UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

Desarrollo de un videojuego empleando RT-DESK con XNA y C#

Proyecto final de carrera
Licenciatura en ingeniería informática
Septiembre 2015

Autor: Carlos Torres Martínez

Director: Ramón Mollá Vayá

A todos los que han fomentado y apoyado
mi pasión por los videojuegos.

Índice

Resumen

Mediante la creación de un videojuego y su adaptación a la tecnología RT-DESK, basada en la simulación discreta de eventos y creada en la universidad politécnica de Valencia, se ha buscado tanto formarse en la creación de los mismos como en contribuir a la investigación de la nueva tecnología y su aplicación a los videojuegos.
La descripción del trabajo realizado en la programación del videojuego y que se encuentra en esta memoria puede servir de ayuda para aquellos que tengan intención de iniciarse en la creación de videojuegos.
Por su parte, la posterior adaptación a la tecnología RT-DESK y el análisis comparativo entre ambas versiones del videojuego han revalidado la capacidad de dicha tecnología y sus posibilidades frente al bucle principal convencional.

Estructura de la memoria

En primer lugar se puede encontrar una **introducción** al proyecto, donde se describe brevemente y justifica su desarrollo y la motivación detrás del mismo, además de describir el contexto en el que se realiza y aclarar algunos conceptos básicos y necesarios para la comprensión de los siguientes apartados.

A continuación se analiza el **estado del arte**, tanto de los videojuegos como de la creación de los mismos y las tecnologías que los apoyan, así como los precedentes que sentaron las bases y fueron inspiración del proyecto.

En el apartado de **análisis** se analiza el problema y se planifica el trabajo que conlleva y habrá que realizar, así como una justificación de las herramientas a usar.

En el siguiente apartado, **implementación,** se detallan las diferentes soluciones llevadas a cabo para las diferentes partes del videojuego.

En **RT-DESK** se trata la segunda parte del proyecto, la adaptación a RT-DESK, donde se explica de qué trata la tecnología así como el proceso realizado al adaptar el juego a la nueva tecnología, y además se aporta un análisis comparativo entre ambas versiones.

Para acabar, en **conclusiones** se encuentran la valoración y consideraciones finales, tanto acerca del trabajo realizado como de la utilidad de las tecnologías usadas y su aplicación en futuros proyectos.

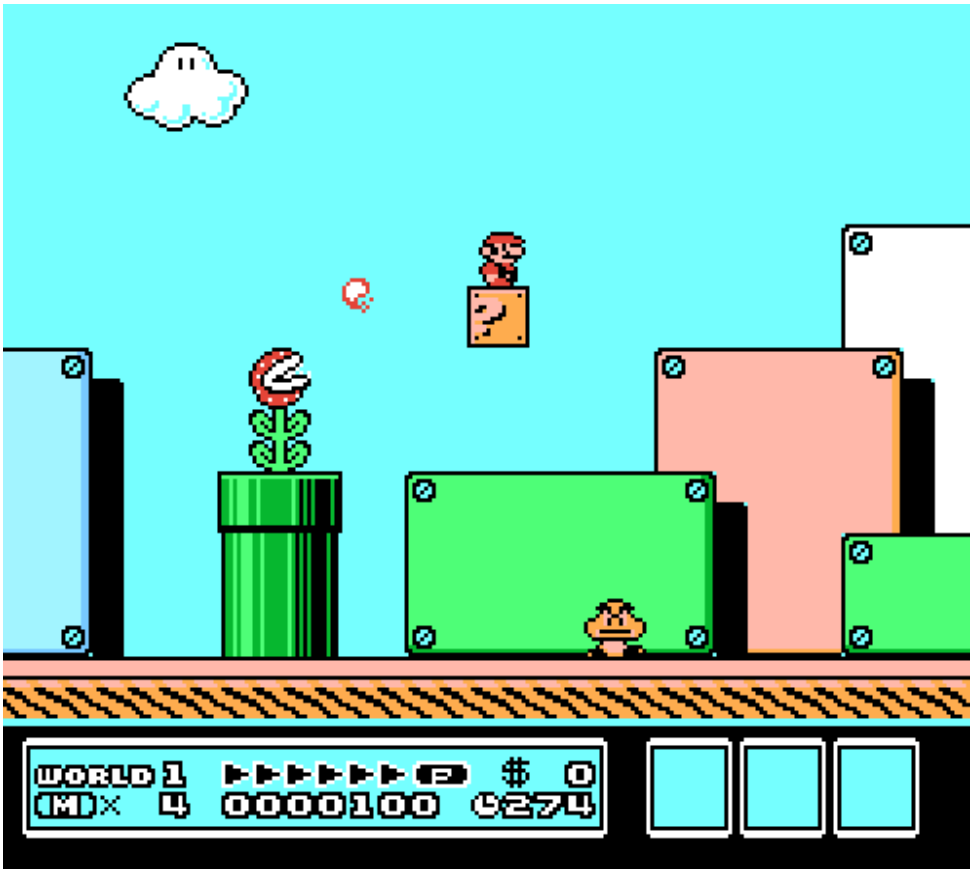
La memoria se cierra con un pequeño **glosario**, la **bibliografía** consultada y el documento de diseño de juego, **anexo A**, del videojuego implementado.

Introducción

¿Qué es un videojuego?

Un videojuego (A veces abreviado simplemente como “juego”) es un juego electrónico en la que el jugador, a partir de una información principalmente visual y sonora, toma decisiones y las transmite mediante un dispositivo de entrada.

La creación de un videojuego está muy relacionada con artes como la música, la pintura o la escritura, pero es esa interacción entre juego y el jugador la que lo define y diferencia del resto de medios de expresión.



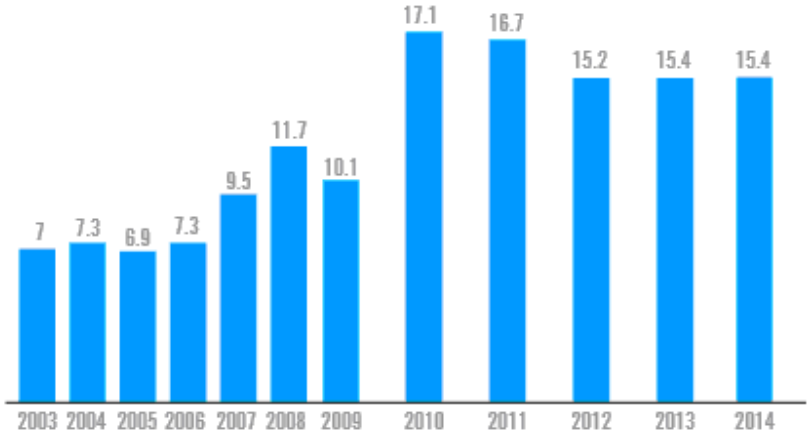
Lanzado por la compañía Nintendo para la consola NES en 1988, Super Mario Bros. 3 se trata de uno de los videojuegos de mayor éxito.

La industria del videojuego

El origen de los videojuegos se remonta a la década de los cincuenta, pero no fue hasta los setenta cuando el descenso en los costes del hardware permitió su comercialización y expansión entre el público, propiciando la creación de una industria y su desarrollo.

Además de funcionar sobre ordenadores personales, surgieron videoconsolas, sistemas cerrados pensados para la ejecución de videojuegos, lo cual permitió tanto reducir los costes de fabricación y venta de los sistemas, así como facilitar su uso, acercando los videojuegos a un mayor número de personas.

De entre las diferentes industrias enfocadas al entretenimiento, los videojuegos ha sido la que más ha crecido en los últimos años, llegando a suponer un volumen de negocio superior al del resto de industrias[1]. Por otro lado, la llegada y expansión del sector móvil y el videojuego social ha hecho que la cantidad de clientes potenciales aumente drásticamente, y las nuevas generaciones nacen y crecen conociendo el concepto de videojuego.
Así pues, estamos en un punto en el que los videojuegos se han convertido en un medio de expresión y una forma de ocio enormemente aceptada y extendida.



Miles de millones en ventas de videojuegos en Estados Unidos[2].

El diseño de videojuegos ha estado influenciado por las limitaciones que imponía la tecnología de cada época, como limitaciones de memoria o de procesamiento. El avance tecnológico ha ido acabando con dichas limitaciones y permitiendo, por ejemplo, el salto de los videojuegos en 2d a los 3d. El mayor detalle en la recreación que la tecnología va permitiendo está estrechamente relacionado con los costes de desarrollo necesarios para aprovecharla, los cuales se han visto incrementados de forma drástica[3].

[1] http://www.fastcompany.com/3021008/why-video-games-succeed-where-the-movie-and-music-industries-fail
[2] http://www.theesa.com/wp-content/uploads/2015/04/ESA-Essential-Facts-2015.pdf
[3] http://kotaku.com/how-much-does-it-cost-to-make-a-big-video-game-1501413649

El auge del videojuego independiente

Si bien el aumento en los costes que suponía desarrollar para una tecnología más avanzada propició, en las décadas de los ochenta, noventa y el nuevo siglo, que cada vez más pequeñas empresas no pudiesen competir en el sector comercial, ha sido ese mismo avance en las tecnologías, de comunicaciones en este caso, lo que permitió la creación de webs donde colgar juegos (Newgrounds, Kongregate, etc.) aunque con escasa o nula repercusión comercial, y posteriormente plataformas de distribución y venta digital, las cuales han permitido disminuir esos costes de distribución y llegar a un mayor público, así como darle a los desarrolladores más pequeños la posibilidad de distribuir comercialmente sus juegos, dando lugar a un nuevo auge del videojuego independiente.

Algunas de las primeras plataformas de distribución digital de videojuegos fueron Steam[4], creada por la compañía Valve en el año 2004, o el servicio Xbox Live Arcade para consolas de Microsoft.

El auge del videojuego independiente y su presencia en la primera línea de la industria ha fomentado tanto la innovación como la recuperación de diferentes tipos de videojuegos que se fueron dejando atrás según la tecnología avanzaba, debido a la mayor oportunidad de estos de encontrar su público y al menor riesgo económico que supone lanzar un juego en una plataforma de distribución digital.

Cave Story, creado por Pixel[5] en el 2004, se trata de uno de los primeros y mejor valorados juegos de este nuevo auge del videojuego independiente.

[4] http://store.steampowered.com/
[5] http://studiopixel.sakura.ne.jp/

Objetivo

El proyecto consta de dos partes:

Creación de un videojuego

Los videojuegos requieren de muchas y diferentes disciplinas para su creación: Diseño, ingeniería, composición musical, grafismo, modelado, escritura, etc. Y dentro de la parte de ingeniería se toca de todo: Programación, redes, inteligencia artificial, hardware, gráficos por computador, teoría de lenguajes, etc.
Mediante la creación del videojuego, tal y como se ha comentado se pretende poner en práctica esos conocimientos obtenidos aplicándolos al desarrollo de un videojuego.

Entre las tareas relativas al proceso de creación del videojuego y que se realizarán en este proyecto encontramos:

- Planificación del proyecto: Planificación temporal, de tareas, etc.
- Diseño: Especificación de la experiencia de juego, incluyendo las reglas y restricciones de la comunicación juego - jugador.
- Obtención de assets. El juego requiere de imágenes, sonidos, etc. estos deben ser creados y editados para su uso en el videojuego.
- Implementación: Diseñar el sistema y programar el sistema.
- Testeo: Comprobar que no existan fallos, y solucionarlos si los hay.

Siguiendo el proceso de creación de videojuegos se pretende aprender y formarse en el mismo.

Port a la tecnología RT-DESK

Una vez realizado el videojuego se continuará con la conversión del mismo a la tecnología RT-DESK, creada en la propia universidad y basada en la simulación discreta desacoplada y sobre la que se entrará en detalle más adelante, documentando el proceso y se realiza una comparación de rendimiento entre ambas versiones del videojuego.

Motivación

Se escogió como proyecto un videojuego principalmente por el interés en la creación de estos. Si bien en nuestro proceso de formación no hemos tocado su desarrollo de forma directa este requiere de muchas disciplinas de las aprendidas, como procesamiento de lenguajes, gráficos, algorítmica o inteligencia artificial, las cuales se esperan poner en práctica para continuar con su formación en ellas y en la creación de videojuegos.

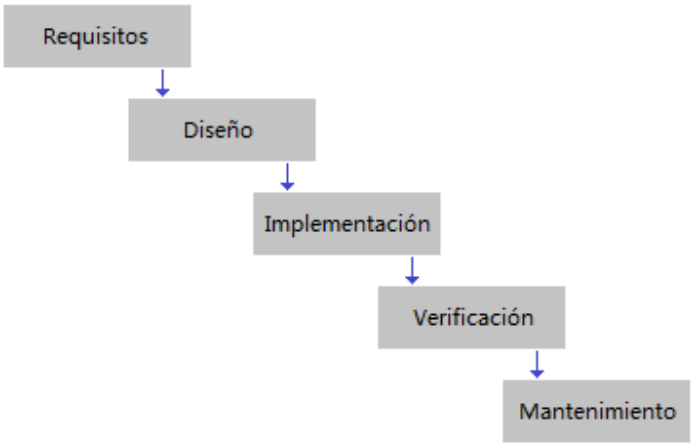
El auge de la distribución digital ha puesto al alcance de todos la posibilidad de vender o compartir su trabajo, y esta democratización ha propiciado la aparición de herramientas

enfocadas al desarrollo independiente, como XNA, o de ofertas pensadas para al pequeño desarrollador, como es el caso de las licencias más baratas de motores como Unity, CryEngine o Unreal Engine, además de un apoyo creciente del desarrollo independiente por parte de las grandes empresas del sector.

Nos encontramos, entonces, en un momento ideal para que los interesados en desarrollar y vender videojuegos puedan dedicarse a ello y hacerse un pequeño hueco en la industria. Además, el proceso de evolución tecnológica es constante, e investigar nuevas tecnologías para su aplicación en videojuegos, en este caso RT-DESK, ofrece una buena oportunidad de destacar y contribuir a esa evolución.

Metodología

La metodología escogida es waterfall[6] (En cascada), la cual se caracteriza por una importante planificación inicial que divide el proyecto en varias fases que se siguen de manera estricta y de forma secuencial para terminar el producto.



Fases de la metodología waterfall.

- Fase de requisitos:
  Se realiza un analisis de las necesidades del usuario y se determinan los objetivos del prdocuto.
- Fase de diseño:
  Es la fase en la que partiendo de los requisitos se diseña el programa que los cumple, además de realizar una rigurosa descripción del mismo que se seguirá en su implementación.
- Fase de implementación:
  En esta fase se realiza la implementación del producto.
- Fase de verificación:
  Se realizan pruebas de testeo sobre la implementación para verificar que cumple los requisitos e intentar minimizar los errores y comprotameintos no deseados.

[6] https://en.wikipedia.org/wiki/Waterfall_model

- Fase de mantenimiento:
  Tras lanzar el producto se sigue ofreciendo un servicio técnico a los usuarios y corrigiendo posibles errores no encontrados en la fase de verificación.

Se trata de una metodología muy extendida entre las grandes compañías, las cuales no pueden permitirse grandes retrasos o realizar grandes cambios, siendo estas las mismas razones por las que se ha escogido para este proyecto, al tener un tiempo limitado y la necesidad de tener una dirección de trabajo clara.
Sin embargo, para el proceso de diseño del videojuego previamente (Fase de requisitos) sí se trabajó de forma iterativa sobre diferentes prototipos tanto como para obtener ciertos conocimientos básicos y necesarios de cara al proyecto como para aclarar las ideas en torno al mismo y poder realizar un GDD[7] (Game Design Document, documento de diseño del juego) con un diseño sólido y verificado hasta cierto punto que se pudiese mantener a lo largo del proyecto, en tanto que uno de los errores comunes en los que se tiende a caer es en diseñar y mejorar el juego según avanza el desarrollo del mismo, alargando este más de lo debido.

[7] https://en.wikipedia.org/wiki/Game_design_document

Estado del arte

El bucle principal

Como aplicación en tiempo real que es, el núcleo de un videojuego es un bucle principal[8] que se ejecuta constantemente y se encarga principalmente de dos fases:

- Actualización:
  El bucle debe actualizar en cada instante los diferentes elementos que forman el videojuego en función del comportamiento programado de los mismos y de la entrada por parte del jugador, gestionada al inicio de este misma fase.
- Dibujado:
  El bucle debe, tras acabar la fase de actualización, dibujar en pantalla el estado del juego en ese instante.

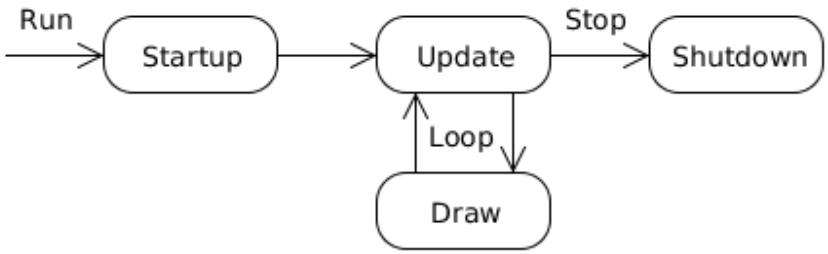


Diagrama del bucle principal básico.

Es esta actualización y dibujado constante la que permite que el jugador interactúe en tiempo real con el videojuego y este se sienta "vivo". Cada iteración del bucle puede tardar un tiempo diferente, sin embargo la velocidad a la que el juego se ejecuta, idealmente, debe de ser constante. Pueden darse diferentes casos en función del tiempo prefijado de iteración y el tiempo de una iteración al ser ejecutada:

- Si en una iteración se han acabado tanto la fase de actualización como la de dibujado y sobra tiempo el bucle debería de esperar a que pase el tiempo suficiente para que la siguiente iteración comience a ejecutarse de forma que haya transcurrido un tiempo prefijado. Sin embargo sí puede realizar algunas tareas en paralelo que no afecten a la velocidad del juego, como cargar assets o realizar cálculos de inteligencia artificial.
- Si la iteración tarda más que el tiempo prefijado de iteración la velocidad de juego disminuye, ralentizándolo. Una solución pasa por controlar la velocidad de los elementos del juego de forma relativa al tiempo de la última iteración, de forma que aún perdiendo fluidez la velocidad de juego se mantenga constante y afecte lo posible a la experiencia del jugador.

[8] http://entropyinteractive.com/2011/02/game-engine-design-the-game-loop/

Típicamente, la aplicación se encarga de realizar una fase de arranque en la que se llevan a cabo tareas como la creación de la ventana de juego o la inicialización de los sistemas necesarios (como el sistema de audio, por ejemplo), para a continuación entrar en el bucle principal, del que solo se sale al acabar la ejecución del programa, pasando primero por una fase de cierre donde se realizan tareas como liberar la memoria reservada por el juego.

Motores y librerías

Un motor de videojuego[9] un framework que provee de una serie de funcionalidades necesarias para el funcionamiento del juego, como puede ser un motor de renderizado para 2D y 3D, motor de físicas y detector de colisiones, animación, audio, scripting, inteligencia artificial, redes, administración de memoria, etc. Todo ello usualmente envuelto en una interfaz gráfica que facilita su uso, aumenta la productividad y además permite que personas no relacionadas con la programación, como puede ser gente del departamento de arte o de sonido, puedan trabajar de forma directa en el producto final, facilitando en última instancia la implementación del diseño, pasando del papel al producto software.

Uno de los puntos fuertes de los motores es la reusabilidad, servir de base para más de un proyecto, evitando repetir el trabajo, ganando tiempo y reduciendo costes. Otra característica que ha ido adquiriendo muchísima importancia es la portabilidad, facilitar lanzar el juego en diferentes plataformas.

Muchos motores se realizan de forma interna en una empresa, como el reciente Fox Engine de Kojima Productions y Konami. Sin embargo, también es típico que se permita su uso a otros desarrolladores mediante una licencia. Algunos de los más famosos en la actualidad son Unreal Engine, de Epic, CryEngine, de Crytek o Unity de Unity Technologies. Ha sido el creciente número de pequeños desarrolladores el que ha impulsado el uso de los motores más económicos, como es el caso de Unity, lo que ha su vez ha fomentado la competencia y la "guerra de precios", permitiendo así que cualquier desarrollador puede acceder a una licencia de varios de los mejores motores en la actualidad.

Otra opción es, en vez de utilizar un motor ya creado, hacerse uno mismo lo necesario partiendo de librerías de gestión de la entrada y la salida. Algunas librerías conocidas y extendidas son XNA, SDL o SFML.

[9] https://en.wikipedia.org/wiki/Game_engine

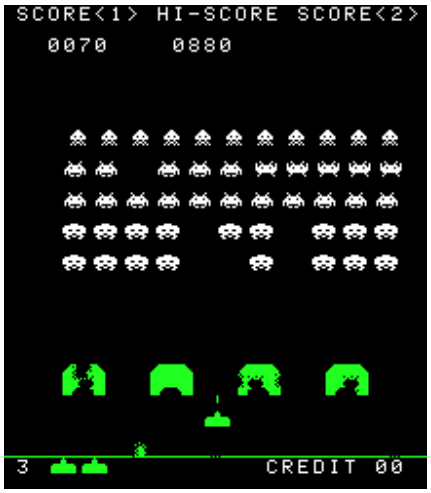
Precedentes, el género shoot ‘em up

Existen diferentes géneros a la hora de catalogar un videojuego en función de diferentes aspectos. Según su temática puede ser bélico, de terror, etc. Según su forma de jugarlo puede ser de acción, de puzzles, de estrategia, etc.

El videojuego realizado en este proyecto pertenece al género shoot ‘em up, concretamente al de los juegos de naves 2d, también conocido coloquialmente en España como “matamarcianos” o “juego de naves” debido al gran número de producciones de temática espacial. Es un género que se caracteriza por la simplicidad de sus controles y por poner a prueba los reflejos y habilidad del jugador con el mando.

Algunas de las señas de identidad del género son el movimiento libre a lo largo de los dos ejes sobre el que por lo normal únicamente influye el jugador, el movimiento fijo de la cámara o el desarrollo de sus niveles, definidos por una serie de obstáculos y enemigos que continuamente se interponen en el camino del jugador, dejando pocos momentos de tranquilidad.

Algunos de los juegos que sirvieron de base para el género son Space invaders, Centipede, Asteroids, Galaga, etc.



Space invaders de Taito, 1978, un pionero del género y de los videojuegos.

El género ganó en popularidad y muchas de las grandes compañías, especialmente entre las japonesas, realizaron juegos y sagas de este género para máquinas recreativas, así como versiones para consolas domésticas cuando las prestaciones de estas lo permitían.

Particularmente conocidos son Konami con su saga Gradius, Treasure con Radiant silvergun, Ikaruga o Gradius V o la aproximación de la compañía Cave al “bullet hell”, un subgénero cuya particularidad es la ingente cantidad de disparos a las que se enfrenta el jugador.

Análisis

Requisitos

Los requisitos que debe cumplir el proyecto son:

- Gestión de los gráficos.
  El juego deberá cargar los assets gráficos que se usaran en el juego, así como controlar la correcta visualización y animación de estos.
- Subsistema de control de Audio.
  Debe permitir gestionar los efectos y la música y permitir controlar su reproducción (Reproducir, pausar, cortar, aplicar efectos de fundido y controlar la repetición de sonidos.)
- Subsistema de control de la entrada.
  Debe proveer de un control y gestión de los dispositivos de entrada usados en el juego, mando y teclado, para facilitar el manejo de la interacción del jugador.
- Menús.
  Implementar un sistema para introducir un menú de juego con diferentes pantallas y diferentes opciones, así como permitir hacer uso del mismo APRA cambiar diferentes características del juego (como podría ser el idioma, el nivel a jugar, etc.).
- Gestión de pantallas de arranque.
  Proveer de un sencillo sistema que permita controlar las pantallas de arranque del juego, ya sea para mostrar el nombre del estudio creador, de las tecnologías empleadas y demás información necesaria.
- Gestión de puntuaciones.
  Como en la mayoría de juegos del género, la competición entre jugadores por alcanzar las máximas puntuaciones es uno de los puntos fuertes que invita a rejugar y mejorar. El juego permitirá introducir a los jugadores su puntuación, si esta se encuentra entre las diez mejores, así como sus iniciales. Además, el menú de opciones contará con una opción para resetear la tabla.
- Personaje principal.
  El personaje principal podrá moverse libremente en cualquier dirección usando el stick analógico del mando, lo cual es un punto fuerte respecto al movimiento discretizado a 8 direcciones de otros juegos, especialmente en PC al estar pensados para ser jugados mediante un teclado y que el mayor número de personas puedan jugarlo. También podrá disparar en diferentes direcciones, no solo en una como es habitual, y esta dirección de disparo será independiente de la dirección del movimiento de la nave, como se puede ver en juegos como Geometry Wars o Smash TV.
- Gestión de enemigos e incluir diferentes tipos.
- Gestión de disparos e incluir diferentes tipos.
- Fondos y gestión de estos.
  No tanto por una decisión de diseño como por el ánimo de aprender, el juego contará con al menos dos tipos de fondos, uno creado a partir de un mapa de tiles y otro generado proceduralmente.
- Gestión de niveles.

El juego contará con diferentes fases y jefes, el jugador podrá jugarlos en orden en el modo historia o seleccionar uno de ellos en el modo de prueba. Se ha incluido está opción para facilitar mostrar el trabajo realizado en el PFC a aquellas personas que no cuenten con la habilidad o el tiempo suficiente para avanzar en el modo historia.

Para más detalle sobre los requisitos del videojuego consultar el GDD en el Anexo A.

En cuanto a requisitos tecnológicos, básicamente se necesita que la tecnología escogida como base del juego debe de ser compatible con RT-DESK.

Planificación temporal

La planificación temporal realizada a partir de los requisitos del juego así como de las fases necesarias en la creación de un videojuego:

Fase	Tarea	tiempo estimado
Requisitos	Brainstorming	1
Diseño	Creación GDD	4
	Diseño UML	4
Implementació	Herramientas	1
	Programa base	2
	Esqueleto de clases	5
	Input Manager	2
	Player + playerBulletManager	7
	EnemyBulletManager + Enemy bullet	5
	EnemyManager + regular enemies	5
	Boss 1	3
	Boss 2	2
	Boss 3	2
	Boss 4	2
	StageManager	4
	Tile map background	3
	procedural background	1
	Menu	6
	AudioManager	5
	Añadir sonidos	2
	Creación de mapas de prueba	6
	OpeningScreenManager	2
	Tutorial + Creditos	2
Verificación	Testeo	2
Port		?
Pruebas		4

Nota: Aunque no está indicado en el diagrama y como se ha mencionado anteriormente, durante los meses anteriores se realizaron una serie de prototipos que concluyeron con un brainstorming final para decidir qué entraba y qué no y la descripción final del videojuego en un GDD.

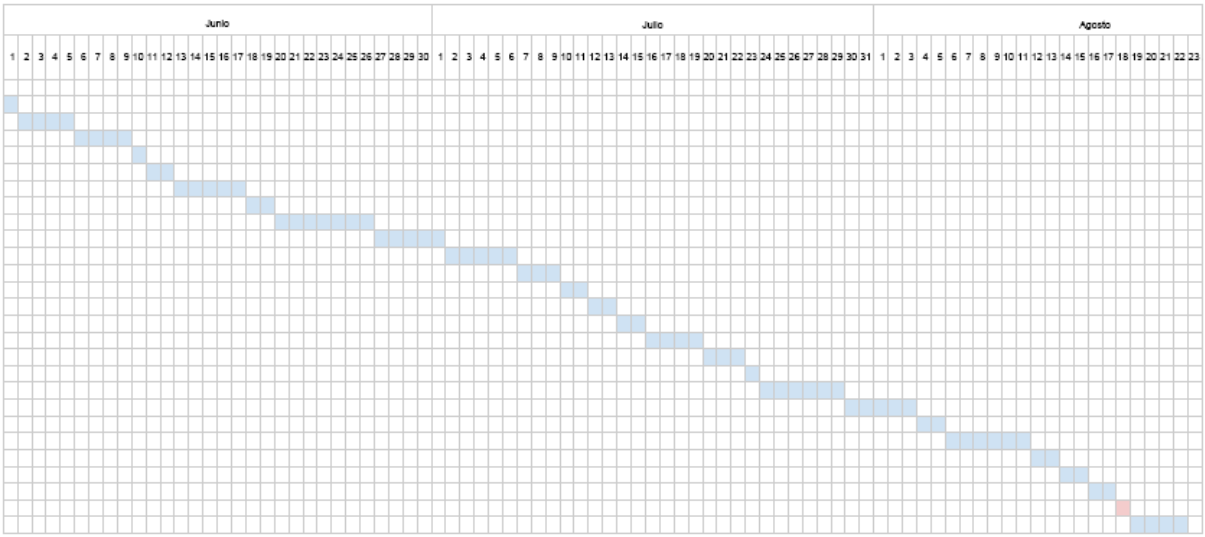


Diagrama de Gantt.

Herramientas

Tecnología escogida

A la hora de escoger la tecnología que serviría de base al videojuego se tuvieron en mente una serie de consideraciones:
Partir de un nivel más bajo usando librerías como XNA o SDL para construir uno mismo la base sobre la que funciona del videojuego tiene dos principales ventajas: El mejor aprendizaje del funcionamiento interno de un videojuego y la libertad que ofrece no estar limitado a las posibilidades de un motor concreto. Sin embargo, para que esa "libertad" extra que se tiene pueda dar sus frutos es necesario un gran trabajo y no suele merecer la pena: Hoy en día los motores de juegos ofrecen opciones más que suficientes para desarrollar casi cualquier videojuego y el ahorro en tiempo resulta enorme. Así pues, una de las principales razones por las que se optó por partir de una librería es la voluntad de aprender y hacerse una idea, al dar el salto a un motor en un futuro, del funcionamiento interno de estos.

De entre las librerías que se consideraron, como SDL, SFML o XNA, finalmente se optó por XNA, en su última versión 4.0, por los conocimientos previos que se tenían del lenguaje C# requerido por esta así como por experiencias previas con ella que facilitarían el desarrollo del proyecto. Además, la tecnología XNA cumple el requisito de ser compatible con RT-DESK, compatibilidad que fue fruto de un trabajo de final de carrera anterior y del que se parte y busca expandir su documentación.

Otras ventaja, compartidas por las otras librerías mencionadas, es que todas son de uso gratuito. Las principales desventajas de XNA frente al resto son, en primer lugar, que su uso se reduce a dispositivos Microsoft, al contrario que las demás que funcionan en una variedad más amplia de dispositivos. Sin embargo, no siendo el objetivo el lanzamiento comercial del juego y no queriendo centrarse en el desarrollo de versiones para diferentes sistemas sino en el aprendizaje en sí, no resultó un inconveniente. El otro inconveniente es que su desarrollo se ha descontinuado, y no se parece que Microsoft vaya a dar más soporte a este tecnología.

Aún así, cabe destacar como ventaja de XNA el desarrollo paralelo que se realizó de forma libre para adaptarla a diferentes plataformas, dando como resultado la tecnología MonoGame[10]. De esta forma, una vez completado su desarrollo la compatibilidad del código realizado con XNA será totalmente compatible con MonoGame permitiendo, hipotéticamente, lanzar versiones del proyecto para sistemas operativos como Linux o consolas que no sean de Microsoft.

Tecnología	Tipo	Lenguaje	Compatibilidad con RT-DESK	Plataformas	Conocimientos previos	Notas extra
XNA	Framework (bajo nivel)	C#	Sí	Dispositivos Microsft	Sí	Ha cesado su desarrollo, aunque existe una implementación libre, Monogame. Mucha documentación. Entorno de trabajo sólido.
Unity	Motor (alto nivel)	C#	En desarrollo	Windows, OSX, Linux, diferentes consolas y dispositivos moviles	No	Versiones gratuitas y de pago. Muy extendido, mucha documentación.
CryEngine	Motor (alto nivel)	C++	No	Windows, OSX, Linux, diferentes consolas	No	Versiones gratuitas y de pago.
SFML	Framework (bajo nivel)	C++, Pyton, Ruby	Sí	Windows, OSX, Linux, Android, IOS	No	No demasiado extendida, sin mucha documentación.
Monogame	Framework (bajo nivel)	C#	Sí	Windows, OSX, Linux, diferentes consolas y dispositivos moviles	No	Implementación libre de XNA.
SDL 2.0	Framework (bajo nivel)	C, C++	Sí	Windows, OSX, Linux, Dreamcast...	Sí	
Unreal engine 4	Motor (alto nivel)	C++, C#, assembly	No	Windows, OSX, Linux, diferentes consolas y dispositivos moviles	No	Versiones gratuitas y de pago.

Otras herramientas

Para el desarrollo del proyecto se ha optado por el uso de herramientas gratuitas, con la excepción de Microsoft Office y RT-DESK. La lista de herramientas usadas es la siguiente:

[10] http://www.monogame.net/

Visual Studio, el IDE (Integrated development environment) de Microsoft con integración con XNA, lo que lo convierte en la opción ideal. Uno de sus puntos fuertes sobre sus competidores es su robustez y su depurador.

RT-DESK[11] (Real Time Discrete Simulation Kernel) es una librería desarrollada en la UPV y diseñada para la gestión temporal de eventos discretos en tiempo real. Ofrece todas las ventajas de la simulación discreta pero además sincroniza la simulación con el tiempo real durante su ejecución. Es una herramienta enfocada tanto para el desarrollo de aplicaciones en tiempo real como pueden ser videojuegos, simuladores de entrenamiento, realidad virtual o realidad aumentada, como para modelar sistemas que estén descritos como una colección masiva de objetos simples que interactúen unos con otros como pueden ser la simulación de partículas o los autómatas celulares. En este proyecto, concretamente, se usará una versión para C# y previamente probada con XNA.

Google Docs y Microsoft Word, editores de texto usados para la creación de toda la documentación relativa al proyecto. Si bien Google Docs permite copias de seguridad en la nube que pueden ser modificadas desde cualquier lugar con internet usando una interfaz cómoda y simple, la ausencia de algunas opciones (Por ejemplo, índices que referencien el número de página de cada apartado) hacen que sea necesario terminar los documentos usando Microsoft Word.

Se ha utilizado OpenOffice Calc, una herramientas gratuita para la edición de hojas de cálculo, para la manipulación de los datos recogidos de los tests y su representación en gráficas.

ArgoUML[12], editor UML libre usado en la creación de algunos de los diagramas usados en la documentación. Una de las ventajas frente a sus competidores es que no requiere de una inscripción para su uso ni limita su funcionalidad en su versión gratuita. La versión para escritorio ofrece las herramientas necesarias para este proyecto.

Tiled[13], un editor de mapas de tiles, se usará para facilitar la generación de mapas de tiles usados para probar dicha funcionalidad del proyecto.

Paint y Gimp[14], editores de imágenes utilizados en la creación y edición de los assets gráficos necesarios para el juego. Paint ofrece una solución rápida y cómoda de editar imágenes a nivel de píxel, ideal para videojuegos sencillos en 2D a baja resolución, pero lo limitado de sus opciones hacen que para realizar una edición más compleja, si esta fuese necesaria, se opte por Gimp.

[11] http://rtdesk.blogs.upv.es/
[12] http://argouml.tigris.org/
[13] http://www.mapeditor.org/
[14] http://www.gimp.org/

Además, también se hará uso de GraphicsGale[15] para facilitar la animación de los mismos al permitir ver el resultado y modificarlo en tiempo real, además de traer una serie de opciones pensadas específicamente para crear y probar animaciones, como el efecto cebolla entre frames o modificar parámetros como la velocidad de cada frame de forma cómoda para hacerse una mejor idea de cómo quedará la animación dentro del juego.

Para la creación de efectos de sonido, aparte del uso de efectos y música libre, Bfxr[16] ofrece una solución ideal, pues permite crear sonidos de diferentes tipos (de salto, de disparo, etc.) forma aleatoria hasta encontrar uno que sea adecuado para el juego, sin necesidad de tener conocimiento alguno acerca de la creación de sonidos. Además, también permite variar algunos parámetros a aquellas personas que sepan algo o simplemente quieran curiosear.

Para la edición de los sonidos, como recortar o editar para facilitar su reproducción en bucle, se utilizará Audacity[17], un editor gratuito de sonido que destaca por su sencillez y variedad de opciones y que además fue el usado durante la carrera en diferentes asignaturas de sonido.

[15] http://www.humanbalance.net/gale/us/
[16] http://www.bfxr.net/
[17] http://audacityteam.org/

Diseño

Diseño UML

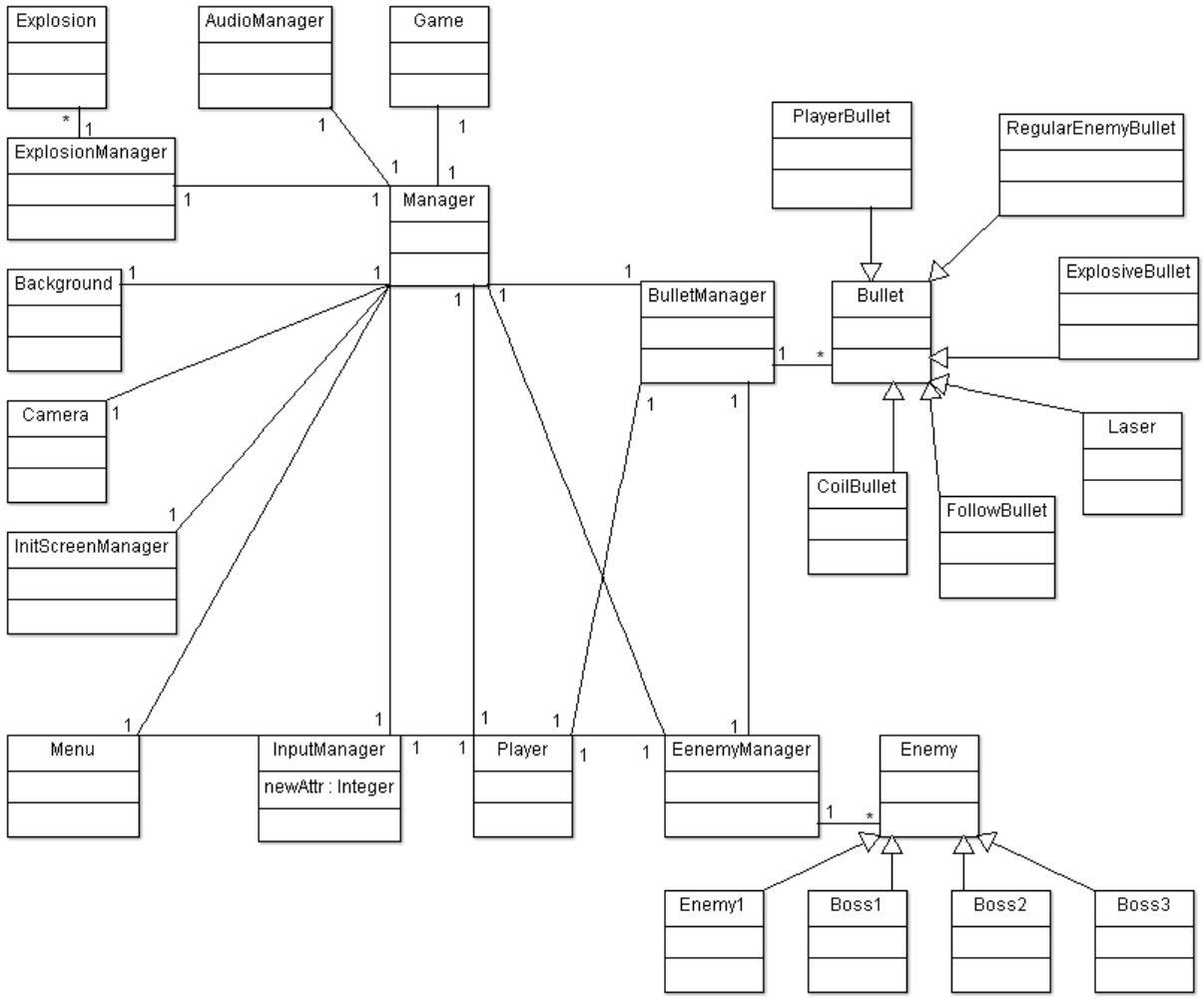
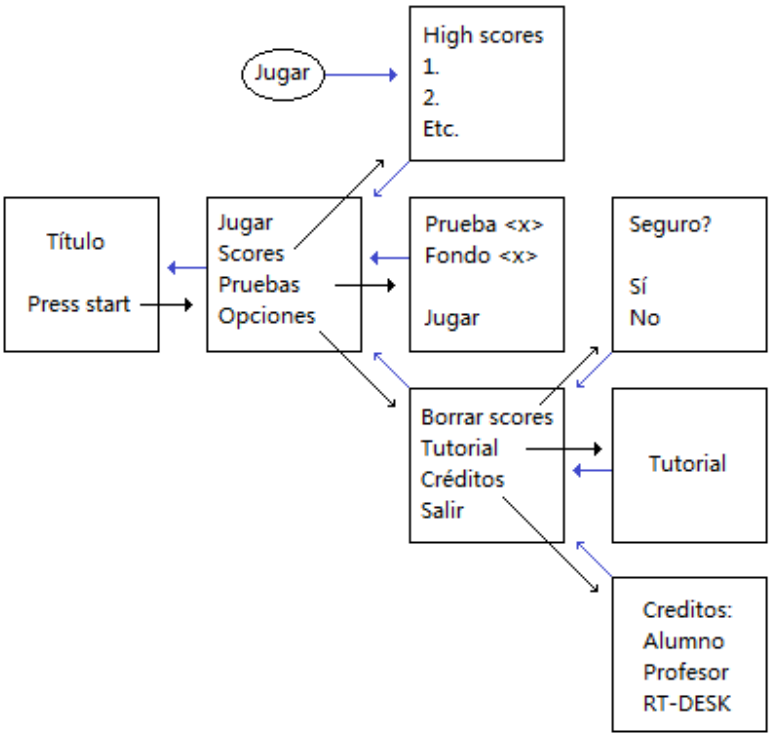


Diagrama UML.

Menú principal



Esquema del menú principal del juego.

Tras pasar las pantallas de introducción (Desarrollador, tecnologías empleadas, etc.) la pantalla en la que el jugador recibe el control del juego es la pantalla de título de juego, también llamada pantalla de “press start”, en la que se muestra el título del juego.
Desde ella se accede únicamente a la pantalla principal, desde donde puede comenzar una partida en al juego en el modo historia, acceder a las tablas de puntuaciones y a los menús de pruebas y opciones.
El menú de pruebas permite al jugador jugar una fase cualquiera, además de permitirle seleccionar qué fondo quiere usar para la partida.
En el menú de opciones se encuentra la opción de borrar la tabla de puntuaciones y devolverla a su estado original, además del tutorial donde se explica el funcionamiento del juego y los créditos, que incluyen el nombre del alumno, del director de proyecto, tecnologías empleadas y agradecimientos.
Además, la opción de borrar puntuaciones lleva a una pantalla de confirmación, para evitar que el jugador pueda borrar sus puntuaciones por equivocación.
Tras acabar una partida (Jugar) el juego vuelve a la pantalla de puntuaciones.

Implementación

Carga de assets

Los assets de un videojuego son sus recursos visuales y sonoros, tales como texturas, modelos o efectos de sonido. Para que el juego pueda hacer uso de ellos XNA provee de un content pipeline[18], que en el momento de la compilación se encarga de procesar los assets a un formato que el juego pueda reconocer y acceder a ellos mediante un content loader en el framework.

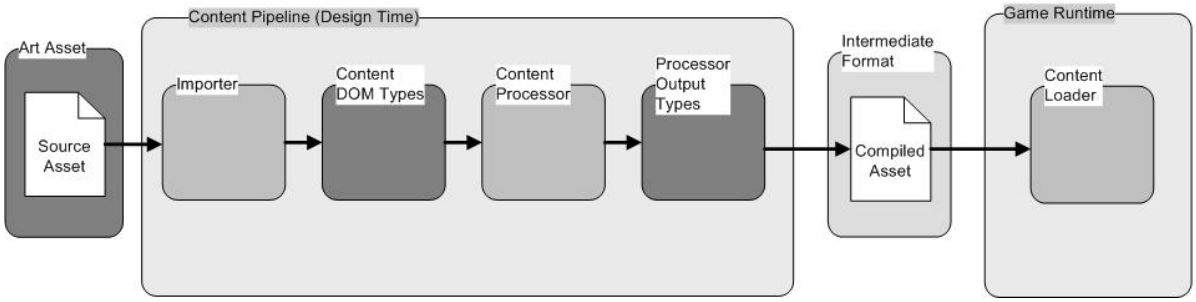


Diagrama del proceso de carga de assets en XNA.

El propio Visual Studio se encarga de, al generar una nueva solución de XNA, crear dos proyectos distintos, uno para la lógica de juego y otro para el contenido, donde a su vez crea una carpeta para los ficheros de audio, para los ficheros de imagen, etc. que el juego va a acceder posteriormente y añadirlos es tan sencillo como copiarlos en su carpeta correspondiente e indicárselo a visual Studio usando su navegador de archivos (o refrescar la carpeta).

Para acceder a ellos utilizamos el método load de la clase ContentManager, una instancia de la cual, Content se encuentra en la clase Game, por ejemplo:
Content.Load<Texture2D>(@"Images/textura")

Nótese que hace falta indicar el tipo de formato del asset a cargar (Texture2D para imágenes o SoundEffect para efectos de sonido), además de que no se indica la extensión del fichero. Esto se debe a que en XNA los assets se referencian mediante una cadena identificadora que por defecto es el nombre del fichero sin la extensión, no por el nombre de fichero.

Los formatos aceptados por XNA son limitados, aunque ofrece la posibilidad de añadir soporte para nuevos formatos. En este proyecto sin embargo se ha limitado su uso a los primeros, .png o .bmp para las imágenes y .wav para los efectos de sonido y música.

[18] https://msdn.microsoft.com/en-us/library/bb447745.aspx

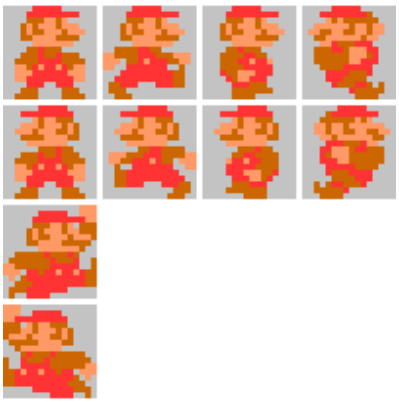
Sprites

Un sprite es un gráfico 2d usado para representar visualmente los elementos del juego. Objetos, enemigos o el personaje principal entre otros. Por ejemplo:

Sprite de Mario del juego Super Mario bros.

Sprite sheet

Los sprites se almacenan en una Imagen donde se guardan los diferentes frames que forman las animaciones de los elementos de un juego. Qué se guarda en cada sprite sheet puede variar según las preferencias de cada uno, pero una práctica extendida suele ser crear un sprite sheet por elemento (Por ejemplo, uno solo para el personaje principal), poniendo en cada fila los frames correspondiente a una acción o estado (Por ejemplo, estar quieto, correr o saltar):



Parte del sprite sheet de Mario del juego Super Mario bros.

Para facilitar su uso generalmente cada frame se guarda y separa del resto usando un tamaño normalizado, de forma que se pueden dibujar en pantalla haciendo uso del mismo atributo de posición.

Se considera por defecto el inicio del sprite su esquina superior izquierda, para acceder a un sprite tenemos que conocer dicha posición y el tamaño del sprite. Al ser los sprites de un tamaño fijo obviamente conocemos dicho tamaño y a partir de él obtenemos la posición del sprite:

Posición X en el sprite sheet = frame de la animación * ancho de sprite

Posición Y en el sprite sheet = ID del estado del sprite * alto de sprite

Dibujado de sprites

Para dibujar en pantalla se hace uso de la función Draw de dibujado de la clase SpriteBatch, que recibe los siguientes parámetros:

- La Texture2D donde hemos almacenado los sprites que queremos dibujar.
- Un Vector2 (Un vector de dos elementos, las coordenadas X e Y) donde indicamos la posición del objeto en el mundo del juego a partir de la cual se dibujará el sprite. En caso de que la cámara esté fija en la posición (0,0), la posición será la posición de la pantalla en la que se va a dibujar. Hay que tener en cuenta que se dibuja a partir del centro del sprite, que por defecto dicha posición es (0,0), la esquina superior izquierda de lo que se va a dibujar.
- Un Rectangle, el cual viene definido por cuatro enteros: Las coordenadas X e Y de la textura a partir de la cual vamos a dibujar, y el ancho y el alto de la región que queremos dibujar.
- Un Color, usado para pintar el sprite de un color especifico, usándose “White” para pintar usando el color original. Además, un color se puede multiplicar por un flota entre 0 y 1 para modificar la transparencia con la que se dibuja el sprite, siendo 1 opaco y 0 completamente transparente (y por tanto invisible en pantalla).
- Un flota que indica la rotación, en radianes, de lo que se va a dibujar.
- Un Vector2 para indicar el centro de giro. Por defecto es (0,0), la esquina superior izquierda, pero para facilitar el giro de objetos sobre si mismo se le debe pasar el centro del sprite que queremos dibujar y rotar.
- Un flota para indicarle la escala con la que se quiere dibujar, siendo 1 dibujar sin escalar, entre 0 y 1 para disminuir el tamaño y mayor de 1 para agrandar el sprite.
- Un SpriteEffects, el cual puede ser “None”, “FlipVertically” o “FlipHorizontally”, usados para voltear horizontal o verticalmente lo que se va a dibujar.
- Y para terminar, un float entre 0 y 1 que indica el orden de dibujado. Se puede escoger cual es delante y cual detrás, en el proyecto usado 0 es delante, y por tanto un sprite que tenga un valor menor que otro se dibuja por delante de este.

Todas las llamadas a la función de dibujado deben estar precedidas por una llamada a la función Begin del SpriteBatch que se va a usar, así como una llamada a su función End al concluir. Además, la función Begin puede recibir como atributo una variable del tipo “SpriteSortMode”, el cual puede tomar como valores “BackToFront”, “FrontToBack” e “Immediate” para indicar como se va a interpretar el valor de orden de dibujado comentado en el último punto de la lista anterior, y que se detalla a continuación:

Orden de dibujado

A la hora de dibujar los sprites en pantalla puede ser importante especificar cuales se dibujan encima de otros. Usualmente los funciones de dibujado sobrescriben el buffer de la pantalla, de forma que los primeros en ser dibujados quedan “al fondo” y los últimos por delante.
Existen soluciones para dibujar en el orden correcto los sprites, como ordenar las llamadas de dibujado, pero XNA provee de una mucho más sencilla, z-ordering.
En un entorno 3D, qué objetos se dibujan encima de los otros viene determinado por su distancia a la cámara, y de forma similar la función de dibujado de XNA permite recibir un valor de profundidad, layerDepth, que internamente usa para dibujar los sprites en el orden correcto.

Sin embargo, hay que tener cuidado al superponer dos sprites con un mismo layerDepth, pues en XNA cual se dibuja por encima tiene un comportamiento aleatorio. Para evitarlo se ha propuesto que estos objetos puedan tener una ligera variación en su layerDepth, de forma que nunca lleguen a coincidir.

Color key

Como puede observarse, lo que queremos dibujar no necesariamente ocupa toda la superficie del frame, y esa superficie sobrante se suele omitir al dibujar indicándole al framework lo que se conoce como color key. En el caso de XNA, el color key viene predefinido sin que sea posible cambiarlo, se trata del fucsia (indicado en RGB como 255, 0, 255). Internamente XNA pone el valor alfa de aquellos pixeles que sean de dicho color a 255, de forma que no se muestran en pantalla.

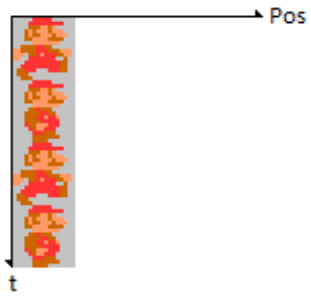
Animación

Para crear la ilusión de movimiento se usa la combinación de dos tipos de animación, la externa y la interna.

La animación externa es aquella en la que se mueve un objeto variando su posición:

Animación externa.

La animación interna es la animación intercalando diferentes frames de animación:



Animación interna.

Al juntar ambas, se crea la ilusión de movimiento:

Animación interna y externa.

Sin embargo, para conseguir un resultado adecuado hay que encontrar una buena relación en la velocidad de ambas animaciones. Por ejemplo, un error típico se produce cuando la diferencia de velocidad de ambas animaciones es muy grande, haciendo que el personaje parezca que “patine” sobre la superficie.
Destacar que uno de los dos tipos de animación puede ser suficiente. Por ejemplo, con la animación externa se podrían mover gotas de lluvia que no requieren de diferentes frames, y con la animación interna animar aquellos elementos que no se desplazan por la pantalla, como pueden ser elementos decorativos del HUD.

Para animar en XNA, usamos sus funciones de dibujado a las cuales se le pasa, aparte de la posición de la pantalla en la que se dibujará, la textura donde se ha almacenado el sprite sheet, así como la posición del tile sheet a partir de la cual se encuentra la porción de dicha textura que se va a dibujar y el tamaño de dicha porción (en anchura y altura).

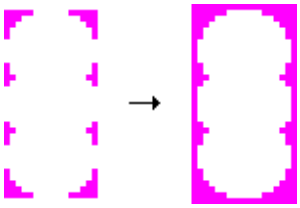
Controlar la velocidad de la animación interna es sencillo, basta con un contador que cada cierto tiempo seleccione el siguiente frame. Habiendo normalizado el tamaño de cada frame y conociendo su tamaño, acceder a cada uno de ellos es tan simple como multiplicar el tamaño de frame por el número de este. Y habiendo guardado en cada fila la animación de un estado diferente(Quieto, correr, saltar, etc.) podemos acceder a ellas de la misma forma.

Problema al dibujar sprites: Artefactos.

Uno de los problemas encontrados a la hora de dibujar sprites fue la aparición de artefactos:

En el caso concreto de XNA, la aparición de dichos artefactos puede ocurrir al dibujar un sprite en una posición no entera (en relación a la posición de la cámara), por ejemplo, al mover la cámara o los sprites con una velocidad cuyo valor sea no entero, o también al aplicar efectos de zoom.

La solución consistió en modificar el sprite sheet del objeto y separar cada frame por un pixel de color transparente:



Lo cual hace que los artefactos, en caso de haberlos, no sean visibles:

Gestión de la entrada

El jugador interactúa con el videojuego mediante una amplia variedad de dispositivos. Joysticks, mandos, teclados, ratones, cámaras, etc En el caso del proyecto la entrada se limita al uso de teclados y mandos.

XNA, mediante el framework Microsoft.Xna.Framework.Input, ofrece una manera de conocer el estado del ratón, así como del teclado y los mandos.

Conocer el estado de una tecla es sencillo:
```
KeyboardState keyState = Keyboard.GetState();
keyState = Keyboard.GetState();
keyState.IsKeyDown(Keys.Up);
```

Y en el caso de los botones del mando:
```
GamePad.GetState(PlayerIndex.One).DPad.Up;
```

En el caso de los sticks devuelve su posición (X e Y, con valores entre 0 y 1). Por ejemplo, para leer el estado del stick izquierdo:
```
GamePad.GetState(PlayerIndex.One).ThumbSticks.Left.X;
GamePad.GetState(PlayerIndex.One).ThumbSticks.Left.Y;
```

Y en el caso de los gatillos, devuelven un valor entre 0 y 1, dependiendo de la presión realizada. Por ejemplo, el gatillo izquierdo:
```
GamePad.GetState(PlayerIndex.One).Triggers.Left;
```

Sin embargo, solo podemos obtener si un botón (o tecla) está pulsado, y al requerir de otra información adicional se decidió crear una clase para gestionar la entrada.

El conjunto de botones dentro del juego que almacenamos es un subconjunto del que nos ofrece el teclado o el mando. Cada botón dentro del juego tiene asociado una tecla del teclado y un botón del mando, pudiendo ambos modificar la información de un botón virtual. Sin embargo, el mando tiene preferencia sobre el teclado, de forma que si hay un mando conectado el juego se controla con el mando, y si no lo hay se controla con el teclado.

Se puede comprobar si hay un mando conectado de la siguiente manera:

```
GamePad.GetState(PlayerIndex.One).IsConnected;
```

La información requerida de cada botón es la siguiente:

- Estado actual.
- Estado previo.
- Se acaba de pulsar.

- Se deja de pulsar.

El estado actual indica si está pulsado o no. El estado previo, el estado en la iteración anterior.
También guardamos si se acaba de pulsar, es decir, si en la iteración actual está pulsado pero no lo estaba en la anterior, y si se deja de pulsar, es decir, si en la iteración actual no está pulsado pero sí lo estaba en la anterior.
Toda esta información se ha encapsulado en una estructura llamada KeySet (conjunto de teclas), de forma que se comprueba al inicio de cada iteración del bucle principal y se envía posteriormente a los objetos que la requieran.

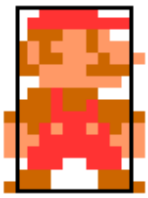
Además, el input manager también se encarga de discretizar y almacenar la dirección del stick izquierdo para permitir y facilitar su uso para la navegación de menús. El conjunto de direcciones discretas es de 4 (Las cuatro direcciones básicas que se consiguen de la cruceta digital necesarias para la navegación por menús). El proceso de discretización es sencillo, asumimos que el stick está apuntando en una dirección cuando su posición en el eje vertical u horizontal es mayor a un valor mínimo establecido. Por ejemplo, si la posición Y del stick es mayor de ese mínimo, se está pulsando “arriba”.
En caso de que dos posiciones estén siendo pulsadas según dicho criterio, se ha optado por quedarse únicamente con una de ellas (La posición derivada de la coordenada, X o Y, de mayor módulo).

Para acabar, comentar que aunque se gestionan tanto el teclado como el mando y se puede reutilizar para cualquier otro proyecto, en este en concreto solo se puede jugar adecuadamente usando un mando al requerir de un stick.

Colisiones

Algunos de los elementos del juego pueden colisionar entre ellos, por ejemplo, las balas de la protagonista con las naves enemigas. Para comprobar las colisiones existen diferentes técnicas, siendo la colisión por cajas y la colisión por píxel las más usadas en los juegos en 2D del género.

La colisión por cajas consiste en asignar una o más cajas de colisiones a los objetos, y calcular la colisión entre estas cajas. Al asignar las cajas se trata de ajustar a la forma irregular del sprite una forma sencilla (un cuadrado o un círculo, por ejemplo), simplificando el problema de las colisiones.



Ejemplo de caja de colisiones rectangular para el sprite de Mario.

A un mismo objeto se le pueden asignar varias cajas, ya sea porque una única forma básica no es suficiente o porque diferentes partes del objeto tienen comportamientos diferentes. Un ejemplo de esto último sería un enemigo al que puedes dañar y destruir diferentes partes, de forma que cada parte necesita su propia caja de colisiones.

Para calcular la colisión entre dos cajas rectangulares, las únicas usadas en el proyecto, se usa la siguiente fórmula:

```
if(A.x + A.cajaAncho >= B.x + B.cajaInicioX && A.x + A.cajaInicioX <= B.x + B.cajaAncho &&
A.y + A.cajaAlto >= B.y + B.cajaInicioY && A.y + A.cajaInicioY <= B.y + B.cajaAlto)
        colision = true;
else
        collision = false;
```

En algunos casos en los que se busca una mayor precisión se emplea la técnica de colisión por pixel, que consiste en detectar pixel a pixel (o pixel con una caja de colisiones) si dos objetos colisionan entre sí. Aún así, se trata de una técnica que requiere de un gran coste de cálculo, así que en la práctica, de usarse, primero se descarta la colisión mediante la técnica de caja de colisiones. Por ejemplo, se calcula la colisión del personaje con todas las balas enemigas, y únicamente si se detecta colisión con una, se mira la colisión por píxel entre dicha bala y el personaje.

Personaje controlado por el jugador

El jugador controla un único personaje, una nave espacial. La especificación de dicho personaje se encuentra en el anexo, la implementación de la cual se detalla a continuación.

Sprite de la nave controlada por el jugador.

Desplazamiento

El desplazamiento de la nave puede ser en cualquier dirección (360º) y a diferentes velocidades. Para conseguir esto se hace uso de uno de los sticks del mando (concretamente el izquierdo), ya que permite la entrada de la dirección (stick analógico de 360º, como el movimiento que se busca permitir realizar) y diferentes grados de inclinación para cada dirección, lo cual se usa para controlar la velocidad del personaje.

En XNA el estado del stick viene dado por coordenadas dentro un círculo unitario. Así, en reposo el stick se encuentra en la posición (0,0), y si se inclinase por completo hacia la derecha, se encuentra en la posición (1,0). De esta forma, la velocidad de la nave en el juego está controlada por la posición del stick en cada coordenada, en tanto que esta representa la inclinación del stick. A más presión del jugador sobre el stick, más inclinación y, por tanto, más velocidad.
Sin embargo, se ha establecido una mínima inclinación del stick a partir de la cual se empieza a mover la nave, para evitar que esta lo haga debido a posibles leves inclinaciones causadas por presión ejercida por el pulgar del jugador al reposar sobre el stick.
Además, la velocidad máxima a la que puede moverse la daba es superior al máximo valor de inclinación (1), así que este se extrapola de [0,1] al rango [velocidad mínima, velocidad máxima].

Disparo

Al contrario que el movimiento analógico de la nave, se ha discretizado las direcciones a las que puede disparar, limitándose a 8 (arriba, abajo, izquierda, derecha y las cuatro diagonales entre ellas).
El jugador escoge en qué dirección quiere disparar usando los cuatro botones posteriores, los cuales generalmente se disponen formando un “rombo”, quedando uno arriba, uno abajo, otro a la izquierda y el último a la derecha. Así, cada uno de estos botones sirve para disparar en la dirección en la que se encuentran. Por ejemplo, con el botón superior el jugador puede disparar hacia arriba, y con el derecho hacia la derecha, además de poder pulsar dos botones adyacentes para disparar en diagonal, por ejemplo, pulsando los botones superior y derecho dispara en diagonal la diagonal formada por ambos botones, arriba hacia la derecha.
Debido a la dificultad de pulsar dos botones al mismo tiempo al tratar disparar en diagonal, se ha dejado un tiempo, muy pequeño, desde que se pulsa uno de los botón de disparo hasta que la nave dispara en el juego, evitando que al disparar en diagonal la mayoría de veces dispare unas pocas balas en una dirección que no es la deseada.

Una vez se ha pulsado un botón de disparo, la nave dispara. Esto implica que desde que la nave empieza a disparar hasta que se suelta se crean balas en un intervalo predeterminado (dos balas en paralelo en cada intervalo) cuya dirección es la dirección de disparo descrita anteriormente, sobre las cuales el jugador ya no tiene control alguno y sobre las que se entra en detalle más adelante.

Además, hay que tener en cuenta que al disparar en una dirección se rota el sprite de la nave protagonista para que encare dicha dirección, la cual no cambia hasta que no se dispare en otra dirección.

Escudos, absorción y sobrecalentamiento

La nave cuenta con dos escudos, de color verde y azul, usados para absorber las balas de su mismo color al entrar en contacto con ellas y evitar el daño.

Al pulsar el botón asignado a un escudo, este se activa. Aparece alrededor de la nave protagonista suavemente, de ser invisible a totalmente visible, efecto para el cual se ha usado la función de dibujado Draw descrita en el apartado de dibujado de sprites. Básicamente, el parámetro Color indica el color del que se tinta la imagen a dibujar (White para usar el color original), y si este parámetro es multiplicado por un float entre 0 y 1 se puede alterar la transparencia de lo que se dibuja.
De esta forma multiplicamos dicho Color por una variable de tipo float donde almacenamos el nivel de transparencia, los valores de la cual parten de 0 (o el valor actual de la transparencia, si dejamos de pulsar y los encendemos rápidamente antes de que se apaguen) al encender el escudo hasta que alcanzan su valor máximo 1. Al soltar los escudos se apagan, con un efecto de desaparición opuesto al mencionado.
Los escudos se guardan como un sprite que almacena el contorno de la nave en el color del escudo, de forma que basta con dibujarlo en la misma posición en la que se dibuja la nave.

Como dice la especificación del juego, si se mantiene encendido el escudo cierto tiempo la nave se sobrecalienta. Se avisa poco antes de que esto ocurra cambiando el color del escudo a rojo (además de contar con una señal sonora).

En caso de que se sobrecaliente no se permite volver a activar los escudos hasta que pase el efecto, el cual visualmente se ha representado “tintando” el sprite de color rojo usando la funcionalidad que para ello ofrece la función Draw descrita en el apartado de dibujado de sprites, concretamente haciendo uso del parámetro Color, asignando Color.Red.

Teletransporte

Otra de las habilidades con las que cuenta el jugador es la posibilidad de teletransportarse una distancia fija en cualquier dirección, esquivando los elementos que haya entre la posición original y la posición destino. La nave se teletransporta en la dirección del movimiento, por lo tanto al igual que el desplazamiento básico se trata de un movimiento que puede realizarse en cualquiera de los 360º. Sin embargo, al tratarse de una distancia fija, esta no depende de cuando se presione el stick, con lo cual hay que sacar el vector unitario que define la dirección del punto en el que se encuentra el stick. Para ello es suficiente con dividir cada componente

del par de componentes X e Y que definen la posición del stick por el módulo del vector que va del origen a dicho punto.

Efecto visual al teletransportarse.

Visualmente se ha creado un efecto de teletransporte que consiste en que la nave deje un rastro, intentando simular un efecto de desplazamiento a gran velocidad. Dicho rastro consiste en una serie de impresiones en pantalla del mismo sprite de la nave, pero más transparentes cuanto más lejos se encuentran.
Las posiciones de estas impresiones, al encontrarse en la misma línea que va del punto original al punto destino de teletransporte se calculan, pues, de la misma forma que se calcula este último, pero decrementando el valor de la distancia de teletransporte por el intervalo que separa las diferentes impresiones (Así como el propio valor de la transparencia).

Sombra

Existe un efecto meramente visual, que no repercute en la jugabilidad, que consiste en la proyección de una sombra sobre el suelo o los elementos que se encuentren por debajo de la nave protagonista. Dicha sombra consiste simplemente en un sprite de color oscuro que se imprime de forma semitransparente sobre los elementos correctos. Dichos elementos son aquellos que, al encontrarse por debajo de la nave, se dibujan con un valor de profundidad (o layer depth) mayor, y por tanto automáticamente XNA se encargará, al ordenar e imprimir en pantalla, de dibujar la sombra únicamente sobre estos.
La sombra se mueve por una subregión de la pantalla, la cual se encuentra en el centro y tiene la misma forma de la pantalla de juego (equidistando los lados de esta subregión de los lados de la pantalla). Dicha posición depende de la posición de la nave, conociendo esta sacamos la posición de la sombra con la siguiente fórmula:
Sombra.pos.X = subregión.posInicial.X + nave.pos.X * (subregión.ancho / pantalla.ancho)
Sombra.pos.Y = subregión.posInicial.Y + nave.pos.Y * (subregión.alto / pantalla.alto)

Fondos

Qué es un mapa o nivel en un juego depende de cada juego, de su definición y requisitos. En este proyecto no existe interacción entre los objetos (Personaje principal, enemigos, etc.) y el escenario, de forma que se han tratado de forma independiente, reduciendo el concepto de escenario al de fondo, un elemento puramente estético.

Existen diferentes opciones a la hora de crear fondos para videojuegos: Imágenes fijas, fondos 3D, fondos hechos a base de tiles, fondos con diferentes planos de Scholl, entre otras.
En este proyecto se han usado las técnicas de mapa de tiles y la generación procedural.

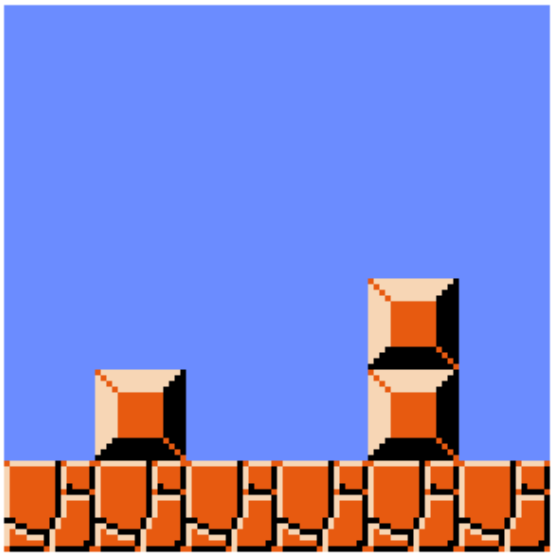
Mapa de tiles

Los mapas creados a partir de tiles tienen su origen en la necesidad de almacenar una gran cantidad de mapas sin que repercuta negativamente en el peso del juego.

Las tiles son pequeñas piezas de mapa de un tamaño generalmente fijo que se almacenan una única vez en un tile set y pueden ser usadas tantas veces como haga falta para formar mapas más complejos, los cuales se definen generalmente como una matriz de enteros, cada uno de los cuales es el identificador de la tile a usar, el cual sirve para encontrar dicha tile dentro del tile set. Por ejemplo:

Tile set

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	2	0
0	2	0	0	2	0
1	1	1	1	1	1



A la izquierda vemos el mapa como una matriz y a la derecha al ser dibujado.

Acceder a la tile correspondiente en el tile set en función de su identificador es sencillo.
Suponiendo un tamaño de tile tamañoTile X tamañoTile, y un tile set con tilesAncho tiles de ancho, accedemos a la posición origen de la tile a partir de la cual se va a dibujar con la siguiente fórmula:
tileX = (tileID % tilesAncho) * tamañoTile
tileY = (tileID / tilesAncho) * tamañoTile

Animación

El mapa no se encuentra fijo, sino que se desplaza para simular el movimiento a través del mismo. Normalmente la porción de mapa que se dibuja depende de la cámara, que a su vez suele depender de la posición del jugador. En el caso de los shoot 'em up, sin embargo, la cámara no suele estar fija sobre el jugador sino que se desplaza de forma automática y constante.
Así, a cada instante actualizamos la posición del mapa en función de la cámara, y dibujamos el trozo de mapa que se va a mostrar en pantalla, evitando así el coste innecesario de dibujar lo que queda fuera de cámara.
Escoger qué trozo de mapa en función de la cámara es sencillo, suponiendo un tamaño de tiles de tamañoTile X tamañoTile, dibujaríamos tantas tiles como cupiesen en pantalla (anchoPantalla / tamañoTile, altoPantalla / tamañoTile) a partir de la posición de la matriz de tiles camaraX % tamañoTile, camaraY % tamañoTile.

Lectura de un fondo de fichero

Los mapas usados han sido almacenados en un fichero de texto y parseados al ser leídos y usados por el juego.
El formato es simple, la primera línea del fichero mapa contiene dos enteros con el ancho y alto del mapa en tiles, y a partir de la siguiente línea se encuentra la matriz de identificadores de tile que definen dicho mapa.

Fondo procedural

La generación procedural[19] hace referencia a la generación de contenido, en este caso el fondo, no de forma manual sino mediante algoritmos. Otra forma de explicar la generación procedural es como "aleatoriedad controlada".

Dicha técnica se ha aplicado a la creación de un fondo para el juego, creando un cielo a partir de diferentes nubes. Dichas nubes pueden moverse a diferentes velocidades dependiendo de su altura, simulando así su mayor o menor distancia respecto a la cámara. Además, las nubes más cercanas (y por tanto las que se dibujan más grandes y a mayor velocidad) se dibujan los objetos del juego y de forma transparente sobre estos, de forma que solo se aplique el efecto de transparencia, simulando entrar en la nube, a los elementos del juego y no al fondo.

Así, a cada tipo de nube se le ha asignado una probabilidad y un tiempo aleatorio entre nubes, de forma que aunque se vayan creando de forma aleatoria en tiempo de ejecución las cantidades y posiciones se mueven dentro de unos parámetros que hacen que el fondo

[19] https://en.wikipedia.org/wiki/Procedural_generation

siempre luzca correctamente, sin escasez ni exceso de nubes, ni estas apareciendo siguiendo patrones no deseados.

Además, se han usado ambas técnicas conjuntamente para formar mapas con mayor detalle. En concreto, usando la técnica del mapa de tiles se ha creado el fondo de un bosque y por encima de este se generan proceduralmente las nubes.

Niveles

Para probar la funcionalidad implementada, se han creado dos niveles (sin contar los niveles de los jefes), uno generado proceduralmente y otro creado a mano.

Como se ha mencionado en el apartado anterior, en este proyecto en concreto no existe interacción entre los elementos del juego, como puedan serlo el propio jugador o los enemigos, y el escenario, y por tanto se han almacenado por separado, permitiendo además cualquier combinación de ambos.

Así pues, en este proyecto un nivel se define exclusivamente por la serie de enemigos que lo forman, y van apareciendo secuencialmente en pantalla.

Nivel procedural

El primer nivel se encarga de crear proceduralmente los obstáculos que forman dicho nivel. Concretamente, filas o columnas de enemigos o balas de diferentes colores se escogen aleatoriamente con un intervalo de tiempo regular entre ellas y aparecen de cualquier extremo de la pantalla.
La ventaja de crear niveles generados proceduralmente es la rapidez con la que, a base de unas pocas reglas, se pueden crear un número infinito de niveles de longitud infinita. Sin embargo, existe el riesgo de que el nivel resulte soso y no cree niveles interesantes, para lo cual no suele bastar con unas pocas reglas sino que se debe profundizar y entender correctamente qué patrones puedan dar lugar a niveles interesantes y como mezclarlos, así como controlar la curva de dificultad.

Nivel creado a mano

El segundo nivel de prueba se ha creado a mano. Para facilitar su creación y modificación y debido a la limitación de tiempo, en vez de leerse el nivel de un fichero y a pesar de que existe dicha funcionalidad, el nivel está creado dentro del propio código.
La desventaja es el tiempo que requiere crear un nivel así, además de que el esfuerzo invertido resulta en un nivel finito, además de la dificultad que a veces puede acarrear realizar modificaciones sobre el mismo, al tener que reestructurar todo el nivel. Sin embargo, al tener control total de los elementos que forman el nivel y como se desarrolla este, es más fácil controlar y crear tramos interesantes, evitar la repetición y gestionar la curva de dificultad.

Subsistema de audio

El proyecto requiere de una forma de gestionar los efectos de sonido y pistas de audio, así como contar con diferentes formas de controlar su reproducción y aplicar efectos. Dicha funcionalidad ha sido encapsulada en un manager de audio.

XNA ofrece dos alternativas para implementar el apartado sonoro de un juego:

XACT(Cross-platform Audio Creation Tool):
Se trata de un engine de sonido que ofrece una serie de funcionalidades ya implementadas, como efectos, crear colas de sonidos y asignarles prioridad a cada elemento de la misma, comprimir pistas, etc que pueden ser utilizados desde una aplicación independiente al código.

También cuenta con una api, incluida en Xna.Framework.Audio, con una serie de clases y funciones que permiten realizar acciones básicas como reproducir sonido, pausarlo o cambiar el nivel de volumen. Es esta alternativa la que se utilizará en el desarrollo de este proyecto.

Este framework consta de una clase básica, SoundEffect, para almacenar una pista de audio .wav, que únicamente puede ser reproducida usando el método de clase Play();

```
SoundEffect soundEffect;
soundEffect = Content.Load<SoundEffect>(@"Audio/fichero");
soundEffect.Play();
```

Para tener control de la reproducción esa pista debe ser instanciada en una variable de tipo SoundEffectInstance, que además de reproducir la pista de audio permite pausar, parar, reanudar, "loopear" y controlar el nivel de volumen durante la reproducción de la misma.
Para instanciar y reproducir usamos el método CreateInstance() de la clase SoundEffect.

```
SoundEffectInstance instance;
instance = Content.Load<SoundEffect>(@"Audio/fichero").CreateInstance();
instance.Play();
```

La principal ventaja en cuanto a funcionalidad de SoundEffect respecto a SoundEffectInstance es que esta última no permite reproducir su sonido varias veces de forma paralela.
Pongamos de ejemplo el sonido al disparar una metralleta. Usando un único sonido de disparo, y si se dispara una rafaga SoundEffect nos permitiría reproducir el sonido tantas veces como disparos, pero al usar SoundEffectInstance se debería crear tantas instancias como disparos, lo cual resulta innecesario.

Añadir que XNA también cuenta, en Xna.Framework.Media, con las clases Song y la clase MediaPlayer, usadas para almacenar pistas de audio en .mp3 y para reproducirlas, respectivamente.

```
Song song;
```

```
song = Content.Load<Song>(@"Audio/prueba");
MediaPlayer.Play(song);
```

La principal desventaja es que solo puede reproducirse únicamente una pista de audio al mismo tiempo (normalmente se reserva su uso para reproducir música de fondo), limitación que no existe en las clases SoundEffect y SoundEffectInstance. En el proyecto se usarán únicamente estas dos últimas, debido a la libertad de de reproducir diferentes pistas de audio de fondo al mismo tiempo (por ej, música de fondo y lluvia).

El subsistema de audio implementado se encarga de todo lo relacionado con el audio del juego y se compone de las siguientes clases:

AudioManager

La clase principal encargada de gestionar y manejar el resto de clases, así que como de ofrecer una serie de métodos de control de las diferentes pistas de audio.

Funciona como una caja negra que puede ser usada por los demás objetos del juego, los cuales le indican que acción quieren reproducir en un sonido en concreto de los almacenados por el manager.

Para permitir esto se ha realizado la siguiente implementación:

Debido a la pequeña cantidad de sonidos que requiere el juego, en vez de cargarlos y liberarlos según sea necesario se dejan cargados desde el inicio. Así, la clase AudioManager cuenta con un vector para sonidos de tipo SoundEffect y uno para los sonidos de tipo SoundEffectInstance:

El vector usado para almacenar los SoundEffect es únicamente del tipo soundData, que contiene el SoundEffect y variables para el control de su cadencia, en los que se profundizará más adelante. Para controlar la reproducción de los mismos se cuenta únicamente con un método:

- playSound(id, vol, pan):
  Reproduce el sonido id (coincide con su posición en el vector de SoundEffects) con un volumen (relativo al volumen global) indicado por vol y un panorama indicado por pan. Vol es un valor entre 0 y 1000 que se normaliza al intervalo 0..1 que maneja unity, y panorama un valor entre -1 y 1 que indica la posición del sonido. Por ejemplo, 0 hace que suene centrado (mismo volumen por ambos altavoces en estéreo), mientras que -1 hace que únicamente suene por el izquierdo.

El vector para almacenar los sonidos de tipo SoundEffectInstance es ligeramente más complejo y requiere de una estructura auxiliar debido a la posibilidad de contar con más opciones para su reproducción y a la necesidad de guardar cierta información relativa al estado y control de cada instancia. Dicha estructura se ha llamado instanceData, en la cual almacenamos:

- La instancia del sonido.
- El volumen actual.
- El volumen máximo con el que queremos que se reproduzca.
- Si hay efecto de “fade”, y si este es de entrada o salida. El efecto de “fade” se utiliza para que la reproducción se inicie o acabe partiendo del silencia y variando suavemente el volumen para que no resulte abrupto.
- La velocidad del efecto de fade (Es el incremento o decremento del volumen por iteración).
- Si tras un efecto de fadeOut (silenciar) queremos que la reproducción para o no.
- Si se ha pausado debido a que el jugador a pausado el propio juego.

Para controlar su reproducción se ha provisto a la clase AudioManager de los siguientes métodos:

- playInstance(id,fadeSpeed)
  Si no está sonando actualmente, reproduce la instancia id además de inicializar las variables del efecto de fade in. Si fadeSpeed tiene como valor 1, no hay efecto de fade in y el sonido toma desde el inicio su valor máximo.
- pauseInstance(id)
  Pausa la reproducción de la instancia id.
- resumeInstance(id)
  Reanuda la reproducción de la instancia id.
- stopInstance(id,fadeSpeed)
  Se encarga de iniciar las variables del efecto de fade out o, en caso de que fadeSpeed sea -1, de parar la reproducción inmediatamente. A diferencia de la función fadeOutInstance (ver a continuación) con stopInstance sí se para la reproducción tras completarse el efecto de fade out, lo cual indicamos poniendo a true la variable stopAfterFade de la clase SoundInstance. La diferencia es que en este caso si queremos reanudar la reproducción tendremos que hacerlo desde el principio de la pista.
- fadeOutInstance(id,fadeSpeed)
  Inicializa los parámetros del efecto de fade out. A diferencia de la función stopInstance, con esta la instancia no para su reproducción, sino que continúa (aún sin sonar) hasta que se decida aplicar un efecto de fade in con fadeInInstance (ver a continuación).
- fadeInInstance(id,fadeSpeed)
  Inicializa los parámetros del efecto de fade in para una instancia que está siendo reproducida.
- setVolumeInstance(id,vol)
  Se encarga de modificar el nivel de volumen de la instancia ID con un valor entero comprendido entre 0 y 1000 que normaliza a un valor entre 0 y 1.

Efecto de fade

El efecto de fade, en sus dos variantes (fade in o de entrada, y fade out o de entrada) se realiza en tiempo de ejecución calculando en cada iteración el nuevo nivel de volumen a partir de los parámetros indicados.

Así, por ejemplo, si en vez de parar la reproducción de una pieza de música (cuando el jugador pierde la partida, por ejemplo), en vez de hacerlo de golpe permitimos que el programador aplique un efecto de fade out, de forma que antes de parar su reproducción el volumen de esa pista decrece gradualmente hasta silenciarse.

Cadencia de sonidos

Existe un problema en la reproducción de sonidos, especialmente en los SFX que se repiten rápidamente, debido a que su reproducción se superpone y no suenan correctamente. Por ejemplo, el sonido de una metralleta que se repite constantemente mientras el jugador pulsa el botón de disparo.
La solución es sencilla, limitar la cadencia con la que un sonido en particular puede repetirse. Implementarlo es sencillo cuando un único objeto puede reproducir dicho sonido, en tanto que ese objeto es el único encargado de controlar la cadencia, pero se complica ligeramente cuando diferentes objetos pueden reproducir un mismo sonido. Para solucionarlo, cada sonido (almacenado en la estructura soundData) cuenta con una serie de atributos auxiliares:

- Una variable indica si existe control o no de la cadencia en dicho sonido.
- Tiempo de espera, indica cuanto el tiempo durante el que no se permite volver a reproducir el sonido.
- Otra variable indica si el sonido se permite poner en cola para reproducirlo una vez acabe el tiempo de espera.
- Indicamos también cuánto tiempo permitimos que un sonido esté en cola, o dicho de otra forma, solo ponemos en cola los que estén a dicho tiempo o menos de que venza el tiempo de espera. Esto se debe a que nos interesa evitar que un sonido suene mucho más tarde de lo que realmente debería, y solo permitimos los que están dentro de un margen de tiempo.
- Y para acabar, el timer que guarda cuánto tiempo ha transcurrido desde su reproducción.

Menús

Para la creación de los menús del juego se ha partido de cero. El objetivo es crear un sistema que permita navegar por una serie de pantallas con diferentes opciones gracias a las cuales el jugador pueda modificar ciertos parámetros y escoger entre diferentes modos de juegos.

Diseño

Se ha definido el menú como una serie de opciones que se agrupan por pantallas.

Una opción del menú puede ser de tres tipos:

- De acción, al pulsar el botón de aceptar se ejecuta una acción, que puede ser navegar a otra pantalla o, por ejemplo, empezar la partida o salir del juego.
- De selección de valor, mediante los botones izquierdo y derecho permite al jugador escoger el valor de un parámetro. Por ejemplo, seleccionar el nivel, el fondo, idioma, etc
- Información. No puede ser seleccionada y simplemente aparece como información extra. Por ejemplo, la frase “¿Estás seguro?” en la pantalla de borrado de datos acompaña a las opciones “sí” y “no”.

Visualmente, una opción es una cadena de texto en pantalla que se resalta al ser seleccionada.

Para almacenar la información relativa a una opción se ha creado la estructura MainMenuOption, que cuenta con los siguientes atributos:

- Posición donde se va a comenzar a dibujar en pantalla, hacia la derecha, la cadena asociada a la opción.
- La cadena que define el texto de la opción, en la que se profundizará más adelante.
- El identificador único de esa opción para facilitar acceder a su acción asociada.
- Si es una opción de tipo de selección de valor y, en caso de serla, la serie de valores que puede tomar. Por ejemplo, la opción de selección de fase tiene los valores “Fase 1”, “Jefe 1”, “Fase 2”, etc Además, se guarda el valor seleccionado actual.

Una pantalla (Screen en el código) es un conjunto de opciones que se muestran en pantalla a la vez. Por ej, una pantalla “opciones” podría contar con las opciones “idioma”, “dificultad” y “controles”. El jugador se puede mover entre ellas con las flechas de dirección arriba y abajo.

Para almacenar la información relativa a una opción se ha creado la estructura MenuScreen, que cuenta con los siguientes atributos:

- Número total de opciones de esa pantalla y cuantas se pueden seleccionar. Las que no se puedan seleccionar, por tanto, se usarán para mostrar información.
- Un vector de tipo MainMenuOption donde almacenar las opciones que forman dicha pantalla.
- El identificador de la pantalla.

- El identificador de la pantalla que la precede, de forma que se pueda volver a ella al pulsar el botón de cancelar.
- El identificador de la opción usada para volver, de forma que esta pase a ser la opción seleccionada al pulsar el botón de cancelar. Por ejemplo, si mediante la opción “settings” pasamos a la pantalla de opciones y podemos movernos entre ellas, pero indistintamente de qué opción haya resaltada al pulsar el botón de cancelar, al volver la opción que debe de estar seleccionada es “settings”.

Para controlar las acciones asociadas a las diferentes opciones se ha empleado un switch que selecciona la ejecución de la función correspondiente. Por ejemplo, la opción con identificador 1 puede tener como cadena “New game”, y su acción asociada es preparar y comenzar la partida.

Fuente

XNA provee con unos pocos tipos de fuentes y tamaños para imprimir texto en pantalla. Sin embargo, para el proyecto se ha optado por una fuente personalizada almacenada en un sprite set.

Así, cada símbolo tiene un identificador que indica su posición en el sprite set y se usa para dibujar correctamente las cadenas en pantalla. Debido al poco texto del juego (se limita a los menús) simplemente se ha creado una pequeña aplicación aparte que transforma cadenas de texto en cadenas de identificadores para ser usadas posteriormente en el juego, en vez de permitir su procesamiento por parte del propio juego durante su ejecución.

Font sheet del juego Megaman, de Capcom.

Simulación discreta desacoplada

RT-DESK

RT-DESK es un núcleo de simulación de aplicaciones gráficas en tiempo real que sigue un paradigma discreto desacoplado. RT-DESK son las siglas de Real Time Discrete Event Simulation Kernel. Se trata de un núcleo de gestión de eventos discretos, ordenados temporalmente, mediante paso de mensajes.

Uno de los principales objetivos de RT-DESK es el cambio de gestión de eventos continuos a discretos. Con esto se deja de muestrear a una frecuencia continua para pasar a declarar eventos discretos que se ejecutarán pasados x segundos. De esta forma todo el tiempo entre que pasa dicho periodo se puede reaprovechar para realizar otras tareas en la CPU.

RT-DESK permite simular eventos continuos creando eventos a una frecuencia constante así como variar esta frecuencia según las necesidades del sistema. Así, en un mismo sistema pueden coexistir simultáneamente comportamientos discretos y comportamientos continuos, incluso dentro del mismo objeto. Esta capacidad permite utilizar RT-DESK dentro de un motor de videojuegos de forma que aumentaría notablemente las capacidades iniciales del motor.

Cabe recalcar que RT-DESK no es una aplicación y no funciona de forma aislada, se trata de un núcleo de simulación que hay que vincular con otro programa o motor gráfico. Una vez integrado este núcleo dentro del motor será capaz de encargarse de la gestión de eventos del sistema y de la comunicación entre objetos mediante paso de mensajes, limitándose a realizar la entrega.

RT-DESK se proporciona como una API desarrollada en C++ (A pesar de existir una versión en C# en pruebas, usada en este proyecto), una serie de ficheros de cabecera y de librería, así como un fichero de cabeceras que permite configurar tipos de mensajes personalizados, la cual puede ser incorporada a cualquier aplicación gráfica en tiempo real.

Evolución de RT-DESK

A partir de una tesis de máster, se desarrolló en C++ un simulador de eventos discretos orientado a simular sistemas mediante teoría de colas. A este simulador se le realizó un ajuste para incluir el tiempo real. El resultado fue una tesis doctoral en la que se aplicaba este núcleo sobre un motor de juegos.

Posteriormente se realizó una reestructuración de todo el código y surge RT-DESK 1.0. Se ha estado probando sobre entornos más actualizados con el fin de validar la vigencia del API, y de dichas pruebas surgen una serie de mejoras que desembocan en la versión actual, RT-DESK 2.0, con las siguientes mejoras:

- Optimiza la gestión de eventos y la gestión de memoria.
- Permite crear nuevos tipos de mensajes.

- Permite los mensajes privados o propietarios.

Paralelamente, diferentes proyectos trataron de portar RT-DESK al lenguaje C#, necesario para su uso en XNA o el motor Unity. La versión usada en este proyecto surge de uno de esos proyectos, *Creación de un videojuego en XNA utilizando RT-DESK*, de David Pérez Climent.

Actualmente diferentes proyectos siguen validando la tecnología y tratando de depurar y mejorar la última versión, así como seguir trabajando en la adaptación al motor Unity.

Funciones principales

- Proporciona las estructuras de datos y las funciones necesarias para modelar el mecanismo de envío/recepción de mensajes.
- Gestiona el envío de mensajes.
- Mantiene los mensajes enviados, y que todavía no han sido recibidos, ordenados por tiempo de ejecución.
- Entrega los mensajes en el periodo de tiempo indicado.

Además RT-DESK aporta un tipo de mensaje básico y sencillo, el cual solo esta formado por el objeto que lo envía, el objeto que lo recibe y el tiempo en el que debe de ser ejecutado a partir del envío. Para optimizar el uso de memoria en tiempo de ejecución dispone de un pool de mensajes, el cual puede hacerse mayor durante la ejecución si hubiera más demanda de mensajes.
Además, a partir de la versión 2.0 los mensajes pueden ser propietarios. Es decir, si un objeto se va a reenviar a si mismo un mensaje durante toda la ejecución, simplemente lo crea en su constructor y lo marca como privado, evitándose muchas llamadas al pool.
Cada tipo de mensaje tiene su propio pool. Si es necesario un mensaje más completo, el programador simplemente crea una clase nueva de mensaje que herede del básico y así puede añadir la información que necesite. Luego solo hay que decirle a RT-DESK cual es el nuevo tipo de mensaje creado y el generará un pool de mensajes de este tipo.

El bucle principal en RT-DESK

En bucle principal descrito en el apartado “estado del arte” todos los objetos se actualizan y dibujan a una misma frecuencia. Sin embargo, no todos los objetos requieren de la misma frecuencia de actualización, y en la práctica muchos objetos se actualizan más de lo que realmente requerirían, el cual es un tiempo desperdiciado que, en caso de necesidad, podría ser usado por objetos que sí requieren de él para actualizarse y mantener una velocidad de ejecución constante.
Algunos juegos intentan paliar esto asignando diferentes frecuencias de ejecución a diferentes elementos (por ejemplo, físicas a 30fps, IA a 15fps, etc.), pero está lejos de lo que permite RT-DESK, que es permitir gestionar y asignar una frecuencia diferente a cada uno de los objetos del juego según las necesidades de este, incluso a distintas funciones del mismo (Como actualizar y dibujar, por ejemplo).
En el paradigma discreto desacoplado cada objeto de la simulación se ejecuta de forma independiente y, como se ha mencionado, con la frecuencia que requiera. Se consigue así usar

la CPU cuando este sea necesario, no usando un tiempo que queda libre para otros menesteres como puede ser la incursión de más objetos o en tiempo de GPU, entre otros, o simplemente liberarlo para reducir el consumo.

Acople de RT-DESK y XNA

Este proyecto usa RT-DESK en forma de DLLs, las cuales se incluyen en el proyecto permitiendo acceder a toda la funcionalidad de RT-DESK.
De la creación de estas DLLs se encargó David Pérez Climent en su proyecto "creación de un videojuego en XNA usando RT-DESK", en cuya memoria describió el proceso, el cual a continuación se resume brevemente.

Adaptación C# y C++

Originariamente RT-DESK se proporcionaba en forma de una biblioteca escrita en C++ compilada a código máquina. XNA, sin embargo, únicamente trabaja usando el lenguaje C#, el cual compila a código intermedio CIL(Common Intermediate Language[20]), ejecutado por la CLR (Common Language Runtime[21]), impidiendo que librerías de C++ sean llamandas directamente desde C#.

C# permite llamar a funciones de bibliotecas nativas mediante el servicio PInvoke, pero no así la creación y utilización de clases nativas, lo cual es un requisito pues RT-DESK necesita crear clases que hereden de RTDESK_CEntity y sobrescriban el método ReceiveMessage, el cual es llamado directamente desde el código nativo. La solución a este problema pasó por usar el lenguaje C++/CLI para crear una clase puente. Así, se crearon diferentes clases C++/CLI, una por cada una de las clases de RT-DESK que se necesitaban utilizar (RTDeskEngine, RTDeskEntity y RTDESK_CMsg).

Adaptación de la clase Game

RT-DESK sustituye al núcleo de un videojuego, el bucle principal. El segundo problema que surgió fue que el bucle que provee la clase Game de XNA, oculto al diseñador, y que además de encargarse de actualizar y dibujar los elementos del juego también se encarga de otras como la gestión de la ventana de juego o la comunicación con los dispositivos gráficos de salida. De esta forma, para permitir que el control de bucle principal pasase a RT-DESK se creó una nueva clase Game que se encargase de esas tareas que hasta ahora realizaba XNA en su bucle principal:
Se creó un proveedor de servicios, implementando la interfaz IServiceProviery, y el servicio GraphicsDeviceService, implementando IGraphicsDeviceService, que proporciona el dispositivo gráfico.
Se gestionó la creación de la ventana de juego, mediante la clase form de System.Windows.Forms.

[20] https://en.wikipedia.org/wiki/Common_Intermediate_Language
[21] https://msdn.microsoft.com/es-es/library/8bs2ecf4%28v=vs.110%29.aspx

Aparte de los mencionados anteriormente, la adaptación de RT-DESK a XNA ha presentado otros problemas. El origen de estos es el mismo, no usar el bucle principal que ofrece XNA y que se encargaba de algunas tareas esenciales:
Problemas con la entrada, la cual no funciona correctamente debido a que el bucle principal de XNA no está ahí para asegurarse que la entrada se actualiza a la misma velocidad que el juego, que es algo que no se permite realizar desde fuera. Se ha intentado ejecutar el bucle de XNA en paralelo al de RT-DESK para asegurar esa actualización periódica de los periféricos de entrada, tanto usando hilos como ejecutando el bucle de rt-desk desde dentro del bucle de xna pero no se llegó a una solución. Una solución segura pasaría por usar otras librerías ajenas a XNA para gestionar la entrada.
Problemas de audio. En particular, con el tipo "sound" (Ver la sección de audio) que permite reproducir sonidos sin controlarlos. Si no está el bucle de XNA funcionando, se lanza una excepción avisando explícitamente de esto mismo y terminando la ejecución. Una solución pasaría por usar únicamente SoundInstances o, de nuevo, usar una librería ajena a XNA como puede ser FMOD, enfocada a la reproducción de audio.

Pruebas

Entorno de pruebas

Un computador formado por procesador Intel Xeon CPU E5-2620 2GHz, 16 GB RAM, tarjeta gráfica NVIDIA GeForce GTX 760, con Windows 7 Professional 64 bits como sistema operativo, usando service pack 1, Microsoft Visual Studio C# Express 2010 y XNA 4.0.

Diseño de las pruebas

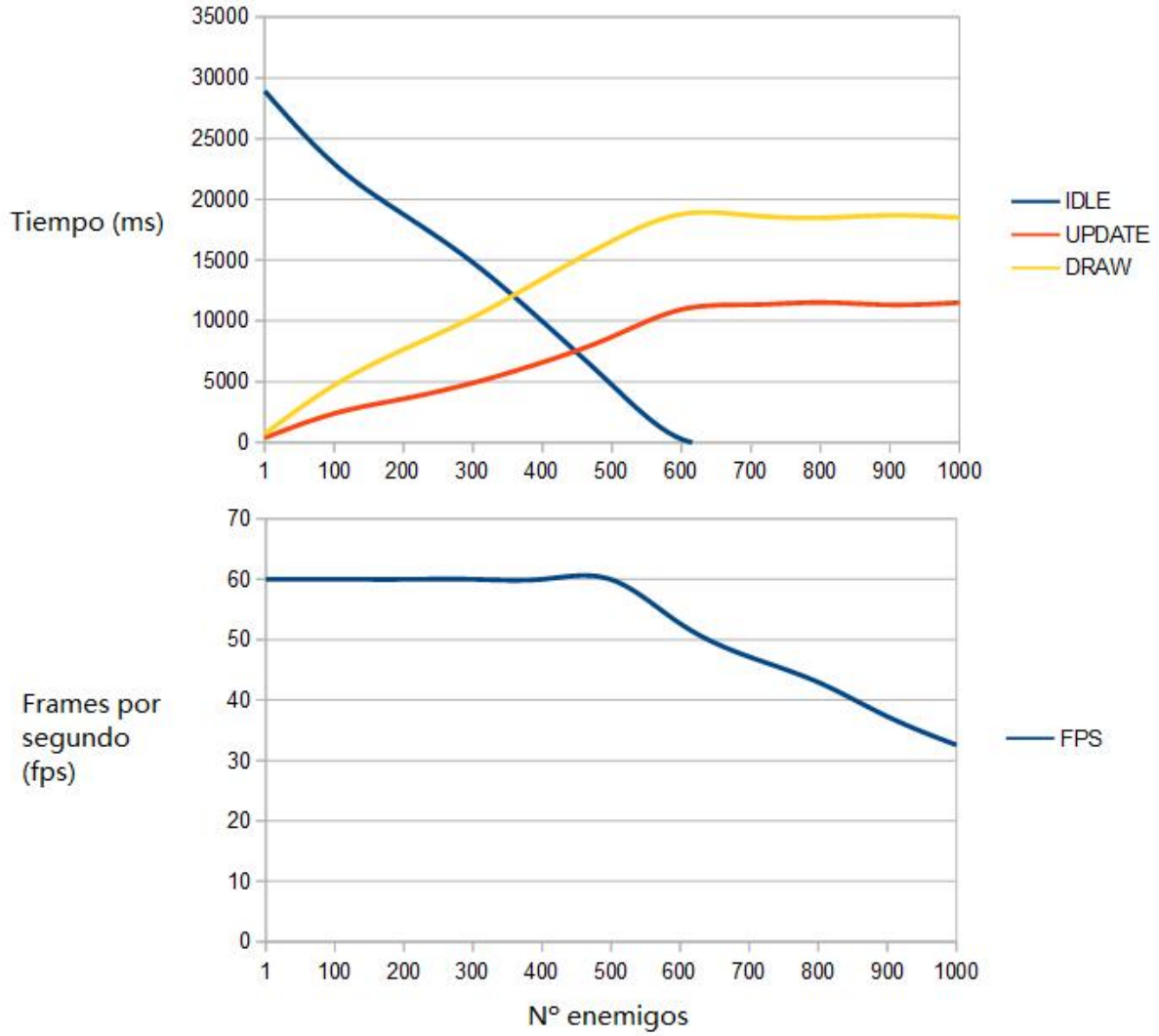
En cada una de las versiones del juego (Continua y discreta, usando RT-DESK), se han realizado dos tests distintos, uno con enemigos "normales" y otra con jefes (Estos disparan balas). En ambos casos estaban activados los fondos por tiles y el personaje principal (sin entrada). La ejecución de todos los elementos es determinista (No se han usado elementos procedurales, ni aleatorios ni entrada del jugador).
En cada test se varia la cantidad de enemigos (De 1000 en 1000 en el caso de la prueba con enemigos normales y de 100 en 100 en el de los jefes), y por cada cantidad se han realizado cinco pruebas distintas para un mayor rigor en la medida y comparación. Cada una de estas pruebas dura 30 segundos y se contabiliza qué fracción se destina a cada una de las siguientes fases:

- Fase de dibujado (Draw), en la que se dibujan los objetos.
- Fase de actualización (Update), en la que se actualizan los objetos.
- Fase de idle, tiempo restante no usado.

Resultados

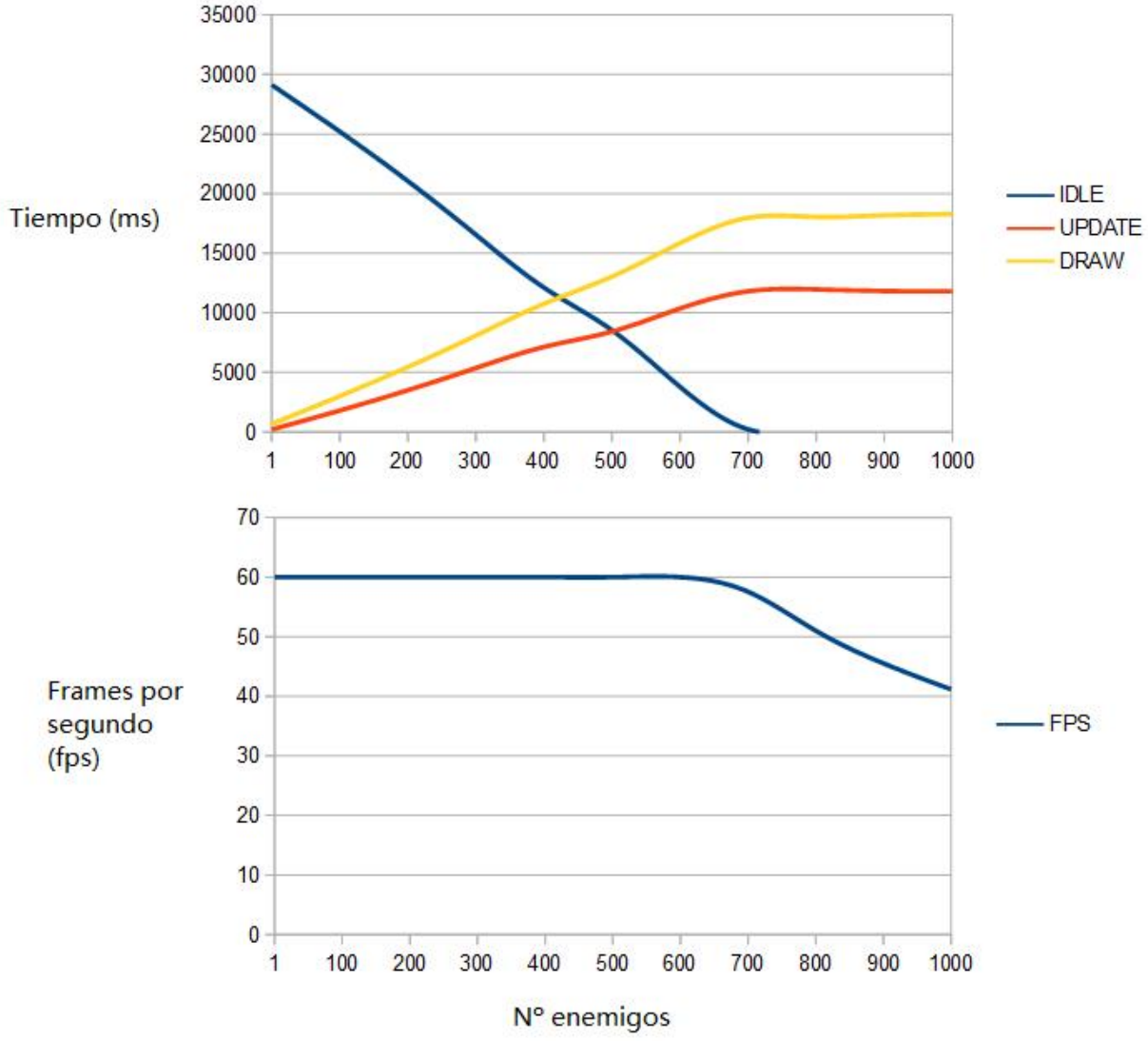
En ambos tests los resultados han sido muy similares, variando únicamente la magnitud (El jefe es más pesado computacionalmente que el enemigo normal, además de disparar, y por tanto hacen falta menos para saturar el sistema). Los siguientes datos pertenecen a la prueba con jefes.

En primer lugar, los resultados de la versión continua:



Se puede observar que el sistema se satura con casi 600 objetos creados, que es cuando la fase de update y de draw suman un tiempo total superior al tiempo de iteración, y por tanto el tiempo de idle es 0. Al saturarse obviamente el tiempo de update y de draw deja de crecer, aunque en cada iteración si es mayor, propiciando una caída de frames (ralentización de la ejecución).

En la versión con RT-DESK se puede observar en la gráfica superior como el tiempo de actualización y dibujado es menor que en la versión continua, resultando en un mayor tiempo de idle y por lo tanto requiere de mayor carga para saturarse (Unos 700 elementos).
En la inferior se muestran los frames por segundo, los cuales empiezan a caer al alcanzar el estado de saturación, de nuevo necesitando para ello más carga que en la versión continua.



La diferencia entre una versión y otra sería la posibilidad de contar con unos 100 jefes extra en la versión de RT-DESK (o unos 2000~ enemigos normales, según la prueba con enemigos), aunque dicho tiempo extra podría ser usado para cualquier tarea.

Conclusiones

El uso de RT-DESK ha sido más sencillo y rápido de lo esperado, además de haber quedado patente sus ventajas frente al bucle principal tradicional.

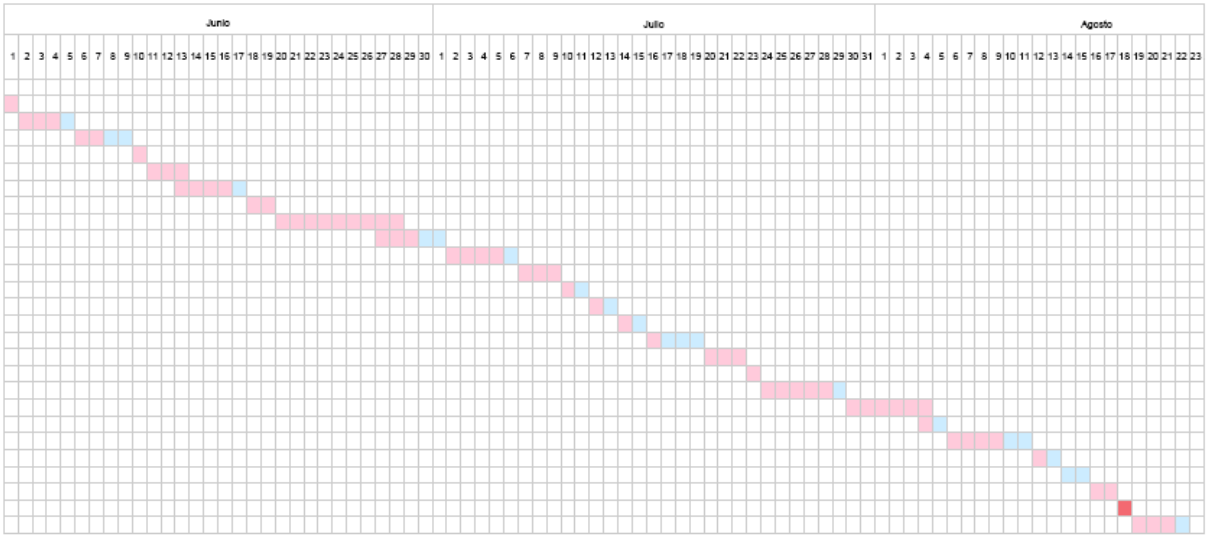
Por otra parte el desarrollo del videojuego, partiendo de cero y realizando todo el diseño, la planificación e implementación ha sido de gran ayuda para entender y permitirme profundizar en una de las aplicaciones de la informática que más interés me despiertan y que apenas se ha tocado en la carrera, la creación de videojuegos.
La lección más valiosa que he sacado en esta parte es sobre la importancia de la planificación. Como medio de expresión, es difícil saber desde el principio cómo va a ser el videojuego que se va a hacer porque siempre hay cosas que depurar o cambiar, cosas que sobre la marcha se descubren, o algunas que simplemente se van a quitar. Sin embargo, como proyecto y como programa que es, gran parte del mismo sí se puede (Y debería) de planificar.
La prueba de esto es que por muchas semanas que dediqué a hacer pruebas pensando acerca del diseño del juego, buscando algo que fuese una base sólida, a la hora de implementarlo cambié cosas. Y además, en el momento en que empecé a planificarlo, a marcarme metas objetivas a distintos plazos, desarrollarlo se hizo más sencillo, y teniendo control de esos aspectos que si se pueden planificar puede uno fácilmente realizar cambios, si fuesen necesarios, en esos otros aspectos más “artísticos”.

En cuanto a la planificación temporal y del esfuerzo que asumí me llevaría, en general he tardado lo que asumí o un poco menos, en parte porque varias de la tareas se simplificaban al haber realizado algunas previas (Por ejemplo, el primer Boss costó más que los siguientes, que se beneficiaron de su código). Una de mis preocupaciones al empezar el proyecto fue el coste de hacer la adaptación a RT-DESK. Sin embargo, y como he comentado antes, resulto sencillo y directo. La tabla y el diagrama de Gantt actualizados se encuentran en la siguiente página.

Relación del trabajo con estudios cursados

La realización del proyecto ha resultado beneficiosa de cara a poner a prueba lo aprendido en la carrera, y especialmente en mi especialidad de ingeniería del software. Se ha aplicado muy especialmente cuestiones referentes a la programación, gráficos por computador y técnicas de diseño y gestión de proyectos, estas últimas se han revelado muy útiles como se ha comentado en el proyecto anterior.
La ingeniería informática es muy amplia y en la carrera se han tocado muchas ramas. No las pondré en prácticas todas, y no todas me gustan, pero entré en la universidad buscando poder hacer juegos, y he aprendido lo suficiente como para poder hacerlo.
Y destacar especialmente la destreza general como ingeniero obtenida a lo largo de la carrera, que me han permitido coger una tecnología completamente desconocida para mi como es RT-DESK y comprenderla y aplicarla de forma sencilla, así como aplicar lo estudiado a algo que no se ha estudiado explícitamente, como son los videojuegos. Un compañero suele decir que en la universidad lo más importante que se aprende es a aprender, una base sobre la que construir el resto de conocimientos, con lo que estoy muy de acuerdo.

Fase	Tarea	tiempo estimado	tiempo real
Requisitos	Brainstorming	1	1
Diseño	Creación GDD	4	3
	Diseño UML	4	2
Implementació	Herramientas	1	1
	Programa base	2	3
	Esqueleto de clases	5	4
	Input Manager	2	2
	Player + playerBulletManager	7	9
	EnemyBulletManager + Enemy bullet	5	3
	EnemyManager + regular enemies	5	4
	Boss 1	3	3
	Boss 2	2	1
	Boss 3	2	1
	Boss 4	2	1
	StageManager	4	1
	Tile map background	3	3
	procedural background	1	1
	Menu	6	5
	AudioManager	5	6
	Añadir sonidos	2	1
	Creación de mapas de prueba	6	4
	OpeningScreenManager	2	1
	Tutorial + Creditos	2	
Verificación	Testeo	2	1
Port		?	10
Pruebas		4	2



Se ha pintado en rosa el tiempo real sobre el tiempo estimado, pintado en azul.

Trabajos futuros

La tecnología RT-DESK está validada por diferentes proyectos, el siguiente paso debería de ser pulirla y facilitar su uso a un mayor número de desarrolladores. Sin embargo, los problemas anteriormente descritos (adaptar el bucle principal, manejar la ventana, problemas de input, limitaciones en el audio) que hacen que para usar RT-DESK junto con XNA haya que renunciar a muchas de las posibilidades de esta (En parte por funcionar como una caja negra) y sustituirlas por otras opciones me hacen pensar en si no sería más adecuado adaptar RT-DESK a otra tecnología que sí permita una acople perfecto entre ellos. De querer continuar por esta linea, sería recomendable centrarse en MonoGame una vez esté completa, la adaptación libre de XNA, y ver si se tiene acceso al código del bucle principal y se puede realizar esa mejor integración.
Aparte de las versiones ya existentes que hacen uso de OpenGL, otro buen candidato y con las ventajas de XNA sería SFML, al usar C++ y no proveer de un bucle principal cerrado la adaptación sería trivial, pero si el proyecto de adaptar RT-DESK al motor Unity saliera adelante esta sería la opción ideal, tanto por ser uno de los más usados como por contar con una tienda desde la cual se podría distribuir (incluso vender) RT-DESK a una gran cantidad de usuarios, pertenecientes además de una comunidad conocida por su gran cantidad de documentación y recursos, lo cual sin duda sería de ayuda para retroalimentar la difusión y comprensión de RT-DESK.

Agradecimientos

Me gustaría agradecer la posibilidad que me ha dado Ramón Mollá, director del proyecto, de hacer como proyecto final de carrera lo que he querido hacer desde que empecé esta, un videojuego, así como por toda la ayuda y consejos durante el desarrollo del mismo.
También, además, a todos los compañeros que han trabajado en los diferentes proyectos de RT-DESK, especialmente a David Pérez, cuya adaptación de XNA a RT-DESK y documentación de la misma han sido imprescindibles para este proyecto.
Y para acabar a mis padres, por “aguantarme en casa”, poniéndome las cosas, otra vez, bastante más fáciles.

Glosario

Bucle principal:
Núcleo del juego encargado de gestionar la entrada y actualizar y dibujar los elementos que lo forman.

GDD:
Game Design Document (Documento de diseño del videojuego). Se trata de un documento que describe los diferentes aspectos del videojuego y que sirve como guía de referencia para todo el equipo.

HUD:
Heads up display, la interfaz gráfica que se muestra durante el juego para mostrar el valor de diferentes atributos de la partida como energía del personaje, vidas, tiempo, etc

RT-DESK:
Real Time Discrete Simulation Kernel, es una librería diseñada para la gestión temporal de eventos discretos en tiempo real.

Shoot ‘em up:
Género de videojuegos donde prima la acción, habilidad con el mando y los reflejos.

Sprite:
Asset gráfico 2d usado para representar visualmente los elementos del juego.

Sprite sheet / Sprite set:
Imagen donde se guardan los diferentes sprites que forman las animaciones de los elementos de un juego.

XNA:
Conjunto de herramientas creadas por Microsoft para facilitar y apoyar la creación de videojuegos.

Fuentes

https://en.wikipedia.org/wiki/Video_game
http://rtdesk.blogs.upv.es
http://www.statista.com/topics/868/video-games
http://entropyinteractive.com/2011/02/game-engine-design-the-game-loop/

Bibliografía:

Jesse Schell, *Art of game design: a book of lenses.*
Aaron Reed, *Learning XNA 4.0.*
David Pérez Climent, *Creación de un videojuego en XNA usando RT-DESK.*
Pablo de la Ossa Pérez, *Zentract: Engine con simulación discreta para videojuegos 2D.*

Anexo A: GDD

Índice

Introducción

“Shoot ‘em up” se trata de un videojuego 2D de corte clásico que, como su nombre (provisional) indica, homenajea al género shoot ‘em up / de naves.

En él el jugador controla una nave espacial que se enfrenta a hordas de enemigos, destruyéndolos mientras esquiva sus disparos. Se trata de un juego en el que se ponen a prueba los reflejos y la destreza con el mando del jugador.

El jugador cuenta dos objetivos, puede jugar intentando sumar el máximo número de puntos posible para alcanzar los primeros puestos de la clasificación o intentar sobrevivir tanto como pueda para avanzar en la historia del juego y descubrir nuevos retos a los que enfrentarse.

Es un juego creado tanto por las ganas de aprender como por mi cariño a este tipo de juegos, y está dirigido a cualquier que comparta la afición por ellos o a los que disfrutan con los retos de habilidad con el mando.

Modos de juego

El juego cuenta con dos modos de juegos principales, el modo historia y el modo libre.

En el modo historia las tablas de clasificación están activadas, y el jugador se enfrenta a una serie de retos predefinidos.

En el modo libre, sin embargo, no se contabilizan la puntuación. En él, desde un menú el jugador puede seleccionar cualquiera de las fases del juego y algunas extra, así como seleccionar en qué escenario quiere jugar. Este modo existe para poder acceder rápidamente a los contenidos del juego, ya sea para su evaluación y muestra o para que cualquiera pueda rejugar sus fases preferidas.

Menús

El jugador puede moverse entre las opciones con las flechas de dirección, seleccionarlas o volver a la pantalla del menú anterior. Hay un tipo de opciones especiales (prueba y fondo, de la pantalla de opciones) que te permiten seleccionar entre diferentes valores con izquierda y derecha (en este caso, la prueba que quieres jugar y con qué fondo). Además, la pantalla de puntuaciones (Scores) permite al jugador meter sus iniciales y su puntuación.

El jugador

El jugador controla una nave espacial.

Sus movimientos son los siguientes:

Desplazamiento

Al ser un juego 2D, el jugador puede desplazarse a lo largo de los ejes X e Y. No existe inercia ni gravedad, tan pronto como el jugador pulsa el stick, la nave se mueve, y frena tan pronto lo suelta.

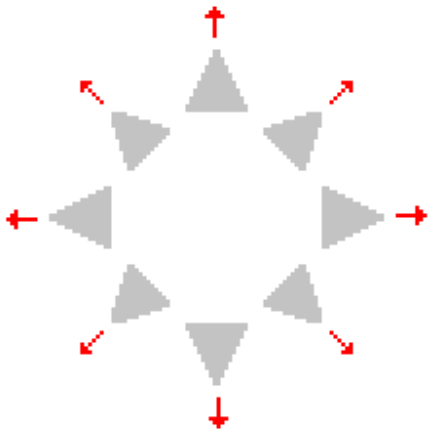
El movimiento se realiza con el stick izquierdo del mando, para permitir al jugador mover la nave en cualquier dirección y cambiar su velocidad (puede desplazarse lenta o rápidamente dependiendo de cuanto desplace el stick). Podría decirse que existe una relación 1:1 entre el stick y el desplazamiento de la nave.

Disparo

Al contrario que en la mayoría de juegos del género, en este juego se permite al jugador escoger en qué dirección va a disparar.

La dirección del movimiento no afecta la dirección de disparo. Se escoge la dirección de disparo y dispara al mismo tiempo usando los 4 botones frontales de la parte derecha del mando (Los clásicos, O, X, cuadrado y triángulo en el mando de playstation, o A, B, X e Y en el de xbox 360). Debido a su disposición (uno arriba, otro abajo, otro a la izquierda y otro a la derecha) son muy cómodos para escoger la dirección del disparo.

Así, por ejemplo, pulsando el botón triángulo del mando de ps3 (el botón de arriba) la nave disparará hacia arriba. Se puede pulsar dos direcciones contiguas, de forma que se dispare en diagonal, haciendo que el jugador pueda disparar en un total de 8 direcciones distintas.



Se ha preferido esta solución frente a permitir disparar en cualquier dirección haciendo uso del stick derecho en parte para que el jugador no tenga la necesidad ni posibilidad de estar pendiente de la dirección del disparo en detrimento de centrarse en el movimiento de la propia nave y el esquivar los disparos enemigos.

Al disparar, dos balas independientes salen en paralelo en la dirección escogida. Estas, al contactar con un enemigo, le restan puntos de energía, o desaparecen al salir de los bordes de la pantalla.

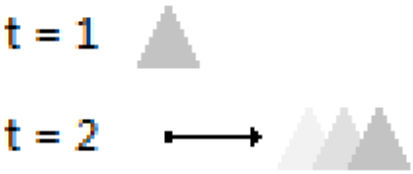
Escudo

La nave cuenta con dos escudo capaces de absorber las balas enemigas azules o verdes (ver tipos de balas más adelante). Este se activan con los botones L1 y R1 (o Lb y Rb en xbox).
Una vez activado, el jugador puede tocar una bala del color del escudo sin sufrir daños. Sin embargo, si activa los escudos y no absorbe ninguna bala de dicho color en los 2 siguientes segundos, el escudo se vuelve rojo para avisar al jugador de que, si no absorbe una bala o apaga los escudos, se va a sobrecalentar.

En caso de sobrecalentamiento la nave pasa a estar de color rojo hasta que se pasen los efectos (4 segundos), además de no poder volver a activar los escudos en ese intervalo.

Teletransporte

La nave cuenta con un movimiento de teletransportación que puede activar tantas veces como quiera.



Al teletransportarse puede esquivar todo tipo de balas y enemigos, pero se debe tener cuidado de no teletransportarse a una posición donde se encuentra uno de ellos.

La dirección del teletransporte es la dirección del movimiento (stick izquierdo). Además, la distancia de teletransporte es fija y no depende de cuanto se incline el stick.

Power up

En el juego existe un power up que, al ser tocado por la nave protagonista, hace que esta cuente durante un tiempo limitado con balas más fuertes. Mientras dura el efecto, la nave y las balas cambian a un color amarillo.

El power up aparece por la parte superior de la pantalla y avanza hasta que sale por la parte inferior de esta o lo coge el jugador.

Condición de derrota

El jugador cuenta únicamente con una “vida”. La nave protagonista se destruye al tocar cualquier enemigo o bala de estos, causando una explosión y dando paso a la pantalla de puntuaciones donde, en caso de entrar en el ranking, el jugador puede introducir sus iniciales. En cualquier caso, se vuelve al menú principal.

Balas enemigas

Las balas enemigas pueden ser de tres colores distintos y de diferentes tipos.

Colores

Las balas pueden ser de color azul, verde, rojo o gris.

Las balas azules pueden ser absorbidas por el escudo de la nave, las grises pueden ser destruidas por los disparos de la protagonista y las rojas únicamente pueden ser esquivadas.
Además, las balas pueden cambiar de color siguiendo diferentes patrones.

Tipos

Bala normal

Su trayectoria es lineal y de velocidad constante (aunque esta puede variar entre balas). Puede ser de cualquiera de los tres colores.

Bala buscadora

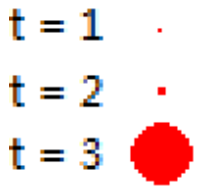
Al ser disparada calcula qué eje (El horizontal o el vertical, teniendo como origen la posición de la nave del jugador) está más alejado y persigue al jugador hasta alcanzar ese eje, momento en el que su trayectoria pasa a ser lineal (manteniendo la última dirección que seguía) hasta que sale de la pantalla. Puede ser de cualquiera de los cuatro colores.

Bala rebotadora

En vez de salir de los límites de la pantalla, al tocarlos rebota. Hay de dos tipos: Las que no desaparecen hasta que no derrotas al que las ha lanzado o las que desaparecen por sí solas tras un rato. Además puede escoger en que paredes rebota y en cuales no, de forma que si sale la bala desaparece.
Puede ser de cualquiera de los cuatro colores.

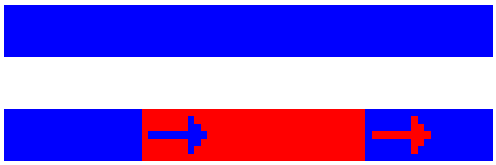
Bala explosiva

Se muestra como un pequeño punto fijo que sirve de aviso, ya que momentos después le sigue una explosión en esa misma posición. Únicamente es de color rojo.



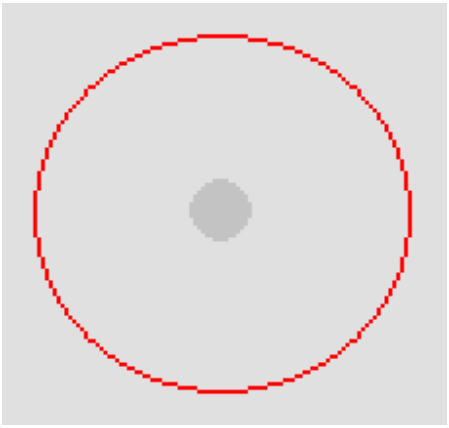
Láser

El láser parte de la nave enemiga hasta el borde de la pantalla. Puede ser de color azul, verde o rojo, además de poder alternar entre ambos siguiendo diferentes patrones.



Onda

Similar al láser pero en vez de ser lineal tiene forma circular y se expande desde su origen (el enemigo que la dispara). Puede ser de color azul, verde o rojo, además de poder alternar entre ambos siguiendo diferentes patrones.



Tipos de enemigos

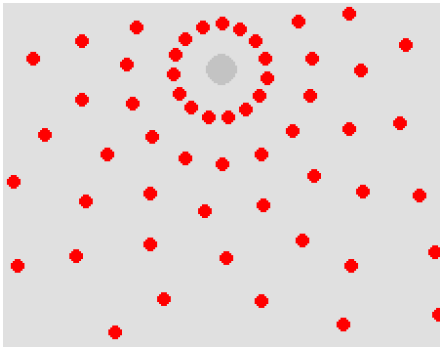
Enemigos

En el diseño de los enemigos normales (no con los jefes) no sigue un patrón concreto sino que se van a crear enemigos básicos cuyo comportamiento se pueda modificar fácilmente para crear un gran número de enemigos diferentes, asignando diferentes patrones de movimientos y de disparos, así como atributos (como energía, nº de puntos).

Por ejemplo, un enemigo que aparezca por la un lado de la pantalla y avance disparando hacia el otro. Un enemigo que aparezca por un lado de la pantalla, dispare y vuelva por donde ha venido. Enemigos que puedan aparecer espontáneamente en el interior de la pantalla, o de forma a la posición del protagonista. También enemigos que dispares siguiendo diferentes patrones de tipo, color, dirección y demás atributos.

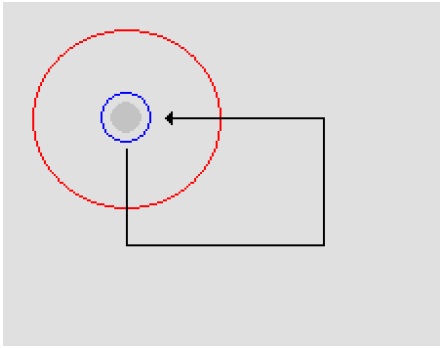
Jefe 1

Este jefe permanece en la parte central superior de la pantalla sin moverse mientras dispara rachas de balas rojas en todas direcciones. La velocidad de las balas oscila entre un mínimo y una máximo. Tras acabar con ⅔ de su vida ralentiza las balas por debajo del mínimo y empieza a añadir más balas, y cuando está a punto de morir a aumentar su velocidad y añadir también balas grises.



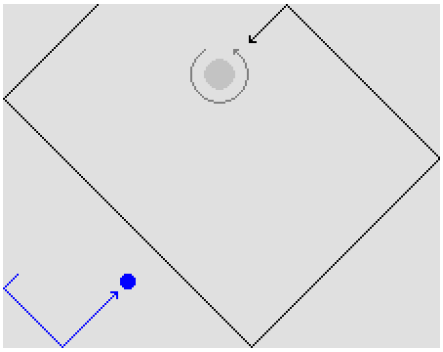
Jefe 2

Este jefe se desplaza por la pantalla siguiendo un recorrido fijo y soltando ondas de cualquiera de los tres colores de forma aleatoria. Poco a poco el intervalo entre ondas se va estrechando y las ondas son más rápidas. Llegado cierto punto, las ondas se limitan a los colores azul y rojo.
Además, una onda puede ser de diferentes colores (porque está dividida a trozos o porque se va alternando).



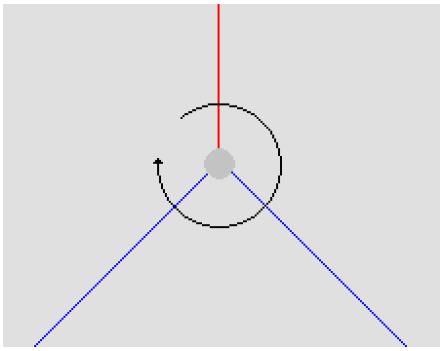
Jefe 3

Este jefe se mueve por la pantalla rebotando en sus esquinas. Cuenta con un escudo que lo envuelve protegiéndolo de disparos y que no puede ser destruido, pero que tiene una abertura por la que poder dispararle. El escudo gira, la dirección del giro cambia de tanto en tanto.
Desde el escudo salen disparadas bolas rebotantes, y cuanto más cerca se está de vencer al jefe, más balas hay en pantalla.
Además, de vez en cuando se lanzan balas destructibles en todas direcciones.



Jefe 4

Este jefe se encuentra en el centro de la pantalla, no se mueve por ella pero rota constantemente. Dispara lasers continuos que rotan junto a él. Conforme se le va quitando vida, la velocidad de gira aumenta, así como el número de lasers y sus patrones, además de añadir algunos disparos extra.



Escenarios

A parte de un simple color de fondo, muy práctico para evitar distracciones visuales, el juego cuenta con otros dos tipos de escenarios, los basados en mapas de tiles y los generados proceduralmente.

Mapa de tiles

Son escenarios creados a mano a partir de un conjunto de tiles. Se aplica un scroll para que de sensación de avance y se crea un loop para que resulte infinito mientras dure la fase.
Usando esta técnica se creará el escenario del bosque.

Mapa procedural

Aunque un mapa procedural puede ser un mapa de tiles, en este caso la técnica se usará para crear un escenario basado en el cielo donde se creen sobre la marcha diferentes elementos de diferentes tipos (por ejemplo, nubes) y velocidades (para simular profundidad) siguiendo unas reglas y restricciones.

En el caso de las nubes además se aplicará un shader o alguna técnica de forma que se impriman de forma semitransparente sobre los otros elementos (protagonista, enemigos, balas, etc.) pero no sobre el propio fondo (de forma que se mantengan blancas).