Videojuego de Match 3

Carlos Garcia Mañero
Grado de Multimedia
Trabajo de fin de Grado - Videojuegos

Consultor: Jordi Duch Gavaldà
Consultor: Helio Tejedor Navarro
Profesor responsable de la asignatura: Joan Arnedo Moreno

Semestre 1 / 2017 – 2018

© Carlos Garcia Mañero

Reservados todos los derechos. Está prohibido la reproducción total o parcial de esta obra por cualquier medio o procedimiento, comprendidos la impresión, la reprografía, el microfilm, el tratamiento informático o cualquier otro sistema, así como la distribución de ejemplares mediante alquiler y préstamo, sin la autorización escrita del autor o de los límites que autorice la Ley de Propiedad Intelectual.

FITXA DEL TREBALL FINAL

Título del trabajo:	Videojuego de Match 3
Nombre del autor:	Carlos Garcia Mañero
Nombre del consultor:	Jordi Duch Gavaldà
Nombre del consultor:	Helio Tejedor Navarro
Nombre del PRA:	Joan Arnedo Moreno
Fecha de entrega (mm/aaaa):	01/2018
Titulación o programa:	Grado de Multimedia
Área del Trabajo Final:	Trabajo de fin de Grado - Videojuegos
Idioma del trabajo:	Castellano
Palabras clave:	Videojuego, Match y Unity

Resumen del Trabajo (máximo 250 palabras): Con la finalidad, contexto de aplicación, metodología, resultados y conclusiones del trabajo

Mi proyecto tiene como finalidad la creación de un videojuego de match 3, un juego de tipo puzle en el que hay que juntar tres o más piezas iguales para hacer jugadas, obtener puntos y así poder superar los niveles del juego.

Luego, en cuanto al contexto de aplicación que tiene, éste es el de entretener a unos posibles jugadores, ya que se ha realizado un videojuego con un fin lúdico que busca llamar la atención del jugador y mantenerlo enganchado.

Después, en cuanto a la metodología de trabajo para realizar este proyecto, me he planificado una serie de tareas mediante un diagrama de Gantt, que me han permitido ir avanzando de forma escalonada en el desarrollo de los diferentes aspectos del videojuego.

Como resultado, he obtenido, a la vez que he aprendido a hacerlo, un videojuego agradable visualmente mediante sus diseños y animaciones, que funciona correctamente y que dispone de diversos modos de juego con sus respectivos niveles dentro de cada uno de éstos.

Para acabar, como conclusión del trabajo, decir que el desarrollo de un videojuego, por pequeño que éste sea, es muy laborioso y en éste intervienen diferentes campos como el diseño gráfico, la animación y la programación para los que hay que aprender a utilizar herramientas diversas, como Adobe Illustrator, Unity y MonoDevelop, tal y como he hecho yo para realizar este videojuego.

Abstract (in English, 250 words or less):

The purpose of my project is the creation of a match 3 video game, a puzzle game where you have to join three or more equal pieces to make plays and get points to pass the levels of the game.

Then, regarding the application context it has, it's to entertain the possible players, giving that the video game it's been made with a playful aim that wants to attract attention of the player and keep him hooked.

After that, as to the work methodology to create this project, I plan a series of tasks through a Gantt diagram, that allowed me to go forward in a staggered manner in the development of the video game's different aspects.

As a result, I achieved, at the same time I learned how to do it, a visually pleasant video game through its designs and animations, that works correctly and that has different game modes with its respective levels inside each one of them.

To sum up, as this works conclusion, I have to say that the development of a video game, no matter how small, is very laborious, and a lot of different fields such as graphic design, animation and programming intervene in it and you have to learn how to use different tools, like Adobe Illustrator, Unity and MonoDevelop, just like I have done to create this video game.

Índice

Introducción 2
Conceptualización 3
Planificación de los objetivos 4
Estructura del videojuego 5
Modos de juego 8
Diseño gráfico 11
Animación 14
Tipografía 16
Sonido 17
Programación 18
Cuadricula 19
PiezaJuego 39
PiezaMovil 42
PiezaColor 44
PiezaBorrable 46
PiezaBorrarLinea 48
PiezaBorrarColor 49
Nivel 50
NivelTipoMovimiento 53
NivelTipoTiempo 55
NivelTipoVirus 57
HUD 59
JuegoAcabado 62
SeleccionEscena 64
Diseño de niveles 66
Testeo de niveles 73
Conclusión 80
Fuentes 81

Introducción

En este proyecto se lleva a cabo la creación de un videojuego. Este videojuego es un juego de puzles en el que en su modo clásico, ya que tiene dos modos de juego más que ya se explicaran más adelante, por niveles, de más fácil a mas difícil, se nos presentan unas cuadrículas con piezas de diferentes tipos en las que hay que cambiar de posición piezas adyacentes, para juntar tres piezas iguales o más formando líneas, hasta lograr la victoria del nivel consiguiendo "x" cantidad de puntos, logrando así una medalla de menor o mayor valor según la cantidad de puntos obtenidos.

En estos niveles se tendrá en cuenta que el jugador solo puede hacer "x" cantidad de movimientos, haciendo que si el jugador no llega a un mínimo de puntos al utilizar todos los movimientos pierda la partida y no pueda pasar al siguiente nivel, teniendo que repetir el nivel en el que se encuentra.

Este videojuego esta dentro del género de puzles o rompecabezas y dentro de éste en el subgénero llamado Match 3. Éste se basa en juntar tres o más piezas del mismo color o forma para hacer jugadas. Dentro del subgénero Match 3 dos de los videojuegos existentes más destacados y que son referentes de este subgénero, y por consiguiente del videojuego que creo en este proyecto, son Candy Crush y Bejeweled. Estos dos han despuntado tanto en navegadores, mediante plataformas como Facebook, como en dispositivos móviles, lo que hace que se produzcan referencias a ellos cuando se habla y se crea algo dentro de este subgénero.

En el videojuego que he creado el jugador interacciona con el juego mediante el puntero del ratón, pudiendo abrir y cerrar el videojuego, desplazarse por las diferentes interfaces de inicio, selección de nivel y nivel de juego y pudiendo interactuar con las piezas, que están dentro de la cuadrículas que forman cada nivel, desplazándolas, intercambiando las posiciones de éstas. El juego a su vez interacciona con el jugador mostrándole información visual sobre la posición de las piezas, de los movimientos que le quedan al jugador, de la puntuación que lleva en el nivel y de las medallas que ha logrado.

Por último, comentar que la plataforma de destino del videojuego de entre todas las que me deja elegir Unity, el motor de videojuegos multiplataforma que he utilizado para hacer el videojuego, será PC, Mac y Linux.

Conceptualización

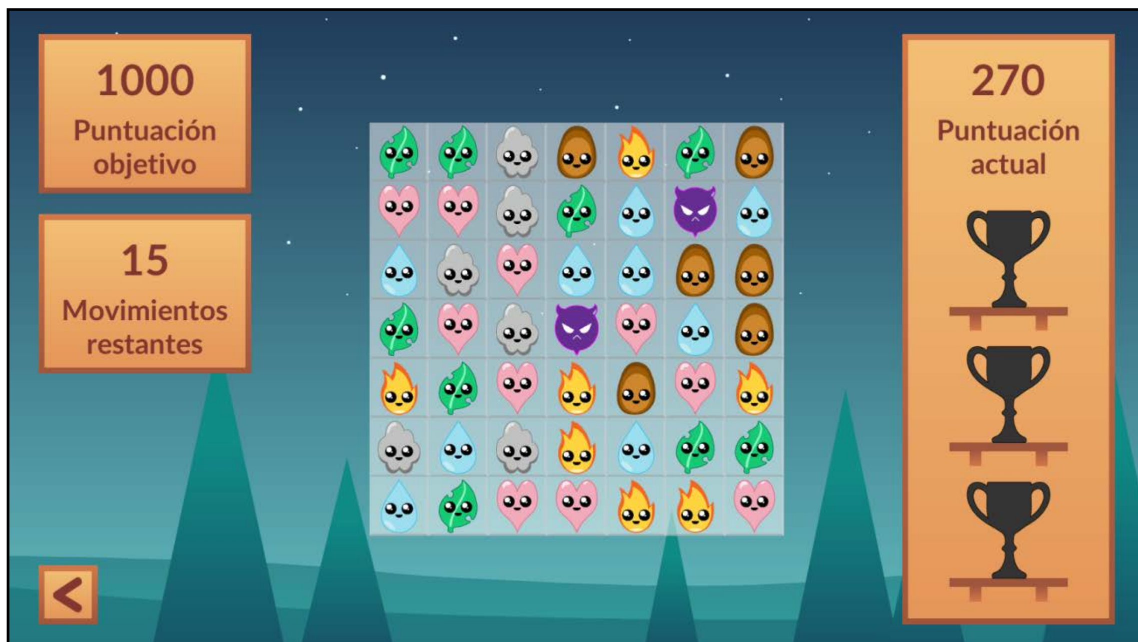
En el caso de mi videojuego no hay una historia o trama definida, en gran parte, porque debido al subgénero al que pertenecerá, dentro del género de puzles, no lo he encontrado necesario para motivar a los jugadores a querer pasar niveles. Ahora bien, si que tiene una ambientación y ésta está basada en diseños 2D de elementos: fuego, agua, viento, tierra, etc.

En el videojuego los personajes serán las piezas de la cuadrícula de cada nivel, siendo estas piezas versiones animadas de diferentes elementos, formándose así unos seres elementales, los cuales están dotados de vida. Por ejemplo: las piezas del elemento fuego son llamas, las piezas del elemento agua son gotas, las piezas del elemento tierra son piedras, etc., y todos con ojos y boca ganando así expresividad y vida.

Por tanto, los actores del juego son estos seres elementales que funcionan como los diferentes tipos de piezas de la cuadrícula de cada nivel, con lo cual, las interacciones que hay entre éstos son la de intercambiar posiciones, cada uno de ellos con sus adyacentes que estén situados en la izquierda, derecha, arriba y abajo, y la de formar líneas de 3 o más piezas iguales para lograr puntos.

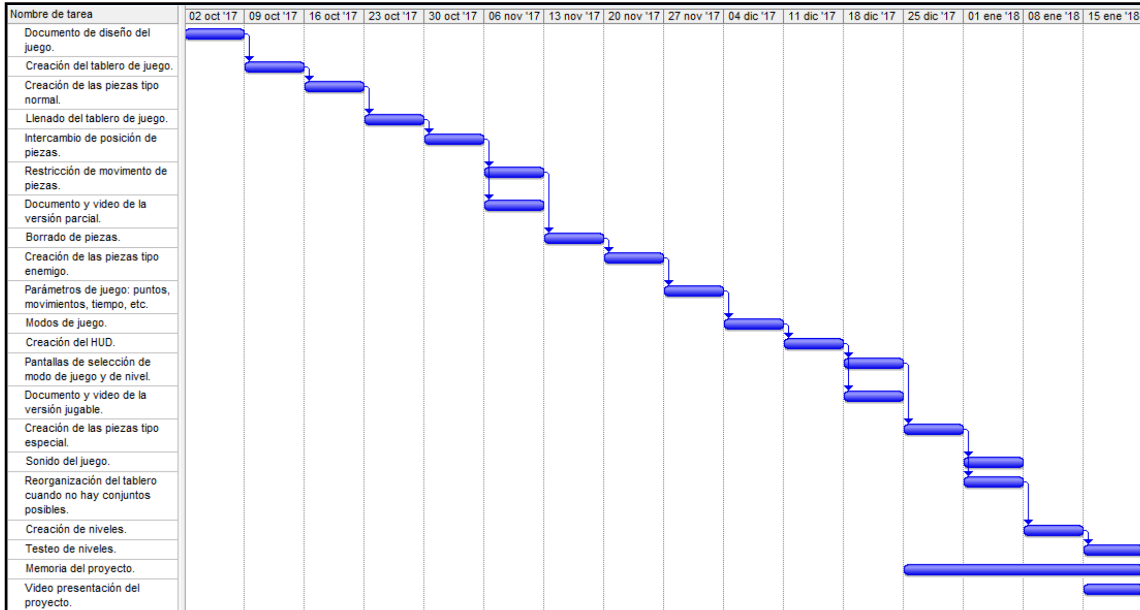
Luego respecto al jugador, ha éste se le plantean dos objetivos. Primero, el de lograr superar todos los niveles de los que dispone el juego y, después, el de lograr obtener la medalla de mayor valor, es decir la de oro, dentro de cada nivel, obteniendo una alta puntuación.

Por último, para que quede más clara la idea de lo que se ha creado, muestro una captura de pantalla de un nivel del juego.



Planificación de los objetivos

Respecto a la planificación de los objetivos del proyecto, me he organizado una serie de tareas mediante un diagrama de Gantt, que me han permitido ir avanzando de forma escalonada en el desarrollo de los diferentes aspectos del videojuego.



Los recursos utilizados para realizar estas tareas que se muestran en el diagrama de Gantt son:

- Unity y MonoDevelop, para la programación, la animación y la interacción de elementos.
- Adobe Illustrator, para la creación de los elementos gráficos.
- Adobe After Effects, para los videos y la presentación.
- Microsoft Office, para la documentación.

Una vez vistos los recursos elegidos para realizar las tareas, comentar que he escogido trabajar con Unity, ya que éste es el motor de videojuegos multiplataforma líder en el mundo y es gratis, mediante su versión Unity Personal, para principiantes, estudiantes y aficionados. Unity soporta el desarrollo 2D con funciones y funcionalidades específicas para llevar a término proyectos como el mío, y además, cuenta con una gran comunidad que le da soporte y con un gran catálogo de contenido gratuito.

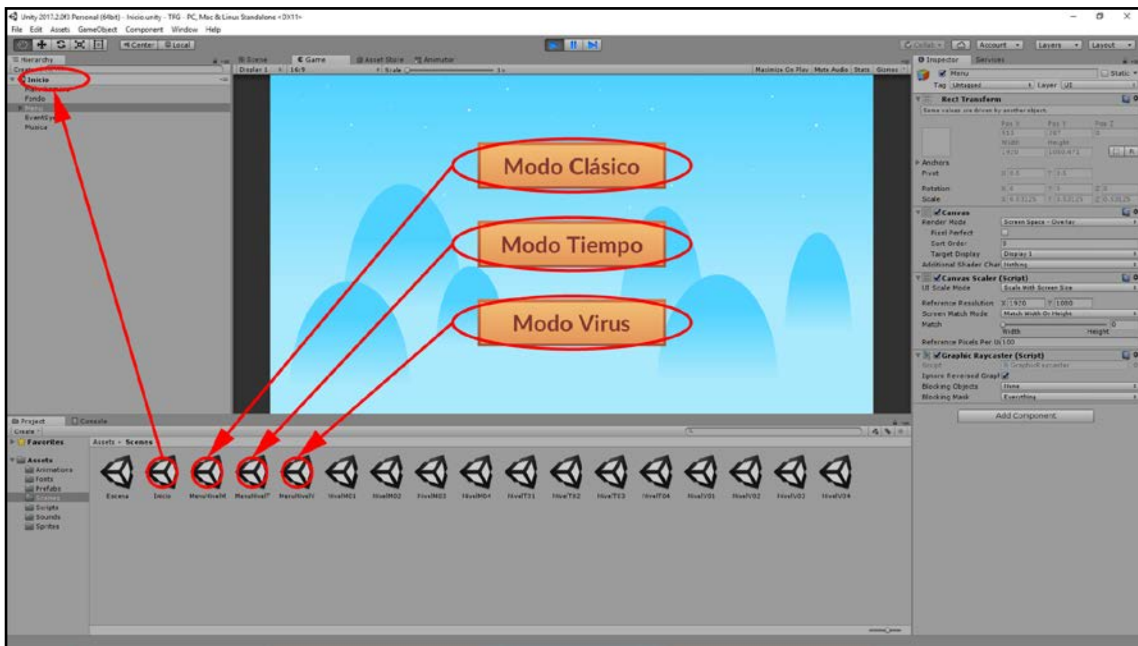
Por último, también comentar que he utilizado Adobe Illustrator para toda la creación de las imágenes que utilizo en el videojuego, ya que es una aplicación de edición de gráficos vectoriales que me permite diseñar unos dibujos que se pueden adaptar a todas las resoluciones.

Estructura del videojuego

En cuanto a la estructura del videojuego esta está formada por una serie de escenas que compondrán una pantalla de inicio, las diferentes pantallas de selección de nivel de cada uno de los modos de juego y los distintos niveles de juego.

A continuación, explico todo esto más en detalle, mostrando capturas de las diferentes escenas que forman la estructura del videojuego y las relaciones que hay entre éstas.

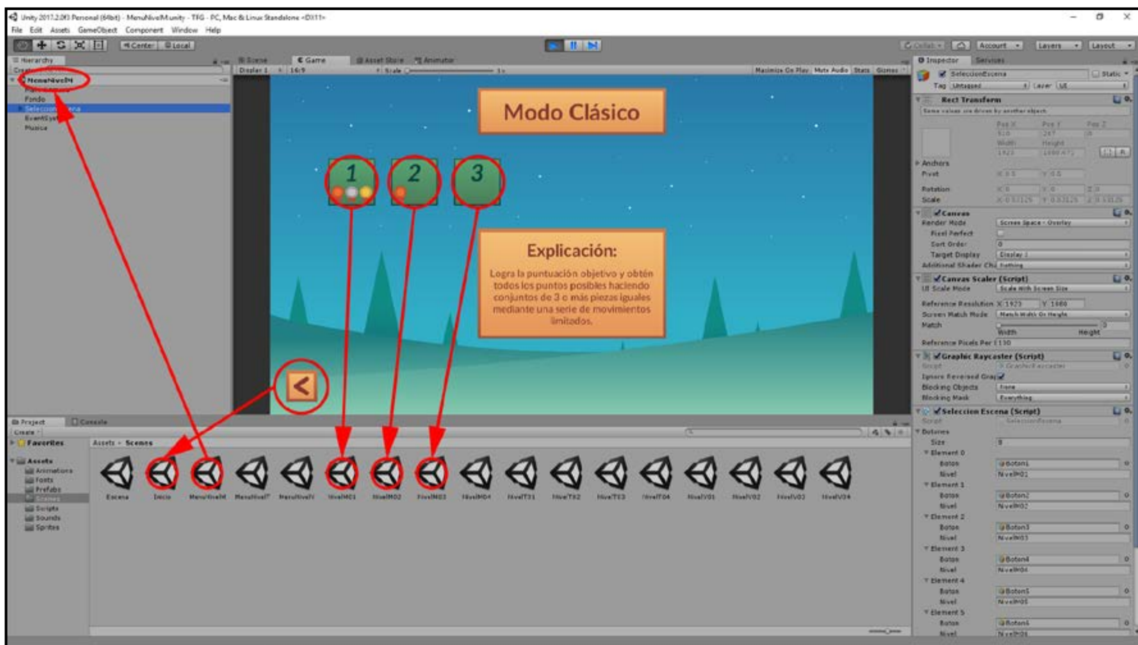
En comenzar el juego empezaremos por la pantalla de inicio, en esta tendremos un menú formado por tres botones que nos permitirán acceder a cada una de las pantallas de selección de nivel de cada uno de los modos de juego.



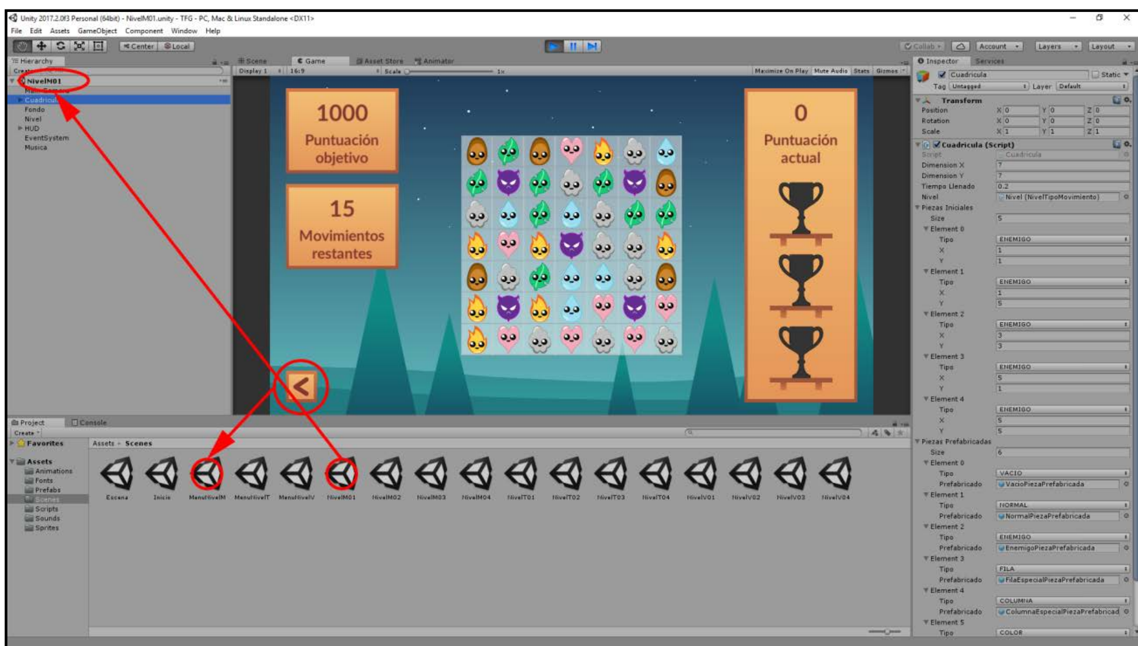
Después, una vez accedemos a una de las pantallas de selección de nivel, a continuación mostrare la del modo clásico, en ésta tendremos en la parte superior el titulo del modo de juego, en la parte inferior un texto explicativo con una breve descripción que explica en qué consiste el modo de juego, en la esquina inferior izquierda un botón para poder volver a la pantalla de inicio y en el centro nos encontraremos con los botones de los niveles del juego.

Estos botones estarán ocultos a excepción del botón del primer nivel de juego y conforme se vayan superando los niveles irán apareciendo, es decir, que en jugar el nivel uno y ganar el nivel, en el menú de selección de nivel se desbloqueara el botón del nivel dos y así sucesivamente.

A la vez que pasa esto, en el botón del nivel que se acaba de superar aparecerá una medalla de bronce y si se logran más puntos aparecerán también una medalla de plata y de oro si se obtienen los necesarios para ello.



Una vez accedemos a uno de los niveles de juego, podemos ver a derecha e izquierda de la pantalla los diferentes elementos informativos de puntuación objetivo, movimientos restantes, puntuación actual y las copas desbloqueadas, sobre los cuales entraremos a hablar más en detalla en el siguiente apartado de la memoria, ya que estos elementos según el modo de juego variaran.



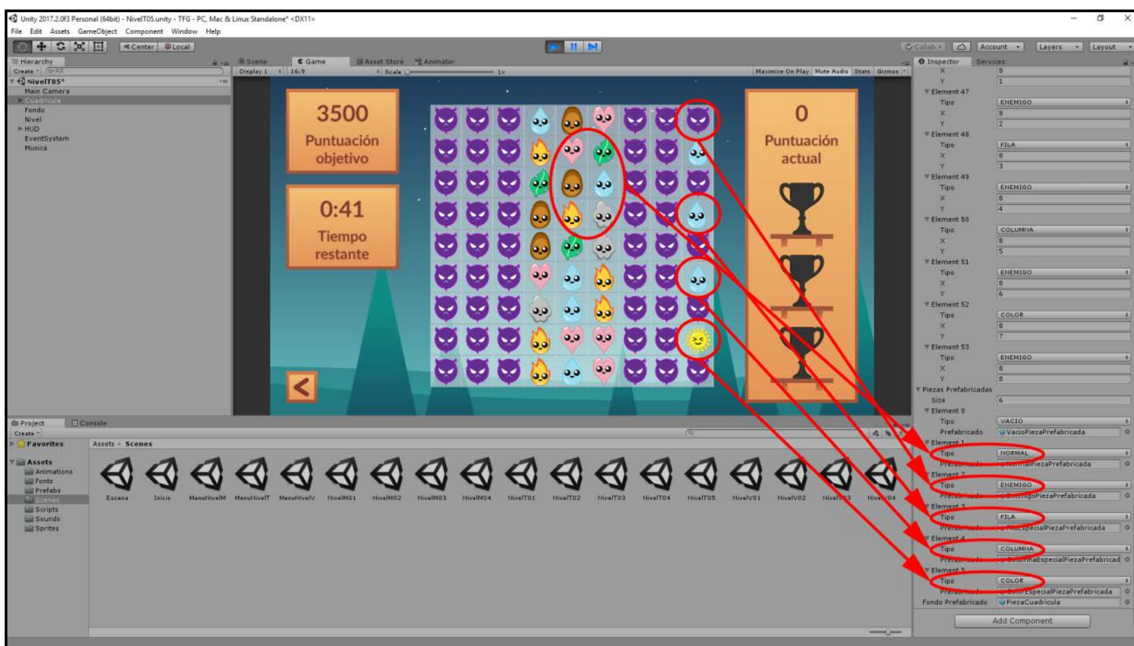
En la pantalla de nivel también estará presente el botón para volver atrás, en este caso a la pantalla de selección de nivel, y como no, en el centro, la cuadrícula de juego con las piezas que ahora pasare a explicar.

De piezas de juego tenemos las 6 normales que serán con las que se tendrán que hacer conjuntos de 3 o más piezas. Cada una de estas representara un elemento diferente debido a la temática que le quiero dar a la estética del juego.

Después, tenemos una pieza tipo enemigo, esta será una pieza inmóvil que estará como un obstáculo para molestar, éstas en producirse un conjunto de forma adyacente a ellas se destruyen y dan puntos.

Después, tenemos una pieza tipo color que es una pieza especial que se formara haciendo conjunto de 5 piezas o más y que en intercambiarse con una pieza borrara todas las piezas del color de esa pieza. En caso de que se intercambiara la posición con otra piza de tipo color se borrarían todas las piezas de la cuadrícula recibiéndose los pertinentes puntos de todas ellas.

Y, por último, tenemos las piezas de tipo columna y fila, que se formaran haciendo conjuntos de 4 piezas, formándose una u otra dependiendo de si se formo en un intercambio en horizontal o en vertical. Estas serán de uno de los 6 colores de las piezas normales, determinado al hacerse el conjunto con las piezas de ese color. Y esteran animadas, la de tipo columna, moviéndose de abajo arriba y de arriba abajo y, la de tipo fila, moviéndose de izquierda a derecha y de derecha a izquierda.



Para terminar, comentar que en acabar la partida nos saldrá un recuadro informándonos sobre si se ha ganado o perdido la partida que mostrara también la puntuación final obtenida.

Este recuadro también contendrá dos botones: El primero de éstos, el de repetir, nos permitirá volver a jugar el nivel de juego y el segundo, el de terminar, nos devolverá al menú de selección de nivel.

Modos de juego

El videojuego cuenta con tres modos de juego, siendo éstos el modo clásico, el modo tiempo y el modo virus. Estos tres modos de juego los explico a continuación, en detalle, mostrando también un nivel de juego funcional de cada uno de éstos.

En el modo clásico, dispondremos de una serie de movimientos, para hacer jugadas y lograr la cantidad de puntos objetivo, lo que permitirá poder superar el nivel. En este modo de juego, al igual que en los otros dos que a continuación explicare, según la puntuación que hagamos obtendremos diferentes copas, obteniéndose, en el caso de este modo, la de bronce con la puntuación objetivo, pero teniendo que hacer más puntos para lograr la de plata y la de oro.

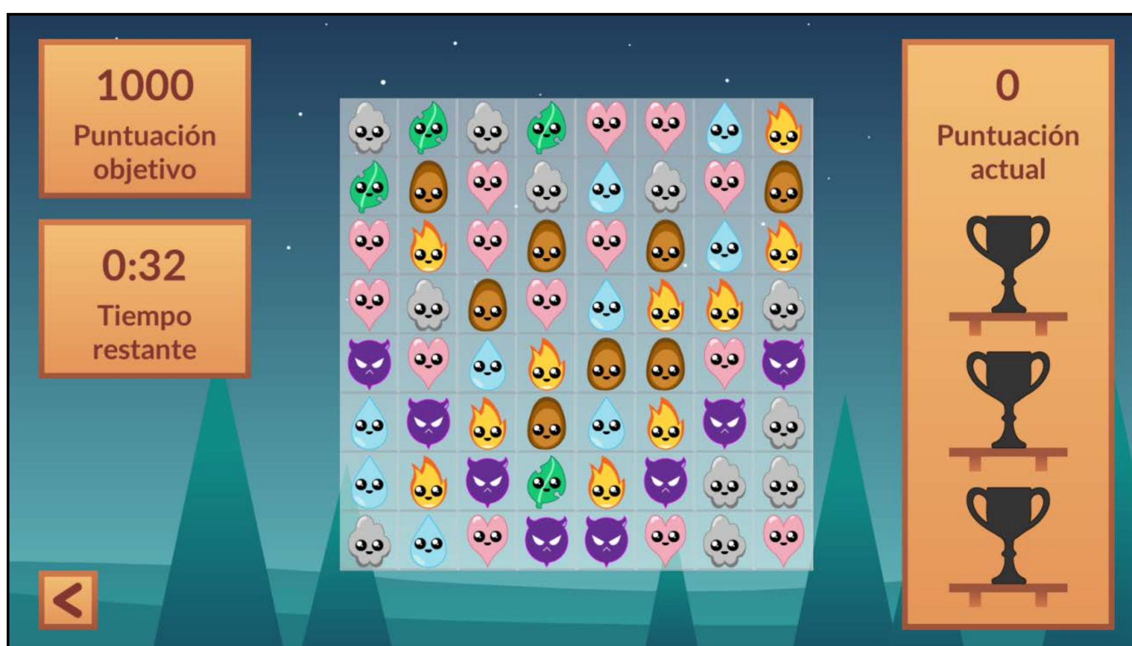
En la interfaz de un nivel del modo clásico, que se puede observar en la imagen que hay a continuación de este párrafo, podemos ver en la izquierda la puntuación objetivo a lograr para superar el nivel y debajo el número de movimientos que quedan por usar, en el centro podemos ver la cuadrícula de juego con las piezas de diferentes tipos y a la derecha la puntuación actual que lleva el jugador y debajo las copas logradas por el momento en el nivel.

En lograr la puntuación objetivo se visualizara la copa de bronce y en ir obteniendo más puntos aparecerán la copa de plata y de oro. Cada pieza normal o especial que se borre da al jugador 10 puntos y cada pieza de tipo enemigo que se borre da al jugador 50 puntos.

En superarse un nivel, en el menú de selección de nivel del modo de juego clásico se desbloqueara el botón para poder acceder al siguiente nivel y aparecerá en el botón del nivel que se acaba de superar una medalla de bronce. También aparecerán una medalla de plata y oro si se obtienen los puntos necesarios para ello.

En el modo tiempo, dispondremos de un tiempo limitado, para hacer jugadas y lograr la cantidad de puntos objetivo, lo que permitirá poder superar el nivel. En este modo de juego, el sistema de copas funcionara igual que en el modo clásico, obteniéndose la de bronce con la puntuación objetivo, pero teniendo que hacer más puntos para lograr la de plata y la de oro.

En la interfaz de un nivel del modo tiempo, que se puede observar en la imagen que hay a continuación de este párrafo, podemos ver en la izquierda la puntuación objetivo a lograr para superar el nivel y debajo el tiempo que queda para acabar, en el centro podemos ver la cuadrícula de juego con las piezas de diferentes tipos y a la derecha la puntuación actual que lleva el jugador y debajo las copas logradas por el momento en el nivel.



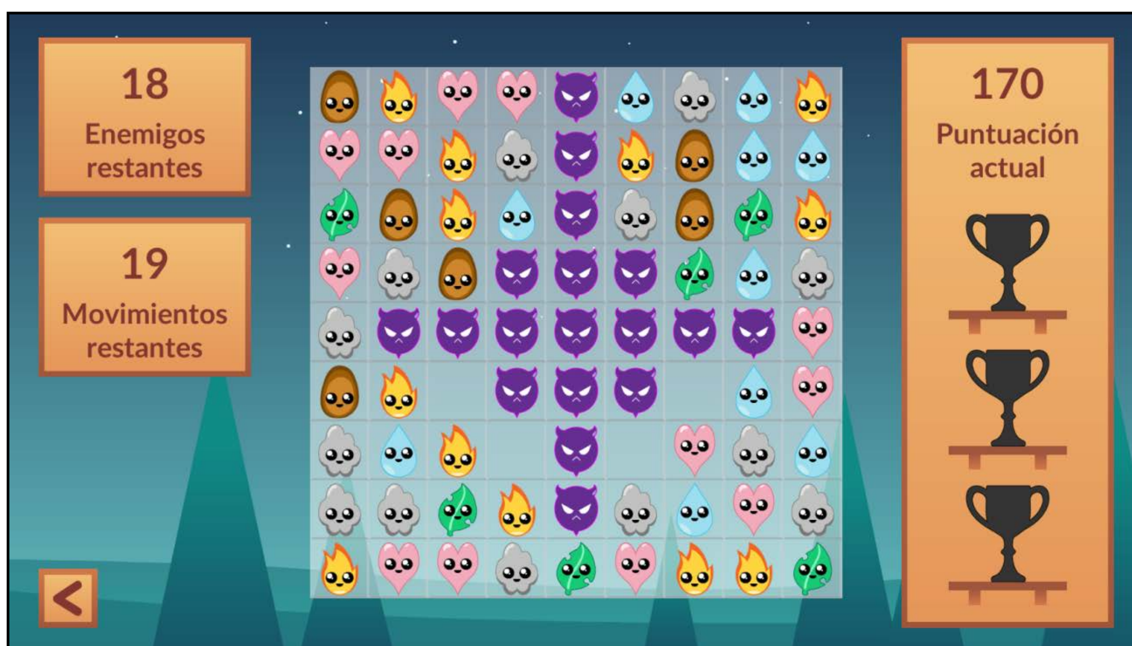
En lograr la puntuación objetivo se visualizara la copa de bronce y en ir obteniendo más puntos aparecerán la copa de plata y de oro. Cada pieza normal o especial que se borre da al jugador 10 puntos y cada pieza de tipo enemigo que se borre da al jugador 50 puntos.

En superarse un nivel, en el menú de selección de nivel del modo de juego tiempo se desbloqueara el botón para poder acceder al siguiente nivel y aparecerá en el botón del nivel que se acaba de superar una medalla de bronce. También aparecerán una medalla de plata y oro si se obtienen los puntos necesarios para ello.

En el modo virus, dispondremos de una serie de movimientos, para hacer jugadas de forma adyacente a unas piezas de juego de tipo virus y así eliminar estas piezas. En este modo de juego para poder superar el nivel habrá que eliminar todas las piezas de tipo virus antes de que se acaben los movimientos permitidos.

En este modo de juego también obtendremos copas según la puntuación que hagamos, por ello, en eliminar a todos los enemigos y superar el nivel siempre obtendremos la copa de bronce ya que la puntuación para obtener ésta siempre será la cantidad de puntos que se obtiene de eliminar todas la piezas de tipo enemigo que hay en la cuadrícula del nivel. Ahora bien, para lograr la de plata y la de oro ya habrá que hacer muchos más puntos, por ello, si nos sobran movimientos, en acabar con los enemigos, por cada movimiento que sobra se obtendrá una gran cantidad de puntos. Comentar también que si no se elimina a todos los enemigos en acabar el juego la puntuación pasara a ser 0, independientemente de los puntos realizados durante la partida, impidiéndose así el poder obtener copas sin superar el objetivo del nivel.

En la interfaz de un nivel del modo virus, que se puede observar en la imagen que hay a continuación de este párrafo, podemos ver en la izquierda los enemigos restantes a eliminar para superar el nivel y debajo el número de movimientos que quedan por usar, en el centro podemos ver la cuadrícula de juego con las piezas de diferentes tipos y a la derecha la puntuación actual que lleva el jugador y debajo las copas logradas por el momento en el nivel.



Según se vayan logrando puntos se visualizaran las copas de bronce, plata y oro. Cada pieza normal o especial que se borre da al jugador 10 puntos, cada pieza de tipo enemigo que se borre da al jugador 50 puntos y cada movimiento que sobre en finalizarse el juego da al jugador 200 puntos.

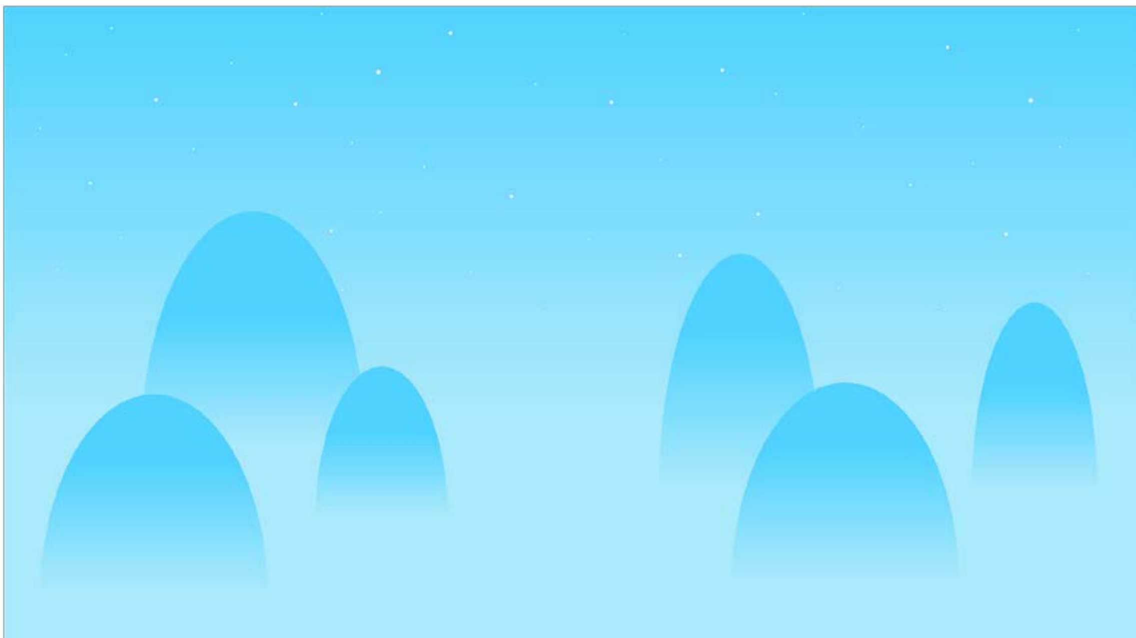
En superarse un nivel, en el menú de selección de nivel del modo de juego virus se desbloqueara el botón para poder acceder al siguiente nivel y aparecerá en el botón del nivel que se acaba de superar una medalla de bronce. También aparecerán una medalla de plata y oro si se obtienen los puntos necesarios para ello.

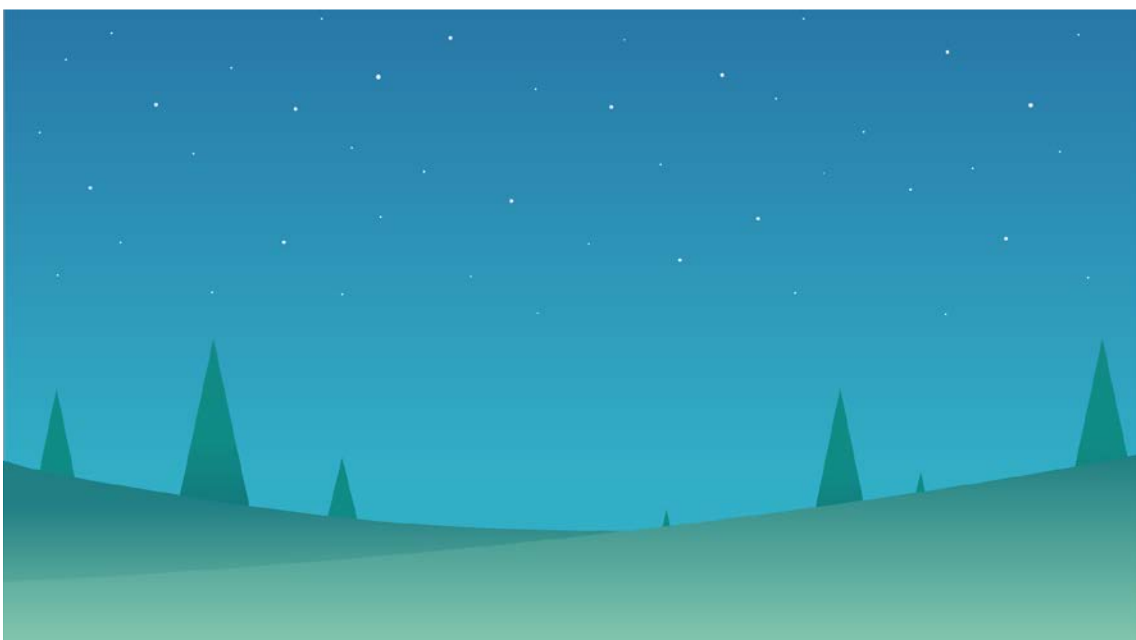
Diseño gráfico

Para realizar el videojuego de Match 3, que he desarrollando en 2D, he tenido que crear una serie de elementos gráficos que me hacen falta para los diferentes elementos que están presentes en las escenas que forman el videojuego.

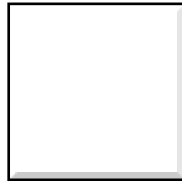
La realización de estos elementos gráficos se ha llevado a cabo en Adobe Illustrator, con el uso de múltiples capas y mediante la creación de líneas y formas geométricas las cuales he ido agrupando, restando, rotando y deformando entre otras cosas.

Para comenzar he creado unos fondos para las escenas de inicio, de selección de nivel y de nivel, las cuales muestro en ese mismo orden a continuación. Éstas las he realizado con unas dimensiones de 1920 x 1080 píxeles.



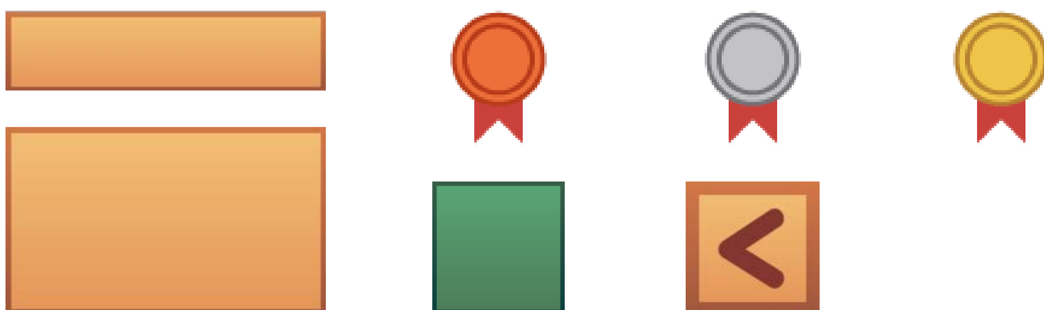


Después, he creado un fondo para las celdas de la cuadricula donde van a ir las piezas, el cual, permite ver mejor la cuadricula y resalta las piezas que hay en cada posición de la cuadricula. Este elemento, aunque aquí no se aprecia, tiene la opacidad al cincuenta por ciento para que se pueda ver a través de él el fondo de la escena.



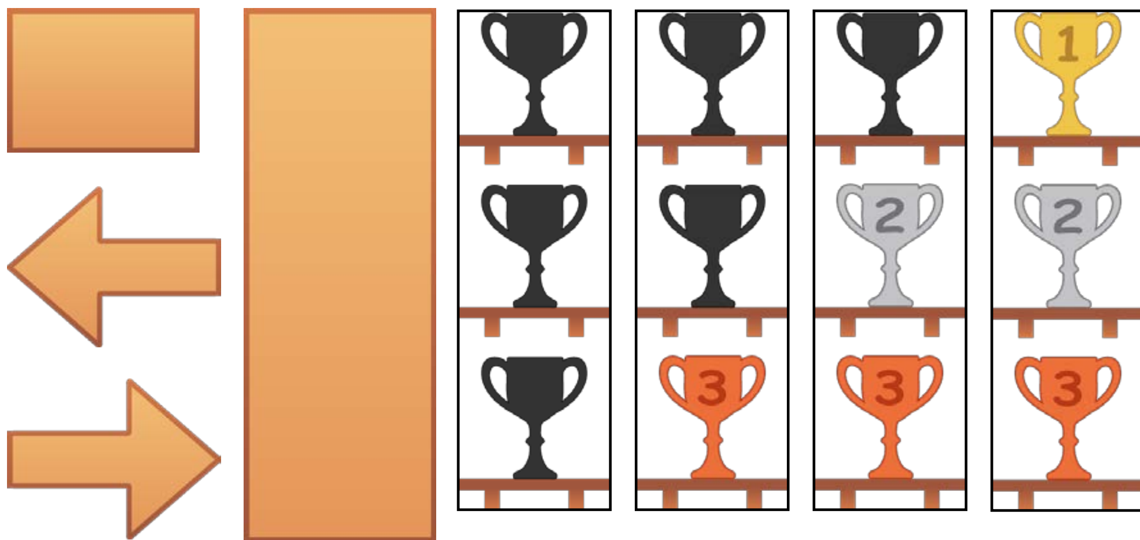
Luego, he creado los seis diseños para cada uno de los modelos de pieza normal que tiene el juego. Cada uno de estos representa un elemento diferente debido a la temática de los elementos que le quiero dar a la estética del juego.

Después, he creado los elementos gráficos para el inicio y el menú de selección, donde se incluyen los botones, las medallas que salen en los botones de selección de nivel según el rango obtenido en dicho nivel y el botón para volver a atrás.

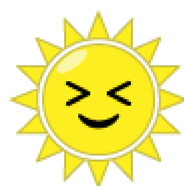


A continuación, se puede ver como he creado el diseño de la pieza tipo enemigo que del juego. Éste representa el elemento virus dentro de la temática del juego. Y he realizado una secuencia ya que utilizo ésta para la animación de borrado de este tipo de pieza en el juego

Luego también he creado los elementos gráficos para el HUD de dentro de los niveles del juego y los elementos gráficos para la interfaz de usuario que se muestra cuando se finaliza un nivel. En la izquierda los fondos para los textos y puntuaciones y en la derecha la secuencia de copas.

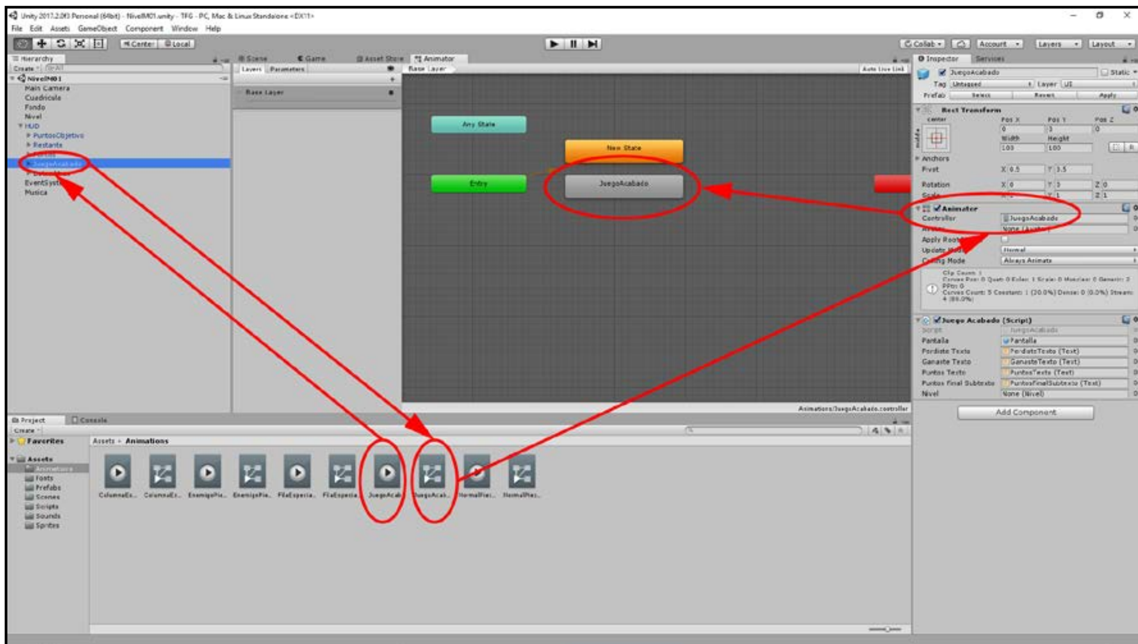


Por último, he creado el diseño para la pieza especial que se utiliza para borrar todas las piezas de un mismo color.



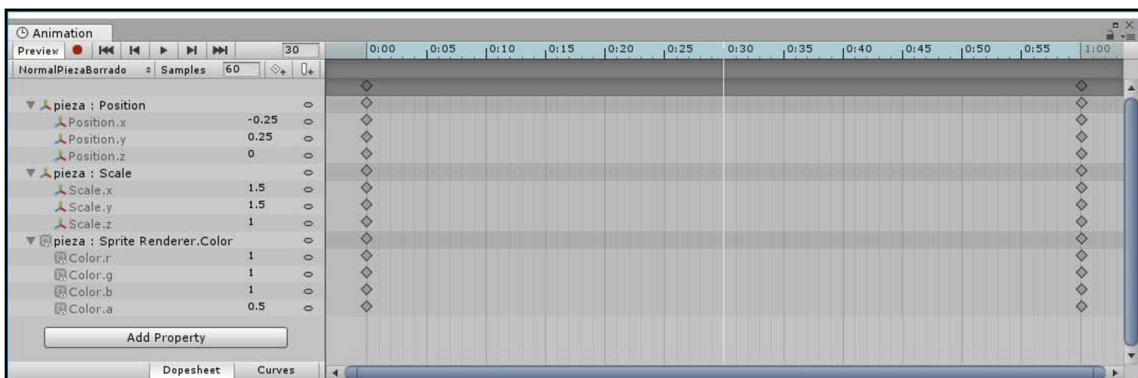
Animación

En el videojuego también se han realizado mediante Unity una serie de animaciones, las cuales, se han creado y añadido en el componente de la animación de los objetos de juego que se han querido animar, creándose así una relación entre el objeto del juego y la animación, que hace que la animación de ese objeto pase a estar controlada por esa animación.

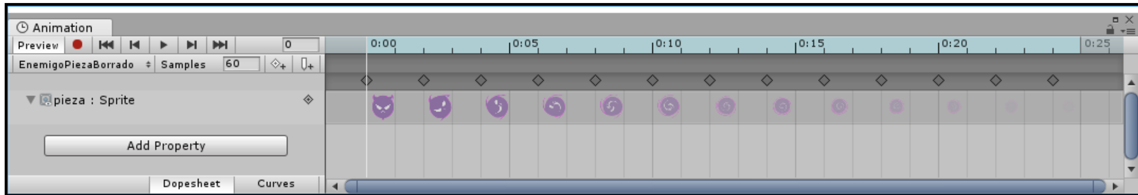


Estas animaciones creadas han sido dos para la eliminación de piezas de la cuadrícula de juego, una para la presentación de elementos una vez acabado un nivel de juego y dos para el movimiento de dos de las piezas especiales de las que dispone el juego. A su vez, cada una de éstas ha sido diferente de las demás en cuanto a propiedades modificadas.

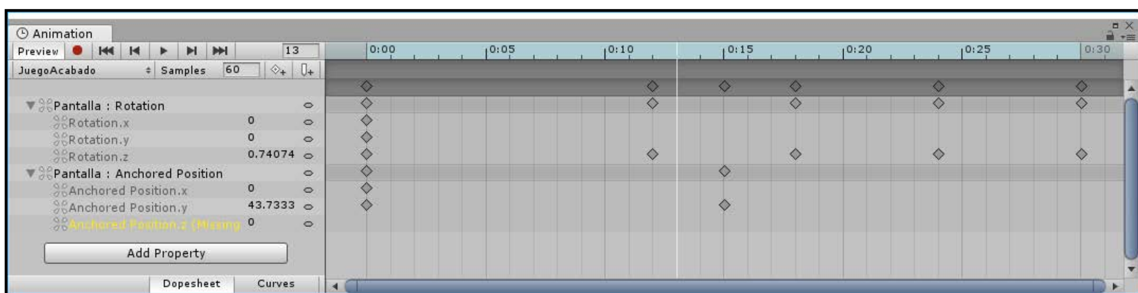
Primero, para la animación general de la eliminación de las piezas, en la línea de tiempo, he modificado ligeramente el posicionamiento de éstas, he aumentado su escalado y he reducido el valor del canal alfa del color hasta llegar a 0 para hacer que, en ser borradas, aumenten de tamaño de forma centrada a la vez que desaparecen.



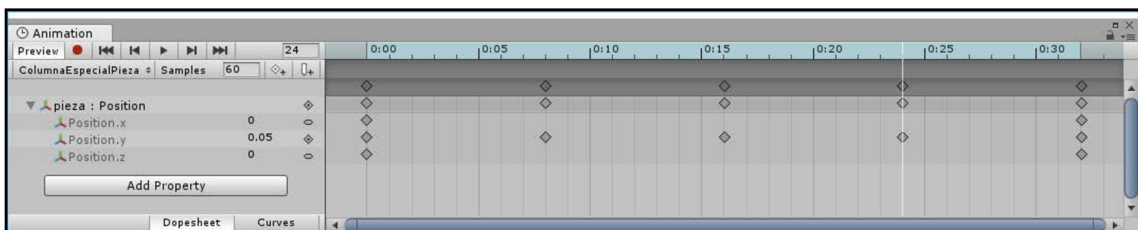
Después, para la animación de la eliminación de las piezas de tipo enemigo, en la línea de tiempo, he añadido diferentes sprites, los cuales forman una secuencia de la pieza, haciendo que se visualice como se va deformando en forma de espiral y desapareciendo, debido al uso del molinete en Illustrator al crear los sprites, a la vez que los he ido escalando y reduciendo la opacidad.



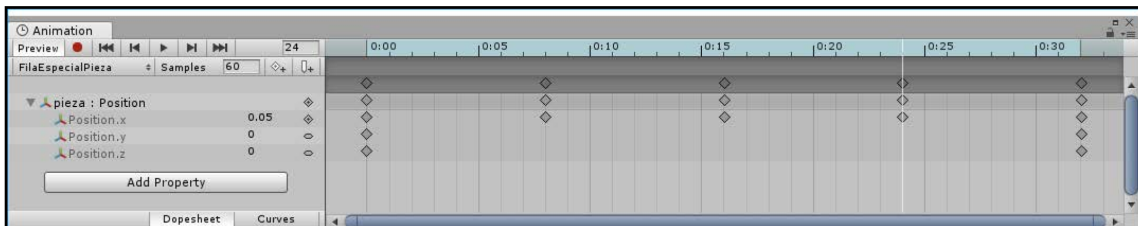
Luego, para la animación del recuadro de información que aparece en acabar un nivel, en la línea de tiempo, he añadido diferentes modificaciones en la rotación del eje “z” para hacer un efecto rebote al desplazar el ancla de la posición “y” de arriba de la pantalla al centro.



A continuación, para la animación de la pieza especial que borra columnas, en la línea de tiempo, he modificado la posición “y” de ésta, para hacer que la pieza se mueva de abajo a arriba y de arriba a abajo continuamente.



Por último, para la animación de la pieza especial que borra filas, en la línea de tiempo, he modificado la posición “x” de ésta, para hacer que la pieza se mueva de izquierda a derecha y de derecha a izquierda continuamente.

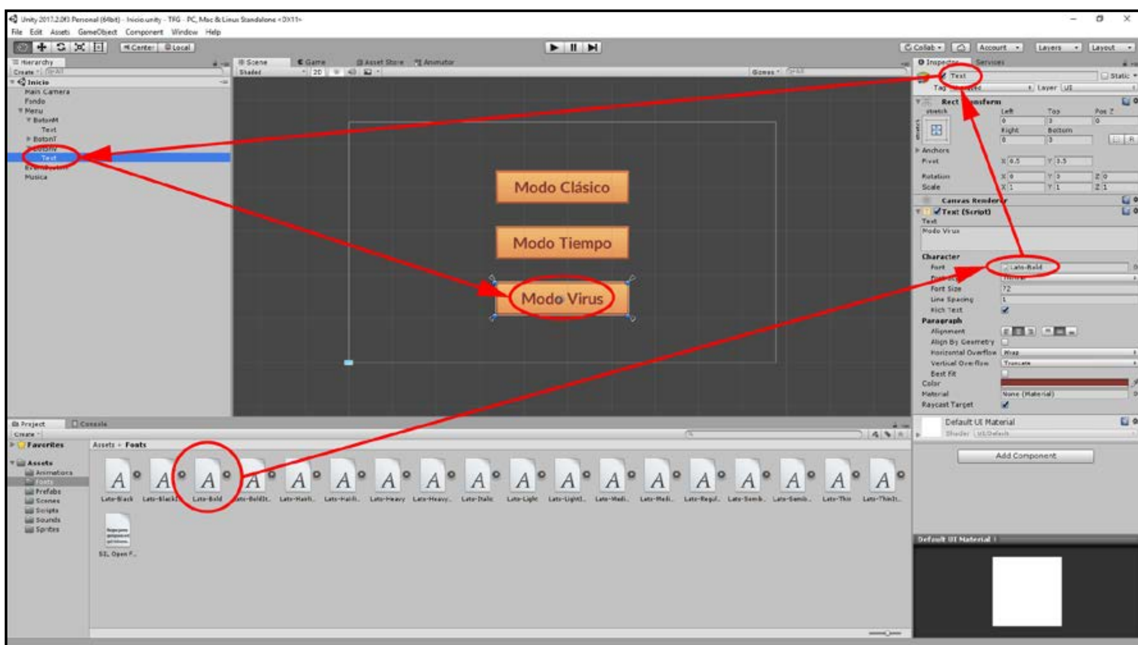


Tipografía

Para la tipografía del videojuego, he utilizado la familia tipográfica “LATO” que he descargado de la página web de Font Squirrel, ya que tiene licencia SIL Open Font License que me permite su uso.

He elegido la familia tipográfica de “LATO” porque es un tipo de letra bastante “transparente” cuando se utiliza en el texto de cuerpo, pero que muestra algunos rasgos originales cuando se usa en tamaños más grandes. Es un tipo de letra que transmite calidez y estabilidad, lo que acompaña de forma correcta al resto de elementos del videojuego.

Estas fuentes tipográficas de la familia “LATO”, las he utilizado para todos los textos del videojuego, poniéndolas como fuente de los caracteres en los scripts de Text según me ha ido conviniendo.

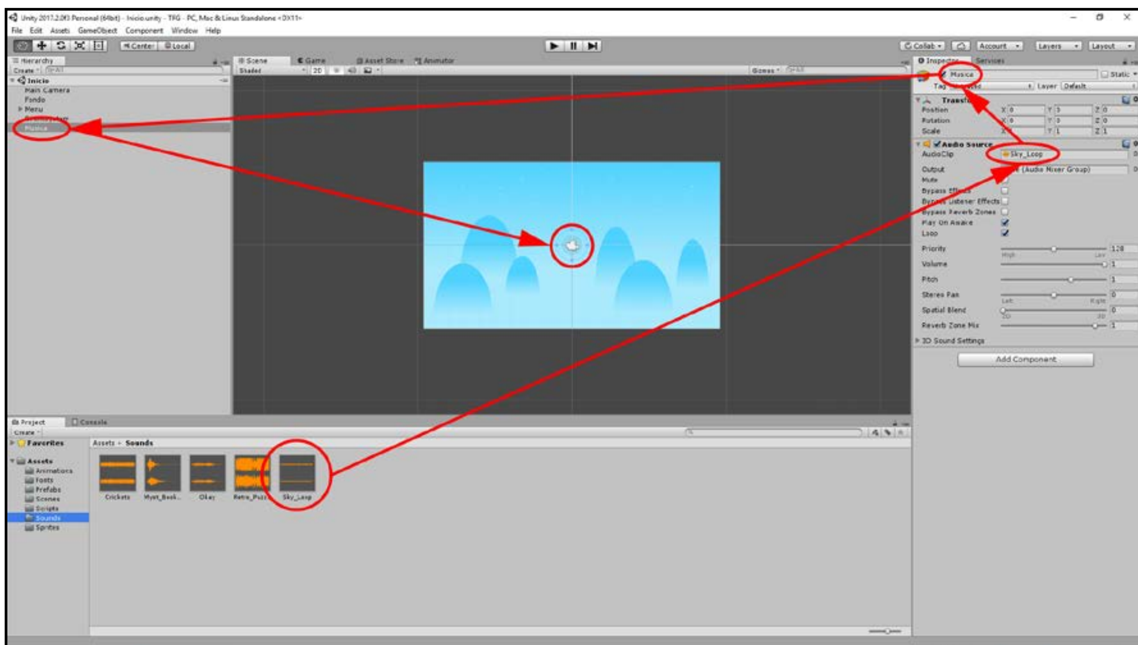


Sonido

Para el sonido del videojuego, he utilizado archivos de sonido que he descargado de la página web de Freesound, ya que tienen licencias Creative Commons que me permiten su uso y un gran repositorio para poder elegir.

Estos archivos de sonido, los he utilizado para la música de fondo del escenario de inicio, del de selección de nivel y del de nivel, para así marcar más la diferencia entre éstos, poniendo una música de ritmo más acelerado en los escenarios de nivel y más tranquila en los otros.

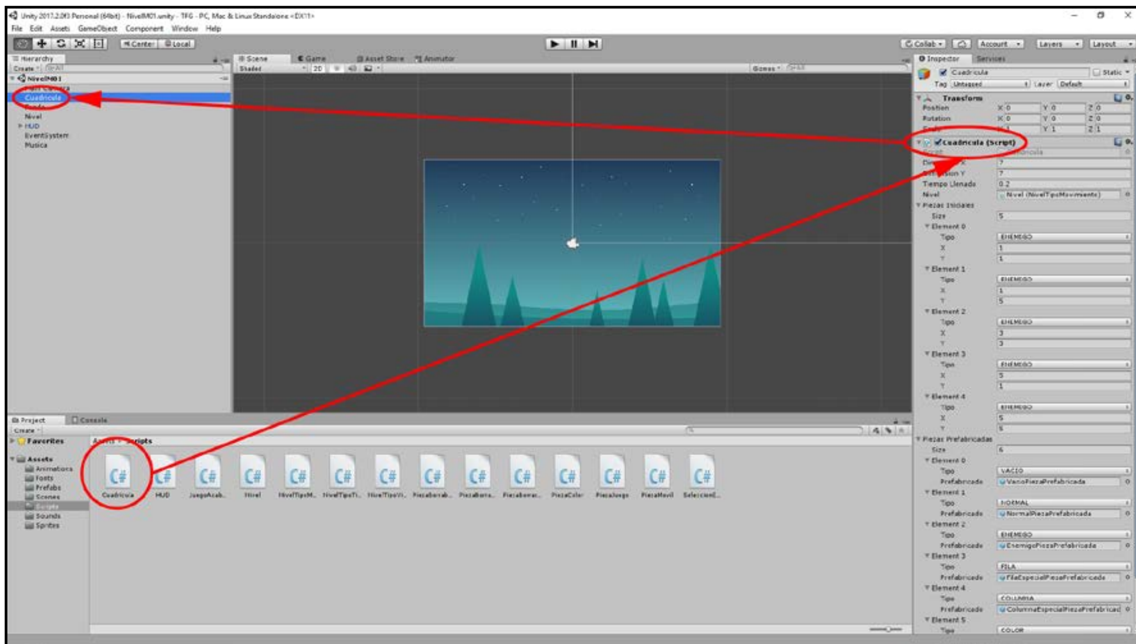
Estos archivos de sonido los he integrado en la escena mediante un objeto de juego al que he llamado música y el componente Audio Source, el cual a su vez me ha permitido hacer una serie de ajustes sobre éstos, como ajustar el volumen o hacer que se reproduzcan en bucle.



Por último, también he puesto efectos de sonido en la creación de las piezas especiales, añadiendo el componente Audio Source a estas piezas para que se reproduzca el sonido en generarse éstas.

Programación

Los objetos creados en el editor de Unity, desde la cuadrícula de los niveles de juego, pasando por el HUD y el nivel, hasta los elementos de selección de escena de los menús de selección de nivel y las piezas prefabricadas de los diferentes tipos de piezas son controlados desde scripts, que están presentes en estos objetos del juego como componentes de los mismos.



Estos scripts dotan a los objetos del juego de funcionalidades, a la vez que permiten ir modificando los parámetros que nos interese de éstos desde el inspector de Unity.

Por tanto, a continuación, voy a hablar de los scripts creados en el videojuego, abarcándolos todos, de uno en uno, explicando la utilización de cada clase, el uso de sus funciones y mostrando el código presente en cada script, el cual, está comentando línea a línea para mayor compresión de los detalles del funcionamiento del mismo.

Cuadricula

Esta clase es la primera y la más grande y se utiliza para gestionar y organizar todo lo que pasa en la cuadrícula del juego. Crea la cuadrícula de juego, crea piezas del juego, verifica si en intercambiar piezas se hacen conjuntos y permite intercambiarlas, borra piezas, llena la cuadrícula de piezas, cuenta las piezas de tipo enemigo que hay, indica cuando se termina la partida y permite reorganizar las piezas cuando no hay movimientos posibles.

Funciones:

- Awake: Se usa para crear lo antes posible la cuadrícula de piezas asignándole unas dimensiones, poniendo un fondo en cada posición, creando las piezas iníciales y luego poniendo en el resto de posiciones piezas vacías que se irán sustituyendo por las piezas normales en producirse el llenado de la cuadrícula.
- Llenado: Se usa para que se ejecute y se aprecie con el paso del tiempo el llenado de piezas de la cuadrícula.
- Se usa para que se ejecuten y se aprecien con el paso del tiempo los movimientos de las piezas.
- PasoLlenado: Se usa para mover las piezas hacia abajo en el llenado.
- ObtenerPosicion: Se usa para modificar la posición de las celdas de la cuadrícula para centrar la cuadrícula.
- GenerarNuevaPieza: Se usa para crear una nueva pieza del juego.
- EsAdyacente: Se usa para indicar si dos piezas son adyacentes.
- IntercambiarPiezas: Se usa para intercambiar la posición de dos piezas.
- PresionarPieza: Se usa para guardar la pieza presionada con el ratón.
- EntrarPieza: Se usa para guardar la pieza donde ha entrado el ratón.
- SoltarPieza: Se usa para hacer el intercambio de piezas en soltar el ratón si estas dos piezas son adyacentes.
- ObtenerConjunto: Se usa para saber si se puede hacer un conjunto con una pieza dada en las posiciones a las que ésta se puede mover.
- BorrarConjuntosValidos: Se usa para borrar todos los conjuntos que se forman, ya sea de inicio, por un intercambio de piezas o porque se han tenido que reorganizar las piezas de la cuadricula por no haber más movimientos posibles.
- BorrarPieza: Se usa para borrar una pieza.
- BorrarEnemigos: Se usa para borrar piezas de tipo enemigo.
- LimpiarFila: Se usa para mirar todas las piezas de la fila que se tiene que borrar.
- LimpiarColumna: Se usa para mirar todas las piezas de la columna que se tiene que borrar.
- LimpiarColor: Se usa para mirar todas las piezas del color que se tiene que borrar.
- ObtenerPiezasTipo: Se usa para contar el número de piezas de tipo enemigo que hay en la cuadrícula.
- JuegoAcabado: Se usa para indicar que la partida ha terminado.
- ReorganizarPiezas: Se usa para reorganizar las piezas cuando no hay movimientos posibles en la cuadrícula. Mira si alguna pieza en intercambiar su posición con otra hace alguna combinación, si no se puede, borra todas las piezas movibles de la cuadrícula y la vuelve a llenar con nuevas piezas.

A continuación se muestra el código de la clase, comentado línea a línea para mayor compresión de los detalles del funcionamiento del mismo.

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

public class Cuadricula : MonoBehaviour {

  // Parámetros para las dimensiones de la cuadricula. Permiten controlar el
tamaño de la cuadricula.
  public int dimensionX;
  public int dimensionY;

  // Parámetro para el tiempo entre llamadas para completar el paso de llenar
 la cuadricula. A menor tiempo mas rápido se llenara.
  public float tiempoLlenado;

  // Parámetro para los tipos de pieza. Contiene el valor CONTADOR en la enum
eración para saber cuántos tipos de piezas hay.
  public enum TipoPieza {
    VACIO,
    NORMAL,
    ENEMIGO,
    FILA,
    COLUMNA,
    COLOR,
    CONTADOR
  };

  // Parámetro para que Cuadricula pueda llamar a las funciones de Nivel.
  public Nivel nivel;

  // Hace que los parámetros del struct sean visualizables en el Inspector.
  [System.Serializable]
  // Estructura para generar piezas al comienzo del juego en posiciones espec
ificas.
  public struct PiezaPosicion {
    public TipoPieza tipo;
    public int x;
    public int y;
  };

  // Parámetro para una matriz de las estructuras de posiciones de piezas.
  public PiezaPosicion[] piezasIniciales;

  // Hace que los parámetros del struct sean visualizables en el Inspector.
  [System.Serializable]
  // Estructura con el tipo de pieza y el GameObject.
  public struct PiezaPrefabricada {
    public TipoPieza tipo;
    public GameObject prefabricado;
  };

  // Parámetro para una matriz de claves y valores. Las claves serán el tipo
de pieza y los valores serán el GameObject.
  public PiezaPrefabricada[] piezasPrefabricadas;

  // Parámetro para mostrar un fondo detrás de cada una de las piezas.
```

```
  public GameObject fondoPrefabricado;

  // Parámetro para asociar cada tipo de pieza con el prefabricado. Se asocia
 una clave que será el tipo de pieza con un valor que será el GameObject.
  // Los diccionarios no se pueden visualizar en el Inspector.
  private Dictionary <TipoPieza, GameObject> piezaPrefabricadaDic;

  // Parámetro para una matriz 2D de piezas del juego.
  private PiezaJuego[,] piezas;

  // Parámetro para almacenar la pieza en que se ha presionado el botón del r
atón.
  private PiezaJuego piezaPresionado;

  // Parámetro para almacenar la ultima pieza donde entro el ratón.
  private PiezaJuego piezaEntrado;

  // Parámetro para hacer que las piezas que se llenan en diagonal no proveng
an de la misma dirección siempre y se vaya intercambiando la dirección.
  private bool inverso = false;

  // Parámetro para saber si el juego ha terminado.
  private bool juegoAcabado = false;

  // Parámetro para saber si la cuadrícula se está llenando.
  private bool estaLlenando = false;

  // Propiedad pública para acceder a la variable.
  public bool EstaLlenando {
    get {
      return estaLlenando;
    }
  }

  void Awake() {
    // Crea un nuevo diccionario.
    piezaPrefabricadaDic = new Dictionary <TipoPieza, GameObject> ();
    // Recorre todos los prefabricados en la matriz de prefabricados.
    for (int i = 0; i < piezasPrefabricadas.Length; i++) {
      // Comprueba si el diccionario ya contiene la clave.
      if (!piezaPrefabricadaDic.ContainsKey (piezasPrefabricadas [i].tipo)) {
        // Si no contiene la clave, agrega al diccionario un nuevo par de cla
ve y valor.
        piezaPrefabricadaDic.Add (piezasPrefabricadas [i].tipo, piezasPrefabr
icadas [i].prefabricado);
      }
    }
    // Recorre todas las filas de 0 a la dimensionX.
    for (int x = 0; x < dimensionX; x++) {
      // Recorre todas las columnas de 0 a dimensionY.
      for (int y = 0; y < dimensionY; y++) {
        // En cada celda de la cuadrícula se instancia el fondo prefabricado.
      GameObject fondo = (GameObject)Instantiate (fondoPrefabricado, ObtenerP
osicion(x, y), Quaternion.identity);
        // Hace que el fondo sea hijo de la cuadrícula.
        fondo.transform.parent = transform;
      }
    }
    // Crea una instancia de la matriz de piezas del juego para que tenga las
```

```
 mismas dimensiones que la cuadricula.
    piezas = new PiezaJuego[dimensionX, dimensionY];
    // Recorre todas las piezas iniciales.
    for (int i = 0; i < piezasIniciales.Length; i++) {
      // Verifica que las coordenadas "x" e "y" de las piezas iniciales estén
 dentro de las dimensiones de la cuadrícula.
      if (piezasIniciales[i].x >= 0 && piezasIniciales[i].x < dimensionX && p
iezasIniciales[i].y >= 0 && piezasIniciales[i].y < dimensionY) {
        // Crea las piezas iniciales.
        GenerarNuevaPieza (piezasIniciales[i].x, piezasIniciales[i].y, piezas
Iniciales[i].tipo);
      }
    }
    // Recorre todas las filas de 0 a dimensionX.
    for (int x = 0; x < dimensionX; x++) {
      // Recorre todas las columnas de 0 a dimensionY.
      for (int y = 0; y < dimensionY; y++) {
        // Verifica que no haya una pieza creada en esa posición.
        if (piezas [x, y] == null) {
          // Crea las piezas del juego de tipo vacío.
          GenerarNuevaPieza (x, y, TipoPieza.VACIO);
        }
      }
    }

    // Llama a la función Llenado para que en iniciar el juego se llene la cu
adrícula. StartCoroutine es necesario para establecer la corrutina.
    StartCoroutine(Llenado ());
  }

  // Se usa para la inicialización.
  void Start() {
  }

  // La actualización se produce una vez por frame.
  void Update() {
  }

  // Llama a PasoLlenado hasta que se llena la cuadrícula.
  // Una corrutina es una función que tiene la habilidad de pausar su ejecuci
ón y devolver el control a Unity para luego continuar donde lo dejó en el sig
uiente frame.
  public IEnumerator Llenado() {
    // Indica que la cuadrícula necesita rellenarse.
    bool necesitaRellenar = true;
    // Indica que la cuadrícula necesita rellenarse.
    bool necesitaRellenar2 = true;
    // Indica que la cuadrícula se está llenando.
    estaLlenando = true;
    while (necesitaRellenar || necesitaRellenar2) {
      // Así hay tiempo entre limpiar las piezas y volver a ponerlas.
      yield return new WaitForSecondsRealtime (tiempoLlenado);
      while (PasoLlenado()) {
        // Invierte el valor de inverso para que vaya cambiando la dirección.
        inverso = !inverso;
        // Así el llenado ocurre gradualmente con el tiempo.
        yield return new WaitForSecondsRealtime (tiempoLlenado);
      }
      // Permite rellenar la cuadrícula si BorrarConjuntosValidos da cierto.
```

```
        necesitaRellenar = BorrarConjuntosValidos();
        // Verifica si ya se han borrado todos los conjuntos validos.
        if (necesitaRellenar == false) {
          // Permite rellenar la cuadrícula si ReorganizarPiezas da cierto.
          necesitaRellenar2 = ReorganizarPiezas();
        }
      }
      // Indica que la cuadrícula no se está llenando.
      estaLlenando = false;
    }

    // Mueve cada pieza un espacio hacia abajo.
    public bool PasoLlenado() {
      // Comienza con el booleano en falso.
      bool PiezaMovida = false;
      // Recorre todas las columnas de abajo a arriba.
      // Pone dimensionY-2 porque se ignora la fila inferior de la cuadricula,
ya que las piezas de esa fila no se pueden mover hacia abajo.
      for (int y = dimensionY - 2; y >= 0; y--) {
        // Recorre todas las filas.
        for (int voltearX = 0; voltearX < dimensionX; voltearX++) {
          // Hace que se recorra desde 0 a dimensiónX.
          int x = voltearX;
          // Hace que se recorra desde dimensiónX a 0.
          if (inverso) {
            x = dimensionX - 1 - voltearX;
          }
          // Obtiene la pieza del juego de la posición actual.
          PiezaJuego pieza = piezas [x, y];
          // Verifica si la pieza es móvil.
          if (pieza.EsMovil ()) {
            // Obtiene la pieza de abajo de la pieza actual.
            PiezaJuego abajoPieza = piezas [x, y + 1];
            // Verifica si la pieza de abajo está vacía.
            if (abajoPieza.Tipo == TipoPieza.VACIO) {
              // Destruye la pieza vacía.
              Destroy (abajoPieza.gameObject);
              // Mueve la pieza actual al espacio de abajo de ésta.
              pieza.ComponenteMovil.Movimento (x, y + 1, tiempoLlenado);
              piezas [x, y + 1] = pieza;
              //Genera en el espacio actual una nueva pieza vacia.
              GenerarNuevaPieza (x, y, TipoPieza.VACIO);
              // Pone el booleano en verdadero.
              PiezaMovida = true;
            // Si no se puede mover al espacio de abajo.
            } else {
              // Recorre las diagonales, abajo a la izquierda y a la derecha.
              for (int diagonal = -1; diagonal <= 1; diagonal++) {
                // Verifica que no sea la pieza de abajo.
                if (diagonal != 0) {
                  // Parámetro para la "x" de la pieza que está en diagonal.
El valor de "diagonal" indica si es la de la izquierda o la de la derecha.
                  int diagonalX = x + diagonal;
                  // Verifica si está "inverso" en verdadero.
                  if (inverso) {
                    // El parámetro recibe el valor de "diagonal" invertido.
                    diagonalX = x - diagonal;
                  }
                  // Verifica si la coordenada "x" de la pieza diagonal está de
```

```
ntro de los límites de la cuadricula.
                if (diagonalX >= 0 && diagonalX < dimensionX) {
                  // Obtiene la pieza de abajo en diagonal.
                  PiezaJuego diagonalPieza = piezas [diagonalX, y + 1];
                  // Verifica si la pieza de abajo en diagonal está vacía.
                  if (diagonalPieza.Tipo == TipoPieza.VACIO) {
                    // Parámetro para saber si hay una pieza arriba de la pie
za de abajo en diagonal que se pueda mover.
                    bool tienePiezaArriba = true;
                    // Recorre todas las piezas de arriba de la pieza de abaj
o en diagonal.
                    for (int arribaY = y; arribaY >= 0; arribaY--) {
                      // Obtiene la pieza de arriba de la pieza de abajo en d
iagonal.
                      PiezaJuego piezaArriba = piezas [diagonalX, arribaY];
                      // Verifica si la pieza de arriba es móvil.
                      if (piezaArriba.EsMovil()) {
                        // Sale del bucle.
                        break;
                       // Si no, verifica si la pieza de arriba no es móvil y
 no es de tipo vacío.
                      } else if (!piezaArriba.EsMovil() && piezaArriba.Tipo !
= TipoPieza.VACIO) {
                        // Pone el booleano en falso.
                        tienePiezaArriba = false;
                        // Sale del bucle.
                        break;
                      }
                    }
                    // Verifica si no tiene una pieza arriba.
                    if (!tienePiezaArriba) {
                      // Destruye la pieza vacía.
                      Destroy (diagonalPieza.gameObject);
                      // Mueve la pieza actual al espacio de abajo en diagona
l de ésta.
                      pieza.ComponenteMovil.Movimento (diagonalX, y + 1, tiem
poLlenado);
                      piezas [diagonalX, y + 1] = pieza;
                      //Genera en el espacio actual una nueva pieza vacía.
                      GenerarNuevaPieza (x, y, TipoPieza.VACIO);
                      // Pone el booleano en verdadero.
                      PiezaMovida = true;
                      // Sale del bucle.
                      break;
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
    // Esto es para la fila superior, ya que no hay piezas encima de la fila
superior.
    // Recorre toda la fila.
    for (int x = 0; x < dimensionX; x++) {
      // Obtiene la pieza del juego.
      PiezaJuego abajoPieza = piezas [x, 0];
```

```
      // Verifica si la pieza está vacía.
      if (abajoPieza.Tipo == TipoPieza.VACIO) {
        // Destruye la pieza vacía.
        Destroy (abajoPieza.gameObject);
        // Crea una nueva pieza encima de la fila superior. Instancia un nuev
o GameObject.
        GameObject nuevaPieza = (GameObject)Instantiate(piezaPrefabricadaDic[
TipoPieza.NORMAL], ObtenerPosicion(x, -1), Quaternion.identity);
        // Hace que la pieza del juego sea hija de la cuadricula.
        nuevaPieza.transform.parent = transform;
        // Asigna la nueva pieza en la coordenada "x" de la fila 0 de la matr
iz de piezas del juego.
        piezas [x, 0] = nuevaPieza.GetComponent<PiezaJuego> ();
        // Inicializa la pieza. Se pone -
1 en lugar de la coordenada y, ya que la matriz no tiene índice negativo.
        piezas [x, 0].Init (x, -1, this, TipoPieza.NORMAL);
        // Usa componenteMovil para mover la pieza. Mueve la pieza de -1 a 0.
        piezas [x, 0].ComponenteMovil.Movimento(x, 0, tiempoLlenado);
        // Usa componenteColor para establecer un color aleatorio.
        piezas [x, 0].ComponenteColor.EstablecerColor((PiezaColor.TipoColor)R
andom.Range(0, piezas [x, 0].ComponenteColor.NumColores));
        // Pone el booleano en verdadero.
        PiezaMovida = true;
      }
    }
    // Devuelve verdadero si ha movido la pieza y en falso si no.
    return PiezaMovida;
  }

  // Modifica la posición de las celdas de la cuadricula para centrar la cuad
ricula.
  public Vector2 ObtenerPosicion(int x, int y) {
    return new Vector2 (transform.position.x -
 dimensionX / 2.0f + x, transform.position.y + dimensionY / 2.0f - y);
  }

  // Función que genera una nueva pieza del juego.
  public PiezaJuego GenerarNuevaPieza(int x, int y, TipoPieza tipo) {
    // Instancia un nuevo GameObject. Toma el tipo de pieza y las coordenadas
 "x" e "y".
    GameObject nuevaPieza = (GameObject)Instantiate(piezaPrefabricadaDic[tipo
], ObtenerPosicion(x, y), Quaternion.identity);
    // Hace que la pieza del juego sea hija de la cuadricula.
    nuevaPieza.transform.parent = transform;
    // Asigna la nueva pieza en la matriz de piezas del juego.
    piezas [x, y] = nuevaPieza.GetComponent<PiezaJuego>();
    // Inicializa la pieza. Llama a la función Init para pasar las coordenada
s "x" e "y", la clase de cuadricula y el tipo de pieza.
    piezas [x, y].Init(x, y, this, tipo);
    // Devuelve la nueva pieza.
    return piezas [x, y];
  }

  // Toma dos piezas del juego y devuelve un booleano.
  public bool EsAdjacente(PiezaJuego pieza1, PiezaJuego pieza2) {
    // Devuelve verdadero si las dos piezas tienen la misma coordenada "x" y
las coordenadas "y" son adjacentes.
    // Y tambien devuelve verdadero si las dos piezas tienen la misma coorden
ada "y" y las coordenadas "x" son adjacentes.
```

```
    return (pieza1.X == pieza2.X && (int)Mathf.Abs(pieza1.Y -
pieza2.Y) == 1) || (pieza1.Y == pieza2.Y && (int)Mathf.Abs(pieza1.X -
pieza2.X) == 1);
 }

 // Intercambia la posición de dos piezas.
 public void IntercambiarPiezas(PiezaJuego pieza1, PiezaJuego pieza2) {
   // Verifica si el juego ha terminado.
   if (juegoAcabado) {
     // Impide que el jugador pueda hacer mas intercambios de piezas una vez
terminado el juego.
     return;
   }
   // Verifica si las dos piezas son móviles.
   if (pieza1.EsMovil() && pieza2.EsMovil()) {
     // Asigna la posición de la pieza 2 a la pieza 1 en la matriz de piezas
     piezas [pieza1.X, pieza1.Y] = pieza2;
     // Asigna la posición de la pieza 1 a la pieza 2 en la matriz de piezas
     piezas [pieza2.X, pieza2.Y] = pieza1;
     // Verifica si cualquiera de las piezas intercambiadas crean un conjunt
o, mirando si la función ObtenerConjunto devuelve un valor diferente a nulo.
     // Tambien verifica si el tipo de una de la piezas es "COLOR".
     if (ObtenerConjunto (pieza1, pieza2.X, pieza2.Y) != null || ObtenerConj
unto (pieza2, pieza1.X, pieza1.Y) != null || pieza1.Tipo == TipoPieza.COLOR |
| pieza2.Tipo == TipoPieza.COLOR) {
       // Almacena las coordenadas de la pieza 1 en unas variables temporale
s para que no se pierdan al mover la pieza.
       int pieza1X = pieza1.X;
       int pieza1Y = pieza1.Y;
       // Pasa la pieza 1 a la posición de la pieza 2.
       pieza1.ComponenteMovil.Movimento (pieza2.X, pieza2.Y, tiempoLlenado);
       // Pasa la pieza 2 a la posición de la pieza 1.
       pieza2.ComponenteMovil.Movimento (pieza1X, pieza1Y, tiempoLlenado);
       // Verifica si la pieza 1 es de tipo "COLOR" y es borrable y si la pi
eza 2 tiene color.
       if (pieza1.Tipo == TipoPieza.COLOR && pieza1.EsBorrable() && pieza2.T
ieneColor()) {
         // Obtiene una referencia al componente "PiezaBorrarColor" de la pi
eza 1.
         PiezaBorrarColor limpiarColor = pieza1.GetComponent<PiezaBorrarColo
r>();
         // Establece el color de "PiezaBorrarColor" con el color de la piez
a 2.
         if (limpiarColor) {
           limpiarColor.Color = pieza2.ComponenteColor.Color;
         }
         // Borra la pieza de tipo "COLOR".
         BorrarPieza (pieza1.X, pieza1.Y);
       }
       // Verifica si la pieza 2 es de tipo "COLOR" y es borrable y si la pi
eza 1 tiene color.
       if (pieza2.Tipo == TipoPieza.COLOR && pieza2.EsBorrable() && pieza1.T
ieneColor()) {
         // Obtiene una referencia al componente "PiezaBorrarColor" de la pi
eza 2.
         PiezaBorrarColor limpiarColor = pieza2.GetComponent<PiezaBorrarColo
r>();
         // Establece el color de "PiezaBorrarColor" con el color de la piez
a 1.
```

```
          if (limpiarColor) {
            limpiarColor.Color = pieza1.ComponenteColor.Color;
          }
          // Borra la pieza de tipo "COLOR".
          BorrarPieza (pieza2.X, pieza2.Y);
        }
        // Llama a función para borrar todos los conjuntos de la cuadrícula.
        BorrarConjuntosValidos();
        // Verifica si la pieza uno del intercambio es una pieza especial.
        if (pieza1.Tipo == TipoPieza.FILA || pieza1.Tipo == TipoPieza.COLUMNA
) {
          // Borra la pieza.
          BorrarPieza (pieza1.X, pieza1.Y);
        }
        // Verifica si la pieza dos del intercambio es una pieza especial.
        if (pieza2.Tipo == TipoPieza.FILA || pieza2.Tipo == TipoPieza.COLUMNA
) {
          // Borra la pieza.
          BorrarPieza (pieza2.X, pieza2.Y);
        }
        // Asigna un valor nulo a las dos piezas del intercambio.
        piezaPresionado = null;
        piezaEntrado = null;
        // Llama a la función Llenado para que se llene la cuadricula.
        // StartCoroutine es necesario para establecer la corrutina.
        StartCoroutine (Llenado());
        // Llama a la función de movimiento de la clase Nivel.
        nivel.EnMovimiento();
      // Si no es así.
      } else {
        // Cambia las piezas a sus posiciones originales.
        piezas [pieza1.X, pieza1.Y] = pieza1;
        piezas [pieza2.X, pieza2.Y] = pieza2;
      }
    }
  }

  // Función para cuando se presiona con el ratón en una pieza.
  public void PresionarPieza(PiezaJuego pieza) {
    piezaPresionado = pieza;
  }

  // Función para cuando se entra con el ratón en una pieza.
  public void EntrarPieza(PiezaJuego pieza) {
    piezaEntrado = pieza;
  }

  // Función para cuando se deja de presionar con el ratón en una pieza.
  public void SoltarPieza(PiezaJuego pieza) {
    // Verifica si la pieza presionada y la pieza sobre la que está el ratón
son adyacentes.
  if (EsAdjacente(piezaPresionado, piezaEntrado)) {
    // Intercambia la posición de las dos piezas.
    IntercambiarPiezas (piezaPresionado, piezaEntrado);
  }
}

  // Función para determinar si se puede hacer un conjunto con una pieza dada
 en las posiciones a las que se puede mover.
```

```
  public List<PiezaJuego> ObtenerConjunto(PiezaJuego pieza, int nuevoX, int n
uevoY) {
    // Verifica si la pieza tiene color.
    if (pieza.TieneColor()) {
      // Almacena el color de la pieza.
      PiezaColor.TipoColor color = pieza.ComponenteColor.Color;
      // Variables para almacenar piezas adyacentes del mismo color que la pi
eza.
      // Piezas adyacentes en posición horizontal.
      List<PiezaJuego> horizontalPiezas = new List<PiezaJuego> ();
      // Piezas adyacentes en posición vertical.
      List<PiezaJuego> verticalPiezas = new List<PiezaJuego> ();
      // Piezas adyacentes que forman parte de un conjunto.
      List<PiezaJuego> conjuntoPiezas = new List<PiezaJuego> ();

      // Verifica horizontalmente si hay coincidencia de color con otras piez
as.
      // Añade a la lista horizontal la pieza dada.
      horizontalPiezas.Add (pieza);
      // Marca la dirección en la que se hacen los recorridos.
      for (int direccion = 0; direccion <= 1; direccion++) {
        // Recorre todas las piezas adyacentes de la pieza dada.
        for (int xAdyacente = 1; xAdyacente < dimensionX; xAdyacente++) {
          // Almacena la coordenada "x" real de la pieza.
          int x;
          // Si direccion es 0 se va hacia la izquierda.
          if (direccion == 0) {
            // La coordenada "x" es nuevoX menos el desplazamiento.
            x = nuevoX - xAdyacente;
            // Si dirección no es 0 se va hacia la derecha.
          } else {
            // La coordenada "x" es nuevoX más el desplazamiento.
            x = nuevoX + xAdyacente;
          }
          // Verifica si la "x" sale de los margenes de la cuadricula.
          if (x < 0 || x >= dimensionX) {
            // Sale del bucle.
            break;
          }
          // Verifica si la pieza adyacente es valida para hacer un conjunto.
          // Verifica si la pieza tiene color y si el color es el mismo que l
a pieza dada.
          if (piezas [x, nuevoY].TieneColor() && piezas [x, nuevoY].Component
eColor.Color == color) {
            // Agrega la pieza a la lista de piezas horizontales.
            horizontalPiezas.Add (piezas [x, nuevoY]);
          // Si no es así.
          } else {
            // Se deja de mirar las piezas adyacentes en esta dirección.
            break;
          }
        }
      }
      // Verifica si hay suficientes piezas para formar un conjunto. Minimo t
res piezas.
      if (horizontalPiezas.Count >= 3) {
        // Recorre la lista de piezas horizontales.
        for (int i = 0; i < horizontalPiezas.Count; i++) {
          // Añade las piezas de la lista de piezas horizontales a la lista d
```

```
e piezas de conjunto.
          conjuntoPiezas.Add (horizontalPiezas [i]);
        }
      }
      // Verifica si hay piezas para formar un conjunto vertical en las pieza
s adyacentes arriba y abajo del conjunto horizontal.
      // Verifica si hay un conjunto horizontal.
      if (horizontalPiezas.Count >= 3) {
        // Recorre todas las piezas del conjunto horizontal.
        for (int i = 0; i < horizontalPiezas.Count; i++) {
          // Marca la dirección en la que se hacen los recorridos.
          for (int direccion = 0; direccion <= 1; direccion++) {
            // Recorre todas las piezas adyacentes de la pieza que se esta mi
rando del conjunto horizontal.
            for (int yAdyacente = 1; yAdyacente < dimensionY; yAdyacente++) {
              // Almacena la coordenada "y" real de la pieza.
              int y;
              // Si direccion es 0 se va hacia arriba.
              if (direccion == 0) {
                // La coordenada "y" es nuevoY menos el desplazamiento.
                y = nuevoY - yAdyacente;
              // Si dirección no es 0 se va hacia abajo.
              } else {
                // La coordenada "y" es nuevoY más el desplazamiento.
                y = nuevoY + yAdyacente;
              }
              // Verifica si la coordenada "y" sale de los margenes de la cua
dricula.
              if (y < 0 || y >= dimensionY) {
                // Sale del bucle.
                break;
              }
              // Verifica si la pieza adyacente es valida para hacer un conju
nto.
              // Verifica si la pieza tiene color y si el color es el mismo q
ue la pieza que se esta mirando del conjunto horizontal.
              if (piezas [horizontalPiezas[i].X, y].TieneColor() && piezas [h
orizontalPiezas[i].X, y].ComponenteColor.Color == color) {
                // Agrega la pieza a la lista de piezas verticales.
                verticalPiezas.Add (piezas [horizontalPiezas[i].X, y]);
              // Si no es así.
              } else {
                // Se deja de mirar las piezas adyacentes en esta dirección.
                break;
              }
            }
          }
          // Verifica si hay suficiente piezas verticales para hacer un conju
nto.
          if (verticalPiezas.Count < 2) {
            // Si no hay suficientes piezas se limpia la lista de piezas vert
icales.
            verticalPiezas.Clear();
          // Si hay suficientes piezas.
          } else {
            // Recorre la lista de piezas verticales.
            for (int j = 0; j < verticalPiezas.Count; j++) {
              // Añade las piezas de la lista de piezas verticales a la lista
 de piezas de conjunto.
```

```
            conjuntoPiezas.Add (verticalPiezas [j]);
          }
          // Se sale del bucle.
          break;
        }
      }
    }
    // Verifica si hay tres o más piezas en la lista de piezas de conjunto.
    if (conjuntoPiezas.Count >= 3) {
      // Devuelve la lista de piezas de conjunto.
      return conjuntoPiezas;
    }
    // Limpia la lista de piezas horizontales.
    horizontalPiezas.Clear();
    // Limpia la lista de piezas verticales.
    verticalPiezas.Clear();
    // Verifica verticalmente si hay coincidencia de color con otras piezas
    // Añade a la lista vertical la pieza dada.
    verticalPiezas.Add (pieza);
    // Marca la dirección en la que se hacen los recorridos.
    for (int direccion = 0; direccion <= 1; direccion++) {
      // Recorre todas las piezas adyacentes de la pieza dada.
      for (int yAdyacente = 1; yAdyacente < dimensionY; yAdyacente++) {
        // Almacena la coordenada "y" real de la pieza.
        int y;
        // Si direccion es 0 se va hacia arriba.
        if (direccion == 0) {
          // La coordenada "y" es nuevoY menos el desplazamiento.
          y = nuevoY - yAdyacente;
          // Si dirección no es 0 se va hacia abajo.
        } else {
          // La coordenada "y" es nuevoY más el desplazamiento.
          y = nuevoY + yAdyacente;
        }
        // Verifica si la coordenada "y" sale de los margenes de la cuadric
ula.
        if (y < 0 || y >= dimensionY) {
          // Sale del bucle.
          break;
        }
        // Verifica si la pieza adyacente es valida para hacer un conjunto.
        // Verifica si la pieza tiene color y si el color es el mismo que l
a pieza dada.
        if (piezas [nuevoX, y].TieneColor() && piezas [nuevoX, y].Component
eColor.Color == color) {
          // Agrega la pieza a la lista de piezas verticales.
          verticalPiezas.Add (piezas [nuevoX, y]);
        // Si no es así.
        } else {
          // Se deja de mirar las piezas adyacentes en esta dirección.
          break;
        }
      }
    }
    // Verifica si hay suficientes piezas para formar un conjunto. Minimo t
res piezas.
    if (verticalPiezas.Count >= 3) {
      // Recorre la lista de piezas verticales.
      for (int i = 0; i < verticalPiezas.Count; i++) {
```

```
          // Añade las piezas de la lista de piezas verticales a la lista de
piezas de conjunto.
          conjuntoPiezas.Add (verticalPiezas [i]);
        }
      }
      // Verifica si hay piezas para formar un conjunto horizontal en las pie
zas adyacentes izquierda y derecha del conjunto vertical.
      // Verifica si hay un conjunto vertical.
      if (verticalPiezas.Count >= 3) {
        // Recorre todas las piezas del conjunto horizontal.
        for (int i = 0; i < verticalPiezas.Count; i++) {
          // Marca la dirección en la que se hacen los recorridos.
          for (int direccion = 0; direccion <= 1; direccion++) {
            // Recorre todas las piezas adyacentes de la pieza que se esta mi
rando del conjunto vertical.
            for (int xAdyacente = 1; xAdyacente < dimensionX; xAdyacente++) {
              // Almacena la coordenada "x" real de la pieza.
              int x;
              // Si direccion es 0 se va hacia la izquierda.
              if (direccion == 0) {
                // La coordenada "x" es nuevoX menos el desplazamiento.
                x = nuevoX - xAdyacente;
                // Si dirección no es 0 se va hacia la derecha.
              } else {
                // La coordenada "x" es nuevoX más el desplazamiento.
                x = nuevoX + xAdyacente;
              }
              // Verifica si la coordenada "x" sale de los margenes de la cua
dricula.
              if (x < 0 || x >= dimensionX) {
                // Sale del bucle.
                break;
              }
              // Verifica si la pieza adyacente es valida para hacer un conju
nto.
              // Verifica si la pieza tiene color y si el color es el mismo q
ue la pieza que se esta mirando del conjunto vertical.
              if (piezas [x, verticalPiezas[i].Y].TieneColor() && piezas [x,
verticalPiezas[i].Y].ComponenteColor.Color == color) {
                // Agrega la pieza a la lista de piezas horizontales.
                horizontalPiezas.Add (piezas [x, verticalPiezas[i].Y]);
              // Si no es así.
              } else {
                // Se deja de mirar las piezas adyacentes en esta dirección
                break;
              }
            }
          }
          // Verifica si hay suficiente piezas verticales para hacer un conju
nto.
          if (horizontalPiezas.Count < 2) {
            // Si no hay suficientes piezas se limpia la lista de piezas vert
icales.
            horizontalPiezas.Clear();
          // Si hay suficientes piezas.
          } else {
            // Recorre la lista de piezas verticales.
            for (int j = 0; j < horizontalPiezas.Count; j++) {
              // Añade las piezas de la lista de piezas verticales a la lista
```

```
 de piezas de conjunto.
              conjuntoPiezas.Add (horizontalPiezas [j]);
            }
            // Se sale del bucle.
            break;
          }
        }
      }
      // Verifica si hay tres o más piezas en la lista de piezas de conjunto.
      if (conjuntoPiezas.Count >= 3) {
        // Devuelve la lista de piezas de conjunto.
        return conjuntoPiezas;
      }
    }
    // Devuelve nulo, ya que no hay piezas para formar un conjunto.
    return null;
  }

  // Función para borrar todos los conjuntos de la cuadrícula.
  public bool BorrarConjuntosValidos() {
    // Indica que la cuadrícula no necesita rellenarse.
    bool necesitaRellenar = false;
    // Recorre todas las columnas de 0 a dimensionY de la cuadrícula.
    for (int y = 0; y < dimensionY; y++) {
      // Recorre todas las filas de 0 a dimensionX de la cuadrícula.
      for (int x = 0; x < dimensionX; x++) {
        // Verifica si la pieza es borrable
        if (piezas [x, y].EsBorrable()) {
          // Verifica si se puede hacer un conjunto.
          List<PiezaJuego> conjunto = ObtenerConjunto (piezas [x, y], x, y);
          if (conjunto != null) {
            // Determina si se tiene que generar una pieza especial.
            // Pone por defecto CONTADOR porque no corresponde con ningun tip
o de pieza.
            TipoPieza especialTipoPieza = TipoPieza.CONTADOR;
            // Obtiene una pieza aleatoria del conjunto.
            PiezaJuego aleatorioPieza = conjunto [Random.Range (0, conjunto.C
ount)];
            // Establece la posición de "x" e "y" de la pieza especial.
            int especialPiezaX = aleatorioPieza.X;
            int especialPiezaY = aleatorioPieza.Y;
            // Verifica si el conjunto contiene cuatro piezas.
            if (conjunto.Count == 4) {
              // Verifica que el conjunto no fue formado por un intercambio d
e piezas.
              if (piezaPresionado == null || piezaEntrado == null) {
                // Asigna la posición de la pieza especial de forma aleatoria
                especialTipoPieza = (TipoPieza)Random.Range ((int)TipoPieza.F
ILA, (int)TipoPieza.COLUMNA);
              // Si no es así y el conjunto fue formado por un intercambio de
 piezas.
              // Verifica si el intercambio de piezas fue entre dos piezas de
 una misma fila.
              } else if (piezaPresionado.Y == piezaEntrado.Y) {
                // Asigna el tipo de pieza especial que borra una fila.
                especialTipoPieza = TipoPieza.FILA;
              // Si no es así. es que el intercambio de piezas fue entre dos
piezas de una misma columna.
              } else {
```

```
                // Asigna el tipo de pieza especial que borra una columna.
                especialTipoPieza = TipoPieza.COLUMNA;
              }
            // Si no es así. Verifica si el conjunto contiene cinco piezas o
más.
            } else if (conjunto.Count >= 5) {
              // Asigna el tipo de pieza especial que borra un color.
              especialTipoPieza = TipoPieza.COLOR;
            }
            // Recorre la lista de piezas.
            for (int i = 0; i < conjunto.Count; i++) {
              // Borra las piezas.
              if (BorrarPieza (conjunto[i].X, conjunto[i].Y)) {
                // Indica que la cuadrícula necesita rellenarse.
                necesitaRellenar = true;
                // Establece la posición de la pieza especial en las piezas d
el intercambio.
                if (conjunto[i] == piezaPresionado || conjunto[i] == piezaEnt
rado) {
                  especialPiezaX = conjunto[i].X;
                  especialPiezaY = conjunto[i].Y;
                }
              }
            }
            // Verifica si hay que generar una pieza especial, mirando si el
tipo de pieza es diferente del tipo CONTADOR.
            if (especialTipoPieza != TipoPieza.CONTADOR) {
              // Destruye la pieza vacía.
              Destroy(piezas [especialPiezaX, especialPiezaY]);
              // Genera la pieza especial.
              PiezaJuego nuevaPieza = GenerarNuevaPieza (especialPiezaX, espe
cialPiezaY, especialTipoPieza);
              // Verifica el color de la primera pieza del conjunto y pone el
 mismo a la pieza especial.
              if ((especialTipoPieza == TipoPieza.FILA || especialTipoPieza =
= TipoPieza.COLUMNA) && nuevaPieza.TieneColor() && conjunto[0].TieneColor())
{
                // Establece el color de la nueva pieza especial con el color
 de las piezas del conjunto.
                nuevaPieza.ComponenteColor.EstablecerColor(conjunto[0].Compon
enteColor.Color);
              // Si no es así.
              } else if (especialTipoPieza == TipoPieza.COLOR && nuevaPieza.T
ieneColor()) {
                // Establece el color de la nueva pieza especial con el color
 del tipo de pieza "COLOR".
                nuevaPieza.ComponenteColor.EstablecerColor(PiezaColor.TipoCol
or.ALGUNO);
              }
            }
          }
        }
      }
    }
    // Devuelve el parámetro que indica si necesita rellenarse la cuadrícula
con nuevas piezas.
    return necesitaRellenar;
  }
```

```
  // Función para borrar una pieza. Toma la posición de la pieza en la cuadrí
cula para borrar la pieza.
  public bool BorrarPieza (int x, int y) {
    // Verifica si la pieza es borrable y si aún no se ha marcado para borrar
se.
    if (piezas [x, y].EsBorrable() && !piezas [x, y].ComponenteBorrable.EstaB
orrandose) {
      // Borra la pieza.
      piezas [x, y].ComponenteBorrable.Borrado();
      // Crea una pieza nueva vacía.
      GenerarNuevaPieza (x, y, TipoPieza.VACIO);
      // Elimina cualquier pieza de tipo enemigo de alrededor de la pieza bor
rada.
      BorrarEnemigos (x, y);
      // Devuelve verdadero ya que se ha borrado la pieza.
      return true;
    }
    // Devuelve falso ya que no se ha borrado la pieza.
    return false;
  }

  // Función para borrar piezas de tipo enemigo.
  public void BorrarEnemigos (int x, int y) {
    // Recorre las piezas adyacentes a la izquierda y a la derecha.
    for (int xAdyacente = x - 1; xAdyacente <= x + 1; xAdyacente++) {
      // Verifica si la pieza adyacente está dentro de los límites de la cuad
ricula.
      if (xAdyacente != x && xAdyacente >= 0 && xAdyacente < dimensionX) {
        // Verifica si la pieza adyacente es de tipo enemigo y si es borrable
        if (piezas [xAdyacente, y].Tipo == TipoPieza.ENEMIGO && piezas [xAdya
cente, y].EsBorrable()) {
          // Llama a la función de borrado.
          piezas [xAdyacente, y].ComponenteBorrable.Borrado();
          // Genera una nueva pieza de tipo vacío.
          GenerarNuevaPieza (xAdyacente, y, TipoPieza.VACIO);
        }
      }
    }
    // Recorre las piezas adyacentes arriba y abajo.
    for (int yAdyacente = y - 1; yAdyacente <= y + 1; yAdyacente++) {
      // Verifica si la pieza adyacente está dentro de los límites de la cuad
ricula.
      if (yAdyacente != y && yAdyacente >= 0 && yAdyacente < dimensionY) {
        // Verifica si la pieza adyacente es de tipo enemigo y si es borrable
        if (piezas [x, yAdyacente].Tipo == TipoPieza.ENEMIGO && piezas [x, yA
dyacente].EsBorrable()) {
          // Llama a la función de borrado.
          piezas [x, yAdyacente].ComponenteBorrable.Borrado();
          // Genera una nueva pieza de tipo vacío.
          GenerarNuevaPieza (x, yAdyacente, TipoPieza.VACIO);
        }
      }
    }
  }

  // Función para mirar la fila que se tiene que borrar.
  public void LimpiarFila(int fila) {
    // Recorre la fila.
    for (int x = 0; x < dimensionX; x++) {
```

```
      // Borra cada una de las piezas de la fila.
      BorrarPieza (x, fila);
    }
  }

  // Función para mirar la columna que se tiene que borrar.
  public void LimpiarColumna(int columna) {
    // Recorre la columna.
    for (int y = 0; y < dimensionY; y++) {
      // Borra cada una de las piezas de la columna.
      BorrarPieza (columna, y);
    }
  }

  // Función para coger el color que se tiene que borrar.
  public void LimpiarColor(PiezaColor.TipoColor color){
    // Recorre todas las piezas de la cuadrícula.
    for (int x = 0; x < dimensionX; x++) {
      for (int y = 0; y < dimensionY; y++) {
        // Verifica si la pieza tiene color y si el color de la pieza es el m
ismo que el color pasado a la función o es "ALGUNO".
        if (piezas [x, y].TieneColor() && (piezas [x, y].ComponenteColor.Colo
r == color || color == PiezaColor.TipoColor.ALGUNO)) {
          // Borra cada una de las piezas de ese color.
          BorrarPieza (x, y);
        }
      }
    }
  }

  // Función para contar cuantas piezas de tipo enemigo hay en la cuadrícula.
 Devolvera todas las piezas de un tipo dado.
  public List<PiezaJuego> ObtenerPiezasTipo(TipoPieza tipo)  {
    // Crea una lista para contener las piezas.
    List<PiezaJuego> piezasTipo = new List<PiezaJuego> ();
    // Recorre todas las piezas de la cuadrícula.
    for (int x = 0; x < dimensionX; x++) {
      for (int y = 0; y < dimensionY; y++) {
        // Verifica si el tipo de pieza coincide.
        if (piezas [x, y].Tipo == tipo) {
          // Agrega la pieza a la lista.
          piezasTipo.Add (piezas [x, y]);
        }
      }
    }
    // Devuelve la lista.
    return piezasTipo;
  }

  // Función para indicar que el juego ha terminado.
  public void JuegoAcabado() {
    // Indica que el juego ha terminado.
    juegoAcabado = true;
  }

  // Función para reorganizar las piezas cuando no hay movimientos posibles e
n la cuadrícula.
  public bool ReorganizarPiezas()  {
  // Parámetro para contabilizar si hay movimientos posibles en la cuadrícula
```

```
  int contador = 0;
  // Indica que la cuadrícula no necesita rellenarse.
  bool necesitaRellenar2 = false;
  // Recorre todas las columnas de 0 a dimensionY de la cuadrícula.
  for (int y = 0; y < dimensionY; y++) {
    // Recorre todas las filas de 0 a dimensionX de la cuadrícula.
    for (int x = 0; x < dimensionX; x++) {
      // Asigna la pieza en "pieza1".
      PiezaJuego pieza1 = piezas [x, y];
      //Verifica si el tipo de la pieza es "COLOR".
      if (pieza1.Tipo == TipoPieza.COLOR) {
        //Indica que la cuadrícula no necesita reorganizarse.
        contador++;
      // Si no es así. Verifica si la "pieza1" es móvil.
      } else if (pieza1.EsMovil ()) {
        // Verifica que la coordenada "x" de la "pieza1" no esté en el límite
 izquierdo de la cuadrícula.
        if (x != 0) {
          // Asigna la pieza en "pieza2".
          PiezaJuego pieza2 = piezas [x - 1, y];
          // Verifica si la "pieza2" es móvil.
          if (pieza2.EsMovil ()) {
            // Asigna la posición de la pieza 2 a la pieza 1 en la matriz de
piezas.
            piezas [pieza1.X, pieza1.Y] = pieza2;
            // Asigna la posición de la pieza 1 a la pieza 2 en la matriz de
piezas.
            piezas [pieza2.X, pieza2.Y] = pieza1;
            // Verifica si cualquiera de las piezas intercambiadas crean un c
onjunto, mirando si la función ObtenerConjunto devuelve un valor diferente a
nulo.
            // Tambien verifica si el tipo de una de la piezas es "COLOR"
            if (ObtenerConjunto (pieza1, pieza2.X, pieza2.Y) != null || Obten
erConjunto (pieza2, pieza1.X, pieza1.Y) != null) {
              //Indica que la cuadrícula no necesita reorganizarse.
              contador++;
            }
            // Cambia las piezas a sus posiciones originales.
              piezas [pieza1.X, pieza1.Y] = pieza1;
              piezas [pieza2.X, pieza2.Y] = pieza2;
            }
          }
           // Verifica que la coordenada "x" de la "pieza1" no esté en el lím
ite derecho de la cuadrícula.
          if (x != dimensionX - 1) {
            // Asigna la pieza en "pieza2".
            PiezaJuego pieza2 = piezas [x + 1, y];
            // Verifica si la "pieza2" es móvil.
            if (pieza2.EsMovil ()) {
              // Asigna la posición de la pieza 2 a la pieza 1 en la matriz d
e piezas.
              piezas [pieza1.X, pieza1.Y] = pieza2;
              // Asigna la posición de la pieza 1 a la pieza 2 en la matriz d
e piezas.
              piezas [pieza2.X, pieza2.Y] = pieza1;
              // Verifica si cualquiera de las piezas intercambiadas crean un
 conjunto, mirando si la función ObtenerConjunto devuelve un valor diferente
a nulo.
              // También verifica si el tipo de una de la piezas es "COLOR"
```

```
                if (ObtenerConjunto (pieza1, pieza2.X, pieza2.Y) != null || Obt
enerConjunto (pieza2, pieza1.X, pieza1.Y) != null) {
                  //Indica que la cuadrícula no necesita reorganizarse.
                  contador++;
                }
                  // Cambia las piezas a sus posiciones originales.
                  piezas [pieza1.X, pieza1.Y] = pieza1;
                  piezas [pieza2.X, pieza2.Y] = pieza2;
                }
              }
              // Verifica que la coordenada "y" de la "pieza1" no esté en el lí
mite superior de la cuadrícula.
              if (y != 0) {
                // Asigna la pieza en "pieza2".
                PiezaJuego pieza2 = piezas [x, y - 1];
                // Verifica si la "pieza2" es móvil.
                if (pieza2.EsMovil ()) {
                  // Asigna la posición de la pieza 2 a la pieza 1 en la matriz
 de piezas.
                  piezas [pieza1.X, pieza1.Y] = pieza2;
                  // Asigna la posición de la pieza 1 a la pieza 2 en la matriz
 de piezas.
                  piezas [pieza2.X, pieza2.Y] = pieza1;
                  // Verifica si cualquiera de las piezas intercambiadas crean
un conjunto, mirando si la función ObtenerConjunto devuelve un valor diferent
e a nulo.
                  // También verifica si el tipo de una de la piezas es "COLOR"
                  if (ObtenerConjunto (pieza1, pieza2.X, pieza2.Y) != null || O
btenerConjunto (pieza2, pieza1.X, pieza1.Y) != null) {
                    //Indica que la cuadrícula no necesita reorganizarse.
                    contador++;
                  }
                  // Cambia las piezas a sus posiciones originales.
                  piezas [pieza1.X, pieza1.Y] = pieza1;
                  piezas [pieza2.X, pieza2.Y] = pieza2;
                }
              }
              // Verifica que la coordenada "y" de la "pieza1" no esté en el lí
mite inferior de la cuadrícula.
              if (y != dimensionY - 1) {
                // Asigna la pieza en "pieza2".
                PiezaJuego pieza2 = piezas [x, y + 1];
                // Verifica si la "pieza2" es móvil.
                if (pieza2.EsMovil ()) {
                // Asigna la posición de la pieza 2 a la pieza 1 en la matriz d
e piezas.
                  piezas [pieza1.X, pieza1.Y] = pieza2;
                  // Asigna la posición de la pieza 1 a la pieza 2 en la matriz
 de piezas.
                  piezas [pieza2.X, pieza2.Y] = pieza1;
                  // Verifica si cualquiera de las piezas intercambiadas crean
un conjunto, mirando si la función ObtenerConjunto devuelve un valor diferent
e a nulo.
                  // También verifica si el tipo de una de la piezas es "COLOR"
                  if (ObtenerConjunto (pieza1, pieza2.X, pieza2.Y) != null || O
btenerConjunto (pieza2, pieza1.X, pieza1.Y) != null) {
                    //Indica que la cuadrícula no necesita reorganizarse.
                    contador++;
                  }
```

```
                // Cambia las piezas a sus posiciones originales.
                piezas [pieza1.X, pieza1.Y] = pieza1;
                piezas [pieza2.X, pieza2.Y] = pieza2;
              }
            }
          }
        }
      }
      // Verifica que no hay movimientos posibles en la cuadrícula.
      if (contador == 0) {
        // Recorre todas las columnas de 0 a dimensionY de la cuadrícula.
        for (int y = 0; y < dimensionY; y++) {
          // Recorre todas las filas de 0 a dimensionX de la cuadrícula.
          for (int x = 0; x < dimensionX; x++) {
            // Verifica si la pieza es movible.
            if (piezas [x, y].EsMovil ()) {
              // Llama a la función de la clase Nivel que resta los puntos ob
tenidos por el borrado de piezas para el reorganizado de las piezas.
              nivel.RestarPuntosReorganizado (piezas [x, y]);
              // Borra las piezas.
              if (piezas [x, y].EsBorrable() && !piezas [x, y].ComponenteBorr
able.EstaBorrandose) {
                // Borra la pieza.
                piezas [x, y].ComponenteBorrable.Borrado();
                // Crea una pieza nueva vacía.
                GenerarNuevaPieza (x, y, TipoPieza.VACIO);
                // Indica que la cuadrícula necesita rellenarse.
                necesitaRellenar2 = true;
              }
            }
          }
        }
      }
    // Devuelve el parámetro que indica si necesita rellenarse la cuadrícula
con nuevas piezas.
    return necesitaRellenar2;
  }
}
```

PiezaJuego

Esta clase es donde se definen las funciones de las piezas del juego, se utiliza para definir las variables de las piezas, poder asignarles los componentes de color, móvil y borrable y poder detectar y controlar las interacciones del ratón sobre las piezas.

Funciones:

- Awake: Se usa para asignar los componentes color, móvil y borrable a unas variables lo antes posible para que no se produzcan excepciones.
- Init: Se usa para inicializar las variables de la pieza.
- OnMouseEnter: Se usa para detectar cuando el ratón entra dentro del área de una pieza.
- OnMouseDown: Se usa para detectar cuando se presiona el ratón en una pieza.
- OnMouseUp: Se usa para detectar cuando se deja de presionar el ratón sobre una pieza.
- EsMovil: Se usa para verificar si la pieza es móvil.
- TieneColor: Se usa para verificar si la pieza tiene color.
- EsBorrable: Se usa para verificar si la pieza es borrable.

A continuación se muestra el código de la clase, comentado línea a línea para mayor compresión de los detalles del funcionamiento del mismo.

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

public class PiezaJuego : MonoBehaviour {

  // Parámetro para la puntuación.
  public int puntos;

  // Parámetros para las coordenadas "x" e "y" de la pieza en la cuadricula.
  private int x;
  private int y;

  // Propiedad pública para acceder a la variable.
  public int X {
    get {
      return x;
    }
    // Establece el valor si la pieza es móvil. Actualiza la variable "x".
    set {
      if (EsMovil ()) {
        x = value;
      }
    }
  }

  // Propiedad pública para acceder a la variable.
  public int Y {
    get {
      return y;
    }
```

```
    // Establece el valor si la pieza es móvil. Actualiza la variable "y".
    set {
      if (EsMovil ()) {
        y = value;
      }
    }
  }

  // Parámetro para obtener información sobre la cuadricula.
  private Cuadricula cuadricula;

  // Propiedad pública para acceder a la variable.
  public Cuadricula CuadriculaReferencia {
    get {
      return cuadricula;
    }
  }

  // Parámetro para el tipo de pieza de la pieza.
  private Cuadricula.TipoPieza tipo;

  // Propiedad pública para acceder a la variable.
  public Cuadricula.TipoPieza Tipo {
    get {
      return tipo;
    }
  }

  // Parámetro para obtener si la pieza es móvil.
  private PiezaMovil componenteMovil;

  // Propiedad pública para acceder a la variable.
  public PiezaMovil ComponenteMovil {
    get {
      return componenteMovil;
    }
  }

  // Parámetro para obtener el color de la pieza.
  private PiezaColor componenteColor;

  // Propiedad pública para acceder a la variable.
  public PiezaColor ComponenteColor {
    get {
      return componenteColor;
    }
  }

  // Parámetro para obtener si la pieza es borrable.
  private PiezaBorrable componenteBorrable;

  // Propiedad pública para acceder a la variable.
  public PiezaBorrable ComponenteBorrable {
    get {
      return componenteBorrable;
    }
  }

  void Awake() {
```

```
    // Asigna a la variable el componente móvil, si no tiene obtendrá nulo.
    componenteMovil = GetComponent<PiezaMovil> ();
    // Asigna a la variable el componente color, si no tiene obtendrá nulo.
    componenteColor = GetComponent<PiezaColor> ();
    // Asigna a la variable el componente borrable, si no tiene obtendrá nulo
    componenteBorrable = GetComponent<PiezaBorrable> ();
  }

  // Se usa para la inicialización.
  void Start() {
  }

  // La actualización se produce una vez por frame.
  void Update() {
  }

  // Inicializa las variables de la pieza.
  public void Init(int _x, int _y, Cuadricula _cuadricula, Cuadricula.TipoPie
za _tipo) {
    x = _x;
    y = _y;
    tipo = _tipo;
    cuadricula = _cuadricula;
  }

  // Función que detecta cuando el ratón entra dentro de un elemento.
  // this hace referencia a la pieza sobre la que se interactúa.
  void OnMouseEnter() {
    cuadricula.EntrarPieza (this);
  }

  // Función que detecta cuando se presiona el ratón en un elemento.
  void OnMouseDown() {
    cuadricula.PresionarPieza (this);
  }

  // Función que detecta cuando se suelta el ratón en un elemento.
  void OnMouseUp() {
    cuadricula.SoltarPieza (this);
  }

  // Verifica si una pieza es móvil mirando si la variable es nula o no.
  public bool EsMovil() {
    return componenteMovil != null;
  }

  // Verifica si una pieza tiene color mirando si la variable es nula o no.
  public bool TieneColor() {
    return componenteColor != null;
  }

  // Verifica si una pieza es borrable mirando si la variable es nula o no.
  public bool EsBorrable() {
    return componenteBorrable != null;
  }
}
```

PiezaMovil

Esta clase se utiliza para mover las piezas de la cuadrícula del juego. Mediante una corrutina se ejecutan y se aprecian con el paso del tiempo los movimientos de las piezas.

Funciones:

- Awake: Se usa para asignar la pieza del juego a la variable de la pieza del juego lo antes posible para que no se produzcan excepciones.
- Movimiento: Se usa para mover piezas del juego.
- MovimientoCorrutina: Se usa para que se ejecuten y se aprecien con el paso del tiempo los movimientos de las piezas.

A continuación se muestra el código de la clase, comentado línea a línea para mayor compresión de los detalles del funcionamiento del mismo.

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

public class PiezaMovil : MonoBehaviour {

  // Parámetro para la pieza del juego.
  private PiezaJuego pieza;

  // Parámetro para el movimiento, corrutina.
  private IEnumerator movimientoCorrutina;

  // Usar Awake asegura que se obtenga la pieza del juego lo antes posible pa
ra que no se produzcan excepciones.
  void Awake() {
    // Asigna a la variable la pieza del juego.
    pieza = GetComponent<PiezaJuego> ();
  }

  // Se usa para la inicialización.
  void Start() {
  }

  // La actualización se produce una vez por frame.
  void Update() {
  }

  // Función que mueve la pieza.
  public void Movimiento(int nuevoX, int nuevoY, float tiempo) {
    // Verifica si la corrutina no es nula.
    if (movimientoCorrutina != null) {
      // Detiene la corrutina.
      StopCoroutine(movimientoCorrutina);
    }
    // Obtengo una nueva corrutina.
    movimientoCorrutina = MovimientoCorrutina(nuevoX, nuevoY, tiempo);
    // Inicia la corrutina.
    StartCoroutine(movimientoCorrutina);
  }
```

```
  // Función para definir el movimiento, corrutina.
  private IEnumerator MovimientoCorrutina(int nuevoX, int nuevoY, float tiemp
o) {
    // Establece los parámetros "x" e "y".
    pieza.X = nuevoX;
    pieza.Y = nuevoY;
    // Posición inicial.
    Vector3 posicionInicial = transform.position;
    // Posición final.
    Vector3 posicionFinal = pieza.CuadriculaReferencia.ObtenerPosicion (nuevo
X, nuevoY);
    // El movimiento va a interpolar entra la posición inicial y la final.
    for (float t = 0; t <= 1 * tiempo; t += Time.deltaTime) {
      pieza.transform.position = Vector3.Lerp (posicionInicial, posicionFinal
, t / tiempo);
      yield return 0;
    }
    // Pone la posición de la pieza en la posición final.
    pieza.transform.position = posicionFinal;
  }
}
```

PiezaColor

Esta clase se utiliza para asignar *sprites* de los elementos a los colores de las piezas normales del juego. Estarán los colores y los sprites en un diccionario de claves y valores para una búsqueda más rápida.

Funciones:

- Awake: Se usa para agregar a un diccionario las claves y los valores de la matriz “coloresSprites”, la relación entre colores y *sprites*, para una búsqueda más rápida.
- EstablecerColor: Se usa para asignar un *sprite* a la pieza del juego según el color que ésta tenga asignado.

A continuación se muestra el código de la clase, comentado línea a línea para mayor compresión de los detalles del funcionamiento del mismo.

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

public class PiezaColor : MonoBehaviour {

  // Enumeración con los posibles colores.
  // Contiene el valor ALGUNO porque hay piezas que pueden coincidir con cual
quier color.
  // Contiene el valor CONTADOR para saber cuántos colores de piezas hay.
  public enum TipoColor {
    AZUL,
    NARANJA,
    VERDE,
    MARRON,
    ROSA,
    GRIS,
    ALGUNO,
    CONTADOR
  };

  // Hace que los parámetros del struct sean visualizables en el Inspector.
  [System.Serializable]
  // Estructura con el tipo de color y el Sprite.
  public struct ColorSprite {
    public TipoColor color;
    public Sprite sprite;
  };

  // Parámetro para una matriz de claves y valores. Las claves serán el tipo
de color y los valores serán el Sprite.
  public ColorSprite[] coloresSprites;
  // Parámetro para el color de la pieza.
  private TipoColor color;

  // Propiedad pública para acceder a la variable.
  public TipoColor Color {
    get {
      return color;
    }
```

```
    set {
      //Llama a la función EstablecerColor.
      EstablecerColor (value);
    }
  }

  // Se obtiene la cantidad de colores posibles.
  public int NumColores {
    // Devuelve la longitud de la matriz coloresSprites que se ha configurado
 desde el Inspector.
    get {
      return coloresSprites.Length;
    }
  }

  // Parámetro para almacenar el Sprite Renderer.
  private SpriteRenderer sprite;
  // Parámetro para asociar cada tipo de color con los sprites. Se asocia una
 clave que será el tipo de color con un valor que será el Sprite.
  // Los diccionarios no se pueden visualizar en el Inspector.
  private Dictionary <TipoColor, Sprite> colorSpriteDic;

  void Awake() {
    // El Sprite Renderer estará en el objeto del juego llamado pieza.
    // Obtiene el componente Sprite Renderer.
    sprite = transform.Find("pieza").GetComponent<SpriteRenderer> ();
    // Crea un nuevo diccionario.
    colorSpriteDic = new Dictionary <TipoColor, Sprite> ();
    // Recorre todos los colores en la matriz de colores.
    for (int i = 0; i < coloresSprites.Length; i++) {
      // Comprueba si el diccionario ya contiene la clave.
      if (!colorSpriteDic.ContainsKey (coloresSprites [i].color)) {
        // Si no contiene la clave, agrega al diccionario un nuevo par de cla
ve y valor.
        colorSpriteDic.Add (coloresSprites [i].color, coloresSprites [i].spri
te);
      }
    }
  }

  // Se usa para la inicialización.
  void Start() {
  }

  // La actualización se produce una vez por frame.
  void Update() {
  }

  // Función que establece un nuevo color.
  public void EstablecerColor(TipoColor nuevoColor) {
    color = nuevoColor;
    // Comprueba si el diccionario ya contiene la clave del nuevo color.
    if (colorSpriteDic.ContainsKey (nuevoColor)) {
      // Si contiene la clave, asignara al Sprite Renderer el Sprite del nuev
o color.
      sprite.sprite = colorSpriteDic [nuevoColor];
    }
  }
}
```

PiezaBorrable

Esta clase se utiliza para borrar las piezas de la cuadrícula del juego. Mediante una corrutina se ejecutan y se aprecian con el paso del tiempo las animaciones del borrado de las piezas.

Funciones:

- Awake: Se usa para asignar la pieza del juego a la variable de la pieza del juego lo antes posible para que no se produzcan excepciones.
- Borrado: Se usa para borrar piezas del juego.
- BorradoCorrutina: Se usa para que se ejecuten y se aprecien con el paso del tiempo las animaciones del borrado de las piezas.

A continuación se muestra el código de la clase, comentado línea a línea para mayor compresión de los detalles del funcionamiento del mismo.

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

public class PiezaBorrable : MonoBehaviour {

  // Parámetro para la animación del borrado de las piezas.
  public AnimationClip animacionBorrado;

  // Parámetro para saber si una pieza esta en proceso de borrado.
  private bool estaBorrandose = false;

  // Propiedad pública para acceder a la variable.
  public bool EstaBorrandose {
    get {
      return estaBorrandose;
    }
  }

  // Parámetro para la pieza del juego.
  // Está protegida para que las clases derivadas puedan acceder a esta varia
ble.
  protected PiezaJuego pieza;

  // Usar Awake asegura que se obtenga la pieza del juego lo antes posible pa
ra que no se produzcan excepciones.
  void Awake() {
    // Asigna a la variable la pieza del juego.
    pieza = GetComponent<PiezaJuego> ();
  }

  // Se usa para la inicialización.
  void Start () {
  }

  // La actualización se produce una vez por frame.
  void Update () {
  }
```

```
  // Función para que otras clases borren la pieza.
  public virtual void Borrado() {
    // Llama a la pieza borrada.
    pieza.CuadriculaReferencia.nivel.EnBorradoPieza(pieza);
    // Pone la variable en verdadero.
    estaBorrandose = true;
    // Inicia la corrutina.
    StartCoroutine (BorradoCorrutina ());
  }

  // Función para el borrado de la pieza, corrutina.
  private IEnumerator BorradoCorrutina() {
    // Obtiene el componente animador.
    Animator animador = GetComponent<Animator> ();
    // Verifica si hay un componente animador.
    if (animador) {
      // Reproduce la animación.
      animador.Play (animacionBorrado.name);
      // Así la animación del borrado de la pieza ocurre gradualmente con el
tiempo.
      yield return new WaitForSecondsRealtime (animacionBorrado.length);
      // Destruye la pieza.
      Destroy (gameObject);
    }
  }
}
```

PiezaBorrarLinea

Esta clase se utiliza para las piezas especiales que borran todas las piezas de una misma fila o columna, haciendo que cuando éstas se borren, se borren todas las piezas de la fila o columna a donde se hizo el intercambio de posición.

Funciones:

- Borrado: Se usa para borrar todas las piezas de una misma fila o columna.

A continuación se muestra el código de la clase, comentado línea a línea para mayor compresión de los detalles del funcionamiento del mismo.

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

public class PiezaBorrarLinea : PiezaBorrable {

  // Parámetro para saber si es una pieza que borra filas o columnas. Si es c
ierto borra filas, si es falso borra columnas.
  public bool esFila;

  // Se usa para la inicialización.
  void Start () {
  }

  // La actualización se produce una vez por frame.
  void Update () {
  }

  // Función para borrar líneas de piezas.
  public override void Borrado() {
    // Llama a la función de la clase base, es decir de PiezaBorrable.
    base.Borrado ();
    // Verifica si es una pieza de fila o de columna.
    if (esFila) {
      // Borra una fila.
      pieza.CuadriculaReferencia.LimpiarFila(pieza.Y);
    // Si no es así.
    } else {
      // Borra una columna.
      pieza.CuadriculaReferencia.LimpiarColumna(pieza.X);
    }
  }
}
```

PiezaBorrarColor

Esta clase se utiliza para la pieza especial que borra todas las piezas de un mismo color, haciendo que cuando ésta se borre, se borren todas las piezas del color con la que hizo el intercambio de posición.

Funciones:

- Borrado: Se usa para borrar todas las piezas de un mismo color.

A continuación se muestra el código de la clase, comentado línea a línea para mayor compresión de los detalles del funcionamiento del mismo.

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

public class PiezaBorrarColor : PiezaBorrable {

    // Parámetro para el color que se tiene que borrar.
    private PiezaColor.TipoColor color;

    // Propiedad pública para acceder a la variable.
    public PiezaColor.TipoColor Color {
        get {
            return color;
        }
        set {
            color = value;
        }
    }

    // Se usa para la inicialización.
    void Start () {
    }

    // La actualización se produce una vez por frame.
    void Update () {
    }

    // Función para borrar todas las piezas de un color.
    public override void Borrado() {
        // Llama a la función de la clase base, es decir de PiezaBorrable.
        base.Borrado ();
        // Borra todas las piezas de un color.
        pieza.CuadriculaReferencia.LimpiarColor(color);
    }
}
```

Nivel

Esta clase se utiliza para el seguimiento del estado de la partida en todos los niveles del juego, establece los puntos actuales en el nivel, actualizándolos conforme se borran piezas, controla la gestión de los puntos en producirse una reorganización de las piezas del juego debido a que ya no haya más movimientos posibles, controla la espera hasta que la cuadrícula se termina de llenar y controla y marca cuando se da la victoria o la derrota en la partida.

Funciones:

- Start: Se usa para establecer los puntos en el nivel.
- JuegoGanado: Se usa para indicar que el jugador ha ganado la partida.
- JuegoPerdido: Se usa para indicar que el jugador ha perdido la partida.
- RestarPuntosReorganizado: Se usa para que en reorganizarse la cuadricula borrándose todas las piezas movibles y volviéndose a rellenar la cuadricula con piezas nuevas no se cuenten los puntos de las piezas borradas.
- EnBorradoPieza: Se usa para sumar los puntos obtenidos, de borrar una pieza, a los puntos que ya tenía el jugador y mostrar los puntos actuales actualizados en la interfaz.
- EsperarCuadriculaLlenado: Se usa para controlar la espera hasta que la cuadrícula se termina de llenar.

A continuación se muestra el código de la clase, comentado línea a línea para mayor compresión de los detalles del funcionamiento del mismo.

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

public class Nivel : MonoBehaviour {

  // Enumeración con los posibles tipos de nivel.
  public enum TipoNivel {
    MOVIMIENTO,
    VIRUS,
    TIEMPO
  };

  // Parámetro para la cuadrícula.
  public Cuadricula cuadricula;

  // Parámetro para el HUD.
  public HUD hud;

  // Parámetros para las puntuaciones necesarias para cada copa.
  public int puntosCopaBronce;
  public int puntosCopaPlata;
  public int puntosCopaOro;

  // Parámetro para la puntuación actual del jugador.
  protected int puntosActuales;

  // Parámetro para saber si el jugador ganó.
```

```
  protected bool gano;

  // Parámetro para el tipo de nivel.
  // Está protegida para que las clases derivadas puedan acceder a esta varia
ble.
  protected TipoNivel tipo;

  // Propiedad pública para acceder a la variable.
  public TipoNivel Tipo {
    get {
      return tipo;
    }
  }

  // Se usa para la inicialización.
  void Start() {
    // Establece la puntuación con la puntuación actual, que será de 0 puntos
 en el inicio de juego.
    hud.ObtenerPuntos(puntosActuales);
  }

  // La actualización se produce una vez por frame.
  void Update() {
  }

  // Función para ganar el juego.
  // Ésta sera virtual para que las clases derivadas puedan anularlo si lo ne
cesitan.
  public virtual void JuegoGanado() {
    // Llama a la función de JuegoAcabado para terminar el juego.
    cuadricula.JuegoAcabado();
    // Indica que el jugador ganó.
    gano = true;
    // Llama a la función para esperar a que la cuadrícula termine de llenars
e.
    StartCoroutine (EsperarCuadriculaLlenado());
  }

  // Función para perder el juego.
  // Ésta sera virtual para que las clases derivadas puedan anularlo si lo ne
cesitan.
  public virtual void JuegoPerdido() {
    // Llama a la función de JuegoAcabado para terminar el juego.
    cuadricula.JuegoAcabado();
    // Indica que el jugador no ganó.
    gano = false;
    // Llama a la función para esperar a que la cuadrícula termine de llenars
e.
    StartCoroutine (EsperarCuadriculaLlenado());
  }

  // Función para cuando el jugador intercambie dos piezas de posición.
  // Ésta sera virtual para que las clases derivadas puedan anularlo si lo ne
cesitan.
  public virtual void EnMovimiento() {
  }

  //Función para restar los puntos obtenidos por el borrado de piezas para el
 reorganizado de las piezas.
```

```
  public virtual void RestarPuntosReorganizado(PiezaJuego pieza) {
    // Decrementa los puntos actuales con los puntos de las piezas.
    puntosActuales -= pieza.puntos;
  }

  // Función para cuando se borra una pieza.
  // Ésta sera virtual para que las clases derivadas puedan anularlo si lo ne
cesitan.
  public virtual void EnBorradoPieza(PiezaJuego pieza) {
    // Incrementa los puntos actuales con los puntos de las piezas.
    puntosActuales += pieza.puntos;
    // Actualiza el HUD cada vez que se actualizan los puntos actuales.
    hud.ObtenerPuntos(puntosActuales);
  }

  // Corrutina que espera hasta que la cuadrícula se termina de llenar.
  protected virtual IEnumerator EsperarCuadriculaLlenado() {
    // Mientras la cuadricula se está llenando espera.
    while (cuadricula.EstaLlenando) {
      yield return 0;
    }
    // Verifica si el jugador ganó.
    if (gano) {
      // Llama a la función de JuegoGanado del HUD.
      hud.JuegoGanado(puntosActuales);
      // Si no es así.
    } else {
      // Llama a la función de JuegoPerdido del HUD.
      hud.JuegoPerdido(puntosActuales);
    }
  }
}
```

NivelTipoMovimiento

Esta clase se utiliza para el seguimiento del estado de la partida en los niveles del juego del modo clásico, establece el tipo de nivel, los puntos, el número de movimientos restantes y la puntuación objetivo en el nivel y marca cuando se da la victoria o la derrota en la partida.

Funciones:

- Start: Se usa para establecer el tipo de nivel, los puntos, el número de movimientos restantes y la puntuación objetivo en el nivel.
- EnMovimiento: Se usa para controlar los movimientos restantes que quedan para que finalice la partida e indicar si se ha ganado o perdido la partida según si se ha llegado a la puntuación objetivo cuando los movimientos se han acabado.

A continuación se muestra el código de la clase, comentado línea a línea para mayor compresión de los detalles del funcionamiento del mismo.

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

public class NivelTipoMovimiento : Nivel {

  // Parámetro para el número de movimientos.
  public int numeroMovimientos;

  // Parámetro para los puntos objetivo.
  public int puntosObjetivo;

  // Parámetro para el número de movimientos utilizados.
  private int movimientosUsados = 0;

  // Se usa para la inicialización.
  void Start() {
    // Establece el tipo de nivel en el tipo "movimiento".
    tipo = TipoNivel.MOVIMIENTO;

    // Establece el tipo de nivel.
    hud.ObtenerTipoNivel(tipo);
    // Establece la puntuación con la puntuación actual, que será de 0 puntos
 en el inicio de juego.
    hud.ObtenerPuntos(puntosActuales);
    // Establece la puntuación objetivo.
    hud.ObtenerObjetivo(puntosObjetivo);
    // Establece el número de movimientos restantes.
    hud.ObtenerRestante(numeroMovimientos);
  }

  // La actualización se produce una vez por frame.
  void Update() {
  }

  // Función para cuando el jugador intercambie dos piezas de posición.
  public override void EnMovimiento() {
    // Incrementa el número de movimientos usados.
```

```
    movimientosUsados++;
    // Actualiza los movimientos restantes en el HUD.
    hud.ObtenerRestante(numeroMovimientos - movimientosUsados);
    // Verifica si se han usado todos los movimientos permitidos.
    if (numeroMovimientos - movimientosUsados == 0) {
      // Verifica si los puntos obtenidos son igual o más que los puntos obje
tivo.
      if (puntosActuales >= puntosObjetivo)  {
        // Llama a "JuegoGanado".
        JuegoGanado();
        // Si no es así.
      } else {
        // Llama a "JuegoPerdido".
        JuegoPerdido();
      }
    }
  }
}
```

NivelTipoTiempo

Esta clase se utiliza para el seguimiento del estado de la partida en los niveles del juego del modo tiempo, establece el tipo de nivel, los puntos, el tiempo de la partida y la puntuación objetivo en el nivel y marca cuando se da la victoria o la derrota en la partida.

Funciones:

- Start: Se usa para establecer el tipo de nivel, los puntos, el tiempo de la partida y la puntuación objetivo en el nivel.
- Update: Se usa para controlar el tiempo que queda para que finalice la partida e indicar si se ha ganado o perdido la partida según si se ha llegado a la puntuación objetivo cuando el tiempo ha acabado.

A continuación se muestra el código de la clase, comentado línea a línea para mayor compresión de los detalles del funcionamiento del mismo.

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

public class NivelTipoTiempo : Nivel {

  // Parámetro para saber cuánto tiempo tiene el jugador.
  public int tiempoSegundos;

  // Parámetro para los puntos objetivo.
  public int puntosObjetivo;

  // Parámetro para cronometrar el tiempo de juego.
  private float cronometro;

  // Parámetro para indicar que el tiempo se ha acabado.
  private bool tiempoAcabado;

  // Se usa para la inicialización.
  void Start() {
    // Establece el tipo de nivel en el tipo "tiempo".
    tipo = TipoNivel.TIEMPO;
    // Establece el tipo de nivel.
    hud.ObtenerTipoNivel(tipo);
    // Establece la puntuación con la puntuación actual, que será de 0 puntos
 en el inicio de juego.
    hud.ObtenerPuntos(puntosActuales);
    // Establece la puntuación objetivo.
    hud.ObtenerObjetivo(puntosObjetivo);
    // Establece el tiempo restantes. Se usa un formato de minutos y segundos
    hud.ObtenerRestante(string.Format("{0}:{1:00}", tiempoSegundos / 60, tiem
poSegundos % 60));
    }

  // La actualización se produce una vez por frame.
  void Update() {
    // Si el tiempo no se ha acabado.
    if (!tiempoAcabado) {
```

```
      // Actualiza el tiempo.
      cronometro += Time.deltaTime;
      // Actualiza el tiempo en el HUD. Marca el tiempo que queda para finali
zar el juego.
      // Mathf.Max se utiliza para evitar valores negativos en el tiempo al f
inalizar el juego.
      hud.ObtenerRestante(string.Format("{0}:{1:00}", (int)Mathf.Max((tiempoS
egundos - cronometro) / 60, 0), (int)Mathf.Max((tiempoSegundos -
 cronometro) % 60, 0)));
      // Verifica si se ha acabado el tiempo.
      if (tiempoSegundos - cronometro <= 0) {
        // Verifica si se ha llegado a la puntuación objetivo.
        if (puntosActuales >= puntosObjetivo) {
          // Llama a "JuegoGanado".
          JuegoGanado ();
          // Si no es así.
        } else {
          // Llama a "JuegoPerdido".
          JuegoPerdido ();
        }
        // Indica que el tiempo se ha acabado.
        tiempoAcabado = true;
      }
    }
  }
}
```

NivelTipoVirus

Esta clase se utiliza para el seguimiento del estado de la partida en los niveles del juego del modo virus, establece el tipo de nivel, los puntos, el número de enemigos a eliminar y el número de movimientos restantes en el nivel y marca cuando se da la victoria o la derrota en la partida.

Funciones:

- Start: Se usa para establecer el tipo de nivel, los puntos, el número de enemigos a eliminar y el número de movimiento restantes en el nivel.
- EnMovimiento: Se usa para controlar la cantidad movimientos utilizados e indicar si se nos hemos quedado sin movimientos y, por tanto, perdido la partida.
- EnBorradoPieza: Se usa para indicar cuantas piezas de tipo virus quedan, obtener puntos adicionales por los movimientos no utilizados en acabar con todas las piezas de tipo virus e indicar que se ha ganado la partida.

A continuación se muestra el código de la clase, comentado línea a línea para mayor compresión de los detalles del funcionamiento del mismo.

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

public class NivelTipoVirus : Nivel {

  // Parámetro para el número de movimientos.
  public int numeroMovimientos;

  // Parámetro para saber que piezas son de tipo enemigo.
  public Cuadricula.TipoPieza[] tipoEnemigo;

  // Parámetro para el número de movimientos utilizados.
  private int movimientosUsados = 0;

  // Parámetro para saber cuantas piezas de tipo enemigo quedan en la cuadric
ula.
  private int enemigosRestantes;

  // Se usa para la inicialización.
  void Start() {
    // Establece el tipo de nivel en el tipo "virus".
    tipo = TipoNivel.VIRUS;
    // Recorre la lista de las piezas de tipo "virus".
    for (int i = 0; i < tipoEnemigo.Length; i++) {
      // Incrementa la cantidad de piezas de tipo "virus" conforme la cantida
d de piezas de esté tipo que hay en la cuadrícula.
      enemigosRestantes += cuadricula.ObtenerPiezasTipo(tipoEnemigo[i]).Count
;
    }
    // Establece el tipo de nivel.
    hud.ObtenerTipoNivel(tipo);
    // Establece la puntuación con la puntuación actual, que será de 0 puntos
 en el inicio de juego.
    hud.ObtenerPuntos(puntosActuales);
```

```
    // Establece el número de enemigos a eliminar, que será el número de piez
as de tipo enemigo que hay en la pantalla.
    hud.ObtenerObjetivo(enemigosRestantes);
    // Establece el número de movimientos restantes.
    hud.ObtenerRestante(numeroMovimientos);
  }

  // La actualización se produce una vez por frame.
  void Update() {
  }

  // Función para cuando el jugador intercambie dos piezas de posición.
  public override void EnMovimiento() {
    // Incrementa el número de movimientos usados.
    movimientosUsados++;
    // Actualiza los movimientos restantes en el HUD.
    hud.ObtenerRestante(numeroMovimientos - movimientosUsados);
    // Verifica si se han usado todos los movimientos permitidos y si no se h
an eliminado todas las piezas de tipo enemigo.
    if (numeroMovimientos -
 movimientosUsados == 0 && enemigosRestantes > 0) {
      // Pone los puntos a 0.
      puntosActuales = 0;
      // Establece la puntuación con la puntuación actual.
      hud.ObtenerPuntos(puntosActuales);
      // Llama a "JuegoPerdido".
      JuegoPerdido();
    }
  }

  // Función para cuando se borra una pieza.
  public override void EnBorradoPieza(PiezaJuego pieza) {
    // Permite seguir llamando a la función base para que actualice los punto
s.
    base.EnBorradoPieza(pieza);
    // Recorre la lista de las piezas de tipo "virus".
    for (int i = 0; i < tipoEnemigo.Length; i++) {
      // Verifica si la pieza borrada concuerda con alguna de la lista de las
 piezas de tipo "virus".
      if (tipoEnemigo [i] == pieza.Tipo) {
        // Reduce el número de piezas de tipo enemigo que quedan.
        enemigosRestantes--;
        // Actualiza los enemigos restantes en el HUD.
        hud.ObtenerObjetivo(enemigosRestantes);
        // Verifica si todas las piezas de tipo enemigo han sido eliminadas.
        if (enemigosRestantes == 0) {
          // Da cierta cantidad de puntos por cada movimientos que haya sobra
do.
          puntosActuales += 200 * (numeroMovimientos - movimientosUsados);
          // Establece la puntuación con la puntuación actual.
          hud.ObtenerPuntos(puntosActuales);
          // Llama a "JuegoGanado".
          JuegoGanado();
        }
      }
    }
  }
}
```

HUD

Esta clase se utiliza para que se muestre y actualice la interfaz de usuario con unos datos u otros, durante la partida, dependiendo del modo de juego en el que nos encontremos jugando.

Funciones:

- Start: Se usa para que al iniciar una partida no se muestre ninguna copa.
- ObtenerPuntos: Se usa para mostrar los puntos actuales que lleva el jugador y para que se muestren las diferentes copas obtenidas según esta puntuación.
- ObtenerObjetivo: Se usa para mostrar la puntuación objetivo.
- ObtenerRestante: Se usa para mostrar los movimientos restantes o el tiempo restante para finalizar la partida.
- ObtenerTipoNivel: Se usa para mostrar unos textos u otros en la interfaz según el modo de juego en el que estemos.
- JuegoGanado: Se usa para llamar a la función que muestra la interfaz de victoria cuando acaba la partida y para almacenar la puntuación más alta.
- JuegoPerdido: Se usa para llamar a la función que muestra la interfaz de derrota cuando acaba la partida.

A continuación se muestra el código de la clase, comentado línea a línea para mayor compresión de los detalles del funcionamiento del mismo.

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

public class HUD : MonoBehaviour {

  // Parámetro para el nivel.
  public Nivel nivel;

  // Parámetro para la pantalla de juego acabado.
  public JuegoAcabado juegoAcabado;

  // Parámetros para los elementos de la UI.
  public UnityEngine.UI.Text puntosObjetivoTexto;
  public UnityEngine.UI.Text puntosObjetivoSubtexto;
  public UnityEngine.UI.Text restanteTexto;
  public UnityEngine.UI.Text restanteSubtexto;
  public UnityEngine.UI.Text puntosTexto;
  public UnityEngine.UI.Text puntosSubtexto;
  public UnityEngine.UI.Image[] copa;

  // Parámetro para el índice de las imágenes de las copas.
  private int copaIndice = 0;

  // Se usa para la inicialización.
  void Start() {
    // Recorre todas las imágenes de las copas.
    for (int i = 0; i < copa.Length; i++) {
      // Verifica que el índice sea igual al actual para comenzar el juego co
n la imagen sin copas.
```

```
      if (i == copaIndice) {
        // Habilita la imagen.
        copa[i].enabled = true;
      } else {
        // Deshabilita la imagen.
        copa[i].enabled = false;
      }
    }
  }

  // La actualización se produce una vez por frame.
  void Update() {
  }

  // Función para establecer la puntuación del jugador.
  public void ObtenerPuntos(int puntos) {
    // Establece el texto con el número de la puntuación actual del jugador.
    // El parámetro de texto espera una cadena y no un entero.
    puntosTexto.text = puntos.ToString();
    // Se muestra de entrada la imagen sin copas.
    int visibleCopa = 0;
    // Verifica si los puntos están entre la copa de bronce y la de plata.
    if (puntos >= nivel.puntosCopaBronce && puntos < nivel.puntosCopaPlata) {
      // Se muestra la imagen de la copa de bronce.
      visibleCopa = 1;
    // Si no es así, verifica si los puntos están entre la copa de plata y la
 de oro.
    } else if (puntos >= nivel.puntosCopaPlata && puntos < nivel.puntosCopaOr
o) {
      // Se muestra la imagen de la copa de plata.
      visibleCopa = 2;
    // Si no es así, verifica si los puntos son igual o más de los necesarios
 para la copa de oro.
    } else if (puntos >= nivel.puntosCopaOro) {
      // Se muestra la imagen de la copa de oro.
      visibleCopa = 3;
    }

    // Recorre todas las imágenes de las copas.
    for (int i = 0; i < copa.Length; i++) {
      // Verifica que el índice coincide con la copa que toca mostrar.
      if (i == visibleCopa) {
        // Habilita la imagen.
        copa[i].enabled = true;
      } else {
        // Deshabilita la imagen.
        copa[i].enabled = false;
      }
    }
    // Almacena el índice de la copa visible.
    copaIndice = visibleCopa;
  }

  // Función para actualizar el texto de la puntuación objetivo.
  public void ObtenerObjetivo(int objetivo) {
    // Establece el texto con el número de la puntuación objetivo.
    puntosObjetivoTexto.text = objetivo.ToString();
  }
```

```
  // Función para los movimientos restantes para el tipo de nivel "movimiento
" y "virus".
  public void ObtenerRestante(int restante) {
    // Establece el texto.
    restanteTexto.text = restante.ToString();
  }

  // Función para el tiempo restante para el tipo de nivel "tiempo".
  public void ObtenerRestante(string restante) {
    // Establece el texto.
    restanteTexto.text = restante;
  }

  // Función para los subtextos según el tipo de nivel.
  public void ObtenerTipoNivel(Nivel.TipoNivel tipo) {
    // Verifica si el tipo de nivel es "movimiento".
    if (tipo == Nivel.TipoNivel.MOVIMIENTO) {
      // Establece los subtextos.
      restanteSubtexto.text = "Movimientos restantes";
      puntosObjetivoSubtexto.text = "Puntuación objetivo";
      puntosSubtexto.text = "Puntuación actual";
    // si no es así, verifica si el tipo de nivel es "virus".
    } else if (tipo == Nivel.TipoNivel.VIRUS) {
      // Establece los subtextos.
      restanteSubtexto.text = "Movimientos restantes";
      puntosObjetivoSubtexto.text = "Enemigos restantes";
      puntosSubtexto.text = "Puntuación actual";
    // si no es así, verifica si el tipo de nivel es "tiempo".
    } else if (tipo == Nivel.TipoNivel.TIEMPO) {
      // Establece los subtextos.
      restanteSubtexto.text = "Tiempo restante";
      puntosObjetivoSubtexto.text = "Puntuación objetivo";
      puntosSubtexto.text = "Puntuación actual";
    }
  }

  // Función para el juego ganado.
  public void JuegoGanado(int puntos) {
    // Muestra la pantalla de juego ganado.
    juegoAcabado.MostrarGanado(puntos);
    // Verifica si la puntuación almacenada es menor que la puntuación nueva.
 El valor predeterminado sera 0 en caso de que el nivel aún no tenga puntuaci
ón.
    if (copaIndice > PlayerPrefs.GetInt(UnityEngine.SceneManagement.SceneMana
ger.GetActiveScene().name, 0)) {
      // Almacena la nueva puntuación.
      PlayerPrefs.SetInt(UnityEngine.SceneManagement.SceneManager.GetActiveSc
ene().name, copaIndice);
    }
  }

  // Función para el juego perdido.
  public void JuegoPerdido(int puntos) {
    // Muestra la pantalla de juego perdido.
    juegoAcabado.MostrarPerdido(puntos);
  }
}
```

JuegoAcabado

Esta clase se utiliza para que una vez se termina una partida se muestre una interfaz de victoria o derrota, según toque, y se pueda seleccionar el volver a jugar el nivel o volver al menú de selección de nivel.

Funciones:

- Start: Se usa para desactivar la interfaz que aparece cuando se acaba una partida en un nivel del juego. Hace que no se muestre durante el transcurso de la partida.
- MostrarPerdido: Se usa para mostrar la interfaz de derrota cuando acaba la partida.
- MostrarGanado: Se usa para mostrar la interfaz de victoria cuando acaba la partida.
- EnClicarRepetir: Se usa para volver a cargar el nivel actual.
- EnClicarTerminar: Se usa para volver al menú de selección de nivel.

A continuación se muestra el código de la clase, comentado línea a línea para mayor compresión de los detalles del funcionamiento del mismo.

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

public class JuegoAcabado : MonoBehaviour {

  // Parámetros para los elementos de la UI.
  public GameObject pantalla;
  public UnityEngine.UI.Text perdisteTexto;
  public UnityEngine.UI.Text ganasteTexto;
  public UnityEngine.UI.Text puntosTexto;
  public UnityEngine.UI.Text puntosFinalSubtexto;

  // Se usa para la inicialización.
  void Start() {
    // Desactiva el objeto de "Pantalla".
    pantalla.SetActive (false);
  }

  // La actualización se produce una vez por frame.
  void Update() {
  }

  // Función para mostrar la pantalla de derrota.
  public void MostrarPerdido(int puntos) {
    // Activa el objeto de "Pantalla".
    pantalla.SetActive (true);
    // Desactiva el texto de victoria.
    ganasteTexto.enabled = false;
    // Establece los puntos del jugador.
    puntosTexto.text = puntos.ToString();
    // Establece el subtexto.
    puntosFinalSubtexto.text = "Puntuación final";
    // Obtiene el componente animador.
    Animator animador = GetComponent<Animator> ();
    // Verifica si hay un componente animador.
    if (animador) {
      // Reproduce la animación.
```

```
      animador.Play ("JuegoAcabado");
    }
  }

  // Función para mostrar la pantalla de victoria.
  public void MostrarGanado(int puntos) {
    // Activa el objeto de "Pantalla".
    pantalla.SetActive (true);
    // Desactiva el texto de derrota.
    perdisteTexto.enabled = false;
    // Establece los puntos del jugador.
    puntosTexto.text = puntos.ToString();
    // Establece el subtexto.
    puntosFinalSubtexto.text = "Puntuación final";
    // Obtiene el componente animador.
    Animator animador = GetComponent<Animator> ();
    // Verifica si hay un componente animador.
    if (animador) {
        // Reproduce la animación.
        animador.Play ("JuegoAcabado");
    }
  }

  // Función para el botón repetir.
  public void EnClicarRepetir() {
    // Carga el nivel actual.
    UnityEngine.SceneManagement.SceneManager.LoadScene(UnityEngine.SceneManag
ement.SceneManager.GetActiveScene ().name);
  }

  // Función para el botón terminar.
  public void EnClicarTerminar() {
    // Verifica si la escena actual es uno de los niveles del modo clásico.
    if (UnityEngine.SceneManagement.SceneManager.GetActiveScene ().name.Conta
ins("NivelM")) {
      // Carga la escena de selección de nivel del modo clásico.
      UnityEngine.SceneManagement.SceneManager.LoadScene("MenuNivelM");
    // Si no es así. Verifica si la escena actual es uno de los niveles del m
odo tiempo.
    } else if (UnityEngine.SceneManagement.SceneManager.GetActiveScene ().nam
e.Contains("NivelT")) {
      // Carga la escena de selección de nivel del modo tiempo.
      UnityEngine.SceneManagement.SceneManager.LoadScene("MenuNivelT");
    // Si no es así. Verifica si la escena actual es uno de los niveles del m
odo virus.
    } else if (UnityEngine.SceneManagement.SceneManager.GetActiveScene ().nam
e.Contains("NivelV")) {
      // Carga la escena de selección de nivel del modo virus.
      UnityEngine.SceneManagement.SceneManager.LoadScene("MenuNivelV");
    }
  }
}
```

SeleccionEscena

Esta clase se utiliza para cargar los niveles desde los menús de selección de nivel, para mostrar las medallas logradas en cada nivel en el menú de selección y para habilitar nuevos niveles en superar los anteriores.

Funciones:

- Start: Se usa para desactivar todos los botones menos el del primer nivel de cada modo de juego e irlos activando conforme se superan los niveles anteriores a la vez que se indican las medallas logradas en los niveles superados.
- PresionarBoton: Se usa para cargar un nivel del juego en presionar un botón.

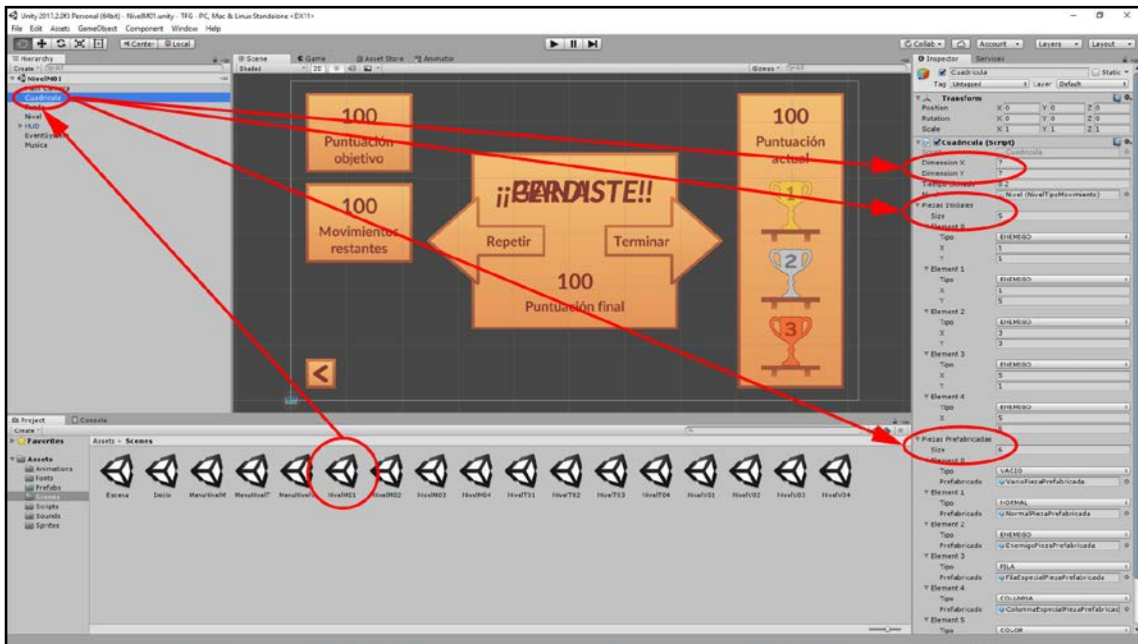
A continuación se muestra el código de la clase, comentado línea a línea para mayor compresión de los detalles del funcionamiento del mismo.

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

public class SeleccionEscena : MonoBehaviour {

  // Estructura para saber qué nivel está asociado a cada botón.
  [System.Serializable]
  public struct BotonPlayerPrefs {
    // Parámetro para el botón.
    public GameObject boton;
    // Parámetro para el nivel.
    public string nivel;
  }

  // Parámetro para una matriz de estas estructuras.
  public BotonPlayerPrefs[] botones;

  // Se usa para la inicialización.
  void Start () {
    // Recorre todos los botones menos el primero.
    for (int i = 1; i < botones.Length; i++) {
      // Desactiva todos los botones menos el primero.
      botones [i].boton.gameObject.SetActive(false);
    }
    // Recorre todos los botones.
    for (int i = 0; i < botones.Length; i++) {
      // Obtiene la puntuación de cada nivel.
      int puntos = PlayerPrefs.GetInt(botones[i].nivel, 0);
      // Recorre las tres medallas.
      for (int medallaIndice = 1; medallaIndice <= 3; medallaIndice++) {
        // Obtiene las tres medallas.
        Transform medalla = botones [i].boton.transform.Find ("Medalla" + med
allaIndice);
        // Verifica si el indice de la medalla es menor o igual que la puntua
ción obtenida.
        if (medallaIndice <= puntos) {
          // Habilita la medalla para que se vea.
          medalla.gameObject.SetActive(true);
          // Verifica que no sea el botón del último nivel.
          if (i < 7) {
```

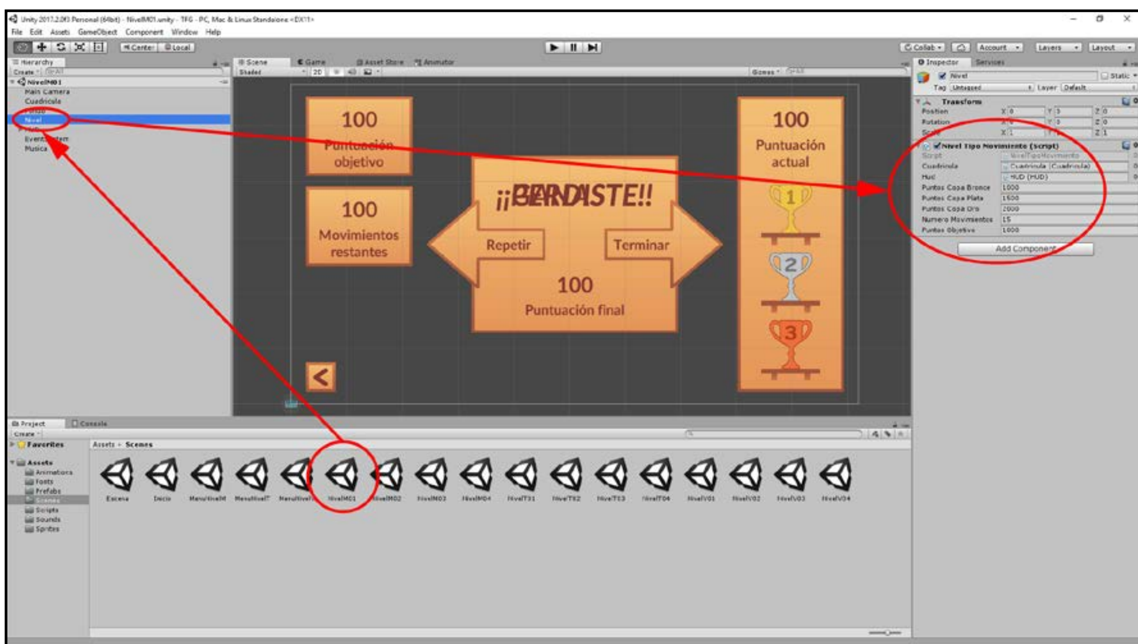
```
            // Habilita el botón del siguiente nivel.
            botones [i+1].boton.gameObject.SetActive(true);
          }
        // Si no es así.
        } else {
          // Deshabilita la medalla para que no se vea.
          medalla.gameObject.SetActive(false);
        }
      }
    }
  }

  // La actualización se produce una vez por frame.
  void Update () {
  }

  // Función para cargar una escena al presionar un botón.
  public void PresionarBoton(string nivelNombre) {
    // Carga una escena.
    UnityEngine.SceneManagement.SceneManager.LoadScene(nivelNombre);
  }
}
```

Diseño de niveles

Para la creación de los niveles del juego he tenido que planificar como quería que fuera cada nivel en cuanto a la organización de la cuadrícula y luego configurarlo mediante el script de la cuadrícula que está como componente del objeto cuadrícula en cada nivel de juego, poniendo las dimensiones de la cuadrícula, los tipos de piezas que tendrá el nivel y donde habrá determinadas piezas al empezar la partida.



En cuanto a los parámetros del nivel en cuanto a movimientos o tiempo disponible para completarlo y la puntuación objetivo y de los rangos de bronce, plata y oro, éstos se han configurado mediante el script de nivel del modo que toque, que está como componente del objeto nivel en cada nivel de juego.



En las siguientes imágenes se pueden observar cada uno de los diseños de los niveles iníciales del modo clásico, acompañadas por los parámetros de puntuación objetivo, puntuación de rango bronce, plata y oro y número de movimientos permitidos.

Diseño del nivel 1 del modo clásico:

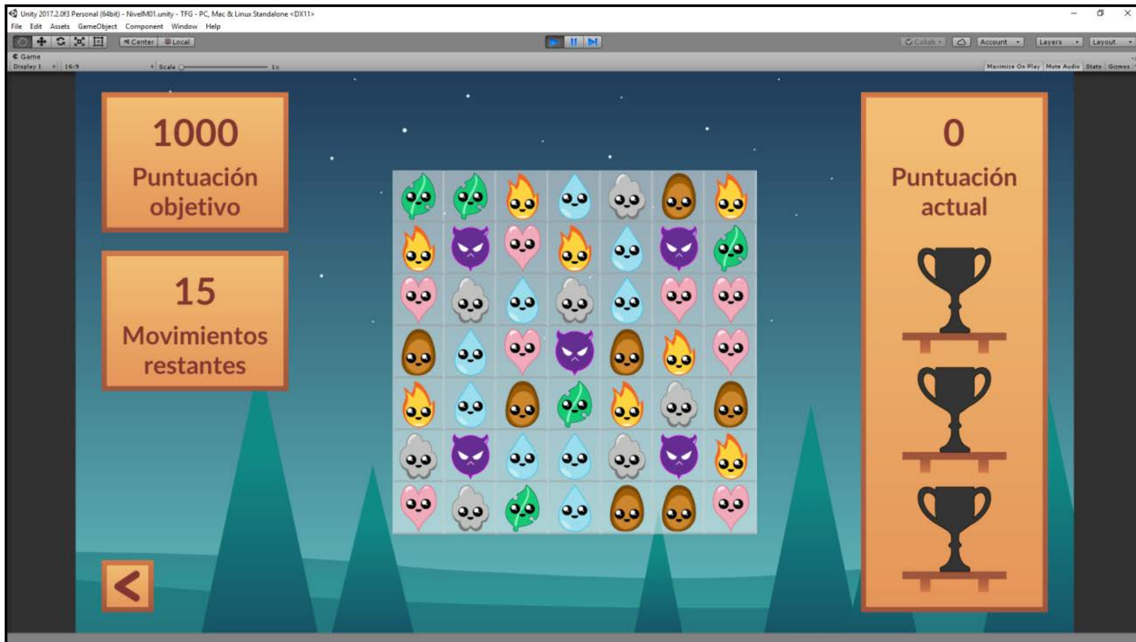


Tabla: 7x7   Objetivo/Bronce: 1000   Plata: 1500   Oro: 2000   Movimientos: 15

Diseño del nivel 2 del modo clásico:

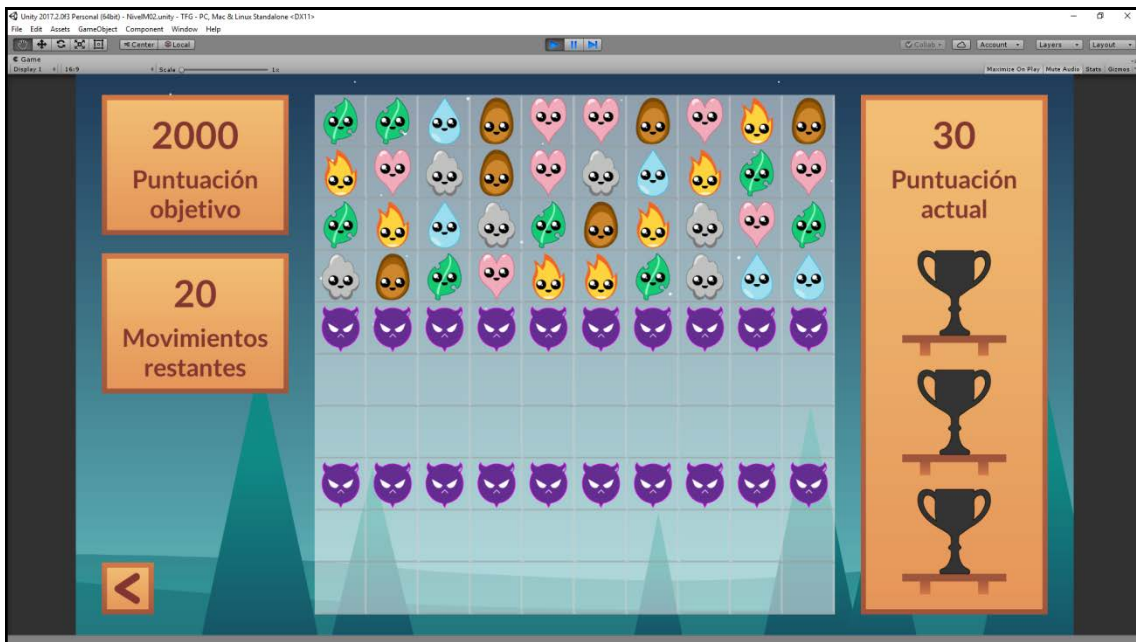


Tabla: 10x10   Objetivo/Bronce: 2000   Plata: 3500   Oro: 4000   Movimientos: 20

Diseño del nivel 3 del modo clásico:

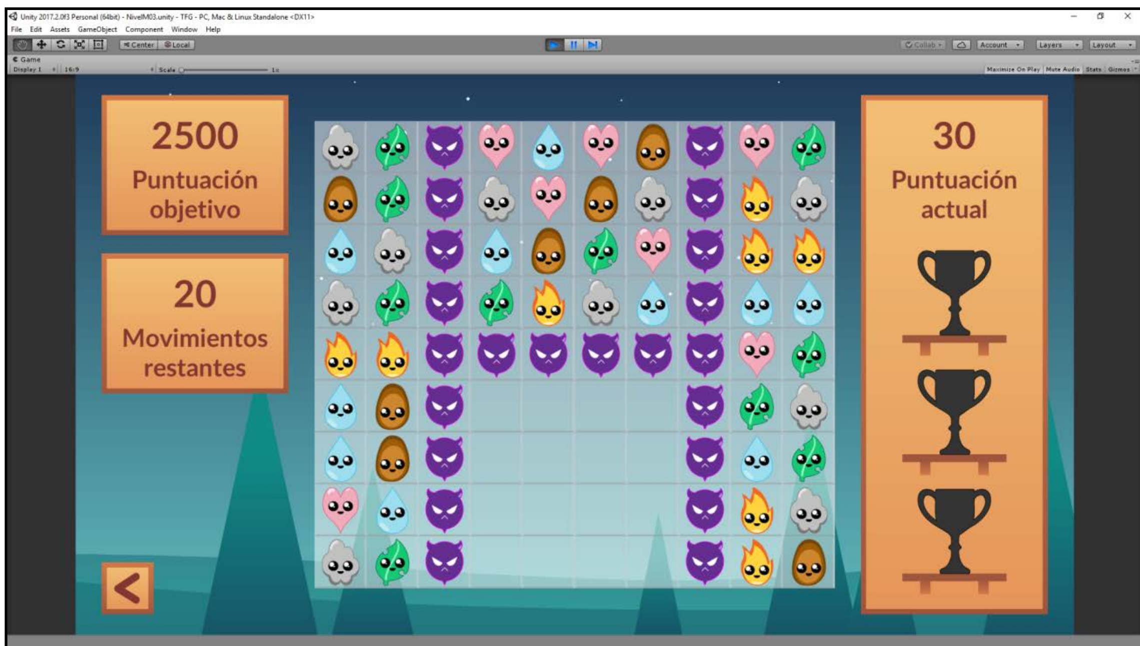


Tabla: 10x9   Objetivo/Bronce: 2500   Plata: 3500   Oro: 4000   Movimientos: 20

Diseño del nivel 4 del modo clásico:

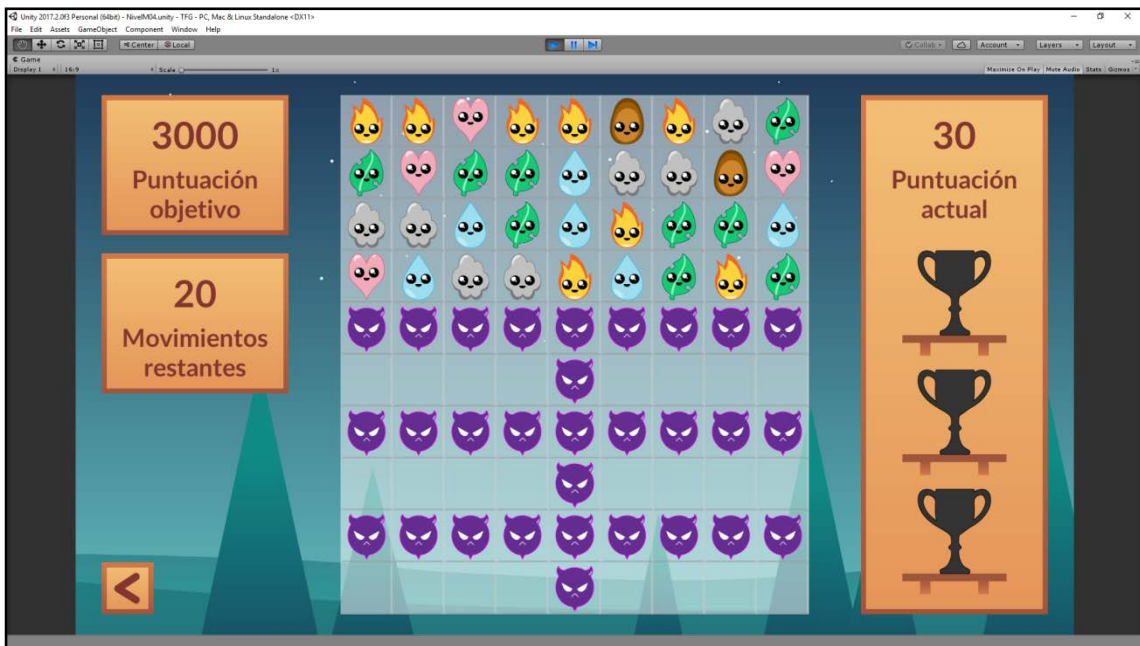


Tabla: 9x10   Objetivo/Bronce: 3000   Plata: 3500   Oro: 4000   Movimientos: 20

En las siguientes imágenes se pueden observar cada uno de los diseños de los niveles iníciales del modo tiempo, acompañadas por los parámetros de puntuación objetivo, puntuación de rango bronce, plata y oro y tiempo permitido.

Diseño del nivel 1 del modo tiempo:

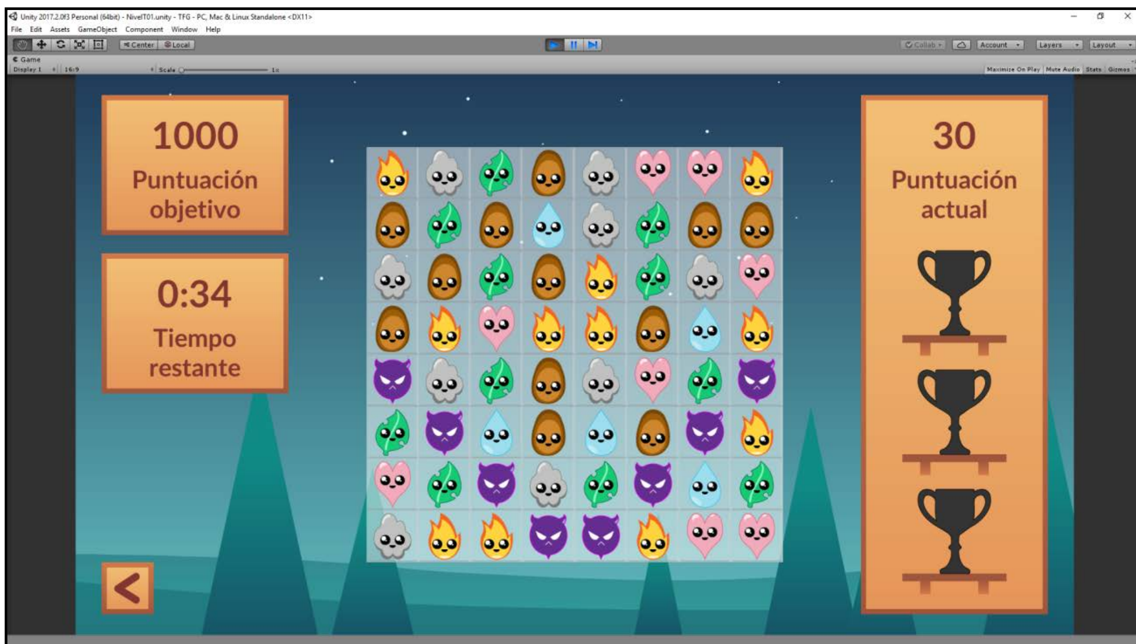


Tabla: 8x8 Objetivo/Bronce: 1000 Plata: 1500 Oro: 2000 Tiempo: 40s

Diseño del nivel 2 del modo tiempo:

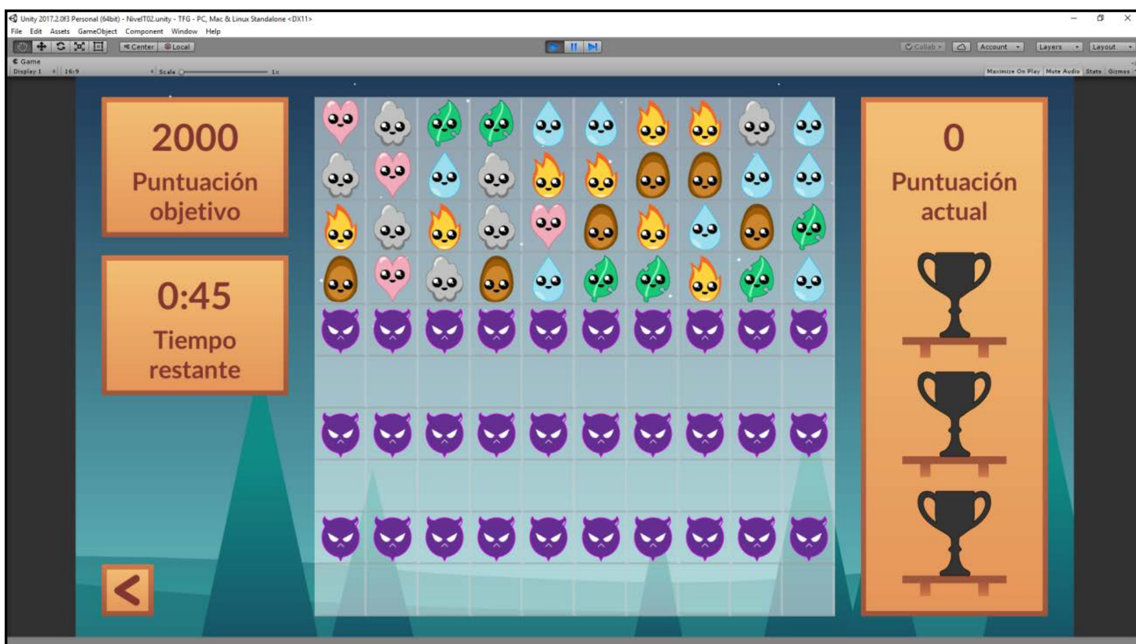


Tabla: 10x10 Objetivo/Bronce: 2000 Plata: 3000 Oro: 4000 Tiempo: 50s

Diseño del nivel 3 del modo tiempo:

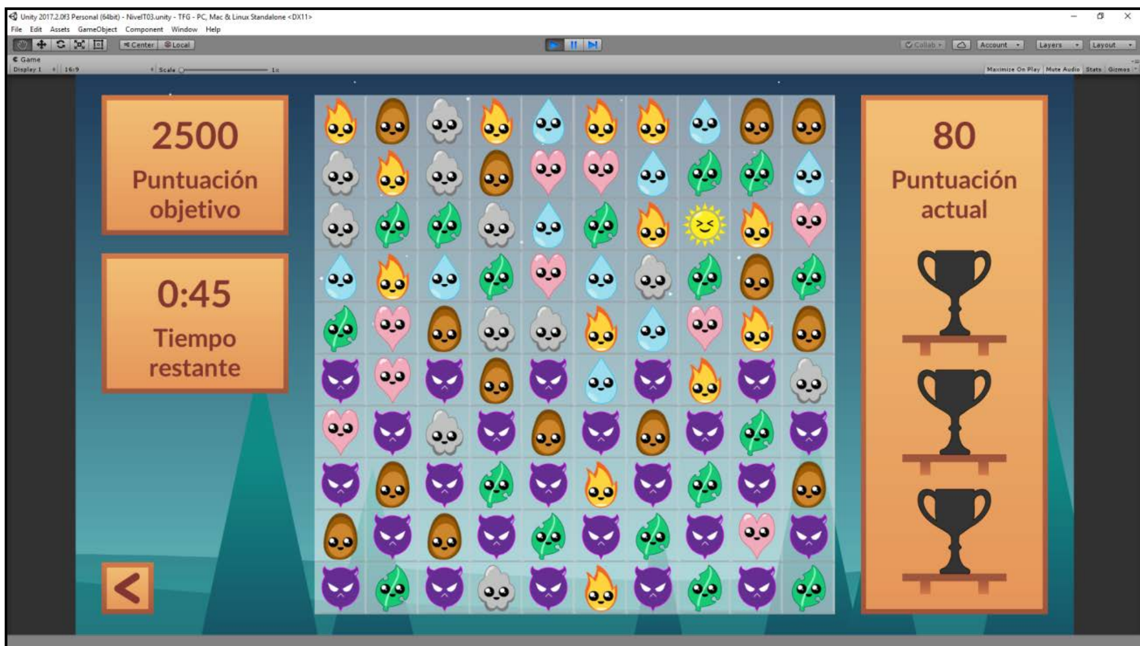


Tabla: 10x10 Objetivo/Bronce: 2500 Plata: 4000 Oro: 5000 Tiempo: 50s

Diseño del nivel 4 del modo tiempo:

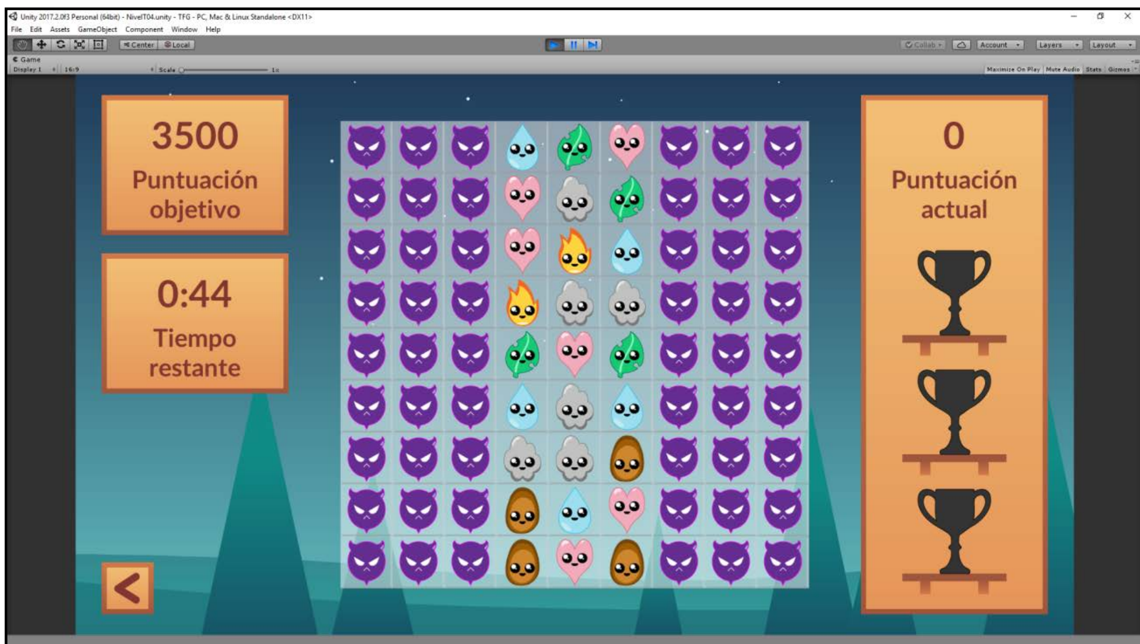


Tabla: 9x9 Objetivo/Bronce: 3500 Plata: 4000 Oro: 4500 Tiempo: 50s

En las siguientes imágenes se pueden observar cada uno de los diseños de los niveles iníciales del modo virus, acompañadas por los parámetros de enemigos a eliminar, puntuación de rango bronce, plata y oro y número de movimientos permitidos.

Diseño del nivel 1 del modo virus:

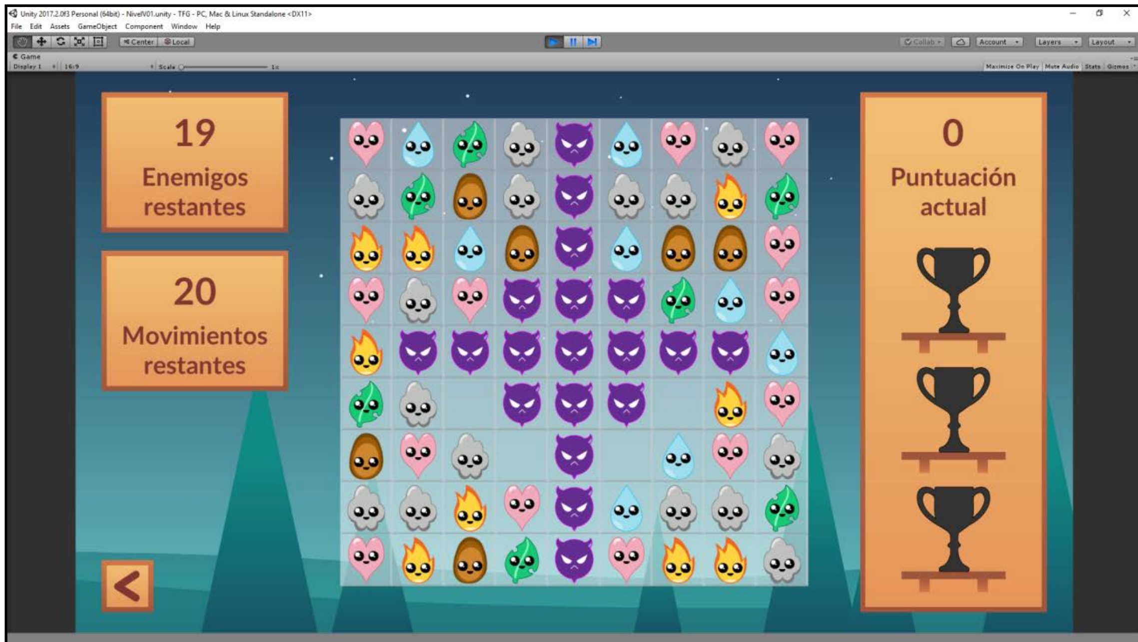


Tabla: 9x9 Bronce: 950 Plata: 3000 Oro: 4000 Movimientos: 20 Enemigos: 19

Diseño del nivel 2 del modo virus:

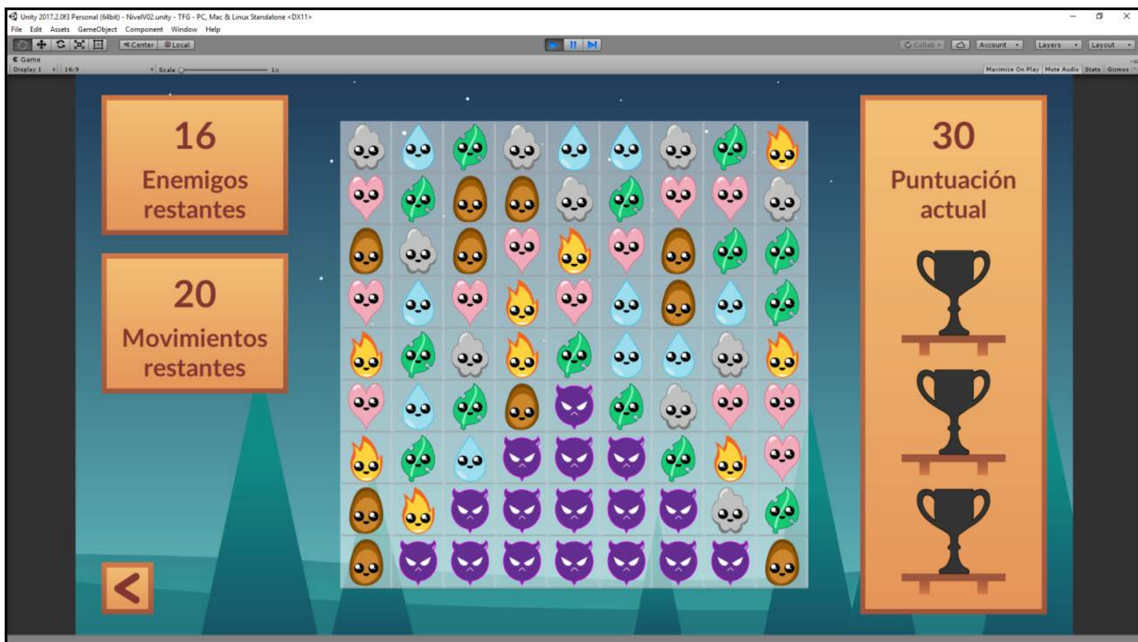


Tabla: 9x9 Bronce: 800 Plata: 3500 Oro: 4000 Movimientos: 20 Enemigos: 16

Diseño del nivel 3 del modo virus:

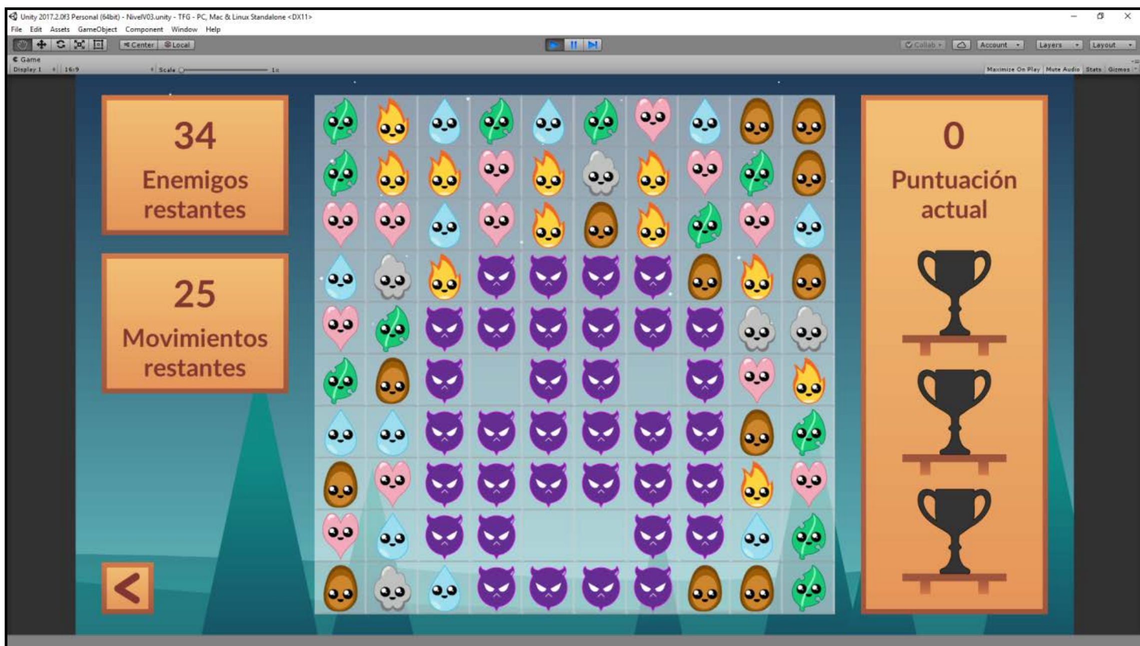


Tabla: 10x10 Bronce: 1700 Plata: 5000 Oro: 6000 Movimientos: 25 Enemigos: 34

Diseño del nivel 4 del modo virus:

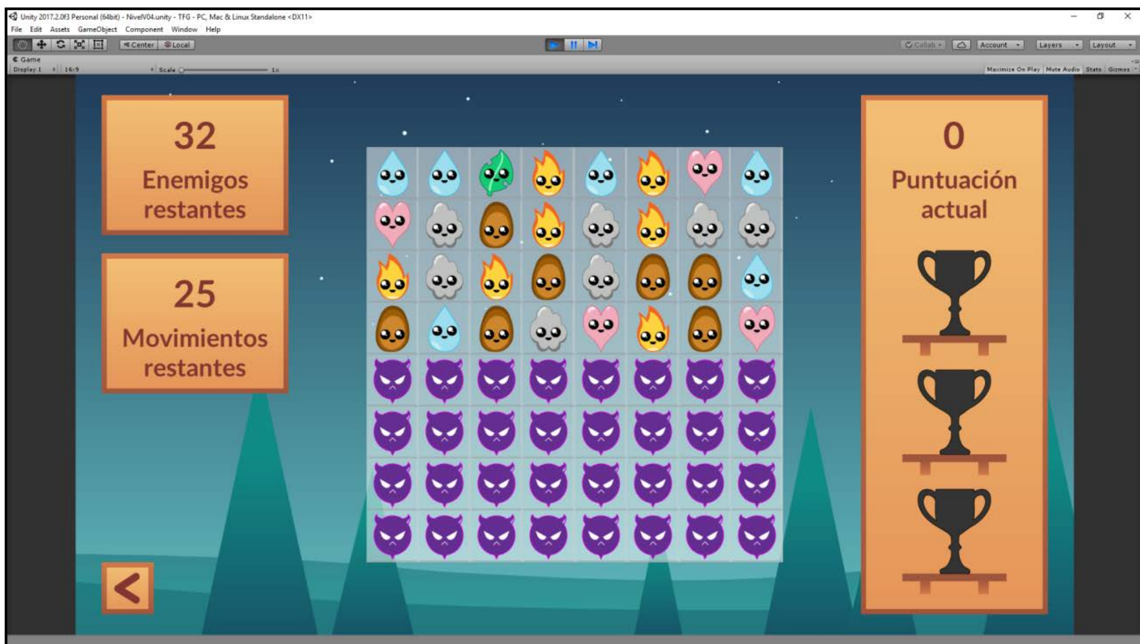


Tabla: 8x8 Bronce: 1600 Plata: 4500 Oro: 5000 Movimientos: 25 Enemigos: 32

Testeo de niveles

En este apartado he recopilado a base de prueba y ensayo una serie de datos sobre cada uno de los niveles iníciales creados para cada uno de los tres modos de juego de los que dispone el juego.

Para ello, he testeado cada uno de los niveles del juego decidiendo primero lo que va a durar el nivel, fijando la cantidad de movimientos que se podrán hacer o el tiempo que durara la partida, y después jugando una veintena de veces cada nivel para observar cuántos puntos se pueden obtener, más o menos, con esa cantidad de movimientos o tiempo.

Una vez recopilada la información de todas las partidas ya he podido deliberar donde fijar la puntuación mínima o objetivo para los modos clásico y tiempo que será a su vez la puntuación del rango bronce.

En el caso del modo virus, funcionaria diferente ya que he hecho que si no se eliminan todos los enemigos la puntuación pase a ser 0. De forma interna, ya que no se le muestra al jugador, he fijado la puntuación para obtener el rango bronce con la puntuación que se obtiene por eliminar a todos los enemigos de manera que así también me aseguro que si se eliminan a todos los enemigos también se obtenga el rango bronce.

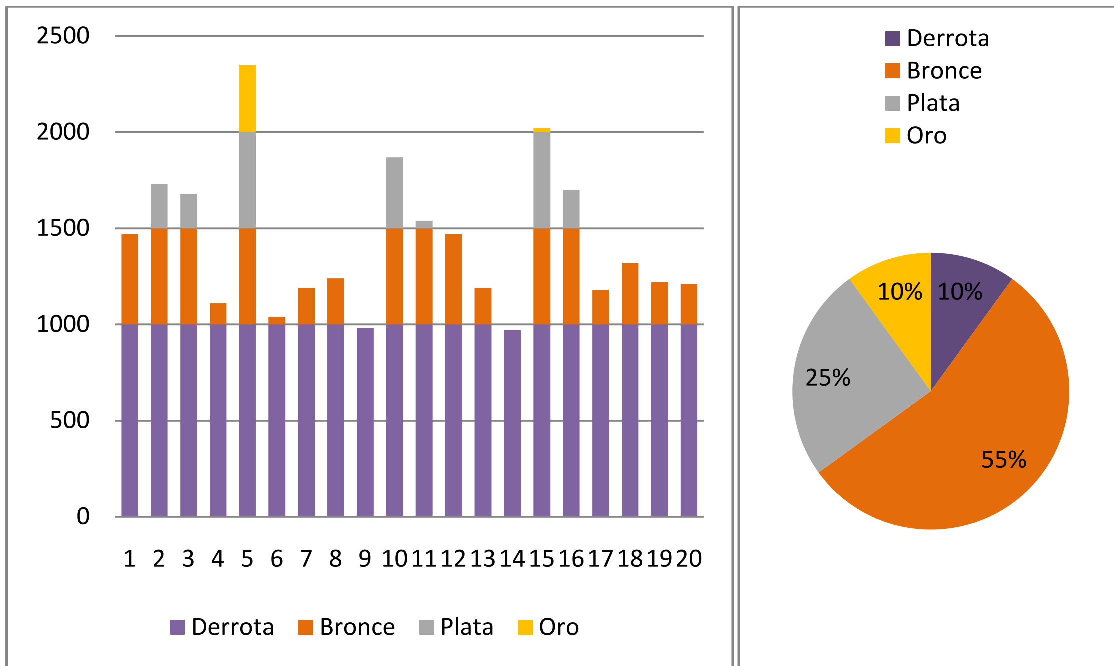
En cuanto a la puntuación para el rango plata y oro ya ha sido mirando las partidas en su global y decidiendo que porcentaje de partidas quería en cada rango mirando así de poner más fácil o difícil su obtención según el nivel del juego. A su vez, aunque las puntuaciones necesarias para el rango plata y oro están ocultas de cara al jugador, también he mirando que el número de dichas puntuaciones será un numero redondo, como 3000, 3500, etc.

También decir que hacer este testeo a los niveles del juego me ha permitido ver que todo está correcto a nivel de funcionamiento. Es decir que la piezas se desplacen e intercambien correctamente, que las piezas especiales borren lo que tienen que borrar, que se reorganice correctamente el tablero cuando no hay mas movimientos posibles, que las puntuaciones se sumen de forma correcta, que los indicadores del HUD muestren lo que tienen que mostrar, etc.

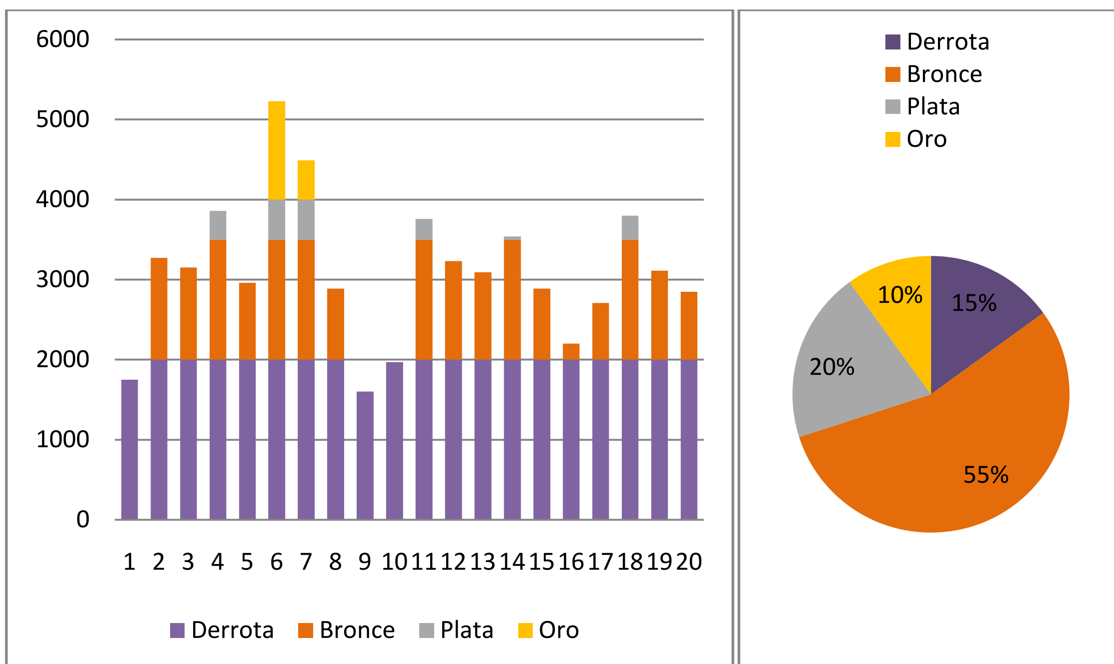
Por último, a continuación, se pueden ver las gráficas con los datos recopilados de cada uno de los niveles iníciales del juego, donde ya se puede ver la información obtenida de forma más detalla y visual de lo que comentaba antes.

En los siguiente gráficos se pueden observar las puntuaciones obtenidas en veinte partidas de cada uno de los niveles iníciales del modo clásico, así como los porcentajes de derrota y de obtención de una puntuación de rango bronce, plata y oro.

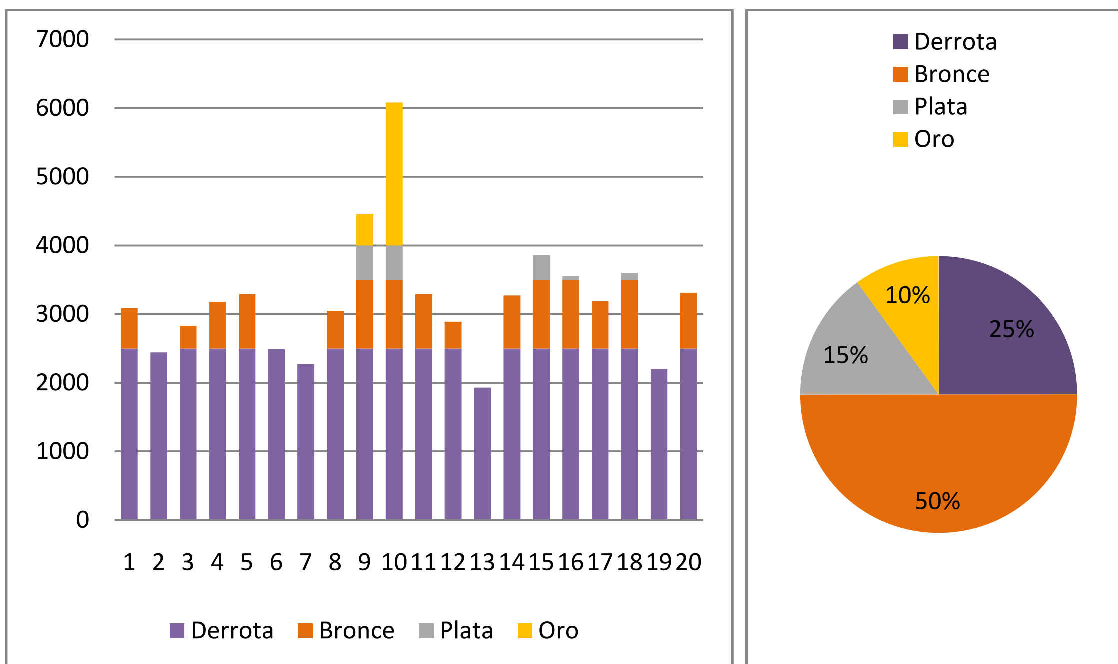
Gráficos del nivel 1 del modo clásico:



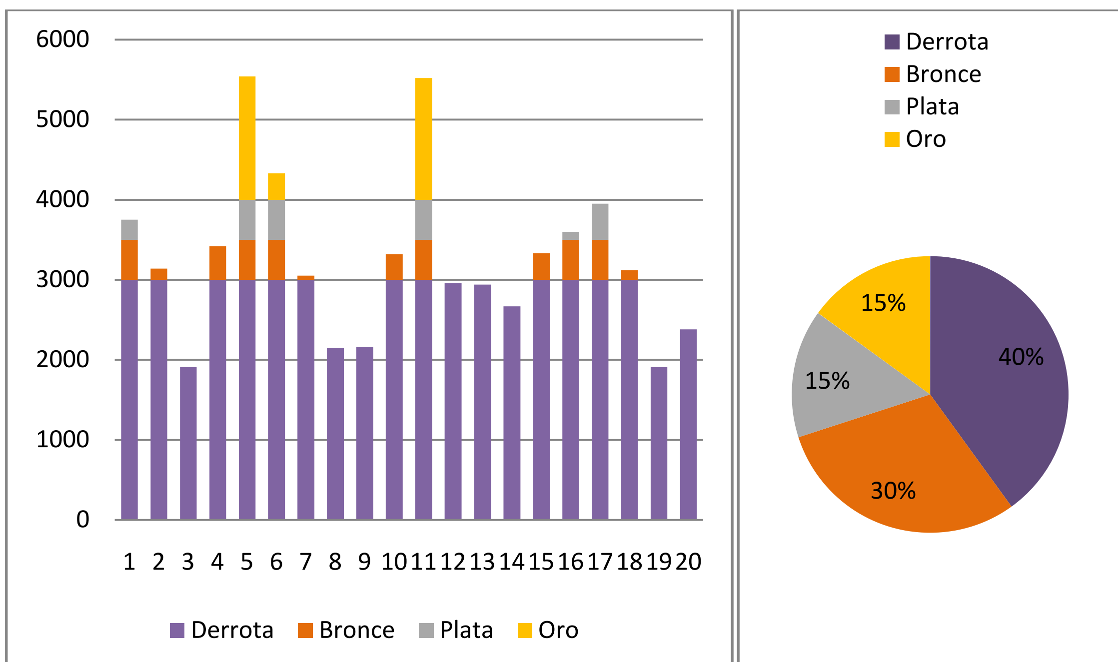
Gráficos del nivel 2 del modo clásico:



Gráficos del nivel 3 del modo clásico:



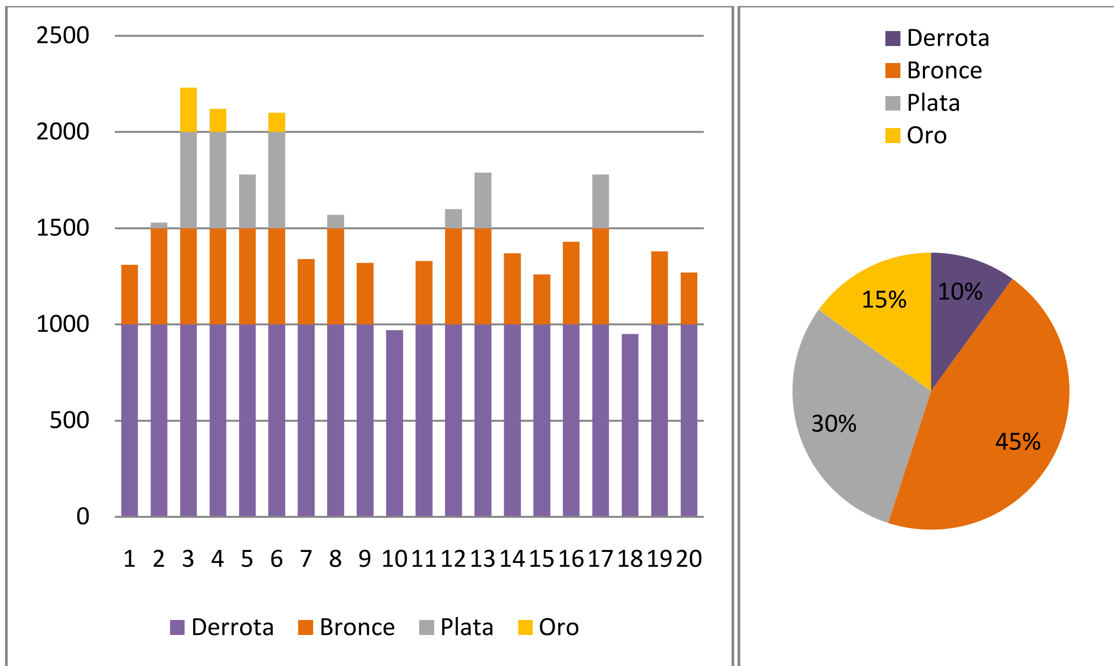
Gráficos del nivel 4 del modo clásico:



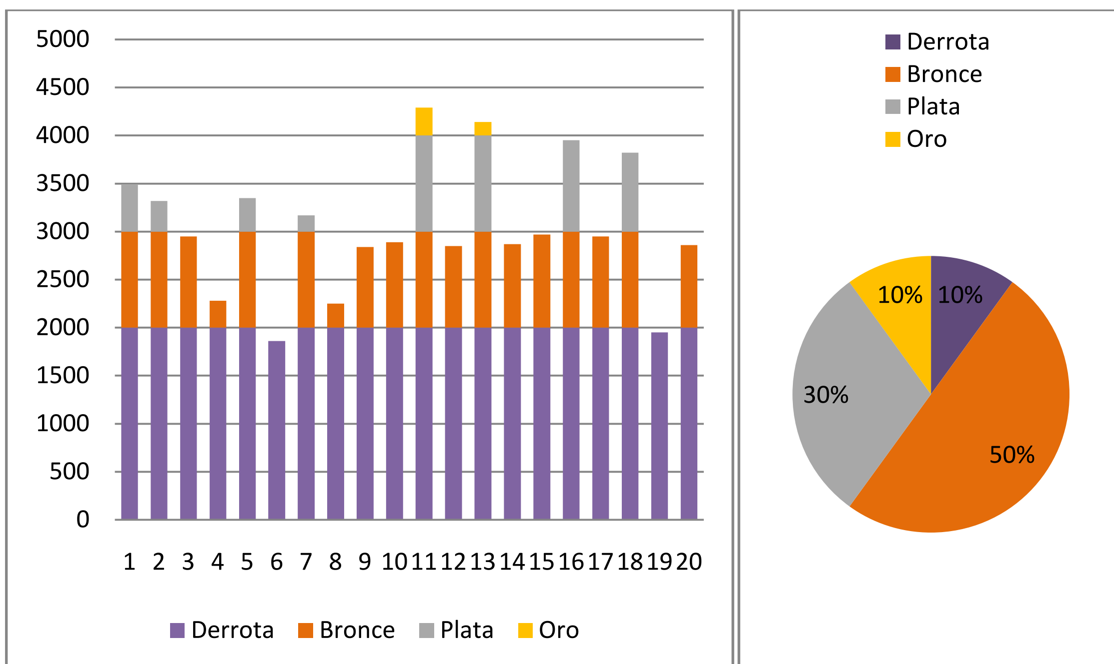
Tras ver los gráficos de los niveles del modo clásico se puede ver la tendencia de incremento en la dificultad con un aumento en las derrotas en los niveles posteriores, y también se puede ver como se mantiene por igual la dificultad de obtener una puntuación de rango oro para hacer rejugables todos los niveles del juego.

En los siguiente gráficos se pueden observar las puntuaciones obtenidas en veinte partidas de cada uno de los niveles iníciales del modo tiempo, así como los porcentajes de derrota y de obtención de una puntuación de rango bronce, plata y oro.

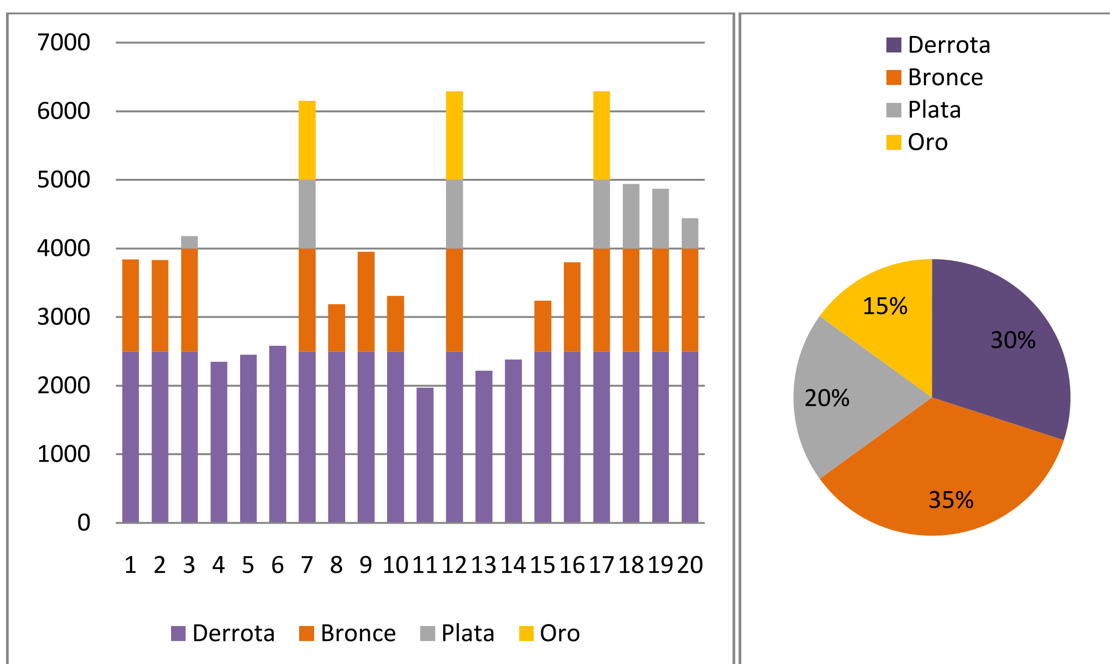
Gráficos del nivel 1 del modo tiempo:



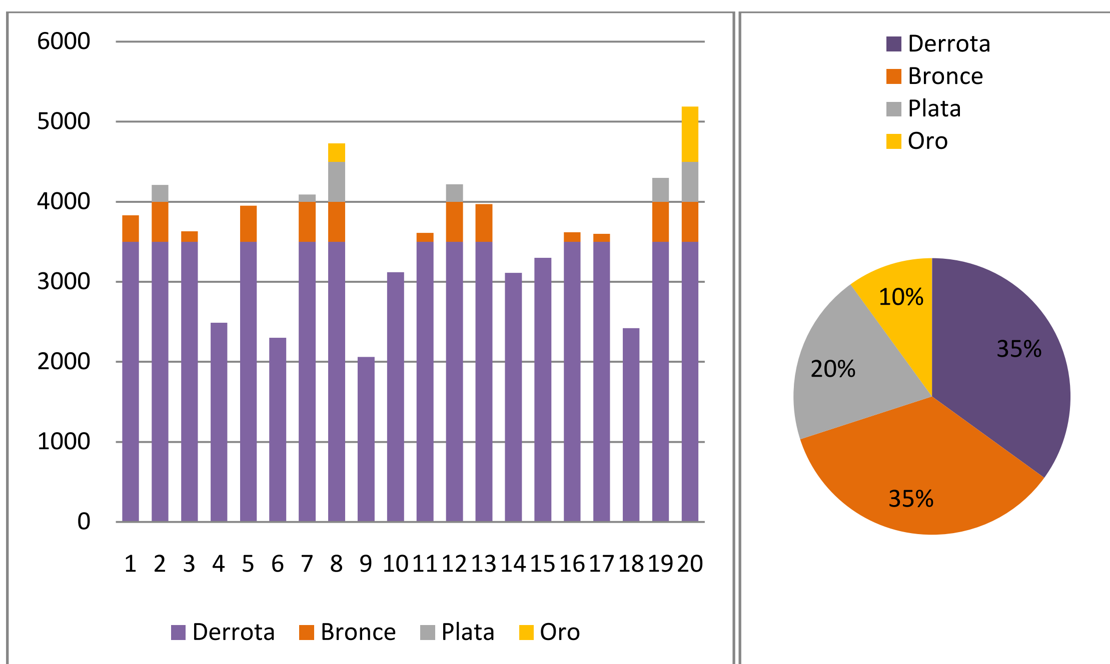
Gráficos del nivel 2 del modo tiempo:



Gráficos del nivel 3 del modo tiempo:



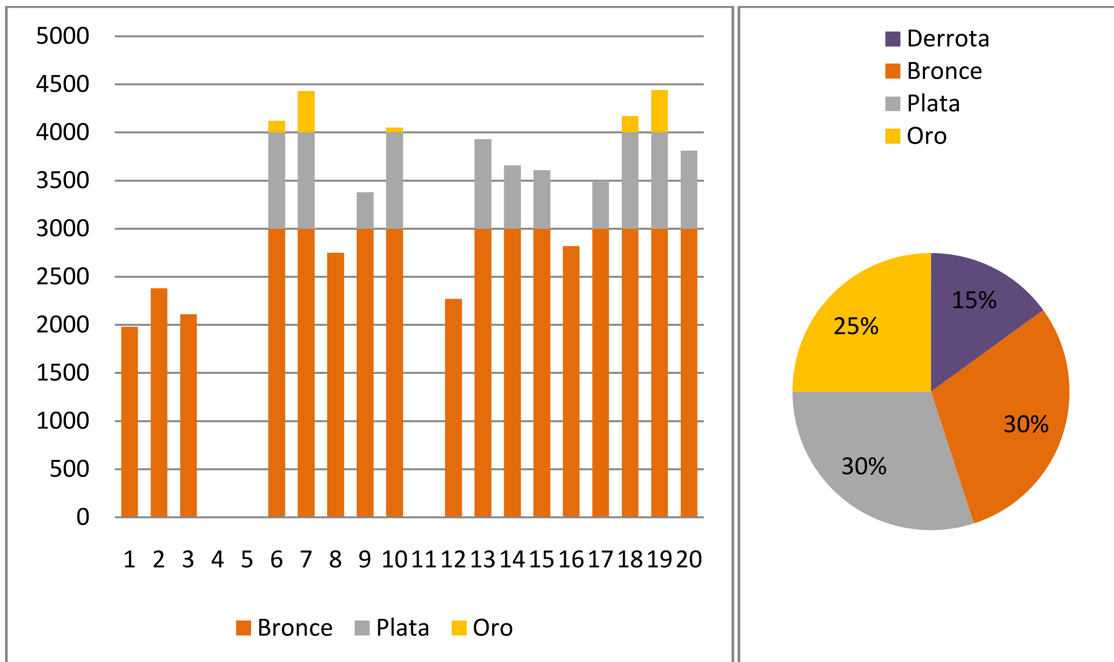
Gráficos del nivel 4 del modo tiempo:



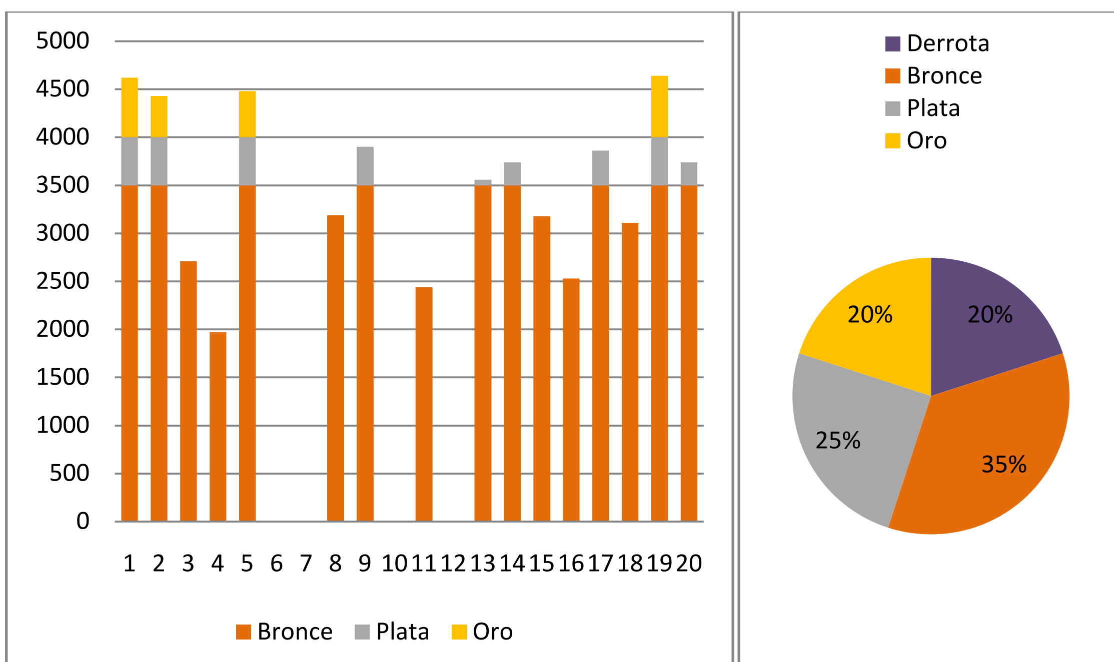
En este caso viendo los gráficos de los niveles del modo tiempo, al igual que en el modo clásico, también se puede ver la tendencia de incremento en la dificultad con un aumento en las derrotas en los niveles posteriores, y también se puede ver como se mantiene por igual la dificultad de obtener una puntuación de rango oro para hacer rejugables todos los niveles del juego.

En los siguiente gráficos se pueden observar las puntuaciones obtenidas en veinte partidas de cada uno de los niveles iníciales del modo virus, así como los porcentajes de derrota y de obtención de una puntuación de rango bronce, plata y oro. En este caso, se puede apreciar como el jugador en no superar el nivel pasa a tener automáticamente una puntuación de cero puntos.

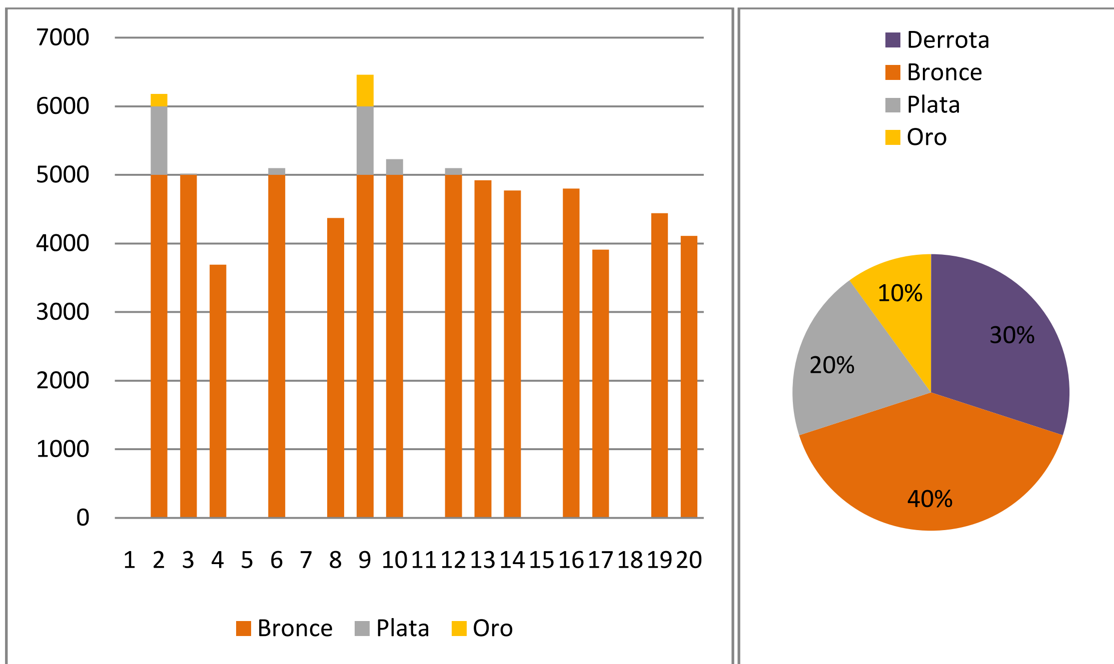
Gráficos del nivel 1 del modo virus:



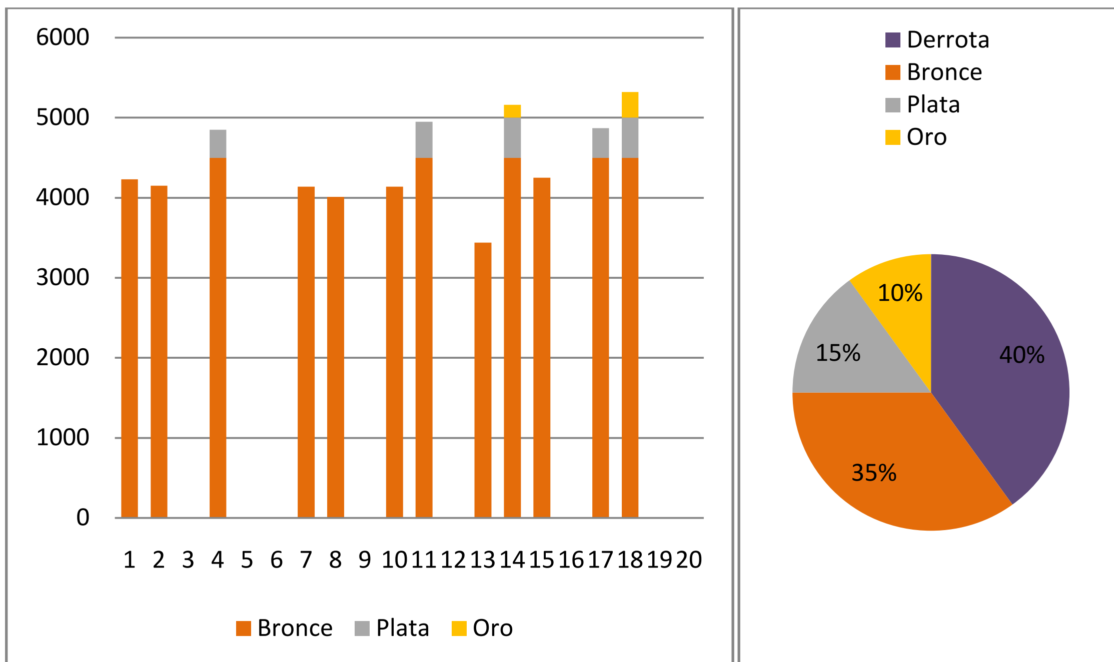
Gráficos del nivel 2 del modo virus:



Gráficos del nivel 3 del modo virus:



Gráficos del nivel 4 del modo virus:



Aquí, al igual que en los otros dos modos, también se puede ver la tendencia de incremento en la dificultad con un aumento en las derrotas en los niveles posteriores, pero, a diferencia de los otros dos modos de juego, también se puede ver como se ha facilitado la obtención de una puntuación de rango oro en los primeros niveles para motivar al jugador.

Conclusión

Como conclusión, decir que se han logrado llevar a término todos los objetivos planificados en las tareas presentes en el diagrama de Gantt, lo cual me ha dado como resultado un videojuego agradable visualmente mediante sus diseños y animaciones, que funciona correctamente y que dispone de diversos modos de juego con sus respectivos niveles dentro de cada uno de éstos.

Luego, también comentar que el desarrollo de un videojuego, por pequeño que éste sea, es muy laborioso y en éste intervienen diferentes campos como el diseño gráfico, la animación y la programación para los que hay que aprender a utilizar herramientas diversas, como Adobe Illustrator, Unity y MonoDevelop, tal y como he hecho yo para realizar este videojuego.

Por último, para finalizar, ya solo decir algunas cosas que de cara a seguir con el desarrollo del videojuego se podrían implementar, como niveles del juego con dimensiones no cuadriculadas, como evitar que se realicen conjuntos de piezas al iniciar el juego, como más tipos de piezas especiales, por ejemplo, una que borre un área, etc. En definitiva, que siempre se puede ir mejorando y añadiendo más características a un videojuego, solo hace falta tiempo, recursos y ganas.

Fuentes

Trabajadores de Unity. "Manual de Unity". [Publicado el 11 de julio de 2017 en la web de Unity <https://docs.unity3d.com/>]. [Fecha de consulta: 02/10/2017]. Enlace: <https://docs.unity3d.com/es/current/Manual/UnityManual.html>

Lin, Wilmer. "Make a Match-Three Puzzle Game in Unity". [Publicado en septiembre de 2017 en la web de Udemy <https://www.udemy.com/>]. [Fecha de consulta: 02/10/2017]. Enlace: <https://www.udemy.com/make-a-puzzle-match-game-in-unity/>

Hecker, Kelley. "Building a Match 3 Game with Unity". [Publicado el 22 de abril de 2016 en la web de Lynda.com <https://www.lynda.com/>]. [Fecha de consulta: 02/10/2017]. Enlace: <https://www.lynda.com/Unity-tutorials/Build-Match-3-Game-Unity/440670-2.html>

DGkanatsios. "Building a match-3 game (like Candy Crush) in Unity". [Publicado el 25 de febrero de 2015 en el blog de DGkanatsios <https://dgkanatsios.com/>]. [Fecha de consulta: 02/10/2017]. Enlace: <https://dgkanatsios.com/2015/02/25/building-a-match-3-game-in-unity-3/>

Dziedzic, Łukasz. "LATO". [Publicado el 20 de diciembre de 2010 en la web de Font Squirrel <https://www.fontsquirrel.com/>]. [Fecha de consulta: 18/12/2017]. Licencia: SIL Open Font License Versión 1.1. Enlace: <https://www.fontsquirrel.com/fonts/lato>

FoolBoyMedia. "Sky Loop". [Publicado el 16 de febrero de 2015 en la web de Freesound <https://freesound.org/>]. [Fecha de consulta: 18/12/2017]. Licencia: Creative Commons Attribution - NonCommercial 3.0. Enlace: <https://freesound.org/people/FoolBoyMedia/sounds/264295/>

Sclolex. "Crickets". [Publicado el 12 de diciembre de 2013 en la web de Freesound <https://freesound.org/>]. [Fecha de consulta: 18/12/2017]. Licencia: Creative Commons Universal 1.0. Enlace: <https://freesound.org/people/Sclolex/sounds/210540/>

CosmicEmbers. "Myst Book Toll". [Publicado el 7 de julio de 2012 en la web de Freesound <https://freesound.org/>]. [Fecha de consulta: 18/12/2017]. Licencia: Creative Commons Attribution 3.0. Enlace: <https://freesound.org/people/CosmicEmbers/sounds/160690/>

Sirkoto51. "Retro Puzzle Music Loop". [Publicado el 27 de enero de 2012 en la web de Freesound <https://freesound.org/>]. [Fecha de consulta: 18/12/2017]. Licencia: Creative Commons Attribution 3.0. Enlace: <https://freesound.org/people/Sirkoto51/sounds/378110/>

Scrampunk. "OKAY!". [Publicado el 6 de mayo de 2016 en la web de Freesound <https://freesound.org/>]. [Fecha de consulta: 18/12/2017]. Licencia: Creative Commons Attribution 3.0. Enlace: <https://freesound.org/people/Scrampunk/sounds/345299/>