose a los límites de éstos, o mediante una cuadrícula con celdas de un tamaño indicado.

Imagen con múltiples sprites que muestran distintas posiciones de un oso al andar



A la izquierda una captura de la vista del inspector con una imagen seleccionada y de la que se ha indicado que es un sprite múltiple. Arriba el editor de sprites con el sprite dividido en cuadrículas de 32x32

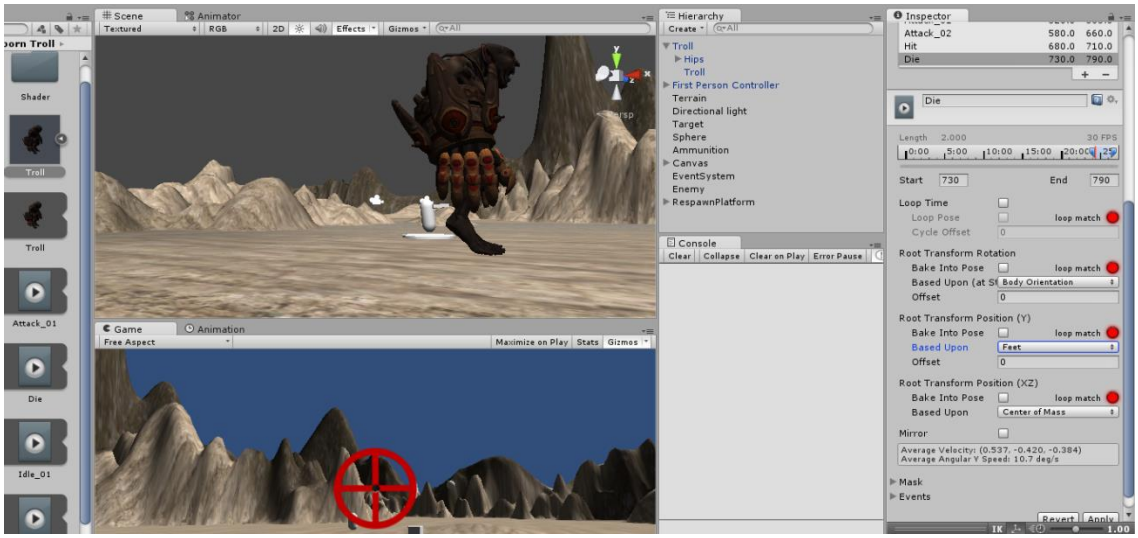
A medida que se adquirían conocimientos sobre todos los elementos de Unity citados, estos se iban poniendo en práctica mediante pequeños experimentos. La captura que se muestra a continuación contiene un ejercicio en el que la alumna juega con el sistema de partículas y finalmente intenta crear el efecto de una llama.



Una vez los elementos mencionados ya eran conocidos, se pasó a seguir un tutorial para crear una demo sencilla del juego SlenderMan con el objetivo de afianzar lo aprendido. El modelo 3D de SlenderMan y los scripts necesarios ya se incluían en el tutorial, pero como elementos separados, por lo que había que importarlos y asignar los scripts a los distintos objetos que se crearían en la escena.

Acabado el proyecto, se crearon los archivos de datos y el ejecutable, consiguiendo así la demo definitiva que funcionaba como cualquier juego normal, sin necesidad de volver a utilizar la plataforma Unity para su uso. A continuación se muestra una captura con el resultado.

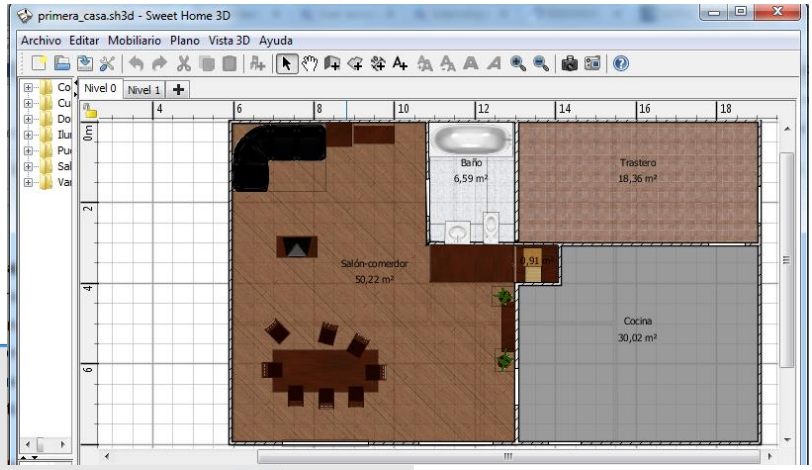
En otro ejercicio más avanzado, se trabajó con las diferentes opciones del terreno, la iluminación, objetos simples con sus colliders, jugador en primera persona, animación de un enemigo (modelo y animaciones obtenidos de la Asset Store de Unity), la interfaz en el juego (vida, munición,…) y scripts que controlarían eventos del juego, comportamiento del enemigo, cambios de escena, etc.



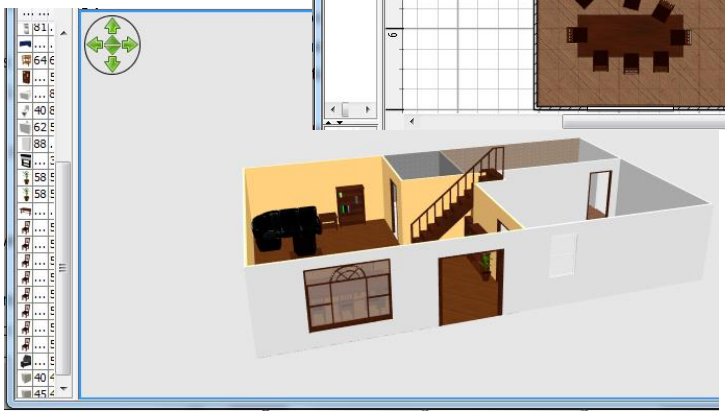
Captura del ejercicio realizado. En ella se puede apreciar al trol enemigo, el cilindro con cámara que es el jugador en primera persona y el terreno entre otros elementos.

También se estudiaron brevemente herramientas de diseño 3D como SweetHome3D y Blender.

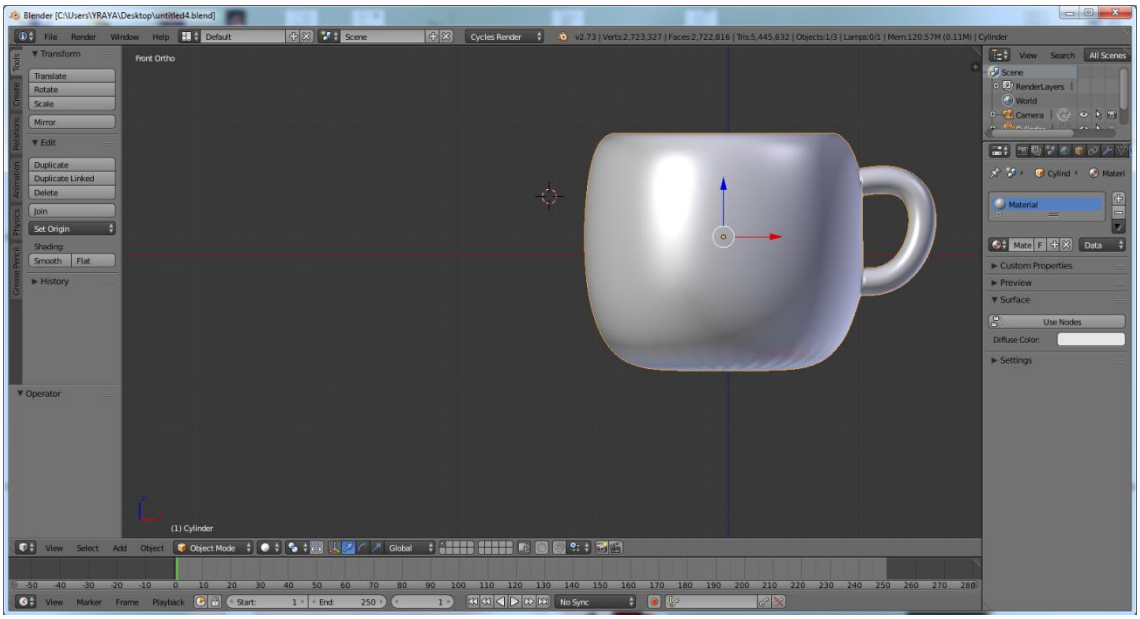
- Sweet Home 3D es una aplicación libre de diseño de interiores que funciona muy bien con Unity y cuya principal utilidad consiste en crear casas con sus respectivos objetos sobre un plano en 2D, con una vista previa en 3D. Existen gran cantidad de elementos gratuitos para usar en Sweet Home 3D (mesas, sillas,cajas, electrodomésticos, vehículos,... incluso animales y personas estáticos). El programa es bastante sencillo de utilizar.
- Blender es un software libre y de código abierto para la creación de elementos 3D completo y potente. El mayor inconveniente radica en lo complicado de su uso, al menos inicialmente, para emplear con soltura la herramienta el usuario debe conocer un buen número de atajos de teclado.



Capturas de un mismo proyecto realizado por la alumna con Sweet Home 3D. A la derecha la vista en 2D y abajo la vista en 3D.



Captura del programa Blender con una taza en 3D creada por la alumna siguiendo un tutorial



6. Análisis y diseño de propuestas

Durante las semanas que siguieron a la familiarización con las herramientas, se dedicó tiempo a pensar y diseñar posibles juegos. Teniendo en cuenta el limitado tiempo disponible, hubo que descartar juegos con grandes requerimientos gráficos y/o complejas historias.

Capturas de bocetos realizados en papel y digitalmente sobre una de los posibles juegos que consistía en deshacerse de mosquitos cada vez más hábiles noche tras noche (inspirado en plantas versus zombis)

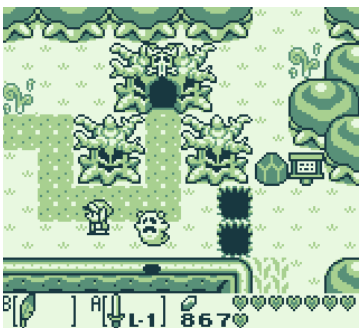
Finalmente, se eligió un juego cuya principal característica consistía en ir rotando entre varios personajes disponibles para poder realizar distintas acciones. El juego tendría lugar en un zoo y sus protagonistas serían animales que trataran de escapar.

El comienzo del juego consistiría en un primer animal, en este caso se decidió que fuera un mono, escapando de su jaula. Una vez fuera, tendría que moverse por el zoológico evitando a los guardias y liberando a los animales que pudiera. Los animales liberados ayudarían a liberar a su vez a otros animales. Cuando todos los animales fueran liberados sería posible escapar del zoo.

En un principio, se pensó en este juego con un diseño 3D, pero pronto se tuvo que abandonar esa idea debido sobre todo a los modelos de los personajes animales, ya que supondrían un coste económico importante si se adquirían ya hechos o un coste de tiempo si la alumna tenía que crearlos por su cuenta.

Se consideraron entonces dos opciones:

- 2D con perspectiva lateral como en el Super Mario Land (izq.)
- 2D con perspectiva top-down (vista desde arriba) como en The Legend of Zelda: Link's Awakening (der.)



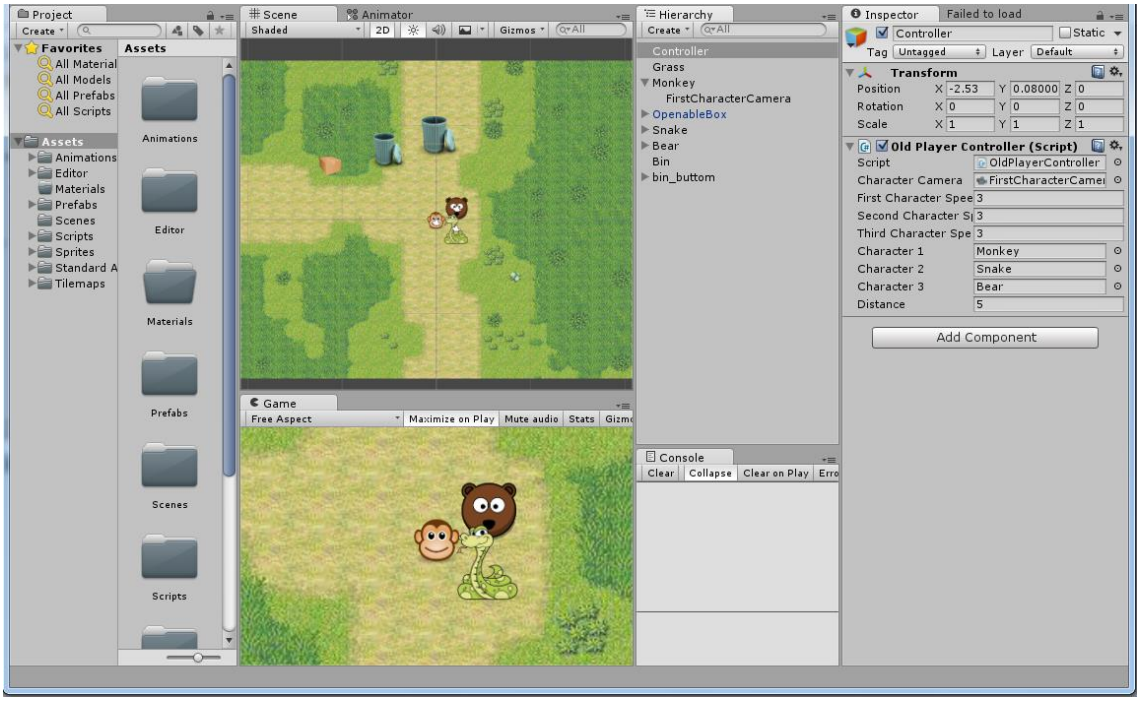
Al final, la alumna se decantó por la perspectiva top-down al considerar que esta otorgaba más posibilidades y se asemejaba más a la idea inicial de 3D. Además, al ser una perspectiva menos popular en Unity, se esperaba que supusiese un reto más interesante.

7. Prototipado

Durante esta fase, se desarrollaron distintas versiones del juego.

7.1. Primer prototipo

Inicialmente, se diseñó una pequeña parte de lo que sería el juego, tratando de crear el mínimo número de assets para obtener un resultado funcional, prácticamente ignorando las características gráficas, ya que éstas se usaban únicamente para diferenciar los elementos de la escena. Los sprites de la escena son simples imágenes que se han importado al proyecto y luego han sido colocados en la escena.



Para este primer prototipo, se pretendía implementar algunas de las funciones más básicas del juego:

- Movimiento de los personajes
- Rotación entre los personajes
- Diferenciación de los personajes a la hora de realizar acciones, por ejemplo, un oso podría abrir una caja con sus garras, pero un mono y una serpiente no podrían.

Antes que nada se comentarán tres de las funciones propias y básicas de Unity que se van a utilizar en este prototipo:

- Start(): Es una función que se ejecuta automáticamente y sólo una vez cuando el script es cargado y está habilitado. Suele emplearse para la inicialización de las variables.
- Update(): Es una función que se ejecuta automáticamente en cada frame del juego. La primera ejecución se realiza después de que finalice Start().
- FixedUpdate(): Es una función similar a Update(), salvo que siempre se ejecuta en intervalos regulares de tiempo, cosa que no sucede con Update(), por lo que es más apropiada para actualizaciones regulares como las físicas de los componentes (componentes con RiggidBody).

7.1.1. Movimiento y rotación de los personajes

Para implementar el movimiento y la rotación entre personajes de la escena, se añadirá a cada uno (el mono, la serpiente y el oso) el componente Riggidbody 2D. Este componente hace que los objetos 2D se vean afectados por el control del motor de física, es decir, permite que fuerzas externas afecten a los objetos que tienen este componente. Dado que el juego que se pretende realizar tiene una perspectiva top-down, el valor que se le dará a la gravedad será cero, ya que de otra forma los personajes se deslizarían hacía abajo o hacia arriba en el juego. También se pondrá como verdadero el valor del atributo “Fixed Angle” para que los personajes no roten, sino que mantengan su posición recta al margen de las fuerzas que puedan afectarles.

Se colocará un objeto cámara como hijo del primer personaje para que cuando el personaje se mueva la cámara se mueva a su vez con él. El primer personaje tendrá también el Tag “CurrentPlayer” para diferenciarlo de los otros cuyo Tag será sólo “Player”.

Para controlar a los personajes se creará un objeto vacío (sin ningún componente) llamado Controller al que se le añadirá el script que contendrá las funciones de movimiento y rotación entre personajes.

En el script del objeto Controller, se declararán las variables necesarias para implementar los comportamientos que necesitamos, tales como la velocidad a la que se moverán los personajes, la distancia entre ellos, los GameObjects implicados, etc.

Las variables que se declaran como públicas son accesibles desde la interfaz de Unity, es decir, si declaramos una variable como “public float speed;” podremos asignar un valor a speed desde la vista del inspector, tanto si no estamos ejecutando el juego como si lo estamos haciendo. Las variables modificadas en tiempo de ejecución desde la interfaz recuperan sus valores anteriores al finalizar la ejecución.

Las variables públicas son útiles para probar distintos valores que afectan al comportamiento de los objetos en el juego de forma rápida y sencilla.

Inicialmente se pretendía que cada personaje tuviera distintas velocidades, por ello se crearía una variable para cada una, pero en versiones posteriores se tendría una única velocidad para todos los personajes por motivos de sencillez a la hora de jugar. Las variables públicas y privadas utilizadas, así como la inicialización de estas últimas en el Start() se muestra a continuación:

```
using UnityEngine;
using System.Collections;

public class OldPlayerController : MonoBehaviour {
    public Camera characterCamera;

    public float firstCharacterSpeed;
    public float secondCharacterSpeed;
    public float thirdCharacterSpeed;

    public GameObject character1;
    public GameObject character2;
    public GameObject character3;

    public float distance;

    private bool updatingPosition;
    private bool changing;
    private bool moveEnable;
    private bool changeEnabled;

    private Vector3 oldFirstCharacterPosition;

    void Start () {
        updatingPosition = false;
        changing = false;
        moveEnable = true;
        changeEnabled = true;

        oldFirstCharacterPosition = character1.transform.position;
    }
```

Para el movimiento del primer personaje, crearemos una función Move() que colocaremos dentro de FixedUpdate(). La función se encargará de modificar la posición del personaje utilizando su componente Riggidbody2D. Para ello se basará en la velocidad del personaje, la dirección en la que se pretende mover (indicada por el usuario mediante las teclas WASD o las fechas de dirección) y la variable estática y sólo de lectura deltaTime de la clase Time. Esta última variable el tiempo en segundos que tardó en completarse el último frame y nos servirá para que el personaje no de saltos por la escena sino que se mueva de forma progresiva por ella.

También en FixedUpdate() controlaremos si la distancia entre los personajes es superior a la que tenemos definida y de ser así actualizaremos la posición del segundo y tercer personaje con una posición antigua de su antecesor. Para que el movimiento de los personajes que siguen al principal no sea muy brusco, emplearemos una corrutina, que nos permite ir actualizando la posición de los mismo frame a frame en lugar de instantáneamente.

```
void FixedUpdate(){
    if (moveEnable){
        Move();
    }
    if ((character1.transform.position -
         oldFirstCharacterPosition).magnitude >= distance){
        //Si estamos actualizando posiciones no iniciamos
        //una nueva corrutina "PositionUpdate"
        if (!updatingPosition && !changing){
            StartCoroutine(UpdatePosition());
        }
    }
}

private void Move(){
    Rigidbody2D rigidBody1 = character1.GetComponent<Rigidbody2D>();
    if(Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.UpArrow)){
        MoveUp(rigidBody1,firstCharacterSpeed);
    }

    if(Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.DownArrow)){
        MoveDown(rigidBody1,firstCharacterSpeed);
    }
    if(Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.LeftArrow)){
        MoveLeft(rigidBody1,firstCharacterSpeed);
    }
    if(Input.GetKey(KeyCode.D) || Input.GetKey(KeyCode.RightArrow)){
        MoveRight(rigidBody1,firstCharacterSpeed);
    }
}

private void MoveRight(Rigidbody2D rigidBody,float speed){
    rigidBody.transform.position += Vector3.right * Time.deltaTime *speed;
}

private void MoveLeft(Rigidbody2D rigidBody,float speed){
    rigidBody.transform.position += Vector3.left * Time.deltaTime *speed;
}

private void MoveUp(Rigidbody2D rigidBody,float speed){
    rigidBody.transform.position += Vector3.up * Time.deltaTime *speed;
}

private void MoveDown(Rigidbody2D rigidBody,float speed){
    rigidBody.transform.position += Vector3.down * Time.deltaTime *speed;
}

IEnumerator UpdatePosition(){
    updatingPosition = true;
    Vector3 secondPosition = character2.transform.position;
    Vector3 thirdPosition = character3.transform.position;

    float step = 0;

    while (step < 1){
        character2.transform.position =
            Vector3.Lerp(secondPosition,oldFirstCharacterPosition,step);
        character3.transform.position =
            Vector3.Lerp(thirdPosition,secondPosition,step);
        step += Time.deltaTime*firstCharacterSpeed;

        yield return null;
    }
    oldFirstCharacterPosition = character1.transform.position;
    updatingPosition = false;
}
```

Dentro de la función Update() controlaremos la rotación entre personajes cuando el jugador pulse la tecla 'Q' o el botón secundario del ratón. Para que la rotación entre ellos no sea brusca también emplearemos una corrutina. Cuando cambiemos los personajes, tendremos no sólo que actualizar sus posiciones, sino también el personaje que tiene la cámara, los Tags "CurrentPlayer" y "Player" y controlar que no se realizan nuevas rotaciones ni movimientos antes de que concluya la corrutina, ya que esto modificaría los datos con los que se está trabajando y provocaría resultados indeseados.

```
void Update () {
    if (changeEnabled){
        if(Input.GetKeyDown(KeyCode.Q) || Input.GetMouseButtonDown(1)){
            StartCoroutine(ChangeCharacter());
        }
    }
}

IEnumerator ChangeCharacter(){
    //Desactivamos el movimiento para que las posiciones
    // no cambien mientras rotamos a los personajes
    moveEnable = false;
    changeEnabled = false;
    changing = true;

    Vector3 firstPosition = character1.transform.position;
    Vector3 secondPosition = character2.transform.position;
    Vector3 thirdPosition = character3.transform.position;

    //Quitamos la camara como hija del character1
    // para que no se mueva
    characterCamera.transform.parent = transform;

    //Cambiamos el tag que indica el jugador actual,
    // ahora sera el segundo
    character1.tag = "Player";
    character2.tag = "CurrentPlayer";

    float step = 0;
    while (step < 1){
        character1.transform.position =
            Vector3.Lerp(firstPosition,thirdPosition,step);
        character2.transform.position =
            Vector3.Lerp(secondPosition,firstPosition,step);
        character3.transform.position =
            Vector3.Lerp(thirdPosition,secondPosition,step);

        step += Time.deltaTime*firstCharacterSpeed*2;

        yield return null;
    }

    //Nos aseguramos de que las posiciones finales
    // sean las deseadas, de lo contrario podriamos
    //acumular un error de posicion al devolver Lerp
    // un numero aproximado pero no igual a uno
    character1.transform.position = thirdPosition;
    character2.transform.position = firstPosition;
    character3.transform.position = secondPosition;

    //Actualizamos la posicion local de los personajes
    GameObject characterAux = character1;
    character1 = character2;
    character2 = character3;
    character3 = characterAux;

    //Actualizar la superposicion de los personajes
    character1.GetComponent<SpriteRenderer>().sortingOrder = 4;
    character2.GetComponent<SpriteRenderer>().sortingOrder = 3;
    character3.GetComponent<SpriteRenderer>().sortingOrder = 2;

    //Actualizamos las velocidades
    float speedAux = firstCharacterSpeed;
    firstCharacterSpeed = secondCharacterSpeed;
    secondCharacterSpeed = thirdCharacterSpeed;
    thirdCharacterSpeed = speedAux;

    //Ahora la camara debe seguir al segundo personaje
    ChangeCamera();

    changing = false;
    moveEnable =true;
    changeEnabled = true;

}
```

```
    private void ChangeCamera(){
        //Cambiar el padre de la camara
        characterCamera.transform.parent = character1.transform;
        //Actualizar la posicion de la camara con respecto al nuevo padre
        characterCamera.transform.localPosition = new Vector3(0,0,-10);
    }
}
```

El atributo “sortingOrder” perteneciente al componente Sprite o SpriteRenderer indica el orden en el que se superponen unos sprites sobre otros, un sprite con un sortingOrder menor que el de otro se verá por detrás de ese en el juego. En este caso se pretende que el primer jugador se vea por encima del segundo y este segundo a su vez por encima del tercero.

El funcionamiento de la mayor parte de los métodos utilizados se puede intuir, quizá el más complicado sea “Vector3.Lerp”. Esta función retorna una posición entre un origen y un destino. Tiene tres parámetros, el primero y el segundo se corresponden con dichas posiciones origen y destino respectivamente; el tercero es un decimal cuyo rango de valores va del cero al uno, ambos inclusive. Si el valor del tercer parámetro es 0, la función retornará una posición igual al origen; si es 1, retornará una igual al destino; si es 0’5, la posición intermedia[1]. En el script se emplea Vector3.Lerp para mover un objeto desde una posición origen a una destino de forma gradual al incrementar poco a poco el valor del tercer parámetro.

Otra de las funciones que podría llamar la atención es la “localPosition”. Los objetos tienen dos posiciones, una relativa a toda la escena y otra relativa a su padre, que es la localPosition. Como nuestra intención es que la cámara esté centrada con el personaje al margen de la situación de ambos en la escena empleamos esta función.

A pesar de que inicialmente se pretendía no añadir muchos comentarios en el código, se ha saltado esta norma para permitir dar explicaciones a personas menos familiarizadas con el funcionamiento de Unity.

7.1.2. Diferenciación de personajes a la hora de realizar acciones

Nuestro siguiente objetivo en este prototipo era conseguir que ciertos objetos sólo pudieran ser afectados por un personaje concreto y no por los otros. Para ello colocaremos el sprite de una caja en la escena con un collider para que los personajes no la atraviesen y un hijo con otro collider trigger a su alrededor que será el encargado de detectar si hay otro collider cerca. Por último, le añadiremos al hijo con el collider trigger el script que controlará su funcionamiento. Queremos que la caja sólo la pueda abrir quien tenga garras (oso).

Crearemos la interfaz “HasClaws” con un método “Claw()” y un script “Bear” que implementará dicha interfaz y que añadiremos al personaje oso.

```
using UnityEngine;
using System.Collections;

public interface HasClaws {

    void Claw();
}
```

[1]Las posiciones en Unity se indican mediante vectores.

```
using UnityEngine;
using System.Collections;

public class Bear : MonoBehaviour, HasClaws {

    public void Claw(){
        Debug.Log("Bear: I'm clawing");
    }
}
```

A continuación, modificaremos el script del objeto hijo la caja para que ésta sólo se abra si el personaje que intenta hacerlo tiene garras o lo que es lo mismo, implementa la interfaz HasClaws.

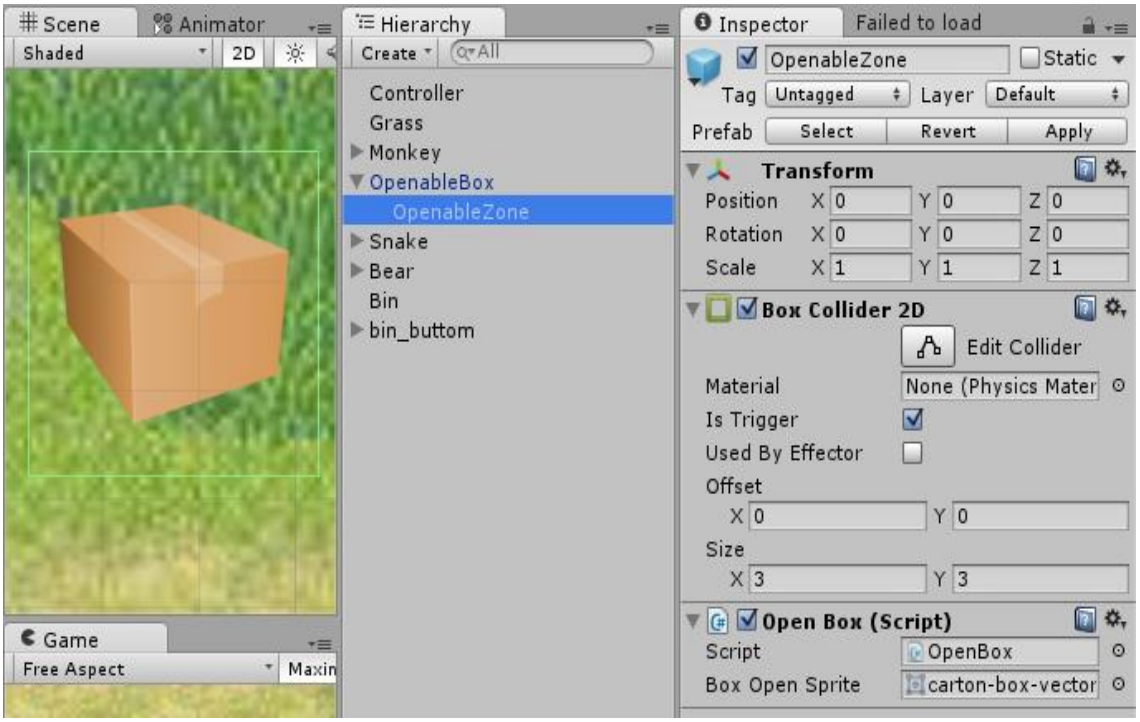
```
using UnityEngine;
using System.Collections;

public class OpenBox : MonoBehaviour {
    public Sprite boxOpenSprite;
    private GameObject box;

    void Start () {
        box = transform.parent.gameObject;
    }

    void OnTriggerStay2D(Collider2D other){
        if(Input.GetKeyDown(KeyCode.E) || Input.GetMouseButtonDown(0)){
            //Comprobamos si el personaje actual puede abrir la caja,
            //para abrir la caja tiene que tener garras/zarpas, los
            //personajes con esta caracteristica tienen una clase
            //que implementa la interfaz "HasClaws"
            HasClaws testInterface = other.gameObject.GetComponent<HasClaws>();

            //Si el personaje tiene esa interfaz el test nos dara distinto de
            //nulo y procederemos a abrir la caja
            if (testInterface!=null){
                box.GetComponent<SpriteRenderer>().sprite = boxOpenSprite;
                //Sonido de abrir caja, rasguño

                //Podemos usar la funcion "Claw" de cada personaje que
                //implemente "HasClaws" para añadir un comportamiento extra
                //distinto para cada personaje
                other.gameObject.SendMessage ("Claw");

                //Una vez abierta la caja, en principio no tiene sentido que
                //pueda seguir siendo "openable" asi que destruimos la
                //OpenableZone y todo su contenido
                Destroy(this.gameObject);
            }
        }
    }

}
```

En el caso del script anterior, el propio scrpit abre la caja cambiando la imagen de ésta cerrada por una de la caja abierta y el método Claw() del oso sólo se emplea para lanzar un mensaje por la consola, pero podría emplearse ese método para un comportamiento más concreto o bien que fuera con ese método con el que se abriera la caja.

El método “SendMessage” de la clase GameObject es la forma por defecto que tiene Unity de ejecutar funciones públicas de un objeto desde otro. Este método tiene un gran inconveniente y es que sólo dispone de un parámetro si queremos enviar valores a la función que se quiere ejecutar y que sería el segundo, el primer parámetro es el nombre de la función.

Con “transform.parent.gameObject” obtenemos el objeto padre del actual, es decir, del que contiene el script que se está ejecutando.

El objeto caja que queríamos que fuera “openable” sólo por un animal con garras y su objeto hijo, que sería la zona desde la que se puede abrir la caja, se muestran a continuación.



En posteriores versiones, en lugar de comprobar si un personaje implementa o no una interfaz, se comprobará que es el adecuado mediante su nombre. Esto se debe a un cambio en la forma de desarrollar el juego, ya que en adelante los objetos sólo podrán reaccionar ante un personaje concreto (el oso, el mono o la serpiente) y no ante una cualidad o característica de los mismos que podría ser complicada de percibir para el jugador.

En este apartado se han descrito prácticamente todos los elementos implicados y su código asociado ya que la mayoría de ellos se emplearán frecuentemente en el futuro, en los siguientes apartados se procurará dar una visión más general para tratar de no abrumar a los lectores con excesiva información.

A continuación se muestran distintas capturas acerca del funcionamiento del prototipo. Resulta imposible mostrar el movimiento de los personajes con imágenes estáticas, pero se espera que permitan formar una idea del resultado.


Ni el mono ni la serpiente pueden abrir la caja.


En cambio el oso sí puede. De forma similar se creó un cubo de basura que sólo pudiese abrir el mono.



7.2. Segundo prototipo

En este segundo prototipo se pretende implementar un “puzzle” completo, es decir, que el resultado de realizar distingas acciones con los personajes suponga la liberación de un animal. En este prototipo se liberará al último animal de todos, el rinoceronte, que posteriormente se encargará de romper el muro para permitir la fuga del resto de los animales, aunque no se realizará esa última escena por el momento.

Otro de los elementos que se pretenden desarrollar en este apartado consiste en el movimiento e inteligencia artificial del enemigo. Los enemigos en este juego serán los guardias del zoo, que se moverán aleatoriamente por la escena y perseguirán al jugador si lo detectan. Los enemigos detectarán al personaje cuando éste se encuentre a menos de cierta distancia de ellos, sólo perseguirán al personaje controlado en ese momento, ignorando al resto de los animales.

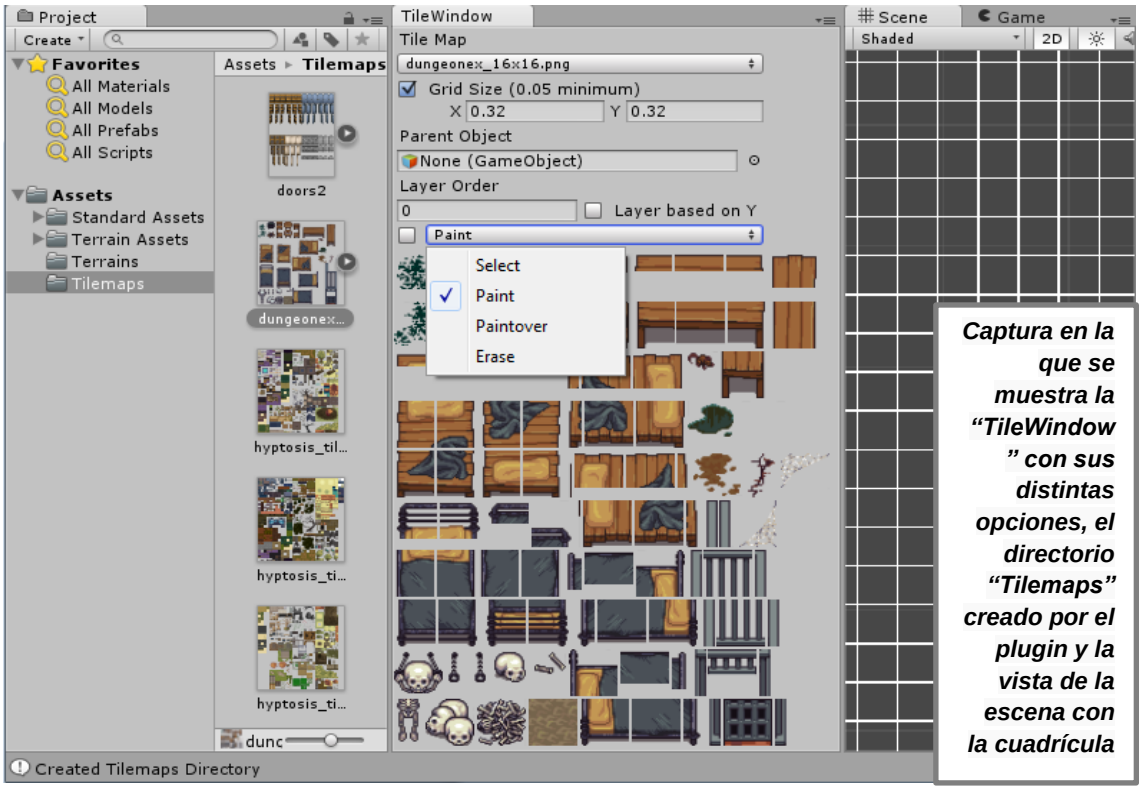
También se intentará indicar mediante imágenes en la UI cuál es el personaje activo (es decir, el que maneja el jugador) en cada momento.

Respecto a la apariencia, se empezará a trabajar con los elementos gráficos con el objetivo de ir explorando posibilidades para el diseño final del juego, en lugar de emplearlos como simples diferenciadores de algunos objetos de la escena.

7.2.1. La interfaz gráfica

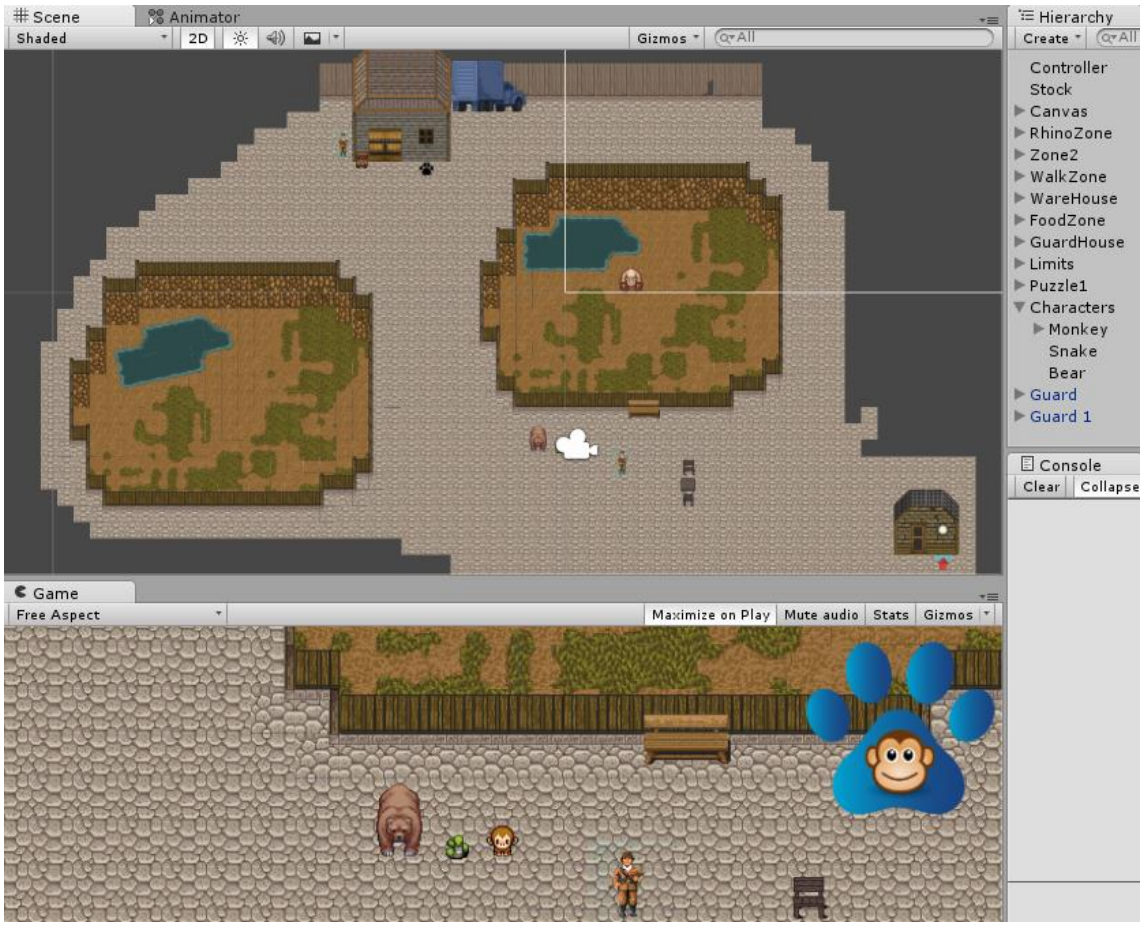
En este prototipo, se intentará generar una interfaz de juego utilizando los diseños clásicos de 32x32 o 64x64 píxeles. Los sprites que se verán en la interfaz se descargarán de distintas páginas con contenido gratuito de este tipo o serán creados por la propia alumna. Puesto que hasta la fecha Unity no dispone, que se sepa, de una herramienta que facilite la creación de mapas mediante tiles, se empleará un plugin que facilitará este mapeado. El plugin añade al proyecto una vista / ventana con nuevas opciones y una carpeta donde colocar los tiles que se vayan a utilizar. Con las herramientas de la nueva ventana podremos

- Mostrar una cuadrícula con celdas de diferentes tamaños que nos permiten ver dónde se colocarían los tiles
- Seleccionar un objeto padre para los sprites que se generarían y así poder organizarlos mejor (se necesitan bastantes tiles para componer una escena)
- Definir a qué profundidad se coloca cada tile para crear superposiciones
- Seleccionar un tile de la escena, lo que nos permitirá cambiar su posición
- Pintar un tile en la escena (existe una opción para pintar el tile sólo si no existe ya otro en la misma posición y con idéntica profundidad y otra opción en la que en caso de encontrar otro tile éste se borra y se coloca el nuevo)
- Borrar un tile de la escena (podemos borrar un conjunto de tiles seleccionándolos en la vista de jerarquía).



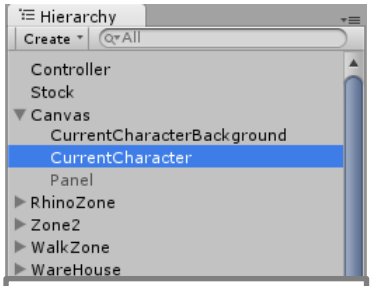
Captura en la que se muestra la "TileWindow" con sus distintas opciones, el directorio "Tilemaps" creado por el plugin y la vista de la escena con la cuadrícula

Con los recursos gráficos descargados y generados y la ayuda del plugin, se dibujaron los elementos de la escena que se pueden observar a continuación:



Para indicar cuál es el personaje activo se han utilizado las nuevas herramientas para la UI que aparecieron con la versión 4.6 de Unity y que, en relación a versiones anteriores, facilitaron en gran medida la tarea de crear todos aquellos componentes que no pertenecen realmente al “mundo del juego” sino que se encuentran entre éste y el usuario (indicadores de daño, vida, munición, botones, etc.)

El funcionamiento de esta nueva UI se basa en un objeto clave llamado *Canvas*. Todos los elementos que vayan a formar parte de la interfaz deben ser hijos de un objeto canvas o no serán visibles en el juego.



Captura de la vista de la jerarquía con el canvas y sus imágenes hijas para mostrar el jugador activo.

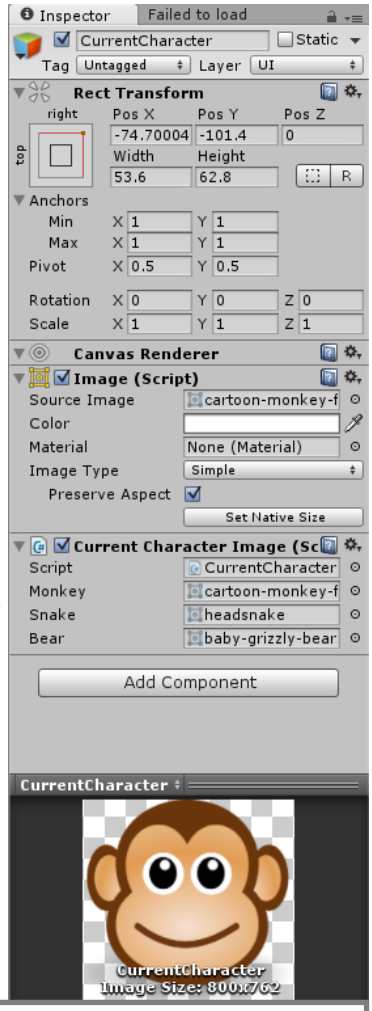
Para la tarea que nos ocupa, se han creado dos imágenes dentro de un canvas, una en la que se mostrará la imagen del personaje actual y otra que servirá de fondo para esta. A la imagen que mostrará los distintos personajes le añadiremos un script que contendrá los sprites de los tres animales posibles y un método público que permitirá cambiar la imagen en función a un nombre dado.

```
public void SetCurrentCharacterImage(string characterName){
    if (characterName == "Monkey"){
        GetComponent<Image>().sprite = monkey;
    }
    if (characterName == "Snake"){
        GetComponent<Image>().sprite = snake;
    }
    if (characterName == "Bear"){
        GetComponent<Image>().sprite = bear;
    }
}
```

A continuación, en el script de Controller, modificamos el método que se encargaba de rotar los personajes para que también indique cuál es el nuevo animal que debe aparecer en la UI. Para ello se añaden las siguientes líneas de código:

```
//Actualizamos la imagen del personaje actual
GameObject currentCharacterUI = GameObject.Find("CurrentCharacter");
currentCharacterUI.SendMessage("SetCurrentCharacterImage",character1.name);
```

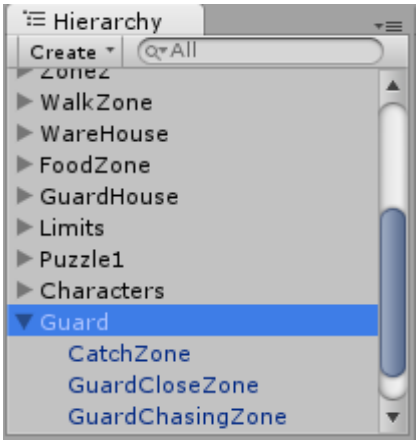
El método “Find” de la clase GameObject permite obtener una referencia a cualquier objeto de la escena a partir de su nombre. Si no se encuentra el objeto se devuelve *null*. Gracias a este método, no tenemos la necesidad de tener todos los objetos con los que vayamos a interactuar referenciados con variables previamente.



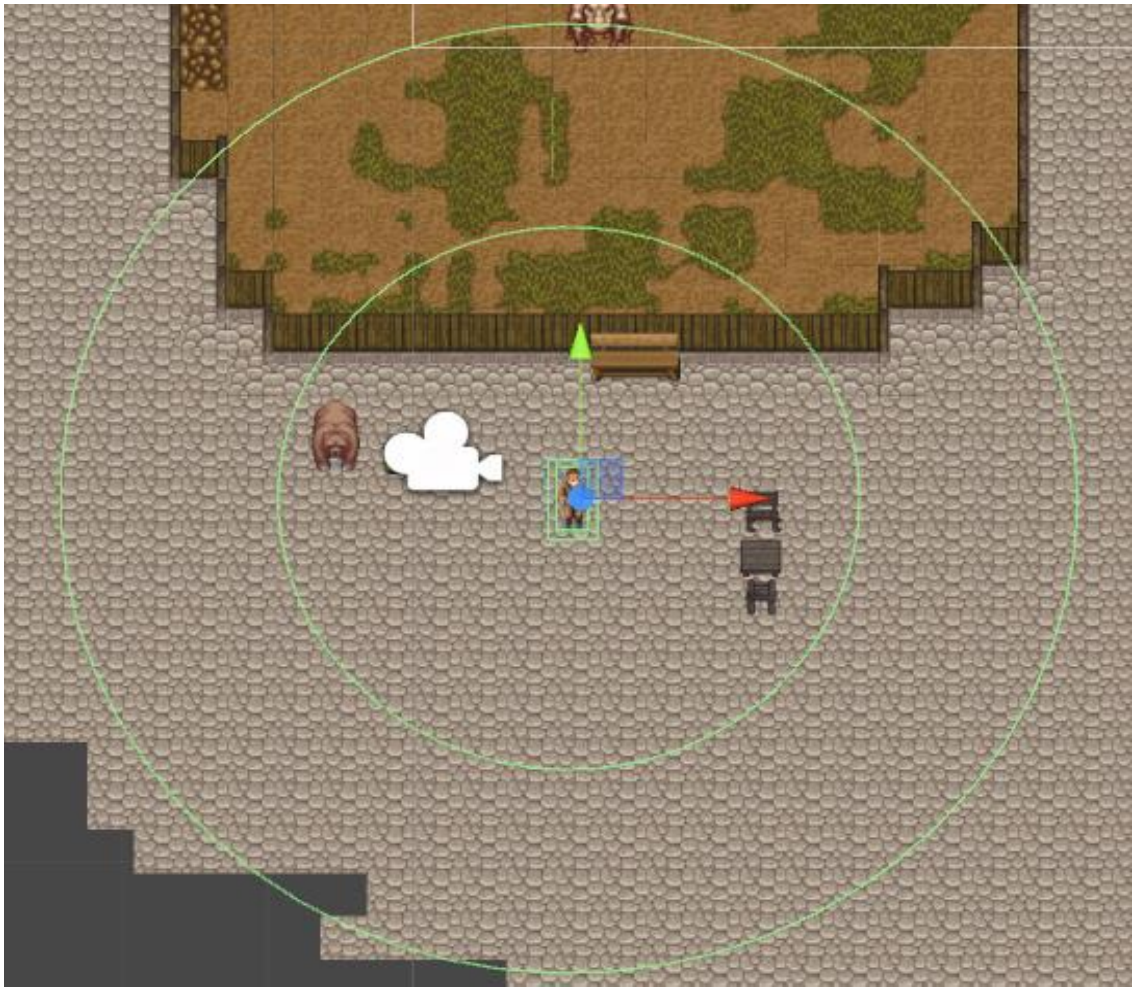
Captura de la vista del inspector para la imagen que contiene el jugador activo.

7.2.2. El enemigo

Para la creación del enemigo, en primer se ha creado un objeto con el sprite del guardia al que se le ha añadido un collider que le permitirá chocar con otros objetos con collider de la escena (vallas, bancos, edificios,…). A este objeto guardia se le han añadido tres hijos que se encargarán de controlar si el jugador se encuentra



a) cerca del guardia, pero aún no lo suficiente como para ser visto
b) tan cerca del guardia que éste lo ve
c) tan cerca del guardia que éste lo captura



En la imagen superior se muestran las tres zonas antes descritas del guardia. El oso está dentro del área de persecución del guarda, por lo que éste irá a por él. Si se encontrara en el círculo más exterior el guardia lo ignoraría y si se encontrara en la zona de captura (el rectángulo exterior del guardia) sería atrapado.

Para controlar el comportamiento del enemigo, se le ha añadido un script que tendrá dos variables públicas para indicar la velocidad por defecto del guardia (se pretende que esta velocidad se incremente cuando persiga al jugador) y la frecuencia con la que cambiará de dirección.

Definiremos ocho posibles direcciones en las que podrá moverse el guardia siempre que no esté persiguiendo al jugador:

- Norte, representada por el vector normalizado (0,1,0).
- Noreste, representada por el vector normalizado (1,1,0).
- Este, representada por el vector normalizado (1,0,0).
- Sureste, representada por el vector normalizado (1,-1,0).
- Sur, representada por el vector normalizado (0,-1,0).
- Suroeste, representada por el vector normalizado (-1,-1,0).
- Oeste, representada por el vector normalizado (-1,0,0).
- Noroeste, representada por el vector normalizado (-1,1,0).

Para que el guardia cambie de dirección aleatoriamente crearemos un método "RandomChangeDirection" que se ejecutará en base a la frecuencia indicada. En este tipo de situaciones, resulta muy útil la función "InvokeReapeating" de Unity, ya que se utiliza precisamente para ejecutar otra función cada cierto tiempo. Tiene tres parámetros, el primero indica el nombre de la función que se pretende ejecutar, el segundo cuándo empezará a ejecutarse y el tercero la frecuencia con la que se hará. En nuestro caso, utilizaremos InvokeRepeating dentro de Start de la siguiente manera:

```
InvokeRepeating("RandomChangeDirection",0,randomChangeDirectionFrequency);
```

RandomChangeDirection es el nombre de la función que queremos ejecutar repetidamente; con un cero en el segundo parámetro indicamos que la ejecución se haga al instante y en el tercer parámetro indicamos que la frecuencia será la que indique la variable pública randomChangeDirectionFrequency.

La dirección se almacenará en una variable global y privada que contendrá un vector normalizado como los que se describieron anteriormente. Esta variable será la que actualicemos en RandomChangeDirection.

```
private void RandomChangeDirection(){
    direction = new Vector3(Random.Range(-1,2),Random.Range(-1,2),0);
}
```

Random.Range(int x,int y) genera un numero aleatorio entre 'x' e 'y' excluyendo 'y'. En este caso los resultados posibles son -1, 0 y 1.

Pero una dirección no basta para mover al enemigo, necesitamos un método que lo haga andar de forma similar a como movíamos a nuestro personaje, por ello crearemos la función Walk.

```
private void Walk(){
    Rigidbody2D rigidBody = GetComponent<Rigidbody2D>();
    rigidBody.transform.position += direction * Time.deltaTime *speed;
}
```

La variable speed es una variable privada que se inicializa con el valor de la variable pública defaultSpeed.

Para evitar que el enemigo se quede estancado hasta que cambie de dirección en el caso de que choque contra algún objeto de la escena, ejecutaremos RandomChangeDirection cada vez que se produzca una colisión. Para ello emplearemos las funciones “OnCollisionEnter2D” y “OnCollisionExit2D” que funcionan de forma muy similar a “OnTriggerEnter2D” y “OnTriggerExit2D”.

```
void FixedUpdate () {
    Walk();
    if (collision){
        RandomChangeDirection();
    }
}

void OnCollisionEnter2D (Collision2D collision){
    this.collision = true;
}

void OnCollisionExit2D (Collision2D collision){
    this.collision = false;
}
```

La variable collision es una variable global y privada que se utiliza para controlar si ha habido una colisión o no.

El comportamiento del guardia cambiará cuando el jugador entre en su zona de persecución. Será la propia zona la que detecte mediante funciones “OnTrigger” la entrada o salida del jugador en la zona y la que indicará si se debe perseguir o no al jugador.

Cuando persiga al jugador, el guardia debe dejar de moverse aleatoriamente y hacerlo en la dirección en la que está el jugador. Se cancelará por lo tanto la invocación actual al método RandomChangeDirection y en su lugar se ejecutará cada segundo “UpdateDirectionToChasePlayer”, una función que actualiza la dirección para apuntar al jugador. La posición del jugador se obtiene mediante su collider que a su vez es proporcionado por parámetros al ejecutar la función Chase. Durante la persecución la velocidad del guardia aumenta un 50%. Si el jugador sale de la zona de persecución se ejecuta la función DontChase para volver al comportamiento anterior.

```
public void Chase(Collider2D currentPlayer){
    chasing = true;
    CancelInvoke();
    this.target = currentPlayer;
    speed *= 1.5f;
    InvokeRepeating ("UpdateDirectionToChasePlayer",0,1);
}

private void UpdateDirectionToChasePlayer(){
    direction = (target.transform.position - transform.position).normalized;
}

public void DontChase(){
    chasing = false;
    speed = defaultSpeed;
    CancelInvoke();
    InvokeRepeating("RandomChangeDirection",0,randomChangeDirectionFrequency);
}
```

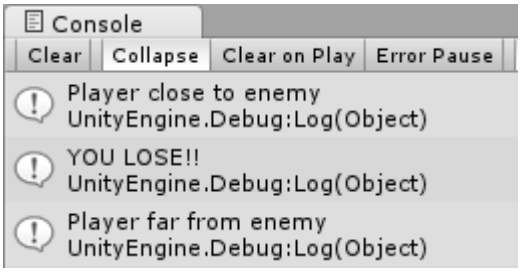
Dado que el jugador puede cambiar de personaje en cualquier momento, el enemigo tendrá que saber cuándo se producen estos cambios para cambiar a su vez de objetivo en su persecución. Para ello crearemos un nuevo método en el controlador que avise a todos los enemigos cuando se produzca un cambio de personaje.

```
private void ChangeAlert(){
    GameObject[] enemyList = GameObject.FindGameObjectsWithTag("Enemy");

    for (int i = 0; i < enemyList.Length; i++){
        enemyList[i].SendMessage("CharacterChanged", character1.GetComponent<Collider2D>());
    }
}
```

Para poder avisar a todos los enemigos de la escena, se emplea la función “FindGameObjectWithTag” y se les asigna en la vista del inspector el tag “Enemy” a cada uno de los enemigos. En el script del enemigo, se crea el método “CharacterChanged”.

```
public void CharacterChanged(Collider2D target){
    this.target = target;
}
```

De momento, las zonas que indican si el jugador se encuentra cerca del enemigo o si es capturado por él sólo advierten de esto mediante la consola de Unity. En ocasiones posteriores se implementará la captura real del jugador, con su consiguiente pérdida de vida, regreso a otro punto del juego, etc.



Por poner un ejemplo, el contenido del script de la CatchZone sería el que se muestra a continuación:

```
using UnityEngine;
using System.Collections;

public class Catch : MonoBehaviour {

    void OnTriggerEnter2D (Collider2D other){
        if (other.tag == "CurrentPlayer"){
            Debug.Log ("YOU LOSE!!");
            //Stop Game
            //Lose animation
            //Restart game / go to the previous saved
        }
    }
}
```

Con la intención de que el jugador sepa más claramente cuándo se encuentra en la zona de persecución del guardia y cuándo no, se decidió implementar un nuevo objeto que sería también hijo del guardia y que pretendería emular el haz de luz de una linterna. Para simular el haz de luz se utilizó un sprite con una imagen cónica de color amarillo que tenía una cierta transparencia, por lo que permitía ver los elementos situados detrás de él. También se indicó que este sprite debería pivotar por su base, la parte más estrecha del haz, para que el comportamiento cuando la linterna rotase fuese lo más natural posible.

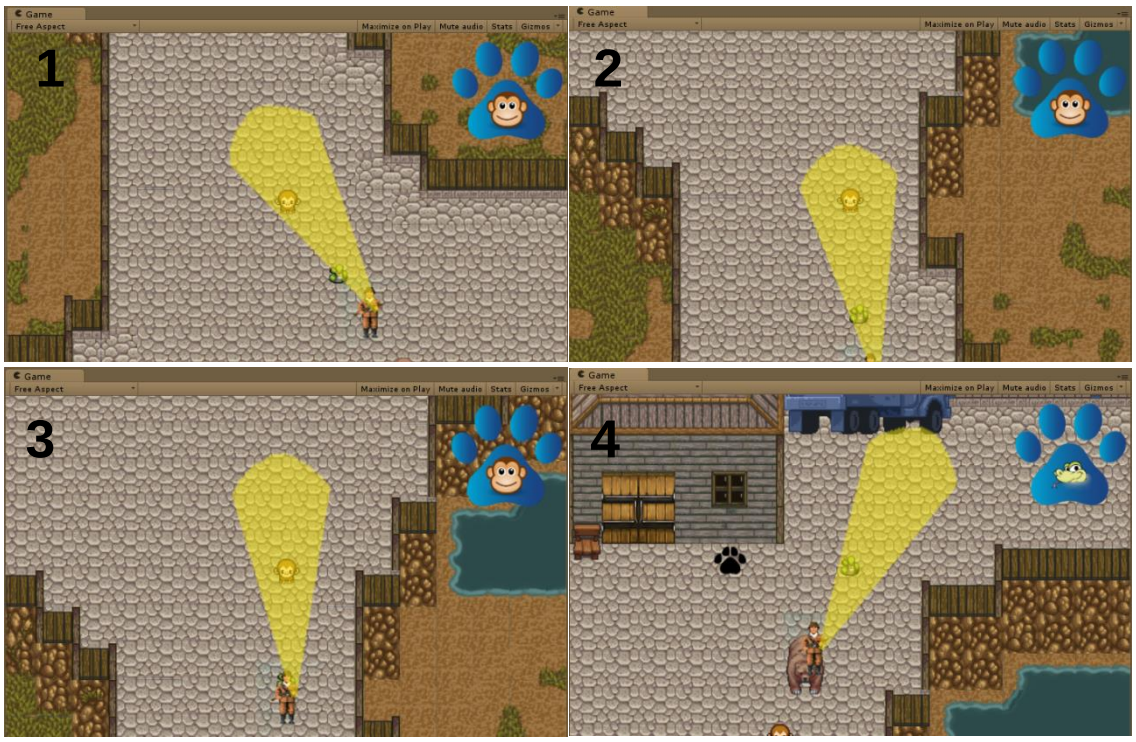
Se añadieron dos variables públicas al script del guardia, la primera sería la propia linterna y la segunda la velocidad a la que la interna rotaría, ya que si la linterna cambiaba instantáneamente de dirección al tiempo que lo hacía el guardia el resultado quedaba un poco extraño cuando el guardia paseaba. En cambio, cuando el guardia está persiguiendo al jugador sí se entienden cambios bruscos en la dirección del haz de luz de la linterna, tratando de enfocar al jugador en todo momento.

Para el comportamiento de la linterna, crearemos un método que se encargue actualizar la rotación de ésta en grados (eulerAngles) en base a la dirección que siga el guardia. Éste método será invocado por todos aquellos que actualicen la dirección en la que se mueve el guardia, es decir, RandomChangeDirection y UpdateDirectionToChasePlayer.

También utilizaremos una corrutina para hacer ese cambio en la rotación de forma más suavizada. Si ya existiese una corrutina actualizando la rotación, la detendremos e iniciaremos otra con los nuevos datos, a no ser que se estuviese persiguiendo al jugador, en cuyo caso la actualización será automática.

```
private void UpdateFlashlightRotation(){
    float rotation = 0;
    //Linterna hacia abajo
    if (direction.y < -0.5) {
        rotation +=180;
        rotation += direction.x * 45;
    }
    //Linterna hacia arriba
    if (direction.y > 0.5) {
        rotation -= direction.x * 45;
    }
    //Linterna hacia un lado
    if (direction.y >= -0.5 && direction.y <= 0.5) {
        rotation -= direction.x * 90;
    }

    if (updatingFlashlightRotation) StopCoroutine("UpdateFlashlightRotationSlowly");
    float initialRotation = flashlight.transform.eulerAngles.z;
    if (!chasing) StartCoroutine(UpdateFlashlightRotationSlowly(initialRotation, rotation));
    else flashlight.transform.eulerAngles = new Vector3(0,0,rotation);
}

IEnumerator UpdateFlashlightRotationSlowly(float rotation, float finalRotation){
    updatingFlashlightRotation = true;
    if (finalRotation < 0){
        finalRotation+=360;
    }

    int direction = 1;
    if (finalRotation - rotation > 180){
        finalRotation -=360;
        direction = -1;
    }

    while (finalRotation > rotation){
        rotation += flashlightRotationSpeed*direction;
        flashlight.transform.eulerAngles = new Vector3(0,0,rotation);
        yield return null;
    }
    flashlight.transform.eulerAngles = new Vector3(0,0,finalRotation);
    updatingFlashlightRotation = false;
}
```

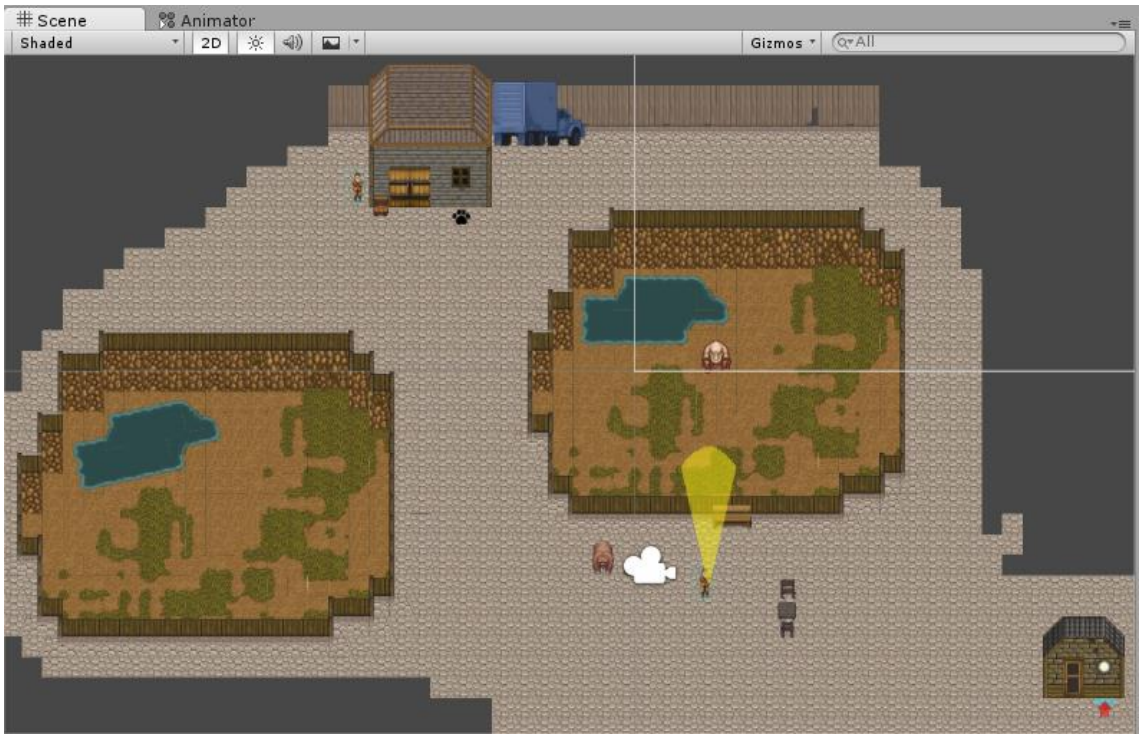
A continuación se muestran capturas de una persecución con linterna.



7.2.3. El puzzle

El puzzle final consiste en liberar al rinoceronte para que éste nos ayude a escapar del zoo. Para liberar al rinoceronte se arrojarán a su hábitat pesadas cajas que formarán una pila por la que éste podrá subir. Las cajas se obtendrían en el almacén y sólo podrían ser cargadas por el oso. Para entrar al almacén, cuya puerta está inicialmente cerrada, deberemos hacer un agujero aprovechando una zona más desgastada de la pared y por el que sólo se podrá colar la serpiente. Para hacer el agujero necesitaremos un martillo que sólo podrá manejar el mono. Es decir, la secuencia de pasos a seguir sería

- Conseguir el martillo
- Abrir el agujero en la pared del almacén
- Abrir la puerta del almacén
- Cargar cajas de una en una y arrojarlas al hábitat del rinoceronte hasta conseguir una pila lo suficientemente alta.



El martillo se encontrará en la caseta del guardia, situada al sureste del mapa. El mono podrá entrar por la ventana de la misma, pero sólo él. El interior de la caseta del guarda ha supuesto la creación de una nueva escena y la posibilidad de poner en práctica los cambios de nivel.

En apartados anteriores se comentó cómo se añadían las escenas al juego con el BuildSettingd. Cuando añadimos una escena en el BuildSetting, ésta tiene un número asignado según su orden. El

orden tiene especial importancia cuando hablamos de la escena con orden cero, ya que ésta será la primera en cargarse cuando se inicie el juego. Respecto al resto de las escenas, tendremos que saber el orden que tienen asociadas para poder cargarlas desde los scripts.

Para realizar el cambio de escena desde el zoo a la casa del guardia y viceversa, se crearon sendas zonas para detectar el paso del personaje activo por encima de ellas y cargar el nivel que correspondiese en cada caso. Esas zonas tendrían el siguiente script:

```
using UnityEngine;
using System.Collections;

public class ChangeScene : MonoBehaviour {
    public int level;

    void OnTriggerEnter2D (Collider2D other){
        if(other.tag == "CurrentPlayer"){
            Application.LoadLevel(level);
        }
    }
}
```

Dentro de la casa del guardia, se creó otra zona de detección, con el sprite de una huella, para indicar dónde se debía interactuar. La huella cambia de color y “brilla” cuando el personaje se coloca sobre ella, y desaparece cuando ya se ha cogido el martillo.



Para poder saber que hemos cogido el martillo en esta escena, necesitamos poder conservar esa información para cuando regresemos a la anterior, por ejemplo en un objeto “stock”. En general, los objetos son completamente independientes entre las escenas y sólo existen en la suya propia, pero es posible indicar mediante scripting que no queremos que un objeto (y en consiguiente tampoco sus hijos) se destruya al cambiar de una escena a otra. El método que impide que un objeto se destruya cuando cargamos un nivel nuevo es “DontDestroyOnLoad” al que se le pasa como parámetro el objeto.

El método DontDestroyOnLoad tiene algunas pegas, sobre todo si planeamos ir saltando de nivel en nivel y regresamos al nivel en el que se crea el objeto que no queremos destruir. El problema de esto es que si creamos un objeto que perdura entre escenas y regresamos a la escena donde se crea, tendremos dos objetos de ese tipo, ya que éste se genera de nuevo independientemente de que nosotros indiquemos que el primero que se creó no se destruya.

Para evitar tener varias copias del mismo objeto, comprobaremos antes que nada si ya existe una instancia de ese objeto en la escena cada vez que se cargue una escena nueva. Esto lo haremos apoyándonos en una variable estática y el método “Awake”. Awake es un método que se ejecuta automáticamente antes incluso que Start cuando el script está cargado y aunque no esté activado. Es el método perfecto para controlar si ya hay una estancia de una cierta clase (en este caso Stock) y en caso de que así sea destruir inmediatamente el nuevo objeto que se acaba de crear.

```
public static Stock stockInstance;

void Awake(){
    //Si stockInstance != null ya hay una instancia de Stock asi que destruimos la actual
    if (stockInstance){
        DestroyImmediate(this.gameObject);
    }else{
        DontDestroyOnLoad(this.gameObject);
        stockInstance = this;
    }
}
```

El script Stock contiene una variable booleana hammerTaken cuyo valor está inicialmente a false, pero que pondremos a true cuando se coja el martillo. Como el stock no se destruye con los cambios de escena, esta información permanecerá.

Una vez se tiene el martillo, es posible abrir un agujero en la pared del almacén.



Una vez hecho el agujero, la serpiente podrá colarse por él para abrir la puerta desde dentro.


Sólo el oso podrá cargar las cajas del almacén hasta el hábitat del rinoceronte.

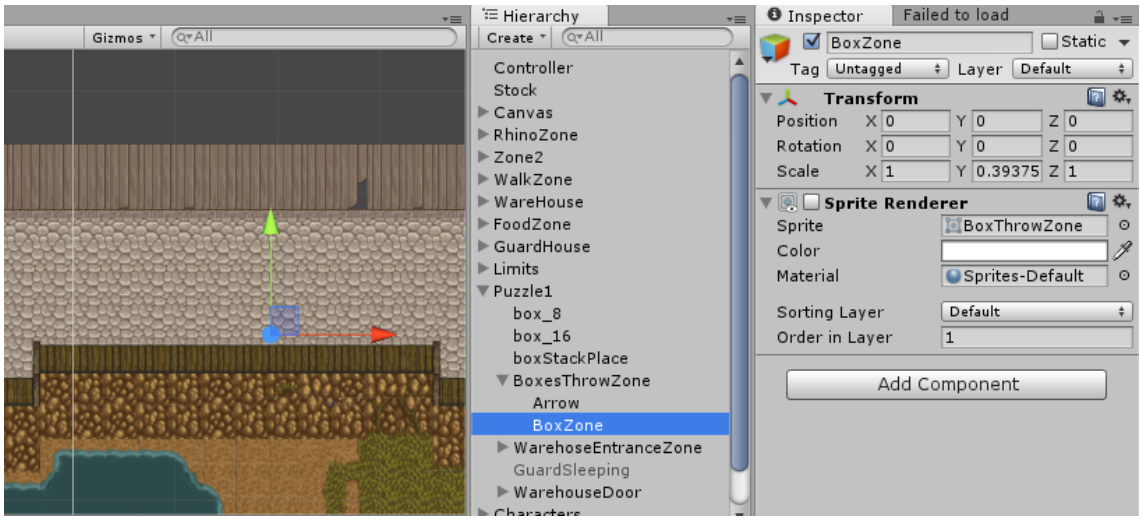
Cuando el oso porta una caja, no se puede rotar entre los personajes y aparece un área para indicar dónde debe dejarla. El área desaparece cuando el oso arroja la caja.

Las cajas se irán acumulando hasta formar una pila de altura suficiente y que el rinoceronte pueda escapar.

La mayoría de las acciones y funciones necesarias para crear este puzzle ya se han explicado anteriormente, así que se nos centraremos en las novedades.

La forma de hacer que los sprites sean visibles en algunos momentos y en otros no es activándolos y desactivándolos según convenga mediante un checkbox en la vista del inspector de Unity o mediante scripting obteniendo el componente SpriteRenderer del objeto e indicando si se habilita o no.

En el caso de la zona de lanzamiento de cajas, sus sprites aparecen deshabilitados inicialmente y pueden ser activados mediante el método “Activate” que se encuentra en el script del objeto “BoxesThrowZone”.



En este caso, en lugar de pasar las referencias a los objetos cuyos sprites se quieren habilitar/deshabilitar o buscarlos mediante el método GameObject.Find, se ha decidido acceder a ellos mediante la jerarquía. Ya que el objeto BoxesThrowZone es el padre de los otros dos, podemos obtener los componentes SpriteRenderer mediante la instrucción transform.GetChild(int childIndex).GetComponent<SpriteRenderer>(). También en el método Activate, habilitaremos/deshabilitaremos el collider de BoxesThrowZone para que sólo detecte al personaje activo si éste lleva una caja.

```
public void Activate(bool activate){
    if (activate){
        GetComponent<Collider2D>().enabled = true;
        transform.GetChild(0).GetComponent<SpriteRenderer>().enabled = true;
        transform.GetChild(1).GetComponent<SpriteRenderer>().enabled = true;
    }else {
        GetComponent<Collider2D>().enabled = false;
        transform.GetChild(0).GetComponent<SpriteRenderer>().enabled = false;
        transform.GetChild(1).GetComponent<SpriteRenderer>().enabled = false;
    }
}
```

En el controlador, permitiremos activar o desactivar la rotación entre personajes mediante los siguientes métodos:

```
public void ActivateChange(){
    changeEnabled = true;
}

public void DisableChange(){
    changeEnabled = false;
}
```

Y en el almacén controlaremos la creación de las cajas y lo que ocurre si se suelta una de ellas. Para poder crear una caja durante el juego, primero tendremos que crearnos un “prefab”. Los prefabs son objetos que forman parte de los assets del proyecto y que el usuario de Unity puede crear a partir de objetos de la escena arrastrando éstos a la carpeta Assets. En nuestro caso, colocamos el sprite de una caja en la escena y luego arrastramos el objeto con el sprite a la carpeta de lso assets para crear nuestro prefab “box”. A continuación, en el script del almacén crearemos una variable pública de tipo GameObject que contendrá nuestro prefab de la caja.

Para instanciar objetos a partir de prefabs se utiliza el método “Instantiate” que posee tres parámetros: el prefab a instanciar, la posición y la rotación. En nuestro caso la caja estará posicionada ligeramente más arriba que la posición del oso (para que parezca que la lleva encima) y no tendrá ninguna rotación (Quaternion.identity se corresponde con la “no rotación”, es decir, el objeto está perfectamente alineado con el mundo). Utilizaremos una variable privada de tipo GameObject llamada box para guardar una referencia a la caja recién instanciada y poder colocarla posteriormente como hija del objeto oso. Para poder guardar la referencia a la instancia en la variable “box”, tendremos que hacer un cast de la instancia a GameObject.

```
//Instanciar una caja
box = Instantiate(boxPrefab, currentCharacterCollider.transform.position
                  +(new Vector3(0,0.25f,0)), Quaternion.identity) as GameObject;
box.transform.parent = currentCharacterCollider.transform;
```

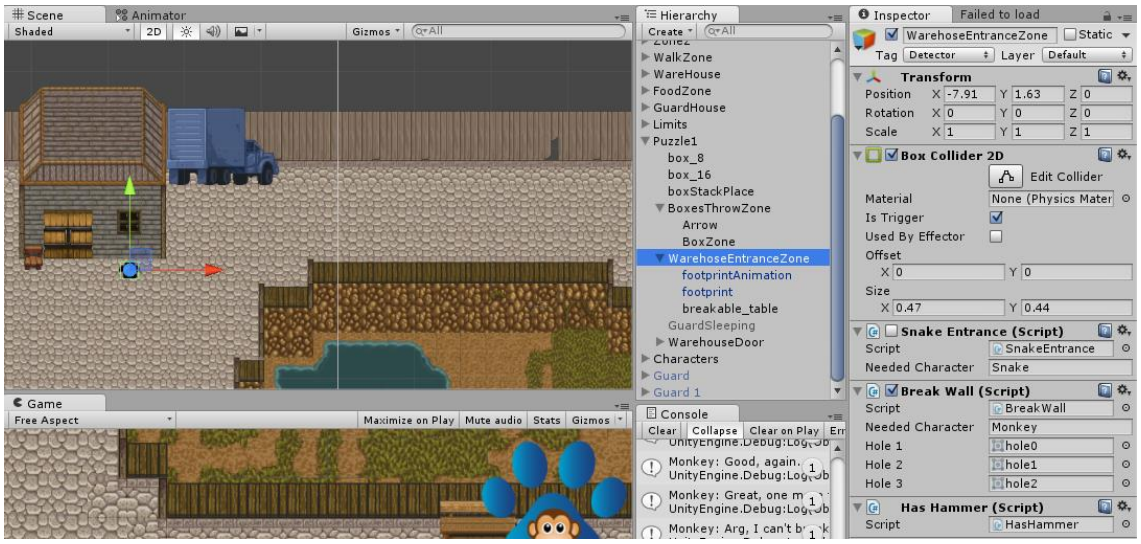
El método encargado controlar lo que ocurre si se suelta una caja es DropBox.

```
public void DropBox(){
    Destroy (box);
    hasNoBox = true;

    //Permitimos de nuevo cambiar de personaje
    GameObject.Find("Controller").SendMessage("ActivateChange");

    //Desactivamos el collider de las zonas de lanzamiento de cajas
    GameObject[] throwZones = GameObject.FindGameObjectsWithTag("BoxThrowZone");
    for (int i = 0; i < throwZones.Length; i++){
        throwZones[i].SendMessage("Activate", false);
    }
}
```

Aunque actualmente sólo existe una zona de lanzamiento de cajas, no se descarta que pueda haber más en un futuro, por lo que se indica a estas que se activen o desactiven por tag.

Los scripts también se pueden habilitar o deshabilitar, en nuestro caso, dado que las localizaciones de la zona donde debe abrir el agujero el mono y la de por donde debe colarse la serpiente coinciden, se han utilizado para ambos comportamientos. Cuando el hueco no ha sido abierto, el script de la serpiente permanece deshabilitado, y cuando se abre, se habilita ese script y se deshabilitan los que se encargan de la pared del almacén (el script hasHammer es el encargado de preguntarle al stock si se tiene el martillo, una vez abierto el hueco, el script es innecesario en ese objeto).

```
this.GetComponent<SnakeEntrance>().enabled = true;
this.GetComponent<HasHammer>().enabled = false;
this.GetComponent<BreakWall>().enabled = false;
```



El script SnakeEntrance controla la apertura o cierre de la puerta del almacén, aunque es el propio almacén el que conoce y coloca los dos sprites disponibles en la puerta (abierta o cerrada).

El método que controla la apertura o cierre de la puerta en SnakeEntrance es el que se muestra a continuación:

```
void OpenCloseWarehouseDoor(){
    //Si se invoca a este metodo significa que el CurrentCharacter esta en la zona y se ha
    // pulsado el boton de "accion"
    //Si el current character es el que necesitamos realizamos las acciones pertinentes
    if (rightCurrentCharacter){
        if (doorIsOpen){
            Debug.Log ("Snake: I'll close the door");
            GameObject.Find("WarehouseDoorZone").SendMessage("CloseDoor");
            doorIsOpen = false;
        }else{
            Debug.Log ("I'm a snake, I can go in");
            GameObject.Find("WarehouseDoorZone").SendMessage("OpenDoor");
            doorIsOpen = true;
        }
    }else{
        Debug.Log ("I'm too big to go in");
        GameObject.Find("thinking").SendMessage("NotAllowedCharacter", currentCharacterCollider.name);
    }
}
```

Y los métodos que invoca de WarehouseDoorZone son los siguientes:

```
public void OpenDoor(){
    isOpen = true;
    transform.parent.gameObject.GetComponent<SpriteRenderer>().sprite = openedDoor;
    transform.GetChild(1).GetComponent<SpriteRenderer>().enabled = true;
}

public void CloseDoor(){
    isOpen = false;
    transform.parent.gameObject.GetComponent<SpriteRenderer>().sprite = closedDoor;
    transform.GetChild(1).GetComponent<SpriteRenderer>().enabled = false;
}
```

Los métodos OpenDoor y CloseDoor son prácticamente idénticos y bien podrían sustituirse por un solo método con un parámetro booleano que indicara si se quiere abrir o cerrar la puerta. Como de momento sólo se están haciendo pruebas, por ahora lo dejaremos para mejoras futuras.

Durante la creación de este prototipo se realizaron gran cantidad de experimentos y los resultados de la gran mayoría de ellos no se llevarían a la práctica en siguientes versiones por cuestiones de tiempo y prioridades. Es el caso de las huellas de los animales. Se diseñó un pequeño ejemplo de cómo podría conseguirse que los animales dejaran huellas allá donde fueran (las huellas irían desapareciendo poco después de ser creadas).

También se contempló la posibilidad de indicar mediante la interfaz que un animal no era el adecuado para cierta acción, como en el caso del mono a la hora de colarse por el agujero o de la serpiente a la hora de llevar cajas.

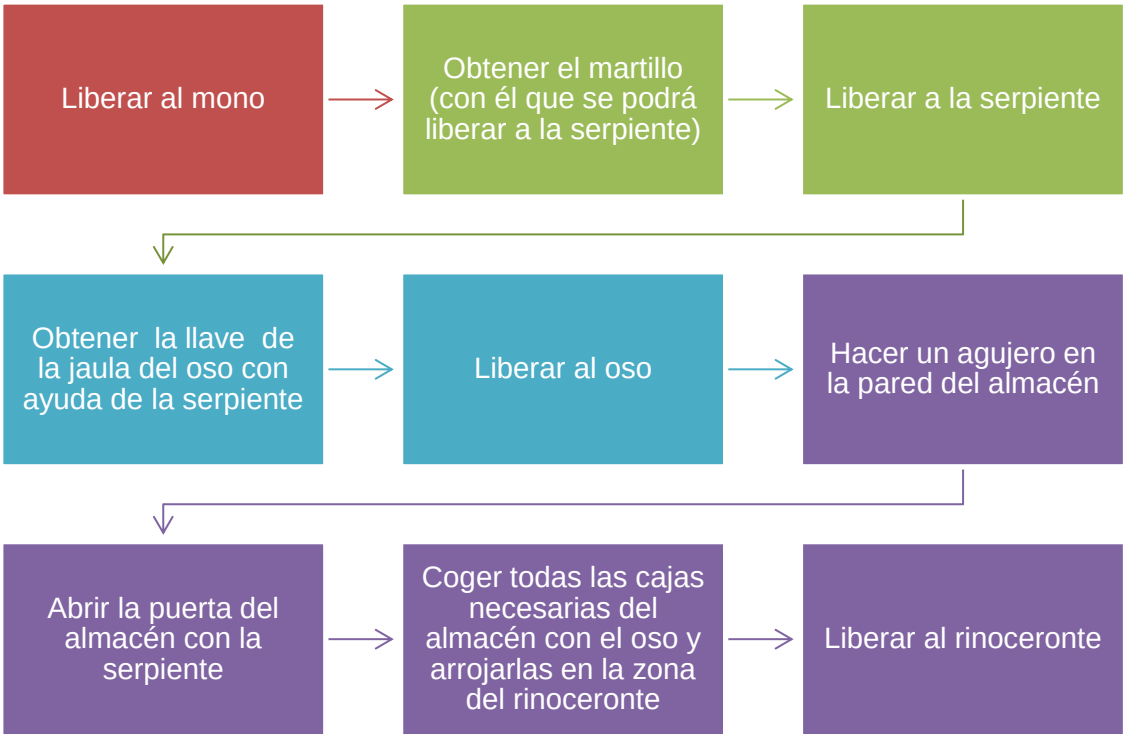
Otro de los experimentos consistió en probar las animaciones al tiempo que se barajaba la posibilidad de añadir un guardia custodiando el almacén y que fuera necesaria otra acción más para distraerlo y poder hacerse con las cajas.

El guardia de las imágenes fue dibujado por la alumna sin seguir el estilo de pixelado del resto de la escena y por eso contrasta viva y negativamente con el resto del entorno.

En definitiva, aunque hayamos llamado a este apartado “Segundo prototipo”, en realidad se trata de una variedad de prototipos con sus distintas pruebas, su toma de decisiones y pivotaciones, y con los que se ha pretendido explorar las diferentes formas de obtener resultados a fin de adquirir conocimientos y destreza de cara a la demo final.

7.3. Definición de los assets finales y tercer prototipo

En el tercer prototipo se pretende crear ya una versión “completa” del juego, con todos los puzzles necesarios para liberar a los tres personajes principales y finalmente al rinoceronte. La secuencia de acciones normal, separando cada puzzle por colores, sería la que se muestra a continuación:



Aunque ésta sería la secuencia de acontecimientos normal, como hemos visto en el apartado anterior para hacer el agujero en la pared del almacén sólo necesitamos al mono con el martillo, y para abrir la puerta del almacén sólo necesitamos que el agujero esté hecho y a la serpiente para colarse por él. Por lo tanto, nada impediría que el jugador obtuviera el martillo, hiciera el agujero en el almacén y luego fuera a liberar a la serpiente, o que se llegara a abrir la puerta del almacén sin tener aún al oso para llevar las cajas.

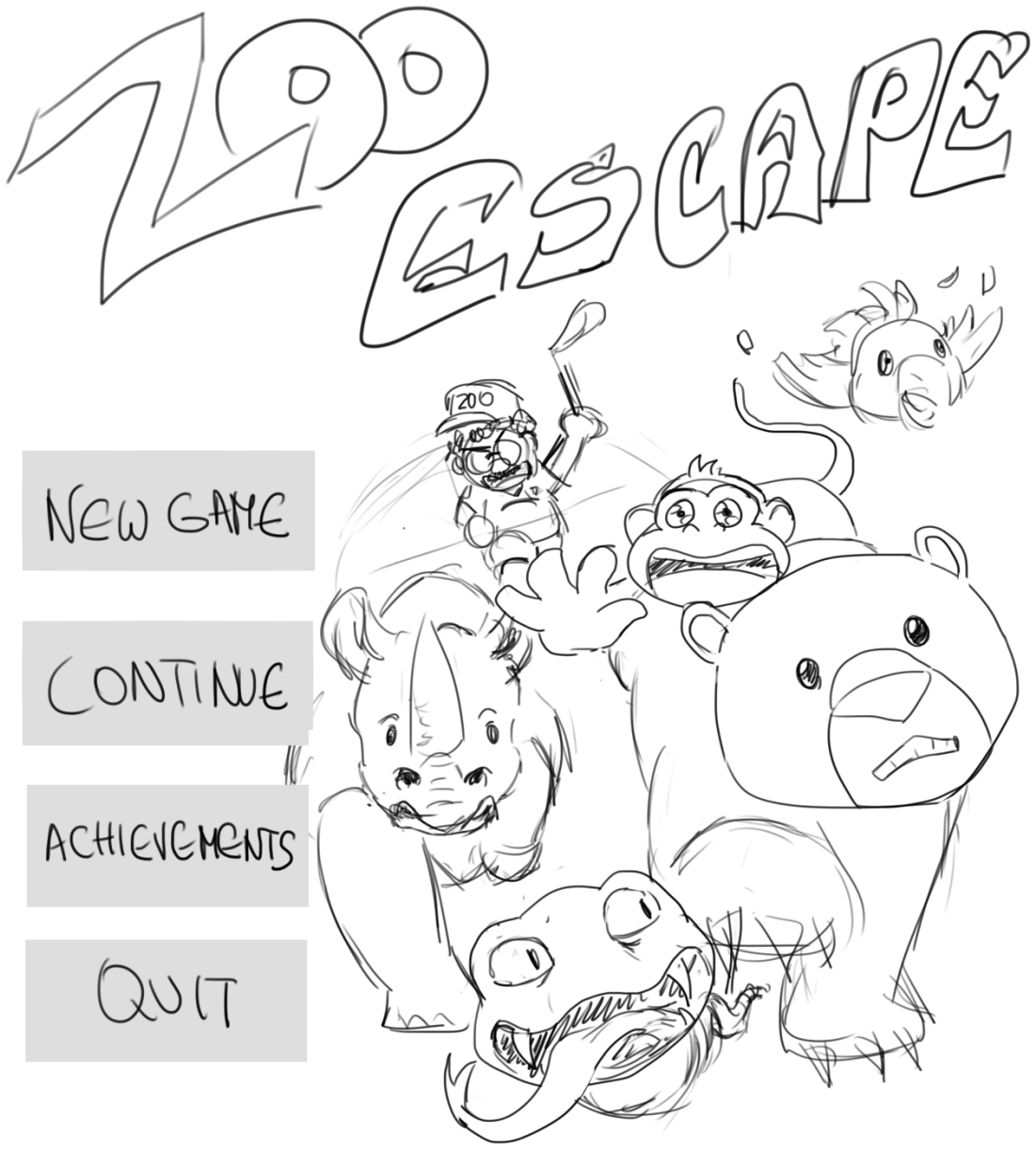
7.3.1. Los assets

En este punto del desarrollo y teniendo más o menos claro cuáles serían los puzles del juego, se tomaron algunos días para concretar exactamente los assets que se necesitarían para la implementación final.

Se definió un posible menú de inicio que contendría un fondo con lo que sería la portada del juego y una serie de botones que permitirían realizar las siguientes acciones:

- Iniciar una nueva partida
- Continuar una partida anterior
- Ver los logros
- Ver los controles
- Salir del juego

Si bien sería deseable implementar las acciones asociadas a todos esos botones, es probable que las limitaciones en tiempo impidan que se lleven todas a cabo, inicialmente nos concentraríamos la acción de en iniciar una nueva partida y salir del juego. Para la pantalla de inicio, se dibujó el siguiente diseño previo para que sirviera de referencia (aunque por despiste se omitió el botón de los controles).



Al margen del Inicio, otra de las opciones con las que se desea contar en el juego es con la opción de pausar éste. El juego se pausaría pulsando una tecla, por ejemplo “Esc”, y mientras se estuviera en este estado, ni los personajes ni los enemigos se podrían mover. También se mostraría un mapa del zoo y se tendría la opción de regresar al menú de inicio, salir del juego o regresar a él volviendo a pulsar la tecla “Esc” en este caso.

Los assets que se considera que serán necesarios para la realización de este juego se muestran en la tabla siguiente.

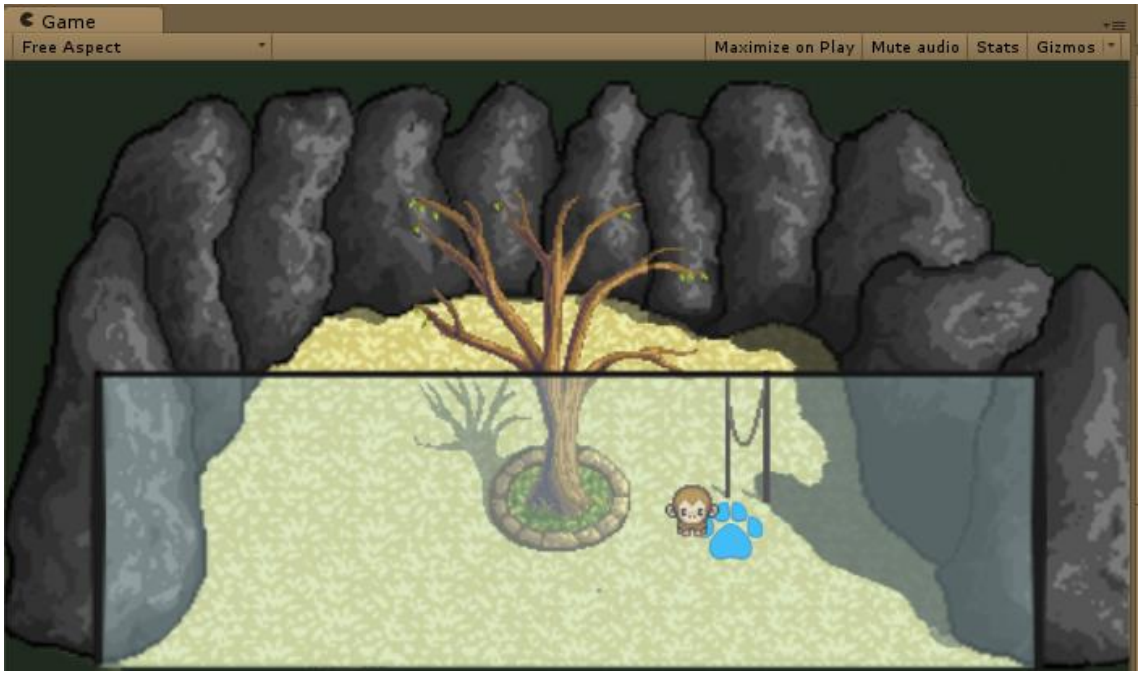
Elementos de la UI de Unity	Para el menú de inicio, el evento “pause” y un posible Stock gráfico: • **Canvas** • **Paneles** • **Imágenes** (fondo de inicio, personaje activo, vidas,…) • **Botones**
Gráficos	Gráficos que muestren las distintas zonas el zoo: • **Hábitat del mono** • **Hábitat de la serpiente** • **Hábitat del oso** • **Hábitat del rinoceronte** • **Casa del guardia** • **Almacén** • **Caminos** • **Otros hábitats decorativos, zona de souvenirs,…** Sprites de los distintos personajes: • **Mono** • **Serpiente** • **Oso** • **Rinoceronte** • **Enemigo** • **Otros animales** que compartan hábitat con los protagonistas. Objetos y elementos decorativos: • **Árboles** • **Bancos** • **Papeleras** • **Rocas** • **Basura dispersa** (chicles, latas de refrescos, papeles,…) • **Martillo** Gráficos de la UI: • **Corazones para las vidas** • **Mochila para los objetos** • **Imagen del personaje activo**
Scripts	• **Control de los personajes** (Movimiento y rotación) • **Control de los enemigos** (Movimiento e IA) • **Control de los distintos elementos de la UI** (Botones, paneles,…) • **Control del stock** • **Control del evento “pause”** • **Control de cada uno de los puzzles** • **Control del menú de inicio** • **Control de los apartados de logros y crontroles del juego**
Animaciones	• **Animación del movimiento de los distintos personajes** • **Animación para cada una de las liberaciones**
Sonidos	• **Música de ambiente** (Menú de inicio y resto del juego) • **Aviso al coger un objeto** • **Sonido de pasos** • **Sonido de golpe** • **Sonido de cristales rotos**

7.3.2. Tercer prototipo

En el segundo prototipo ya se implementó lo que se correspondería con el puzzle 4, liberar al rinoceronte. En este prototipo se pretende implementar los puzzles 1, 2 y 3 que se corresponden con la liberación del mono, la serpiente y el oso respectivamente.

Inicialmente, sólo controlaremos al personaje “Mono” por lo que no podremos hacer ninguna rotación. En los niveles en los que sólo hay un personaje, como en el hábitat del mono y la casa del guarda, se emplea para el movimiento de éste un script que es una versión sencilla del que se ha visto en apartados anteriores, sin la opción de rotar entre personajes.

El mono comenzará en su hábitat, del que tendrá que escapar balanceándose con la cuerda que se encuentra en un columpio para saltar a una de las rocas que bordean la zona y escapar por allí.



Desde el dibujo de la huella que aparece junto al columpio, el jugador podrá subirse a éste y comenzar a balancearse. Pulsando la tecla/botón de acción (E en el teclado, clic primario en el ratón) el mono se soltará y se dará una de las siguientes situaciones:

- El mono se encuentra balanceándose hacia atrás en el momento de soltarse y se choca contra el árbol.
- El mono se encuentra en el centro del columpio y se baja de él.
- El mono se encuentra balanceándose hacia delante en el momento de soltarse, pero no tiene fuerza suficiente para alcanzar el borde de la roca y se choca.
- El mono se encuentra balanceándose hacia delante en el momento de soltarse y tiene el impulso suficiente para llegar a subirse a la roca y escapar.

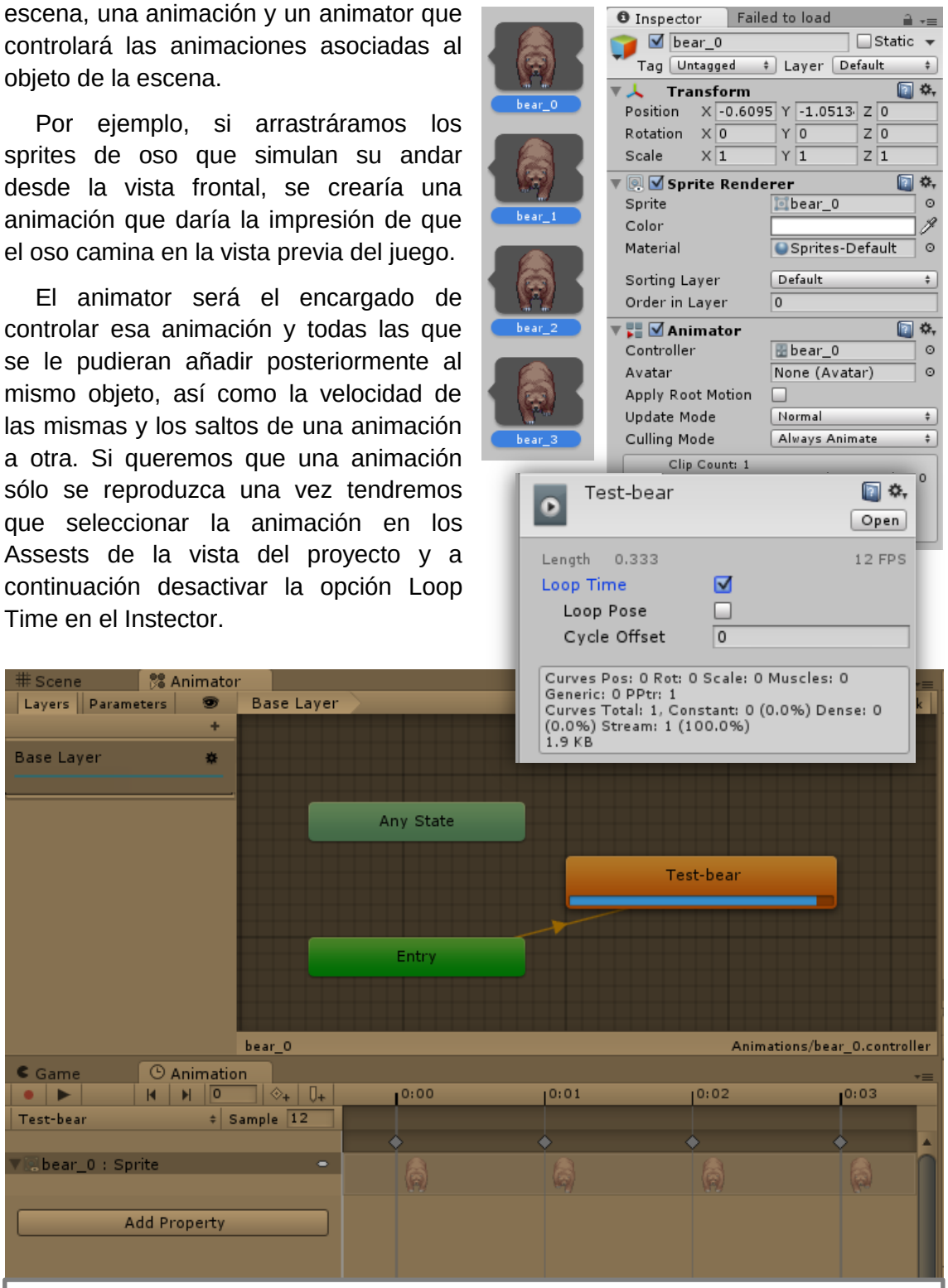
Para implementar este puzzle, se ha decidido empezar a jugar con las animaciones.

Crear una animación en Unity no es complicado, basta con arrastrar al mismo tiempo una secuencia de sprites hasta la escena. Con ello se creará un objeto en la

escena, una animación y un animator que controlará las animaciones asociadas al objeto de la escena.

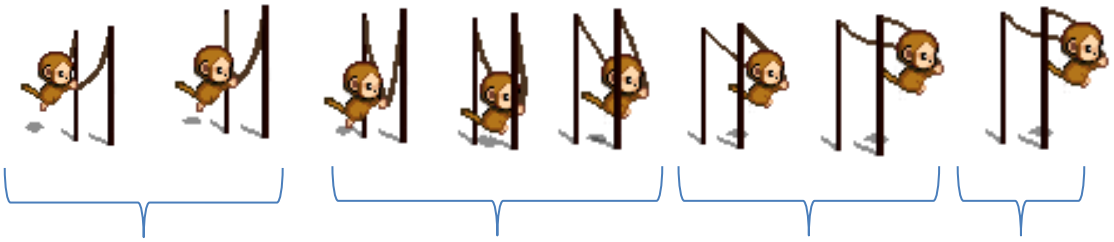
Por ejemplo, si arrastráramos los sprites de oso que simulan su andar desde la vista frontal, se crearía una animación que daría la impresión de que el oso camina en la vista previa del juego.

El animator será el encargado de controlar esa animación y todas las que se le pudieran añadir posteriormente al mismo objeto, así como la velocidad de las mismas y los saltos de una animación a otra. Si queremos que una animación sólo se reproduzca una vez tendremos que seleccionar la animación en los Assests de la vista del proyecto y a continuación desactivar la opción Loop Time en el Instector.

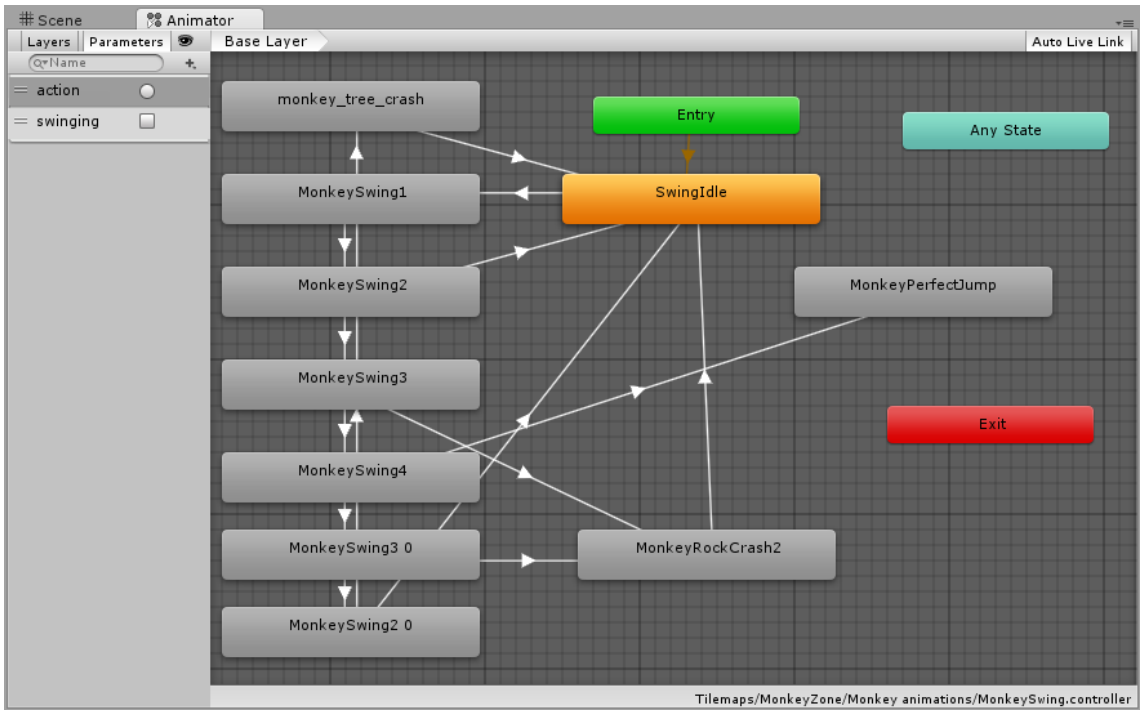


Captura en la que se muestran las vistas Animator y Animation del osos andando durante la ejecución de la vista previa del juego

Si bien la creación de animaciones no es complicada, la gestión de múltiples de ellas en un mismo objeto sí puede serlo. En el caso de nuestro mono balanceándose, se ha dividido lo que sería la animación completa en cuatro partes que se corresponderían con las cuatro posibles situaciones descritas en la página anterior,



Si se encuentra ejecutándose el primer grupo de los sprites de arriba, el mono se golpearía contra el árbol; si fuese el segundo, se bajaría del columpio; si fuese el tercero se chocaría contra la roca y si fuese el último lograría escapar. Este comportamiento se controla de manera similar en el Animator.



Se han definido dos variables para controlar la transición entre animaciones:

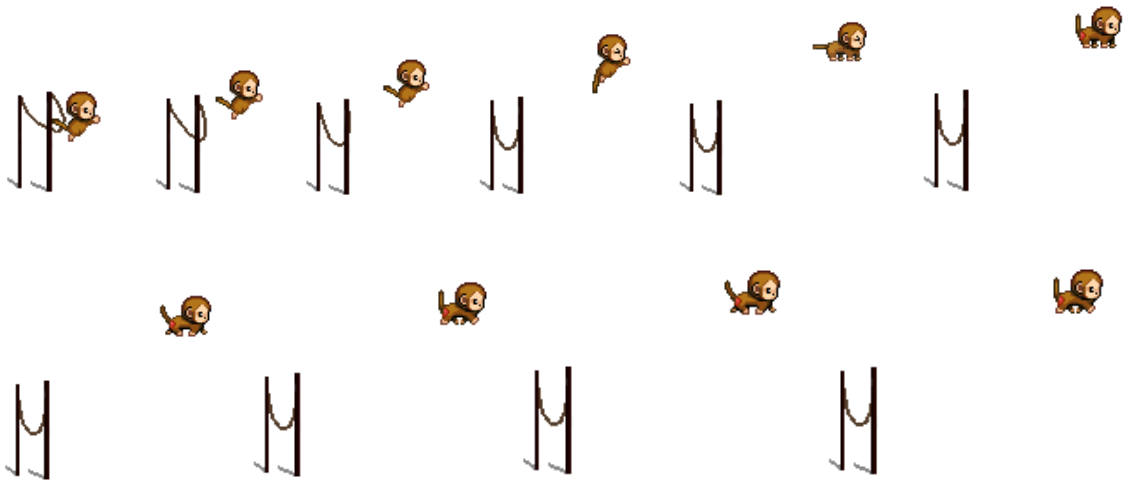
- swinging es una variable booleana que controla si el mono se está columpiando o no. Mientras no se esté columpiando, el animator ejecutará la animación de reposo (idle) que consistirá en la cuerda moviéndose sóla ligeramente de lado a lado, cuando empiece a columpiarse, el animator ejecutará secuenciamente las animaciones del mono columpiándose.
- action es un disparador, los disparadores funcionan de manera similar a las variables booleanas, pero se mantienen por defecto con un valor false salvo en el momento en el que se activan e inmediatamente después vuelven a recuperar el valor false por sí sólos. Utilizaremos el disparador action para saber cuándo pulsa el botón de acción el jugador y pasar a la animación que corresponda en cada caso (choque con el árbol, idle, choque con la roca o escapar).

La animación “MonkeyPerfectJump” contiene un evento al final que invoca al método “GoOut” presente en el script “SwingControler” del objeto que representa el columpio. Éste método carga el siguiente nivel.

El script SwingController también se encarga de interactuar con el Animator a través de las dos variables definidas para controlar las animaciones que se deben mostrar cada vez que el jugador pulsa la tecla de acción. El contenido del método Update de este script es el que se muestra a continuación:

```
void Update () {
    if (swinging){
        if(Input.GetKeyDown(KeyCode.E) || Input.GetMouseButtonDown(0)){
            animator.SetBool("swinging",false);
            animator.SetTrigger("action");
            swinging = false;

            Invoke("RestartPlayer",1);
        }
    } else if (canSwing){
        if(Input.GetKeyDown(KeyCode.E) || Input.GetMouseButtonDown(0)){
            swinging = true;
            HideMPlayer();
            animator.SetBool("swinging",true);
        }
    }
}
```

Mientras duren las animaciones del mono balanceándose o saltando del columpio, el personaje del mono debe permanecer oculto o de lo contrario veríamos dos monos en la escena, cuando hayan acabado estas animaciones, se mostrará de nuevo.



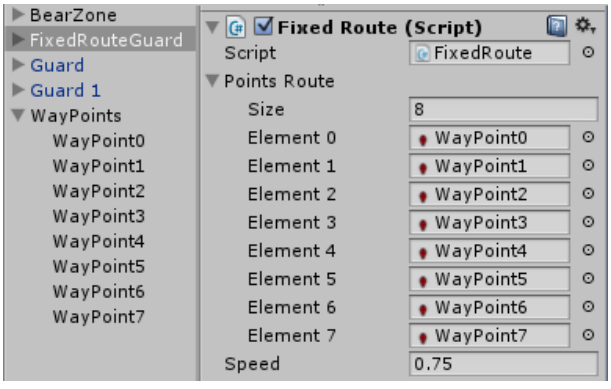
Arriba: secuencia de sprites de la animación de salto perfecto
Abajo: secuencia de sprites de la animación de salto y choque contra la roca

Una vez ha logrado salir de su hábitat, el mono se encuentra en la zona donde patrullan los guardias. Esta escena es la misma que se desarrolló en el segundo prototipo con algunos cambios.

En primer lugar hay otro guardia patrullando, pero éste no lo hace de forma aleatoria como los anteriores enemigos creados, sino que sigue una ruta fija que podemos ver indicada por los señaladores rojos en la siguiente captura:

El guardia con la ruta fija dispondrá, además, de la llave para abrir la puerta del hábitat del oso. Si intentamos acercarnos a él para quitársela sin tener a la serpiente en el equipo nos atrapará. El funcionamiento de este guardia es similar al de los otros, pero con un rango de acción más corto, pero más eficiente, es decir, podremos acercarnos más a él sin ser detectados, pero será imposible escapar de él si nos persigue.

Para hacer que el guardia siguiera una ruta fija, primero se crearon los puntos de ruta como objetos vacíos que simplemente servirían para señalizar las posiciones. A continuación se añadió una variable pública en el script de este enemigo que contendría un array de GameObjects para poder almacenar los puntos de ruta.



También se añadió otra variable pública para controlar la velocidad, y variables privadas para controlar el índice en el array del punto de ruta al que se pretende ir, la dirección a seguir y si está permitido caminar o no (esta última variable entrará más en juego cuando se tenga a la serpiente).

Codificaremos en el script para que el guardia se mueva inicialmente hasta el primer punto de ruta. Cuando se encuentre en ese punto, se pasará al siguiente objetivo actualizando el índice que lo indica. La dirección se obtiene empleando la misma fórmula que hemos descrito en los anteriores enemigos cuando persiguen al jugador, sólo que esta vez el "target" es el punto de ruta. El control de la linterna y los eventos de perseguir al jugador se codificaron en otro script del mismo asignado al mismo objeto para no añadir excesivo código en el script dedicado al movimiento.

```
// Use this for initialization
void Start () {
    walkEnabled = true;
    targetPointIndex = 0;
    direction = (pointsRoute[0].transform.position - transform.position).normalized;
    SendMessage("UpdateFlashlightRotation",direction);
}

// Update is called once per frame
void FixedUpdate () {
    if (walkEnabled){
        if (transform.position == pointsRoute[targetPointIndex].transform.position){
            ChangeTargetPoint();
        }else{
            float step = speed * Time.deltaTime;
            transform.position =
                Vector3.MoveTowards(transform.position,pointsRoute[targetPointIndex].transform.position,step);
        }
    }
}

private void ChangeTargetPoint(){
    if (targetPointIndex+1 == pointsRoute.Length){
        targetPointIndex = 0;
    } else {
        targetPointIndex++;
    }

    direction = (pointsRoute[targetPointIndex].transform.position - transform.position).normalized;
    SendMessage("UpdateFlashlightRotation",direction);
}

public void WalkEnabled(bool enabled){
    walkEnabled = enabled;
}
```

También se han añadido las grandes imágenes de la cabeza del oso, del mono y la serpiente que indican la localización de sus respectivos hábitats.

La forma de obtener el martillo tampoco ha variado desde el segundo prototipo, por lo que no se describirá este proceso. Una vez tenemos el martillo, podemos liberar a la serpiente que se encuentra en su hábitat.

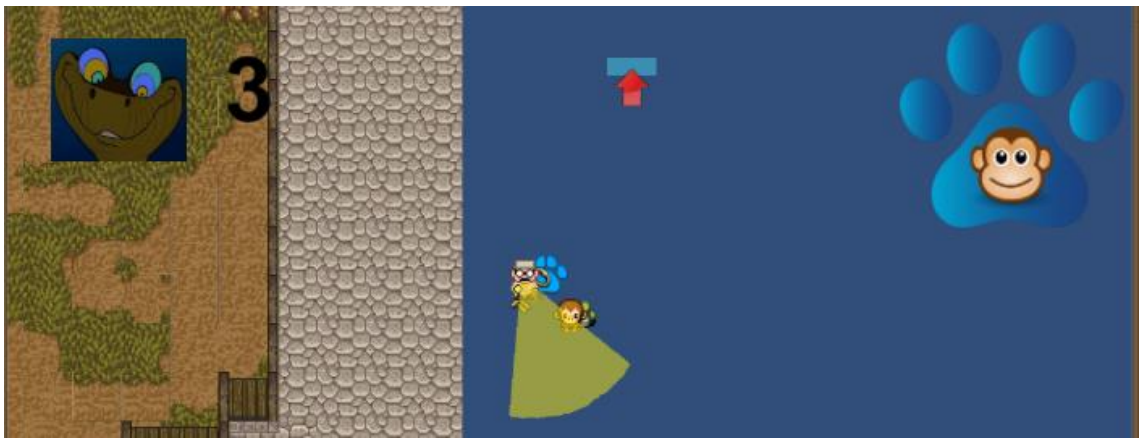
Cuando el mono se coloca sobre la huella situada frente al tanque de la serpiente, puede utilizar el martillo para romper el cristal y liberarla. La serpiente se une automáticamente al equipo y ya es posible rotar entre personajes. Para poder conseguir esto, ha sido necesario modificar el script del controlador y crear algunos "prefabs".

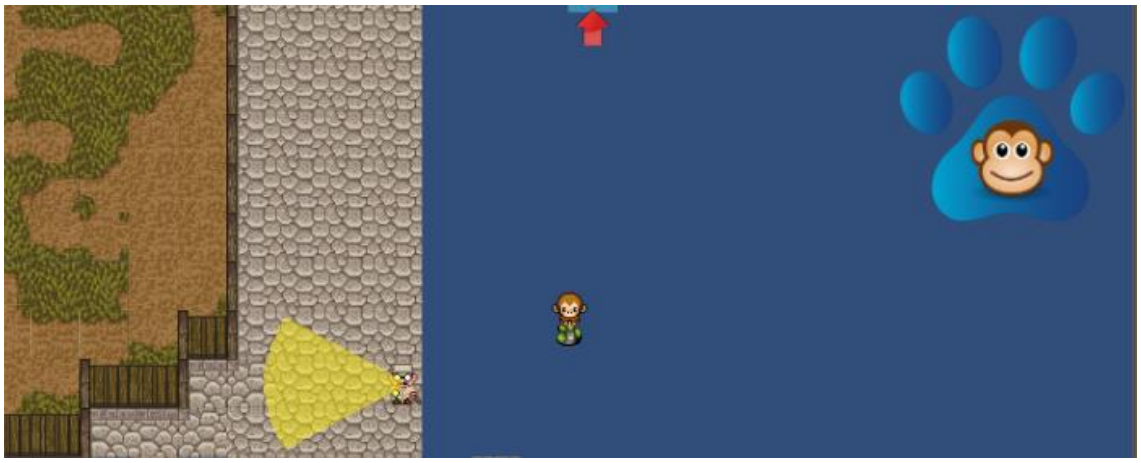
El script que controla el movimiento y rotación de los personajes ahora se encuentra en un objeto llamado “Characters” cuyos hijos son los personajes jugables. Inicialmente el único hijo del objeto “Characters” será el mono, pero más adelante se le añadirán los otros animales. El nuevo script controlador se basa en el número de hijos que tiene “Characters” para manejar las rotaciones y seguimientos. En lugar de actualizar una variable “character1” para que se corresponda siempre con el primer personaje, “character2” con el segundo y “character3” con el tercero como hacíamos en el anterior controlador, se emplea una variable entera que indica el índice del objeto hijo que es el personaje activo, evitando gran cantidad de asignaciones con respecto a la versión anterior.

Cuando se libera a un personaje y éste pasa a formar parte de nuestro equipo, se instancia un prefab del personaje como hijo del objeto “Characters”. Hay que tener en cuenta que los objetos instanciados tienen siempre el nombre del prefab a partir del que se crearo más la sentencia “(Clone)”, es decir, si tenemos un prefab llamado “Snake” y lo utilizamos para instanciar nuestra serpiente, el nuevo objeto se llamará “Snake(Clone)”.

A continuación se muestra el resultado de completar el puzzle 2, liberando a la serpiente y añadiéndola al equipo.

Una vez que tenemos a la serpiente, podremos hipnotizar temporalmente al guardia de la ruta fija para quitarle la llave del hábitat del oso.





Para llevar a cabo el “estado de hipnosis”, cuando el personaje activo entra en la zona de persecución del guardia el script de éste comprueba si la serpiente está en el equipo y si es así el guardia se queda congelado y se inicia una cuenta atrás. Si la cuenta atrás termina o aún no se ha liberado a la serpiente el guardia perseguirá al personaje. La imagen de la serpiente hipnotizando y la cuenta atrás se muestran en la UI mediante una imagen y un texto en un canvas. Cuando se obtiene la llave ésta desaparece del guardia.

Una vez que tenemos la llave, podemos liberar al oso. Éste se añadirá a nuestro equipo de forma análoga a como lo hizo la serpiente.

El cuarto puzzle consistiría en la liberación del rinoceronte y ya fue implementado en el segundo prototipo.

8. Implementación de la demo

Ahora que ya tenemos un prototipo casi completo del juego, a falta del menú de inicio y algunas características más, pasaremos al desarrollo de la demo final.

Algunos de los cambios que se pretenden realizar en esta demo con respecto al tercer prototipo son los siguientes:

- Implementación de un menú de inicio con las opciones
  - Iniciar nueva partida
  - Cargar partida guardada
  - Ver los logros conseguidos y por conseguir
  - Ver los controles del juego
  - Salir del juego
- Implementación del sistema de control de vidas
- Implementación de la opción de pausar / despausar el juego
- Creación de una mochila en la que se muestren los objetos conseguidos durante el juego, cada objeto de la mochila tendrá un nombre, una imagen identificativa y un número que indique la cantidad de objetos que tenemos del mismo tipo.
- Adición de música y efectos sonoros
- Implementación de la escena de créditos

Tras un tiempo de consideración, se decidió también darle un lavado de cara a la apariencia del juego empleando gráficos generados por la propia alumna.

8.1. Casos de uso de las interfaces de inicio y de la partida

Cuando un usuario ejecute este juego lo primero que aparecerá será la pantalla del menú de inicio, donde podrá realizar las acciones que ya comentadas:

- **Iniciar una nueva partida** provocará un salto al primer nivel del juego.
- **Cargar una partida** guardada hará que se cargue el nivel en el que el jugador se encontrase durante el último guardado junto con las modificaciones que se hubiesen realizado tales como animales liberados, objetos conseguidos, etc.
- **Ver los logros conseguidos y por conseguir** mostrará una nueva pantalla con dichos logros destacando aquellos ya obtenidos (Desde esta pantalla se podrá regresar al inicio).
- **Ver los controles del juego** mostrará una nueva pantalla con una imagen explicativa de los controles (Desde esta pantalla se podrá regresar al inicio).
- **Salir del juego** cerrará la aplicación del juego.

Como ya se ha visto en apartados anteriores y añadiendo las nuevas opciones que se pretenden implementar, durante una partida el jugador podrá

- **caminar** con su personaje
- **rotar** entre personajes siempre que tenga al menos dos en el equipo y no haya ninguna situación que lo impida
- **realizar acciones** siempre que sea posible
- **abrir (o cerrar si ya está abierta) la mochila** para ver los objetos conseguidos
- **pausar (o despausar si ya se ha pausado) el juego**; mientras el juego esté pausado se mostrará un mapa del zoo y las opciones para volver al menú de inicio o salir del juego

Las posibles acciones del jugador según éste se encuentre frente a la interfaz de inicio o jugando una partida se recogen en los siguientes diagramas de casos de uso.

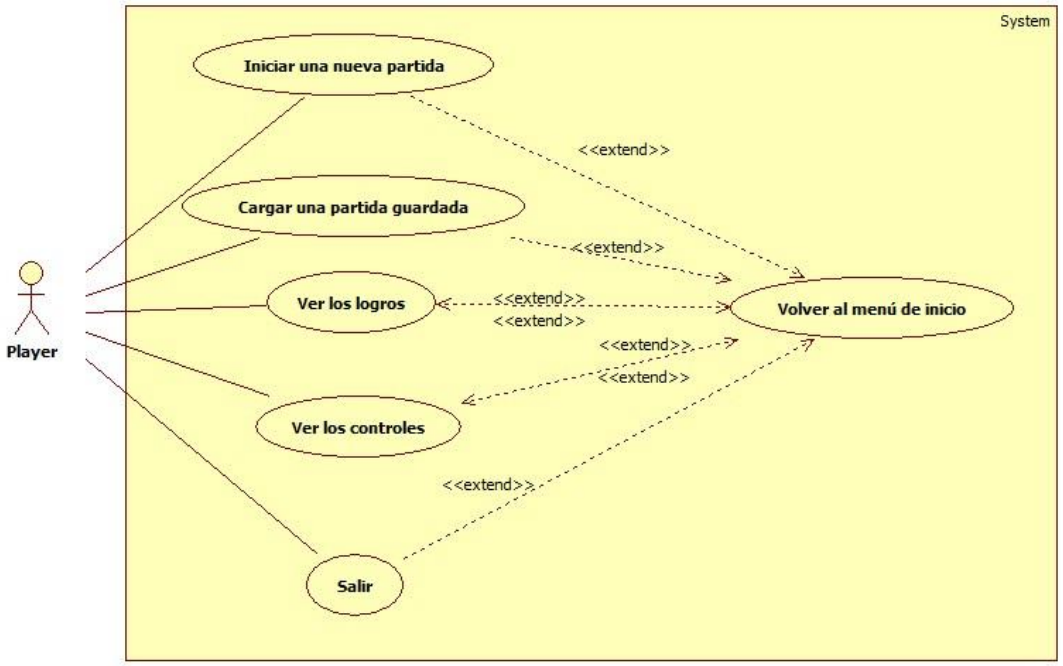


Diagrama de casos de uso para la interfaz de inicio

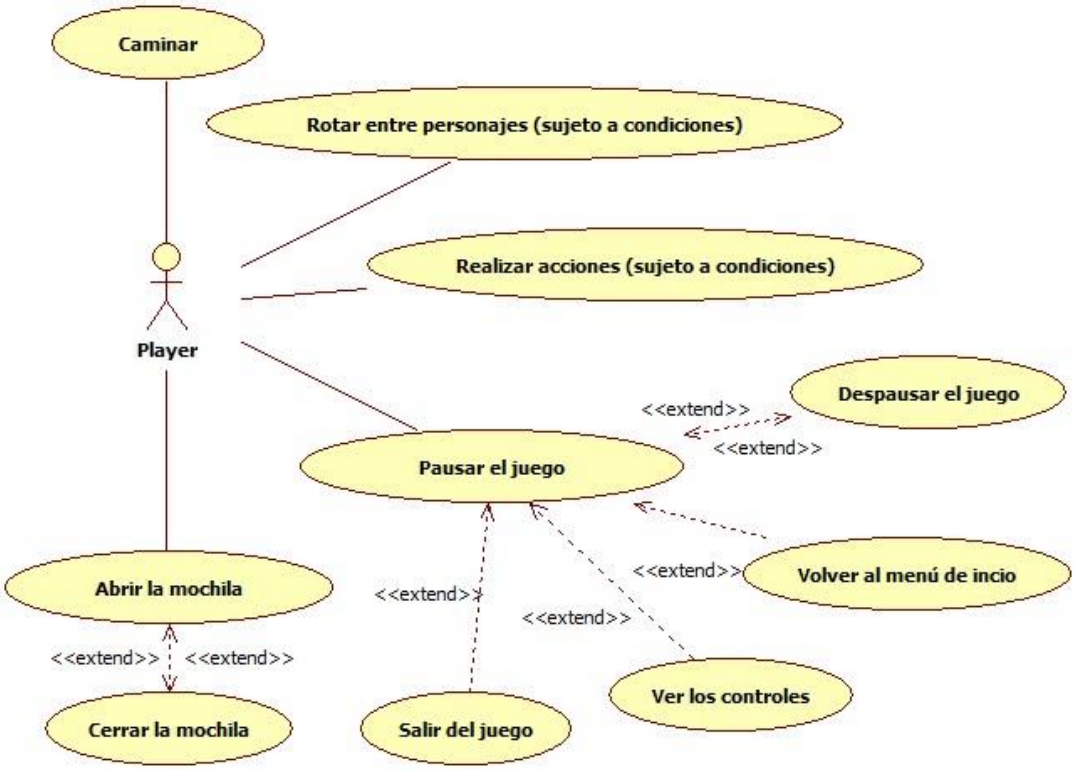
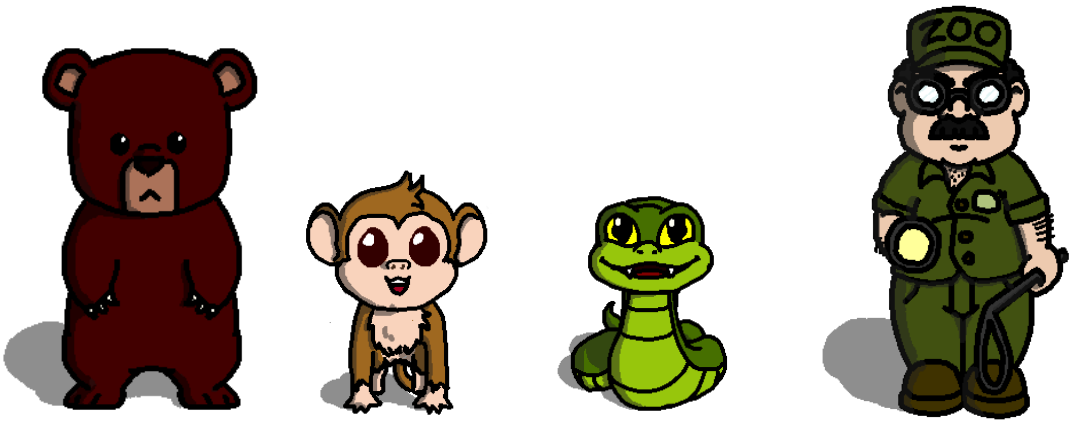


Diagrama de casos de uso durante una partida

8.2. Nuevos gráficos

Con el fin de hacer la interfaz del juego más atractiva y personal todos los elementos gráficos necesarios fueron creados desde cero por la alumna e incorporados en la demo final. Los personajes se dibujaron inicialmente sobre el papel para luego ser digitalizados y añadidos a los assets del proyecto.

Boceto de los personajes principales realizado en papel



Personajes ya digitalizados

Además de los personajes, se realizaron gráficos para representar todo el mapa del zoo como los distintos hábitats, el camino, objetos y elementos decorativos. Algunos hábitats fueron dibujados por partes para permitir que los sprites de los personajes se mostraran superpuestos sobre algunas partes del hábitat y detrás de otras. Es el caso del hábitat del mono, en el que éste debe aparecer por encima del terreno, pero por detrás de la cristalera y del árbol si se encuentra detrás de éste.

A la derecha podemos ver una descomposición del hábitat del mono en la que se aprecian las distintas imágenes que lo forman. Las imágenes que representan las sombras son semitransparentes y se les asignará un orden por encima de otros sprites como por ejemplo el de los personajes. La cristalera también es semitransparente y mientras estemos en el hábitat del mono, será el sprite que mayor orden tenga, es decir, se verá por encima del resto de los sprites.

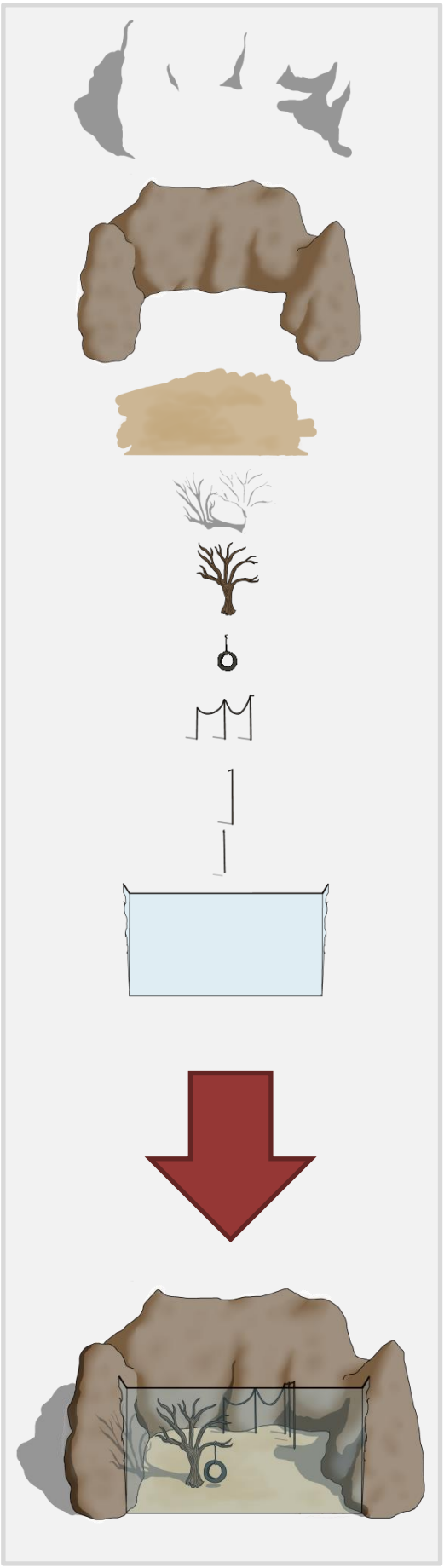
En la parte inferior de esta página se pueden observar también ejemplos de algunas imágenes que representan objetos y elementos decorativos del juego como los bancos y papeleras, así como imágenes que formarán parte de la UI como los corazones que indicarán la vida restante o la mochila.

Los gráficos del resto de los hábitats se podrán observar en capturas del juego en los próximos apartados.

Elementos que se utilizarán en la UI

Objetos

Elementos decorativos



8.3. Cambios estructurales y modificaciones en algunos scripts

Para esta demo final, se ha querido seguir una estructura más organizada en la jerarquía de los objetos y en las funciones de las que se encarga cada uno. Una estructura que se emplea en los proyectos de videojuegos consiste en asignar controladores o managers para manejar diferentes eventos según estén relacionados con el juego en general, con un nivel en particular, con el/los personaje/s, etc. Los proyectos con esta estructura suelen tener al menos un ”Game Manager”, un “Level Manager” y un “Player Manager”. En nuestro caso, se han definido los siguientes controladores:

- **Menu Manager**. Estará presente sólo en la escena correspondiente al menú de inicio y será el encargado de gestionar las distintas opciones de éste.
- **Game Manager**. Controla todo el juego a nivel general, lo utilizaremos para gestionar las vidas restantes, los cambios de escena, etc.
- **Level Manager**. Controla eventos propios del nivel en el que se encuentra el jugador. Existe un LevelManager distinto en cada escena del juego, incluso aunque su comportamiento sea el mismo, suponemos que esto es así para facilitar la adición de funcionalidades diferentes en cada nivel.
- **Player Manager**. Lo utilizaremos para controlar las acciones de los personajes, su orden en la escena, la cámara que sigue al personaje principal,… Básicamente son las mismas tareas de las que ya se encargaba el objeto “Characters”.
- **Stock Manager**. Se encargará de controlar los ítems adquiridos durante el juego tanto a nivel interno como en la UI.
- **Credits Manager**. Estará presente sólo en la escena de los créditos y será el encargado de gestionar las distintas opciones de éste

También tendremos en cada nivel un objeto “Environment” para agrupar los sprites que forman parte del entorno de la escena.

Uno de los cambios que se desprende de esta nueva estructura es que ahora un cambio de escena no lo hace cualquier script, sino que se manda un mensaje al Game Manager indicando la escena que quiere cargar y será éste el que se encargará de hacerlo y de realizar todas las acciones que considere oportunas, como por ejemplo, almacenar la localización del guardia con ruta fija para que éste no se encuentre siempre en el mismo lugar cada vez que se carga su escena.

Otro de los grandes cambios con respecto al último prototipo es el uso de una nueva clase llama “**State**”, una clase estática que emplearemos para almacenar información acerca del estado del juego y que no está asociada a ningún objeto.

Inicialmente, se pensaba que todos los scripts en Unity debían estar asociados al menos a un objeto para poderse utilizar, lo cual era un pensamiento erróneo y State es una prueba de ello, de hecho al ser una clase estática no puede extender de MonoBehavior y por lo tanto no puede asociarse a ningún objeto. Pero esto no es un problema, ya que al no depender de un objeto, State existe en todo momento en el juego, lo que la convierte en una clase ideal para almacenar información.

Aunque en un principio la idea parecía una locura para la alumna, State almacena variables públicas y estáticas, por lo que éstas son perfectamente accesibles para todos los scripts del juego sin tener que instanciar la clase. Esta situación que tan propensa a error parecía inicialmente, resultó ser una forma cómoda de resolver gran cantidad de inconvenientes. Uno de ellos era la persistencia de información relevante tras los cambios de escena. Las escenas se resetean a su estado original al ser cargadas nuevamente, por lo que todos los cambios realizados en las mismas (coger objetos, modificar posiciones, etc.) se pierden a no ser que almacenemos esa información en otro lugar. Resultaba muy engorroso tener que encasquetarle toda la información al Game Manager y luego estar invocándolo constantemente para saber si tal o cual acción habían tenido lugar ya o no. Es el caso del martillo, por ejemplo. Inicialmente había que preguntarle al Game Manager si ya se había conseguido el martillo, o la llave del hábitat del oso, o si la serpiente ya estaba en el equipo, etc., con la clase State, basta con ejecutar la instrucción "State.hasHammer" para saber si tenemos el martillo y actuar en consecuencia. Tener el martillo implica

1) que no lo podremos coger de nuevo, por lo que si al cargar la casa del guardia comprobamos que ya tenemos el martillo, destruiremos inmediatamente el script que se encarga de añadirlo si el jugador lo coge así como el sprite del martillo en la caja de herramientas
2) que podremos realizar acciones que impliquen haber conseguido el martillo

Si no tenemos el martillo, cuando el jugador lo consiga, se indicará que ya se ha adquirido con la instrucción "State.hasHammer = true;" y no se volverá a modificar esa variable a no ser que se inicie una nueva partida. Este comportamiento se repite en todos los scripts que utilizan las variables de la clase State. Para cada variable, sólo un script modifica su valor (al margen de la propia clase en la que se encuentran cuando se inicia una nueva partida), aunque sí pueden ser consultadas por más de un script. Siguiendo con el ejemplo del martillo, los cripts que controlan la acción de romper la pared del almacén, la acción de romper la vitrina de la serpiente y la adquisición del martillo leen el contenido de la variable State,hasHammer, pero sólo el último modifica su valor cuando se adquiere el objeto.

Todas las variables de la clase State tienen una inicialización por defecto que realiza la propia clase. El único método de la clase es el método "Reset", también público y estático, que se emplea para devolver a las variables su valor original y que solamente es invocado por el script del Game Manager en el momento de iniciar una nueva partida bien porque se han perdido todas las vidas o bien porque el jugador ha decidido abandonar la partida actual y volver al inicio.

Uno de los scripts más utilizados en este juego se llama **Behind**. Behind responde a la necesidad de tener algún mecanismo que controle cuándo un sprite se debe ver delante de otro, y cuando es el otro el que debe verse por encima. Por poner un ejemplo, cuando un personaje pasa "por delante" de un árbol lo lógico es que su sprite se vea por encima del árbol, mientras que si pasa por detrás es el árbol el que debe verse por delante. Con los gráficos de los prototipos anteriores construidos a base de tiles, podíamos asignar el orden por separado a cada uno de los tiles pertenecientes a una misma imagen general, para crear el efecto de profundidad. Pero ahora que las imágenes son completas sólo hay un posible orden para cada una de ellas o al menos para cada una de sus partes, lo que obliga a buscar otra solución para la profundidad.

La manera en que se ha resuelto este problema de estética ha sido mediante la creación del script Behind, que cambia el orden de los sprites que entran en una cierta zona haciéndolo inferior al del objeto que posee el script, y devolviéndoles su orden original cuando salen de la zona de “behind”. Behind se vale a su vez de otra clase denominada “Orderable” que almacena la información e implementa los métodos necesarios para devolverle a cada sprite su orden.

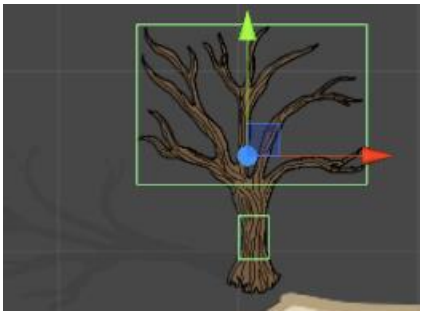
```
using UnityEngine;
using System.Collections;
using System;

public class Orderable: IEquatable<Orderable>{
    private string name;
    private int order;

    public Orderable(string name, int order){
        this.name = name;
        this.order = order;
    }

    public string GetName(){
        return name;
    }

    public int GetOrder(){
        return order;
    }

    public bool Equals(Orderable orderable){
        return String.Equals(name,orderable.GetName());
    }
}
```

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Behind : MonoBehaviour {
    private List<Orderable> orderables;
    private int thisObjectOrderInLayer;

    void Start () {
        orderables = new List<Orderable> ();
        thisObjectOrderInLayer = GetComponent<SpriteRenderer> ().sortingOrder;
    }

    void OnTriggerEnter2D(Collider2D other){
        if (other.tag != "Adornment" && other.tag != "Environment") {
            Orderable orderable =
                new Orderable(other.name,other.GetComponent<SpriteRenderer> ().sortingOrder);
            orderables.Add (orderable);
            other.GetComponent<SpriteRenderer> ().sortingOrder = thisObjectOrderInLayer - 1;
        }
    }

    void OnTriggerExit2D(Collider2D other){
        if (other.tag != "Adornment" && other.tag != "Environment") {
            Orderable orderable =
                new Orderable(other.name,other.GetComponent<SpriteRenderer> ().sortingOrder);
            int index = orderables.IndexOf(orderable);
            other.GetComponent<SpriteRenderer> ().sortingOrder = orderables [index].GetOrder();
            orderables.RemoveAt (index);
        }
    }
}
```

Behind no hace efecto sobre los sprites de los adornos o el entorno ya que son precisamente éstos los que tendrán este script y no nos interesa que se afecten entre ellos. De todas formas, en la mayoría de los casos, los objetos del entorno o decorativos son estáticos, por lo que no se moverán para colocarse unos detrás de otros. La distinción que se hace entre los elementos del entorno y los decorativos es que, aunque en teoría estos últimos pertenecerían al entorno, su función es principalmente decorativa, mientras que los primeros dibujan estructuras (paredes, suelo, hábitats completos,..).

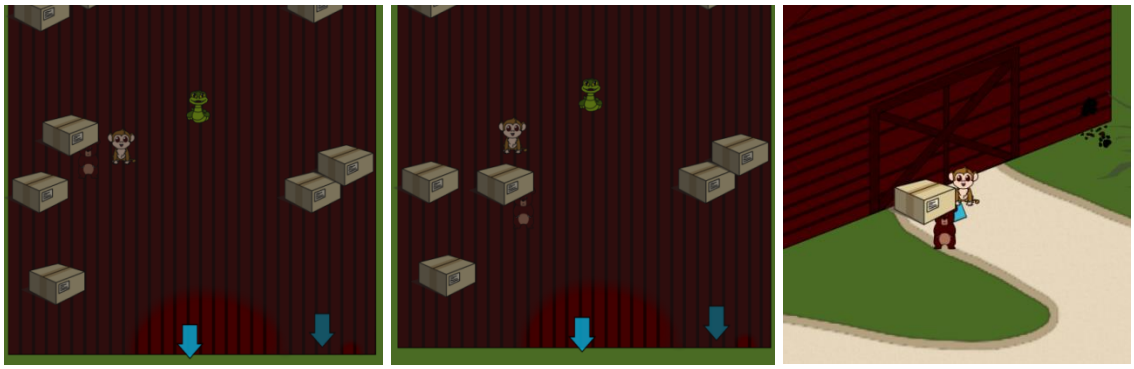
En las imágenes siguientes puede verse el funcionamiento del script Behind.

Como se puede observar, el mono se encuentra detrás del árbol en la primera escena, pero delante de él en la última.



Para que el sprite de un objeto permita que otros “pasen por delante” o “por detrás” de él, además del script Behind estos deben tener un collider trigger que delimite la zona en la que los otros sprites deben estar detrás. Estos objetos también suelen tener un collider sólido en su base que evita que sean “traspasables” por todas partes.

El cuarto puzzle también ha sufrido algunos cambios. Ahora las cajas no se instancian mágicamente desde la puerta del almacén, sino que se encuentran dentro de éste, en una nueva escena; el oso puede cargarlas para sacarlas de allí y soltarlas donde le plazca.



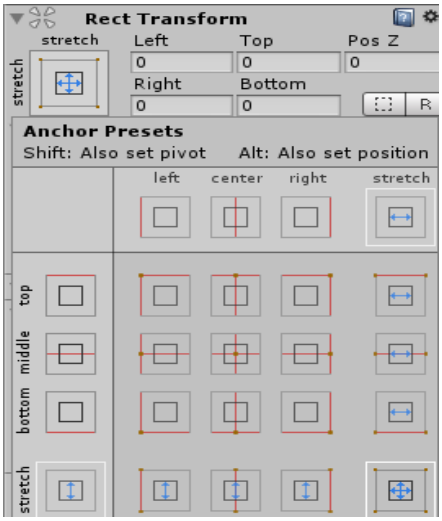
Las cajas son persistentes entre escenas, por lo que pueden ser llevadas a cualquier parte del juego donde pueda llegar el oso y si se sueltan en una escena, permanecerán en ese nivel y posición mientras el oso no las mueva de nuevo.

Al final del puzzle se añadió, a su vez, la animación del rinoceronte escapando de su hábitat que se comentará con más detalle en el apartado de animaciones.

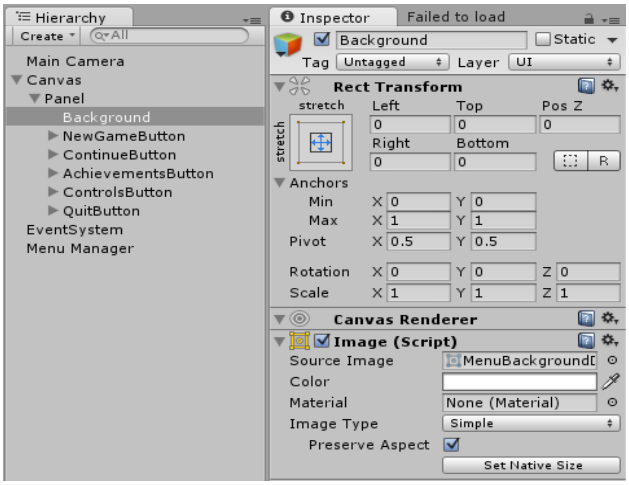
8.4. El menú de inicio y controles del juego

Para crear el menú de inicio se utilizaron una nueva escena con su cámara, cuatro tipos de objetos de la UI de Unity (canvas, panel, botón y texto), el objeto “EventSystem” necesario para el manejo de botones y el Menu Manager.

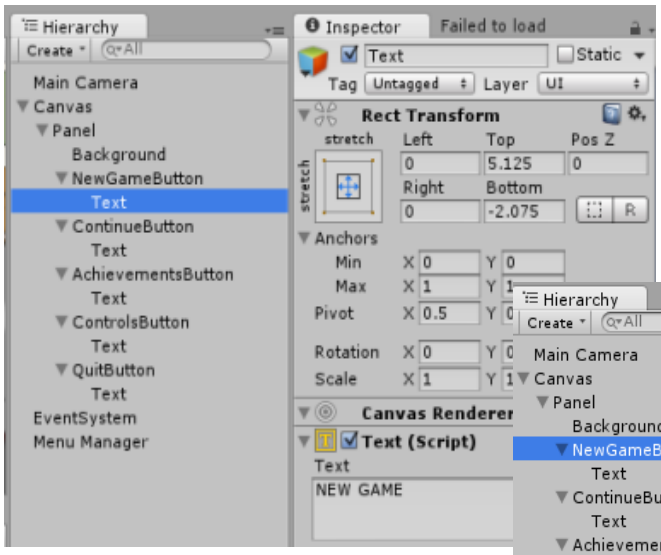
Dentro de un canvas, colocaremos el panel que contendrá a su vez los botones y el fondo del menú. Los objetos de la UI poseen un componente que permite definir su comportamiento en la interfaz (sus “anclas”, si cubren toda la pantalla, si se deforman en función de ésta o mantienen su apariencia original,…). Un objeto de la UI se puede anclar a una de las esquinas de la pantalla, a uno de los laterales o al centro, de forma que cuando el tamaño de la pantalla varíe, el objeto se mantendrá anclado a la zona indicada. También se puede definir si queremos que el objeto cubra toda la pantalla a lo ancho, a lo largo, o en ambos casos.



En el caso de la imagen de fondo del menú, se ha indicado que cubra toda la pantalla, pero con una peculiaridad: que se conserve su aspecto original. De esta forma, la imagen cubrirá la pantalla a lo ancho o a lo alto, dependiendo de qué lados lleguen a los bordes de la pantalla sin deformarse la imagen. Como este juego ha sido pensado para ser jugado en un PC, el fondo creado es más ancho que alto. Y para evitar un acabado “feo”, con espacios a los lados de la imagen cuando esta no cubre exactamente las dimensiones de la pantalla, la imagen se ha creado sin fondo y se ha aprovechado la posibilidad de añadir un color de fondo en la cámara para completar el dibujo.

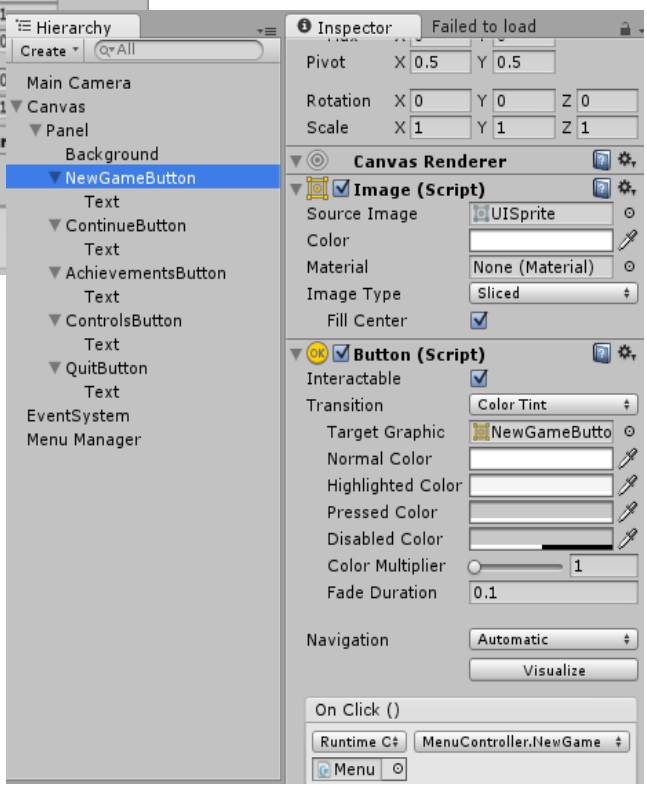


Los botones se anclarán a la esquina inferior izquierda de la pantalla. Estos objetos tendrán a su vez objetos hijos que contendrán el texto mostrado.



Los botones tienen un script asociado que permite definir si queremos que se invoque a algún método cuando se pulse, sólo tenemos que indicar cuál es el objeto que tiene los métodos que nos interesan y qué método (o concretos) queremos ejecutar.

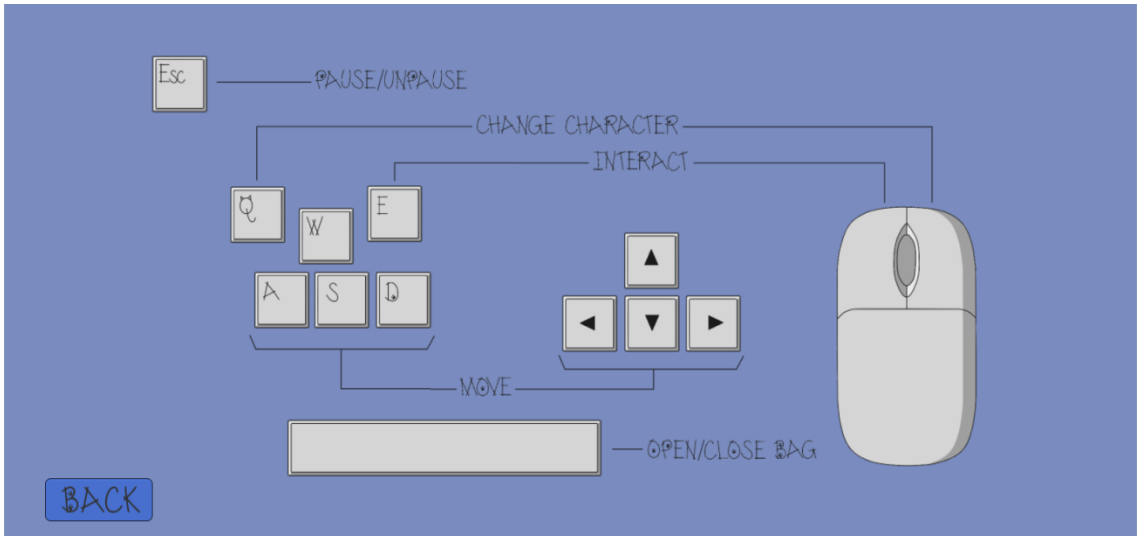
En la imagen de la derecha podemos observar que el método asociado al botón “New game” es el “MenuController.NewGame”.



MenuController es el script del Menu Manager que contiene las acciones disponibles al pulsar los botones. Lamentablemente en esta demo no ha habido tiempo para implementar las opciones de continuar el juego y mostrar los logros.

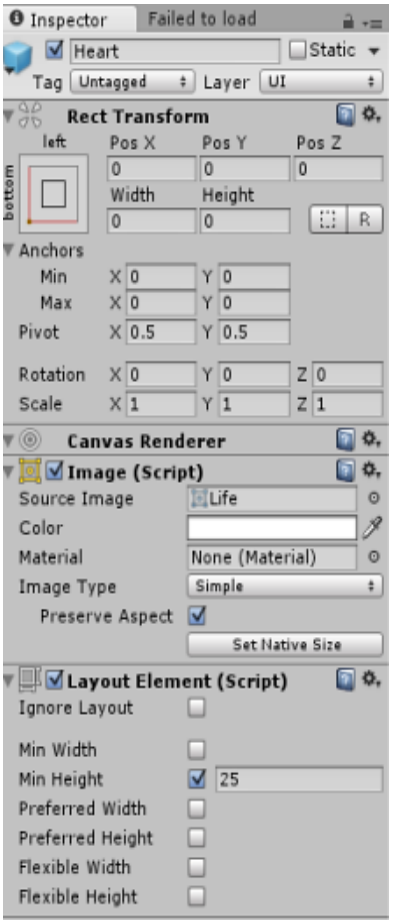
```
using UnityEngine;
using System.Collections;

public class MenuController : MonoBehaviour {

    public void NewGame(){
        Application.LoadLevel (1);
    }

    public void ContinueGame(){
        //Not implemented
    }

    public void ShowAchievements(){
        //Not implemented
    }

    public void ShowControls(){
        Application.LoadLevel (5);
    }

    public void QuitGame(){
        Application.Quit();
    }
}
```

La opción de mostrar los controles carga una nueva escena con la imagen de los controles del juego. En la pantalla de los controles la única acción posible es pulsar el botón “BACK” de la esquina inferior izquierda para volver al menú.



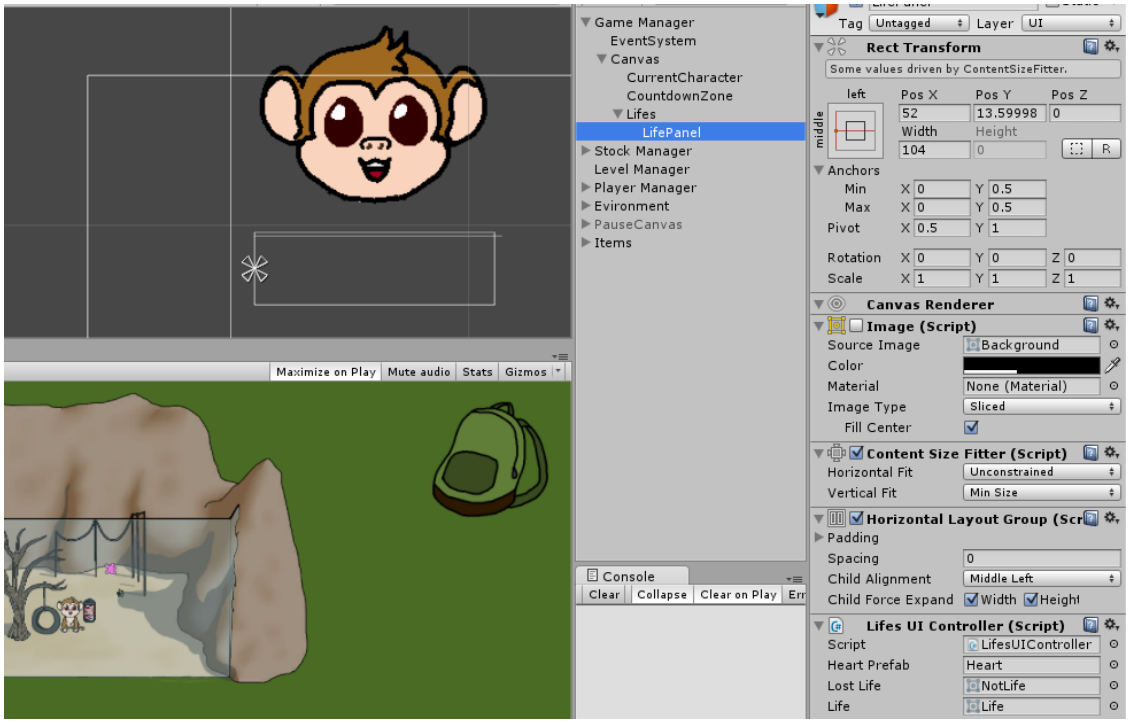
8.5. Sistema de control de vidas

Las vidas del jugador las controla el Game Manager. En el script “GameControl” presente en este objeto se encuentra la variable pública “initialLifes” que indica con cuántas vidas se empieza la partida. Otra variable privada llamada “currentLifes” controla la vida del jugador durante toda la partida. Cuando un enemigo captura al jugador manda un mensaje “Catch” al Game Manager, se decrementa una vida y este hecho se muestra a través de la interfaz. Cuando el número de vidas llega a cero se inicia una nueva partida con las vidas iniciales de nuevo.



Para gestionar las vidas de forma dinámica en la interfaz, se creó un Prefab con dos componentes: Image y Layout Element. El primero contendría la imagen que mostraremos del corazón y el segundo es un script de Unity que permite que un elemento sea gestionado con un layout, en este caso, en base a una altura mínima.

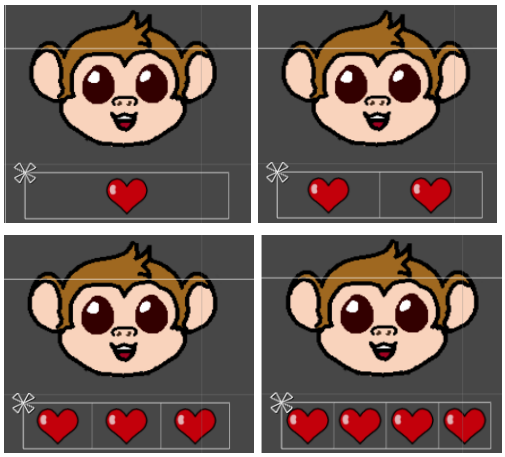
Dentro del canvas que ya se empleaba para mostrar el personaje activo, se creó otro panel denominado “Lifes” y que contendría la representación gráfica de las vidas del jugador. Como hijo del “Lifes”, se creó otro panel con un componente “Horizontal Layout Group”, que ordena sus objetos layout hijos horizontal y consecutivamente.

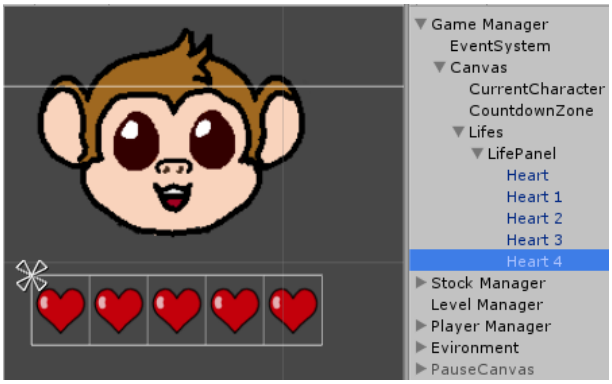


Al nuevo “LifePanel” se le añadió un script para controlar el número de “corazones” que debía mostrarse y de qué tipo (los corazones rojos simbolizan una vida y los oscuros tras barrotes, una vida perdida). Así pues, el prefab que creamos será usado por este script para instanciar las imágenes que representan el estado de las vidas en la interfaz. Además del prefab, el script necesita que se le proporcionen los dos sprites de corazones disponibles.

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class LifesUIController : MonoBehaviour {
    public GameObject heartPrefab;
    public Sprite lostLife;
    public Sprite life;

    private int currentLifes;

    public void SetInitialLifes(int lifes){
        if (this.transform.childCount > 0) DestroyChildren();
        currentLifes = lifes;
        GameObject life;
        for (int i=0; i<lifes; i++) {
            life = Instantiate(heartPrefab,this.transform.position,Quaternion.identity) as GameObject;
            life.transform.SetParent(this.transform);
        }
    }

    public void RemoveLife(){
        currentLifes--;
        transform.GetChild (currentLifes).GetComponent<Image> ().sprite = lostLife;
    }

    public void AddLife(){
        transform.GetChild (currentLifes).GetComponent<Image> ().sprite = life;
        currentLifes++;
    }

    private void DestroyChildren(){
        int children = transform.childCount;

        for (int i=0; i<children; i++) {
            Destroy(this.transform.GetChild(i).gameObject);
        }
    }
}
```

A continuación se muestran unas imágenes con el funcionamiento de la interfaz según las vidas iniciales que se hayan definido y a la derecha una captura en la que el jugador ha perdido una vida.





8.6. Pause

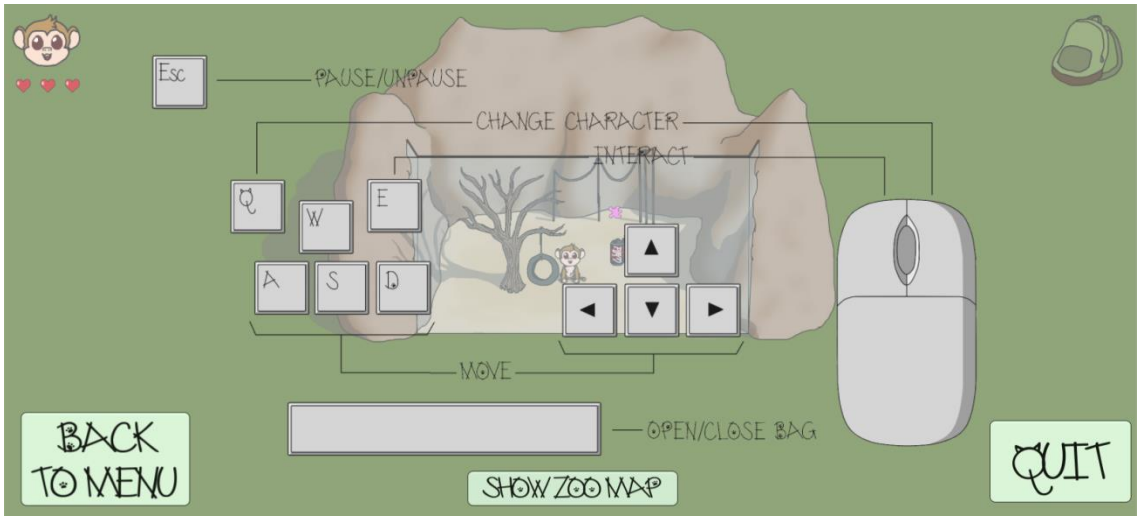
Al pausar el juego se mostrarán las opciones de volver al menú principal, mostrar los controles y salir, así como una imagen del mapa del zoo.

Mientras el juego esté pausado, ningún personaje podrá moverse, ni los controlados por el jugador ni los enemigos; sin embargo, se podrá abrir o cerrar la mochila ya que esto afecta al desarrollo de la partida, sino que se emplea simplemente para saber los objetos obtenidos.

Tanto si pulsamos en el botón “BACK TO MENU” como si lo hacemos en “QUIT”, se nos mostrará una ventana modal para confirmar cada acción. Mientras la ventana modal esté activa, los botones fuera de la misma permanecerán deshabilitados y no se podrá manipular la mochila ni reanudar el juego.

Si pulsamos sobre “SHOW CONTROLS”, la imagen del mapa del zoo se cambiará por la de los controles del juego y el botón pasará a mostrar el texto “SHOW ZOO MAP”; si pulsamos de nuevo en él, se cambiará la imagen con los controles para mostrar de nuevo el mapa del zoo y el botón recuperará el texto “SHOW CONTROLS”.

Si el jugador despausa el juego dejando visible la imagen de los controles la interfaz presentará esa misma imagen al pausar de nuevo en lugar del mapa del zoo.



Para crear el evento de pausar/despausar el juego se emplearon muchos de los elementos ya vistos en el menú de inicio.

Dentro de un canvas, se creó en primer lugar un panel que contendría la imagen a mostrar (mapa o controles) y los botones con las tres opciones posibles y sus respectivos textos. Además se añadieron otros dos paneles que se corresponderían con las dos ventanas modales. Las ventanas modales tienen una imagen de fondo de color plano, otra imagen con color plano que hace de borde de la ventana, la imagen del personaje del mono, un botón de confirmación y otro de negación.

En un script llamado “PauseController”, se crearon métodos para los comportamientos asociados a cada botón. En el script se almacenan en variables referencias a todos los elementos que se manipularán (el “PauseCanvas”, el objeto que contiene la imagen de fondo, los sprites de las dos imágenes posibles, el texto del botón que intercambia dichas imágenes y las dos ventanas modales), además de dos variables privadas para controlar si el juego se encuentra pausado o no y si se está mostrando o no una ventana modal.

Para controlar a nivel del juego si se está mostrando una ventana modal añadimos una nueva variable booleana a la clase State llamada “ShowingModalWindow”. Si el jugador intenta manipular la mochila o pausar/reanudar la partida, se comprobará primero que no haya una ventana modal abierta.

El juego se pausa o reanuda según corresponda pulsando la tecla escape (“Esc”).

```
void Update () {
    if (Input.GetKeyDown (KeyCode.Escape) && !State.showingModalWindow) {
        TogglePause();
    }
}
```

Mientras el juego no esté pausado, el canvas con todos los elementos de la interfaz de “pause” estará desactivado y todos los personajes podrán moverse con normalidad. Al pausar el juego, se mostrará dicho canvas y se pondrá el valor de la variable “Time.timeScale” a cero, haciendo que el movimiento de los personajes, que depende del tiempo, sea nulo[1]. Cuando el jugador indique que quiere reanudar la partida, se ocultará de nuevo el canvas y se indicará que la escala de tiempo vuelve a ser uno.

```
private void TogglePause(){
    pause = !pause;
    pauseCanvas.SetActive(pause);

    if (pause) {
        Time.timeScale = 0;
    } else {
        Time.timeScale = 1;
    }
}
```

Para implementar cuándo se muestra el mapa y cuando los controles, se creó un método que intercambia las imágenes y actualiza el texto del botón según corresponda cuando éste es pulsado.

```
public void ToggleControlsMap(){
    if (showingControls) {
        ShowZooMap ();
    } else {
        ShowControls();
    }
    showingControls = !showingControls;
}

private void ShowControls(){
    backgroundImage.GetComponent<Image> ().sprite = controlsImage;
    controlsMapButtonText.text = "SHOW ZOO MAP";
}

private void ShowZooMap(){
    backgroundImage.GetComponent<Image> ().sprite = zooMapImage;
    controlsMapButtonText.text = "SHOW CONTROLS";
}
```

Para el botón de volver al menú, crearemos primero un método llamado “TryBackToMenu” que mostrará la ventana modal para confirmar si se regresa al menú o no y desactivará los otros botones.

```
public void TryBackToMenu(){
    State.showingModalWindow = true;
    ActiveButtons (false);
    backToMenuConfirmPanel.SetActive (true);
}
```

Para desactivar los botones se les ha asignado el tag “Button” y, antes de mostrar una ventana modal, se hace una búsqueda empleando dicho tag y se deshabilitan.

---

[1]Recordemos que la forma de actualizar la posición de los personajes era igualándola a un vector multiplicado por la velocidad y por “Time.deltaTime”, al hacer Time.timeScale igual a cero, Time.deltaTime retorna también cero y hace que la operación anterior siempre dé como resultado a su vez cero.

```
private void ActiveButtons(bool active){
    GameObject[] buttons = GameObject.FindGameObjectsWithTag ("Button");
    foreach (GameObject button in buttons) {
        button.GetComponent<Button>().interactable = active;
    }
}
```

Si el jugador indica a través de la ventana modal que efectivamente quiere ir al menú, se indicará al Game Manager que ejecute su método “BackToMenu” que resetea el juego como si se hubiese perdido, recura la escala de tiempo original y cambia a la escena del menú antes de destruirse a sí mismo. A continuación se muestran en primer lugar el método “BackToMenu” del “PauseController” y tras él los métodos “BackToMenu” y “ResetGame” del “GameController”.

```
public void BackToMenu(){
    GameObject.Find ("Game Manager").SendMessage ("BackToMenu");
}
```

```
public void BackToMenu(){
    ResetGame ();
    Time.timeScale = 1;
    Application.LoadLevel (0);
    Destroy (this.gameObject);
}

private void ResetGame(){
    State.Reset ();

    GameObject[] managers = GameObject.FindGameObjectsWithTag ("Manager");
    foreach (GameObject manager in managers) {
        Destroy (manager);
    }
}
```

Si el jugador indica mediante la ventana modal que no desea volver al menú se desactiva la ventana modal de nuevo y se rehabilitan los botones.

```
public void CancelBackToMenu(){
    backToMenuConfirmPanel.SetActive (false);
    ActiveButtons (true);
    State.showingModalWindow = false;
}
```

Se procede de forma análoga cuando el jugador pulsa el botón “QUIT”.

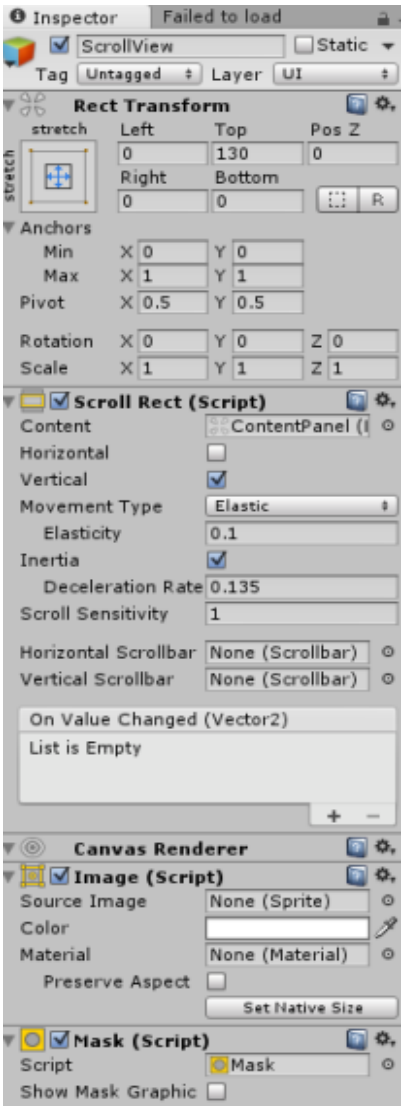
```
public void TryQuit(){
    State.showingModalWindow = true;
    ActiveButtons (false);
    quitConfirmPanel.SetActive (true);
}

public void Quit(){
    Application.Quit ();
}

public void CancelQuit(){
    quitConfirmPanel.SetActive (false);
    ActiveButtons (true);
    State.showingModalWindow = false;
}
```

8.7. La mochila

Para mostrar los objetos contenidos dentro de una mochila se procede de manera similar a cómo se muestran las vidas del personaje, pero empleando un “Vertical Layout Group” en lugar de uno horizontal y añadiendo otros elementos como el scroll.

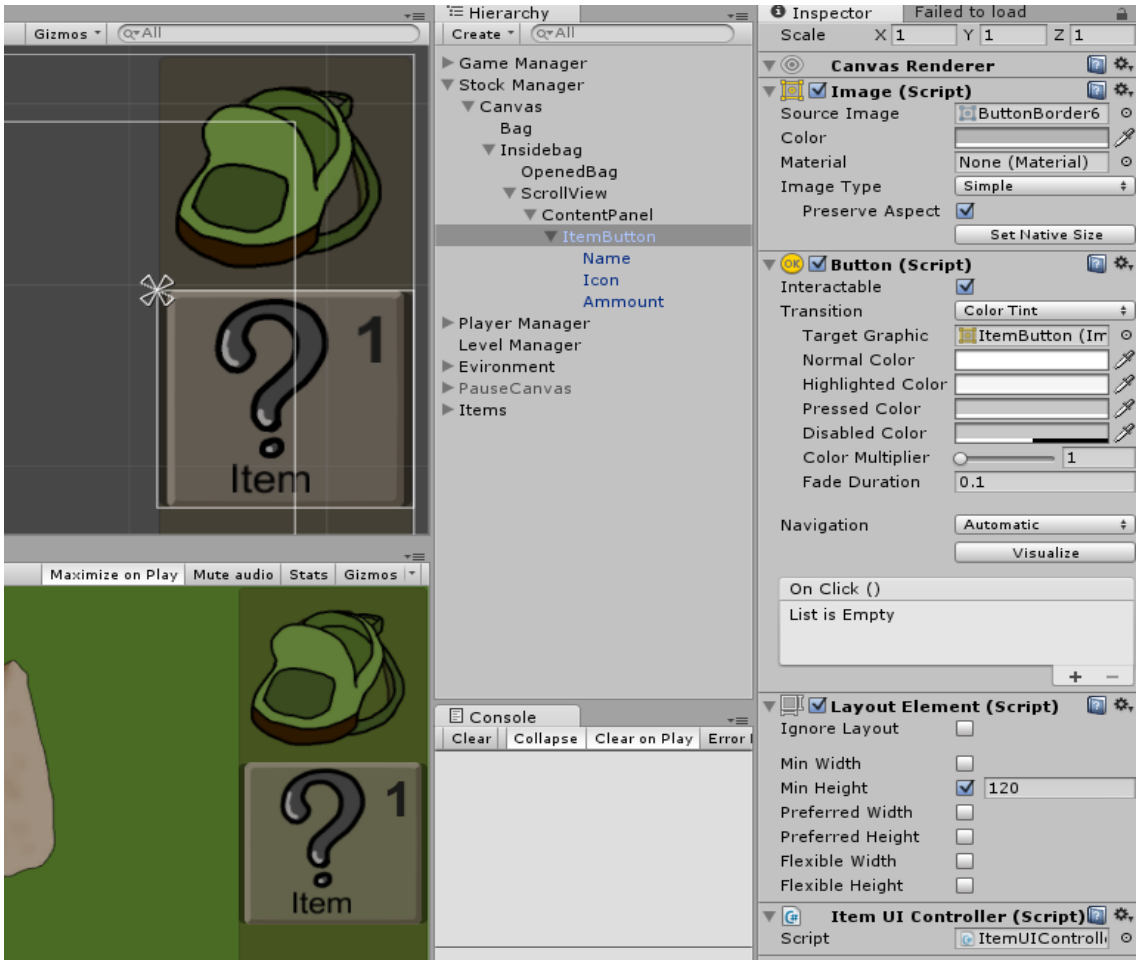
Mientras no se abra la mochila, sólo estará activa dentro del canvas la imagen de la misma cerrada. Cuando el jugador la abra, se activará el panel “Insidebag” que contiene la imagen de la mochila abierta y un objeto con un componente scroll que permite mover arriba o abajo los elementos que se encuentren en su “ContentPanel”.

En el componente Scroll indicamos que queremos un scroll vertical y que el movimiento sea tipo elástico, es decir, que cuando lleguemos a los elementos de los extremos superior o inferior se produzca ese efecto elástico si intentamos seguir haciendo scroll. El “ScrollView” también tendrá una máscara para que los elementos sólo se vean dentro de sus dimensiones de forma análoga a como se hizo con las vidas.



Dentro del ContentPanel tendremos un componente “Vertical Layout Group” que provocará que sus objetos layout hijos estén ordenados vertical y consecutivamente.

Para poder instanciar dinámicamente objetos dentro del “ContentPanel” crearemos un prefab básico que será un botón con una imagen y dos textos, la imagen será el icono del objeto, un texto será su nombre y otro será la cantidad. El motivo por el que se ha decidido utilizar un botón en lugar de una imagen sencilla se debe a que, aunque actualmente no está implementado, se pretende que en un futuro el jugador tenga que seleccionar objetos de la mochila para realizar acciones clicando sobre los mismos; actualmente basta con tener un objeto en la mochila para poder realizar acciones que necesiten de él. Al igual que sucedía con los elementos del panel de vidas, tendremos que indicar que el prefab que representará los ítems conseguidos es un “Layout Element” para que pueda ser ordenado.



El prefab que representa en la mochila los ítems obtenidos tiene también un script que permite actualizar el icono, el nombre y la cantidad.

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;

public class ItemUIController : MonoBehaviour {

    public void SetName(string itemName){
        transform.GetChild(0).GetComponent<Text>().text = itemName;
    }

    public void SetIcon(Sprite icon){
        transform.GetChild(1).GetComponent<Image>().sprite = icon;
    }

    public void SetAmmount(int ammount){
        transform.GetChild(2).GetComponent<Text>().text = ammount.ToString();
    }
}
```

El manejo de los ítems tanto en la UI como a nivel interno así como la apertura o cierre de la mochila lo realiza el Stock Manager mediante su script “StockController”.

El Stock Manager está presente durante toda la partida, en su script, almacena una lista con todos los ítems obtenidos que se corresponde con la representada en la mochila. Los ítems se representan mediante una clase “ItemUI” que contiene el nombre, el icono y la cantidad. Esta clase implementa las interfaces IComparable e IEquatable para poder emplear adecuadamente las funciones de las listas, para utilizar dichas interfaces tendremos que añadir la librería “System”. La clase también tiene un constructor que permite crear el ítem con los tres atributos que posee, así como getters y setters para dichos atributos. Los métodos Add y Remove permiten aumentar o disminuir en una unidad el número de elementos de un mismo objeto.

```
public void Add(){
    this.ammount++;
}

public bool Remove(){
    if (this.ammount == 0)return false;
    else {
        ammount--;
        return true;
    }
}

public int CompareTo(ItemUI item){
    return String.Compare(itemName,item.GetName());
}

public bool Equals(ItemUI item){
    return String.Equals(itemName,item.GetName());
}
```

La mochila podrá ser abierta o cerrada según corresponda gracias al StockController siempre que no esté activa una ventana modal.

```
void Update () {
    if (Input.GetKeyDown("space") && !State.showingModalWindow){
        ChangeBagState(!bagOpened);
    }
}

public void ChangeBagState(bool state){
    insideBag.SetActive (state);
    bagOpened = state;
}
```

Si cualquier script del juego desea añadir un ítem al stock, tendrá que enviarle un mensaje al Stock Manager con el ítem que se desea añadir. El método que se utilizará para añadir ítems será “AddItem”.

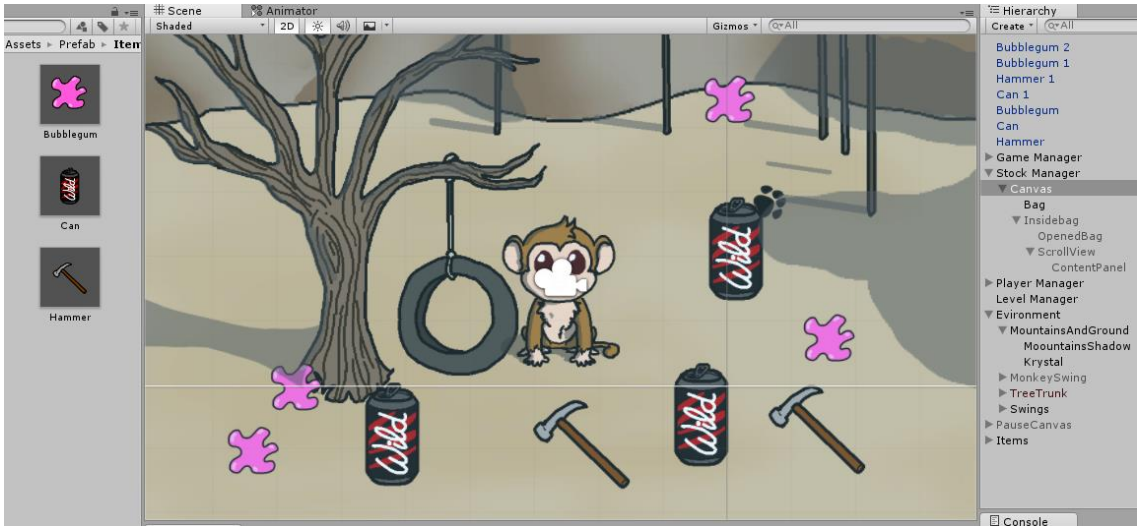
```
public void AddItem(ItemUI item){
    int index = itemList.IndexOf(item);

    if (!bagOpened) insideBag.SetActive (true);
    if (index < 0){
        itemList.Add(item);
        GameObject itemUI = Instantiate(itemUIPrefab, new Vector3(0,0,0), Quaternion.identity) as GameObject;

        itemUI.transform.GetChild(0).GetComponent<Text>().text = item.GetName();
        itemUI.transform.GetChild(1).GetComponent<Image>().sprite = item.GetIcon();
        itemUI.transform.GetChild(2).GetComponent<Text>().text = item.GetAmmount().ToString();

        itemUI.transform.SetParent (contentPanel.transform);
    } else {
        itemList[index].Add();
        contentPanel.transform.GetChild(index).SendMessage("SetAmmount",itemList[index].GetAmmount());
    }
    if (!bagOpened) insideBag.SetActive (false);
}
```

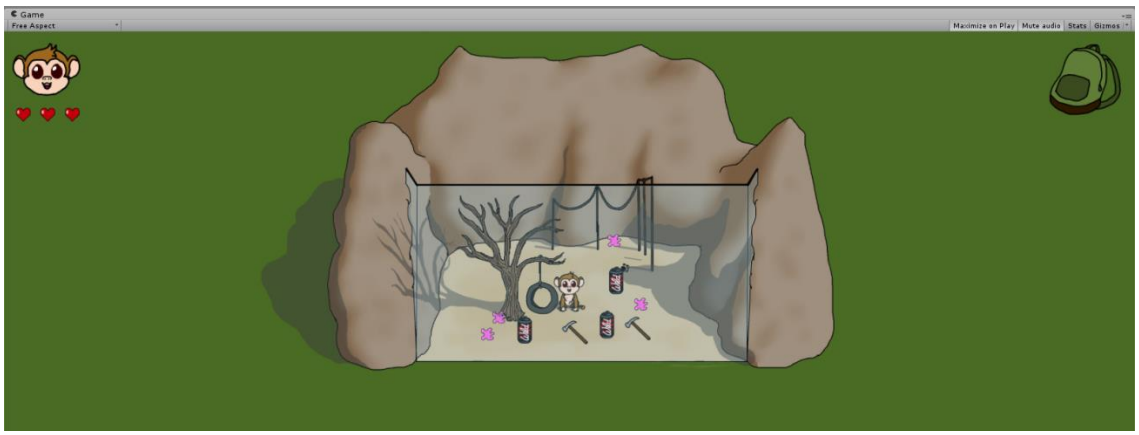
Si se desea añadir un tipo de ítem que ya existe en el stock, en lugar de ello se aumentará la cantidad del mismo. Es necesario abrir la mochila si está cerrada para añadir un ítem ya que de lo contrario el panel estaría inactivo y no se nos permitiría hacerlo. Al terminar de añadir un ítem se deja la mochila en el estado en el que se encontraba.

Se procede de forma similar cuando se desea eliminar un ítem del Stock.

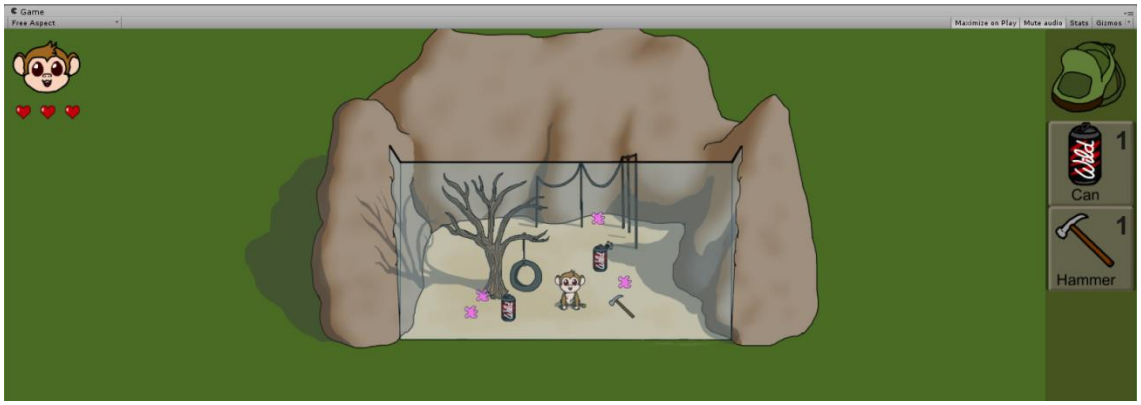
```
public bool RemoveItem(ItemUI item){
    int index = itemList.IndexOf(item);
    if (index >= 0){
        itemList[index].Remove();
        contentPanel.transform.GetChild(index).SendMessage("SetAmmount",itemList[index].GetAmmount());
        if (itemList[index].GetAmmount() == 0){
            itemList.RemoveAt(index);
            Destroy (contentPanel.transform.GetChild(index).gameObject);
        }
        return true;
    }else return false;
}
```

Para poder probar el funcionamiento del stock se colocaron en la escena varios ítems. Estos objetos poseen un sprite, un collider trigger para detectar si el jugador está cerca y un script que contiene su nombre y la cantidad junto con un método “Take” que indica al Stock Manager que añada el ítem cuando el jugador pulsa el botón de acción dentro de la zona del trigger. El objeto se destruirá una vez es obtenido.

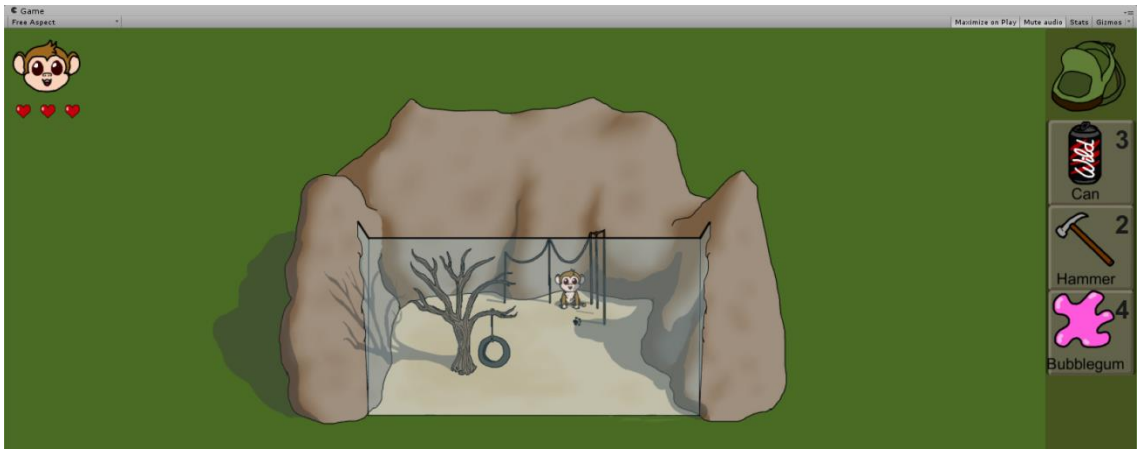




En primer lugar cogeremos sólo un par de objetos y comprobaremos si este hecho se ve reflejado en la mochila.

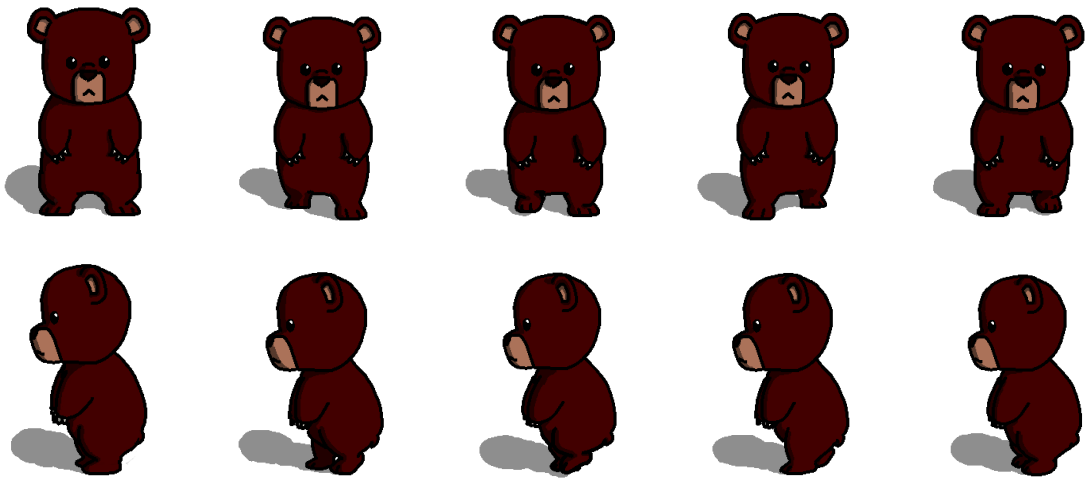


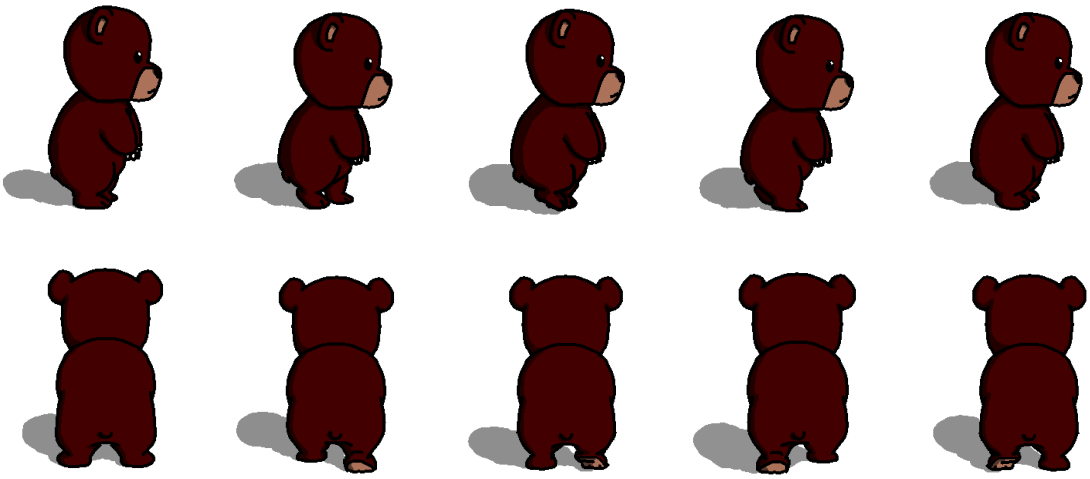
Y por último comprobaremos, cogiendo todos los objetos, si al añadir objetos de un tipo que ya existe en el stock se incrementa la cantidad de los mismos.



8.8. Las animaciones

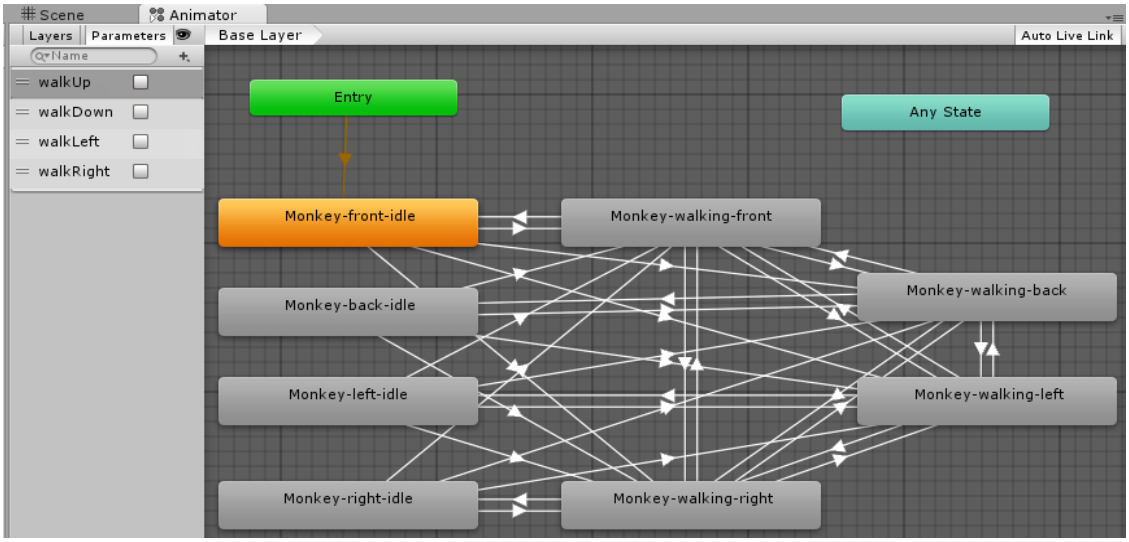
Para dar movimiento a los personajes cuando caminaran, se dibujaron sprites de todos ellos simulando distintos pasos en cuatro direcciones diferentes (norte, sur, este y oeste). A continuación se muestra un ejemplo con los sprites del oso estático y caminando.





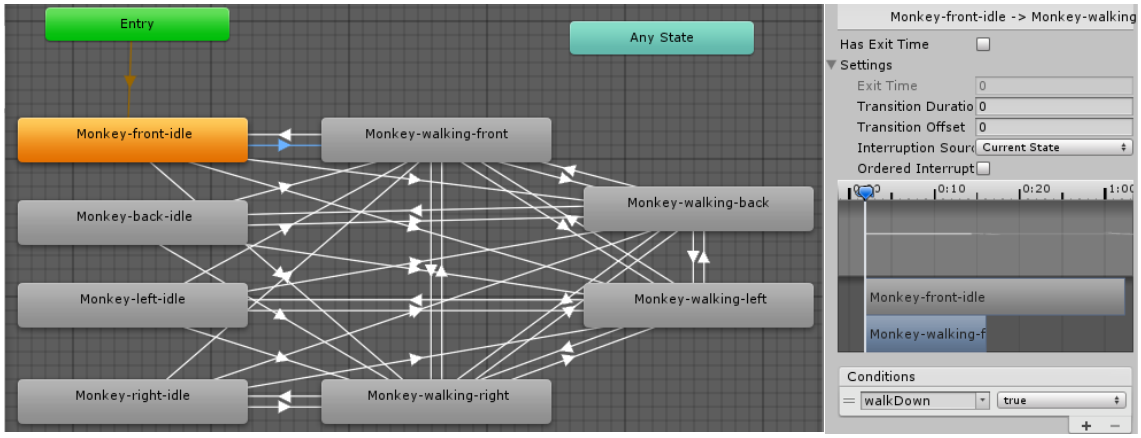
Ya hemos visto en otros apartados cómo se creaban las animaciones y cómo funcionaba el animator así que no nos entretendremos mucho en ello.

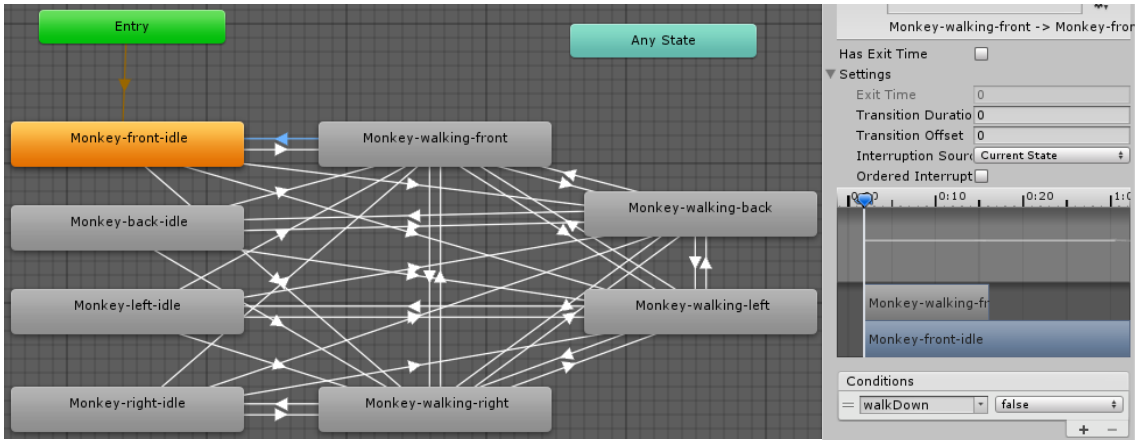
Se crearon controladores para las animaciones de cada uno de los personajes jugables (mono, serpiente y oso). A continuación se muestra una imagen del “animator” correspondiente al mono, los controladores de los otros personajes jugables son análogos.



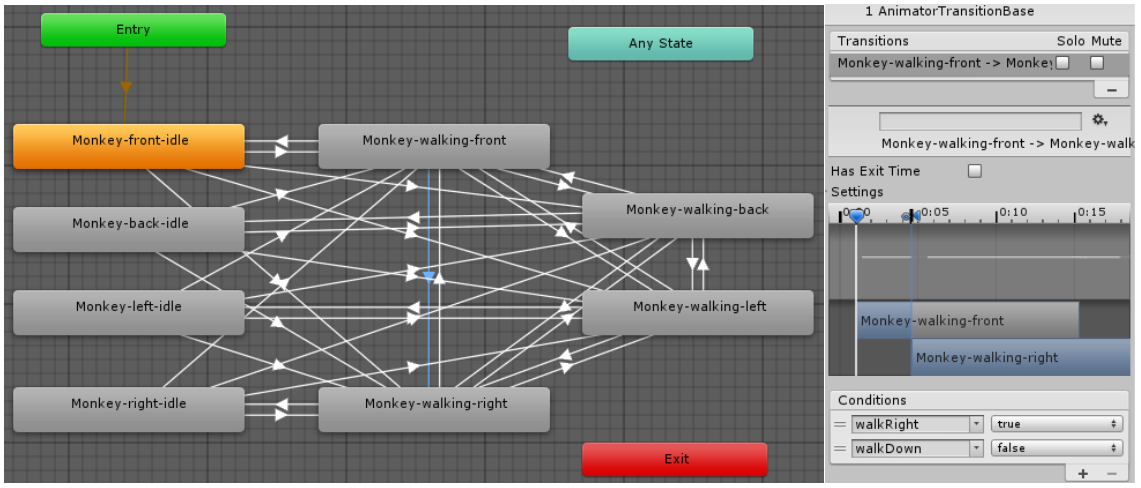
Aunque pueda parecer confuso, el funcionamiento del controlador de la imagen es sencillo. Mientras el jugador no pulse ninguna tecla de movimiento, el personaje permanecerá en una posición de “idle”, es decir, en reposo, dependiendo del movimiento previo que se haya hecho. Inicialmente el personaje se encuentra en el estado “Monkey-front-idle” que muestra al personaje del mono visto de frente y sin caminar. Si el jugador pulsa una tecla de dirección, el estado se cambia por el que contenga la dirección que corresponda a la tecla pulsada. Los estados contienen animaciones en bucle que pueden ser interrumpidas en cualquier momento en el que se cumplan las condiciones de una de sus transiciones salientes. Puesto que no se dispone de animaciones de los personajes en las direcciones diagonales, para evitar que las animaciones de caminar se “superpongan” cuando se pulsan varias teclas de dirección a la vez, si ya se está reproduciendo una animación del personaje andando no se cambiará hasta que deje de pulsarse su tecla asociada.

Las transiciones entre estados se controlan mediante las variables booleanas “walkDown”, “walkUp”, “walkRight” y “walkLeft”. Cuando el valor de alguna de esas variables pasa a ser “true” se salta al estado con la animación que corresponda y cuando ese valor pasa a ser “false”, si ninguna de las otras variables tiene el valor verdadero, se pasa al estado “idle” adecuado. Por ejemplo, el estado inicial es “Monkey-front-idle”, cuando el jugador pulsa la tecla asociada a caminar hacia abajo se pasa al estado “Monkey-walking-front” (el personaje se ve de frente) y cuando se suelta la tecla se regresa al idle.





En la siguiente imagen se muestran las condiciones para pasar de la animación de caminar hacia abajo a la animación de caminar hacia la derecha.



El manejo de las variables del animator del personaje activo lo realiza el Player Manager al tiempo que trata el movimiento, mediante el nuevo método “AnimationContro()l” que se invoca cada vez que se ejecuta “Move()”.

```
private void AnimationControl(){
    Animator animator = transform.GetChild(currentCharacterIndex).GetComponent<Animator> ();

    if(Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.UpArrow)){
        animator.SetBool ("walkUp",true);
    } else animator.SetBool ("walkUp",false);

    if(Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.DownArrow)){
        animator.SetBool ("walkDown",true);
    } else animator.SetBool ("walkDown",false);

    if(Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.LeftArrow)){
        animator.SetBool ("walkLeft",true);
    } else animator.SetBool ("walkLeft",false);

    if(Input.GetKey(KeyCode.D) || Input.GetKey(KeyCode.RightArrow)){
        animator.SetBool ("walkRight",true);
    } else animator.SetBool ("walkRight",false);
}
```

El método AnimationControl comprueba si se ha pulsado alguna tecla de dirección y si es así coloca el valor “true” en la variable correspondiente, de lo contrario coloca el valor “false”. Inicialmente este control se intentó hacer con variables “trigger”, pero estos provocaban comportamientos indeseados cuando se mantenía pulsada una tecla de movimiento. También se intentaron utilizar los métodos “GetKeyDown” y “GetKeyUp” en el control, pero estos procedimientos daban problemas porque en ocasiones el evento de pulsar o despulsar una tecla no era detectado en el frame adecuado.

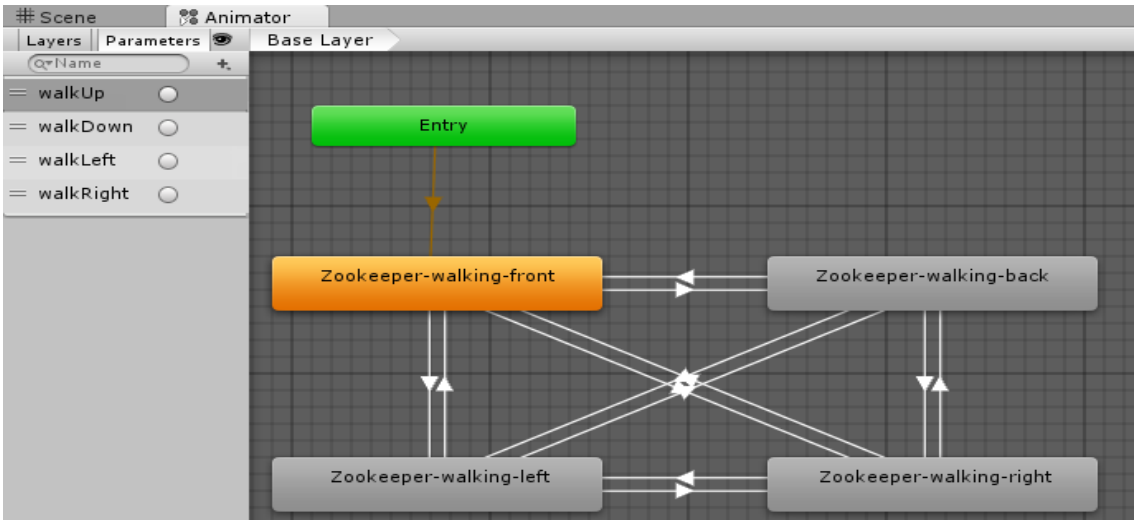
Para el caso de los personajes que siguen al personaje activo, se crearon dos métodos para activar las animaciones correspondientes a cada uno cuando se mueven desde una posición a otra.

```
private void ActiveAnimation (Vector3 direction, Animator animator){
    if (direction.y < -0.5) animator.SetBool ("walkDown", true);
    if (direction.y > 0.5) animator.SetBool ("walkUp", true);
    if (direction.y >= -0.5 && direction.y <= 0.5) {
        if (direction.x >=0) animator.SetBool ("walkRight", true);
        else animator.SetBool("walkLeft", true);
    }
}

private void DisableAnimation(Animator animator){
    animator.SetBool ("walkDown", false);
    animator.SetBool ("walkUp", false);
    animator.SetBool ("walkRight", false);
    animator.SetBool("walkLeft", false);
}
```

El funcionamiento consiste en obtener la dirección que va a seguir un personaje y proporcionársela junto con su “animator” al método ActiveAnimation, que seleccionará la animación que considere en base a la dirección. Cuando el movimiento termina, se invoca al método DisableAnimation indicándole el “animator” cuyas animaciones debe deshabilitar.

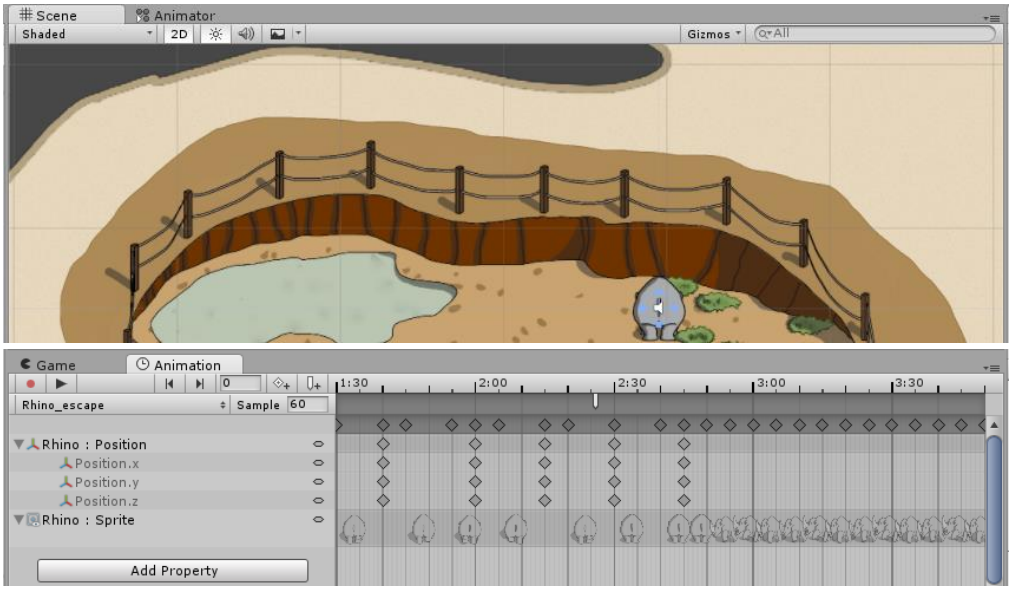
El procedimiento que se ha seguido para la animación de los guardias caminando es similar: se obtiene la dirección que van a seguir y se activa la animación que corresponda a esa dirección.

Inicialmente, en ocasiones, los guardias se quedaban parados cuando de forma aleatoria se obtenía la dirección (0,0,0), pero ese hecho se corrigió indicando que si se obtenía esa dirección, se debía cambiar por otra. Debido a este cambio, los guardias siempre están caminando, por lo que no tienen un estado “idle” para cuando se detienen y su controlador es más sencillo que el de los personajes jugables, en ellos se han empleado triggers en lugar de variables booleanas.

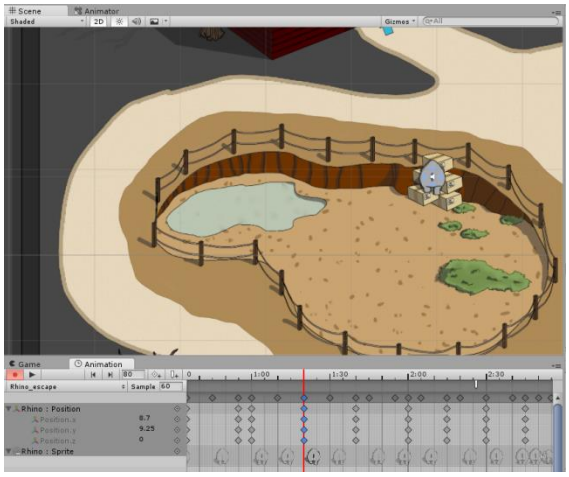


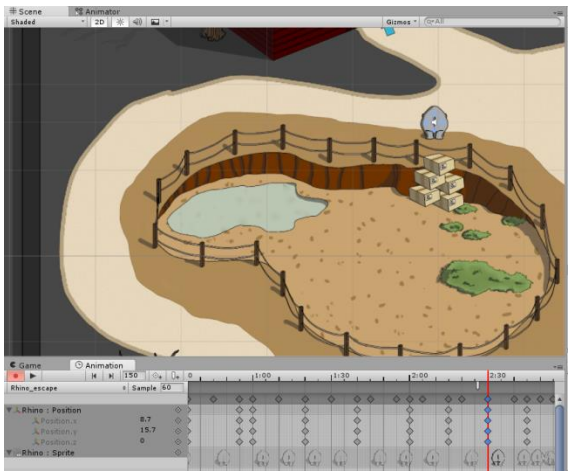
Además de las animaciones de los personajes caminando, se crearon otras para mostrar eventos en el juego. Es el caso del rinoceronte escapando de su hábitat.

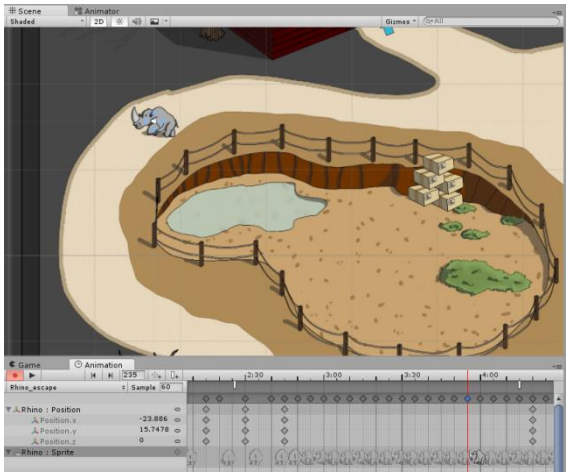
Para crear la animación del rinoceronte saliendo de su hábitat se empleó la vista de animaciones o “Animation”, mediante la cual se iba indicando qué sprite debía mostrarse en cada momento y en qué posición, ya que el rinoceronte se desplaza primero hacia arriba para salir del hábitat y a continuación hacia la izquierda para romper el muro y escapar del zoo. La animación se inicia desde scripting activando el animator del rinoceronte.

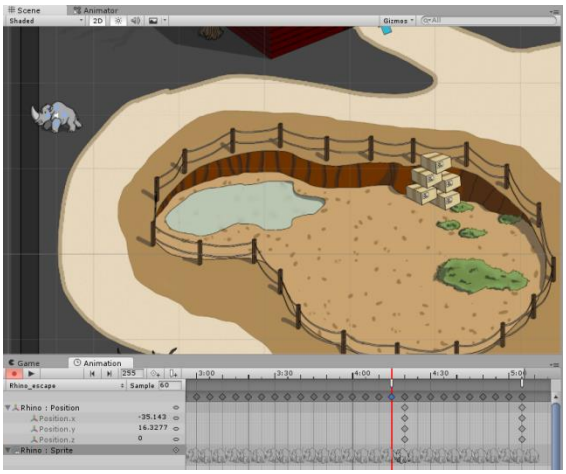


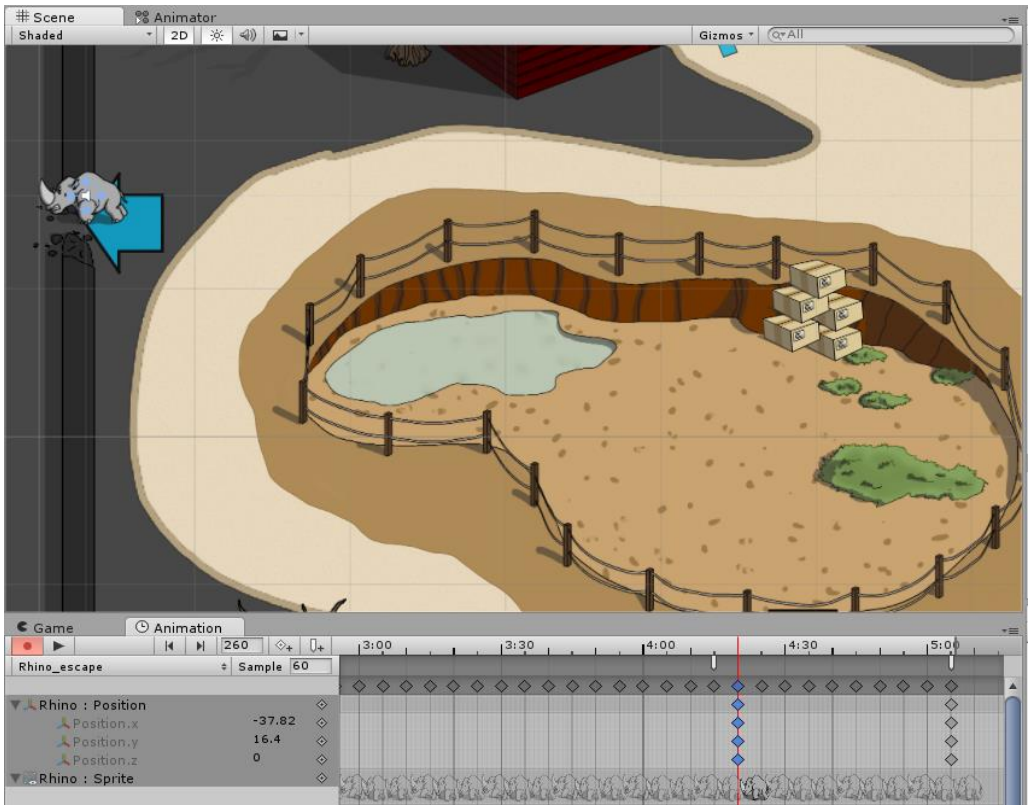
En esta página se muestran algunas escenas correspondientes a la animación del escape del rinoceronte.





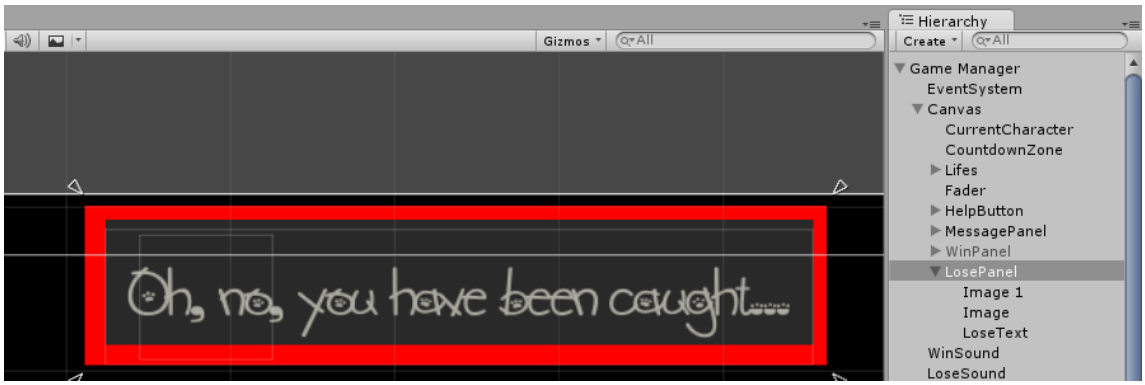






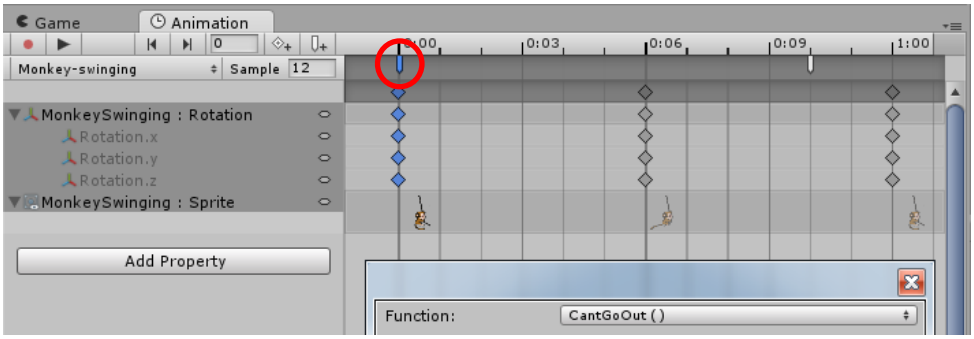
Cuando el personaje jugable pasa por el agujero que ha abierto el rinoceronte en el muro, se activa otra animación para indicar que se ha ganado mostrando el mensaje “WE MADE IT!!!”. La animación consiste simplemente en ampliar la escala del panel que aparece para que parezca que las teclas surgen desde un punto y se acercan. El tipo de elementos que conforman el panel ya se ha mencionado en apartados anteriores (panel, imagen y texto).

De forma análoga, se creó un panel con su animación para el caso en el que se perdían todas las vidas.



Otra de las animaciones añadidas es la del mono columpiándose para salir de su hábitat. Este evento se ha controlado de forma menos elaborada que en su anterior versión debido a la falta de tiempo para desarrollar todos los sprites que conllevaría la versión completa (chocar contra la pared, contra el árbol y saltar). En esta demo, las animaciones que vemos son las del mono columpiándose y escapando siempre se salte mientras éste se columpia hacia delante.

El control de si se puede salir o no se realiza mediante una variable que se modifica en la propia animación llamando a los métodos “CanGoOut()” o “CantGoOut()” según corresponda. En la imagen siguiente se muestra la función asociada al primer evento.



Los métodos CanGoOut y CantGoOut se encuentran en un script asociado al objeto que tiene la animación, ya que de otra forma no serían accesibles para ella.

```
using UnityEngine;
using System.Collections;

public class SwingController : MonoBehaviour {
    private bool canGoOut;

    // Use this for initialization
    void Start () {
        canGoOut = false;
    }

    public void CanGoOut(){
        canGoOut = true;
    }

    public void CantGoOut(){
        canGoOut = false;
    }

    public bool GoOut(){
        return canGoOut;
    }
}
```

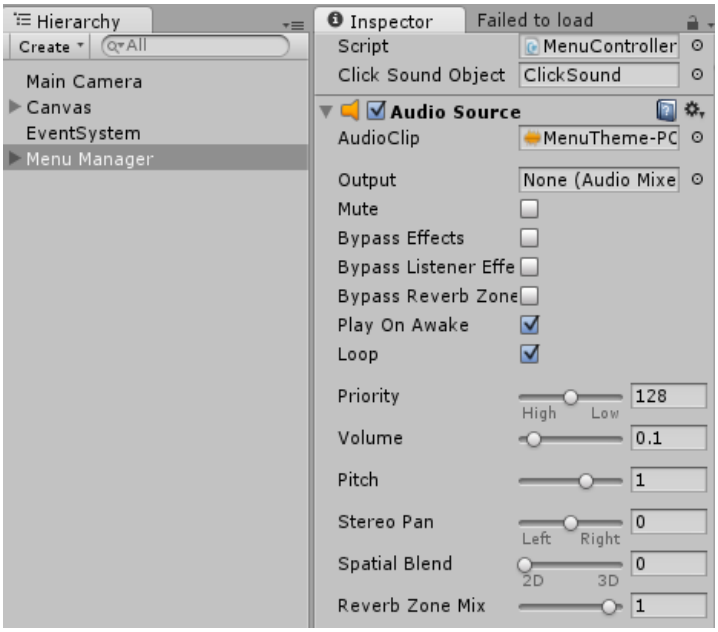
En el script que contiene el detector para saber si el jugador pulsa el botón/tecla de acción cerca del columpio, se controlan el resto de los eventos. Si el mono no se está columpiando cuando el jugador pulsa el botón de acción en la zona establecida, se inicia la animación del mono columpiándose; si el mono ya se estaba columpiando, se comprueba si puede salir mediante el método GoOut del script SwingController. La forma de acceder al método GoOut se descubrió recientemente, en ocasiones anteriores siempre se empleaba el método “SendMessage” para interaccionar entre scripts. Si el mono puede salir, se inicia la animación del mono saliendo, de lo contrario, sólo se detiene la animación del mono columpiándose y se regresa al estado anterior a columpiarse.

```
            if (swinging) {
                bool canGoOut = monkeySwinging.GetComponent<SwingController>().GoOut();
                if (canGoOut) {
                    monkeySwinging.SetActive(false);
                    transform.parent.GetComponent<SpriteRenderer>().enabled = true;
                    monkeyGoingOut.SetActive (true);
                    Invoke ("GoOut",1.35f);
                } else Swinging(false);
            } else {
                Swinging(true);
            }
        }
    }
}

private void Swinging(bool state){
    swinging = state;
    monkeySwinging.SetActive(state);
    transform.parent.GetComponent<SpriteRenderer>().enabled = !state;
    GameObject.Find("Player Manager").SendMessage("MoveEnabled", !state);
    currentCharacterCollider.GetComponent<SpriteRenderer>().enabled = !state;
}

private void GoOut(){
    GameObject.Find("Player Manager").SendMessage("MoveEnabled", true);
    GameObject.Find ("Game Manager").SendMessage ("ChangeScene", 2);
}
```

8.9. Música y efectos sonoros

Para obtener los sonidos que se emplearían en el juego se realizó una búsqueda de los mismos entre distintas páginas con contenido gratuito. Los links donde se obtuvieron dichos sonidos, donde se puede saber quién es el autor, se encuentran en la sección “Música y efectos sonoros” de la bibliografía.

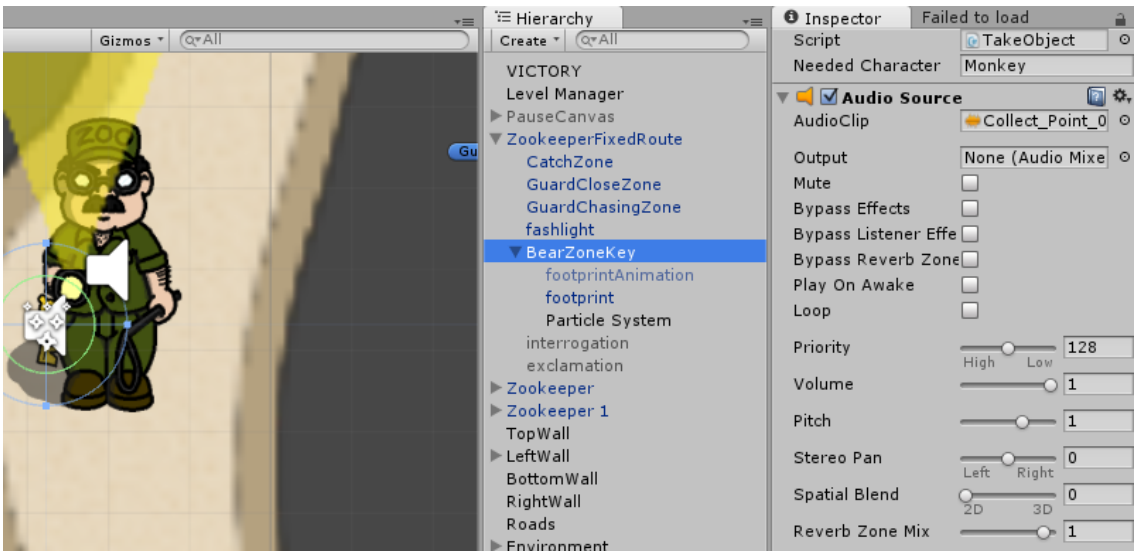
Una vez importados los sonidos, éstos pueden ser adjuntados a cualquier objeto del juego como componentes “AudioSource”.



En el caso de la música del juego, ésta se ha añadido a cada escena agregándola como un componente del Menu Manager, del Level Manager o del Credits Manager según corresponda, indicando que debe reproducirse en bucle y desde el principio. El tema no es el mismo en todas las escenas.

En el caso de los efectos sonoros, queremos que se reproduzcan en momentos muy concretos, por lo que tendremos que indicar cuándo se inician mediante scripts.

Un ejemplo de ello sería el momento en el que se coge un ítem, al que se le ha querido añadir un sonido para indicar al jugador este hecho. Para concretar, nos centraremos en el momento en el que se obtiene la llave del hábitat del oso. Al objeto que contiene el sprite de la llave se le ha añadido el componente con el sonido que indica que se ha cogido un ítem.

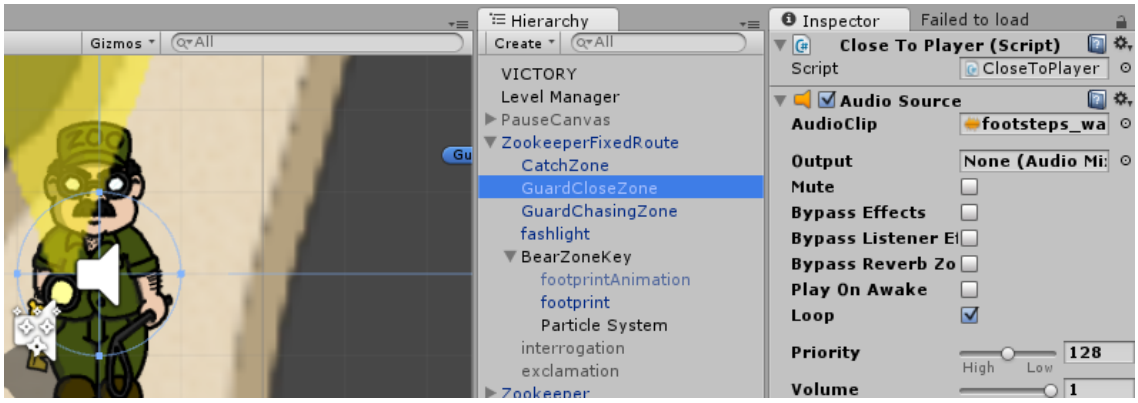


Ahora, desde el script presente en el mismo objeto, podemos obtener el componente “AudioSource” mediante la instrucción “GetComponent<AudioSource>()”.

Una vez tengamos el AudioSource, podremos iniciar o detener el sonido mediante los métodos “Play()” y “Stop()”, respectivamente. En este caso y como este sonido no se reproduce en un bucle, sólo tendremos que preocuparnos de iniciarlo cuando se coge la llave.

```
// Update is called once per frame
void Update () {
    if (currentCharacterCollider !=null && rightCurrentCharacter){
        if(Input.GetKeyDown(KeyCode.E) || Input.GetMouseButtonDown(0)){
            Take();
            State.bearZoneKeyTaken = true;
            Destroy(this.gameObject);
        }
    }
}

private void Take(){
    Debug.Log("Key taken");
    GetComponent<AudioSource>().Play();
    ItemUI item = new ItemUI ("BearKey",GetComponent<SpriteRenderer>().sprite,1);
    GameObject.Find ("Stock Manager").SendMessage("AddItem",item);
    State.bearZoneKeyTaken = true;
}
```

Un caso de efecto sonoro en bucle sería el de los pasos del guardia. Se ha añadido un sonido de pasos para el momento en el que el jugador está cerca del guardia, incluso aunque aún no haya sido visto.



En este caso, sí se detendrá la reproducción del sonido con el método Stop cuando el jugador salga de la zona de proximidad con el enemigo.

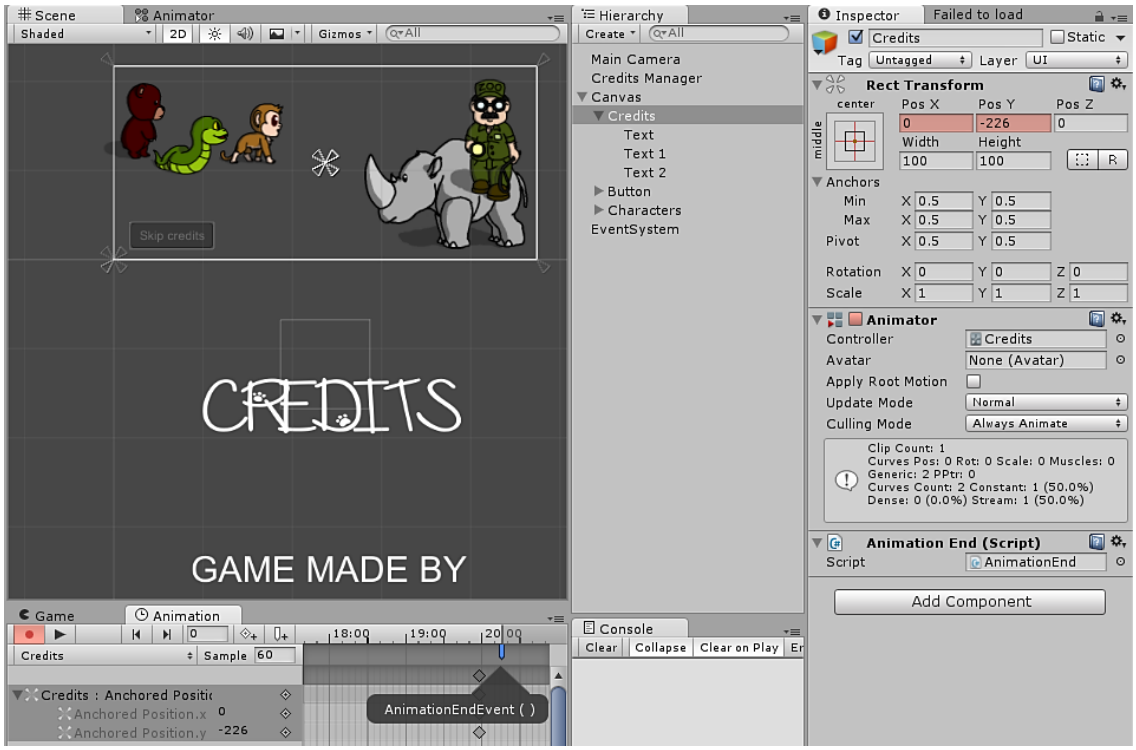
```
void OnTriggerEnter2D (Collider2D other){
    if (other.tag == "CurrentPlayer"){
        GetComponent<AudioSource>().Play();
        Debug.Log("Player close to enemy");
    }
}

void OnTriggerExit2D (Collider2D other){
    if (other.tag == "CurrentPlayer"){
        GetComponent<AudioSource>().Stop();
        Debug.Log("Player far from enemy");
    }
}
```

El resto de efectos sonoros utilizados son los siguientes:

- Cristales rotos. En el momento en el que el mono rompe la pecera de la serpiente para liberarla.
- Golpe seco. En el momento en el que el mono golpea la pared del almacén para abrir un agujero.
- Golpe con rocas. En el momento en el que el rinoceronte rompe el muro del zoo.
- Pisadas rápidas: En el momento en el que el rinoceronte está corriendo para escapar del zoo.
- Alerta. Cuando el personaje activo es detectado por un enemigo y comienza la persecución.
- Clic. Cuando se pulsan botones de la interfaz.
- Victoria. Cuando se vence en el juego.
- Derrota. Cuando se pierden todas las vidas en el juego.

8.10. Los créditos

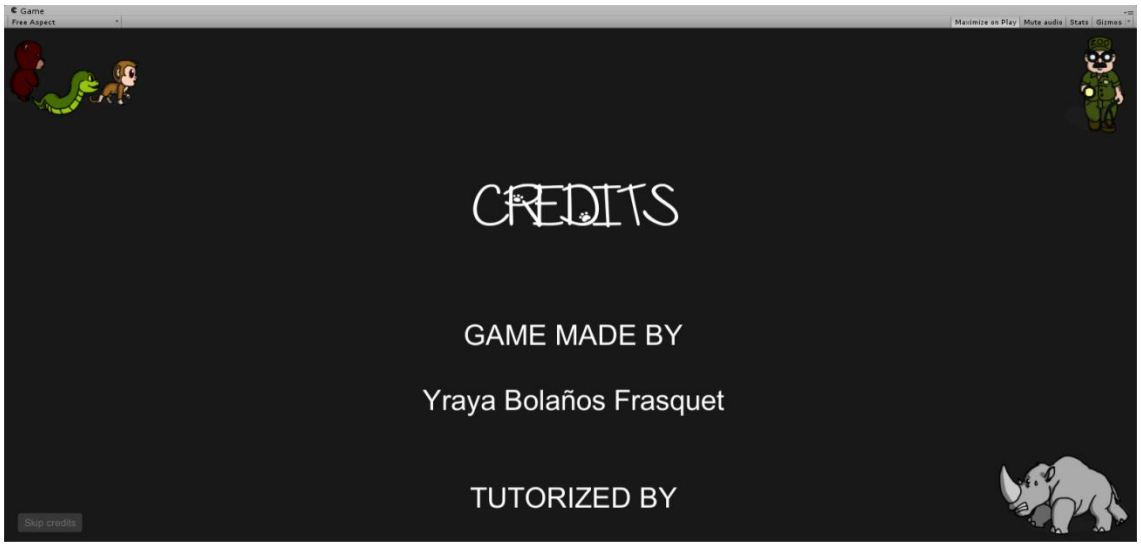
Para implementar los créditos se emplearon elementos ya conocidos de la UI (Canvas, Image,Text y Button), sonidos y animaciones. Se creó una animación que desplazara el texto de los créditos hacia arriba y que al finalizar invocara a un método para indicar que había terminado y que conllevaría que se cargara el menú de inicio. También se añadieron animaciones de los personajes caminando a modo decorativo.




```
using UnityEngine;
using System.Collections;

public class AnimationEnd : MonoBehaviour {

    public void AnimationEndEvent(){
        GameObject.Find ("Credits Manager").SendMessage ("AnimationEnd");
    }
}
```

```
using UnityEngine;
using System.Collections;

public class CreditsController : MonoBehaviour {
    private AudioSource clickSound;

    void Start(){
        clickSound = GetComponent<AudioSource> ();
    }

    public void SkipCredits(){
        clickSound.Play ();
        AnimationEnd();
    }

    public void AnimationEnd(){
        Application.LoadLevel (0);
    }
}
```

A continuación se muestra una captura con el resultado final.

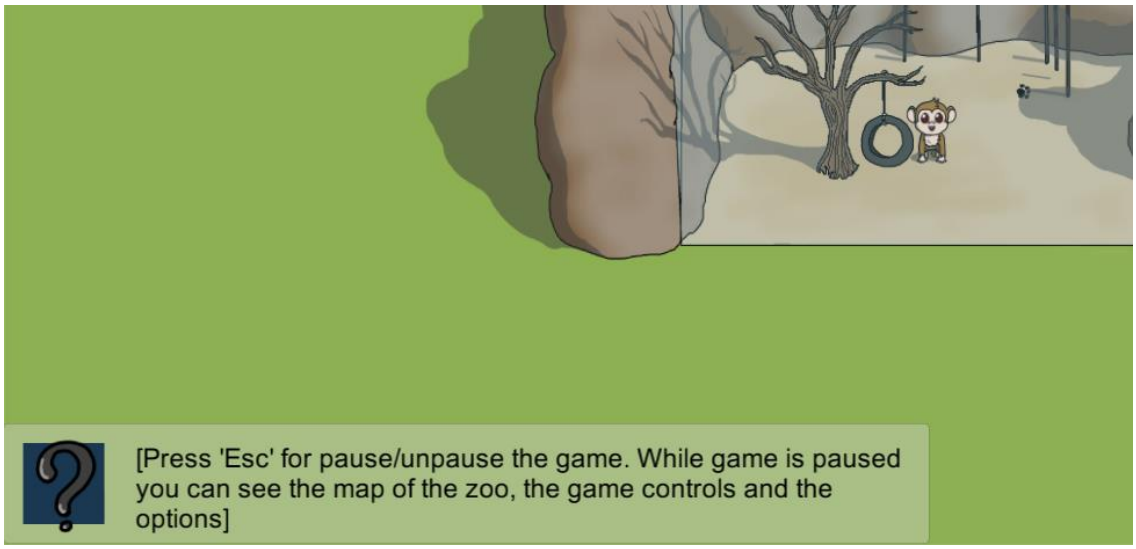


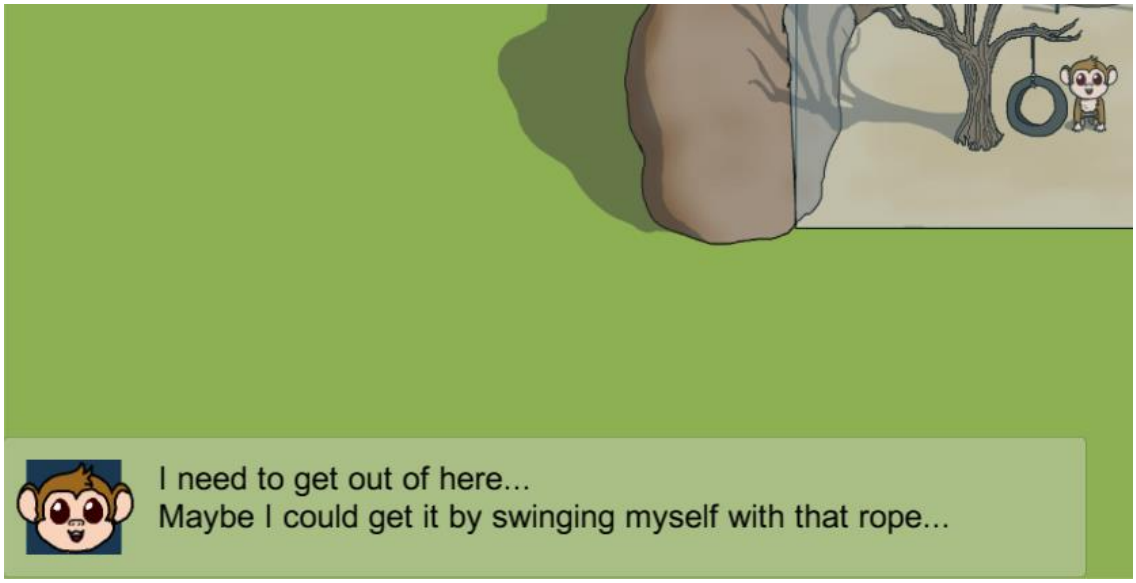
8.11. Sistema de ayuda

Dado que los pasos a seguir en el juego pueden resultar confusos para un jugador novel, se ha implementado un sistema de ayuda mediante mensajes que pueden mostrarse pulsando un botón “HELP” en la esquina inferior izquierda de la pantalla.

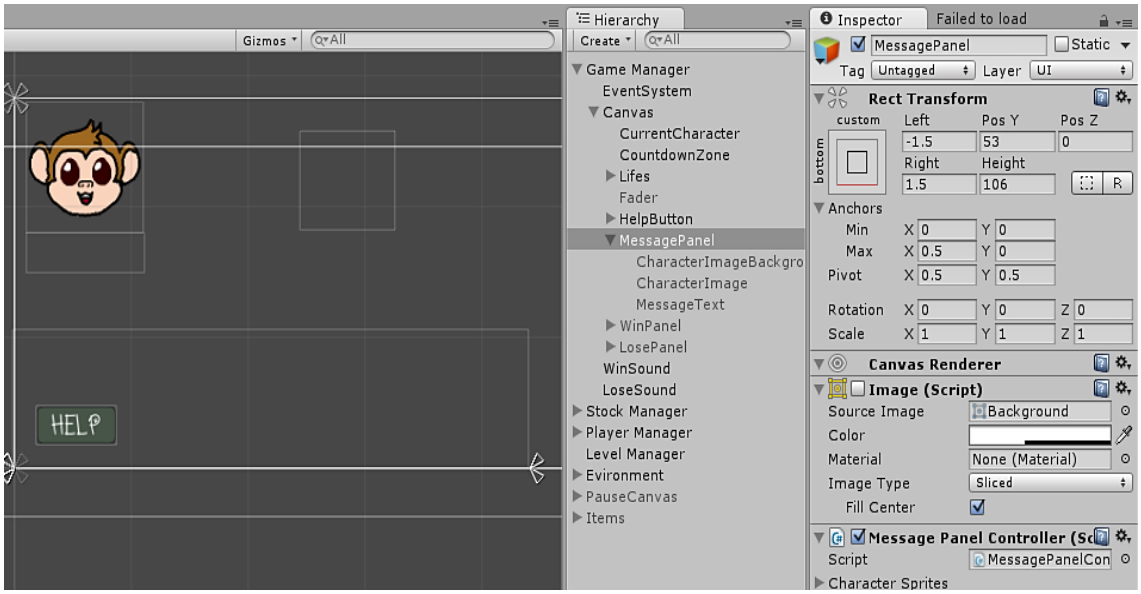
Al inicio de la partida, aparece un mensaje explicando cómo se pausa el juego y los elementos que se muestran cuando éste está pausado. A partir de ahí se mostrarán algunos mensajes según sucedan eventos en el juego (por ejemplo cuando intentamos liberar a un animal sin tener el objeto necesario) o si el jugador pulsa el botón de ayuda. Los mensajes serán una guía sobre lo que se debería hacer a continuación según la opinión del personaje inicial (el mono).

En la siguiente captura aparece el mensaje explicativo inicial, a continuación una captura mostrando el botón HELP y por último una captura en la que se ha pulsado el botón de ayuda.





Para implementar el sistema de ayuda se ha creado un panel con las imágenes necesarias y el texto del mensaje. A dicho panel se le ha agregado un script llamado “messagePanelController” que contiene las variables y los métodos necesarios para su utilización.



Para mostrar un mensaje, se tendrá que indicar quién es el personaje que habla, cuál es el mensaje a mostrar y durante cuánto tiempo debe mostrarse.

```
public void ShowMessage(string speakerName, string message, float duration){
    foreach (Sprite speakerSprite in characterSprites){
        if (speakerSprite.name == speakerName){
            speakerImage.GetComponent<Image>().sprite = speakerSprite;
            break;
        }
    }
    speakerMessage.GetComponent<Text> ().text = message;

    MessageEnable (true);
    Invoke ("DisableMessage", duration);
}

private void MessageEnable(bool enable){
    showingMessage = enable;
    GetComponent<Image> ().enabled = enable;
    speakerBackgroundImage.SetActive (enable);
    speakerImage.SetActive (enable);
    speakerMessage.SetActive (enable);
}

private void DisableMessage(){
    MessageEnable (false);
}
```

Se podrá saber si hay un mensaje mostrándose en el momento actual mediante el método “ShowingMessage”.

```
public bool ShowingMessage(){
    return showingMessage;
}
```

El encargado de mostrar los mensajes de ayuda cuando el usuario pulse el botón HELP será el Game Manager. Para ello se definen una serie de mensajes de ayuda para cada momento. El mensaje a mostrar se decide en base al estado del juego controlado por distintas variables del método State. A esta clase se le han añadido algunas variables nuevas para este propósito, tales como saber si el jugador ha intentado liberar a algún animal con o sin éxito.

El método del ManagerController que se invoca cuando el jugador pulsa el botón de ayuda se llama, como no podía ser de otro modo, Help, que se apoya a su vez en otro método privado encargado de interactuar con el panel de mensajes.

```
public void Help(){
    if (!messagePanel.GetComponent<MessagePanelController> ().ShowingMessage ()) {
        if (!State.monkeyZoneFirstEscape){
            ShowMessage(initialMessage);
        } else if(!State.snakeVisited) {
            ShowMessage(freeSnakeMessage);
        } else if (!State.hammerTaken){
            ShowMessage(takeTheHammerMessage);
        }else if (!State.snakeAdded){
            ShowMessage(breakSnakeTankGlassMessage);
        } else if (!State.bearVisited){
            ShowMessage(freeBearMessage);
        } else if (!State.bearZoneKeyTaken){
            ShowMessage(getBearZoneKeyMessage);
        } else if (!State.bearAdded){
            ShowMessage(openTheBearHabitatDoorMessage);
        } else if (!State.rhinoVisited){
            ShowMessage(freeRhinoMessage);
        } else if (!State.wallBroken){
            ShowMessage(breakWarehouseWallMessage);
        } else if (!State.warehouseDoorOpen){
            ShowMessage(openWarehouseDoorMessage);
        } else if (State.boxesThrown == 0){
            ShowMessage(throwTheBoxesMessage);
        } else if (!State.rhinoFreed){
            ShowMessage(keepThrowingBoxesMessage);
        } else {
            ShowMessage(escapeMessage);
        }
    }
}

private void ShowMessage(string message){
    messagePanel.GetComponent<MessagePanelController>().ShowMessage("MonkeyHead", message, 5);
}
```

Para mostrar el mensaje explicativo inicial, se invoca en el Start del GameController al método “ShowInitialMessage()”.

```
private void ShowInitialHelMessage(){
    string message = "[Press 'Esc' for pause/unpause the game. " +
        "While game is paused you can see the map of the zoo," +
        "the game controls and the options]";
    messagePanel.GetComponent<MessagePanelController>().ShowMessage("InfoImage", message, 8);
}
```

Aunque el Game Manager sea el que más hace uso del panel de mensajes, cualquier script puede utilizarlo para indicar que un personaje dice algo.

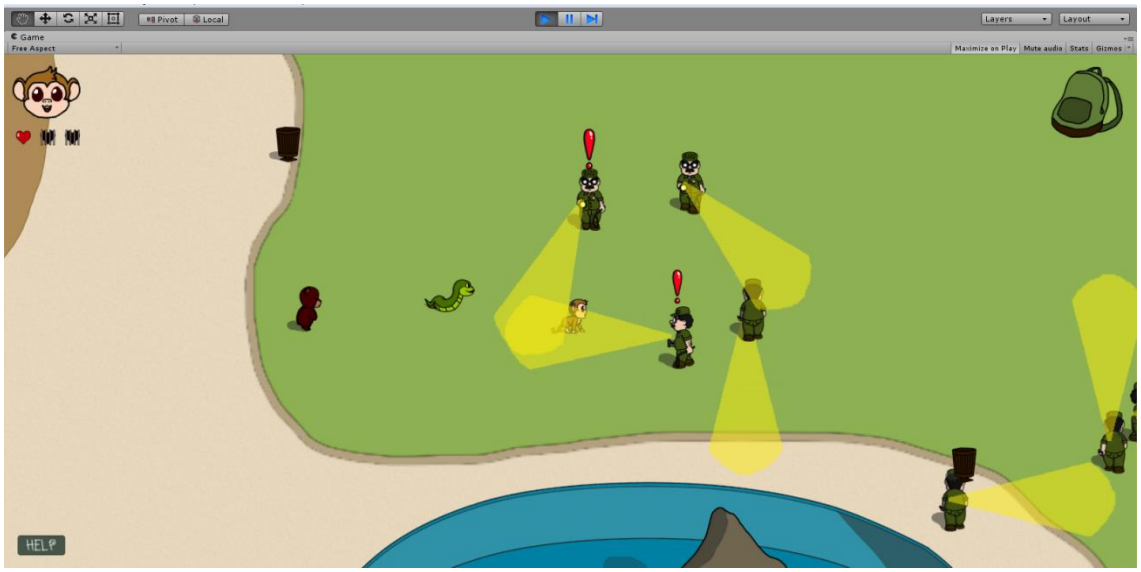
8.12. Agregar enemigos a medida que avanza el juego

Uno de los defectos de este juego es que los enemigos no son una gran amenaza, debido a que son pocos y fáciles de esquivar. Una de las soluciones que se ha implementado para este problema consiste en generar más enemigos a medida que se completan distintas etapas. Para añadir enemigos se ha creado un script en el Level Manager de la escena del zoo llamado “ZookeeperAddition” que dado el prefab del enemigo y una posición hace aparecer tres enemigos en dicha posición por cada animal liberado.

```
using UnityEngine;
using System.Collections;

public class ZookeeperAddition : MonoBehaviour {
    public GameObject zookeeperPrefab;
    public GameObject spawnPostion;

    void Start () {
        if (State.snakeAdded) SpawnEnemy (3);
        if (State.bearAdded) SpawnEnemy (3);
    }

    private void SpawnEnemy(int numberOfEnemies){
        for (int i = 0; i < numberOfEnemies; i++)
            Instantiate(zookeeperPrefab,spawnPostion.transform.position, Quaternion.identity);
    }
}
```

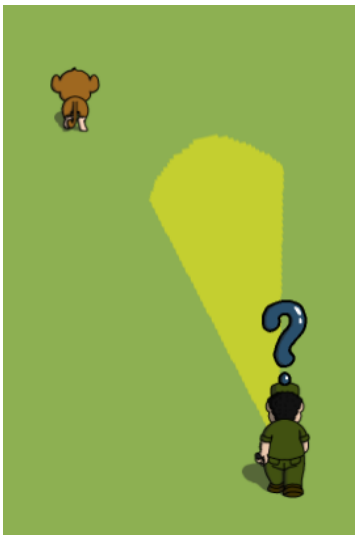
Esta es una versión sencilla del script, en un futuro, cuando se implementen los rescates de más animales, se tendrá una variable que indicará el número de animales rescatados y los enemigos que se deben generar por cada uno, incluso se podría crear un sistema de dificultad y hacer aparecer más enemigos o menos en función del nivel elegido por el jugador.

Las posibilidades de encontrarse con un enemigo aumentan notoriamente tras la implementación de este apartado.



Otras posibilidades para aumentar la dificultad del juego serían incrementar el ratio de detección de los guardias y/o su velocidad.

8.13. Otros elementos añadidos

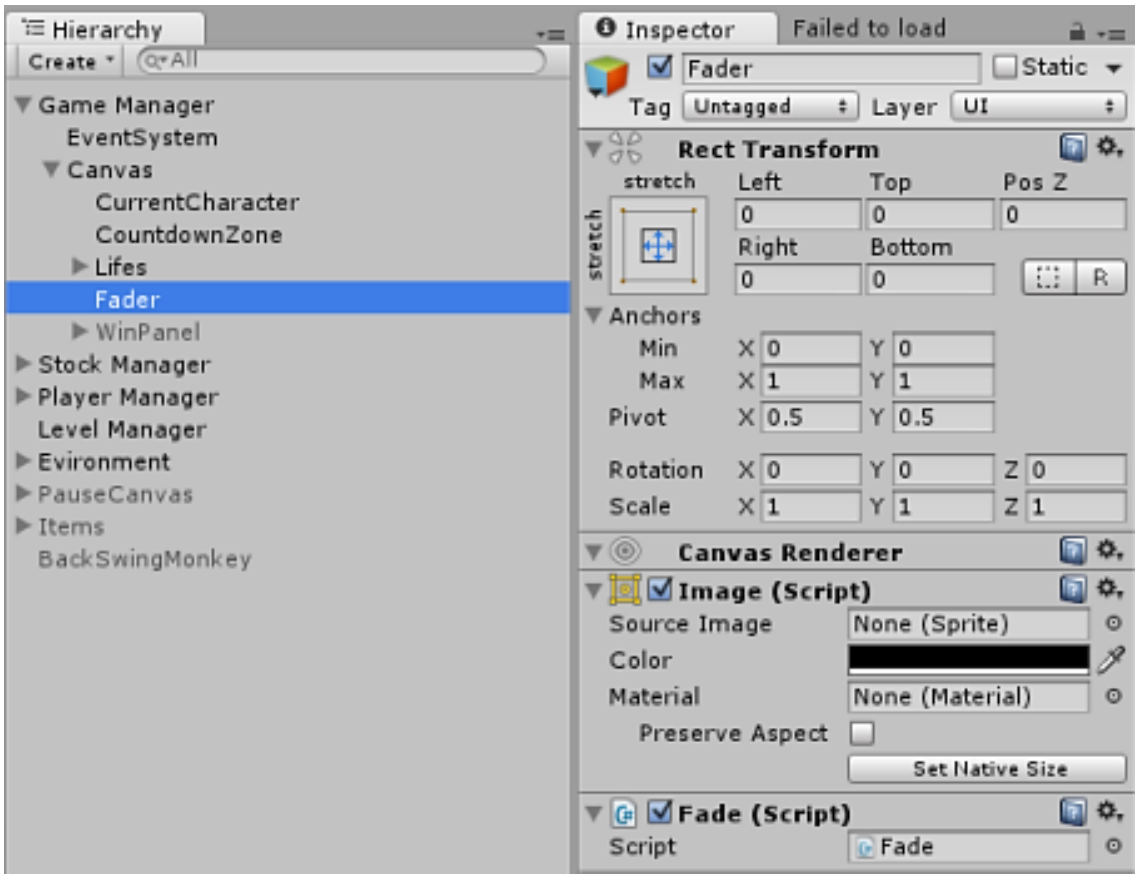


Con intención de que el jugador sepa fácilmente cuándo ha sido descubierto por un guardia y cuándo éste le ha perdido la pista, se añadió un sprite con una exclamación en rojo que se mostrará sobre la cabeza del guardia cuando el jugador sea detectado junto con un efecto sonoro de alerta y un sprite con una interrogación en azul que se mostrará cuando el guardia deje de seguir al jugador.

Para implementar esta funcionalidad sólo fue necesario importar los sprites, añadirlos a los objetos que controlan la “zona de persecución” y activar dichos sprites cuando el personaje activo entra o sale de esa zona según corresponda, así como el efecto sonoro cuando se entra en la zona.

La detección de los ítems que se pueden obtener se facilitó añadiéndoles un sistema de partículas a cada uno.

Como detalle gráfico, se ha añadido un efecto “fade in” al inicio de cada partida y un efecto “fade out” cuando se produce una victoria o una derrota. Para crear estos efectos se utiliza un objeto con una imagen de color negro que ocupa toda la pantalla y un script de control que emplea el método “CrossFadeAlpha”.



El método “CrossFadeAlpha” posee tres parámetros, con el primero indicaremos el grado de opacidad objetivo de 0 a 1 donde 1 sería la imagen completamente opaca, con el segundo especificaremos la duración del fade y con el tercero si éste ignora el Time.Scale o no.

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class Fade : MonoBehaviour {
    private Image faderImage;

    // Use this for initialization
    void Start () {
        faderImage = GetComponent<Image> ();
    }

    public void FadeIn(){
        faderImage.CrossFadeAlpha (0, 1.5f, false);
    }

    private void FadeOut(){
        faderImage.CrossFadeAlpha (1, 1.5f, false);
    }
}
```

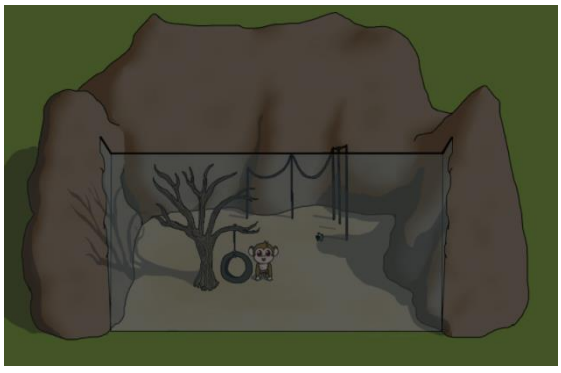
El script que usará el “fader” será el GameController. Al inicio del juego:

```
void Start () {
    currentLifes = initalLives;
    lifePanel.SendMessage ("SetInitialLifes",initalLives);
    fader = GameObject.Find ("Fader");
    if (fader != null) fader.SendMessage ("FadeIn");

    ShowInitialHelMessage ();
}
```

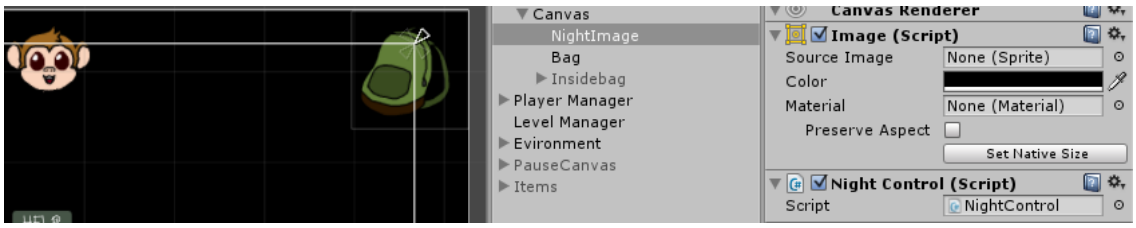
y en los eventos de victoria y derrota:

```
public void Lose(){
    Debug.Log ("YOU LOSE!!!");

    loseSoundObject.GetComponent<AudioSource> ().Play(); //Sonido de derrota
    losePanel.SetActive (true);
    fader.SendMessage ("FadeOut");
    Invoke ("UpdateAfterLose",3);
}

private void UpdateAfterLose(){
    ResetGame ();

    currentLifes = initalLives;
    lifePanel.SendMessage ("SetInitialLifes",initalLives);
    GameObject.Find("CurrentCharacter").SendMessage("SetCurrentCharacterImage","Monkey");

    Application.LoadLevel (1);
    losePanel.SetActive (false);
    if (fader != null) fader.SendMessage ("FadeIn");
}

public void Win(){
    winSoundObject.GetComponent<AudioSource> ().Play(); //Sonido de victoria

    winPanel.SetActive (true);
    fader.SendMessage ("FadeOut");
    Debug.Log ("YOU WIN");
    Invoke ("LoadCredits",3);
}
```

A continuación se muestran tres capturas en diferentes momentos del “fade in”.

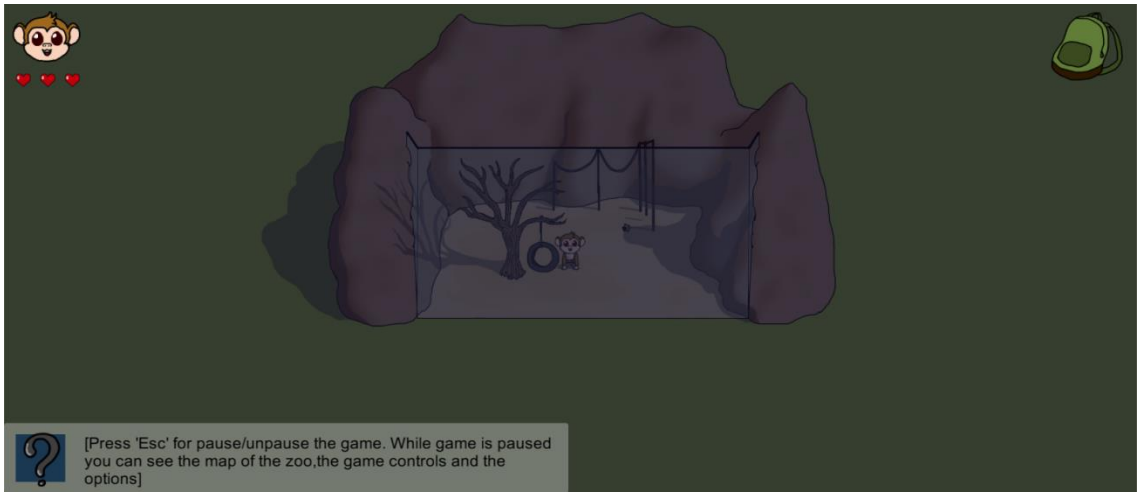


Por último, para añadir un efecto de oscuridad, ya que se supone que el juego se desarrolla pro la noche y por eso los guardias llevan linternas y no ven bien, se utilizó también una imagen que cubría toda la escena y a la que se asignó mediante scripting un color y transparencia concretos.




```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class NightControl : MonoBehaviour {

    void Start () {
        GetComponent<Image>().color = new Color32 (10,5,30,170);
    }
}
```

En las siguientes capturas se muestra el juego tras aplicarle esta imagen con transparencia y un ligero tono azulado.



9. Normativa y legislación

El crecimiento de las tecnologías de la información ha generado una gran cantidad de nuevas posibilidades a la hora de delinquir. Ya que el presente trabajo se desarrolla en éste ámbito, nos centraremos en aquellos aspectos de la ley que se refieren a los delitos informáticos.

El delito informático es aquel que alude a actividades ilícitas realizadas por medio de dispositivos tecnológicos como ordenadores, smartphone, portátiles, tabletas o de internet, aunque la gran proliferación de dispositivos conectados a ésta última hace que el concepto se extienda hasta las televisiones (smartTV), consolas de videojuego, incluso a coches, aparatos de navegación aérea, acuática o terrestre e incluso sistemas de geoposicionamiento GPS, casas conectadas a internet (domótica) o dispositivos para vestir (wearable).

A los riesgos para la privacidad o el espionaje ya conocidos, se añaden otros delitos tradicionales como fraude, robo, estafa, chantaje, falsificación y malversación de caudales públicos. Los perjuicios ocasionados por este tipo de delitos son superiores a la delincuencia tradicional y también es mucho más difícil descubrir a los culpables.

La Organización de Naciones Unidas (ONU) reconoce los siguientes tipos de delitos informáticos:

- Fraudes cometidos mediante manipulación de ordenadores.
- Manipulación de los datos de entrada.
- Daños o modificaciones de programas o datos computarizados.

En España, los delitos cometidos por medios informáticos se condenan de igual forma que sus análogos en la vía pública, como por ejemplo los insultos y amenazas. Las sanciones para este tipo de delitos se recogen en la **Ley-Orgánica 10/1995**, de 23 de Noviembre/BOE número 281, de 24 de Noviembre de 1.995.

A nivel europeo, a fin de regular los delitos concernientes al ámbito informático, se redactó en Budapest el **Convenio sobre la Ciberdelincuencia** el 23 de noviembre de 2001. Dicho convenio fue firmado por España en el mismo día y ciudad (Disposición 14221 del BOE núm. 226 de 2010).

El “Convenio sobre la Ciberdelincuencia" permitió la definición de los delitos informáticos y algunos conceptos relacionados con los mismos. Los delitos de este tip fueron clasificados en cuatro grupos:

- Delitos contra la confidencialidad, la integridad y la disponibilidad de los datos y sistemas informáticos.
    - Acceso ilícito a sistemas informáticos.
    - Interceptación ilícita de datos informáticos.
    - Interferencia en el sistema mediante la introducción, transmisión, provocación de daños, borrado, alteración o supresión e estos.
    - Abuso de dispositivos que faciliten la comisión de delitos
- Delitos informáticos.
    - Falsificación informática que produzca la alteración, borrado o supresión de datos informático que ocasionen datos no auténticos.
    - Fraudes informáticos.

- Delitos relacionados con el contenido
    - Delitos relacionados con la pornografía infantil.
- Delitos relacionados con infracciones de la propiedad intelectual y derechos afines.

En el ámbito internacional no son muchos los países que cuentan con una legislación apropiada. Entre aquellos que sí la poseen se encuentran España, Francia, Alemania, Inglaterra, Holanda, Estados Unidos y Chile.

- **Francia.** En Francia, la Ley 88/19 del 5 de enero de 1988 sobre el fraude informático contempla:
    - Acceso fraudulento a un sistema de elaboración de datos.
    - Sabotaje Informático.
    - Destrucción de datos.
    - Falsificación de documentos informatizados.
- **Alemania.** En Alemania, para hacer frente a la delincuencia relacionada con la informática, el 15 de mayo de 1986 se adoptó la Segunda Ley contra la Criminalidad Económica. Esta ley reforma el Código Penal (art. 148 del 22 de diciembre de 1987) para contemplar delitos como el espionaje de datos, la estafa informática, la falsificación de datos probatorios, la alteración de datos y el sabotaje informático entre otros.
- **Inglaterra.** Tras varios casos de hacking, **e**n Agosto de 1990, comenzó a regir la Computer Misuse Act (Ley de abusos Informáticos) por la cual cualquier intento, exitoso o no, de de alterar datos informáticos con intención criminal se castiga con hasta cinco años de cárcel y multas de cuantías si límite en función del delito cometido.
- **Holanda.** Hasta el día 1 de marzo de 1993, día en que entró en vigencia la Ley de Delitos Informáticos, Holanda era un paraíso para los hackers. Esta ley contempla con artículos específicos sobre técnicas de Hacking y Phreacking, y los virus están considerados de manera especial.
- **Estados Unidos.** Este país adoptó en 1994 el Acta Federal de Abuso Computacional que modificó al Acta de Fraude y Abuso Computacional de 1986. En julio del 2000, se establece el Acta de Firmas Electrónicas en el Comercio Global y Nacional.
- **Chile.** Chile fue el primer país latinoamericano en sancionar una Ley contra Delitos Informáticos. La ley 19.223, publicada en el Diario Oficial, el 7 de junio de 1993 señala que la destrucción o inutilización de un sistema de tratamiento de información puede ser castigado con prisión de un año y medio a cinco. Como no se estipula la condición de acceder a ese sistema, puede encuadrarse a los autores de virus. Si esa acción afectara los datos contenidos en el sistema, la prisión se establecería entre los tres y los cinco años. El hacking, definido como el ingreso en un sistema o su interferencia con el ánimo de apoderarse, usar o conocer de manera indebida la información contenida en éste, también es pasible de condenas de hasta cinco años de cárcel; pero ingresar en ese mismo sistema sin permiso y sin intenciones de ver su contenido no constituye delito. Dar a conocer la información almacenada en un sistema puede ser castigado con prisión de hasta tres años, pero si el que lo hace es el responsable de dicho sistema puede aumentar a cinco años.

9.1. Leyes que afectan al proyecto

Dado que en el presente proyecto no se manejan datos de carácter personal ni se manipula software o dispositivos de terceros, los aspectos legales se resumen al uso de las licencias de los programas empleados y de demás recursos utilizados (gráficos, sonidos, plugins, tutoriales,..).

En lo que se refiere a **Unity**, la licencia que se ha empleado para desarrollar este proyecto es la licencia gratuita denominada Unity Personal (Acuerdo de licencia de software de Unity). Esta versión no puede ser utilizada por:

- una Entidad comercial que haya A) alcanzado ingresos brutos que superen los USD 100,000, o B) recaudado fondos (incluyendo per no limitado a financiación colectiva) que superen los USD 100,000, en cada caso durante el año fiscal más reciente;
- una Entidad no comercial con un presupuesto total anual que supere los USD 100,000 (para toda la Entidad no comercial (no solo un departamento)) para el año fiscal más reciente; o
- una persona (que no actúe en nombre de una Persona jurídica) o un Propietario único que haya alcanzado ingresos brutos anuales que superen los USD 100,000 por el uso del Software Unity durante el año fiscal más reciente, que no incluya ningún ingreso devengado que no esté relacionado con el uso del Software Unity por parte dicha persona.

Tanto el software de **Blender** como el **de Sweet Home 3D** se encuentran bajo la Licencia Pública General de GNU en inglés GNU General Public License (GPL, or "free software"). Esta licencia otorga las siguientes libertades:

- Los usuarios son libres de usar el software para cualquier propósito
- Los usuarios tienen la libertad para distribuir el software
- Los usuarios pueden estudiar cómo funciona el software y modificarlo
- Los usuarios pueden distribuir versiones modificadas del software

La música correspondiente al tema principal del juego, el menú y los créditos se encuentra bajo la licencia de Creative Commons Attribution 3.0, que permite copiar y redistribuir el material en cualquier medio o formato así como remezclar, transformar y construir sobre el material, siempre y cuando se dé crédito a los autores, se proporcione un enlace a la licencia y se indique si se han realizado cambios.

La música correspondiente al tema que se escucha en las distintas zonas del juego al margen de la principal (almacén, casa del guardia y hábitats) así como los efectos sonoros de pasos, clicks, aviso de persecución, victoria, derrota y recoger objetos fueron obtenidos de la Asset Store de Unity, que permite su utilización sólo como componentes integrados de un juego y en quipos que pertenezcan al usuario final o a sus empleados pertenecientes al mismo sitio (Asset Store Provider Agreement).

Los efectos sonoros correspondientes al golpe en la pared del almacén y en el muro se encuentran bajo el Free Sound Effects End user licence agreement, que nos permite usar estos elementos en nuestro juego siempre que se dé crédito a . freesfx.co.uk (http://www.freesfx.co.uk).

Finalmente, el sonido de los cristales rotos se encuentra bajo el Sound Effects License Agreement que nos permite utilizarlo nuestro juego.

El resto de los recursos utilizados se obtuvieron de páginas con contenido gratuito, donde el requiso más exigente para su uso consiste en dar crédito de a los creadores de los mismos. Por este motivo, dichas páginas han sido referenciadas en la bibliografía, aun cuando la mayoría de los recursos no han sido empleados en la demo final del juego.

10. Conclusión y mejoras futuras

La realización del presente trabajo ha permitido a la alumna obtener gran cantidad de conocimientos relacionados con las fases de desarrollo de un videojuego así como desmontar ideas preconcebidas sobre las mismas. También se han aprendido a manejar nuevas herramientas software y a conseguir los assets necesarios con pleno conocimiento de las licencias bajo las que se ofrecen, as se ha afianzado información y habilidades ya adquiridas.

En proyectos futuros se intentará hacer estimaciones temporales más precisas, dedicar más tiempo a las fases de desarrollo y análisis de propuestas y obtener más feedback de posibles usuarios, tanto durante las citadas fases como durante la de prototipado.

La limitación del tiempo ha dejado muchas ideas en el tintero y algunos bugs por corregir. Algunas de las características que no se han implementado son las siguientes:

- Guardar partida. Actualmente y debido a la dificultad adicional que suponía no se ha podido implementar el sistema de guardado.
- Sistema de logros. Aunque más sencillo de implementar que la opción de guardar partida, el sistema de logros se ha dejado de lado en esta demo para implementar características más importantes o mucho más sencillas de desarrollar. Algunos de los logros a conseguir podrían ser los siguientes:
    - Pasarse el juego por primera vez
    - Encontrar X cantidad de ítems de un mismo tipo
    - Pasar a menos de X distancia de un guardia sin ser capturado
- Modificadores de velocidad. Una de las ideas que no se ha llegado a implementar es la de emplear objetos para aumentar o disminuir la velocidad a la que se mueve el usuario. Así, por ejemplo, si el personaje pisa un chicle iría más despacio durante un tiempo, pero si se tomara el contenido de una lata de “Wild Cola” obtendría mayor velocidad durante cierto tiempo.
- Sistema de dificultad. Se deseaba crear un sistema que permitiera al usuario seleccionar la dificultad con la que quería jugar. Un nivel bajo contendría una menor cantidad de enemigos y/o mayor número de vidas, mientras que un nivel alto sucedería lo contrario. En el caso de que se implementaran los modificadores de velocidad, en el nivel fácil se generarían más modificadores positivos y menos negativos con respecto al nivel normal y sería al revés en el nivel difícil.
- Añadir más detalles gráficos y puzzles. Pareciera que el mono, la serpiente, el oso y el rinoceronte son los únicos animales del zoo. En un futuro se pretende añadir más animales en los distintos hábitats ya creados y crear zonas nuevas con nuevos animales para liberar.
- Animales coleccionables. Con la adición de nuevos personajes, no se incrementaría el número de animales que puede haber en el equipo. Es decir, aunque podríamos liberar a más personajes, sólo podríamos tener hasta tres en el mismo equipo, los otros o bien serían NPCs (non-player character) o bien tendrían que esperar en sus hábitats u otro lugar hasta que el jugador quisiera sustituir a un miembro del equipo por uno de ellos.

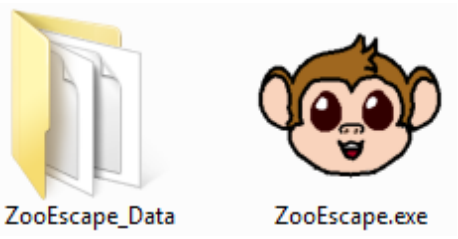
11. Manual de usuario

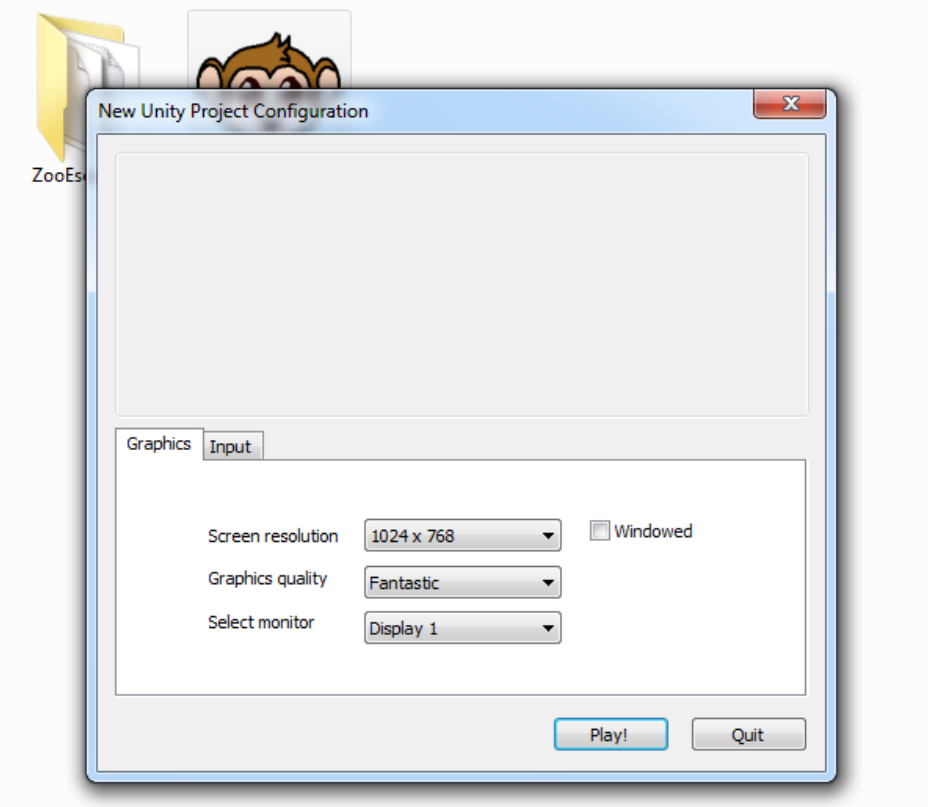
11.1. Contexto y objetivo del juego

Este juego está ambientado en un zoológico al que sus animales no están nada contentos de pertenecer. Procurando siempre no ser capturados por los guardias del zoo, se deberán realizar distintas acciones que concluirán con la liberación de distintos animales con el objetivo último de escapar todos del encierro.

11.2. Ejecutar el juego

Para poder jugar se proporciona un archivo ejecutable y una carpeta que contiene los datos del juego. Haciendo doble clic sobre el archivo ejecutable, se mostrara una ventana emergente en la que se podrán configurar distintas opciones de la pantalla del juego (resolución, la calidad de los gráficos, monitor en el que se visualizará y si queremos jugar en pantalla completa o no). También se podrá cambiar la configuración de los elementos de entrada de datos.





Cuando la configuración sea la deseada, sólo será necesario pulsar el botón “Play!” para jugar y tras una breve aparición del logo de Unity comenzará el juego.

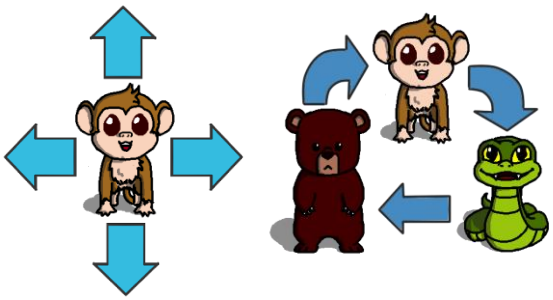
11.3. Menú de inicio

Una vez iniciado el juego, lo primero que aparecerá será el menú de inicio con las posibles acciones para el jugador.

Desde la pantalla de inicio, el jugador puede iniciar una nueva partida, ver los controles del juego o salir de él (las modificaciones necesarias para poder guardar una partida y por consiguiente poder continuarla, así como el apartado de logros no han sido implementados en esta demo).

11.4. Controles

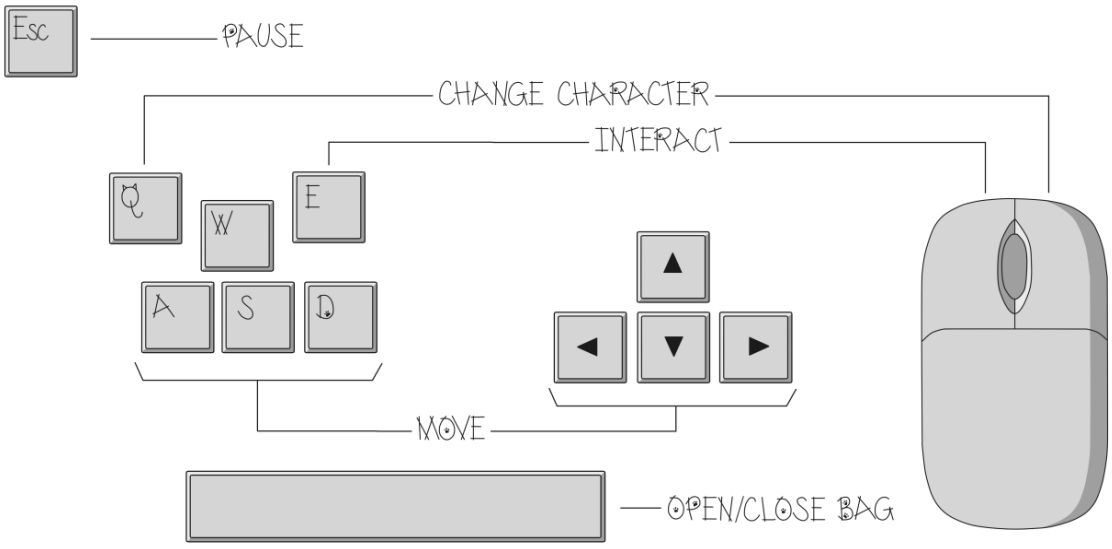
Durante una partida, el jugador tiene las siguientes opciones:



- **Caminar** con su personaje mediante las flechas de dirección y la combinación de teclas WASD.
- **Rotar** entre personajes si se tiene más de un animal en el equipo y no se está implementando ninguna acción que impida la rotación (como en el caso de que un animal esté cargando un objeto). Para rotar se emplea la tecla Q o el botón secundario del ratón.
- **Realizar acciones** siempre que se esté en una zona que lo permita con la tecla E o el botón primario del ratón.

- **Abrir o cerrar** la mochila para mostrar o no respectivamente los objetos conseguidos empleando la barra espaciadora.
- **Pausar/des-pausar el juego** con la tecla Esc. Mientras el juego esté pausado, el jugador podrá
  - ver los controles y el mapa del zoo
  - regresar al juego
  - regresar al menú de inicio
  - salir del juego

Todos los controles del juego se encuentran representados en la imagen que se muestra a continuación:



12. Bibliografía

12.1. Aportaciones, historia y situación actual:

- https://riunet.upv.es/bitstream/handle/10251/45702/Trabajo%20final%20carrera.pdf?sequence=1
- http://boletines.prisadigital.com/LibroBlancoDEV%20alta_compr.pdf
- http://www.dev.org.es/es/publicaciones
- http://lapatriaenlinea.com/?nota=146612
- http://www.elotrolado.net/wiki/Historia_de_los_videojuegos
- http://www.otakufreaks.com/historia-de-los-videojuegos-el-origen-y-los-inicios/
- http://www.fib.upc.edu/retro-informatica/historia/videojocs.html
- https://es.wikipedia.org/wiki/Historia_de_los_videojuegos
- https://es.wikipedia.org/wiki/Videojuego
- http://indicelatino.com/juegos/historia/origenes/
- https://es.wikipedia.org/wiki/Industria_de_los_videojuegos
- http://www.neoteo.com/los-motores-graficos-mas-importantes-de-la-histori/

12.2. Motores de desarrollo de videojuegos

- http://es.slideshare.net/nfmarchetti/evolucin-de-los-motores-grficos-presentation
- https://es.wikipedia.org/wiki/Desarrollo_de_videojuegos
- https://es.wikipedia.org/wiki/Motor_de_videojuego
- http://www.vidaextra.com/listas/si-quieres-hacer-tus-propios-juegos-estos-son-los-mejores-motores-que-vas-a-encontrar
- http://www.vidaextra.com/listas/4-motores-graficos-para-perder-el-miedo-y-lanzarse-al-desarrollo-de-videojuegos
- http://blogthinkbig.com/motores-graficos/
- https://es.wikipedia.org/wiki/Videojuego_independiente
- https://www.unrealengine.com/what-is-unreal-engine-4
- http://www.ozom.cl/unreal-engine-4-unity-5-y-cryengine-los-3-motores-graficos-que-compiten-por-la-supremacia/
- https://es.wikipedia.org/wiki/CryEngine
- https://www.unrealengine.com/branding-guidelines-and-trademark-usage
- https://unity3d.com/es/public-relations/brand
- http://docs.cryengine.com/display/SDKDOC1/Home
- http://www.valvesoftware.com/company/index.html
- http://www.garagegames.com/products/torque-3d
- http://cocos2d.org/
- http://quintagroup.com/cms/python/cocos2d
- http://pcgamingwiki.com/wiki/Engine:UbiArt_Framework
- http://www.stencyl.com/
- http://docs.yoyogames.com/
- http://aijom.blogspot.com.es/2013/09/juego-sencillo-en-game-maker.html
- https://unity3d.com/es/unity

- https://www.yeeply.com/blog/desarrollo-de-juegos-con-unity-3d/
- http://deusexmachina.es/unity-3d-el-motor-indie/
- https://www.yeeply.com/blog/ventajas-e-inconvenientes-de-desarrollar-juegos-con-unity-3d/
- https://www.yeeply.com/blog/comparativa-unity-cocos2d/
- http://thebloodyshadowslan.blogspot.com.es/2013/08/programacion-en-unity-unity-js-vs-c.html
- http://altenwald.org/2012/06/15/lenguaje-boo-y-mono-para-videojuegos/
- https://es.wikipedia.org/wiki/Boo_(lenguaje_de_programaci%C3%B3n)
- http://docs.unity3d.com/es/current/Manual/LearningtheInterface.html
- http://trinit.es/unity/tutoriales/manuales/2%20-%20Introducci%F3n.pdf
- http://docs.unity3d.com/es/current/Manual/animeditor-UsingAnimationEditor.html
- http://docs.unity3d.com/es/current/Manual/Animator.html
- http://docs.unity3d.com/es/current/Manual/MonoDevelop.html
- https://msdn.microsoft.com/es-es/library/system.collections(v=vs.110).aspx
- http://blogs.unity3d.com/es/2014/06/30/unity-4-6-new-ui-world-space-canvas/

12.3. Experimentos

- http://www.sweethome3d.com/es/
- https://www.blender.org/about/
- https://alittlebigof.wordpress.com/2013/04/08/unity-3d-tutorial-para-novatos-2-objetos/
- https://www.youtube.com/watch?v=Op8KiqGoGnI (tutorial taza3D, Blender)

12.4. Tiles y fuentes:

- http://hasgraphics.com/free-tiles/
- http://www.lostgarden.com/2006/02/250-free-handdrawn-textures.html
- http://opengameart.org/content/lots-of-free-2d-tiles-and-sprites-by-hyptosis
- http://opengameart.org/content/lpc-dungeon-elements
- http://crankeye.com/resources/RPG%20Maker%20VX/VX%20Characters/_door0003.png/show
- http://www.rpg-maker.fr/index.php?page=characters&type=monstres
- http://forum.zdoom.org/viewtopic.php?f=37&t=36278
- https://www.iconfinder.com/icons/27880/arrow_red_up_icon#size=128
- https://www.youtube.com/watch?v=_x0bMTxP7Yw (tilemapper)
- http://fuentes.gratis.es/

12.5. Música y efectos sonoros

- http://www.playonloop.com/2015-music-loops/forgotten-woods/ (MainTheme)
- https://www.assetstore.unity3d.com/en/#!/content/22024 (ZonesTheme)
- http://www.playonloop.com/2014-music-loops/waving-grass/ (MenuTheme)
- https://freemusicarchive.org/music/Rolemusic/~/03_rolemusic_-_enthalpy (CreditsTheme)

- https://www.assetstore.unity3d.com/en/?gclid=CPX4iK6b3cYCFeTMtAodNy8FRw#!/content/22161 (Clicks and EnemyAlert sounds)
- https://www.assetstore.unity3d.com/en/?gclid=CPX4iK6b3cYCFeTMtAodNy8FRw#!/content/32831 (Win, Lose and PickUp sounds)
- https://www.assetstore.unity3d.com/en/?gclid=CPX4iK6b3cYCFeTMtAodNy8FRw#!/content/2924 (Steps sounds)
- http://www.audiomicro.com/glass-breaking-glass-break-glass-breaking-free-sound-effects-45075 (BrokenCrystal sound)
- http://www.freesfx.co.uk/soundeffects/stone-rock/ (BreakWarehouseWall sound)
- http://www.freesfx.co.uk/soundeffects/stone-rock/ (BreakZooWall sound)

12.6. Normativa y legislación:

- http://delitosinformaticos.com/legislacion/espana.shtml
- http://noticias.juridicas.com/base_datos/Penal/lo10-1995.l2t10.html
- http://www.boe.es/diario_boe/txt.php?id=BOE-A-2015-3439
- http://www.boe.es/diario_boe/txt.php?id=BOE-A-2015-3440
- http://gitsinformatica.com/legislacion.html
- http://www.boe.es/boe/dias/2010/06/23/pdfs/BOE-A-2010-9953.pdf
- https://unity3d.com/es/legal
- https://unity3d.com/es/legal/eula
- https://www.blender.org/about/license/
- http://www.gnu.org/copyleft/gpl.html
- http://www.coe.int/t/dghl/cooperation/economiccrime/Source/Cybercrime/TCY/ETS_185_spanish.PDF
- http://www.boe.es/diario_boe/txt.php?id=BOE-A-2010-14221
- http://www.unirioja.es/dptos/dd/redur/numero8/diaz.pdf
- http://apw.cancilleria.gov.co/tratados/AdjuntosTratados/3012f_CE-2001%20CIBER.PDF
- http://www.boe.es/boe/dias/2010/09/17/pdfs/BOE-A-2010-14221.pdf
- http://148.204.211.134/polilibros../z_basura/HTML/UNIDAD%201/CONTENIDO/Cont%20Unidad%201_1%20EU.htm
- http://www.inf.utfsm.cl/~lhevia/asignaturas/infoysoc/topicos/Etica/8_legislacion_acerca_uso_internet.pdf
- http://www.segu-info.com.ar/delitos/estadosunidos.htm
- http://www.uncitral.org/pdf/spanish/texts/electcom/ml-elecsig-s.pdf
- http://www.segu-info.com.ar/delitos/alemania.htm
- http://www.poderjudicialmichoacan.gob.mx/tribunalm/biblioteca/almadelia/Cap4.htm
- https://books.google.es/books?id=H7dxAgAAQBAJ&pg=PA66&lpg=PA66&dq=Computer+Misuse+Act+(Ley+de+Abusos+Inform%C3%A1ticos)&source=bl&ots=yUpVzw2i0F&sig=5Ga_MDtxDmXYuXqnYuieRsN3MDE&hl=es&sa=X&ei=pdSSVbilCcfpUuvigMAF&ved=0CC4Q6AEwAg#v=onepage&q=Computer%20Misuse%20Act%20(Ley%20de%20Abusos%20Inform%C3%A1ticos)&f=false
- http://www.segu-info.com.ar/delitos/holanda.htm

- http://www.eumed.net/rev/cccss/04/rbar2.htm
- http://www.segu-info.com.ar/delitos/francia.htm
- http://www.eumed.net/rev/cccss/14/ecra.html
- http://www.informaticaforense.com.co/index.php/francia/77--ley-relativa-al-fraude-informatico
- http://www.leychile.cl/Navegar?idNorma=30590
- http://delitosinformaticos.com/legislacion/chile.shtml
- http://www.segu-info.com.ar/delitos/chile.htm

12.6.1. Licencias

- https://unity3d.com/es/legal/eula (Unity 5)
- http://www.gnu.org/licenses/gpl-3.0.en.html (GNU General Public License)
- http://unity3d.com/legal/as_provider (Unity Asset Store)
- http://www.freesfx.co.uk/info/eula/ (Free Sound Effects)
- http://www.audiomicro.com/legal-docs/sound-effects-license (Sound Effects)