UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escuela Técnica Superior de Ingeniería Informática
Universidad Politéctica de Valencia

Creación de un videojuego en XNA utilizando RT-DESK

Proyecto Final de Carrera
Ingeniería Informática
Julio 2013

David Pérez Climent
Director: Ramón Mollá Vayá

A Paloma Latorre,

por darme el impulso que necesitaba

para hacer lo que quería hacer

y por hacer Olympic Punch posible

Indice

Introducción

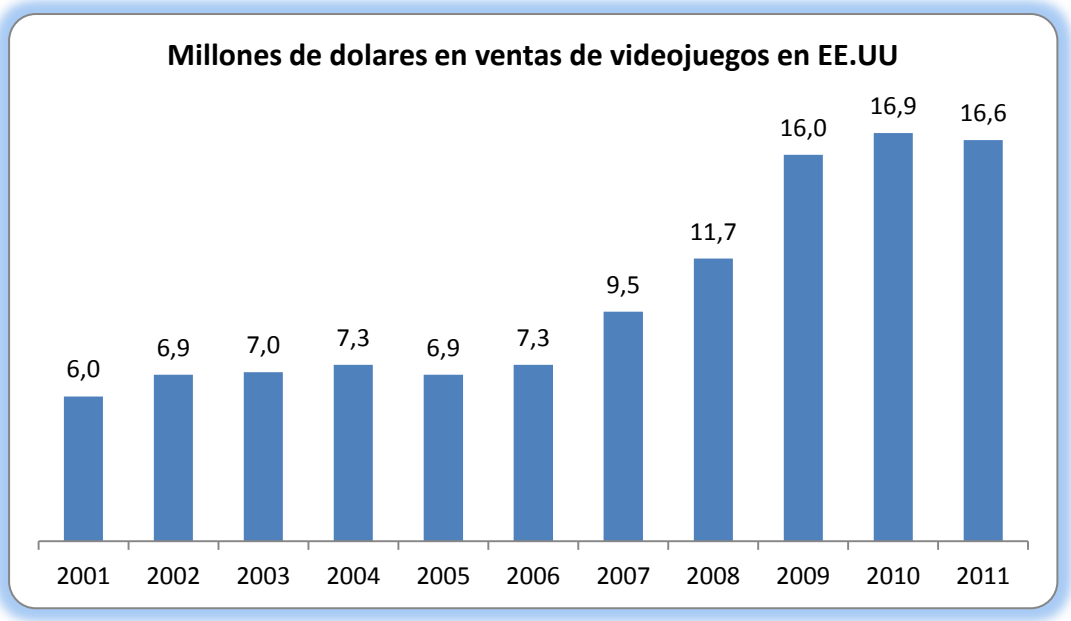
"Jugar es una actitud u ocupación voluntaria ejecutada dentro de unos límites establecidos de espacio y tiempo, conforme a unas reglas aceptadas libremente y absolutamente vinculantes, cuyo objetivo es la misma acción de jugar y a la que acompaña al sentimiento de tensión, de alegría y la conciencia de que es diferente de la vida normal"

Johan Huizinga- Homo Ludens (1938)

Motivación

La industria del ocio digital

Solo en Estados Unidos, en 2011 los consumidores invirtieron un total de 24,75 billones de dólares en la industria de los videojuegos [ESA]. Si bien es un pequeño decremento con respecto al ejercicio anterior derivado del contexto económico actual, el número de jugadores sigue creciendo, lo que hace prever que el crecimiento continuará en el futuro. En los últimos años, la industria ha sufrido un crecimiento vertiginoso, alcanzado en los años 2005-2009 un crecimiento anual del 10.6%.

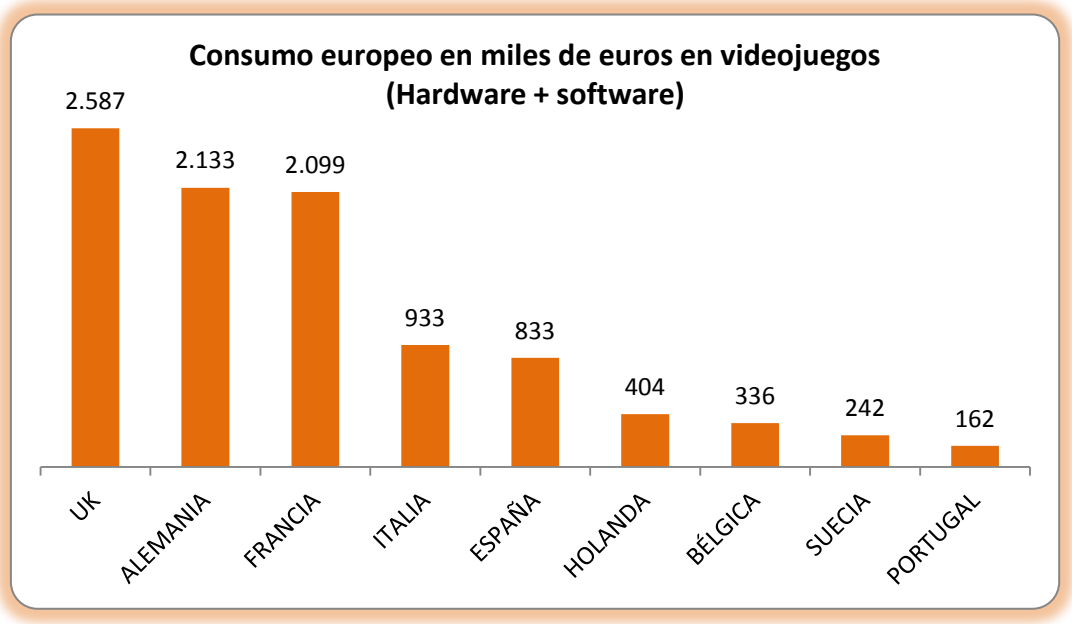


De forma paralela a este crecimiento, el proceso de desarrollo de un videojuego ha sufrido grandes cambios. En 1985, el desarrollo de un videojuego típico era llevado a cabo por un equipo de 2 a 6 personas durante de 4 a 12 meses de desarrollo con un presupuesto de 20-80 miles de euros. Actualmente un videojuego medio se realiza por unas 100 personas durante 2 o 3 años con un presupuesto de 20-30 millones de dólares: En los videojuegos de más potencia (los llamados AAA), estas cifras pueden incrementarse considerablemente. Este es el caso del videojuego Star Wars: The Old Republic que tuvo un presupuesto de 200 millones de dólares y empleo a más de 650 personas.

Con estas cifras no es de extrañar que en los años 2005 a 2009 creciera un 8.65% anual el empleo directo en la industria del videojuego y que actualmente empleé a más de 32.000 trabajadores directos en EE.UU, con un sueldo medio de $89.781 [ESA].

En España, en 2011 se alcanzó un consumo de 980 millones de euros en el sector que si bien representa un decremento del 15% con respecto al año anterior, el número de jugadores sigue creciendo, alcanzándose un 24% de jugadores entre la población adulta [ADESE].

Tradicionalmente, España viene siendo el cuarto o quinto mercado europeo en ventas de videojuegos, sin embargo, en cuanto a producción del ocio digital ocupa el décimo puesto, pues apenas el 1% de las ventas son productos nacionales.



En cuanto a las plataformas hay una clara supremacía de las videoconsolas frente al PC. En España, las ventas de juegos para consola alcanzaron 468 millones de euros frente a los 29 millones de euros de PC [ADESE]. El hecho de que tradicionalmente las videoconsolas sean más cómodas para jugar, junto con el hecho de que son sistemas propietarios que generan grandes beneficios ha propiciado que las multinacionales se hayan esforzado por fortalecer a las consolas por encima del PC. Un claro ejemplo de esto podemos verlo en la diferencia de precio entre los juegos de PC y consola

En los últimos años se han extendido nuevas plataformas para jugar, como los smartphones o los juegos de las redes sociales. La International Data Corporation (IDC) cifra en 491 millones el número de smartphones vendidos en 2011 en todo el mundo con un crecimiento anual del 58%. Con semejante número de potenciales usuarios, y su relativamente nueva aparición, los smartphones se han convertido en una de las plataformas favorita de muchos desarrolladores.

Tarea multidisciplinar

Por un lado, una de las grandes ventajas que presenta la realización de este este proyecto frente a otros posibles proyectos es la colaboración con una persona externa al mundo de la informática. En una gran parte de los proyectos de creación de software se necesita la colaboración estrecha de miembros de diferentes disciplinas, por lo que saber coordinarse con el grupo y comunicar conceptos propios del gremio de forma que todo el equipo pueda entenderlos es de vital importancia. En este aspecto, el desarrollo de un videojuego es el máximo exponente. Es necesaria la colaboración de grafistas, modeladores, programadores, músicos, diseñadores, guionistas, etc. por lo que la creación de un videojuego es una buena forma de practicar todas estas habilidades.

Por otro lado, dentro de la parte técnica de un videojuego, propia del trabajo de informáticos, un videojuego utiliza técnicas de muy diversas áreas de la informática por lo que la implementación de un videojuego es una buena forma de poner en práctica muchos de los conocimientos adquiridos durante la carrera. Algunas de estas áreas son:

- **Gráficos por computador:** Un videojuego requiere gráficos para que el jugador pueda percibir el mundo que se ha creado para él. Uno de los requisitos más importantes de estos gráficos es que sean en tiempo real.
- **Inteligencia Artificial:** Define el comportamiento de todos los personajes que no son controlados por el jugador. La IA de un videojuego debe de ser lo bastante "inteligente" como para presentar un reto al jugador pero no tanto como para que el jugador se vea superado, puesto que no se divertirá.
- **Ingeniería del software:** Los videojuegos son obras de ingeniería muy complejas que pueden contar fácilmente con más de 1 millón de líneas de código y resulta imprescindible aplicar las técnicas y herramientas que nos proporciona la ingeniería del software.
- **Ingeniería del hardware:** Los videojuegos modernos requieren de enormes potencias de cálculo para funcionar y es necesario conocer a fondo en funcionamiento del hardware en el que se van a ejecutar para poder exprimirlo al máximo.
- **Redes:** Una gran cantidad de juegos multijugador nos permiten jugar simultáneamente con jugadores repartidos por todo el globo.
- **Teoría de lenguajes y compiladores**: Los videojuegos manejan una cantidad enorme de datos. Para una mayor productividad es importante tener separados los datos de la lógica que los maneja por lo que es común almacenarlos en ficheros externos con gramáticas diseñadas para tal fin, por ejemplo en XML.

Los videojuegos independientes

Un videojuego AAA con un presupuesto de 100 millones de dólares necesita vender entre 2 y 3 millones de copias solo para recuperar la inversión. De ahí que las grandes compañías puedan correr pocos riesgos y apuesten sobre seguro con licencias de series de éxito y personajes consolidados. Aún en los juegos "nuevos" los diseñadores se ven obligados a repetir los mismos patrones una y otra vez, creando verdaderos clones de los juegos de éxito y en el mejor de los casos la innovación se ve reducida a "es una mezcla entre Gears of War y Mass Effect" [DDV].

Frente a este fenómeno están los videojuegos independientes, videojuegos producidos por pequeños estudios o programadores amateur que desarrollan sin el apoyo económico de los grandes distribuidores. Cuentan con pequeños presupuestos, por lo que no tienen que vender enormes cantidades de copias para no entrar en perdidas y pueden correr más riesgos, teniendo total libertad creativa. Por supuesto, deben prescindir de gráficos impresionantes, de grandes argumentos y doblajes de actores profesionales. Esto les permite centrase en lo que hace grande a un videojuego, que sea divertido y enganche sin estar tiranizados por la tecnología.

Imagen 1: Limbo, un videojuego independiente ampliamente galardonado

Si bien la aparición de los juegos independientes data de los años 70, cuando la industria de los videojuegos aún no se había establecido como tal, es en la última década cuando han experimentado un gran crecimiento debido, sobre todo, al auge de los teléfonos inteligentes y de las descargas digitales. Los teléfonos inteligentes son dispositivos mucho menos potentes que un PC lo que conlleva juegos con una potencia muy limitada. Esté hecho, junto con la naturaleza casual, esporádica y breve del tiempo de juego en los dispositivos móviles se traduce en juegos sencillos y de ritmo pausado que pueden ser desarrollados por pequeños estudios con calidad completamente profesional. Por otra parte, la distribución digital abre las

puertas a los pequeños desarrolladores para difundir sus juegos por sí mismos, al margen de las grandes distribuidoras, con un alcance mundial a millones de usuarios y con una nula o mínima inversión. Ejemplos de plataformas de distribución digital son el Play Store de Android y el App Store de Apple, XBLIG y XBLA de Xbox 360, WiiWare de Wii y Steam y Desura para juegos de PC, Mac y Linux.

Hay varios festivales de videojuegos independientes, como el Independent Game Festival, IndiCade y iFest. Estos festivales ofrecen a los desarrolladores independientes un escaparate donde presentar sus juegos y ofrecen importantes premios a los juegos más innovadores.

Lo mejor de todo es que los consumidores no tenemos que elegir. Podemos jugar a un juego indie nuevo y refrescante por que no se parece en nada al resto de juegos y luego irnos a jugar al nuevo Gears of War con una capacidad técnica que nos deja impresionados.

Simulación discreta desacoplada

Tradicionalmente, el núcleo las aplicaciones gráficas en tiempo real, en general, y de los videojuegos en particular, es un bucle en la que de forma acoplada se alternan fases de simulación con fases de dibujado lo que conlleva que la frecuencia a la que deben de realizarse los cálculos afecte a la frecuencia de dibujado. Para asegurar que todos los objetos del mundo virtual realizan sus cálculos adecuadamente y son muestreados a la velocidad necesaria, la aplicación típicamente ejecuta el bucle de forma cíclica el mayor número de veces posible. Es decir, se sigue un paradigma continuo acoplado que presenta el principal problema es la mala gestión de la potencia de cálculo. Como todos los objetos se muestrean a la misma frecuencia, objetos que necesiten una simulación más rápida podrían submuestrearse, mientras que objetos con menos requisitos se sobremuestrarian, con el consecuente desperdicio de potencia de cálculo.

Un cambio de paradigma a discreto desacoplado consiste en que cada entidad de la simulación se ejecute de forma independiente, con una frecuencia de muestreo adecuada a su propia frecuencia interna, incluyendo la visualización y otros aspectos. De esta forma solo se realiza uso de la CPU cuando es necesario, liberando potencia de cálculo que podría o bien emplearse en aumentar la calidad de la aplicación, por ejemplo añadiendo más objetos o realizando procesos más complejos o bien simplemente liberar la CPU/GPU para reducir el consumo.

Objetivos

El objetivo del presente proyecto es doble. En primer lugar, se pretende desarrollar desde cero la “demo” de un videojuego en 2D para PC y Xbox 360, abarcando la totalidad de las tareas que el desarrollo de un videojuego conlleva. Estas tareas abarcan temas completamente dispares:

- Diseño del videojuego.
- Planificación y gestión del desarrollo.
- Creación del contenido visual
- Creación o elección de la música y los efectos de audio.
- Diseño del software.
- Programación del videojuego.
- Testeo y depuración

El proyecto surge fruto de la colaboración con la diseñadora gráfica e ilustradora Paloma Latorre Ortiz, alumna de la Escuela de arte y superior de diseño de Valencia que se encargará del diseño del aspecto del juego y la creación del contenido visual.

Para esta titánica tarea se ha elegido utilizar XNA, un framework de desarrollo de videojuegos para PC, Xbox 360 y Windows Mobile proporcionado por Microsoft que facilita enormemente el desarrollo de videojuegos y que además permite distribuirlos de forma muy sencilla.

En segundo lugar se pretende realizar una portabilidad del juego desarrollado al paradigma discreto desacoplado mediante RT-DESK, un núcleo de simulación de aplicaciones gráficas en tiempo real que sigue en paradigma discreto desacoplado mediante la gestión de eventos ordenados temporalmente y que ha sido desarrollado en la UPV. Se tomaran las mediciones necesarias y se realizara una comparativa del comportamiento del sistema siguiendo ambos paradigmas.

Estructura de la obra

En primer lugar, en la sección de **Introducción** que nos ocupa se expone todo aquello que justifica el presente proyecto, se introduce su estructura y se presenta el trabajo a realizar.

La creación de un videojuego es un proceso realmente complejo, y si bien tiene muchas similitudes con el proceso de creación de cualquier software, también tiene muchas características propias. Debido a la falta de formación específica previa el primer paso para poder realizar este proyecto con éxito es realizar un profundo estudio de las metodologías, herramientas y procesos propios del desarrollo de un videojuego. En **Estado del arte** hablaré del proceso de creación de un videojuego, sus fases, los diferentes roles implicados, las herramientas más utilizadas y las principales diferencias con el desarrollo de software tradicional.

Para la creación de un videojuego el primer paso es determinar para que plataforma vamos a desarrollar, con que herramientas y que juego queremos realizar. Es muy importante que estas tareas se realicen de forma simultánea, ya que están muy relacionadas. Por una parte, ciertos tipos de juegos son más adecuados para unas plataformas que para otras. Por ejemplo, los juegos de rol con montones de acciones posibles para elegir en cada momento son más adecuados para un PC que cuenta con teclado mientras que juegos de acción trepidante donde se requiere que el jugador reaccione en segundos son más adecuados para una consola con mando. Por otra parte es importante conocer las opciones que nos proporciona una plataforma para poder explotarlas en nuestro juego y los juegos similares que puedan haber, para aprovechar los nichos de mercado. De estos aspectos hablaré en **Análisis del proyecto** y **Diseño del juego.**

Una vez terminado el diseño del juego y su documentación, hay que planificar su desarrollo. En **Planificación** hablaré de la estimación inicial de coste temporal de cada una de las tareas y su distribución a lo largo del tiempo que dura el proyecto. También hablaré de los mecanismos empleados para conseguir una buena coordinación del equipo y mi labor como director.

A continuación, bajo el enfoque de la programación orientada o a objetos realizaré el **Diseño del software,** estableciendo mediante diversos diagramas UML las diferentes unidades lógicas que forman el software de nuestro juego, sus relaciones y su comportamiento.

Con todo preparado, llegamos a la **Implementación** del videojuego. Hablaré de todos los aspectos técnicos relevantes en la codificación del juego y los algoritmos empleados.

Simulación discreta dinámica

Conclusiones.

Consideraciones generales

Sencillez vs eficiencia: Puesto que el juego a producir es un juego en 2D que requiere poca potencia de cálculo, a la hora de su programación he optado por una codificación sencilla, fácil de entender e intuitiva frente a un código optimizado más intrincado.

Desarrollo profesional: El videojuego que se pretende realizar es relativamente sencillo y el equipo de producción está formado únicamente por dos personas. Con estas características algunas de las técnicas empleadas propias del desarrollo de videojuegos como una profunda planificación, un desarrollo iterativo o un diseño exhaustivo no están completamente justificadas por no recompensar todo el esfuerzo que conllevan. Aun así, con ánimo de aprender se ha tomado la decisión de realizar un desarrollo lo más profesional posible, intentando aproximarnos a los procesos de desarrollo que podría seguir cualquier empresa de videojuegos.

Estado del arte

Computer and video games have reached a critical mass. Today, nearly every device with a screen plays games, providing interactive entertainment experiences for a wide and diverse population. The creativity of our developers and publishers produces an ever-expanding variety of games to choose from in different formats and across all platforms."

— Michael D. Gallagher, president and CEO, Entertainment Software Association

El bucle principal

Un videojuego es una aplicación gráfica en tiempo real que debe realizar los cálculos de actualización de todos los objetos del mundo virtual y realizar el proceso de visualización sin perder la sensación de tiempo real para el usuario. Tradicionalmente está tarea se realiza mediante un bucle que de forma sistemática y periódica alterna la fase de simulación con la de visualización, típicamente con la mayor frecuencia posible. Es lo que se conoce como el bucle principal. Esta estructura aparentemente tan sencilla puede en la práctica implementarse de varias maneras. A continuación se citan unas cuantas de ellas según [DEWITT], ilustrándolas mediante un pseudocódigo muy cercano a C++.

Para poder analizar correctamente las diferentes implementaciones hay que distinguir entre velocidad de visualización o FPS, el número de visualizaciones que se realizan por segundo, y la velocidad del juego, el número de actualizaciones del juego por segundo.

Sencillo:

Simplemente se alternan las fases de actualización y visualización.

```
while(true) {
    update_game();
    display_game();
}
```

Esta implementación no realiza ningún control del tiempo, por lo que el juego funcionará a diferentes velocidades según el hardware por lo que la experiencia de juego puede ser completamente diferente en un dispositivo y en otro y además impredecible para el desarrollador.

Velocidad de juego constante

El juego se ejecuta a una velocidad constante. Por ejemplo para una frecuencia de actualización de CICLOS_SEG actualizaciones por segundo:

```
while( true ) {
    update_game();
    display_game();
    next_game_tick  +=  (1000 / CICLOS_SEG)
    sleep_time = next_game_tick - GetTickCount();
    if ( sleep_time >= 0 ) {
       Sleep( sleep_time );
    }
 }
```

El código del juego con esta solución es muy fácil de desarrollar, puesto que el juego se ejecuta siempre a la misma velocidad y los cálculos no dependen del tiempo. Si el hardware es lo bastante potente el juego funcionará sin problemas a la velocidad indicada, pero si el hardware no es lo bastante potente el juego se ralentizará, pudiendo incluso ir en algunos momentos más rápido y en otros más lento dependiendo de la carga de la CPU y pudiendo perderse completamente la experiencia de juego deseada.

En esta solución el desarrollador debe seleccionar la velocidad del juego. Con una velocidad alta los dispositivos de bajas prestaciones pueden ser incapaces de ejecutar el juego, mientras que una velocidad baja conduce a una desperdicio de potencia y calidad visual de los dispositivos más potentes por lo que es una decisión importante.

Velocidad del juego dependiente

El juego funciona a la máxima velocidad posible pero los cálculos se realizan según el tiempo transcurrido entre los dos frames.

```
int prev_frame_tick;
int curr_frame_tick = GetTickCount();

while( true ) {
   prev_frame_tick = curr_frame_tick;
   curr_frame_tick = GetTickCount();

   update_game( curr_frame_tick - prev_frame_tick );
   display_game();
}
```

La codificación del juego resulta más complicada porque los cálculos realizados tienen que ser dependientes del tiempo. El comportamiento del juego es el mismo sea cual sea el dispositivo. A pesar de su aparente idoneidad y de ser una solución muy utilizada, puede causar problemas tanto en dispositivos potentes como poco potentes. Por una parte, en dispositivos poco potentes una frecuencia de actualización baja puede afectar al tiempo de respuesta a la entrada del usuario lo que afecta al tiempo de reacción del usuario y a la experiencia de juego. En dispositivos muy potentes, se produce en primer lugar un uso intensivo de la CPU innecesario que limita el tiempo disponible para otras aplicaciones en funcionamiento en el dispositivo y reduce la duración de la batería en dispositivos móviles. Además, como sabemos, los números en como flotante y sus operaciones tienen un error fruto de la discretización al ser representados en un ordenador. Al aumentar innecesariamente el número de operaciones realizadas para un mismo resultado también aumentamos dicho error lo que se puede traducir en un mal funcionamiento.

Velocidad de juego constante y máximo FPS

En este caso el juego se ejecuta con una velocidad constante definida por el programador en todos los dispositivos mientras que la visualización se realiza a la mayor frecuencia posible dependiendo del dispositivo. Cabe destacar que si la frecuencia de visualización es mayor que la frecuencia de actualización varios fotogramas consecutivos dibujarán exactamente la misma escena. Además se establece un máximo de ciclos consecutivos de actualización sin realizar visualización lo que previene que en dispositivos de muy bajas prestaciones la visualización no llegue a realizarse nunca.

```
Int next_game_tick = GetTickCount();
int loops;

while( true ) {
    loops = 0;
    while( GetTickCount() > next_game_tick && loops < MAX_FRAMESKIP)  {
            update_game();
            next_game_tick  +=  (1000 / CICLOS_SEG);
            loops++;
    }

  display_game();
}
```

Esta implementación es una versión mejorada de la versión con velocidad de juego constante, que permite que en dispositivos menos potentes el juego siga funcionando correctamente a costa de reducir los FPS en vez de que se produzca un ralentizamiento global del sistema. Si el dispositivo es muy lento y los FPS caen por debajo de (CICLOS_SEG / MAX_FRAMESKIP) el sistema empezará a ralentizarse de forma global. En equipos muy potentes el sistema presenta la principal ventaja de un desperdicio de la potencia de cálculo en forma de fotogramas de visualización repetidos e innecesarios, aunque esto podría evitarse con algunas mejoras. Tal y como discutiré posteriormente, XNA funciona con una versión similar a este bucle.

Velocidad de juego constante y FPS con predicción

En muchas ocasiones el juego puede funcionar correctamente con frecuencias de actualización bajas, especialmente algunos componentes como la entrada del usuario y la IA. La siguiente solución pretende que la lógica del juego pueda funcionar a bajas frecuencias de actualización y que al mismo tiempo se produzcan movimientos suaves y constantes en la pantalla. Para ello el juego se ejecuta a una velocidad constante y la visualización se realiza a la máxima frecuencia, al igual que en la solución anterior, pero la visualización realiza una interpolación del estado de los objetos en función del momento en el que se realice en relación a la actualización. Por ejemplo, si la actualización se realiza cada 30 ms, y en el momento de la visualización han pasado 15 ms desde la última actualización se le pasa un parámetro de 0.5 a la función de visualización que interpola la posición de los objetos entre frames de actualización para producir movimientos más suaves.

```
Int next_game_tick = GetTickCount();
float interpolation;

while( true) {
    int loops = 0;
    while( GetTickCount() > next_game_tick && loops < MAX_FRAMESKIP)  {
            update_game();
            next_game_tick += SKIP_TICKS;
            loops++;
    }
    interpolation = GetTickCount() + ((1000 / CICLOS_SEG) - next_game_tick )
             /((1000 / CICLOS_SEG)
    display_game( interpolation );
}
```

Esta solución tiene la principal desventaja de que es más compleja de implementar, ya que la visualización debe de realizarse teniendo en cuenta este nuevo parámetro. Más aún cuando la evolución de la posición del objeto depende de las acciones del usuario y debe de realizarse una predicción de donde se encontrará el objeto en el siguiente fotograma. Esto puede producir que ciertos objetos sean representados en posiciones incorrectas o incluso inaccesibles, ya que antes de alcanzar dicha posición se detectaría una colisión. No obstante, si el juego se actualiza a la suficiente velocidad estos errores únicamente serán mostrados una pequeña fracción de segundo que difícilmente será apreciada por el usuario.

Simulación dinámica

Si bien algunas de las propuestas analizadas anteriormente realizan un cierto desacoplo entre las fases de actualización y visualización este se realiza de forma sistemática y limitada. De igual forma, en los casos analizados todos los objetos del sistema realizan su fase de actualización a la misma frecuencia. En la práctica, no todos los objetos tienen las mismas necesidades de ejecución, por lo que está practica produce que objetos que necesiten una simulación más rápida podrían submuestrearse, mientras que objetos con menos requisitos se sobremuestrarian, con el consecuente desperdicio de potencia de cálculo. Algunas implementaciones de videojuegos, intentan paliar este efecto ejecutando diferentes aspectos del videojuego a diferentes frecuencias, por ejemplo la IA a 25 fps y la física a 60 fps, pero está clasificación se realiza en función de categorías y no de forma independiente para cada objeto.

Un cambio de paradigma a discreto desacoplado consiste en que cada entidad de la simulación se ejecute de forma independiente, con una frecuencia de muestreo adecuada a su propia frecuencia interna . De esta forma solo se realiza uso de la CPU cuando es necesario, liberando potencia de cálculo que podría o bien emplearse en aumentar la calidad de la aplicación, por ejemplo añadiendo más objetos o realizando procesos más complejos o bien simplemente liberar la CPU/GPU para reducir el consumo. Esto incluye además a la visualización que de esta forma está completamente desacoplada de la frecuencia(s) de actualización y otros aspectos como la entrada de usuario.

Roles en el desarrollo de un videojuego

Producción

El rol de productor es quizá uno de los más difíciles de definir puesto que su denominación y sus funciones varían en gran medida según cada empresa. En general, puede decirse que el productor se encarga de asegurar que el proyecto se desarrolle según lo planificado y dentro del presupuesto. El productor realiza la planificación del desarrollo e impone los plazos, organiza el trabajo, coordina al equipo y consigue y gestiona recursos.

Hay principalmente dos tipos de productores:

- **Productor externo**:
  Los productores externos son empleados por el Publisher o distribuidor del juego. Sus funciones están enfocadas principalmente a supervisar uno o varios proyectos e informar al distribuidor del estado del proyecto así como de cualquier posible problema que surja.

- **Productor interno**:
  Los productores internos trabajan para la propia desarrolladora e interpretan un rol de mayor importancia estando fuertemente implicados en el desarrollo de un solo juego.

Algunas de sus funciones pueden ser:

- Gestionar el equipo de desarrollo.
- Representar al equipo y al producto hacia el exterior.
- Negociación de contratos
- Mantener horarios y presupuestos
- Supervisar el desarrollo.

En pequeños equipos de desarrollo, el productor puede interactuar directamente con el equipo creativo y de programación, pero en equipos de mayor tamaño contactará con los jefes de programación, arte, diseño, testeo, etc. estableciéndose una jerarquía.

Lo que hace principalmente un productor es gestionar, por ello sus principales **herramientas** son herramientas de gestión:

- Suite Office (texto, hoja cálculo, bases de datos, calendario, …)
- Gestión de proyectos
- Diagramas de Gantt
- Diagramas de Pert
- Mapas conceptuales
- Reuniones, teléfono, etc.

Es un rol imprescindible en el desarrollo de un videojuego, ya sea llevado a cabo por una gran empresa o un pequeño equipo “indie”. Si el equipo no cuenta con nadie realizando dicho rol de forma específica, esté será asumido por una o varias personas que realicen sus actividades de forma más espontánea, aunque estén desempeñando otros roles.

Diseño

Los diseñadores del juego se encargan de concebir y definir la experiencia del juego, desde cómo se juega hasta los diferentes niveles, pasando por los personajes, los enemigos, etc. creando el documento de Diseño del juego. Tienen que tener una idea global del juego y todos sus detalles y saber comunicárselo al resto del equipo. Además, son los encargados de controlar que el concepto del juego se mantenga a lo largo de todo el desarrollo y de que el trabajo de todo el equipo este cohesionado.

En contra de lo que se suele pensar muchas veces, diseñar un juego no es solo “tener la idea”, de hecho en muchos casos puede que la idea en la que esté trabajando un diseñador no sea suya. El trabajado de un diseñador es un trabajo de refinamiento. Sobre una idea bruta inicial, la va puliendo, extrayendo todo lo innecesario y dejando las partes adecuadas, manteniendo la esencia del juego.

Imagen 3: Hideo Kojima, creador de la saga Metal Gear

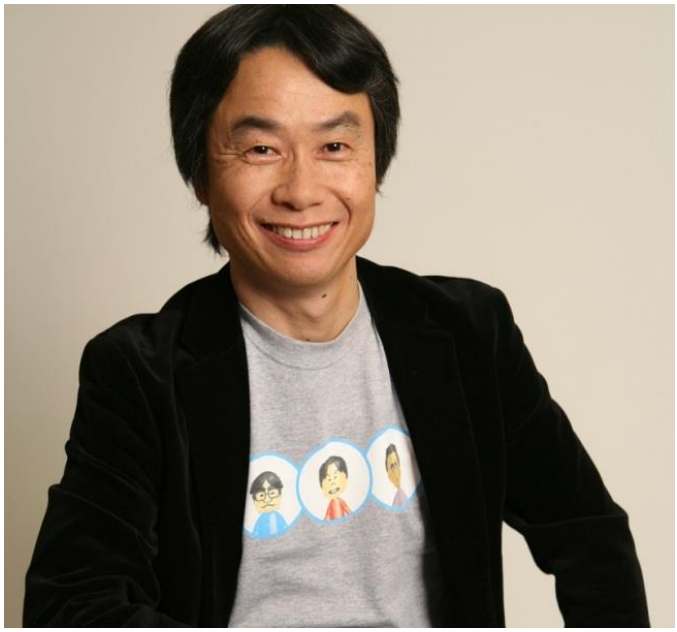


Imagen 2: Shigeru Saito, creador de Mario, Donkey Kong y Zelda

El trabajo del diseñador se extiende a lo largo de todo el proceso de desarrollo. Comienza muy temprano, creando y definiendo el concepto del juego. Si el proyecto continua, el diseñador pule y extiende el concepto del juego para elaborar el documento del diseño y un guion si es necesario. Cuando ya ha comenzado el desarrollo del juego el diseñador tiene que seguir puliendo y definiendo mecánicas, diseñando niveles y enemigos, creando diálogos, etc. además de mantener la comunicación con el resto del equipo para asegurarse que todos los desarrolladores entienden y mantienen la esencia del juego. Para finalizar, en las últimas fases del desarrollo tienen que asegurarse que el juego está bien balanceado y la experiencia de juego es buena, haciendo los ajustes necesarios.

Las herramientas que utiliza un diseñador son:

- Papel y lápiz.
- Suite Office (Procesador de texto, hoja de cálculo, etc.)
- Mapas conceptuales y diagramas
    - CMapTools
    - Visio
- Prototipado y Mockups:
    - Flash
    - Unity
    - GameMaker
- Editores de niveles
- Programación con scripts

Ser diseñador exige una gran capacidad creativa para crear muchas ideas nuevas pero también entender y considerar las restricciones técnicas.

Programación

El rol de programador es uno de los que emplea a más personas en el desarrollo de un videojuego, junto con el de grafista. Por una parte, los programadores deben encargarse de desarrollar el software que una todos los elementos desarrollados por el resto del equipo e implemente la funcionalidad requerida por los diseñadores. Por otra parte deben desarrollar herramientas, como editores de niveles o paquetes gráficos que permitan al resto del equipo desarrollar los contenidos.

Los programadores empiezan a trabajar en fases muy tempranas del proyecto. Al principio, durante la fase de diseño colaboran con los diseñadores para crear prototipos jugables que ayuden a definir las mecánicas y el alcance del juego. Posteriormente, si el proyecto lo requiere, tendrán que preparar o adaptar el motor del juego para que cumpla con los requisitos y desarrollar aquellas herramientas necesarias. Durante la parte principal del desarrollo el programador trabaja junto con los diseñadores para desarrollar la lógica del juego y va encajando todas los recursos del resto del equipo. En la parte final del proyecto el trabajo se centra en la optimización para mejorar el rendimiento y en la corrección de bugs.

En proyectos grandes, dada la diversidad y complejidad de las tareas que realizan los programadores se dividen según la tarea que realizan:

- Programador junior
- Programador jefe
- Programador de herramientas
- Programador de motor
- Programador gráfico/efectos especiales.
- Programador o Ingeniero de audio.
- Programador de inteligencia artificial.
- Programador de redes

Grafismo

Es uno de los roles que más gente necesita en el desarrollo de un videojuego, junto con el de programación. Los grafistas son los responsables de diseñar y crear todos los contenidos que aparecen en pantalla, desde los personajes y los escenarios hasta los menús y la interfaz de forma que el jugador sea capaz de sumergirse en el mundo que se ha creado para él y se consiga la experiencia de juego deseada.

Al comienzo del desarrollo, los grafistas trabajan junto con los diseñadores para definir el estilo gráfico característico del juego y definir el aspecto visual de todos sus elementos de forma que el mundo creado esté cohesionado. En esta fase se realizan las pruebas de arte conceptual. Más adelante, en la fase de producción, los grafistas van creando poco a poco los recursos gráficos, siempre basándose en el diseño inicial y manteniendo el estilo definido. Los recursos gráficos pueden ser de muy diferente naturaleza, desde gráficos de personajes 2D o texturas hasta modelos 3D de personajes o escenarios. Por último, los grafistas también se dedican a reparar bugs visuales y revisar la imagen de conjunto, así como otras actividades secundarias que no son directamente parte del videojuego como la caja del juego, las instrucciones o los

tráileres.

Imagen 4: Concept art para el diseño de un personaje

Las herramientas de los grafistas abarcan un gran abanico de tipos de programas:

- Programas de dibujo: Photosop, Gimb, Illustrator, PaintTool SAI
- Programas de modelado: 3DStudio Max, Blender, Maya
- Editores de niveles
- Programas de tratamiento de video: Adobe Premiere

Dada la diversidad de las tareas, en proyectos complejos los grafistas se especializan en:

- Artista conceptual
- Artista de texturas
- Modelador 3D
- Animador 3D
- Creador de sprites 2D
- Animador 2D

Fases en el desarrollo de un videojuego

Concepto

Se define la idea general del videojuego, describiendo, sin profundizar mucho, sus características más importantes y los aspectos más diferenciadores. Todo lo mencionado se recoge en el **documento de concepto**, que será la base para el diseño del juego y puede usarse además para realizar presentaciones del juego y/o buscar financiación. Más concretamente, el documento de concepto puede contener:

- Concepto de alto nivel.
- Mecánicas de juego.
- Ambientación y arte conceptual.
- Plataformas hardware objetivo.
- Público objetivo.
- Análisis competitivo.
- Análisis de riesgos.

Pre-producción

Se refina y desarrolla el concepto del videojuego creando una descripción completa y más detallada del mismo que constituye el **documento de diseño**. Este debe incluir todos los detalles necesarios para poder construir el videojuego. Además, se deciden las herramientas y metodologías que se emplearán durante el desarrollo como lenguaje de programación, librerías y motor del juego.

Para ayudar a definir el diseño de un videojuego, evaluar su potencial y detectar posibles problemas tecnológicos se desarrollar uno o varios **prototipos** que capturan la esencia del videojuego e implementar aquellos aspectos más conflictivos desde un punto de vista tecnológico. Además, se desarrollan todas aquellas **herramientas** que se vayan a utilizar durante la fase de producción.

Con todo esto, se realiza una primera estimación de costes y tiempo y se crea en **plan de proyecto** que indica cómo se va a desarrollar el videojuego. Este está formado a su vez por:

- Plan de personal.
- Plan de recursos.
- Presupuesto.
- Cronograma de desarrollo.

Por último se define como va a ser implementado el software y su arquitectura mediante diferentes herramientas de la ingeniería del software, normalmente diagramas UML, y se recoge en el **documento de diseño técnico**.

Si se trabaja con un distribuidor, esta fase puede determinar si se continua con el proyecto o no.

Producción

Es la fase que más personal requiere. Se crean todos los **contenidos** del juego:

- Código del juego.
- Arte gráfico.
- Sonidos.
- Niveles.

Aunque no es el objetivo de esta fase, las horas de trabajo sobre el proyecto pueden hacer que algunas características cambien o se eliminen, o incluso que se introduzcan algunas nuevas. Por tanto hay que completar y actualizar los documentos generados previamente. Puesto que el desarrollo de un videojuego es algo muy complejo, y no todo se puede considerar y tratar desde el principio, es importante entender el desarrollo como un proceso orgánico y saber adaptarse con presteza a las dificultades e imprevistos que ocurran.

Suelen definirse, sobre todo si se trabaja con una distribuidora, un conjunto de **milestones** o hitos, que no son más que un conjunto de características a ser implementadas para una fecha determinada. Algunos de los más comunes son:

Alfa: Toda la funcionalidad del juego y todas sus características están programadas y medianamente optimizadas y el juego puede jugarse de principio a fin, incluyendo todas las misiones extras y todos los niveles de dificultad. No obstante, quedan detalles por arreglar o mejorar y los gráficos del juego y sonidos no están completos.

Beta: Se terminan e integran todos los contenidos y se pone especial énfasis en arreglar el mayor número de errores posible. También se ajustan aquellos pequeños parámetros que influyan en la jugabilidad, como daños de las armas, cantidad de vida, etc.

Cierre de código: En programación, cualquier pequeño cambio añadido puede añadir y multiplicar los errores. Por ese motivo, hay un punto en el desarrollo en el que se decide que ya no se cambiará ninguna característica del código nada más que para corregir errores con el fin de que no aparezcan nuevos errores.

Post-producción

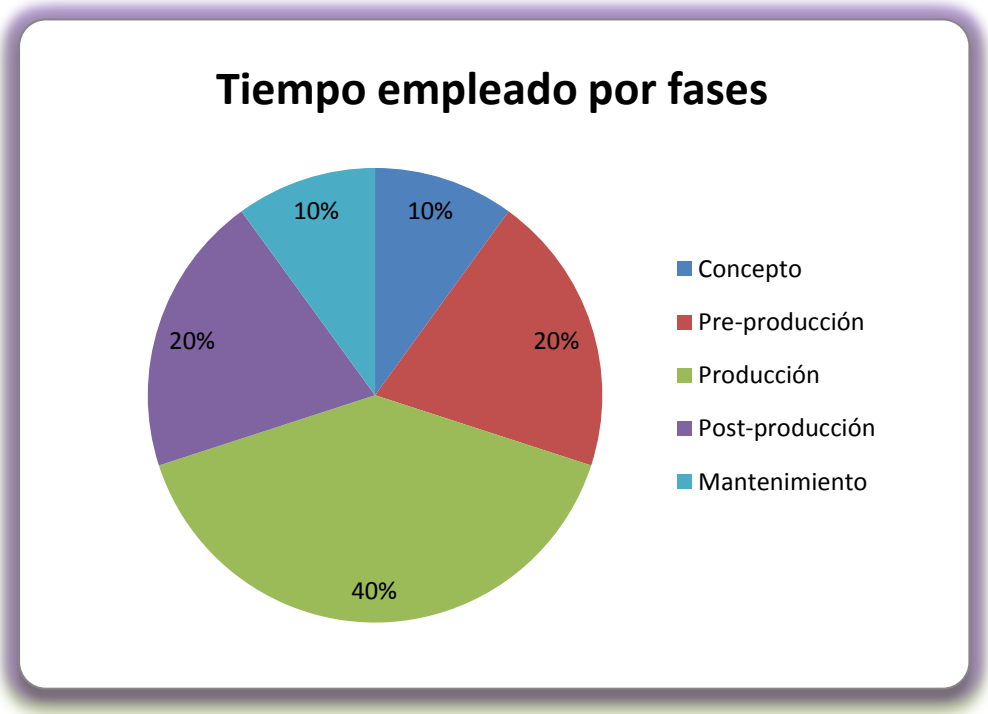
Durante la post-producción el juego es probado de forma masivamente con objeto de detectar y subsanar todos los errores por pequeños que sean. Hablamos no solamente de errores de programación, también errores gráficos, como “agujeros” en las texturas, errores de doblaje, textos que aparezcan ligeramente desplazados de su posición, etc.

Se pueden realizar dos tipos de pruebas. En primer lugar están las **pruebas alfa**, pruebas que se realizan internamente, con los propios miembros del equipo de desarrollo y en segundo lugar están las **pruebas beta**, que se realizan con personas externas al equipo.

Por último se prepara todo el material necesario para marketing y se obtienen las certificaciones necesarias para que el juego pueda ser distribuido.

Mantenimiento

Una vez el juego a salido al mercado, los jugadores pueden encontrar errores que habían pasado inadvertidos, especialmente en PC, donde las combinaciones posibles de hardware son enormes. Además, en juegos multijugador, los jugadores pueden estar jugando de formas que no se habían pensado y es necesario realizar ajustes. Para estas correcciones se utilizan los **parches**. De similar manera, los desarrolladores pueden decidir añadir contenido nuevo al juego mediante **actualizaciones**.



Metodologías de desarrollo de un videojuego

Como producto software que es un videojuego pueden aplicarse todas las metodologías y herramientas propias del desarrollo del software convencional, no obstante un videojuego tiene muchas características propias diferenciadoras del software convencional que hay que tener en cuenta y que hará que los métodos tengan que ser adaptador y que algunos resulten más adecuados que otros.

Típicamente existen dos tipos de metodologías:

- Deterministas: Predicen y planifican todo el desarrollo desde el comienzo intentando minimizar los cambios durante el desarrollo. Ejemplo de esta vertiente es el desarrollo **Waterfall**.
- Ágiles: Asumen que todo puede cambiar y que puede hacerlo rápidamente por lo que inciden en adaptarse a los cambios con rapidez y eficacia. Ejemplos son **Scrum**[SCRUM] y **eXtreme Programming**[XP].

Cada una de estas metodologías inciden en aspectos contrarios:

Deterministas	**Ágiles**
Procesos y herramientas	Individuos e sus interacciones
Documentación exhaustiva	Software en funcionamiento
Negociación de un contrato	Colaboración del cliente
Seguir un plan	Respuesta al cambio

Por supuesto no tienen por qué seguirse unas metodologías u otras y es posible realizar cualquier tipo de mezcla de ellas.

Waterfall

Fue la primera metodología en adoptarse y la más utilizada a día de hoy en el desarrollo de videojuegos sobretodo en compañías de gran tamaño. Se basa en un desarrollo secuencial que atraviesa las fases de Requisitos, Diseño, Implementación, Verificación y Mantenimiento.

Ventajas:

- Facilidad de adopción por el equipo de desarrollo ya que requiere poca implicación y trabajo extra por su parte.
- Facilita la administración del proyecto ya que esta se centraliza y se sabe en todo momento en que fase del desarrollo nos encontramos.
- Fácilmente escalable a cualquier número de desarrolladores y aplicable a equipos grandes.
- Especialmente útil en proyectos de baja capacidad creativa como pequeñas modificaciones, secuelas o expansiones de juegos.

Inconvenientes:

- Pueden producirse juegos menos divertidos y de menos llamativos para los jugadores ya que todo el juego tiene que estar diseñado desde el principio y no se pueden hacer cambios en base a los resultados a lo largo del desarrollo.
- No responde bien ante cambios, ni tecnológicos y de requisitos. El coste de hacer un cambio crece conforme va progresando el desarrollo.
- Para dar el videojuego por terminado es necesario que se hayan completado totalmente todas las fases, por lo que es más probable que el proyecto fracase por una mala planificación que en otras metodologías.
- Mala comunicación entre las distintas áreas ya que esta se suele realizar a través de documentos que son sensibles a errores de interpretación.
- Existe poca interacción con los jugadores ya que estos no pueden intervenir hasta las últimas fases del desarrollo por lo que es más difícil saber si el juego es divertido y transmite las sensaciones requeridas.

Scrum

Scrum es una metodología que enfatiza en la gerencia para el desarrollo de software de forma iterativa e incremental y no indica prácticas específicas a seguir durante el desarrollo. Es por ello que en muchas ocasiones se emplea conjuntamente con XP y que permite que el proceso se adapte a cualquier tipo de proyecto.

Roles en Scrum:

- **Dueño del producto**: Es el responsable del proyecto. Gestiona la **Pila de producto**, es decir, establece los requisitos que debe cumplir el juego, los especifica de forma clara y los ordena según su prioridad. Debe mantener la Pila de producto actualizada durante todo el proceso de desarrollo y es el responsable de que el equipo entienda todos los detalles del juego.
- **El equipo de desarrollo**: Desempeña todo el trabajo necesario para en cada **Sprint** entregar un incremento del producto. Los equipos de desarrollo de Scrum son auto-organizados y gestionan su propio trabajo, solo ellos deciden que hacer en cada iteración y cómo hacerlo. Aunque los miembros del equipo son multifuncionales y pueden desempeñar tareas distintas (programador, grafista, diseñador, etc.) la responsabilidad del desarrollo recae en todo el equipo como un todo. Estos equipos, óptimamente están formados por de 3 a 9 miembros.
- **El Scrum Máster**: Asegura que el equipo entiende y emplea Scrum.

En primer lugar, el **Dueño del producto** crea la **Pila de producto**, que no es más que una fuente de requisitos en forma de lista ordenada; con todas las características conocidas del producto. Después se suceden una serie de iteraciones o **Sprints** realizadas por el **equipo de desarrollo**. Un **Sprint** es un bloque de tiempo de entre 2 y 4 semanas durante el que se desarrolla alguna característica nueva del juego y se obtiene una versión del juego utilizable y potencialmente entregable. Para ello en primer lugar, el **equipo de desarrollo** realiza una reunión de **planificación de sprint** en la que se decide que parte de la **Pila de producto** y cómo va a ser desarrollado durante el **Sprint** y se recoge en la **Pila de Sprint.** Durante el **Sprint**, el equipo mantiene pequeñas reuniones diarias, o **Scrum diario** en el que cada miembro informa al resto del equipo del trabajo conseguido, el trabajo planificado y posibles problemas que hayan surgido. En base a la experiencia adquirida durante en **Sprint**, en cualquier momento se puede hacer una replanificación del **Sprint**, se pueden añadir o quitar características a la **Pila de Sprint** de la **Pila de Producto** y se pueden añadir o quitar características de la **Pila de Producto**. Finalmente el equipo discute con todos los interesados sobre lo que se ha conseguido durante el Sprint en la **Revisión del Sprint** y estudia las técnicas empleadas y las adapta en base a los resultados en la **Retrospectiva del Sprint**.

Ventajas:

- Al ser iterativo e incremental se obtienen versiones jugables del producto a intervalos regulares de tiempo lo que permite una mejor comunicación con el cliente y los usuarios.
- Al disponer siempre de una versión jugable se facilita que el proyecto termine prematuramente para adaptarse a las circunstancias.
- Mayor motivación por parte del equipo gracias a su gran implicación en el proyecto.
- Permite abandonar características que resultan poco prometedoras conforme se van desarrollando así como incorporar nuevas características que se van descubriendo.
- Especialmente útil para juegos más experimentales.

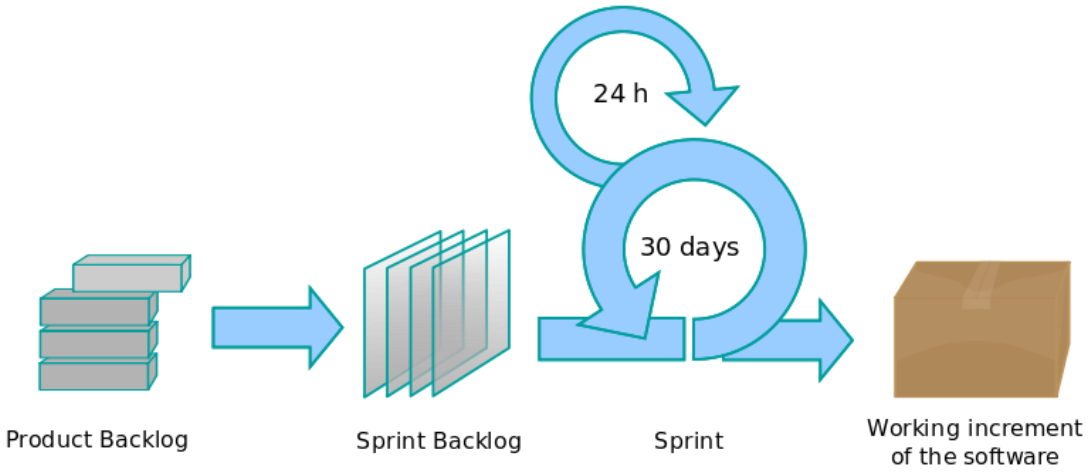


Imagen 5: Metodología de trabajo Scrum

Inconvenientes:

- Difícil de adoptar y de dominar, especialmente en equipos grandes.
- No genera tanta documentación como otras metodologías.
- En proyectos muy largos es posible que se pierda la perspectiva y la dirección y que el proyecto no termine.
- Es necesario involucrar profundamente al cliente en el desarrollo, lo que no siempre es posible

Extreme Programming

Extreme Programming básicamente es una colección de valores, reglas y buenas prácticas que han sido utilizadas por la industria durante muchos años. Extremme Programming las identifica y agrupa para que sean usadas en conjunto y obtener así el mayor beneficio.

Se basa en 5 valores:

- Comunicación: Todo el mundo forma parte del equipo y trabajan juntos en todas las fases del desarrollo. hay una fuerte interacción entre todos los miembros con reuniones diarias.
- Simplicidad: Hacer lo necesario y lo que se ha pedido, pero nada más. El progreso hacia la solución final se realiza mediante pequeños pasos.
- Respeto: Todo el mundo merece reconocimiento dentro del equipo, se respeta el trabajo de los demás, por ejemplo no haciendo cambios innecesarios en el código de otros.
- Valor: Hacer los cambios necesarios sin tener miedo a que el código deje de funcionar.
- Retroalimentación: Enviar copias funcionales del proyecto al cliente con cada iteración y mantener comunicación constante de lo que le gusta y lo que no.

Una de las prácticas comunes es la programación por parejas, en la que dos programadores trabajan simultáneamente en el mismo ordenador, discutiendo las mejores soluciones y revisando el código.

Ventajas:

- Distribución del conocimiento y las responsabilidades entre todo el equipo.
- La fuerte interacción con el cliente y los jugadores permiten alcanzar un alto grado de satisfacción de ambos.
- Entrenamiento de programadores con menos experiencia por parte de los que tienen más experiencia mediante la programación por parejas.

Inconvenientes:

- Algunas de las prácticas son difíciles de adoptar.
- Las prácticas están pensadas principalmente para programadores y no necesariamente consideran otros roles como grafistas y diseñadores, imprescindibles en el desarrollo de un videojuego.
- Difícil aplicación en equipos de trabajo grandes.

Mi metodología

Después de analizar las diferentes metodologías más empleadas en el desarrollo de videojuegos, tanto ágiles como clásicas, opté por hacer una aproximación más sencilla a la metodología presentada en [SOFT_ENG] por John P. Flynt, pero prescindiendo de algunas etapas y algunos de los artefactos que he considerado menos importantes en un proyecto tan pequeño como el que no ocupa. Dicha metodología es una aproximación hibrida entre metodologías ágiles y metodologías deterministas.

Se parte del diseño del juego completo y completamente especificado en el documento de diseño del juego que hará la vez de especificación de requisitos. Inicialmente se divide el desarrollo del juego en Iteraciones, donde cada iteración añade una serie de funcionalidades requeridas por el videojuego al producto total, hasta llegar a la versión completa. Para cada una de las Iteraciones se especifican todas las características que se esperan desarrollar en ella y se realiza una estimación de su coste. Con todo ello ya es posible realizar una planificación de todo el proceso de desarrollo y obtener una primera estimación del tiempo total de desarrollo.

Una vez conforme con la división en Iteraciones del proyecto, se realiza el diseño del software del juego completo mediante diagramas UML, pero a diferencia de en otras metodologías, John P. Flynt propone realizar el diseño iteración a iteración, esto es, un diagrama de clases para cada iteración. Como no se termina el diseño hasta que se ha realizado el diseño de cada una de las iteraciones se consigue coherencia y que el diseño sea sólido, pero el realizarlo de esta manera proporciona flexibilidad y que la atención del diseñador se centre en realizar mejoras locales en la arquitectura.

Una vez terminado el diseño se realiza la implementación de las diferentes iteraciones de forma secuencial, hasta terminar el desarrollo, de manera que no se empieza con una iteración hasta que no se ha terminado completamente la anterior. Para cada una de las iteraciones se realiza una fase de codificación y una fase de pruebas, donde se realizan tanto pruebas de software como pruebas de jugabilidad, pudiendo cambiar el diseño del juego.

Si bien, esta fue la metodología que pensé en seguir en un principio, la falta de experiencia en algunas tareas hizo que la metodología no se siguiera de forma estricta y se realizaran algunos ajuntes. Por ejemplo, no puede realizar el diseño del software completo con anterioridad a la codificación.

Herramientas de desarrollo

XNA

A la hora de decidir que videojuego se va a realizar hay que tener en cuenta simultáneamente 3 aspectos: El propio diseño del juego, la plataforma en la que va a funcionar y la tecnología empleada en su desarrollo, puesto que están completamente ligados y una decisión en 1 de los aspectos limita o enfoca al resto. Desde el principio teníamos completamente claro que queríamos hacer un videojuego que se controlara con un mando y no con teclado y ratón o una pantalla táctil. Además, quería centrarme en la parte de creación de un videojuego, realizando una programación al mayor nivel de abstracción posible del hardware y de los detalles más técnicos y que el desarrollo fuera rápido. A todo ello hay que unirle además nuestra intención de obtener un producto que pudiéramos distribuir fácilmente y con el menor coste posible.

Todas nuestras exigencias, nos llevaron a **XNA**. XNA es un Framework gratuito para el desarrollo de videojuegos que proporciona un amplio conjunto de librerías de clases específicas para el desarrollo de juegos. XNA compila a un código intermedio, Common Intermediate Language, que se ejecuta en una variante de la Common Language Runtime que está optimizada para juegos. Al ejecutarse en la CLR, técnicamente los juegos podrían escribirse en cualquier lenguaje de la plataforma .NET, aunque oficialmente solo está soportado C#; por lo que este ha sido el lenguaje empleado en el desarrollo. XNA nos permite el desarrollo de videojuegos para Xbox 360, PC y Windows Mobile.

XNA encapsula las características técnicas de más bajo nivel propias de la codificación de un juego, haciendo que el programador únicamente tenga que preocuparse de los detalles de más alto nivel. Por una parte, proporciona carga y gestión de gráficos 2D y 3D y se encarga de su renderizado, por lo que el desarrollo del proyecto no me ha conllevado ninguna programación gráfica. De forma similar, proporciona clases para la carga y gestión de sonidos y la interacción son el usuario, ya sea a través del teclado, una pantalla táctil y el mando de la Xbox.

XNA además facilita la distribución de nuestro videojuego ya que los juegos desarrollados en XNA pueden ser subidos a Xbox Live Indie Games por los desarrolladores donde millones de usuarios de Xbox 360 podrán acceder directamente a ellos desde la consola. El procedimiento es el siguiente: En primer lugar el desarrollador debe registrarse como usuario Premium del XNA Creators Club, con un coste de 99$ anuales. Una vez se quiere subir el juego, este entra en un proceso de revisión, donde son otros desarrolladores independientes con cuanta Premium de XNA Creators Club quienes prueban nuestro juego y lo validan o rechazan si tiene bugs. Cuando el juego finalmente es aprobado, podemos decidir el precio de venta, y Microsoft gestiona las ventas y nos ingresa periódicamente el 70% de los beneficios.

Las principales desventajas es que no es totalmente multiplataforma (Aunque existe una implementación de XNA no oficial que soporta más plataformas como Linux, Mac, Android, IOS,

etc), que al funcionar sobre la CLR el rendimiento no es tan elevado como podría serlo en juegos desarrollados con C++ y que la cuota por publicación es más elevada si la comparamos por ejemplo con Android, donde es de 30$ una única vez.

Visual Studio

El IDE proporcionado para el desarrollo en XNA es el conocido **Visual Studio** que proporciona toda suerte de facilidades para la codificación y la depuración. Desde su primera versión en 2006, XNA ha ido evolucionando con diferentes versiones, actualizándose de igual manera la versión de Visual Studio compatible con XNA. Desde Visual Studio 2005 para la primera versión hasta Visual Studio 2010 en la versión 4.0 de XNA, su última versión. Debido a que XNA es gratuito y que no se poseían excesivos conocimientos previos, se decidió utilizar su última versión, la 4.0 y Visual Studio 2010. Como siempre se ha tenido en mente la posibilidad de vender el juego fruto del proyecto, se optó inicialmente por utilizar la versión gratuita de Visual Studio compatible con C#, Visual Studio C# 2010 Express. No obstante, como ya discutiré posteriormente, para la integración con RT-DESK se utilizó la versión profesional que permite crear soluciones uniendo proyectos que usen diferentes lenguajes de programación.

Control de Versiones

Un sistema de control de versiones te previene de la posibilidad de destruir parte de tu trabajo, manteniendo versiones de tu código anterior y permitiendo que varias personas trabajen en el mismo proyecto de forma simultanea asegurándose que todos trabajen siempre con las versiones más actualizadas de los ficheros y evitando que el trabajo de unos interfiera con el trabajo de otros.

Para este cometido se decidió utilizar **Subversion**, un sistema de control de versiones libre bajo licencia Apache/BSD ampliamente utilizado y que ya se encontraba instalado en los servidores del DSIC. Para acceder al repositorio de Subversion hace falta un cliente instalado en todas las maquinas que quieran acceder al repositorio. Como cliente de Subversion se ha utilizado TortoiseSVN que provee integración con el explorador de Windows, por lo que su uso es muy intuitivo para la gente familiarizada con dicho sistema operativo.

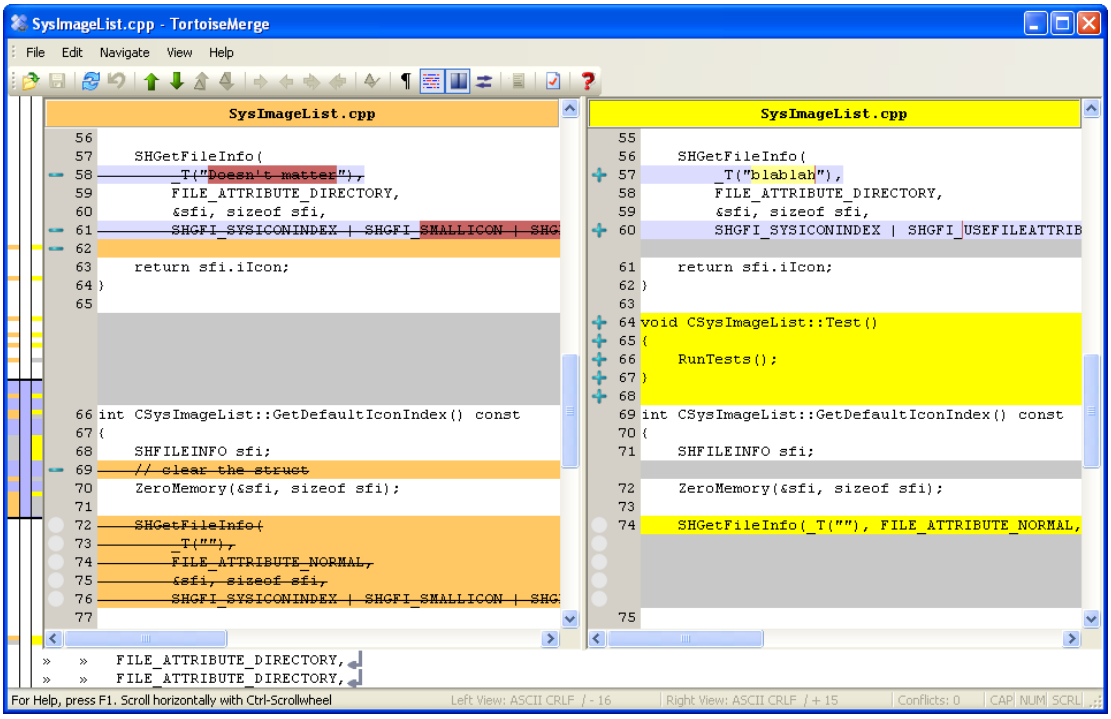


Imagen 6: Diferencias entre dos versiones del mismo archivo utilizando TortoiseSVN

Doxygen

Doxygen es una herramienta que genera automáticamente documentación externa a partir de código fuente documentado según una sintaxis específica, lo que ayuda a que la documentación sea siempre consistente con el código. Soporta entre otros lenguajes C, C++, C#, Java, Python y Fortran, siendo la herramienta más utilizada para este fin.

Diseño del juego

Anyone can make the simple complicated. Creativity is making the complicated simple.

-Charles Mingus

Descripción del juego

Olympic Punch es un juego multijugador competitivo , de 2 a 4 jugadores que une plataformas y lucha bajo el contexto de la mitología griega clásica modernizada a la época actual. En Olympic Punch controlarás a un dios griego con aspecto gracioso y modernizado y competirás con el resto de dioses griegos controlados por otros jugadores a los que tendrás que derrotar. Para derrotarlos tendrás que ir recogiendo plegarias que irán apareciendo por el escenario, que es de donde los dioses griegos sacan su fuerza; y que te permitirá realizar ataques que dañarán al resto de jugadores o los fastidiarán de alguna forma. Los jugadores irán corriendo continuamente por el escenario en busca de ser el primero en recoger la plegaria. Además, podrán robarse las plegarias unos a otros saltándose encima.

Olympic Punch proporciona partidas de corta duración, de 5 min aproximadamente, de ritmo rápido y frenético que harán que los jugadores permanezcan en tensión a lo largo de la partida, y con una componente caótica que hará que las partidas no se decidan hasta el último momento. El juego va dirigido especialmente a los grupos de amigos que se reúnen en un mismo sitio para jugar, especialmente a aquellos entre 13 – 30 años, de forma que el juego debe provocar los piques entre los amigos. Para una descripción más detallada consultar el documento de diseño incluido en los apéndices.

Historia

Zeus convoca a los dioses del Olimpo en una reunión. Les explica, que en la situación de decadencia que se encuentra su religión, se ha dado cuenta de que la solución y la fórmula para volver a ser aclamados por los mortales, es convertir su religión politeísta en una religión monoteísta, como la religión católica o musulmana. Así, él pretende convertirse en el único dios al que adorar.

El resto de los dioses responden a esta proposición con enfado e indignación, y cada uno de ellos reclama a Zeus el derecho a ser ellos el único dios, pues ellos también tienen poderes y son deidades.

A partir de ahí, se genera una discusión que irá calentándose hasta convertirse en una batalla campal entre los dioses helénicos para conseguir la supremacía sobre la religión, los mortales y el mundo entero.

Juegos similares

Smash Bross (Nintendo 64 / GameCube / Wii):

Es quizás el juego más popular en el que se enfrentan hasta 4 jugadores en el mismo escenario. Une todos los personajes de Nintendo de otros juegos para enfrentarse. El juego cuanta ya con tres entregas, por lo que es un juego de éxito.

Mario Wars (PC)

Juego Independiente gratuito que imita a Super Mario pero desde una perspectiva multijugador competitiva en el que los jugadores se enfrentan nada más que saltándose encima. Juego de escaso éxito y reconocimiento, pero muy divertido y con potencial

Playstation allstars battle royale (Playstation 3)

La respuesta de Sony al Super Smash de Nintendo, dos generaciones de consolas después. Básicamente copia la misma jugabilidad pero utilizando los personajes de Sony. El hecho de que Sony repita la misma fórmula que Nintendo nos indica que es un género con éxito.

Estética

El apartado gráfico de Olympic Punch se basa en la cultura pop japonesa, inspirado de la estética 'kawaii' (que significa mono o adorable en japonés), donde predominan los colores vivos y los personajes se representan como chibis, dibujos desproporcionados con la cabeza muy grande en comparación al resto del cuerpo, trazos más bien redondeados y extremidades pequeñas. Todo ello para denotar ese toque de ternura que ofrece la estética 'kawaii'.

Así, en el juego todos los personajes son 'chibis', para dar también un aspecto humorístico. Esta misma estética colorida y amable se encuentra también, como es lógico, en los menús y en el HUD, para mantener la cohesión.

Así se juega

Cada jugador controlará a un dios griego de aspecto y rol modernizado enfrentándose a hasta 3 jugadores en un escenario consistente en una pantalla fija, es decir no hay desplazamiento de la cámara. En principio el jugador podrá elegir su dios de entre 8 disponibles, aunque solo se han llegado a implementar 4. Cada dios, contará con 3 ataques diferentes característicos y ligados a su rol antiguo y moderno y con un coste en plegarias diferente.

Inicialmente, se planteó la posibilidad de que cada dios contará con un número de ataques mayor, y que en el momento del jugador de seleccionar personaje, este además pudiera seleccionar 3 ataques entre el total de ataques, pero esta idea pronto se descartó para conseguir mayor sencillez y reducir costes.

El jugador podrá:

- Moverse horizontalmente.
- Saltar. Al pulsar el botón de salto el personaje comenzará a saltar, si el botón se mantiene pulsado el salto será más largo. Cuando el personaje esté en el aire, el jugador seguirá pudiendo controlar su movimiento.
- Lanzar los tres ataques, si tiene plegarias suficientes, cada uno asignado a un botón.
- Bloquear. Una vez pulsado el botón de bloquear, durante un periodo de tiempo muy breve de tiempo saldrá un escudo de energía alrededor del personaje. Durante este tiempo los ataques que sufra le causarán menos daño y puede evitar algunos efectos como parálisis o ser lanzado por los aires. Como el escudo dura muy poco tiempo y tiene mucho retraso el jugador tendrá que usarlo en el momento justo. Esta opción añade al juego mayor dificultad y una curva de aprendizaje más pronunciada, que hace que el juego sea más difícil de dominar.

El juego contará con los siguientes modos de juego:

- El modo Arcade, es un modo para un solo jugador. En él, el jugador deberá ir derrotando consecutivamente a todos los dioses con dificultad creciente.
- El modo multijugador permite enfrentarse hasta a 4 jugadores desde una misma consola. Se podrán elegir varios modos de juego: A vidas, a tiempo, etc. Así como configurarlos: nº de vidas, tiempo, aparición de objetos, etc. Los jugadores podrán formar equipos añadir personajes controlados por la IA y elegir su dificultad. Por último, los jugadores podrán elegir escenario.

Inicialmente, el juego iba a tener 4 modos de juego. Aparte de los dos citados anteriormente, el juego iba a contar con un modo Arcade Hardcore, muy similar al modo Arcade, pero con más personajes y mayor dificultad. Por simplicidad y tiempo, finalmente se decidió eliminarlo. De igual forma, se planteó la posibilidad de desarrollar un modo multijugador online, en el que los jugadores podrían enfrentarse a través de internet, pero el coste de implementación resultaba excesivo. Por razones de tiempo tampoco pudo implementarse la IA. Consultar el documento de diseño para más información.

Personajes

Se realizó el diseño de 8 personajes jugables: Zeus, Hades, Artemisa, Afrodita, Ares, Poseidón, Hefesto y Apolo, pero por motivo de exceder el tiempo del proyecto finalmente se implementaron 4. De igual forma, de los 3 ataques por personaje tenían pensado incluirse solo pudieron incluirse 2. A continuación solo se muestran los personajes y ataques desarrollados, para conocer más detalles y conocer todos los personajes consultar el anexo. Cada uno de los personajes, presenta su rol clásico como dios griego, y un nuevo rol actual para darle mayor personalidad.

Zeus, dios del rayo, CEO multimillonario

Soberano de hombres y dioses, rey del Olimpo, dios de la luz, del cielo sereno y el rayo. Padre de todos, Preside las manifestaciones celestes (lluvia, rayo, relámpago, trueno...), mantiene el orden y la justicia en el mundo, no se deja dominar por sus caprichos (A excepción de los amorosos).

Imagen 7: Diseño final de Zeus

Imagen 8: Bocetos previos

Como dios del rayo, Zeus controla completamente la electricidad y puede utilizarla a su favor para realizar todo tipo de ataques. Zeus es un personaje con una jugabilidad general que cuenta tanto con ataques de un solo objetivo como ataques en área. Además, cuenta con un ataque que le permite aumentar su fuerza a coste de su vida.

- Rayo: Ataque principal de Zeus. Lanza un rayo que avanza desde la posición de Zeus hacia delante hasta que choque con algo o alguien. El rayo se propagará hacia adelante aunque de manera muy rápida. Otra posibilidad es que el rayo no se detenga cuando choque con alguien y que pueda dañar a varios enemigos.
- Carga eléctrica: Zeus acumula energía eléctrica sobre si mismo, cargando su cuerpo de electricidad como una pila y electrificándose por un breve periodo de tiempo. Mientras este electrificado los ataques de Zeus serán más fuertes y los enemigos que toque sufrirán pequeñas descargas que les causarán daños. Como contra partida, mientras este electrificado perderá un poco de vida a cada instante. Zeus se descargará finalmente tras haber utilizado una determinada cantidad de energía o tras un breve periodo de tiempo. Una buena estrategia para este personaje será guardar bastantes puntos de energía para utilizarlos una vez este cargado.

Hades , Dios del Inframundo, gótico

Dios del inframundo. Sus dos hermanos eran Zeus (el menor de todos) y Poseidón. Hades es el único dios que no obtiene su poder de las plegarias de los mortales, si no de las almas de los muertos. Así pues, Hades no puede utilizar su poder para atacar. En su lugar, Hades utiliza el poder de las almas. Cuenta con un ataque que le permite obtener parte de las almas de sus enemigos para posteriormente utilizar su poder. Hades contará además con la ayuda de su perro de tres cabezas Cerbero.

Imagen 10: Diseño final de Hades

Imagen 9: Bocetos previos

- Cosecha de alma: Hades extiende una gran mano fantasmal que puede entrar en el corazón de sus enemigos y arrebatarles un pedazo de su alma. Es el único ataque de Hades que o bien utiliza plegarias, o bien no tiene ningún coste. Este ataque le permite obtener energía a Hades.
- Ataque de cerbero: Un ataque controlado de larga distancia. Cerbero aparece ante Hades cargando a toda velocidad contra el enemigo y permanecerá en el juego durante un breve periodo de tiempo. Durante ese periodo no parara de correr en ningún momento. Mientras dura el ataque, el jugador no podrá realizar otros ataques con Hades, si bien si podrá moverse y saltar. A cambio, con sus botones de acción, el jugador podrá controlar a Cerbero. Tendrá un botón para saltar, un botón para atacar y un botón para cambiar la dirección. Si Cerbero llega a una pared, automáticamente cambiará de dirección. Se trata de un ataque muy eficaz y destructivo pero muy difícil de controlar, sobre todo, si el jugador no quiere dejar quieto a Hades y que sea un blanco fácil.

Artemisa, Diosa de la caza, arquera infalible

Señora de los Animales. Hermana melliza de Apolo. Fue la diosa helena de la caza, los animales salvajes... A menudo se la representaba como una cazadora llevando un arco y flechas. El ciervo y el ciprés les estaban consagrados.

Imagen 11: Diseño final Artemisa

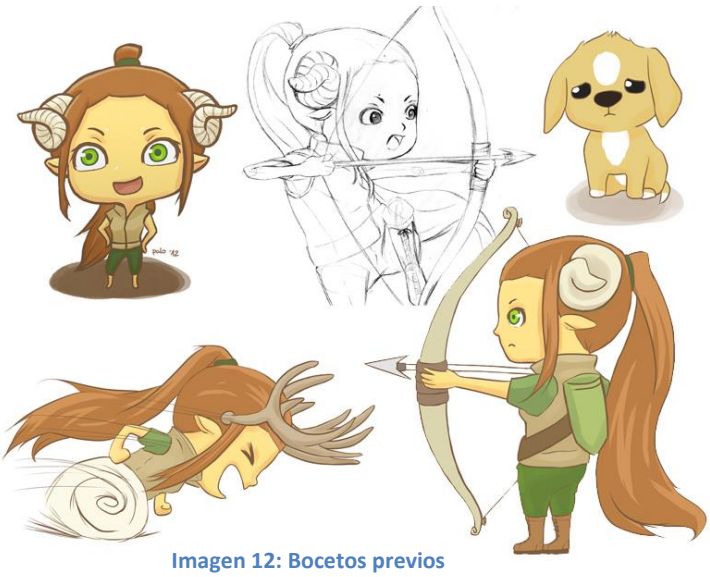


Imagen 12: Bocetos previos

Su rol principal es el de arquera. Se trata, pues, de un personaje de ataque a distancia que intentará alejarse del resto de jugadores para dañarles desde lejos y evitar que la dañen. Cuanta además con un par de ataques que le permiten dañar y alejar a los enemigos que se acerquen demasiado.

- Trampa para osos: Artemisa se agacha y entierra un cepo en el suelo que causará daños a un enemigo cuando pase sobre él. El cepo permanecerá semi-oculto en el suelo, de forma que el resto de jugadores puedan detectarlo y sepan que hay un cepo colocado pero sea muy fácil pasarlo por alto y olvidarse de él. No ocurrirá nada si Artemisa lo pisa. No pueden colocarse dos cepos en el mismo sitio.
- Tiro con arco: Disparo básico con el arco que requiere poca energía y causa pocos daños. Una vez Artemisa lo active, entrará en el modo disparo. En este modo Artemisa no podrá moverse y con el joystick controlará la dirección del disparo. El jugador deberá mantener el botón pulsado mientras Artemisa tensa el arco, y se carga una barra de potencia que aparecerá sobre ella. El daño causado por el disparo dependerá de lo tenso que este el arco, así como la trayectoria de la flecha. Si el jugador no suelta el botón un tiempo después de haberse cargado la barra al máximo. Artemisa se cansará y el disparo se cancela. No se gastará energía si el disparo se ha cancelado. Además el jugador también podrá cancelar el disparo.

Afrodita, Diosa del amor, apasionada de la moda

Diosa del amor, la lujuria, la belleza, la sexualidad y la reproducción. Afrodita puede influir en el corazón de los dioses para controlar sus sentimientos en su beneficio. Afrodita dispone de un par de ataques para enamorar a sus enemigos y causarles curiosos efectos que durarán un tiempo. Además, Afrodita puede utilizar el amor no correspondido para causar daños a los enemigos previamente enamorados y puede utilizar el poder del amor para curarse.

Imagen 14: Diseño final de Afrodita

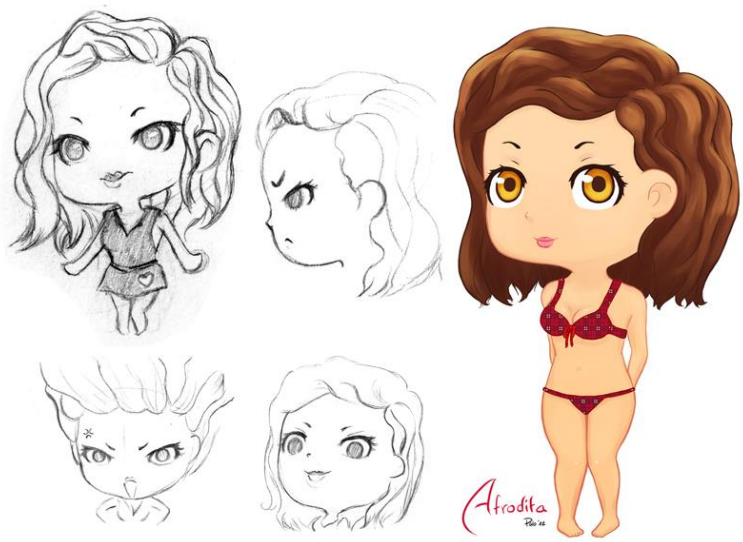


Imagen 13: Bocetos previos

- Amor loco: Es un ataque lineal que puede impactar a un solo enemigo. El enemigo afectado pasará a estar enamorado durante algún tiempo y le será muy difícil pensar en otra cosa que no sea en su amado/a. Mientras dure el efecto el jugador afectado verá sus controles invertidos: Izquierda-derecha y arriba-abajo.
- Rompecorazones: Un ataque lineal que impacta en un solo objetivo. Solo puede afectar a enemigos que estén bajo el efecto de amor loco o amor añorado. El personaje se da cuenta de que su amor el imposible y que nunca será correspondido, sufriendo grandes daños. Termina el estado enamorado.
- El poder del amor: Afrodita utiliza el poder del amor para curarse una pequeña cantidad de vida.

HUD

La interfaz mostrará para cada jugador una imagen del personaje, un icono con cada uno de los ataques seleccionados y asignados a los botones, y su barra de vida. Estos iconos se colocarán en la parte superior de la pantalla.

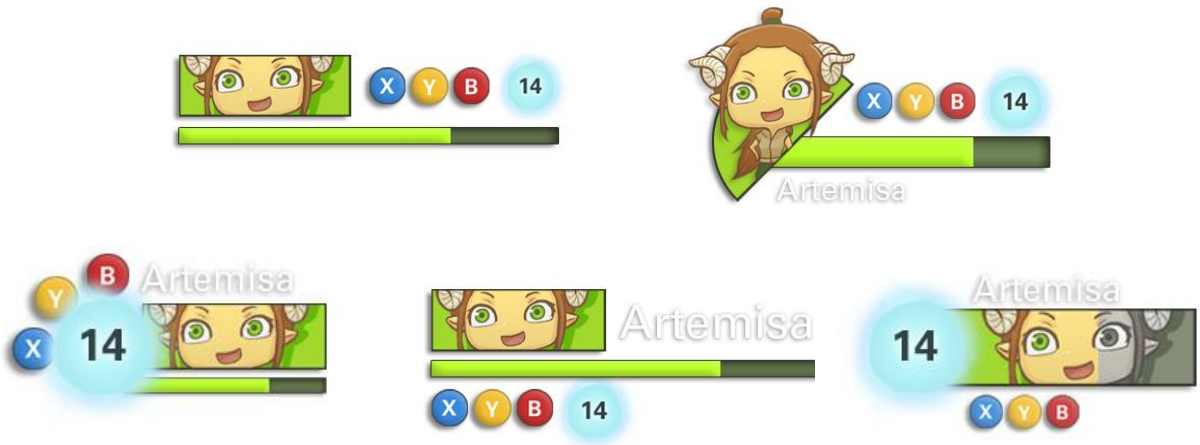


Imagen 15: Diferentes diseños de HUD de un jugador

Escenarios

Se diseñó de forma que hubiera un escenario por cada personaje, de forma que sea algo característico de él, bien de su rol clásico, o de su rol moderno. Esto forma un total de 8 escenarios, de los que solo se pudieron implementar dos. Para conocer el resto de escenarios diseñados consultar el apéndice de diseño.

Bosque (Artemisa)

Un bosque de árboles muy grandes con grandes troncos. Los jugadores podrán ir saltando entre las diferentes ramas de los árboles y en los puentes de madera que habrá entre algunos árboles. De vez en cuando los árboles dejarán caer fruta con la que los jugadores podrán recuperar una pequeña parte de la vida.

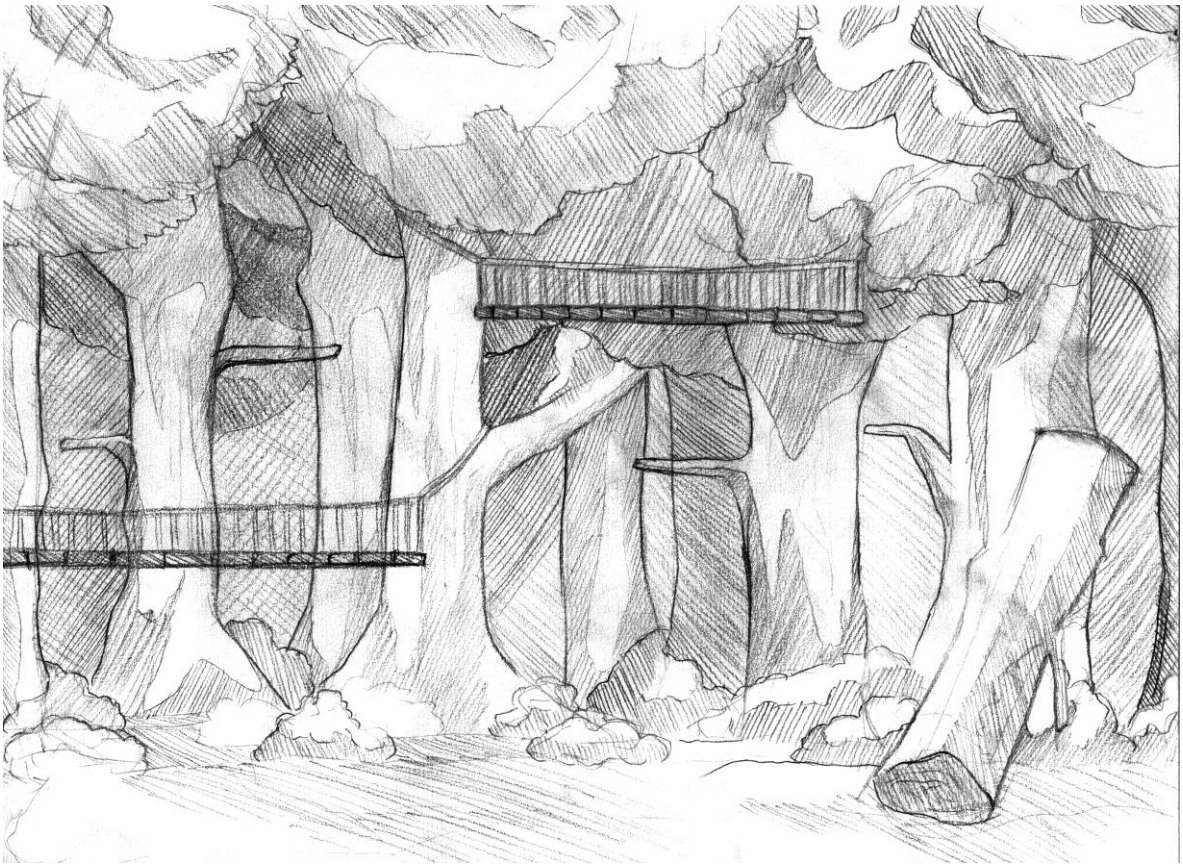


Imagen 16: Primeros bocetos del bosque

Tártaro (Hades)

Capa más profunda del inframundo, una gran prisión fortificada rodeada por un río de fuego llamado Flegetonte. Es el lugar donde van las almas de los condenados y sufren un castigo se adecuado a su crimen. Por ejemplo Sísifo, que era un ladrón y un asesino, fue condenado a empujar eternamente una roca cuesta arriba sólo para verla caer por su propio peso. Tántalo, que disfrutaba de la confianza de los dioses conversando y cenando con ellos, compartió la comida y los secretos de los dioses con sus amigos. Su justo castigo fue ser sumergido hasta el cuello en agua fría, que desaparecía cada vez que intentaba saciar su sed, con suculentas uvas sobre él que subían fuera de su alcance cuando intentaba agarrarlas.

El escenario cuenta con unos desagües de lava situados en la parte superior, que de forma aleatoria y alternada tras un pequeño aviso verterán lava sobre el escenario. Los jugadores tendrán que estar pendientes de cuando sucede esto para esquivarla, ya que si les da les causará daño.

Imagen 17: Diseño avanzado del Tartaro

Justificación del producto

Es un juego pensado para poder jugar con 4 jugadores en cualquier momento, aunque se disponga de poco tiempo. El hecho de poder divertirse en compañía, junto con la estética simpática y desenfadada del juego y sus toques de humor, lo hacen un juego agradable y adecuado para las reuniones en casa con los amigos, en un amplio rango de edades.

Este tipo de juegos de partidas cortas para muchos jugadores si bien son muy comunes en Wii, el multijugador de Xbox está más enfocado a los juegos de larga duración y con partidas online. Además los juegos multijugador de la plataforma que se juegan desde una misma consola suelen ser para dos jugadores, por lo que pensamos que estamos enfocando el juego en una dirección muy poco explotada en la plataforma. Por otra parte, el juego presenta una jugabilidad, relativamente similar a Super Smash Bross y Playstation all stars, juegos de gran éxito en Nintendo y Playstation respectivamente, pero sin nada similar en Xbox, por lo que podemos aprovecharnos de esta cadencia en la plataforma.

Planificación

“There is nothing so useless as doing efficiently that which should not be done at all.”

— Peter F. Drucker

Introducción

Para el desarrollo del presente proyecto decidí invertir un año completo, de junio de 2012 a junio de 2013. En un proyecto de tal duración y en el que participan más de una persona es de suma importancia tener todo el proyecto planeado desde el inicio con una estimación inicial de los costes de cada una de las tareas. Esta planificación inicial, si bien probablemente no se seguirá estrictamente dada la dificultad de estimar el coste de las tareas previamente a su realización y que el desarrollo de un videojuego es un proceso orgánico en el que se añaden, cambian y eliminar características; permitirá realizar un seguimiento para medir la desviación con respecto al plan y reaccionar adecuadamente a los imprevistos.

En primer lugar, se realizó el diseño del videojuego completo, antes de realizar ninguna planificación. Dicha tarea, me llevo aproximadamente 3 meses. Al tiempo restante, si le restamos el tiempo estimado de migración al paradigma discreto desacoplado, unos dos meses, obtenemos el tiempo disponible para la creación del videojuego.

Ya sabiendo el juego que se iba a desarrollar, sus características y el tiempo disponible para su desarrollo, se procedió a realizar una estimación del coste de producción de cada una de las características del videojuego y a una clasificación según su prioridad. Puesto que desde el primer momento quedo patente que el diseño del videojuego era demasiado extenso y complejo para el tiempo disponible, hubo que realizar una selección de características. Esta selección de características se reduce fundamentalmente en la elección de los personajes a desarrollar entre todos los propuestos por el diseño. Cuando la magnitud de las características seleccionadas ya era acorde al tiempo disponible, se procedió a trazar un plan de desarrollo.

Planificación

A la hora de realizar la planificación, cuando hay que realizar un plan de trabajo que involucra a más de una persona, es de gran importancia realizarlo de forma que el trabajo entre los participantes esté sincronizado, que no se generen cuellos de botella por dependencias entre el trabajo de varias personas y que nadie se quede sin cosas que hacer. En este caso además, mi compañera tenía unas necesidades de plazo y objetivos diferentes a los míos, por lo que al realizar la planificación he tenido la dificultad extra de que esta cumpla con las necesidades de todos.

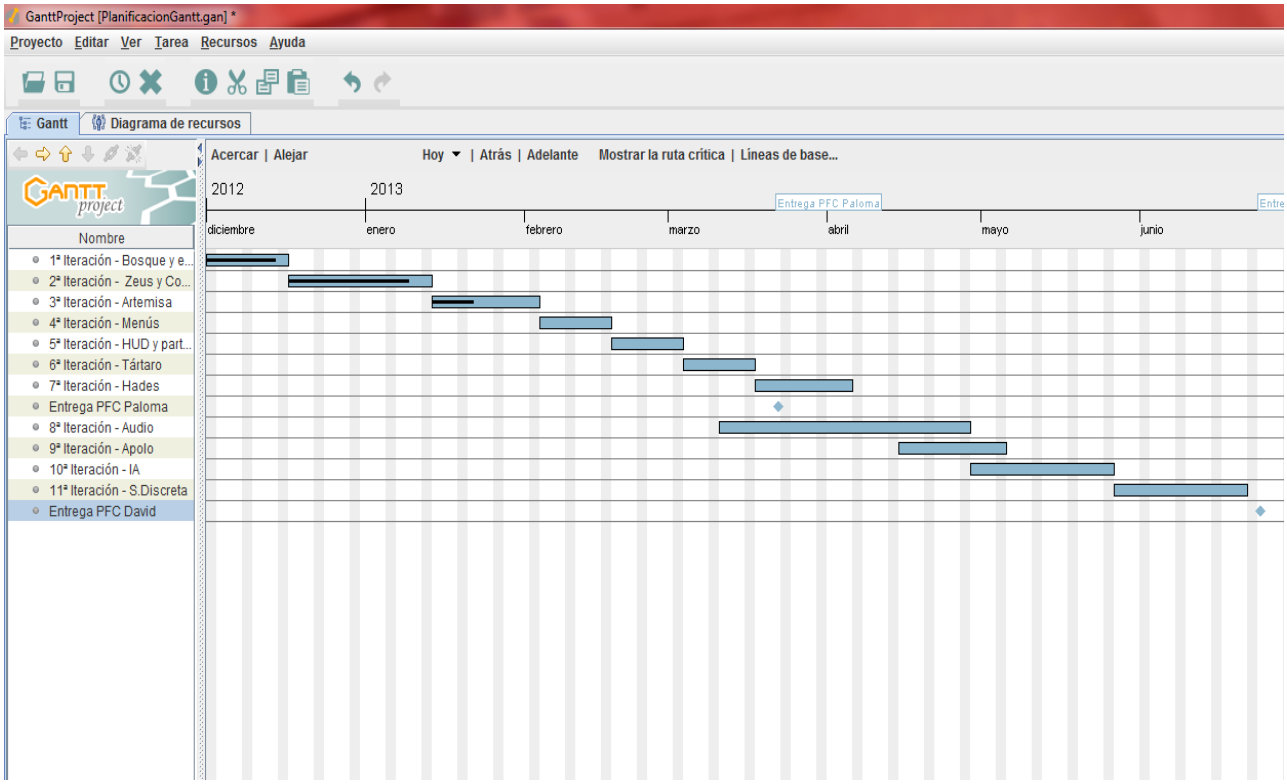


Imagen 18: Diagrama de Gantt de la planificación realizado con la herramienta GanttProject

1ª Iteración: Bosque y Estructura básica

Tiempo estimado: 3 Semanas.

Desarrollar el control básico del jugador sobre los personajes (movimiento, salto y recoger objetos) y su interacción con un escenario estático. Para ilustrar estas características se desarrollará el control de Zeus con todas sus animaciones de movimiento y salto definitivas y perfectamente sincronizadas con las acciones del jugador en el escenario del Bosque con todas sus características. Además, se implementará el soporte necesario para mostrar los FPS y que el juego se visualice correctamente en todas las resoluciones, adaptándose la zona de juego para ocupar el mayor espacio posible de la pantalla, añadiéndose barras verticales o horizontales si la razón de aspecto del juego y la pantalla son diferentes.

David	Paloma
• Interacción con el jugador a través del mando (movimiento, salto y recoger objetos). • Física para los saltos. • Soporte para animaciones • Sincronización de las animaciones de movimiento de Zeus. • Control de las colisiones con los elementos del escenario (suelo, plataformas, etc.). • Sincronización de las animaciones del fondo. • Soporte para mostrar FPS. • Independencia de la resolución	• Animación “idle” de Zeus. • Animación de correr. • ¿Animación de andar? • Animación de salto. • Dibujo del escenario Bosque. • Añadir animaciones al escenario.

2ª Iteración: Zeus y combate

Tiempo estimado: 4 Semanas.

Soporte para la interacción multijugador gestionando más de un mando de forma simultánea. Implementación de las características básicas de interacción entre jugadores: Lanzamiento de los ataques, impactando y dañando a los jugadores y colisionando con el escenario, uso del escudo para protegerse de los ataques y salto de un jugador sobre otro. Para ejemplificarlo se desarrollarán todos los ataques de Zeus y se creará una partida en la que podrán interaccionar de 1 a 4 jugadores todos controlando a Zeus.

David	Paloma
• Soporte para multijugador. • Creación del concepto de "ataque". • Interacción con los jugadores para permitirles lanzar ataques y usar el escudo. • Sincronización de las animaciones de los diferentes ataques. • Detección y gestión de los impactos (colisiones) de los ataques de Zeus con el escenario y con otros jugadores. • Implementación del escudo y su estado de defensa asociado. • Detección y gestión del salto de un jugador sobre otro. • Distinción entre el estado cargado y el estado normal. • Sincronización de las animaciones de movimiento de Zeus en estado cargado. • Sincronización del efecto de electrocución en los objetivos impactados por los ataques de Zeus.	• Animación de ataque "Rayo" de Zeus, tanto del propio Zeus como del rayo. • Animación de Zeus realizando el ataque carga eléctrica. • Animación Idle de Zeus en estado cargado. • Animación de correr de Zeus en estado cargado. • ¿Animación de andar de Zeus en estado cargado? • Animación de salto de Zeus en estado cargado. • Animación del ataque Electro-explosión, tanto del propio Zeus como del ataque. • Animación de Zeus utilizando el escudo. • Animación de electrocución para los objetivos impactados por los ataques de Zeus.

3ª Iteración: Artemisa

Tiempo estimado: 3 Semanas.

Desarrollar y añadir al juego un nuevo personaje, Artemisa; con todas sus características: Todas sus animaciones de movimiento, salto, lanzamiento de escudo perfectamente desarrolladas y sincronizadas con las acciones del jugador. Todos sus ataques y sus efectos desarrollados.

David	Paloma
• Sincronización de las animaciones de movimiento, salto y escudo de Artemisa con las acciones del jugador. • Sincronización de las animaciones de los diferentes ataques de Artemisa. • Gestión de los ataques de Artemisa y sus efectos. • Interacción con el jugador para controlar la fuerza y dirección de los disparos de Artemisa. • Física para la trayectoria de las flechas. • Objetos trampa de osos (ataque de Artemisa), y flecha perdida. • Implementación del efecto gráfico de sufrir daño.	• Animación “idle” de Artemisa. • Animación de correr de Artemisa. • ¿Animación de andar? • Animación de salto de Artemisa. • Animación del ataque cabezazo. • Animación de Artemisa utilizando el escudo. • Sprite de Artemisa en posición de tiro con arco de forma que se pueda cambiar la posición de la flecha con código. • Sprite de la flecha. • Animación del ataque trampa para osos (Artemisa agachándose) • Sprite de la trampa para osos. • Sprites de los personajes sufriendo daño.

4ª Iteración: Menús

Tiempo estimado: 2 Semanas.

Implementar el soporte de menús al juego y añadir la jerarquía de menús necesaria para dar soporte a una partida multijugador: pantalla de título, menú principal, configurar partida, escoger personajes, escoger escenario y fin de partida. Los menús estarán completos con todas sus opciones, si bien las opciones que aún no estén implementadas se desactivarán.

David	Paloma
• Programación de la lógica de los diferentes menús. • Implementación del esquema de navegación entre las opciones del menú y entre los diferentes menús. • Sincronización de las animaciones de transición entre los diferentes estados de las opciones de los menús con las acciones del jugador. • Elemento básico de los menús: “opción de menú” con tres estados: normal, seleccionado y desactivado. • Elementos más complejos para permitir a 4 jugadores seleccionar personaje simultáneamente.	• Diseño del esquema de los diferentes menús. • Diseño de la apariencia de las opciones del menú. • Animación de las transiciones entre estados de las opciones del menú. • Creación de los sprites / animaciones presentes en los menús.

5ª Iteración: HUD y concepto de partida

Tiempo estimado: 2 Semanas.

Desarrollar el HUD de las partidas de forma que refleje el estado actual de los jugadores: jugador seleccionado, vida, nº de vidas, plegarias acumuladas, hechizos disponibles, hechizos en proceso de recarga y tiempo restante de partida.

Implementar el concepto de partida, teniendo está una configuración, un principio, un final y unos resultados. El tipo de partida y el HUD dependerán de la configuración de partida seleccionada. Para ello, además desarrollar todas aquellas características restantes que dan soporte a la partida: Aparición y recogida de las plegarias, proceso de carga de los hechizos.

David	Paloma
• Desarrollar el concepto de partida. • Diferentes tipos de partida según la configuración seleccionada. • Aparición de las plegarias de forma pseudoaleatoria por el escenario. • Implementación del coste de los hechizos y gasto de plegarias. • Programación del HUD de cada jugador para que refleje el estado interno de cada jugador. • Diferente disposición de los elementos del HUD según el número de jugadores y el tipo de partida seleccionado, teniendo en cuenta la “Title Safe Zone”. • Detección de los eventos de inicio y final de partida.	• Creación del HUD individual de cada jugador para reflejar toda la información necesaria. • Diseño de los iconos de cada uno de los hechizos. • Diseño, creación y animación de las plegarias. • Creación de los efectos de inicio de partida y fin de partida. (3,2,1… fight, End).

6ª Iteración: Tártaro

Tiempo estimado: 2 Semanas.

Desarrollar y añadir al juego un nuevo escenario, el Tártaro, con todas sus características desarrolladas: animaciones y lógica.

David	Paloma
• Control de las colisiones de los personajes con los elementos del escenario: plataformas, suelo, paredes, etc. • Programación de la lógica del escenario que hace caer de forma aleatoria lava de los desagües. • La lava daña a los jugadores. • Sincronización de las animaciones del escenario. • Desarrollo de un efecto de partículas empleando chispas que genere la sensación de un ambiente caliente y agobiante.	• Creación del escenario el Tártaro. • Creación de las animaciones y efectos del escenario. • Creación y animación del sprite “caída de lava”.

7ª Iteración: Hades

Tiempo estimado: 3 Semanas.

Desarrollar y añadir al juego un nuevo personaje, Hades, con todas sus características: Todas sus animaciones de movimiento, salto, lanzamiento de escudo perfectamente desarrolladas y sincronizadas con las acciones del jugador. Todos sus ataques y sus efectos desarrollados.

David	Paloma
• Sincronización de las animaciones de movimiento, salto y escudo de Hades con las acciones del jugador. • Sincronización de las animaciones de los diferentes ataques de Hades. • Gestión de los ataques de Hades y sus efectos. • Implementación de una nueva característica en Hades, las almas. • Programación de Hades para que sus ataques cuesten almas en vez de plegarias. • Gestión del control de Hades para que el jugador controle a Cerbero en vez de a Hades si este está activado.	• Animación “idle” de Hades. • Animación de correr de Hades. • Animación de salto de Hades. • Sprite de Hades sufriendo daño. • Animación del ataque Cosecha de alma. • Animación de Hades utilizando Ataque de Cerbero. • Diseño de Cerbero. • Animación de Cerbero corriendo. • Animación de Cerbero saltando. • Animación de Cerbero mordiendo.

8ª Iteración: Audio

Tiempo estimado: Se desarrollará poco a poco simultáneamente a las últimas iteraciones.

Desarrollar todos los archivos de audio del juego: efectos del escenario, sonidos de los ataques, música de fondo, etc. y programar su reproducción acorde con el estado del juego. Para desarrollar los archivos de audio se recurrirá tanto a bancos de sonidos libres que podrán modificarse para adaptarlos a las necesidades del proyecto como a sonidos creados por nosotros.

- Efectos para los ataques.
- Efectos de los escenarios.
- Una canción de fondo para cada uno de los escenarios desarrollados.
- Efectos de sonido para los combates: sufrir daño, aparición de plegarias, etc.
- Una canción para los menús.
- Efectos de sonido para los menús.

9ª Iteración: Apolo

Tiempo estimado: 3 Semanas.

Desarrollar y añadir al juego un nuevo personaje, Apolo; con todas sus características: Todas sus animaciones de movimiento, salto, lanzamiento de escudo perfectamente desarrolladas y sincronizadas con las acciones del jugador. Todos sus ataques y sus efectos desarrollados.

David	Paloma
• Sincronización de las animaciones de movimiento, salto y escudo de Apolo con las acciones del jugador. • Sincronización de las animaciones de los diferentes ataques de Apolo. • Gestión de los ataques de Apolo y sus efectos. • Gestión de los diferentes estados de emisión de color de Apolo, cambiando su color mediante programación y aplicando los efectos asociados. • Interacción con el jugador para controlar la dirección del ataque láser rojo. • Interacción con el jugador para controlar la posición y explosión del ataque bomba de luz verde. • Sincronización de las animaciones de los ataques de Apolo con efecto multicolor.	• Animación “idle” de Apolo. • Animación de correr de Apolo. • ¿Animación de andar? • Animación de salto de Apolo. • Animación de Apolo utilizando el escudo. • Sprite de Apolo sufriendo daño. • Sprite de Apolo lanzando el ataque láser rojo, de forma que la dirección del rayo sea independiente de la animación. • Animación del láser rojo, tanto en forma normal como en forma multicolor. • Animación del ataque descomposición en luz azul, tanto en forma normal como en forma multicolor. • Animación de Apolo lanzando el ataque bomba de luz verde, normal/multicolor. • Animación de la bomba en idle normal/multicolor. • Animación de la explosión de la bomba normal/multicolor.

10ª Iteración: IA

Tiempo estimado: 4 Semanas.

Diseñar y desarrollar la IA del juego de forma que sea posible añadir personajes controlados tanto por el jugador como por la IA. Los personajes de IA podrán tener 3 dificultades diferentes, y tienen que comportarse lo más parecido posible a como lo haría un jugador.

David
• Diseño de la arquitectura de la IA. • Programación de la IA básica, común para todos los personajes y sus 3 dificultades. • Programación de la IA especifica de Zeus, que determine sus estrategias. • Programación de la IA especifica de Artemisa, que determine sus estrategias. • Programación de la IA especifica de Hades, que determine sus estrategias. • Programación de la IA especifica de Apolo, que determine sus estrategias. • Actualizar el menú para poder añadir personajes controlados por la IA.

Coordinación

Una vez realizado un plan de trabajo en el que todos los miembros están conformes, puede comenzarse con el desarrollo. Las tareas de planificación y coordinación no terminan una vez ha empezado el desarrollo del videojuego, si no que se extienden a lo largo de todo el proceso. Para mantener al equipo al tanto del estado del desarrollo y saber detectar las variaciones con respecto al plan y poder adaptarse con facilidad se siguieron las siguientes normas:

- Mantener actualizado el plan de trabajo y corregir y adaptar la planificación continuamente según el transcurso del desarrollo.
- Una reunión presencial al inicio de cada iteración para discutir la forma en la que se desarrollarían los objetivos de dicha iteración, y realizar una planificación más profunda del transcurso de la iteración.
- Una reunión presencial al final de cada iteración para discutir los objetivos cumplidos y no cumplidos, realizar una reestimación del coste de las tareas restantes y realizar una posible reeplanifinación del desarrollo.
- Comunicación diaria a través de videoconferencia o chat para informar del trabajo realizado así como de posibles problemas encontrados y realizar una coordinación de los contenidos desarrollados en el producto final.

Cambios en la Planificación

Tal y como era de esperar, a lo largo del proceso de desarrollo hubo que ir adaptando la planificación inicial para adaptarnos a los imprevistos surgidos, a nuestro mejor conocimiento sobre las tareas y a los cambios realizados sobre el juego. Estos son los cambios más importantes:

- Inicialmente estaba previsto implementar 3 ataques para cada uno de los personajes, pero un retraso acumulado producido por una mala estimación a la baja de los costes de desarrollo nos llevó a tomar la decisión de implementar solo 2 ataques por personaje.
- Dado que el personaje de Apolo era muy complejo de desarrollar y no nos sobraba el tiempo, se decidió implementar en cuarto lugar a Afrodita en vez de a Apolo.
- Finalmente la parte de IA tuvo que dejarse de lado para poder cumplir con el resto de apartados del proyecto.

Diseño UML

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.

— C.A.R. Hoare, The 1980 ACM Turing Award Lecture

Introducción

El lenguaje empleado para el desarrollo en C#, un lenguaje orientado a objetos, por lo que de forma natural para realizar el modelado del sistema se ha empleado UML, el lenguaje de modelado de sistemas software más utilizado en la actualidad. Esté nos permitirá no solo diseñar el sistema, sino además documentarlo.

La arquitectura diseñada está construida sobre la propia arquitectura base de XNA que proprciona un bucle principal y una serie de servicios a través de unas clases predefinidas:

- ContentManager: Proporciona servicios para cargar imágenes y sonidos en XNA y poder utilizarlos desde el código.
- Texture2D: Representa una imagen cargada por XNA y lista para dibujarse en pantalla.
- Sound: Permite albergar y reproducir tanto efectos de sonido simples como disparos o explosiones como canciones completas de banda sonora. Entre otras características permite controlar el volumen, iniciar y detener la reproducción y reproducir en bucle.
- GamePad: Permite acceso a los mandos conectados al PC o la Xbox.

En el centro de la arquitectura de XNA se encuentra la clase Game, una clase virtual que proporciona un bucle principal oculto al desarrollador, comunicación con la tarjeta gráfica y código de renderizado. En todo juego escrito en XNA debe haber una clase, la clase principal, que herede de esta clase. El desarrollador debe de sobrescribir los métodos Update y Draw proporcionados por está, y el bucle interno de XNA se encargara de llamarlos según crea conveniente.

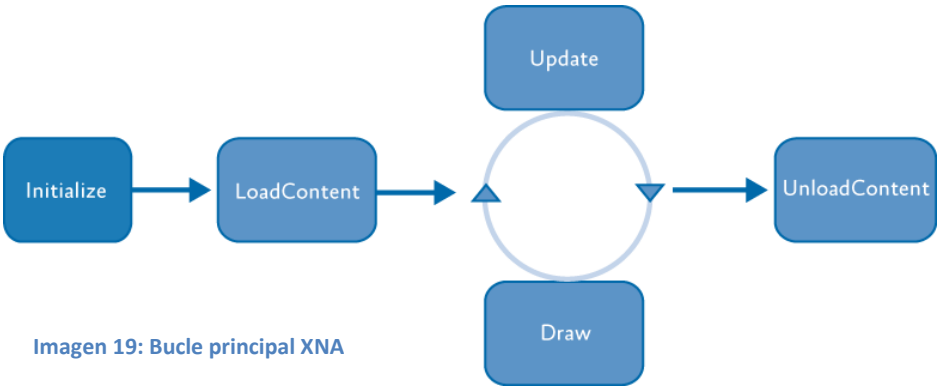


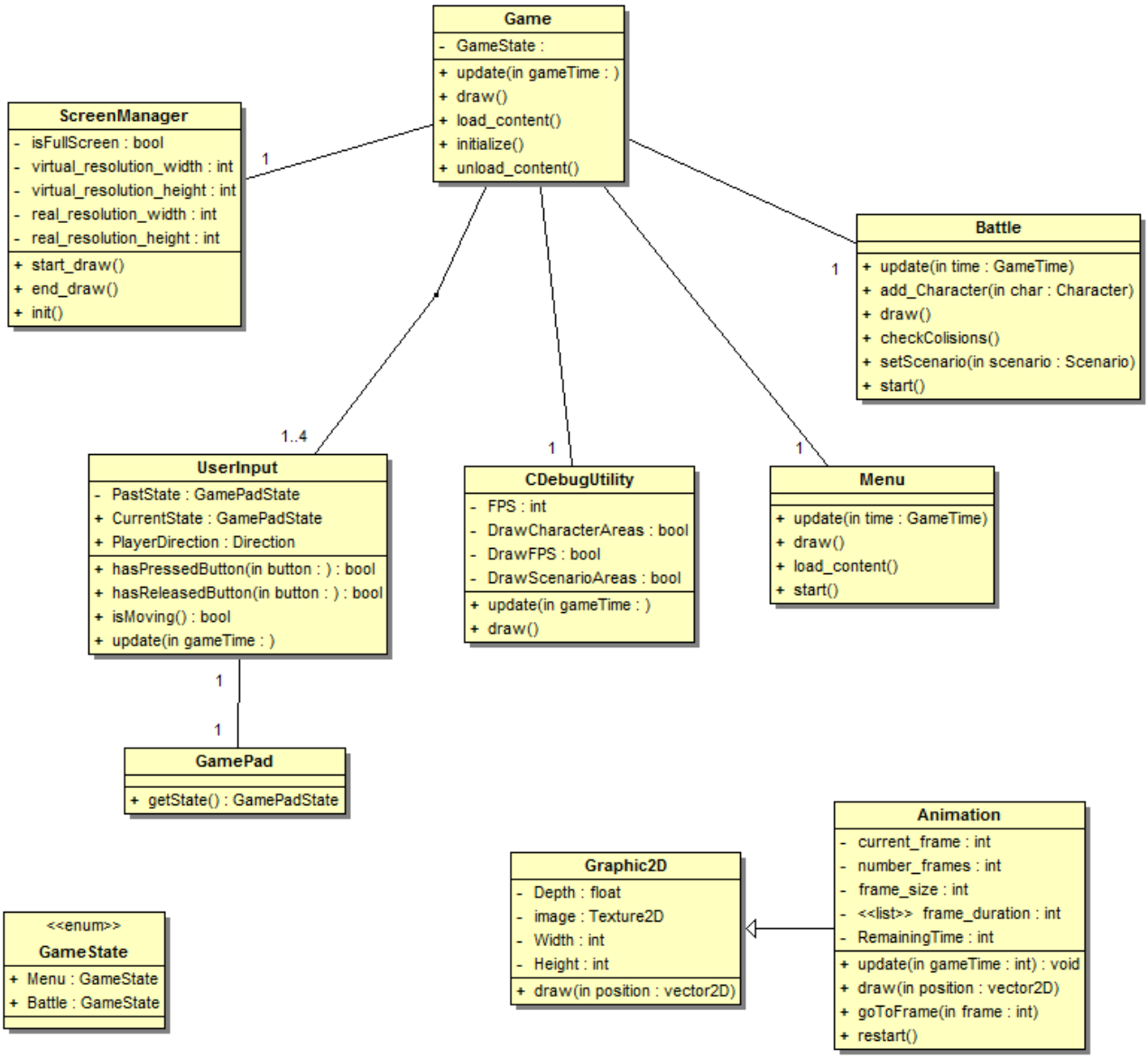
Imagen 19: Bucle principal XNA

En el juego se distinguen dos estados principales, el menú, en el que el jugador o jugadores seleccionan la configuración de la partida, los personajes y el escenario en el que van a luchar y la parte del juego en sí, en la que los jugadores se enfrentan. Como estas fases están bastante diferenciadas y comparten pocos elementos entre si la arquitectura del juego se ha realizado distinguiendo estos dos estados. Además de los diagramas de clases, que son los más comunes y más utilizados en el diseño del software, he realizado diagramas de secuencia, que ayudan a tener una visión más clara del funcionamiento del sistema.

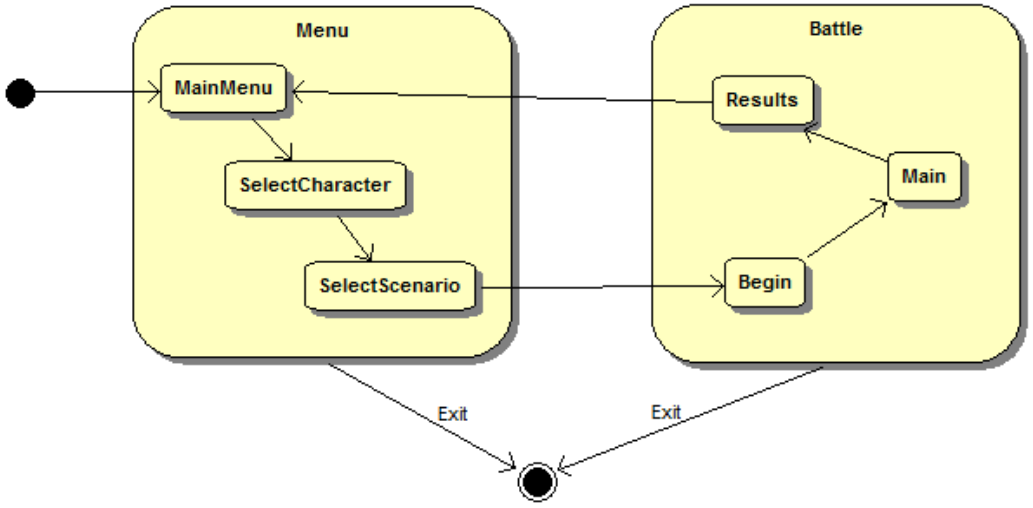
Arquitectura general

La arquitectura general del juego nos muestra los componentes principales que forman el videojuego.

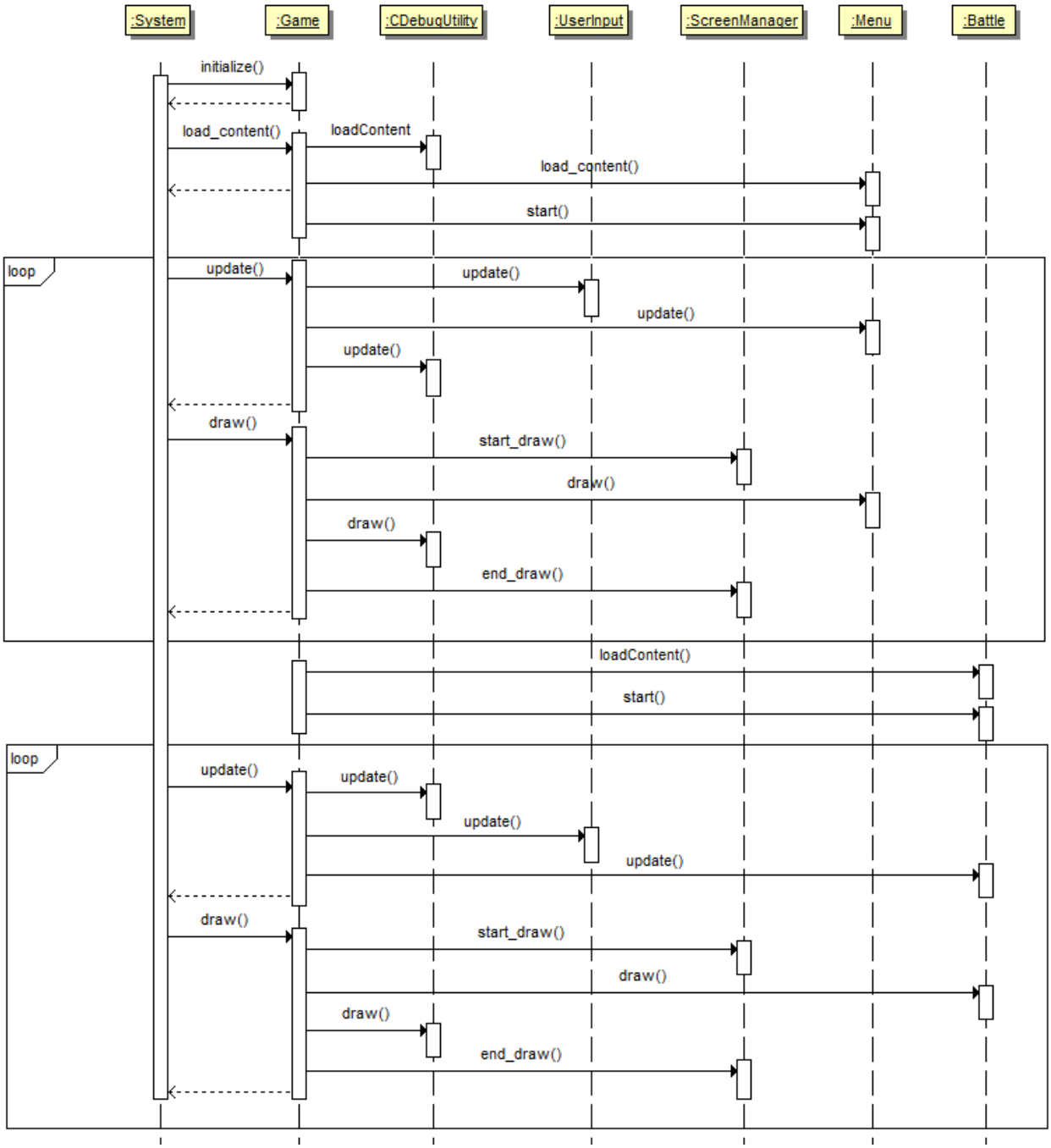
- En el centro de la arquitectura esta la clase **Game** que hereda de la clase Game proporcionada por XNA que he comentado anteriormente.
- La clase ScreenManager se encarga de la comunicación con la tarjeta gráfica y de que el juego se visualice correctamente en cualquier resolución.
- La clase UserInput gestiona la entrada del usuario.
- Las clases Graphic2D y Animation representan contenidos gráficos representables en pantalla y cualquier clase que dibuje algo en pantalla contendrá un elemento de este tipo. Dado la cantidad de clases que hacer uso de este recurso, por claridad, se ha optado por definirlo aparte.



En el siguiente diagrama de estados, podemos ver los dos principales estados del juego, Menu y Batalla, sus subestados, y las transiciones entre estados.

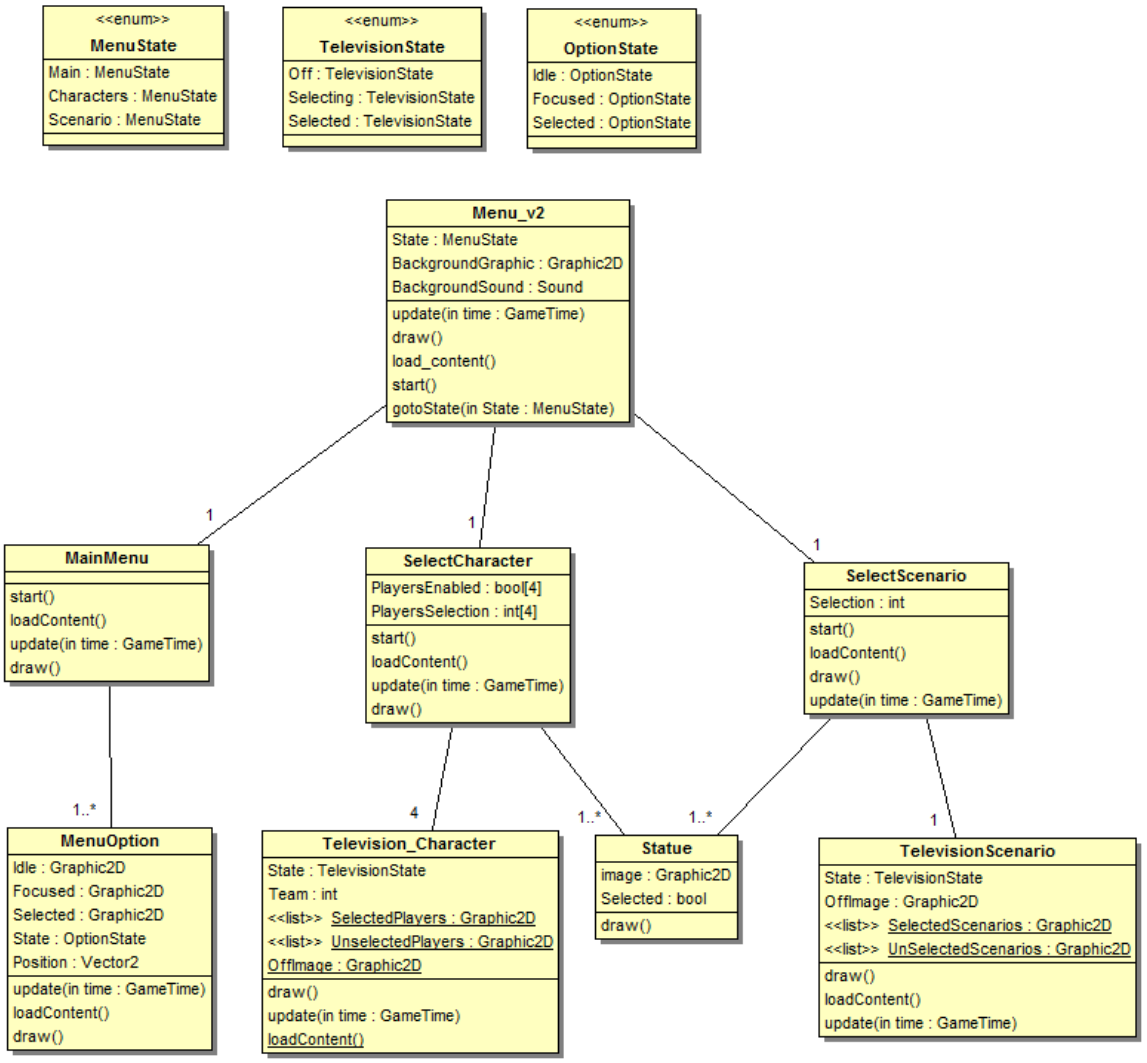


En el diagrama de secuencia, en primer lugar podemos distinguir como desde la clase Game se realiza una propagación de las llamadas de los métodos loadContent(), update() y draw a Menu y Battle, que a su vez seguirán propagando la llamada a sus clases miembro hasta abarcar todas las clases de la estructura. En segundo lugar, podemos distinguir dos bucles, que realizan continuamente y de forma secuencial los métodos update y draw. La ejecución de un bucle u otro dependerá del estado en el que se encuentre el juego.

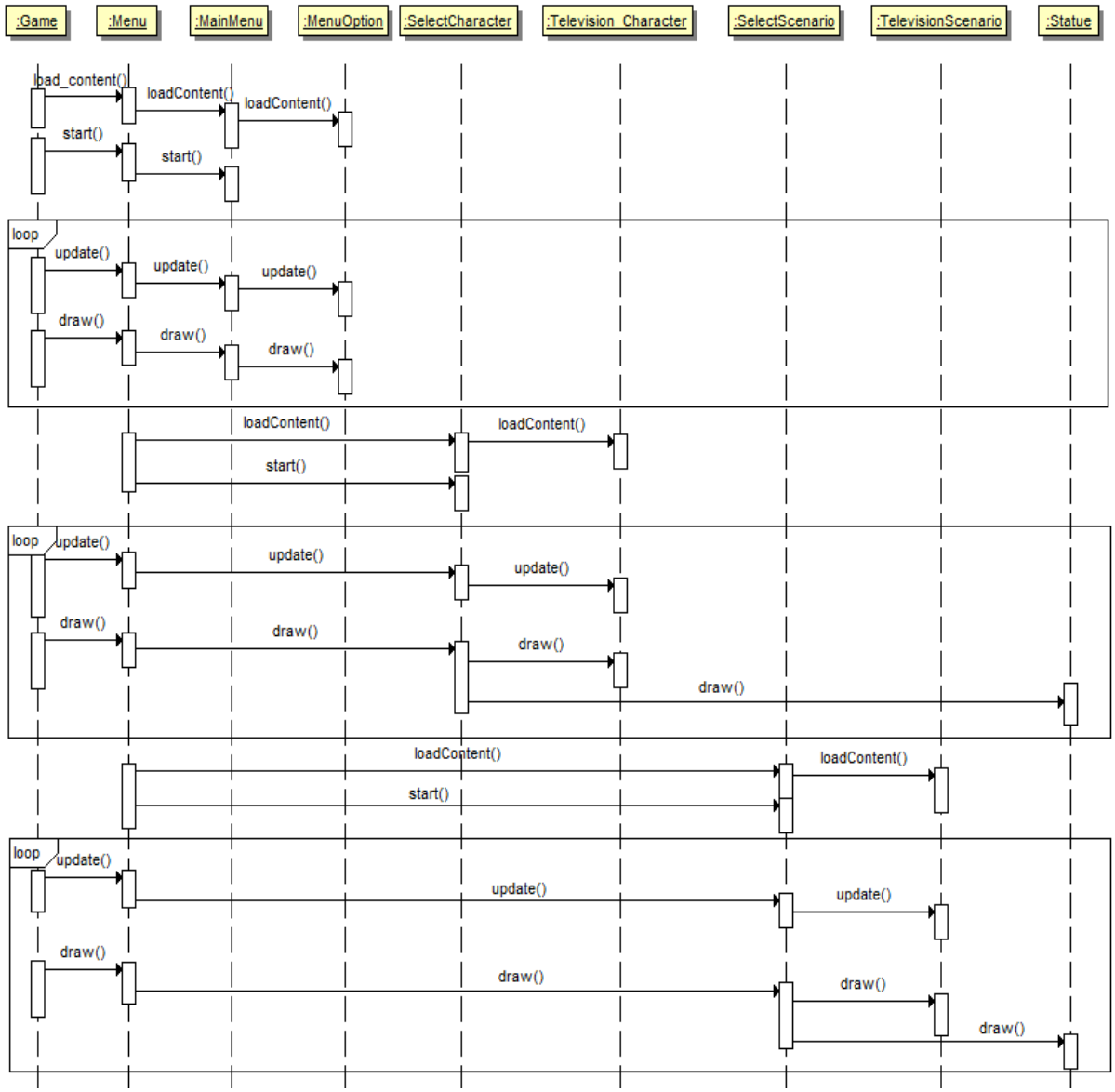


Arquitectura de menú

La arquitectura del menú nos muestra 3 clases que interaccionan directamente con la clase Menu, MainMenu, SelectCharacter y SelectScenario. Cada una de ellas se corresponde con cada uno de los posibles estados en los que se puede encontrar el menú y se corresponde al mismo tiempo con un submenú.



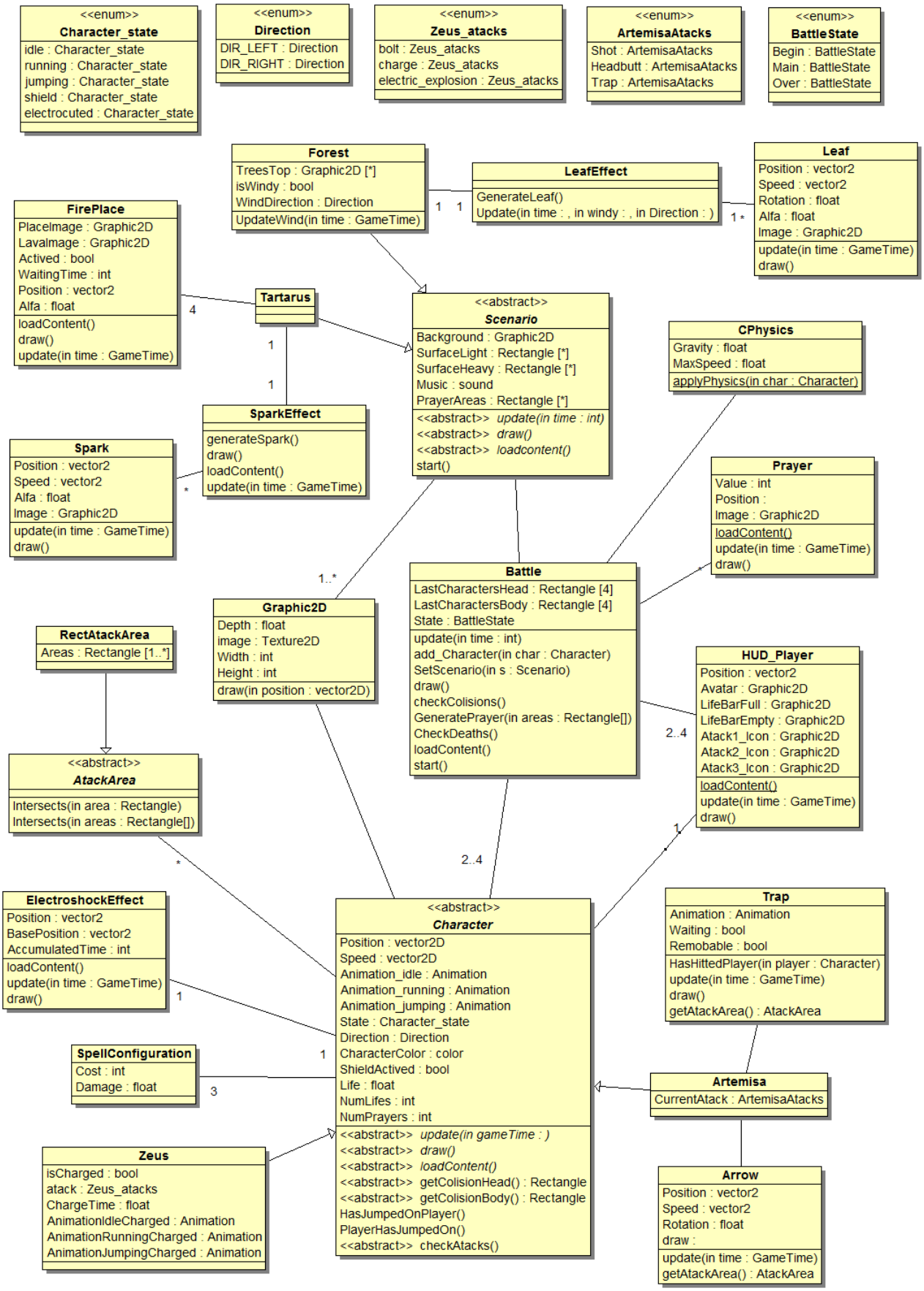
De forma similar al diagrama de secuencia general, en el diagrama de secuencia del menú, podemos observar como desde la clase Menú se continua con la propagación de las llamadas de los métodos loadContent(), update() y draw(). En este caso, podemos observar tres bucles de llamadas a update() y draw() que se corresponden con cada uno de los estados del menú.



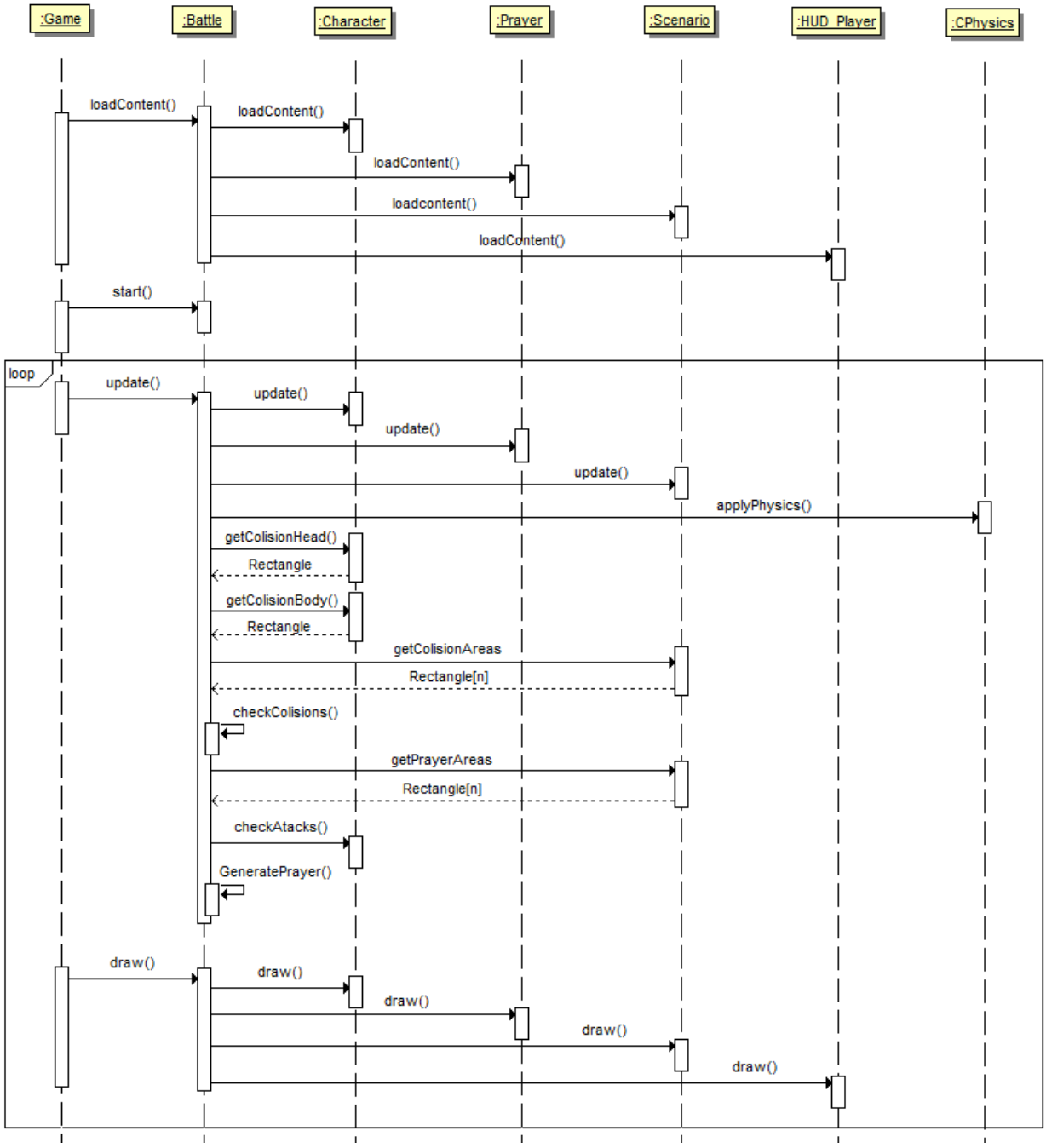
Arquitectura de juego

La arquitectura de la Batalla en la más compleja y extendida, puesto que forma el núcleo del juego. El núcleo de funcionamiento lo formas las clases:

- La clase central, **Battle**, se encarga de generar las plegarias según la configuración de la partida, de realizar la detección de colisiones Personaje-Personaje y Personaje-Escenario y su correcta gestión, de controlar el inicio y fin de partida, de controlar toda la configuración de la partida y de propagar adecuadamente las llamadas.
- La clase **Physics** es una clase estática que se encarga de la física del juego.
- La clase abstracta **Character**, de la que heredan los diferentes personajes como Artemisa y Zeus, proporciona la estructura básica para permitir a sus clases descendientes realizar el control del personaje, respondiendo a las acciones del usuario y gestionando y actualizando todas sus animaciones. Se encarga además, de gestionar los ataques del personaje, detectando la colisión de los ataques con otros personajes y con el escenario y actualizando su estado. El código proporcionado por **Character** es mínimo, y el grueso de la implementación se realizará en sus clases descendientes puesto que cada personaje es muy diferente del resto.
- La clase **HUD_Player** se encarga de mostrar el estado de un personaje que tiene asociado en pantalla.


<<enum>>
Character_state
idle : Character_state
running : Character_state
jumping : Character_state
shield : Character_state
electrocuted : Character_state
<<enum>>
Direction
DIR_LEFT : Direction
DIR_RIGHT : Direction
<<enum>>
Zeus_atacks
bolt : Zeus_atacks
charge : Zeus_atacks
electric_explosion : Zeus_atacks
<<enum>>
ArtemisaAtacks
Shot : ArtemisaAtacks
Headbutt : ArtemisaAtacks
Trap : ArtemisaAtacks
<<enum>>
BattleState
Begin : BattleState
Main : BattleState
Over : BattleState
Forest
TreesTop : Graphic2D [*]
isWindy : bool
WindDirection : Direction
UpdateWind(in time : GameTime)
LeafEffect
GenerateLeaf()
Update(in time : , in windy : , in Direction :)
Leaf
Position : vector2
Speed : vector2
Rotation : float
Alfa : float
Image : Graphic2D
update(in time : GameTime)
draw()
FirePlace
PlaceImage : Graphic2D
LavaImage : Graphic2D
Actived : bool
WaitingTime : int
Position : vector2
Alfa : float
loadContent()
draw()
update(in time : GameTime)
Tartarus
<<abstract>>
Scenario
Background : Graphic2D
SurfaceLight : Rectangle [*]
SurfaceHeavy : Rectangle [*]
Music : sound
PrayerAreas : Rectangle [*]
<<abstract>> update(in time : int)
<<abstract>> draw()
<<abstract>> loadcontent()
start()
CPhysics
Gravity : float
MaxSpeed : float
applyPhysics(in char : Character)
SparkEffect
generateSpark()
draw()
loadContent()
update(in time : GameTime)
Spark
Position : vector2
Speed : vector2
Alfa : float
Image : Graphic2D
update(in time : GameTime)
draw()
Prayer
Value : int
Position :
Image : Graphic2D
loadContent()
update(in time : GameTime)
draw()
Graphic2D
Depth : float
image : Texture2D
Width : int
Height : int
draw(in position : vector2D)
Battle
LastCharactersHead : Rectangle [4]
LastCharactersBody : Rectangle [4]
State : BattleState
update(in time : int)
add_Character(in char : Character)
SetScenario(in s : Scenario)
draw()
checkColisions()
GeneratePrayer(in areas : Rectangle[])
CheckDeaths()
loadContent()
start()
RectAtackArea
Areas : Rectangle [1..*]
HUD_Player
Position : vector2
Avatar : Graphic2D
LifeBarFull : Graphic2D
LifeBarEmpty : Graphic2D
Atack1_Icon : Graphic2D
Atack2_Icon : Graphic2D
Atack3_Icon : Graphic2D
loadContent()
update(in time : GameTime)
draw()
<<abstract>>
AtackArea
Intersects(in area : Rectangle)
Intersects(in areas : Rectangle[])
ElectroshockEffect
Position : vector2
BasePosition : vector2
AccumulatedTime : int
loadContent()
update(in time : GameTime)
draw()
<<abstract>>
Character
Position : vector2D
Speed : vector2D
Animation_idle : Animation
Animation_running : Animation
Animation_jumping : Animation
State : Character_state
Direction : Direction
CharacterColor : color
ShieldActived : bool
Life : float
NumLifes : int
NumPrayers : int
<<abstract>> update(in gameTime :)
<<abstract>> draw()
<<abstract>> loadContent()
<<abstract>> getColisionHead() : Rectangle
<<abstract>> getColisionBody() : Rectangle
HasJumpedOnPlayer()
PlayerHasJumpedOn()
<<abstract>> checkAtacks()
Trap
Animation : Animation
Waiting : bool
Remobable : bool
HasHittedPlayer(in player : Character)
update(in time : GameTime)
draw()
getAtackArea() : AtackArea
SpellConfiguration
Cost : int
Damage : float
Artemisa
CurrentAtack : ArtemisaAtacks
Zeus
isCharged : bool
atack : Zeus_atacks
ChargeTime : float
AnimationIdleCharged : Animation
AnimationRunningCharged : Animation
AnimationJumpingCharged : Animation
Arrow
Position : vector2
Speed : vector2
Rotation : float
draw :
update(in time : GameTime)
getAtackArea() : AtackArea

El diagrama de secuencia aclara un poco más el funcionamiento



Implementación

"Measuring programming progress by lines of code is like measuring aircraft building progress by weight."

- Bill Gates

Desarrollo para XNA

Tal y como se ha comentado, XNA es un Framework para el desarrollo de videojuegos que proporciona un amplio conjunto de librerías de clases específicas para el desarrollo de juegos por lo que muchas de las tareas comunes en el desarrollo de videojuegos ya están implementadas.

XNA proporciona un bucle principal integrado a través de la clase Game, por lo que el desarrollado solo debe escribir los métodos update y draw. El desarrollador puede configurar XNA para que ejecute su bucle principal de dos modos diferentes:

- **Tiempo fijo**: El desarrollador le indica a XNA un tiempo, **t**, y este se encarga de que cada **t** instantes de tiempo se llame al método update, ejecutando además, el método draw inmediatamente después del método update. Si XNA no puede conseguir la velocidad deseada, ya sea bien porque el método update o draw están llevando demasiado tiempo o por que el sistema se ha relentizado, por ejemplo; por un ciclo de recogida de basura, empezará a saltarse llamadas al método draw con el objetivo de conseguir la frecuencia deseada en el método update.
- **Tiempo variable**: XNA ejecuta los métodos update y draw de forma alternativa lo más rápido posible, y es tarea del programador llevar un control del tiempo y controlar que el juego se ejecute a la misma velocidad en todos los sistemas y controlar el tiempo transcurrido entre ciclos.

La segunda opción proporciona una mayor libertad, y mayor potencia de cálculo, pero es más compleja de desarrollar. Por comodidad y por ser suficiente para el juego que iba a desarrollar decidí realizar la implementación siguiendo el primer modo, ejecutando el juego a 60 fps.

Animacion

XNA no ofrece soporte nativo para animaciones de sprites por lo que es necesario implementarlo. La clase Animation, cuenta con una sola imagen que contiene todos los fotogramas de la animación, ya sea bien en una única fila o en una única columna. La clase se encarga de dibujar la parte de la imagen que corresponda con el fotograma actual en cada momento. En el momento de su creación, una instancia de la clase Animation toma como parámetros el número de fotogramas incluidos en la imagen, un array con la duración en milisegundos de cada uno de los fotogramas, el nivel de profundidad en el que será dibujada la imagen, y si la secuencia de fotogramas es vertical o horizontal. Según esta construcción, cada uno de los fotogramas puede tener una duración independiente del resto pero todos los fotogramas tienen el mismo tamaño.

Posteriormente se puede configurar la animación para indicar los límites entre los que tiene que moverse la animación de forma que se reproduzca solamente una subsecuencia de la animación y el modo de reproducción. La animación puede reproducirse en dos modos diferentes. En uno de ellos, una vez la animación ha alcanzado el último fotograma de la secuencia vuelve a empezar desde el principio, y en el otro la animación se reproduce con un vaivén, de manera que tras llegar al último fotograma la animación se reproduce hacia atrás, y así continuamente.

Imagen 20: Secuencia de animación del personaje Hades corriendo

La clase animación cuanta con sus propios métodos Update y Draw que le permiten actualizar el fotograma actual según el tiempo transcurrido y dibujarse en pantalla. El método Draw cuenta además con varias sobrecargas para dibujar la animación invertida vertical o horizontalmente e incluso dibujar solamente una parte de la imagen del fotograma actual. Por último cuenta con métodos propios de la reproducción de una secuencia de video como Play(), Stop(), StopAtFrame(), goToFrame(), etc.

Batalla

La clase Battle es la clase central de la parte jugable del juego. Es lo más parecido al motor del juego y se encarga de realizar la coordinación de los diferentes elementos del juego: Creación de las plegarias, recogida de las plegarias por los jugadores, detección y gestión de las colisiones jugador-jugador y jugador escenario y lógica de la partida según la configuración elegida. Sus atributos principales son los elementos de la partida: Un Array de 4 **jugadores**, un Array de 4 **HUD**, una lista de **plegarias** y un **escenario**.

Hay dos tipos de **colisiones** en el juego. Las colisiones donde los objetos chocan físicamente, afectándose uno a otro en su posición y velocidad, como cuando un personaje salta sobre otro o un personaje anda por encima de una plataforma y las colisiones de daño, donde se causa daño a un personaje, por ejemplo a través del ataque de otro personaje. Las primeras, son gestionadas por la clase Battle, mientras que las segundas son gestionadas por el propio objeto que causa el daño.

Para la detección de colisiones físicas se ha utilizado como elemento básico el rectángulo. Todas las detecciones de colisiones se resumen en detectar si dos o más rectángulos intersectan, lo que es sumamente sencillo y además ya viene implementado por XNA. De esta forma, los personajes y los escenarios definen de alguna forma sus zonas de colisión mediante rectángulos que evolucionan según el estado del personaje/escenario y su posición en pantalla. La clase Battle de esta forma no tiene que conocer el estado interno del escenario o de los personajes, únicamente tiene que conocer sus zonas de colisión. Esto es posible gracias a los métodos getColisionAreas implementados tanto por los personajes como por los escenarios.

Las colisiones de los personajes están definidas por dos rectángulos, uno para la cabeza y otro para el cuerpo, dado la naturaleza desproporcionada de su aspecto, lo que además permite realizar acciones diferentes según el lugar donde se detecte la colisión y simplificar los cálculos: por ejemplo, para detectar si un personaje está sobre una plataforma solo hace falta comprobar el cuerpo. Cada personaje se encargará de definir la correcta posición y tamaño de las áreas de colisión según el estado del personaje.

Imagen 21: Ejemplo de zonas de colisión de Zeus corriendo

Para los escenarios, se definen principalmente dos listas diferentes de rectángulos, una para zonas de colisión compactas y otra para las “one way”. La diferencia reside en que las zonas de colisión compactas no se pueden atravesar en ningún momento, y el personaje choca con ella si las atraviesa desde cualquier posición, mientras que en las colisiones “one way” el personaje solo colisiona con ella si la atraviesa desde arriba, es decir; se permite que el personaje la atraviese saltando de abajo a arriba. De igual forma será el propio escenario el que se encargue de gestionar estas áreas.

En cuanto a las zonas de daño, se ha creado la interfaz **AtackArea** con los métodos:

- bool intersecs(Rectangle rect)
- bool Intersects(Rectangle[] rects

Que permite crear áreas de colisión se cualquier forma siempre que se pueda determinar su intersección con rectángulos. Aunque de momento esta interfaz solo la implementa la clase AtackAreaRect que representa una lista de rectángulos en el futuro se podrán implementar otras formas.

Los métodos más importes de esta clase son los siguientes:

- **Start():**
  Puesto que los personajes y el escenario se escogen poco a poco a través de los menús, es necesario crear un método que se encargue de cargar los contenidos necesarios para los personajes y escenario seleccionado y realizar la inicialización de la batalla según la configuración elegida.

- **GeneratePrayer():**
  Genera una nueva plegaria en el escenario. Cada escenario contiene y controla una lista de areas donde es posible generar una plegaria. La clase Battle genera una plegaria en función de estas áreas. En primer lugar elige un área de todas posibles de forma aleatoria, pero ponderada según el tamaño del área, y el segundo lugar genera una posición aleatoria dentro del área seleccionada.

- **Update(GameTime gameTime):**
  Implementa la lógica del juego y sus colisiones. Ejecuta la siguiente secuencia de acciones:
  1. Comprueba si han muerto personajes en el ciclo anterior y en caso afirmativo actúa según la configuración de la partida.
  2. Guarda las posiciones actuales de los personajes y sus zonas de colisión para en caso de que se produzca una colisión, saber la forma en la que se ha producido, por ejemplo; si el personaje intersecciona con una plataforma saber si anteriormente se encontraba arriba o debajo de esta.
  3. Llama a los métodos Update de todos los personajes, que actualizan su estado y su posición en función de las acciones del jugador.
  4. Llama al método Update del escenario que actualiza su estado.
  5. Aplica las físicas sobre los personajes, lo que básicamente cambia su velocidad y su posición hacia abajo, incluso si estos estaban sobre alguna plataforma.
  6. Comprueba las colisiones:
     a. Con el escenario. Si el personaje intersecciona con una plataforma, según el tipo de plataforma (compacta o “one way”) y la posición en la que se encontraba el

jugador previamente se producirá una colisión o no. Si se produce una colisión se corrige la posición del jugador. Como la física ha desplazado la posición del personaje hacía abajo, si no se ha producido ninguna posición el personaje se encuentra en el aire cayendo y hay que cambiar su estado.

b. Entre personajes. Si dos personajes colisionan, y además en el ciclo anterior un personaje estaba encima de otro se ha producido el salto de un personaje sobre otro y hay que obrar en consecuencia. Se cambia el estado de los personajes y su velocidad. El que ha saltado pasa a estado "Saltando" y recibe un impulso hacia arriba, el otro recibe un impulso hacia abajo y sufre una deformación gráfica.

7. Todos los personajes ejecutan su método CheckAtacks que comprueba si sus ataques interseccionan con el resto de jugadores y les causa los daños y efectos oportunos. Esto se realiza después de que se haya llamado al método update de todos los jugadores, y no dentro de él para que al realizar las comprobaciones todos los personajes hayan actualizado su posición el mismo número de ciclos.
8. Se comprueba si los personajes están en contacto con una plegaria y la pueden recoger, y se genera una nueva plegaria si ha pasado el tiempo suficiente desde que se generó la anterior.

- **Draw():**

  Dibuja toda la escena en la pantalla propagando la llamada Draw a todos los objetos necesarios. Según el tipo de partida y el número de jugadores dibuja el HUD en una posición u otra de la pantalla. Al tratarse de gráficos en 2D sin profundidad, es muy importante que el desarrollador defina explícitamente que imágenes deben dibujarse encima de otras y cuales debajo, o en otras palabras definir un orden para el dibujado de las imágenes. XNA permite asignar una profundidad a una textura en el momento de dibujarla, de forma que independientemente del orden en que se dibujen las texturas, al terminar el dibujado XNA las ordenará y mostrará correctamente. Dado el gran número de imágenes que se pueden estar dibujando simultáneamente y la dificultad para el desarrollador de controlar todas las profundidades para que las imágenes se muestren en el orden deseado, se ha optado por colocar todas las profundidades en una clase estática llamada Depths mediante atributos estáticos. Esto permite ver las profundidades y observar el orden de un vistazo en un solo fichero.

Character

La clase Character es una clase abstracta que proporciona la estructura básica para crear personajes heredando directamente de ella. Por una parte proporciona métodos virtuales que definen la interfaz que deben respetar todos los personajes para comunicarse con el resto del juego y por otra proporciona mediante atributos y métodos implementados aquellos comportamientos que son comunes a todos los personajes.

Así pues la clase Character contiene atributos y constantes referentes a:

- **Configuración del movimiento**: Puesto que todos los personajes se van a mover de la misma forma o al menos de forma muy similar se pueden definir a este nivel la velocidad, y la forma del salto. El salto se puede definir a través de 3 parámetros:
    - La cantidad de impulso que recibe el jugador la primera vez que pulsa el botón de salto.
    - El impulso extra que recibe el jugador por cada ciclo que mantiene pulsado el botón de salto una vez está en el aire.
    - El número de ciclos que puede recibir impulso extra, ya que no queremos que el jugador salte infinitamente.
- **Estado del jugador a lo largo de la partida**, como la vida, el número de plegarias recogidas la posición y la velocidad.
- **Control del escudo**, puesto que el escudo de todos los jugadores funciona igual y únicamente varía el color de un jugador a otro.
- **Interfaz** del estado del jugador para el resto de clases, para saber por ejemplo si se le puede causar daño al jugador o si se le puede aplicar movimiento.
- **Animaciones y efectos comunes a todos los juagadores:** Como las animaciones de Idle, Jumping y Running, y los efectos gráficos de electrocución y sufrir daño.

Cada personaje es el sí mismo un autómata de estados que va cambiando de un estado a otro en función de las acciones del jugador y de su interacción con el resto de jugadores. En cada uno de los estados se mostrarán unas animaciones u otras y el jugador podrá realizar unas acciones u otras.

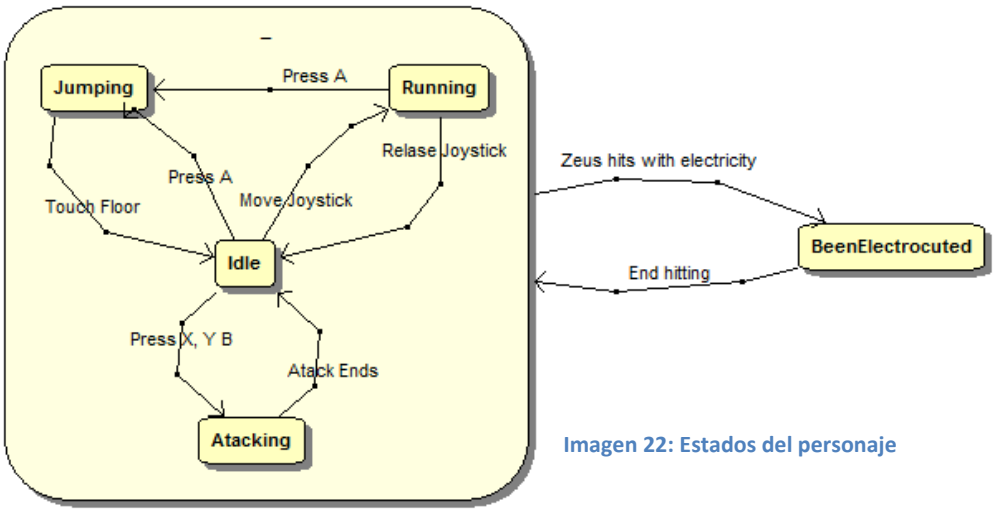


Imagen 22: Estados del personaje

Para una correcta experiencia multijugador es imprescindible que todos los personajes desarrollados estén equilibrados y que un jugador gane a otro solo dependa de su pericia. Para equilibrar el juego correctamente es necesario realizar muchas pruebas e ir variando el daño producido por los ataques poco a poco mientras se tienen en cuenta el daño del resto de los ataques. Con tal fin decidí colocar el daño de todos los ataques en una clase estática llamada Damages a través de atributos estáticos lo que facilita su modificación y comparativa.

Por último, como ya he comentado anteriormente, la clase Character proporciona una serie de métodos:

- **Update(GameTime gameTime):**
  Método básico de actualización que se llama cada ciclo y deben implementar las clases sucesoras. Realiza la actualización de los elementos comunes a todos los personajes: Control y animación del escudo, efecto de aplastamiento del personaje cuando le salta un personaje encima y actualización de la posición en función de la velocidad.
- **Draw():**
  Deben implementarlo las clases sucesoras. Dibuja el escudo.
- **GetColisionHead(), GetColisionBody():**
  Métodos puramente virtuales. Cada personaje define dos zonas de colision, una para la cabeza y otra para el cuerpo y lo hace mediante una serie de rectángulos distintos para cada uno de los estados y sus respectivas animaciones. Estos rectángulos se definen en coordenadas de la textura asociada a la animación, lo que conlleva los siguientes problemas:
  - Cuando el personaje está mirando hacia la izquierda hay que reflejar el rectángulo horizontalmente tomando como eje la posición del personaje.
  - Al pasar de un estado a otro y cambiar la zona de colisión tiene que mantenerse el valor de Posicion + Colision.Botton para que si el personaje estaba en el suelo permanezca en el suelo, y no sufra ciclos de caída, por ejemplo al estar parado y empezar a correr.
- **GetColisionAreas:**
  Devuelve las dos áreas anteriores en un array.
- **GetTotalArea:**
  Devuelve un rectángulo que inscribe a todas las zonas de colisión del personaje. Útil para cuando se requiere colisiones a groso modo y las colisiones independientes podrían dar problemas.

Una de las tareas más costosas a lo largo del desarrollo de todos los personajes, ha sido medir e implementar pequeños desplazamientos en la posición del personaje al cambiar de una animación a otra, para que el personaje apareciera siempre en la misma posición. Esto es debido a que no en todas las animaciones el personaje aparece en la misma posición exacta dentro del frame ni en todas las animaciones los frames miden los mismo, ya que, por ejemplo, en una animación el personaje puede tener los brazos pegados al cuerpo y en otra extendidos.

Zeus

Uno de los ataques de Zeus, el ataque recarga, cambia el estado de Zeus durante un tiempo, cargándose de electricidad lo que debe mostrarse gráficamente. Por este motivo es necesario distinguir entre el estado normal y el estado cargando mediante un booleano y las animaciones de Zeus de los diferentes estados tradicionales de un personaje están duplicadas. La lógica del personaje distingue entre ambos estados para actualizar unas animaciones u otras y de forma similar a la hora del renderizado se elegirá una animación u otro dependiendo de si está cargado o no. Este estado es de una duración limitada, por lo que se dibuja una barra de carga que se va vaciando automáticamente y que indica la duración encima de la cabeza de Zeus.

En otro ataque desarrollado, Zeus lanza un rayo. En este caso el efecto se compone de dos animaciones independientes, Zeus en sí y el rayo. Como he dicho anteriormente, Zeus puede encontrase cargado o no cuando lance este ataque. Para el propio Zeus, se cuenta con dos animaciones diferentes que hay que seleccionar según el estado, pero para el rayo se trata de una sola animación de la que vario su tamaño en función del estado. Además, en la animación del rayo todos los frames contienen el rayo al mismo tamaño, desplegado hasta el final, pero quise hacer que el rayo apareciera en la mano de Zeus y fuera creciendo hasta alcanzar el tamaño máximo. Esto se consigue mostrando únicamente una parte del rayo que va aumentando poco a poco con el paso del tiempo mientras la animación de reproduce el bucle. De igual forma va aumentando el tamaño de la zona de impacto del ataque, definida como un rectángulo.

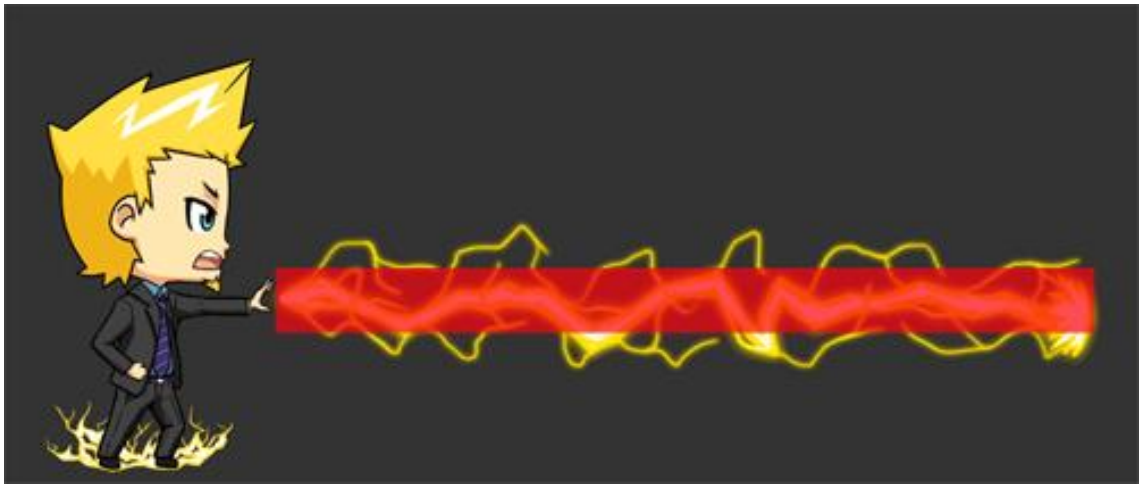


Imagen 23: Zeus lanzando un rayo y zona de colisión del ataque

La animación del rayo cuenta con 6 frames, pero sigue siendo completamente correcta si se voltea verticalmente. Para aumentar el número de frames y dar mayor efecto de aleatoriedad al rayo, la animación se reproduce tanto normal como volteada. Cada vez que la animación se encuentra en el frame central, hay una porcentaje de que la animación se volteé, y seguirá reproduciéndose volteada hasta que se vuelva a voltear de forma aleatória.

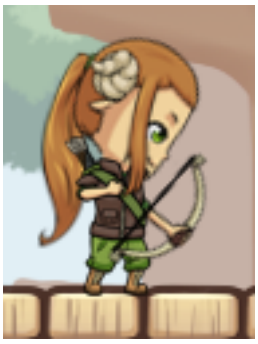
Artemisa

El principal ataque de Artemisa consiste en un disparo con su arco. Se quería conseguir que Artemisa pudiera disparar en todas las direcciones, controlando el jugador la dirección con el joystick. Para que esto quedara realista es necesario que el brazo de Artemisa indique la dirección del disparo apuntando en la misma dirección que el joystick del jugador, y que artemisa incline la cabeza arriba y abajo en la misma dirección.

Con ese objetivo se ha montado un "esqueleto" de Artemisa formado por una serie de partes del cuerpo sueltas y unos puntos de anclaje alrededor de los cuales estos pueden girar. En primer lugar se anclan al cuerpo la cabeza y el brazo en su lugar correspondiente que girarán con respecto al cuello y el hombro respectivamente. Estos en principio se giran para que apunten en la misma dirección que indique el jugador con el joystick, pero se define un ángulo máximo y mínimo para que el movimiento resulte más realista. Este ángulo es diferente para el brazo y para la cabeza. Además, si el ángulo es superior a 90 o inferior a -90 se voltea horizontalmente al personaje para que apunte hacia la izquierda y se establecen los respectivos ángulos máximo y mínimo volteados. Esta tarea es especialmente complicada, puesto que si el personaje está volteado deben de voltearse los puntos de anclaje, voltearse las imágenes y convertir en ángulo a un ángulo del primer o cuatro cuadrantes para que todo funcione correctamente.

Imagen 26: Esqueleto de Artemisa

25: Mirando hacia arriba



24 Mirando hacia abajo

Una vez construido el esqueleto básico se mejoró para que la coleta y el mechón de pelo tuvieran un comportamiento natural y cayeran siempre hacia abajo independientemente de la rotación de la cabeza. Esto conlleva una complicación extra puesto que en este caso el punto de anclaje no está fijo, sino que depende de la rotación de la cabeza y hay que primero voltearlo en caso de que sea necesario y posteriormente girarlo al igual que la cabeza. En este caso hay que aplicar el inverso del ángulo aplicado a la cabeza para, deshacer dicho movimiento. Además, para una mejor sensación de movimiento al igual que en la animación Idle, se hizo que el pelo tuviera un leve movimiento constante de balanceo hacia adelante y hacia atrás, independiente de la rotación de la cabeza. Hay además una distinción entre cuando está apuntando para disparar y cuando ya ha disparado, dibujando en un caso el arco

con la cuerda tensa, y la flecha encima con el mismo ángulo que el brazo y en otro caso el arco con la cuerda extendida.

Como apuntar es bastante difícil, se dibuja una guía de mini flechas que indican la trayectoria que seguirá la flecha una vez disparada. Para ello se van realizo consecutivamente ciclos de simulación de la física para definir la posición de las flechas hasta que estas se salen de la pantalla. En cada momento las flechas apuntan en la misma dirección que su velocidad. Lo mismo se aplica a la flecha real una vez disparada que va variando su velocidad, posición y orientación en función de la física aplicada sobre ella.

En su segundo ataque, Artemisa coloca una trampa en el suelo que causará daño a los enemigos que las pise. Para ello en primer lugar se realiza un test, comprobando colisiones mediante rectángulos, para saber si el lugar en el que se colocaría la trampa es correcto. Un lugar es correcto si la trampa se queda completamente encima de una plataforma, si no se solapa con otras trampas y no se solapa directamente con otros jugadores. Si la posición es incorrecta se dibuja la trampa con un tinte rojizo y se la va haciendo desaparece disminuyendo su opacidad poco a poco. Si por el contrario la posición es correcta, la trampa se introduce en una lista de trampas activas que pose Artemisa donde permanecerá a la espera hasta que algún jugador colisione con ella.

Una vez se detecta la colisión de la trampa con un personaje distinto de Artemisa, la trampa se activa, causando daño al objetivo que iniciará su animación de daño y activará un booleano que le impedirá moverse e iniciándose una animación de cierre de la trampa. Una vez la trampa ha finalizado su animación, se elimina de la lista de trampas activas y se vuelve a permitir al jugador moverse cambiando su booleano.

Scenario

De forma similar a la clase CCharacter, la clase Scenario es una clase abstracta que proporciona la estructura básica para crear escenarios heredando directamente de ella. Proporciona tanto métodos virtuales que definen la interfaz que deben respetar todos los escenarios como atributos y métodos que implementan aquellos comportamientos que son comunes a todos los escenarios.

La clase escenario contiene algunos de los atributos ya comentados anteriormente como las listas de áreas de colisiones y áreas donde es posible generar las plegarias y que las clases descendientes tendrán que gestionar convenientemente, creándolos y actualizándolas si es necesario. Además, contiene una imagen de fondo, una canción reproducida en bucle y los límites que delimitan el espacio jugable que típicamente coinciden con el espacio en pantalla. A los típicos métodos abstractos de Update(), LoadContent() y Draw() se le une el método Start() que realiza las acciones que el escenario debe realizar una única vez, como comenzar la reproducción del sonido de fondo.

Forest

Para este escenario se ha pretendido simular los efectos del viento sobre la vegetación, moviéndose las copas de los árboles y los arbustos según la dirección e intensidad del viento, con un efecto de partículas que simula la caída de hojas de los árboles y que se mueven con el viento y con un efecto sonoro acorde. La clase controla el viento mediante los métodos StartWind(), StopWind() y UpdateWind().

El viento viene determinado por cuatro parámetros:

1. **IsWindy**: Un booleano que indica si actualmente el viento está soplando.
2. **Direction**: Indica si el viento sopla hacia la izquierda o hacia la derecha.
3. **TimeWind**: Tiempo restante hasta que el viento deje de soplar si está activo.
4. **TimeWait**: Tiempo restante hasta que el viento vuelva a soplar si no está activo.

Que son generados aleatoriamente cada vez que el viento sopla.

Podemos distinguir 3 estados diferentes en el viento: Activo, parado y parando. Los elementos que presentan movimiento con el viento tienen largas animaciones, pero no se reproducen de forma completa. El fotograma central de la animación representa el estado de reposo, el estado en el que permanece la animación cuando no sopla viento. Cuando sopla el viento, según su dirección se reproducirá la primera o la segunda mitad de la animación en forma de vaivén.

En realidad, la animación comenzará en el fotograma central y se moverá en una dirección pero una vez llegado al final de la animación no volverá hasta el fotograma central sino que se moverá entre un número menor de fotogramas.

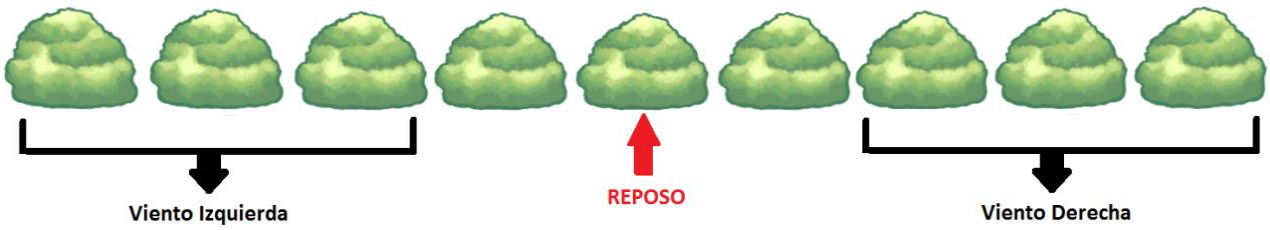


Imagen 27: Reproducción de la animación de un arbusto según la dirección del viento

Cuando se inicia un viento, se configuran las animaciones según la dirección del viento, se comienza la reproducción del efecto de sonido y se inicia la reproducción de las animaciones, pero no todas al mismo tiempo. Se establece un pequeño retraso en el inicio de las animaciones según la posición de los elementos en el escenario y la dirección del viento, para simular que el viento viene en dicha dirección.

Una vez deja de soplar el viento, los elementos deben de volver a su frame de reposo, pero deben de hacerlo de forma progresiva. Se configuran las animaciones para que vuelvan al frame de reposo y a una menor velocidad que la velocidad de reproducción cuando soplaba el viento. Además se disminuye progresivamente el volumen del efecto de sonido para simular que el viento deja de soplar poco a poco.

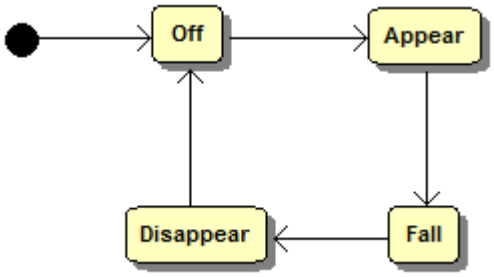
Por último el escenario cuenta con un efecto de partículas que simula las hojas cayendo de los árboles. Este efecto se consigue con las clases **Leaf** y **LeafEffect**. Cada hoja es un objeto independiente que se mueve cayendo describiendo círculos girando alrededor de un punto exterior a la hoja a una velocidad propia de cada hoja. Además en caso de que haya viento a la velocidad se le suma un componente horizontal según la dirección del viento que se va reduciendo progresivamente cuando el viento deja de soplar. Las hojas se detienen al alcanzar el límite inferior del escenario para desaparecer progresivamente.

Las hojas se crean de forma aleatoria tanto si hay viento como si no, aunque en caso de haberlo se crean con mayor frecuencia. Si no hay viento, las hojas se generan aleatoriamente en una posición pegada al borde superior de la pantalla mientras que si hay viento también se generan en el lateral por el que entra el viento en el escenario. El efecto cuenta con 4 texturas de hojas diferentes. En el momento de la creación se decide aleatoriamente que textura se va a utilizar y si se aplica algún efecto de volteo vertical o horizontal para crear mayor diversidad.

Tartaro

Este escenario presenta tres características especiales, un rio de lava que fluye continuamente en el inferior del escenario, un efecto de partículas que representan trozos de ceniza ardiendo y que surgen desde el rio de lava y unos desagües que vacían chorros de lava de manera intermitente situados en la parte superior del escenario y que dañan a los personajes.

El rio de lava que fluye continuamente se consigue mediante dos sprites, de forma que el final de uno coincide con el principio del otro y viceversa, así no hay más que ir dibujando uno detrás de otro continuamente. Los desagües, pueden verse como autómatas de estados con 4 estados que se reproducen de forma cíclica.



El desagüe está en reposo, en el estado **Off**, hasta que vence un contador de tiempo que se ha definido de forma aleatoria. En este momento pasa al estado **Appear** en el que el desagüe se ilumina progresivamente interpolando entre un sprite con el desagüe apagado y otro encendido. Además se reproduce una pequeña animación de lava apareciendo.

Imagen 28 Desagüe encendido y desagüe apagado

Cuando la lava ya ha aparecido, pasamos al estado **Fall** en el que la lava comienza a caer. Para mayor diversidad se elige de forma aleatoria entre dos sprites diferentes de diferente longitud que además pueden voltearse. Una vez la lava alcanza el extremo inferior de la pantalla se añade además una efecto de salpicadura en el área de caída. Cuando la lava ha terminado de caer pasamos al estado **Dissappear** en el que la chimenea vuelve a apagarse, y se programa un nuevo temporizador para volver a empezar.

El efecto de partículas consiste en unas partículas ascendentes de velocidad, tamaño y color aleatorio que en este caso presenta la particularidad de que las partículas cambian su velocidad en el eje X cada cierto tiempo de forma aleatoria.

RT-DESK

¿Qué es RT-DESK?

RT-DESK es un núcleo de simulación de aplicaciones gráficas en tiempo real que sigue el paradigma discreto desacoplado. Es el acrónimo de **R**eal **T**ime **D**iscrete **E**vent **S**imulation **K**ernel. Básicamente es un núcleo de gestión de eventos discretos ordenados temporalmente modelado mediante el paso de mensajes.

Un evento es el cambio de estado de una entidad del sistema que ocurre en un instante de tiempo determinado. RT-DESK hace hincapié en la gestión discreta de los eventos. La gestión discreta difiere de la gestión continua en cuanto a que en ella continuamente se está preguntando al sistema sobre la ocurrencia del evento, mientras que en la gestión discreta la ocurrencia del evento desencadena la respuesta del sistema. De esta forma se libera a la CPU del trabajo repetitivo y poco productivo de preguntar constantemente por la ocurrencia del evento.

Con una aplicación gráfica implementada con RT-DESK todos los objetos del sistema se modelan mediante eventos discretos. Esto permite simular eventos continuos según una frecuencia de tiempo constante o permite ir modificando la frecuencia de los eventos en función de las necesidades del sistema. Además, en la aplicación pueden coexistir simultáneamente comportamientos discretos y comportamientos continuos, incluso, dentro del mismo objeto. Esta dualidad permite que RT-DESK funcione tanto en aplicaciones gráficas en tiempo real, para la que está diseñado, como por ejemplo videojuegos y aplicaciones de realidad virtual, como en aplicaciones de simulación discreta no interactivas.

RT-DESK no es una aplicación, y no puede funcionar de forma aislada. Es un núcleo de simulación que debe integrarse con una aplicación gráfica en tiempo real y que una vez integrado únicamente se encargará de la gestión de los eventos del sistema y de la comunicación de los objetos mediante el paso de mensajes. No realiza ninguna tarea asociada a los mensajes y es el propio objeto receptor perteneciente a la aplicación y no a RT-DESK el que ejecutará una tarea asociada al mensaje. RT-DESK es una herramienta desarrollada para dar soporte al desarrollo de aplicaciones gráficas en tiempo real mediante el paradigma discreto desacoplado.

En definitiva, sus funciones son:

- Modelar el mecanismo de paso de los mensajes.
- Gestionar el proceso de envió de mensajes, asegurando que lleguen a su destino en el instante de tiempo indicado y por orden de ocurrencia.

RT-DESK se proporciona como un API desarrollado en C++ en forma de una colección de ficheros cabecera y un fichero dll que se pueden incorporar a cualquier aplicación gráfica en tiempo real o de simulación. El código fuente no puede ser accedido por parte de los programadores externos, aunque si pueden acceder a su funcionalidad.

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 llamado DESK [DESK]. Posteriormente se le realizaron una serie de ajustes para incluir el tiempo real y poder soportar simuladores de entornos virtuales interactivos. El resultado fue una tesis doctoral [S.H] en la que se aplicaba este núcleo a un sistema de tiempo real como son los videojuegos y se le llamo GDESK [GDESK]. Las principales mejoras de esta versión fueron:

- **La gestión de Tiempos**: En un simulador de eventos discretos el tiempo del simulador viene determinado únicamente por el tiempo de ocurrencia de los eventos. El reloj de simulación avanza cada vez que se produce un evento en el sistema. Los eventos se ejecutan consecutivamente, sin esperas entre ellos. Este tiempo no tiene relación alguna con el tiempo real, pues lo que interesa es el comportamiento del sistema para extraer conclusiones. Por el contrario, en una aplicación en tiempo real los eventos deben ocurrir en el instante de tiempo real en el que se ha indicado que ocurran. GDESK utiliza una aproximación mixta en la que el tiempo avanza con la ocurrencia de eventos, al igual que en DESK, pero únicamente si el tiempo real ha alcanzado el tiempo del evento.
- **La definición de Objetos**: En GDESK la aplicación pasa a ser un conjunto de objetos que intercambian mensajes que activan eventos. Todos los objetos de la aplicación heredan de la clase base de GDESK permitiendo que todos los objetos hereden la gestión de mensajes entrantes y salientes.
- **La definición de Mensajes**: Un mensaje modela la comunicación entre dos objetos o el comportamiento de un objeto. Un mensaje siempre lo genera un objeto y lo recibe un objeto. Parte de los atributos del mensaje son propios del simulador y parte los define el programador de la aplicación. Cuando un mensaje llega a un objeto este deja de tener utilidad y pasa a la pool de mensajes.
- **Determinar la finalización de la ejecución**: La finalización en GDESK debe de realizarse por medio de un evento de petición explicita del usuario o bien por una condición de error.

Más adelante se realizaron una serie de mejoras para el mantenimiento y control de la simulación además de una re ordenación de todo el código. A este nuevo avance se le ha llamado RT-DESK, e introduce las siguientes mejoras:

- **Temporizador interno** de alta resolución para control y monitorización
- **Preparación** para añadir un **Monitor** del núcleo.
- **Refactoring interno** que ha cambiado la nomenclatura del API, optimizando algunas funciones, añadiéndose nuevas y generando documentación automática.

Por último se ha estado probándolo en entornos más actualizados con el fin de validar la vigencia del API. Los resultados pueden verse en una tesina de máster [CPD]

Funcionamiento de RT-DESK

Como ya he comentado anteriormente RT-DESK modela la ocurrencia de eventos mediante el paso de mensajes utilizando dos entidades básicas: los objetos y los mensajes. Cuando un objeto envía un mensaje puede hacerlo en tiempo real, donde el mensaje se enviará inmediatamente, o puede asociarle un tiempo donde el mensaje no se enviará hasta transcurrido dicho tiempo de simulación. RT-DESK mantiene los eventos ordenados en función del tiempo en el que han de ser entregados hasta que el tiempo transcurrido de simulación excede y entonces son enviados al objeto destinatario encargado de su recepción, que ejecutará algún método en consecuencia. De esta forma RT-DESK controla a los objetos de la aplicación mediante el paso de mensajes.

La parte central de RT-DESK es el Dispatcher, que se encarga de realizar el paso de mensajes, recibiendo el mensaje, almacenándolo y entregándolo a su destinatario en el instante preciso del tiempo de simulación. También es el responsable de mantener el orden temporal de todos los eventos pendientes de envío. En este marco cualquier objeto de la puede enviar mensajes de eventos a otros objetos o a sí mismo. Que cada objeto se ejecute con una frecuencia propia, que satisfaga sus propias necesidades se consigue mediante el envío de mensajes a sí mismo. Si un objeto desea ejecutar un método cada t unidades de tiempo, no tiene más que enviarse un mensaje a si mismo cada vez que finalice dicho método para recibirlo en t unidades de tiempo y volver a ejecutarlo. Todos los elementos de la aplicación pasan a ser objetos independientes que se ejecutan según sus propias necesidades, incluyendo cada elemento del mundo virtual, el proceso de renderizado, la entrada de usuario, etc. De esta forma se consigue que cada objeto solo consuma CPU cuando es imprescindible y que el renderizado se ejecute con su propia frecuencia independiente de la lógica de la aplicación.

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

Benchmark

“Contrary to much of our profession's folklore, a good testing question doesn't necessarily have a definitive, expected result. Tests that are designed to confirm a prevailing theory tend not to reveal new information.”

-Michael Bolton, in “How Testers Think“.

Preparando el proyecto

La clase Game

RT-DESK sustituye al núcleo de un videojuego, esto es, al bucle principal realizando una gestión discreta y desacoplada de todos los eventos del sistema. El primer problema que surge al integrar RT-DESK con XNA es que como he comentado anteriormente XNA proporciona un bucle principal ya implementado y oculto para el desarrollador a través de la clase Game. Más concretamente todos los juegos de XNA deben de heredar de la clase Game y sobrescribir los métodos Update() y Draw() de esta. La clase Game se encarga de llamar a estos métodos de forma automática una vez iniciado el juego con el método Game.run().

Para poder utilizar RT-DESK la primera idea fue no ejecutar la llamada al método Game.run() y programar mi propio bucle principal que llamase a los métodos Update() y Draw() pero está solución no dio resultado. Descubrí que en el bucle principal interno de la clase Game, además de realizar las llamadas a los métodos Update() y Draw() se realizan otras tareas como gestión de la ventana del juego y paso de mensajes. Para que todo funcionara correctamente la solución final consistió en implementar mi propia clase de juego que además del bucle principal realizara las tareas de gestión de la ventana del juego y paso de mensajes.

En primer lugar la clase Game contiene un proveedor de servicios que facilita la comunicación entre algunos componentes imprescindibles del juego. En C# se definen los proveedores de servicios como un mecanismo para la comunicación de objetos de forma sencilla, escalable y descentralizada. Un servicio no es más que un método que implementa una función concreta. Un servicio se registra en un proveedor de servicios con un identificador que define el tipo de servicio prestado y posteriormente cualquier clase que quiera hacer uso de un tipo de servicio este no tiene más que preguntar por él al proveedor de servicios y obtendrá una implementación de este servicio. Para el correcto funcionamiento de mi propia clase Game es necesario crear un proveedor de servicios implementando la interfaz *IServiceProvidery* el servicio *GraphicsDeviceService* implementando *IGraphicsDeviceService* que proporciona el dispositivo gráfico allí donde se solicite.

En segundo lugar, mi clase Game debe realizar el control de la ventana de juego. La forma de realizar esto en .NET es mediante la clase *form* del espacio de nombres *System.Windows.Forms* que representa una ventana en cualquier tipo de aplicación gráfica. La clase Game debe configurar la ventana de juego, configurando el formato y permitiendo a su usuario modificar en todo momento su tamaño y si es o no pantalla completa. El resultado final es muy parecido al que obtenemos con OpenGL y Glut. El usuario define un método de callback Idle y *System.Windows.Forms* realizará la gestión de los eventos de usuario y de ventana, llamando al método de Idle siempre que no haya eventos que procesar. En este método se creará el núcleo del videojuego, el bucle principal o RT-DESK según el caso, cediendo el control a *form* cuando se detecte algún evento de usuario.

Acople C++ y C#

RT-DESK se proporciona en forma de biblioteca de clases escrita en C++ por lo que está compilada a código máquina. XNA, por el contrario trabaja en C#, lenguaje orientado a objetos desarrollado y estandarizado por Microsoft como parte de su plataforma .NET que compila al código intermedio CIL para ser ejecutado por la CLR. Esta diferencia hace que las librerías de C++ no puedan ser llamadas directamente desde C#.

C# proporciona el servicio **PInvoke** que permite llamar a funciones de bibliotecas nativas, pero este no permite la creación y utilización de clases nativas. Para poder crear y utilizar clases nativas desde C#, normalmente hay dos soluciones:

1. Crear una extensión en C++ de esta librería, ya sea en la propia librería o en una nueva librería que transforme los métodos de las clases de la librería nativa, incluyendo los constructores en funciones globales que reciban dicha clase como parámetro y posteriormente utilizar **PInvoke** para acceder a estas funciones. Por ejemplo, si tenemos la clase:

   ```
   class CUnmanagedTestClass {
   public:
       CUnmanagedTestClass();
       void PassInt(int nValue);
   };
   ```

   Tendríamos que crear las siguientes funciones en C++:

   ```
   CUnmanagedTestClass* CreateTestClass()
   {
        return new CUnmanagedTestClass();
   }

   void CallPassInt(CUnmanagedTestClass* pObject, int nValue)
   {
        pObject->PassInt(nValue);
   }
   ```

   Y utilizar PInvoke desde C#:

   ```
   [DllImport("ExampleUnmanagedDLL.dll")]
   static public extern IntPtr CreateTestClass();

   [DllImport("ExampleUnmanagedDLL.dll")]
   static public extern void CallPassInt(IntPtr pTestClass, int Value);
   ```

2. Utilizar el lenguaje C++/CLI para crear una clase que actúe de puente entre C++ y C#. C++/CLI es un lenguaje creado por Microsoft con el fin de facilitar la integración entre código nativo y código CIL. Es muy parecido a C++ y proporciona soporte para crear clases tanto nativas como CIL. Las clases CIL son compatibles con todos los lenguajes de la plataforma .NET por lo que desde C++/CLI podemos crear una clase CIL que utilice

directamente a las clases nativas y posteriormente C# utilizará esta clase. Si tomamos como ejemplo la clase anterior, en C++/CLI tendríamos que crear la siguiente clase:

```
public ref class CBridge
{
public:
    CBridge();
    ~CBridge();
    void PassInt(int nValue);

private:
    CUnmanagedTestClass* m_pUnmanagedTestClass;
};

CBridge::CBridge()
{
    this->m_pUnmanagedTestClass = new CUnmanagedTestClass();
}

CBridge::~CBridge()
{
    delete this->m_pUnmanagedTestClass;
    this->m_pUnmanagedTestClass = NULL;
}

void CBridge::PassInt(int nValue)
{
    this->m_pUnmanagedTestClass->PassInt(nValue);
}
```

Que podríamos usar directamente en C#.

No obstante, dada la particularidad de mí caso, donde RT-DESK necesita crear clases que hereden de RTDESK_CEntity y sobrescriban su método ReceiveMessage() que será llamado directamente desde el código nativo, la única opción posible era utilizar el lenguaje C++/CLI para crear una clase de puente.

Con esta solución tengo que crear una clase en C++/CLI por cada una de las clases de RT-DESK que necesite utilizar, en mi caso para RTDeskEngine, RTDeskEntity y RTDESK_CMsg. Las clases RTDeskEngine y RTDESK_CMsg siguen el esquema introducido anteriormente pero la clase RTDeskEntity es diferente, ya que tiene un método virtual y se necesita que las clases CLI descendientes implementen este método de forma que sea llamado desde el código nativo y hay el problema de que las clases CLI no pueden heredar de las clases nativas. Para solucionar este problema se crean dos clases, una nativa (UnmanagedEntity) y una CLI (ManagedEntity).

```
class UnmanagedEntity : public RTDESK_CEntity
{
private:
	gcroot<ManagedEntity^> managed;
public:
	UnmanagedEntity(ManagedEntity^ managedEntity);
	void ReceiveMessage(RTDESK_CMsg *pMsg);
};
```

```
public ref class ManagedEntity abstract
{
private:
	UnmanagedEntity* unmanaged;
public:
	ManagedEntity();
	virtual void ReceiveMessage(ManagedMsg^ Msg) = 0;
	void SendMsg(ManagedMsg^ Msg, ManagedEntity^ Receiver, double DeltaTime);
};
```

La clase nativa puede heredar directamente de la clase RTDesk_CEntity y tiene un miembro del tipo de la clase CLI, de forma que cuando cuando se ejecute el método ReceiveMessage() esta redirige la llamada al método ReceiveMessage() de la clase CLI. De esta forma las clases pueden descender de ManagedEntity e implementar su método ReceiveMessage() y su atributo de la clase UnmanagedEntity se encargará de llamar al método cada vez que el mismo reciba una llamada al suyo. De igual forma la clase ManagedEntity tiene un atributo del tipo UnmanagedEntity para redirigir los métodos de RTDesk_CEntity que necesiten ser accedidos desde C#.

```
UnmanagedEntity::UnmanagedEntity(ManagedEntity^ managedEntity):RTDESK_CEntity ()
{
	managed = managedEntity;
	Message = gcnew ManagedMsg();
}

void UnmanagedEntity::ReceiveMessage(RTDESK_CMsg *pMsg)
{
	managed->ReceiveMessage(Message->toManaged(pMsg));
}
```

```
ManagedEntity::ManagedEntity(ManagedEngine^ Engine)
{
	unmanaged = new UnmanagedEntity(this);
	unmanaged->SetMsgDispatcher(Engine->Engine->MsgDispatcher);
}

ManagedEntity::!ManagedEntity()
{
	if (unmanaged) {
		delete unmanaged;
		unmanaged = 0;
	}
}

void ManagedEntity::SendSelfMsg(ManagedMsg^ Msg, double DeltaTime)
{
	unmanaged->SendSelfMsg(Msg->toUnmanaged(), DeltaTime);
}

void ManagedEntity::SendMsg(ManagedMsg^ Msg, ManagedEntity^ Receiver, double
DeltaTime)
{
	unmanaged->SendMsg(Msg->toUnmanaged(), Receiver->unmanaged, DeltaTime);
}
```

Las pruebas

Para estudiar el diferente comportamiento del paradigma continuo acoplado y discreto desacoplado se van a realizar dos pruebas diferentes. En primer lugar, en el escenario Tártaro desarrollado, se va introducir únicamente personajes controlados por la IA, variando su número entre 1 y 300 aproximadamente en ambos paradigmas y analizando no solo la diferencia de comportamiento entre las dos versiones si no también la diferencia de su evolución ante los diferentes niveles de carga. Dada la naturaleza aleatoria del comportamiento de la IA, para poder analizar correctamente la diferencia entre ambos sistemas con independencia de la aleatoriedad se realizaran 10 repeticiones para prueba.

En segundo lugar, en el mismo escenario, se realizará una prueba sin personajes donde se irá variando de forma similar el número de partículas creadas en el efecto de chispas del escenario. Como en la versión continua la partículas únicamente se pueden crear cada 16.6 ms, frecuencia de actualización de todo el sistema, para poder comparar los resultados en la versión discreta las partículas también se crearán únicamente cada 16.6 ms, variando el número de partículas que se crean en cada ciclo.

Por comodidad, se han creado dos proyectos diferentes en Visual Studio, uno con la versión continua acoplada y otro con la versión discreta desacoplada. Como he comentado anteriormente, debido a la naturaleza de RT-Desk para poder ejecutarlo en XNA ha sido necesario crear mi propia clase Game que sustituya a la clase Game de XNA y crear mi propio bucle principal. Para mayor objetividad, aunque no era necesario, he utilizado también mi propia clase Game en las pruebas de la versión continua acoplada, ejecutando todo el juego con una frecuencia de 60 fps.

Para cada una de las pruebas realizadas, tanto en su versión continua acoplada como en discreto desacoplado se van a recoger el tiempo acumulado de todas las fases de dibujado, actualización y tiempo de ocio a lo largo de 1 minuto de ejecución. Para ello, la medida de tiempos se integra con el bucle principal de la aplicación. Además mediremos los fps de dibujado conseguidos con ambos sistemas.

Podemos definir los siguientes métodos principales que de una forma u otra están en ambos sistemas:

- Render: Se encarga de dibujar la escena. La toma de tiempos se realiza acumulando el tiempo transcurrido entre el inicio y el final del dibujado.
- Idle: Es el tiempo de ejecución no útil. Representa el tiempo libre de CPU disponible para otros usos. La función se encarga de que el sistema este a la espera hasta que se requiera la CPU. La toma de tiempo de ejecuta de igual que el anterior.
- Update: Se encarga de realizar la fase de simulación del sistema. Se mide el tiempo transcurrido entre el principio y el final.

La diferencia entre el modelo continuo y el discreto es la forma en la que ejecutan estos métodos.

En el método continuo:

```
while (!End)
{
     Idle();
     Update(new GameTime(Time.Elapsed, StepTime));
     Render();
     nextGameTick += skipTicks;
}
```

Se alternan las fases de Update() y Render() continuamente. El método Idle() se encarga de hacer esperar al sistema para que funcione a la frecuencia deseada.

En el modo discreto:

```
while (!End)
{
      Idle();
      nextGameTick = Engine.Simulate();
}
```

Solamentente tenemos de forma directa el método Idle() que se encarga de hacer esperar al sistema. El método Render() es ejecutado con una frecuencia de 60 ciclos por segundo por una clase independiente que hereda de RTDeskEntity de forma completamente independiente de la simulación. De igual forma, el método de Update() se encuentra repartido entre todos los objetos del sistema que ejecutan su propio método Update() con su propia frecuencia e independientemente del resto. Para medir el tiempo de Update() en este contexto, se acumula el tiempo entre antes y después de la llamada a Engine.simulate() y se le resta el tiempo de render.

Entorno de pruebas

Hardware

Placa base: Asus CROSSHAIR IV FORMULA

Procesador: AMD Phenom II X6 1090T, 6 núcleos 3.2 Ghz

RAM: 8GB 1333 MHz (4x2GB), Doblebanda

Gráfica: Nvidia Geforce gtx 470, 1280MB GDDR5

Software

SO Windows 7 Profesional, Service Pack 1

Microsoft **Visual C#** 2010 Express

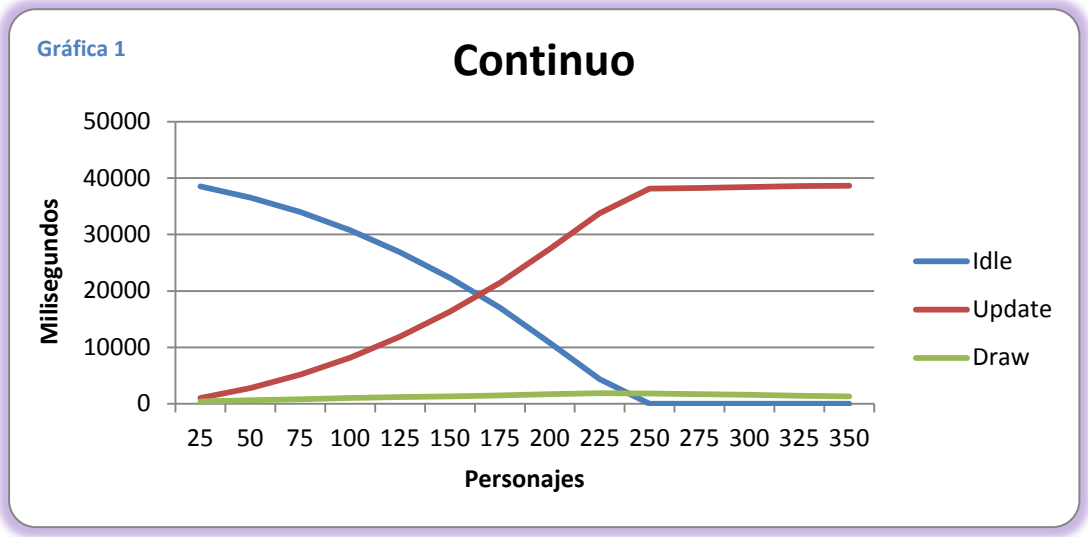
XNA 4.0

Resultados

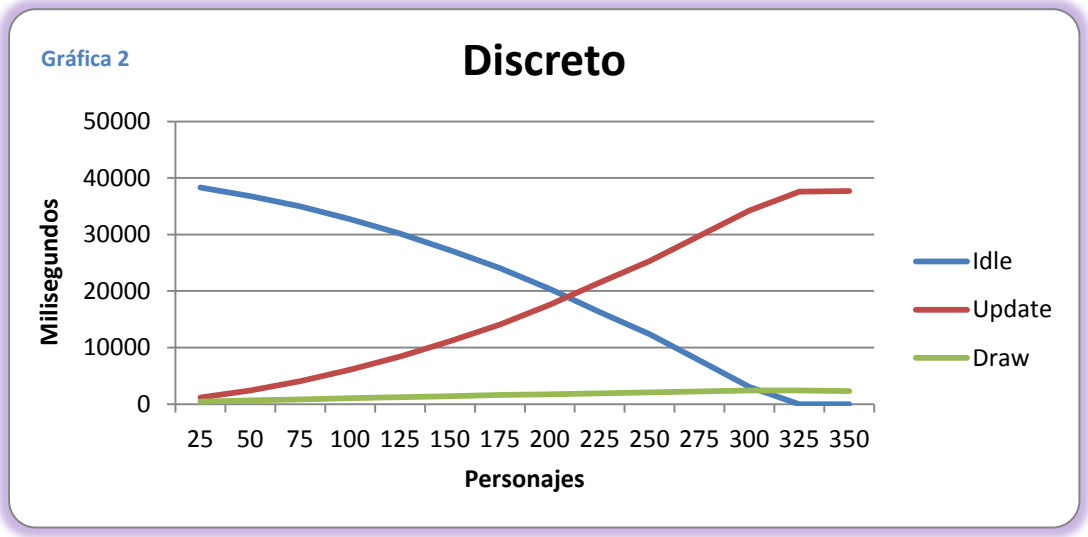
Prueba Personajes

En primer lugar mencionar que la migración al paradigma discreto desacoplado se realizó de tal manera que la experiencia del usuario y el resultado visual final de la aplicación es el mismo en ambos paradigmas.

Gráfica 1



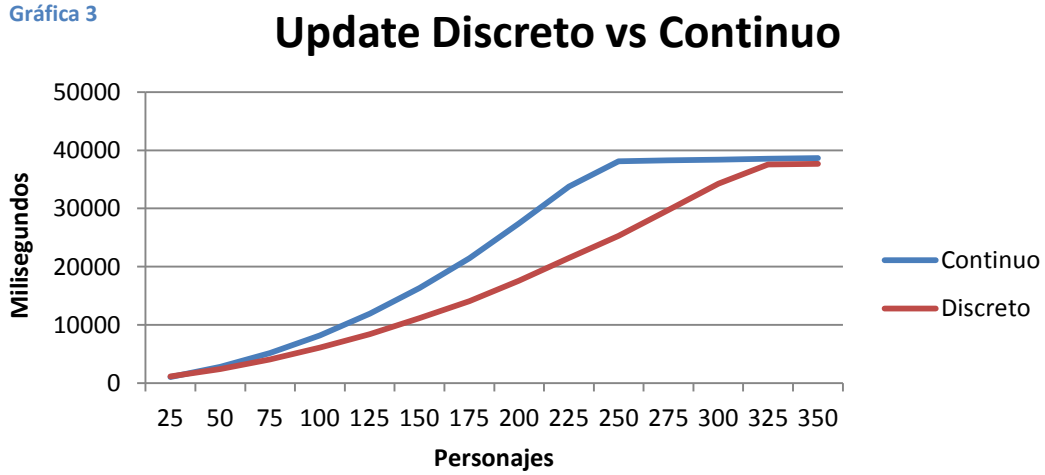
Gráfica 2



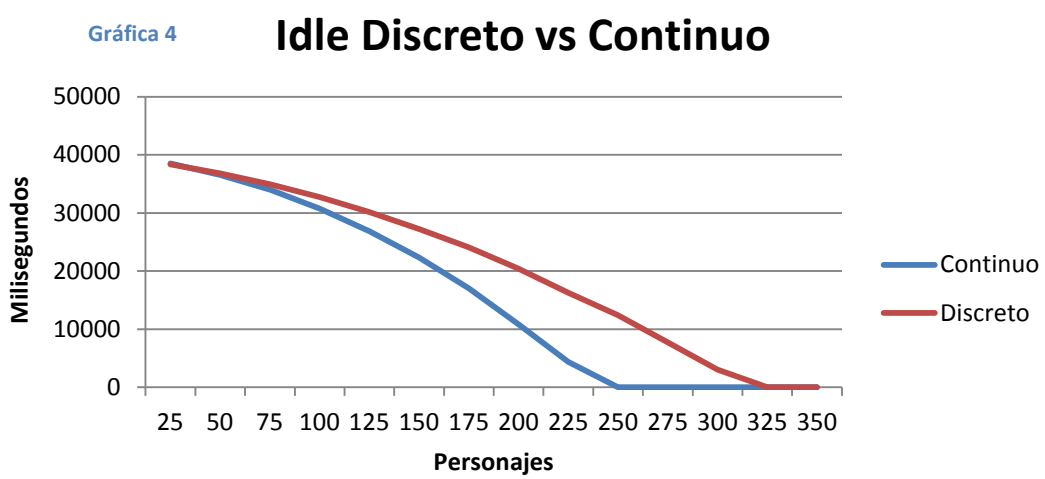
Como podemos observar en las gráficas 1 y 2, el comportamiento del sistema en ambos paradigmas es similar. En ambos sistemas la fase de dibujado se realiza con la misma frecuencia (60 fps). En cuando a la fase de Update, se ejecuta a 60 fps en el paradigma continuo y en el paradigma Discreto cada objeto se ejecuta según su propia frecuencia lo que en ambos caso deja tiempo libre recogido en la fase Idle.

En ambos sistemas al aumentar el número de personajes aumenta progresivamente el coste de la fase de Update, lo que conlleva una reducción del tiempo de Idle hasta que este se agota y el sistema se satura. A partir de este punto se produce una ralentización global del sistema. La fase de dibujado también aumenta ligeramente al aumentar el número de personajes pero el aumento es despreciable comparado con el aumento de la fase de Update.

Gráfica 3



Gráfica 4

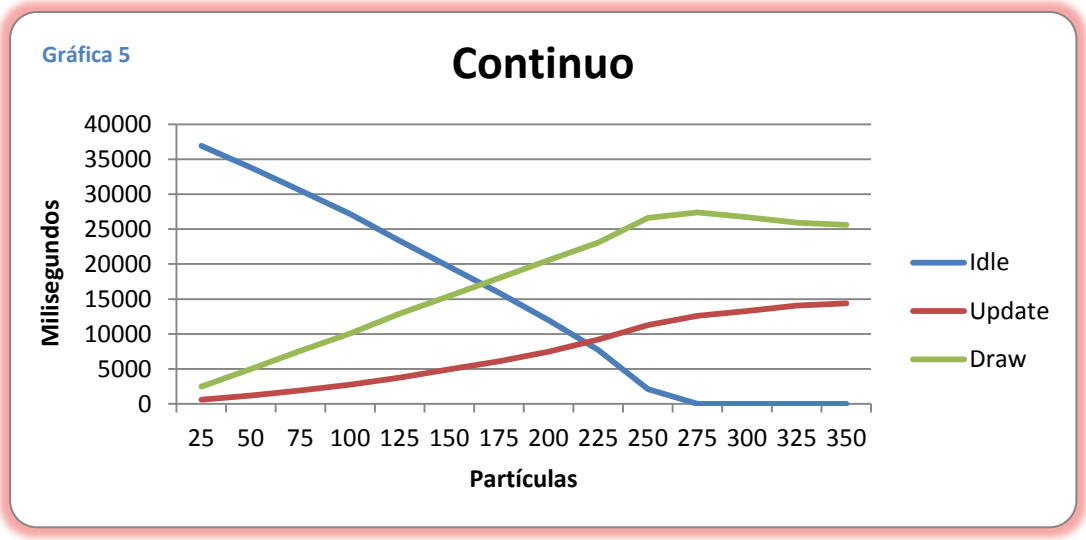


En las gráficas 3 y 4 podemos ver en primer lugar como en el caso discreto el sistema tarda más en saturarse. Mientras que para el modo continuo con 250 el sistema se ha saturado y ralentizado, en el modo discreto el sistema no se satura hasta los 325 personajes. Por otro lado podemos ver se reduce el tiempo de Update utilizando RT-DESK en mejora del tiempo de Idle y que esta reducción es mayor cuanto mayor es el número de personajes. Por ejemplo, para 250 personajes, el tiempo de Idle aumenta 10000 milisegundos.

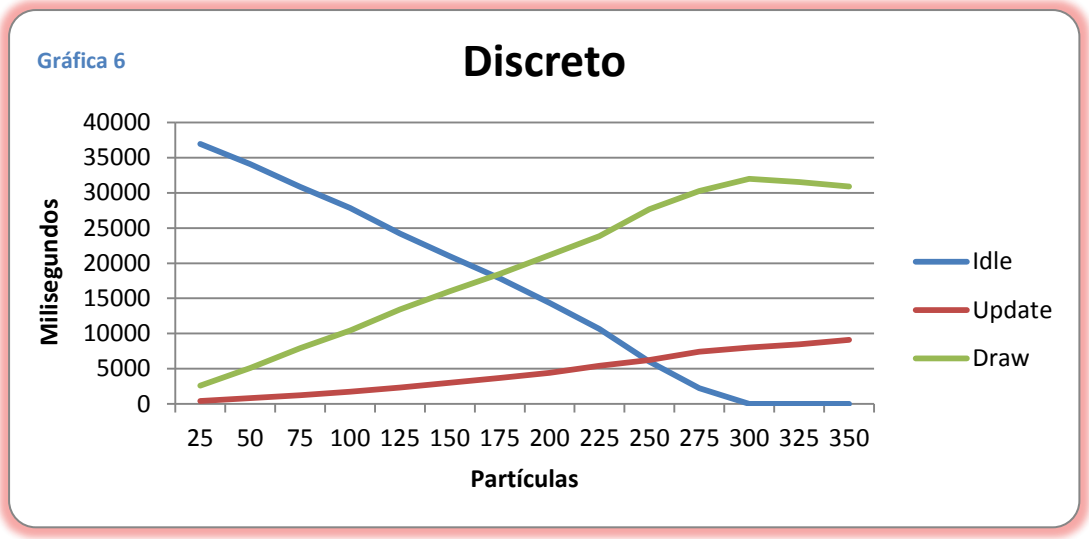
Prueba partículas

En primer lugar vamos a analizar el comportamiento de ambos sistemas. Cabe recordar que el número de partículas que se analiza no es el número total de partículas en escena, si no el número de partículas generadas cada 1/60 ms que se moverán y desaparecerán según su propia configuración.

Gráfica 5

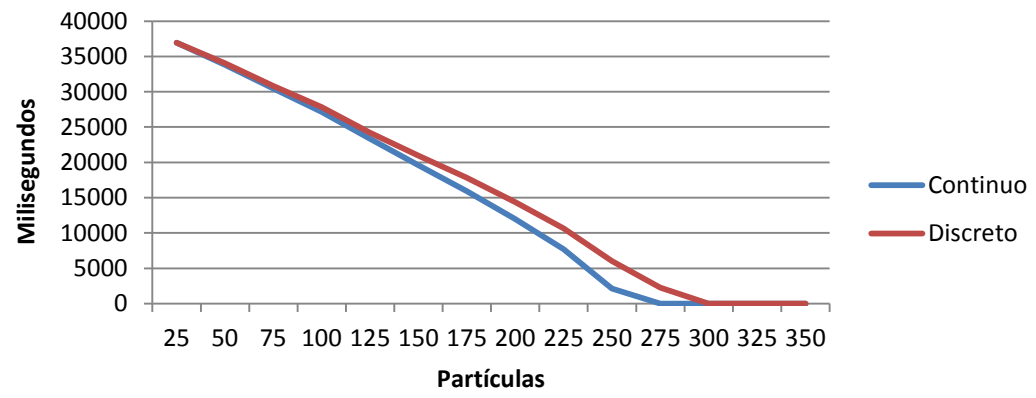


Gráfica 6

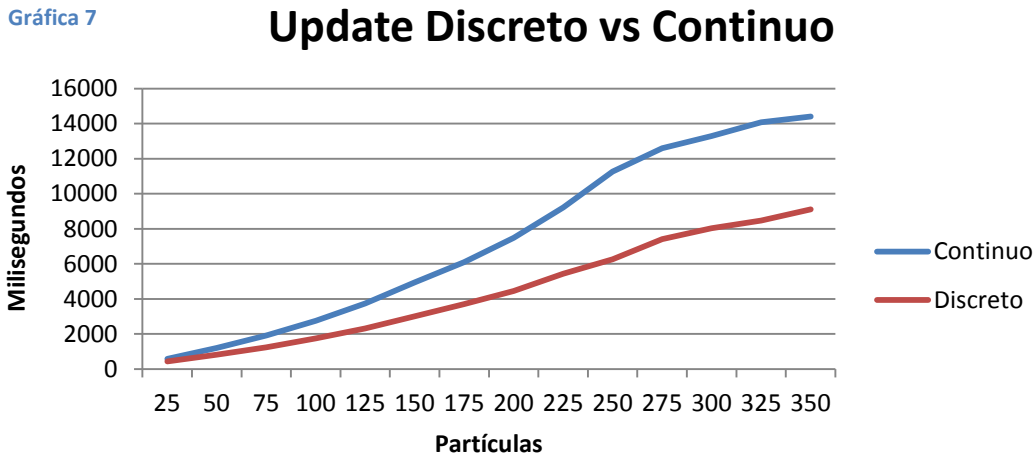


Como podemos ver en las gráficas 5 y 6 el comportamiento de ambos sistemas es similar, al igual que en la prueba anterior. Pero a diferencia de en la prueba anterior, el coste que más aumenta con el número de partículas es el coste de visualización. Esto puede ser debido al bajo coste de actualización de las partículas y a que al aumentar el número de partículas generadas el número total de partículas en escena aumenta exponencialmente. Se observa además una curiosidad, y es que una vez el sistema se ha saturado, el coste de update sigue aumentado en detrimento del coste de visualización.

Gráfica 8



Gráfica 7



En las gráficas 7 y 8 podemos observar como al igual que en la prueba anterior el sistema discreto tarda necesita más partículas para saturarse. Los resultados son muy similares a los obtenidos anteriormente, aunque la mejora es menor, debido al alto coste de la fase de visualización y al bajo coste de la fase de actualización; obteniendo, por ejemplo, para 275 partículas una reducción de 5500 ms aproximadamente en el tiempo de Update.

Conclusiones

A lo largo del proyecto se ha desarrollado desde cero la demo de un videojuego, atravesando todas sus fases. Este desarrollo ha resultado una muy valiosa experiencia para aprender de primera mano cómo se desarrolla un videojuego y una carta de presentación para entrar en la industria. Uno de los aspectos más importantes de la experiencia ha sido la colaboración de principio a fin con un grafista, lo que no solo ha sido de gran ayuda para entender la industria, si no que ha sido muy enriquecedor y ha permitido obtener una calidad en el producto que de otra forma no habría sido posible.

XNA se ha demostrado como una potente herramienta para el desarrollo de videojuegos. Su nivel de abstracción es, en mi opinión, el adecuado para introducirse en el desarrollo de videojuegos. Es código puro y duro, pero con todas las facilidades y utilizando código de alto nivel. En los últimos meses del desarrollo, Microsoft confirmo que retiraba el soporte a XNA, por lo que no es una herramienta que recomendaría aprender a nadie que no la sepa ya. Nos obstante existe una implementación libre de XNA (MonoGame) que permanece en activo y que además permite la distribución en una gran variedad de plataformas como Android, IOS, PC, Linux, MAC, PS Vita, Ouya, etc.

Uno de los principales errores en el videojuego desarrollado es su gran complejidad, sobre todo para un equipo de 2 personas sin experiencia. En la fase de diseño se diseñaron 8 personajes y 8 escenarios, así como 4 ataques para cada personaje. De todo eso, finalmente se ha llegado a implementar 2 personajes con 2 ataques, 2 personajes sin ataques y 2 escenarios. Por lo que gran parte del tiempo invertido en el diseño no ha sido productivo, más aún, cuando el tiempo invertido fue tan grande. La inexperiencia y la ambición me hicieron cometer ese error. Nuestra idea inicial además, era tener un videojuego mucho más avanzado al terminar el proyecto, que con poco tiempo extra pudiéramos publicar y comercializar, por lo que haberse decidido por un videojuego de menor envergadura habría sido una buena idea.

Por otro lado, han quedado patentes lo problemas del paradigma continuo acoplado, siendo el principal de ellos la mala gestión de la potencia de cálculo. Esta mala gestión es fruto de la existencia de una sola frecuencia de ejecución para todo el sistema que conlleva que objetos con bajas frecuencias se sobre-muestren y objetos con altas frecuencias puedan sub-muestrearse, generando además un comportamiento anómalo del sistema. El paradigma discreto desacoplado propuesto es una solución a todos estos problemas.

RT-DESK es una herramienta que nos permite realizar este cambio de paradigma. La gestión de eventos en RT-DESK se modela mediante el paso de mensajes entre los objetos del sistema, lo que nos permite definir tanto comportamientos continuos, que se repiten cada cierto tiempo, como discretos, que se accionan en algún momento determinado.

Los resultados obtenidos demuestran que RT-DESK es capaz de realizar grandes mejoras de rendimientos en las aplicaciones de tiempo real, aunque la cuantía de la mejora dependerá de cada aplicación. Mi experiencia personal es que el cambio de mentalidad que requiere el cambio de paradigma no es sencillo. Más aún cuando se pretende portar una aplicación ya desarrollada según el paradigma continuo acoplado al nuevo paradigma, puesto que resulta una tarea tediosa. No obstante, una vez asimilado el nuevo paradigma, este tiene más potencia y versatilidad que el viejo paradigma, especialmente para aplicaciones desarrolladas desde cero.

Terminos

- **Common Intermediate Language (CIL)**: Es un lenguaje ensamblador orientado a objetos que es ejecutado por la máquina virtual de CLR. Los lenguajes del .NET Framework compilan a CIL.
- **Common Language Runtime** (**CLR**): Es un entorno de ejecución para los códigos de los programas que corren sobre la plataforma Microsoft .NET. El CLR es el encargado de compilar una forma de código intermedio llamada Common Intermediate Language (CIL), al código de maquina nativo, mediante un compilador en tiempo de ejecución. Es una implementación del estándar Common Language Infrastructure (CLI). Los desarrolladores que usan CLR escriben el código fuente en un lenguaje compatible con .NET, como C# o Visual Basic .NET. En tiempo de compilación, un compilador .NET convierte el código a CIL.
- **Common Language Infrastructure (CLI)**: Es una especificación estandarizada que describe un entorno virtual para la ejecución de aplicaciones, cuya principal característica es la de permitir que aplicaciones escritas en distintos lenguajes de alto nivel puedan luego ejecutarse en múltiples plataformas tanto de hardware como de software sin necesidad de reescribir o recompilar su código fuente.

Bibliográfia

[ESA] The Entertainment Software Association – www.theesa.com

[ADESE] Asociación Española de Distribuidores y Editores de Software de Entretenimiento – www.adese.es

[DDV] Diseño de videojuegos. Da forma a tus sueños – Daniel González, RA-MA, 2011.

[SCRUM] http://www.scrum.org

[XP] http://www.extremeprogramming.org/

[SOFT_ENG] Software Engineering For Game Developers – John P. Flynt, PH.D. with Omar Salem

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

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

[S.H] Simulacion Hibrida Como Nucleo de Simulacion de Aplicaciones Graficas en Tiempo Real. Tesis Doctoral de Inmaculada Garcia Garcia. Universidad de Valencia, Servicio de publicaciones, 2004

[CPD] Cambio de paradigma en el control de aplicaciones gráficas en tiempo real. Trabajo fin de máster de Vicente Broseta, 2012.

[DEWITT] http://www.koonsolo.com/news/dewitters-gameloop/

Anexo I Documento diseño

Versión del documento de diseño y fecha de última revisión

1 (30/10/12) por David Pérez

Plataformas

- Xbox 360 mediante Xbox Live Indie Games
- Posible portabilidad a PC

Título

Olympic Punch!

Género

Multijugador, plataformas, lucha

Temática

Mitología griega modernizada

Target

El juego va dirigido especialmente a los grupos de amigos que se reúnen en un mismo sitio para jugar, especialmente a aquellos entre 13 – 30 años.

Descripción general

Juego multijugador competitivo de plataformas, de 2 a 4 jugadores en el que controlarás a un dios griego con aspecto gracioso y modernizado y competirás con el resto de dioses griegos controlados por otros jugadores a los que tendrás que derrotar. Para derrotarlos tendrás que ir recogiendo plegarias que irán apareciendo por el escenario, que es de donde los dioses griegos sacan su fuerza; y que te permitirá realizar ataques que dañarán al resto de jugadores o los fastidiarán de alguna forma. Los jugadores irán corriendo continuamente por el escenario en busca de ser el primero en recoger la plegaria. Las plegarias serán lo suficientemente abundantes como para que los jugadores no se detengan y puedan estar atacándose continuamente. Además, en el escenario también aparecerán objetos que los jugadores podrán utilizar para atacarse y/o fastidiarse y podrán robarse las plegarias unos a otros saltándose encima.

Aspectos clave

- Juego social en el que puedes jugar con tus amigos ya sea desde la misma consola o por internet.
- Las partidas son de duración corta (Unos 5 min).
- Las partidas tiene un ritmo frenético y caótico.
- Puedes fastidiar a tus amigos de mil maneras. El juego debe promover los "piques", y que los jugadores se interpongan en sus objetivos mutuamente continuamente.
- El juego tendrá un ambiente gracioso y divertido. Los personajes serán chibis (personajes desproporcionados con una cabeza muy grande, un cuerpo muy pequeño y con un toque infantil) de los principales dioses griegos con un aspecto actual y modernizado.
- Podrás elegir como personaje entre varios dioses griegos: Zeus, Hades, Ares, Poseidón, Artemisa, Hefesto, Apolo, Afrodita cada uno con habilidades especiales propias y una jugabilidad diferenciada.
- Múltiples modos de juego diferentes con diferentes reglas: Sin plegarias, plegarias infinitas, a vidas, a tiempo, etc.
- Muchos escenarios diferentes con alguna característica propia que le dé un toque diferente a la jugabilidad.
- Posibilidad de jugadores controlados por la IA para poder jugar en solitario o añadir más personajes cuando los jugadores lo consideren.
- Personajes y ataques bloqueados que el jugador irá desbloqueando conforme vaya jugando.

Storyline

Zeus convoca a los dioses del Olimpo en una reunión. Les explica, que en la situación de decadencia que se encuentra su religión, se ha dado cuenta de que la solución y la fórmula para volver a ser aclamados por los mortales, es convertir su religión politeísta en una religión monoteísta, como la religión católica o musulmana. Así, él pretende convertirse en el único dios al que adorar.

El resto de los dioses responden a esta proposición con enfado e indignación, y cada uno de ellos reclama a Zeus el derecho a ser ellos el único dios, pues ellos también tienen poderes y son deidades.

A partir de ahí, se genera una discusión que irá calentándose hasta convertirse en una batalla campal entre los dioses helénicos para conseguir la supremacía sobre la religión, los mortales y el mundo entero.

Referencias

Smash bross brawl (Jugabilidad, Wii)

Mario Wars (Jugabilidad, PC)

Wakfu, Dofus (Estilo gráfico, PC)

playstation allstars battle royale (Jugabilidad, Playstation 3)

Estética

Aspectos generales: El juego tendrá una estética amable y con un toque ligeramente infantil. Los personajes serán chibis (desproporcionados con la cabeza muy grande y aspectos de niños), y los colores serán vivos para conseguir una estética coherente.

Ejemplo de portadas:

Ejemplos de estilo in-game:

Jugabilidad

Modos de juego

Desde el menú, el jugador podrá elegir tres opciones de juego: Arcade, Arcade Hardcore multijugador y online.

- El modo Arcade, es un modo para un solo jugador. En él, el jugador deberá ir derrotando consecutivamente a todos los dioses disponibles al principio del juego controlados por la IA en diferentes escenarios y con dificultad creciente.
- El modo Arcade hardcore es muy similar al modo Arcade pero más largo y de mayor dificultad. El jugador tendrá que enfrentarse a todos los dioses, y no solo a los disponibles al principio del juego. Este modo no estará disponible hasta que el jugador haya desbloqueado todos los personajes.
- El modo multijugador permite enfrentarse hasta a 4 jugadores desde una misma consola. Se podrán elegir varios modos de juego: A vidas, a tiempo, etc. Así como configurarlos: nº de vidas, tiempo, aparición de objetos, etc. Los jugadores podrán formar equipos añadir personajes controlados por la IA y elegir su dificultad. Por último, los jugadores podrán elegir escenario.
- El modo online es totalmente equivalente al modo multijugador pero con opciones para jugar con jugadores en otras consolas mediante internet.

Selección de personaje

Cada uno de los dioses dispondrá de 4 ataques diferentes, aunque solo 3 estarán disponibles al principio. En cada una de las partidas el jugador solo puede utilizar 3 ataques diferentes, por lo que en el momento de elegir el personaje el jugador también deberá elegir sus ataques y asignarlos a los botones disponibles de la forma que prefiera.

Progreso en el juego

Conforme el jugador juegue al juego y vaya cumpliendo ciertos logros irá desbloqueando contenido que no estaba disponible al instalar el juego: escenarios, ataques, personajes. Al finalizar el modo Arcade se desbloqueará el tercer ataque del personaje con que hayamos finalizado.

Combate

Una vez empezada una partida, el jugador podrá:

- Moverse horizontalmente.
- Saltar. Al pulsar el botón de salto el personaje comenzará a saltar, si el botón se mantiene pulsado el salto será más largo. Cuando el personaje esté en el aire, el jugador seguirá pudiendo controlar su movimiento.
- Lanzar los tres ataques que haya elegido, si tiene plegarias suficientes, cada uno asignado a un botón. Los ataques además pueden tener un retraso, un tiempo después de lanzarlos en los que no se podrá volver a lanzar.
- Bloquear. Una vez pulsado el botón de bloquear, durante un periodo de tiempo muy breve de tiempo saldrá un escudo de energía alrededor del personaje. Durante este tiempo los ataques que sufra le causarán menos daño y puede evitar algunos efectos como parálisis o ser lanzado por los aires. Como el escudo dura muy poco tiempo y tiene mucho retraso el jugador tendrá que usarlo en el momento justo.
-

Habilidades

Zeus, dios del rayo

Como dios del rayo, Zeus controla completamente la electricidad y puede utilizarla a su favor para realizar todo tipo de ataques. Zeus es un personaje con una jugabilidad general que cuenta tanto con ataques de un solo objetivo como ataques en área. Además, cuenta con un ataque que le permite aumentar su fuerza a coste de su vida.

- Rayo: Ataque principal de Zeus. Lanza un rayo que avanza desde la posición de Zeus hacia delante hasta que choque con algo o alguien. El rayo se propagará hacia adelante aunque de manera muy rápida. Otra posibilidad es que el rayo no se detenga cuando choque con alguien y que pueda dañar a varios enemigos.
- Carga eléctrica: Zeus acumula energía eléctrica sobre si mismo, cargando su cuerpo de electricidad como una pila y electrificándose por un breve periodo de tiempo. Mientras este electrificado los ataques de Zeus serán más fuertes y los enemigos que toque sufrirán pequeñas descargas que les causarán daños. Como contra partida, mientras este electrificado perderá un poco de vida a cada

instante. Zeus se descargará finalmente tras haber utilizado una determinada cantidad de energía o tras un breve periodo de tiempo. Una buena estrategia para este personaje será guardar bastantes puntos de energía para utilizarlos una vez este cargado.

- Chispazo: Un pequeño ataque a corta distancia. Zeus acumula potencial eléctrico en su mano y toca a un enemigo dándole un fuerte calambrazo. No causa grandes daños pero lanza a su oponente por los aires.
- Electro-explosión: Hace surgir energía eléctrica en todas direcciones que daña a los enemigos y puede dejarlos paralizados un breve periodo de tiempo.

Hades, dios del inframundo

Hades es el único dios que no obtiene su poder de las plegarias de los mortales, si no de las almas de los muertos. Así pues, Hades no puede (recoger plegarias ni) utilizar su poder para atacar. En su lugar, Hades utiliza el poder de las almas. Cuenta con un ataque que le permite obtener parte de las almas de sus enemigos para posteriormente utilizar su poder. Hades contará además con la ayuda de su perro de tres cabezas Cerbero.

- Cosecha de alma: Hades extiende una gran mano fantasmal que puede entrar en el corazón de sus enemigos y arrebatarles un pedazo de su alma. Es el único ataque de Hades que o bien utiliza plegarias, o bien no tiene ningún coste. Este ataque le permite obtener energía a Hades.
- Impacto de alma: Un ataque de medio alcance que lanza un espíritu capaz de impactar en un enemigo.
- Hedor: Hades expulsa el olor de 1000 cadáveres putrefactos en un ataque de área que envenena a los enemigos cercanos. Los enemigos envenenados pierden vida poco a poco. El efecto del veneno desaparecerá con el tiempo.
- Ataque de cerbero: Un ataque controlado de larga distancia. Cerbero aparece ante Hades cargando a toda velocidad contra el enemigo y permanecerá en el juego durante un breve periodo de tiempo. Durante ese periodo no parara de correr en ningún momento. Mientras dura el ataque, el jugador no podrá realizar otros ataques con Hades, si bien si podrá moverse y saltar. A cambio, con sus botones de acción, el jugador podrá controlar a Cerbero. Tendrá un botón para saltar, un botón para atacar y un botón para cambiar la dirección. Si Cerbero llega a una pared, automáticamente cambiará de dirección. Se trata de un ataque muy eficaz y destructivo pero muy difícil de controlar, sobre todo, si el jugador no quiere dejar quieto a Hades y que sea un blanco fácil.

Artemisa, diosa de la caza

Su rol principal es el de arquera. Se trata, pues, de un personaje de ataque a distancia que intentará alejarse del resto de jugadores para dañarles desde lejos y evitar que la dañen. Cuanta además con un par de ataques que le permiten dañar y alejar a los enemigos que se acerquen demasiado.

- Cabezazo: Artemisa desarrolla su cornamenta y carga contra el enemigo causándole daños y lanzándole por el aire hacia atrás. Artemisa cargará durante un breve espacio de tiempo, y se detendrá si impacta con algún personaje o algún elemento del escenario. Si impacta contra algún elemento inmóvil, sufrirá daños ella, y estará confundida durante un breve periodo de tiempo, durante el cual no podrá hacer nada.
- Trampa para osos: Artemisa se agacha y entierra un cepo en el suelo que causará daños a un enemigo cuando pase sobre él. El cepo permanecerá semi-oculto en el suelo, de forma que el resto de jugadores puedan detectarlo y sepan que hay un cepo colocado pero sea muy fácil pasarlo por alto y olvidarse de él. No ocurrirá nada si Artemisa lo pisa. No pueden colocarse dos cepos en el mismo sitio.
- Tiro con arco: Disparo básico con el arco que requiere poca energía y causa pocos daños. Una vez Artemisa lo active, entrará en el modo disparo. En este modo Artemisa no podrá moverse y con el joystick controlará la dirección del disparo. El jugador deberá mantener el botón pulsado mientras Artemisa tensa el arco, y se carga una barra de potencia que aparecerá sobre ella. El daño causado por el disparo dependerá de lo tenso que este el arco, así como la trayectoria de la flecha. Si el jugador no suelta el botón un tiempo después de haberse cargado la barra al máximo. Artemisa se cansará y el disparo se cancela. No se gastará energía si el disparo se ha cancelado. Además el jugador también podrá cancelar el disparo.
- Ataque por especificar. (Halcón, flecha, trampa??)

Poseidón, dios de mar

Como dios del mar que es, se mueve como pez en el agua. Mientras que el resto de personajes bajo el agua no pueden respirar y se mueven más despacio, Poseidón puede bucear con total libertad. Poseidón inundará los escenarios siempre que le sea posible, para desatar todo su potencial y tomar ventaja sobre sus adversarios.

- Inundación: El nivel del mar sube hasta que el agua cubre todo el escenario durante algún tiempo. El nivel del agua irá subiendo lentamente y, cuando se acabe el efecto, bajará poco a poco. Los jugadores que se encuentren sumergidos saltarán y se moverán con normalidad pero de forma más lenta. También podrán atacar con normalidad. Poseidón cambiará y dejará de andar para ponerse a bucear libremente. Para bucear el jugador se moverá a los lados con el joystick y usará el botón A para subir. Si no pulsa el botón A se irá hundiendo lentamente. El resto de jugadores sufrirán algún daño cada cierto tiempo mientras estén bajo el agua.

- Estocada de tridente: Un golpe a corta distancia que causa daños y lanza a los enemigos por el aire hacia atrás si les impacta.
- Lanzamiento de tiburón: Poseidón lanza un tiburón blanco hambriento que nadará en hacia adelante a toda velocidad hasta que impacte con algo o alguien. Si impacta contra un jugador este sufrirá daños. Este ataque solo puede utilizarse bajo el agua.
- Medusa gigante: Solo puede lanzarse bajo el agua. Poseidón atrapa a un enemigo que este enfrente suyo con una gran medusa. Mientras haya un enemigo atrapado en la medusa ni el enemigo ni Poseidón se moverán. El enemigo atrapado perderá vida y energía cada cierto tiempo mientras este atrapado. El enemigo atrapado deberá pulsar repetidamente un botón para intentar escaparse lo antes posible mientras que Poseidón deberá pulsar repetidamente un botón para evitar que el enemigo se escape. El tiempo que permanecerá atrapado dependerá del número de pulsaciones de cada jugador.

Hefesto, dios de la forja

Hefesto no tiene ataques que dañen directamente a sus contrincantes. Como dios de la forja, Hefesto puede forjarse poderosas armaduras que le darán grandes beneficios para enfrentarse a sus enemigos. Hay tres tipos de equipo que Hefesto es capaz de forjarse: Botas, cascos y corazas. Una vez que Hefesto se haya forjado un equipo, lo llevará puesto el resto de la partida. Además, una vez Hefesto lleve puesto unas botas, un casco y una coraza podrá utilizar una habilidad especial: Metal Suit, que lo convierte en una armadura completa. Cuando Hefesto comience a forjarse un equipo, deberá pulsar correctamente una secuencia de botones para que la forja se realice con éxito. Si pulsa algún botón incorrectamente la forja se cancelará y tendrá que volver a empezar, pero no gastará la energia. La forja también se cancelará si Hefesto recibe algún ataque directo, pero no si le saltan encima.

- Forja de casco de hierro: Hefesto se forja un casco que hará que los enemigos que le salten encima pierdan ellos energía en vez de perderla Hefesto.
- Forja botas de hierro: Forja unas pesadas botas de hierro. Con ellas Hefesto causará daños a los enemigos sobre los que salte, además de quitarles energía normalmente.
- Forja coraza de hierro: Con ella Hefesto verá reducidos los daños que sufra en un %.
- Forja botas incandescentes: Unas botas puestas al rojo vivo, que prenden todo lo que tocan. Mientras Hefesto vaya caminando, el suelo que pise arderá durante un breve periodo de tiempo, de forma que podrá hacer que haya una pequeña parte del escenario en llamas. Los enemigos que entren en contacto con el fuego sufrirán daños.
- Metal suit: Una habilidad especial que solo puede ser activada cuando Hefesto lleve un casco, unas botas y una coraza. Hefesto se introduce en una gran armadura completa de hierro de aspecto temible y mejora los efectos de reducción de daños, daños causados al saltar encima y perdida de energía de los jugadores que salten sobré él. Además, la armadura consta con unos lanzallamas en los puños, que podrá gastar sin coste alguno de energía para causar daños a los enemigos. Una vez activado el Metal suit, esté necesitará un mantenimiento de energía cada cierto tiempo. Si a Hefesto se

le agota la energía, el Metal suit se romperá, y perderá todas las partes de equipo que llevaba puestas, teniendo que volver a empezar desde el principio.

Ares, dios de la guerra

Ares representa la violencia, la fuerza bruta, la agresividad y la crueldad de las batallas. Como fanático de las peleas, Ares disfruta con la sangre y puede luchar causando grandes daños sin preocuparse por su propia salud. Se trata de un soldado moderno que lucha utilizando armas de fuego: bazooka, escopeta, etc.

- Disparo de bazooka: Dispara un bazookazo que causa grandes daños en un área. Una vez lo active, Ares entrará en el modo disparo. En este modo no podrá moverse y con el joystick controlará la dirección del disparo. Será necesario pulsar el botón una segunda vez para realizar el disparo. Si se dispara demasiado cerca Ares también sufrirá daños.
- Disparo de escopeta: Un ataque a corta distancia. La primera vez que se active, Ares se equipará con la escopeta en las manos, pero no realizará ningún disparo. Mientras la escopeta esté equipada contará con varios disparos, que se irán gastando al volver a pulsar el botón de la escopeta. Si Ares realiza otro ataque diferente mientras la escopeta esté equipada perderá los disparos que le queden. Con los disparos de la escopeta no se puede apuntar, irán en línea recta en la dirección en la que este yendo Ares, no obstante; Ares no tendrá que detenerse para disparar, y podrá disparar incluso mientras salta. Los disparos de la escopeta abarcan una pequeña área, y tienen un alcance limitado. Si no han impactado con nada dentro de su alcance, será como si hubieran impactado al final de su alcance.
- Frenesí: Ares está completamente inmerso y motivado con la batalla y entra en un estado de máxima violencia y agresividad. Ares entrará en modo "bersek" durante un tiempo. Mientras este en modo "bersek" los ataques de Ares realizan el doble de daño, y los ataques que le impacten le causarán el doble de daño.
- Metralleta: La metralleta realiza un gran número de disparos de muy baja potencia de forma continua. Con el Joystick derecho el jugador podrá dirigir el ataque mientras este dure. Además, Ares podrá moverse con total libertad mientras dure el ataque. Muy útil para atacar a los enemigos escurridizos mientras se los persigue o para atacar a varios enemigos.

Afrodita, diosa del amor

Afrodita puede influir en el corazón de los dioses para controlar sus sentimientos en su beneficio. Afrodita dispone de un par de ataques para enamorar a sus enemigos y causarles curiosos efectos que durarán un tiempo. Además, Afrodita puede utilizar el amor no correspondido para causar daños a los enemigos previamente enamorados y puede utilizar el poder del amor para curarse.

- Amor loco: Es un ataque lineal que puede impactar a un solo enemigo. El enemigo afectado pasará a estar enamorado durante algún tiempo y le será muy difícil pensar en otra cosa que no sea en su amado/a. Mientras dure el efecto el jugador afectado verá sus controles invertidos: Izquierda-derecha y arriba-abajo.
- Amor añorado: Un ataque lineal que puede afectar a un solo enemigo. El objetivo se enamora perdidamente de Afrodita, y no puede estar sin ella. El objetivo sufrirá daños cada segundo si está a mayor distancia de Afrodita de una distancia determinada.
- Rompecorazones: Un ataque lineal que impacta en un solo objetivo. Solo puede afectar a enemigos que estén bajo el efecto de amor loco o amor añorado. El personaje se da cuenta de que su amor el imposible y que nunca será correspondido, sufriendo grandes daños. Termina el estado enamorado.
- El poder del amor: Afrodita utiliza el poder del amor para curarse una pequeña cantidad de vida.

Apolo, dios de la luz

Apolo domina la luz en todas sus formas y colores y puede utilizar su versión más peligrosa, el láser para dañar a sus enemigos. Apolo puede utilizar ataques que utilizar luz de diferentes colores: rojo, verde y azul. Una vez utilizado un ataque de un color, Apolo retendrá parte de la luz utilizada, pasando a un estado permanente en el que emitirá luz del color utilizado, cambiando su color. Las reglas de los colores de Apolo son las siguientes:

- Si Apolo no emite ningún color y utiliza un ataque de un color determinado, Apolo emitirá ese color: Rojo, verde o azul. Si su siguiente ataque es del mismo color, será más fuerte.
- Si Apolo emite un color primario (rgb) y utiliza un ataque, su color pasará a la mezcla de los colores utilizados hasta el momento (cian, verde y morado). Si su siguiente ataque es de cualquiera de los colores que componen su color, este será más fuerte.
- Si Apolo emite un color no primario y utiliza un ataque del color que le falta para completar el trio r-g-b, Apolo habrá absorbido todos los colores y pasará a un estado especial, en el que su color irá variando de forma intermitente y su próximo ataque de cualquier color será más fuerte y tendrá una animación diferente, en la que se alternarán de forma intermitente todos los colores. Tras este ataque Apolo volverá al estado inicial, donde no emite ningún color.

- Si Apolo emite un color no primario, y utiliza un ataque de alguno de los colores que ya pose, pasará a emitir el color de su último ataque.
- De esta forma Apolo tendrá diferentes estrategias según la forma en la que combine los colores de sus ataques.
- Descartar uno:
- Rayo láser rojo: Apolo lanza un rayo láser desde sus manos de gran potencia que tendrá toda la longitud posible desde él hasta impactar con algún objeto y lo mantendrá activo por un breve periodo de tiempo. Mientras el láser este activo, Apolo no podrá moverse, pero con el Joystick podrá dirigir el láser. El láser atravesará a los enemigos, lo que le permitirá dañar a más de un enemigo si están alineados. Los enemigos sufrirán daño continuamente mientras estén en contacto con el láser.
- Danza del láser verde: Apolo utiliza el láser para crearse una espada en cada mano y empieza a girar en forma de torbellino. Mientras este girando el jugador podrá dirigirlo con el joystick, y tendrá que pulsar repetidamente el botón A para prolongar la duración del ataque.
- Descomposición en luz azul: Apolo se desmaterializa en luz azul para desplazarse brevemente hacia delante a la velocidad de la luz, tras lo cual, vuelve a materializarse. Los enemigos que estén en su trayectoria sufrirán daños. Se trata de un ataque a corta distancia, que además le permite a Apolo desplazarse muy rápidamente.
- Espada de luz verde: Un ataque muy potente de corto alcance: Un tajo utilizando una espada de luz verde que crea Apolo.
- Disparo láser Azul. Un único disparo de luz que viaja en línea recta.

Puntuaciones

Diseño del sistema de puntuaciones

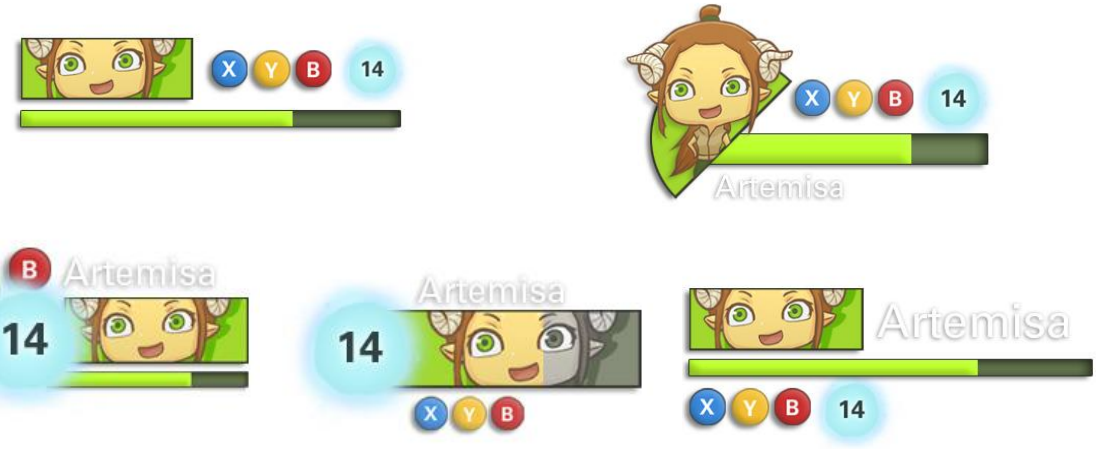
Modo on-line

El videojuego debe enviar toda la información necesaria por la red para poder jugar con cualquier usuario de Xbox live Indie como si se estubiera jugando desde la misma videoconsola.

Controles

HUD

La interfaz mostrará para cada jugador una imagen del personaje, un icono con cada uno de los ataques seleccionados y asignados a los botones, y su barra de vida. Estos iconos se colocarán en la parte superior de la pantalla.



Personajes

Zeus

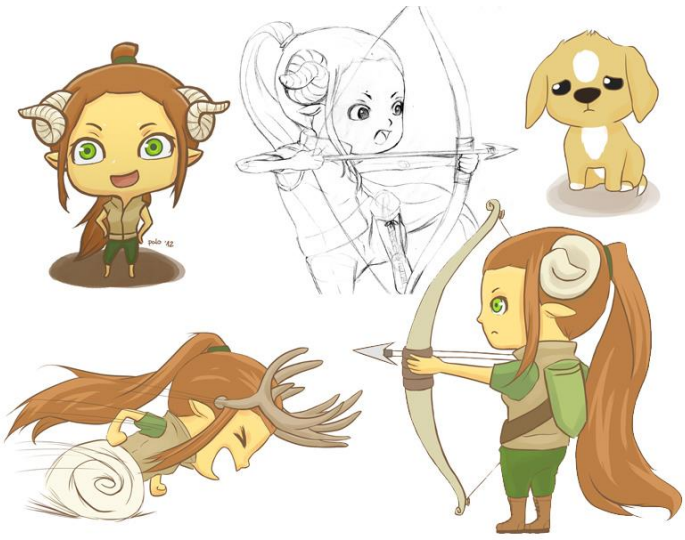
Soberano de hombres y dioses, rey del Olimpo, dios de la luz, del cielo sereno y el rayo. Padre de todos, Preside las manifestaciones celestes (lluvia, rayo, relámpago, trueno...), mantiene el orden y la justicia en el mundo, no se deja dominar por sus caprichos (A excepción de los amorosos). Como dios del rayo, Zeus controla completamente la electricidad y puede utilizarla a su favor para realizar todo tipo de ataques.

Hades

Dios del inframundo. Sus dos hermanos eran Zeus (el menor de todos) y Poseidón. Hades es el único dios que no obtiene su poder de las plegarias de los mortales, si no de las almas de los muertos. Así pues, Hades no puede utilizar su poder para atacar, en su lugar, Hades utiliza el poder de las almas.

Artemisa

Señora de los Animales. Hermana melliza de Apolo. Fue la diosa helena de la caza, los animales salvajes... A menudo se la representaba como una cazadora llevando un arco y flechas. El ciervo y el ciprés les estaban consagrados.
Su rol principal es el de arquera. Se trata, pues, de un personaje de ataque a distancia que intentará alejarse del resto de jugadores para dañarles desde lejos y evitar que la dañen.



Ares

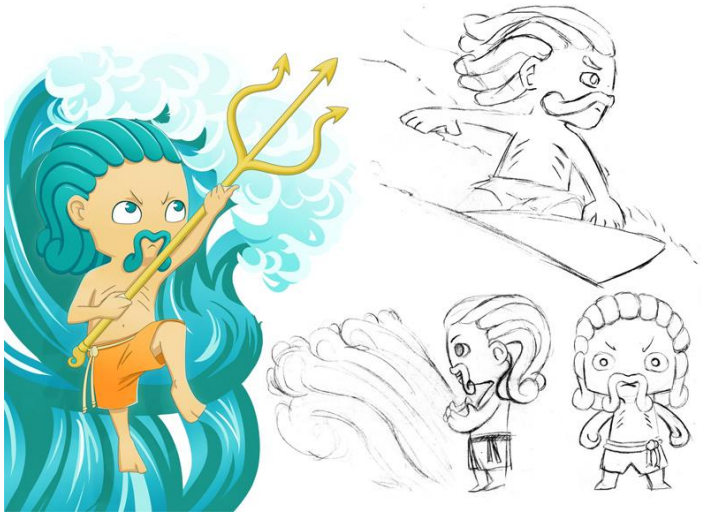
Dios de la guerra, representa la violencia, la fuerza bruta, la agresividad y la crueldad de las batallas. Como fanático de las peleas, Ares disfruta con la sangre y puede luchar causando grandes daños sin preocuparse por su propia salud. Se trata de un soldado moderno que lucha utilizando armas de fuego: bazooka, escopeta, etc.

Hefesto

Dios del fuego y la forja, así como de los herreros, los metales y la metalurgia. Hefesto era bastante feo, lisiado y cojo. Incluso se dice que, al nacer, Hera lo vio tan feo que lo tiró del Olimpo. No tiene ataques que dañen directamente a sus contrincantes. Como dios de la forja, Hefesto puede forjarse poderosas armaduras que le darán grandes beneficios para enfrentarse a sus enemigos. Hay tres tipos de equipo que Hefesto es capaz de forjarse: Botas, cascos y corazas.

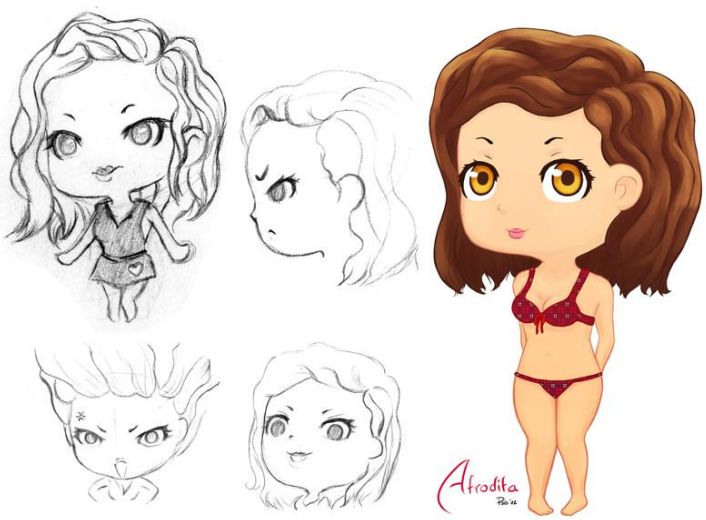
Poseidón

Como dios del mar que es, se mueve como pez en el agua. Mientras que el resto de personajes bajo el agua no pueden respirar y se mueven más despacio, Poseidón puede bucear con total libertad. Poseidón inundará los escenarios siempre que le sea posible, para desatar todo su potencial y tomar ventaja sobre sus adversarios.



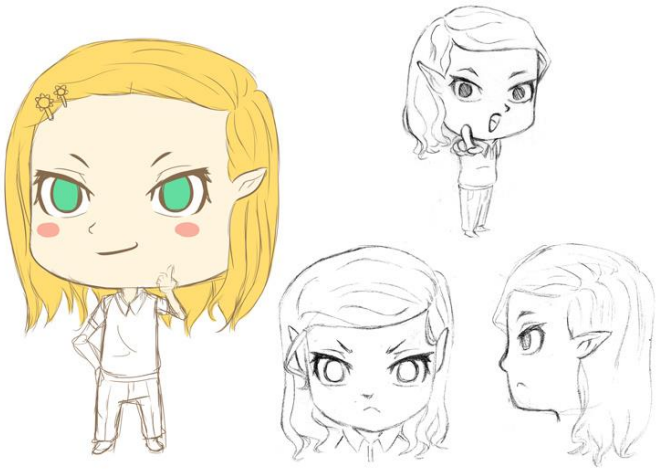
Afrodita

Diosa del amor, la lujuria, la belleza, la sexualidad y la reproducción. Afrodita puede influir en el corazón de los dioses para controlar sus sentimientos en su beneficio. Afrodita dispone de un par de ataques para enamorar a sus enemigos y causarles curiosos efectos que durarán un tiempo. Además, Afrodita puede utilizar el amor no correspondido para causar daños a los enemigos previamente enamorados y puede utilizar el poder del amor para curarse.



Apolo

Uno de los más importantes dioses olímpicos de la mitología griega. Dios de la luz y el sol. Hermes creó la lira para él, y el instrumento se convirtió en un atributo común de Apolo.



Items

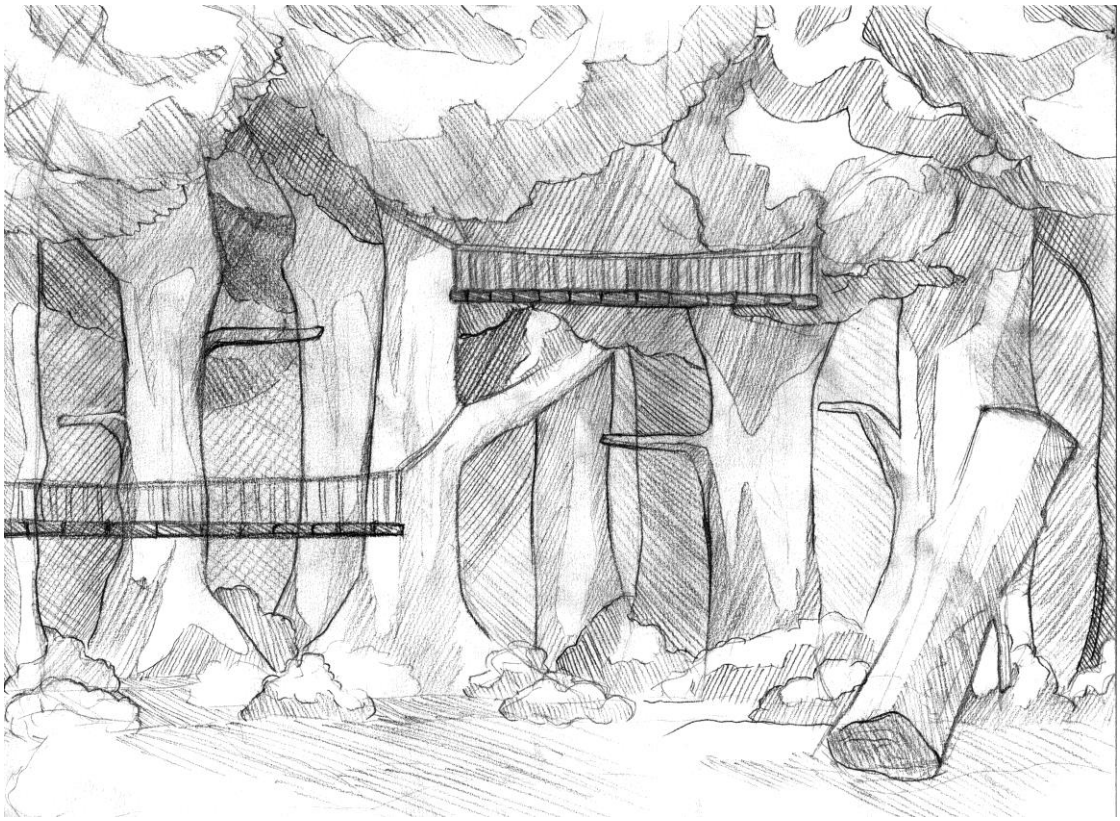
Las plegarias

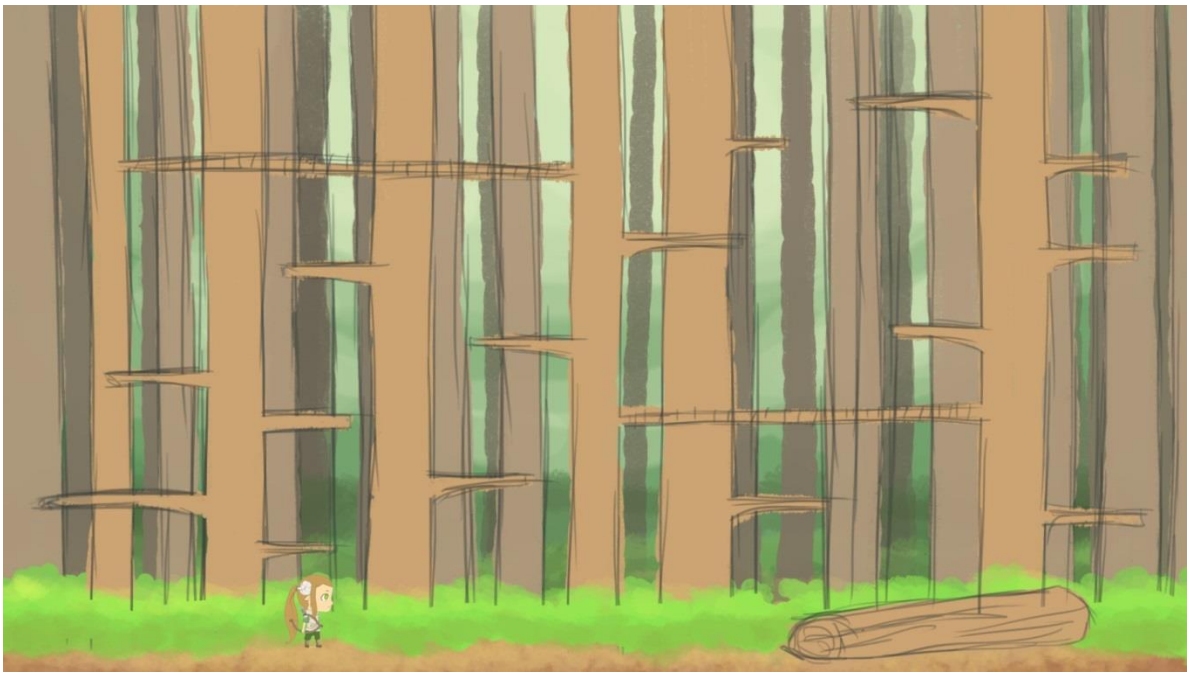
Niveles

Un total de 8 escenarios, uno por cada uno de los dioses, siendo cada escenario algo característico del dios asociado.

Bosque (Artemisa):

Un bosque de árboles muy grandes con grandes troncos. Los jugadores podrán ir saltando entre las diferentes ramas de los árboles y en los puentes de madera que habrá entre algunos árboles. De vez en cuando los árboles dejarán caer fruta con la que los jugadores podrán recuperar una pequeña parte de la vida.





Tártaro (Hades):

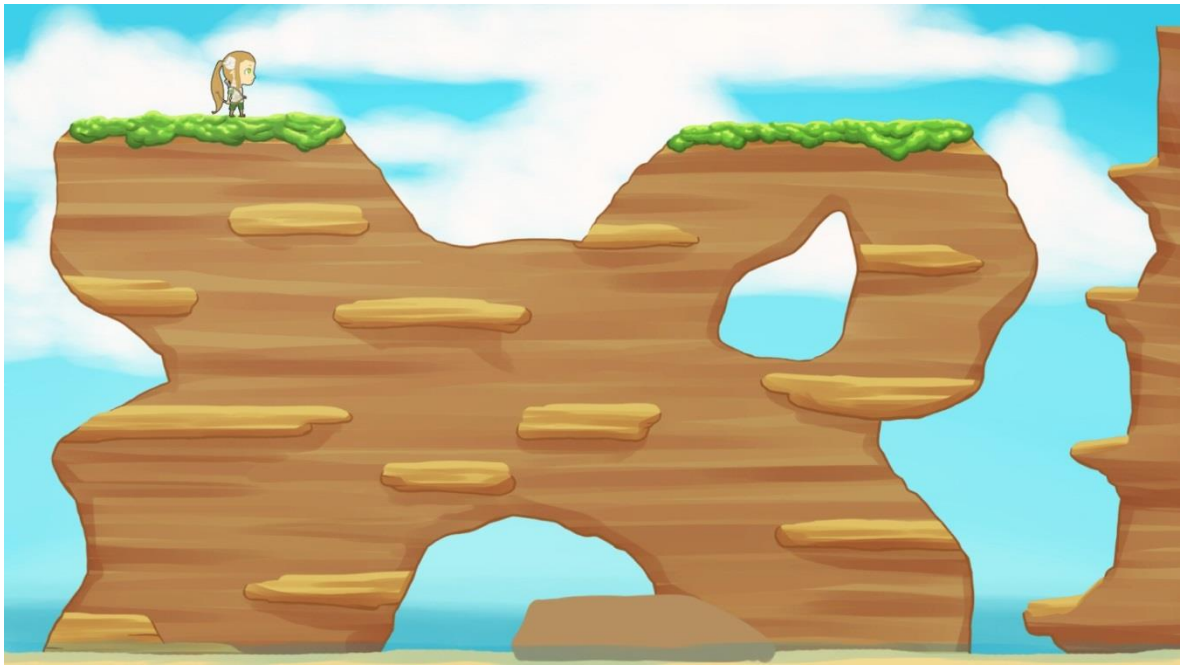
Capa más profunda del inframundo, una gran prisión fortificada rodeada por un río de fuego llamado Flegetonte. Es el lugar donde van las almas de los condenados y sufren un castigo se adecuado a su crimen. Por ejemplo Sísifo, que era un ladrón y un asesino, fue condenado a empujar eternamente una roca cuesta arriba sólo para verla caer por su propio peso. Tántalo, que disfrutaba de la confianza de los dioses conversando y cenando con ellos, compartió la comida y los secretos de los dioses con sus amigos. Su justo castigo fue ser sumergido hasta el cuello en agua fría, que desaparecía cada vez que intentaba saciar su sed, con suculentas uvas sobre él que subían fuera de su alcance cuando intentaba agarrarlas.

El escenario cuenta con unos desagües de lava situados en la parte superior, que de forma aleatoria y alternada tras un pequeño aviso verterán lava sobre el escenario. Los jugadores tendrán que estar pendientes de cuando sucede esto para esquivarla, ya que si les da les causará daño.

Peñón en la playa (Poseidón):

Un gran peñón sobre la playa, al lado de un acantilado.

De forma aleatoria, tras un aviso, puede haber subidas de marea inundándose el escenario hasta distintos niveles. En las zonas inundadas los personajes se moverán más despacio, excepto Poseidón.

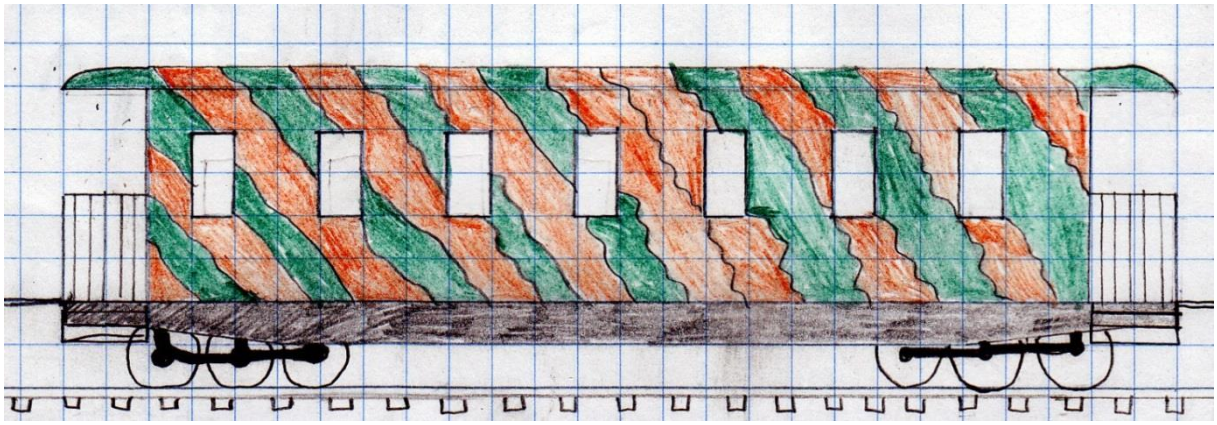


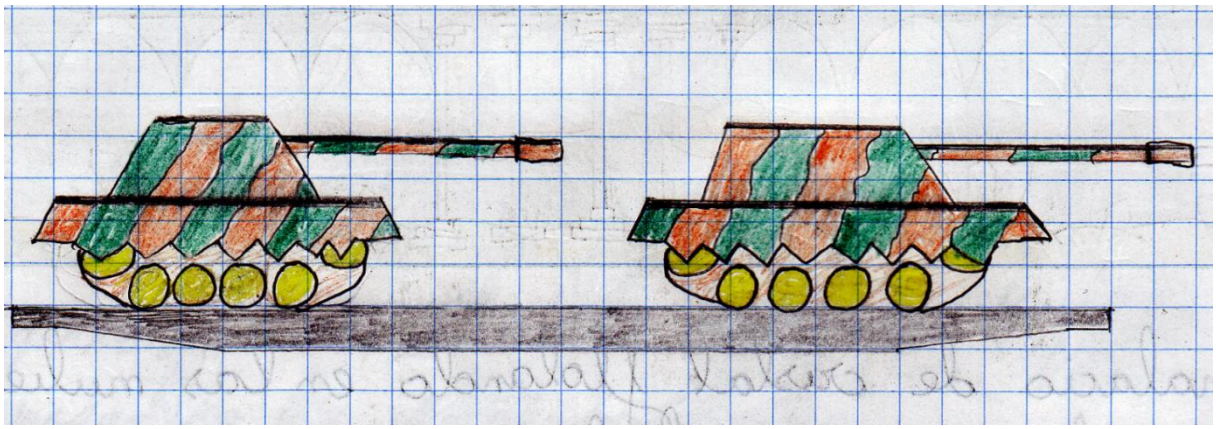
Tren militar (Ares):

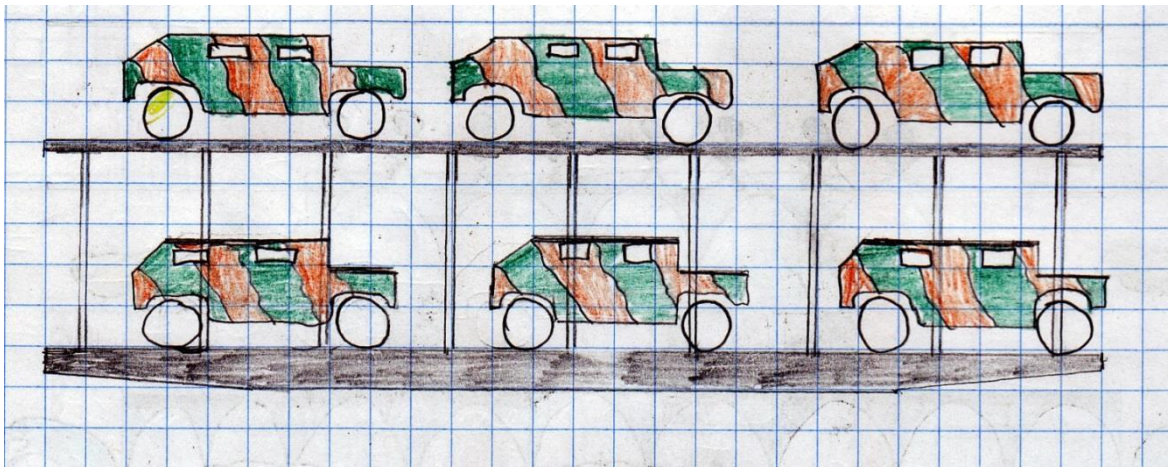
Tren militar de transporte de tropas y vehículos en marcha en el que los jugadores irán por encima.

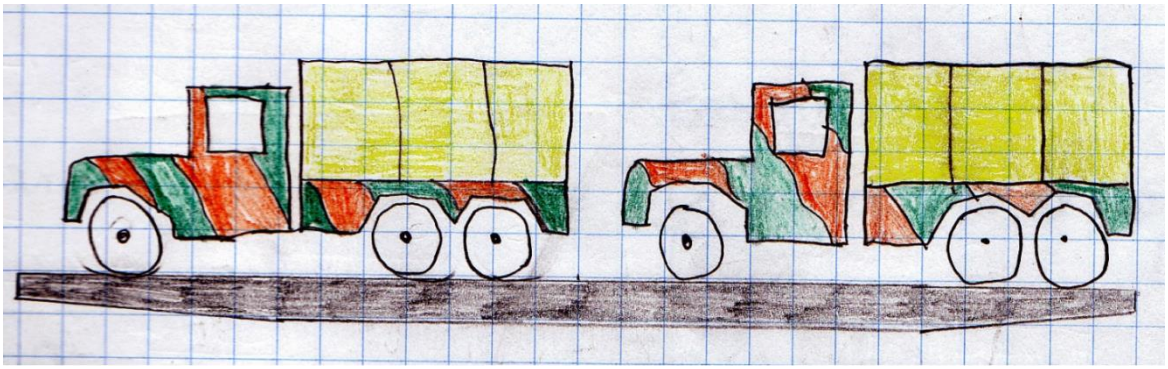
La pantalla muestra en todo momento exclusivamente un vagón/vagón y medio y se irá desplazando hacia delante sobre el tren de forma continua por lo que los personajes tendrán que ir avanzando. Se trata de una pantalla de scroll horizontal. El tren es infinito.

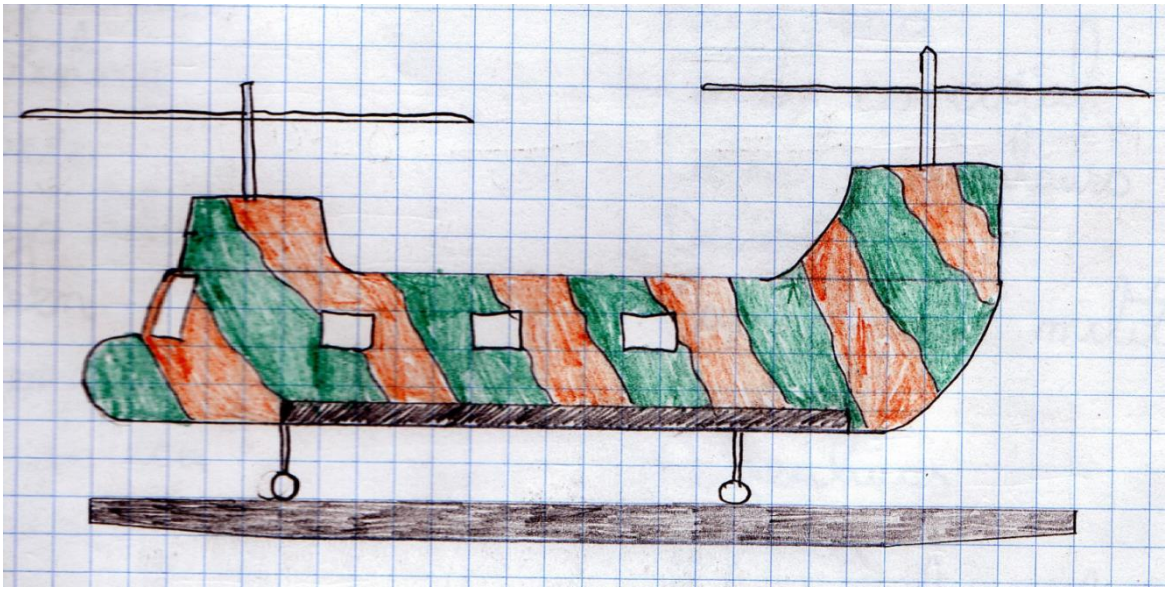
El tren cuenta con al menos 4 tipos diferentes de vagones: Vagón de vehículos ligeros, vagón de vehículos pesados, vagón de pasajeros y vagón de helicóptero. Los vagones se irán colocando para formar el tren conforme la pantalla vaya avanzando de forma pseudo-aleatoria. Además, en algunos vagones, algunos elementos de colocarán de forma aleatoria en el momento de generar el vagón. Por ejemplo, en los vagones de vehículos se cuenta con varios tipos de vehículos que pueden ser colocados en varias posiciones y con varias orientaciones.









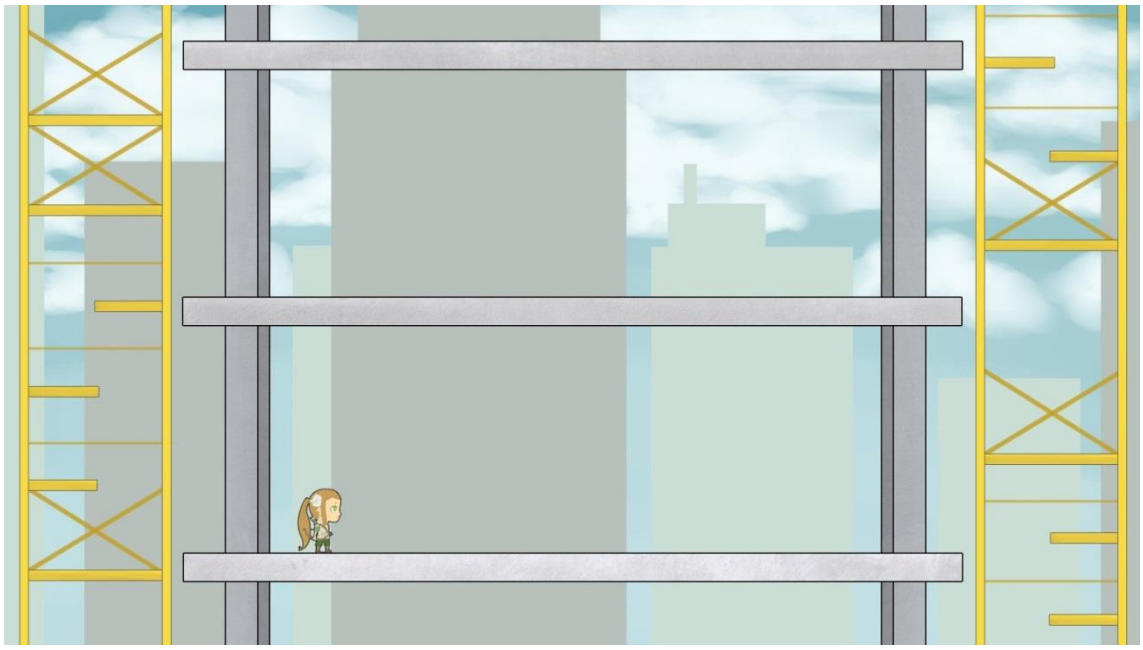


Rascacielos en obras (Hefesto):

Un rascacielos en obras rodeado de andamios por el que los jugadores tendrán que estar continuamente subiendo.

La pantalla muestra en todo momento tres plantas del edificio y se irá desplazando hacia arriba de forma continuada, independiente del movimiento de los jugadores, obligando a estos a ir subiendo. Si los jugadores cruzan el limite inferior de la pantalla se considerará que se han caído y perderán vida, volviendo a aparecer en un posición más elevada. Se trata de una pantalla de scroll vertical. El edificio es infinito. Además, de forma aleatoria, tras un aviso algunas plantas se desplomarán, haciendo que el edificio baje de forma rápida y aplastando a los jugadores que hayan permanecido en dicha planta. Los jugadores tendrán que usar los andamios para subir entre plantas, puesto que no es posible subir directamente de una planta a otra.

Los andamios de los laterales serán generados de forma aleatoria mientras la cámara avanza. Hay 4 tipos de niveles en el andamio: plataforma completa, media plataforma a la derecha, media plataforma a la izquierda y ninguna plataforma. Las plantas contarán con diferente material de obra, como por ejemplo ladrillos, tablones y sacos que se colocará en las plantas de forma aleatoria.



Centro Gangnam (Afrodita):

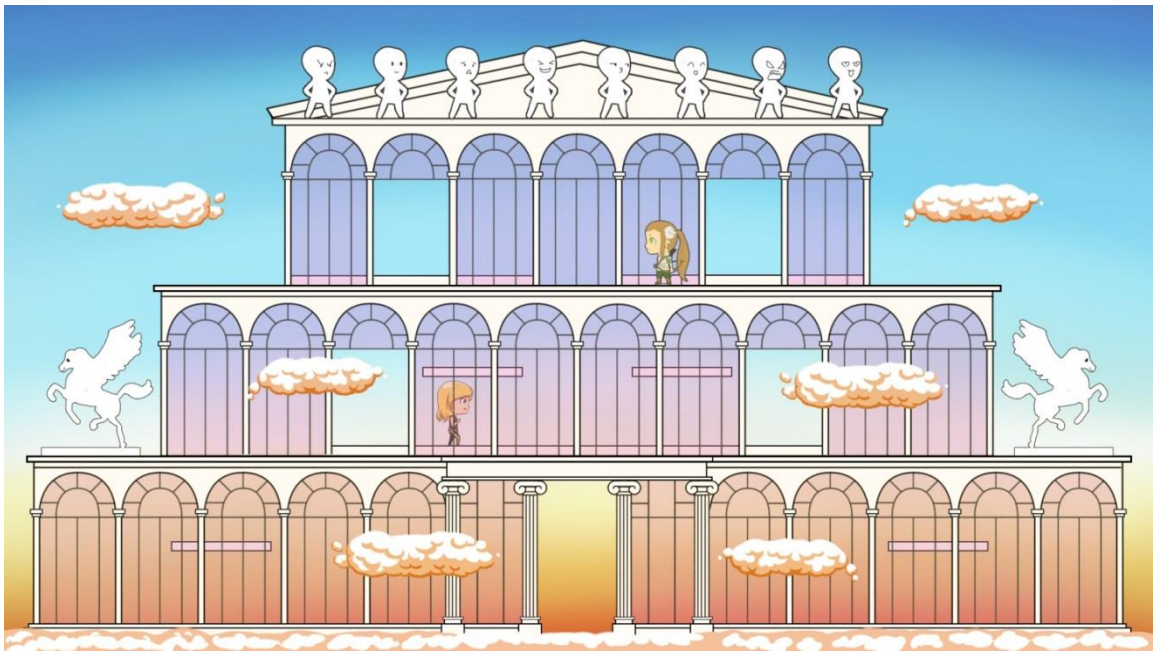
El centro comercial más chic y con más estilo de la ciudad. El centro cuenta con 3 plantas: moda, complementos y joyería; comunicadas entre sí por un par de ascensores en la zona central que estarán en continuo movimiento.

Olimpo (Zeus):

Un gran palacio de cristal rodeado de nubes situado en la cima del monte olimpo, el hogar de los dioses.

En este escenario es posible estar en dos niveles de profundidad distintos: fuera del palacio y dentro del palacio. Los jugadores pueden pasar de un nivel a otro en las puertas del palacio, para ello deberán situarse en la puerta y pulsar arriba o abajo respectivamente.

En la parte exterior hay nubes sobre las que es posible saltar y que se moverán continuamente con el viento. Aparecerán por un lado de la pantalla generadas aleatoriamente y desaparecerán por el otro, cambiando la velocidad de movimiento y su sentido con los cambios de viento que se realizarán de forma también aleatoria.

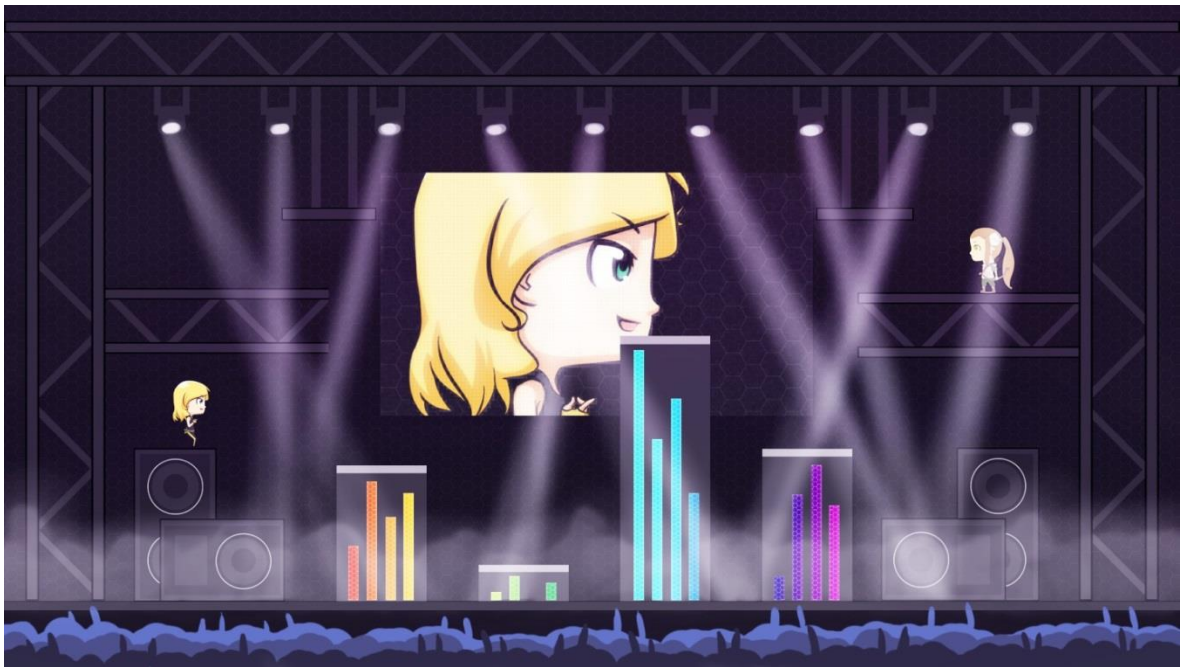


Concierto pop (Apolo):

Un glamuroso escenario de un multitudinario concierto Pop donde Apolo es la estrella.

La jugabilidad del escenario la darán una serie de plataformas decoradas con efectos de ecualizadores que subirán y bajarán continuamente. Además, el escenario contará con una serie de efectos y animaciones que darán todo un espectáculo:

- Efecto de humo en la parte inferior del escenario que se moverá, aparecerá y desaparecerá.
- Efecto de luces causado por los focos del escenario que se moverán de forma aleatoria.
- Animación del público vitoreando el espectáculo.
- Una gran pantalla que mostrará imágenes de los participantes así como algunas otras animaciones.



Miscelánea (realmente ***Lista De Cosas Que Me Importan Una Mierda™***)

El diseño del juego es muy completo y abarca un juego considerablemente grande para la experiencia del equipo de desarrollo. Si hay problemas de tiempo se puede suprimir en nº de personajes y escenarios, así como eliminar el modo online o la posibilidad de personajes controlados por la IA.

Memorabilia

Es un juego pensado para poder jugar con 4 jugadores en cualquier momento, aunque se disponga de poco tiempo. El hecho de poder divertirse en compañía, junto con la estética simpática y desenfadada del juego y sus toques de humor, lo hacen un juego agradable y adecuado para las reuniones en casa con los amigos, en un amplio rango de edades.

Este tipo de juegos de partidas cortas para muchos jugadores si bien son muy comunes en Wii, el multijugador de Xbox esta más enfocado a los juegos de larga duración y con partidas online. Además los juegos multijugador de la plataforma que se juegan desde una misma consola suelen ser para dos jugadores, por lo que pensamos que estamos enfocando el juego en una dirección muy poco explotada en la plataforma.

Por otra parte, el juego presenta una jugabilidad, relativamente similar a Super Smash Bross y Playstation all stars, juegos de gran éxito en Nintendo y Playstation respectivamente, pero sin nada similar en Xbox, por lo que podemos aprovecharnos de esta cadencia en la plataforma.

Por último, al poder jugarse tanto desde una misma consola como online es posible que las personas que primero jueguen juntas luego quieran jugar online, por lo que el juego puede tener una mayor difusión.

Problemática

Problemas internos del juego (complicaciones para desarrollarlo):

- Nula experiencia de los miembros del equipo en el desarrollo de videojuegos.
- Al no contar con compositor y música propia el proyecto tendrá que adaptarse a la música libre que podamos encontrar.

Problemas externos al juego externos (complicaciones para venderlo):

- La plataforma Xbox Live Indie Games es una plataforma poco promocionada por Microsoft y desconocida para una gran parte de los propietarios de una Xbox 360, por lo que suele ser una plataforma con bajas ventas.

Tiempo de juego

Se trata de un videojuego de partidas cortas multijugador, por lo que no hay un tiempo de juego establecido. La duración de los modos arcade es bastante reducida: se estima entre 30’ – 1h.