
Trabajo de Fin de Grado

GRADO DE INGENIERÍA INFORMÁTICA

Facultad de Matemáticas
Universidad de Barcelona

# Motor Gráfico para Juegos

Carlos Ruiz Gonzalez

Director: Oriol Pujol Vila
Realizado en: Departamento de Matemáticas y Análisis de la UB
Barcelona, 29 de Diciembre de 2014

This project is about the construction of a game engine, the game engine's editor and a game. The game engine and editor are made with an *Entity Component System* architecture and with the technologies *WinRt*, *DirectX* and *WIC*, *XAML* was used to do the editor's *GUI*. The programming language used was *C++*, the *IDE* was *Visual Studio Express 2013* and the target *OS* is *Windows 8*. The *Direct X Game* and *Direct X and XAML App* templates from *Visual Studio Express 2013* were used as the base for the game engine and game engine's editor, respectively.

In this memory it is explained concepts like *Entity Component System* and how it was applied to this project. The *Entity Component System* is initially explained from a *Object Oriented Design* perspective. Then, there's a little break from *ECS*, and there is a section about "the ability to deal with the cache of the computer of an *OOD* system". At the end, during the explanation of the implementation of the *ECS*, these two themes are brought together.

Also, there are a few components of the engine that are explained more deeply. These components are the *Graphic Engine* and the *Physic Engine*. The *Graphic Engine* follows a simple approach that consist on using a *Camera*, and drawing everything that is inside that *Camera*. The *Physic Engine* follows a "not so traditional" approach. Basically, it updates every game object in the Y axis and then the collisions are managed, then it does the same but in the X axis.

Also, the design done to create the *ECS* is explained. Which *Components* where made, how the *Processors* handle them and the *Assemblages* to generate more *Entities*.

To create a game there are two steps. Firstly create it with the game engine's editor. Secondly, save it with the editor, the result will be a .txt file. Thirdly and finally, add that file to the game engine project, change the program to use that file and play the game engine.

**Índice**

## 1. Terminologia

- Siglas y Acrónimos, más sus definiciones:

*GE:* Game Engine
*GEE:* Game Engine's Editor
*OOD:* Object Oriented Design = Diseño Orientado a Objetos
*DOD:* Data Oriented Design = Diseño Orientado a Datos
*DDD:* Data Driven Design = Diseño Dirigida por Datos

*MMOG:* Massive multiplayer online game. Son juegos donde la cantidad de jugadores activos es inmensa (del orden de miles) y donde las acciones de cada jugador se llevan a cabo en un entorno común al de todos los jugadores.
*RPG:* Role-Playing Game. Juego en el que se utilizan atributos para simular realismo. Por ejemplo, se usa un atributo "vida" para transmitir lo cerca o lejos que está un enemigo de morir. Se suele controlar uno o más personajes, poder recoger objetos del mundo, meterlos en un inventario, equiparse armaduras y armas, mejorar los atributos de un personaje...
*RTS:* Real-Time Strategy games. Son juegos donde se controla un gran número de entidades y normalmente el objetivo es destruir o aliarse con todos los demás jugadores. Eso solo significa que es un Strategy game, el Real-Time viene de que las acciones son en tiempo real, no en turnos. Es decir, cuando un jugador está realizando acciones los otros jugadores también pueden hacer acciones a su vez. Esto tiene más sentido si se compara con su género gemelo, el TBS.
*TBS:* Turn-Based Strategy games: Son juegos de estrategia en que las acciones de los jugadores se ejecutan mientras el juego está en un estado llamado "turno". Hay un turno para cada jugador y mientras sea el turno del jugador los demás jugadores deben esperar o tienen sus acciones muy limitadas.
*PC:* playable-character. GameObject que representa a algún tipo de ser (persona, perro...) y que cuyas acciones pueden ser controladas por el jugador.
*NPC*s: Non playable character. Son entidades del juego que interactúan con el jugador, normalmente uno se refiere a entidades como "ciudadanos de una ciudad", "bestias del bosque",... una piedra no interactúa con un jugador. A no ser que hable.
*CS:* Component Systems = Sistema de Componentes
*ES:* Entity System = Sistema de Entidades, lo mismo que CS.
*ECS:* Entity Component System = Sistema de Entidades y Componentes
*CES:* Component Entity System = Componentes-Entidades-Sistemas
*C:* Components
*E:* Entities
*S:* Systems
*P:* Processors = Systems, exactamente lo mismo, pero llegado un punto se pensó que este nombre era más correcto.

*GO:* Game Object = Una entidad dentro de un juego. Puede ser un pájaro, una moneda,... algo individualizable dentro de nuestro juego. Este concepto tiene el mismo significado tanto si usamos Entities-Components Systems como si no.
*GUID:* globally unique identifier = Normalmente un unsigned integuer. Identifica algo dentro de un sistema de manera unequivoca. Se usa como referencia a dicho elemento.

*WinRT*: Windows Runtime = API que permite la comunicación de un programa con el sistema operativo Windows 8.

*DXGI:* DirectX Graphic Infrastructure = API de DirectX en la que se apoyan todas las demás APIs de DirectX (Direct2D, DirectWrite, Direct3D...). Permite que las demás Apis utilizen recursos comunes y que haya interoperatibilidad. Además, permite la posibilidad de hacer uno mismo una API que funcione junto a las otras APIs de DirectX.
*WIC:* Windows Imaging Component = API que permite el uso de imágenes. Se puede usar junto a las APIs de DirectX como la 2D o la 3D.
*SO:* Sistema Operativo: Programa que gestiona programas.

- Definiciones:

*Product Manager:* Es el encargado en un proyecto que controla que el producto sea el que se quería en un principio.

*Shooter*: Género de juegos en los que se lleva una arma de fuego y se puede hacer "daño" a otras entidades del juego con acciones de disparo.
*Mods:* Son modificaciones de los juegos que no han sido hechos por los desarrolladores del juego si no por usuarios aficionados a la programación y a los videojuegos.

*Power-up:* En un juego, un power-up o potenciador mejoras los atributos de un Game Object de manera momentánea, hasta que "muera" o para siempre. Hay más formas de duración de un Power Up pero esos son algunos de los ejemplos más emblemáticos.
*Inventario:* En un juego RPG, un inventario es un espacio para todo elemento coleccionable por el jugador del juego. Suele representarse como una lista, o a veces un espacio físico donde se ven imágenes de los objetos recolectados.

*Blob:* En diseño de software, Blob es un antipatrón de software. Cuando usamos el patrón Blob estamos teniendo una sola clase, o clases, con muchas responsabilidades (Baja Cohesion y Alto Acoplamiento). Cuando se dice Blob también se puede referir uno a un amasijo de bytes, donde muchos elementos y funcionalidades están entremezcladas.
*Stall:* Estado de un proceso de un SO, cuando necesita datos o instrucciones para continuar procesando pero no tiene.
*Middleware:* software que funciona como puente entre dos o más componentes software. Un Game Engine suele servir como middleware entre el juego y el sistema operativo, por ejemplo.
*Scripts*: Código que un engine puede entender, sirve para añadir funcionalidades avanzadas.

*DirectX:* Una serie de APIs para poder utilizar la tarjeta de vídeo y así mostrar algo por pantalla, el audio,... en general es una API para los componentes hardware que utilizan componentes periféricos, como la tarjeta de vídeo o sonido.
*Direct2D1.1:* API para gráficos 2D que utiliza DXGI (DirectX Graphic Infrastructure) para mostrar imágenes en dos dimensiones por pantalla.
*DirectWrite:* API para escribir texto de la familia DirectX.

## 2. Introducción

Este proyecto consiste en la creación de un Game Engine (GE), un Game Engine's Editor (GEE) y un juego de ejemplo con el objetivo de estudiar una arquitectura de software para GE llamada Entities-Components System (ECS). Para ello se han utilizado las siguientes tecnologías: Windows Runtime (WinRT) API, Direct2D1.1 y DirectWrite de la familia DirectX y Windows Imaging Components (WIC). El lenguaje usado és C++ y la IDE utilizada ha sido Visual Studio Express.

### 2.1. Motivación

Siempre me han gustado los videojuegos. Mi carrera en el tema de hacer un videojuego empezó con el uso de editores de mapas como el editor de mapas de *Age of Mithology* o el editor de mundos de *Warcraft III*. En el fondo ya estaba usando el editor que acompañaba al engine, aunque fuese uno simple.

Más tarde probé a usar engines como por ejemplo el *RPGMaker 2003*, un editor/engine para hacer juegos RPG, aunque últimamente está siendo bastante popular entre programadores novicios y se están haciendo juegos sobre todo de terror o intriga.

Finalmente, ya en una asignatura de la carrera utilicé un framework para hacer juegos en varias plataformas, *Libgdx*. También miré como hacer mods para el juego *Minecraft* y también hice algunas prácticas personales con el engine *CreationKit* para el juego *The Elder Scrolls V: Skyrim*.

Todas estas veces que intentaba hacer juegos o mapas, estaba utilizando un engine, o algo similar. Utilizar un engine significa significa coger todos los recursos (imágenes, sonidos,...) que uno desee y mediante el editor del engine poner estos elementos como uno desee. En el fondo, la parte pesada del juego la está haciendo el engine y, a parte de que siempre he querido saber el significado de "que es hacer un juego", como estaba estudiando ingeniería informática me pareció natural intentar hacer un engine por mi mismo. Esa es mi motivación.

### 2.2. Estado del arte

Actualmente para hacer un juego lo usual es utilizar un game engine (GE) conocido ya que proporciona muchas ventajas, algunas de las cuales son:

- La parte de bajo nivel(cercano al hardware) ya está hecha, con lo cual el equipo de desarrollo se puede dedicar directamente al juego.
- Funciona además de middleware entre el juego y diversas plataformas (sistemas operativos).
- Tienen un amplio soporte por parte de la comunidad de desarrolladores, tanto los desarrolladores que crearon el engine como los desarrolladores que lo utilizan. Normalmente este tipo de ayudas vienen en forma de guías, tutoriales, entradas de blogs o temas en foros.

Y algunas de las desventajas son:

- Suelen costar dinero... o algún tipo de pago. El equipo de desarrollo se puede acostumbrar al uso de un engine en particular. Eso puede tener malas consecuencias a la larga como la falta de capacidad de hacer un engine propio, algo que puede ser poco deseable, o a no poder dominar tan bien otros engines.
- Si una plataforma carece de engine, obviamente, no se puede utilizar uno. Esto era más posible antes, cuando había plataformas como la *GameBoy Advance* de *Nintendo* que apenas

contaban con un GE de uso público. También cuando aparece una nueva plataforma no hay ningún GE dirigido a dicha plataforma, claro que debido a relaciones entre compañías una vez sale una nueva plataforma suele aparecer un nuevo engine para esa plataforma o un engine ya existente aumenta su movilidad y proporciona servicios para esta nueva plataforma.

- Según la arquitectura del GE puede ser que hayan cosas que no se puedan hacer o que, aunque se puedan hacer, implicarían un gran coste de algún tipo (computacional, memoria,...). Cada vez esto pasa menos, sobre todo debido a que hay unos géneros bastante estáticos dentro del mundo de los videojuegos, con lo cual se puede hacer un GE ya teniendo en cuenta para que géneros será utilizado. Pondré algunos breves ejemplos a continuación:

    - Unreal Engines Family: Engines hechos por la compañía Epic Games y especializados para hacer shooters. Algunos de los juegos más famosos son *Unreal Tournament* y *Gears of War*.

    - Quake Engines Family: Creados por la compañía id Software y también para hacer shooters, como los *Quake* o los *Medal of Honor*.

    - Creation Kit y otros Construction Sets: Creados por Bethesda Softwork, son los engines creados para hacer los juegos de la serie *The Elder Scrolls* de Bethesda Softworks. En principio están dirigidos a la creación de juegos de mundo abierto donde hay objetos, Non Playable Characters (NPCs), el jugador posee una características de combate, combates, misiones, historia,... el género supongo que sería open-world RPG.

    - Unity Engine: Creado por Unity Technologies, es un engine que permite hacer juegos de varios géneros, como RPGs, Real -Time Strategy games (RTS), juegos de acción, combate,... está mucho más orientado al 3D que al 2D y su arquitectura es una variante de ECS, aunque se sabe poco de la implementación exacta. Con Unity se han creado juegos tan distintos como *SuperHot* o *Hearthstone: Heroes of Warcraft*.


*Figura 1: Imagen del juego Hearthstone: Heroes of Warcraft, aquí se podía ver como el jugador de abajo (yo) estaba en desventaja.*

### 2.3. Objetivos del proyecto

El principal objetivo es el estudio del ECS para la creación de videojuegos. Como sub objetivos y demostración del estudio se aplicará para la creación de un GE y GEE y finalmente se mostrará un juego simple plenamente funcional.

Además se utilizarán unas APIs y tecnologías especificadas en las sección '3. Tecnologías'.

### 2.4. Distribución memoria

La memoria seguirá la estructura especificada a continuación.

En el capítulo 1 se encontrarán todos los términos y acrónimos utilizados en la memoria, junto a su descripción si es necesario.

En el capítulo 2 se ha hecho una introducción al proyecto. En esta introducción además de un resumen del proyecto, se explican mis motivaciones, estado del arte, objetivos del proyecto y la explicación de como se distribuye la memoria (esta misma sección).

En el capítulo 3 se explicarán las tecnologías utilizadas en este proyecto.

En el capítulo 4 se explicarán las bases teóricas del proyecto, centrándose en la explicación de ECS y su comparación con Object Oriented Design (OOD).

El capítulo 5 tratará del análisis y diseño del proyecto.

El capítulo 6 tratará de la implementación del proyecto.

En el capítulo 7 se explicarán las mejoras futuras que se podrían hacer en el proyecto en su estado actual.

En el capítulo 8 se explicarán las conclusiones a las que he llegado después de hacer el proyecto.

En el capítulo 9 tendremos las referencias y la bibliografías de las que se ha extraído información para utilizarlas en este proyecto.

## 3. Tecnologías

Las tecnologías usadas se dividen en tecnologías para crear el GE y el GEE y tecnologías que se han usado para todo lo demás (por ejemplo para hacer imágenes o para hacer ficheros de texto como esta memoria). No se van a explicar en profundidad ya que no es el tema central del proyecto.

- Tecnologías usadas para hacer GE y GEE:
  - *Visual Studio Express 2013* como gestor de código y proyecto, compilador y debugger.
  - *C++/CX* es el lenguaje elegido para hacer código. *C++/CX* es *C++* pero con algún aspecto añadido como las *class ref* usadas junto a las API de *Windows 8*.
  - *DirectX*, en especial *Direct2D1.1* es la API usada para mostrar imágenes por pantalla.
  - *WIC* para el uso y movimiento de datos de imágenes. Es útil ya que permite el uso de imágenes de manera rápida y comoda junto a *Direct2D*.
  - *XAML* se usa únicamente en el GEE, es usada para hacer la interfaz gráfica de la aplicación.
  - Plantilla para juegos con DirectX: Es una plantilla que viene con Visual Studio Express y que ahorra algo de trabajo, como la puesta a punto de DirectX.

- Otras tecnologías:
  - *Gimp2* para hacer imágenes. Debido a que se han hecho imágenes de baja resolución se podría haber usado cualquier otra aplicación de edición de imágenes, pero con esta ya estaba familiarizado (al menos para el trabajo hecho en el proyecto).
  - *LibreOffice Writer* para este documento.
  - Para visualizar ficheros .txt como valores hexadecimales por motivos de debugg se ha usado *Hex Editor Pro*. Debido a que la creación de un fichero *ASCII* depende del sistema operativo hacia comprobaciones. Un ejemplo de diferencia es que en Windows para expresar un "newline" es /r/n mientras que en sistemas UNIX suele ser /n. Esto era necesario ya que se leen ficheros .txt para generar mundos, en *ASCII*.
  - La página web http://www.correctorortografico.com/ para la corrección de documentos.
  - Plantilla para XAML: Plantilla que viene con Visual Studio Express, ahora algo de trabajo como la puesta a punto de DirectX para XAML.
  - Dia, para la creación de diagramas de flujo, de clases y otras estructuras similares.
  - La página web http://www.web2pdfconvert.com/ para convertir otras páginas web en pdf.

## 4. Bases teóricas del proyecto

En este proyecto se tienen en cuenta algunos conceptos sobre el diseño de software y el comportamiento del hardware poco comunes, así que se dedicará una parte de la memoria en explicarlos.

Los dos temas principales son ECS y la memoria caché, y ese será el orden de explicación. En implementación de ECS en la sección de "Implementación" se explicará como se juntan estos dos temas, uno de diseño y otro de implementación, respectivamente.

### 4.1. Entity Systems

Antes de nada hay que aclarar algo, los términos ECS, ES y CS varían en sus interpretaciones y cada desarrollador les da un significado algo distinto. Así que cada vez que se intente hablar de uno de estos temas hay que dar una información introductoria sobre su significado para uno mismo.

Las interpretaciones más comunes sobre que son estos conceptos son:

- Entity System o Component System: Dos nombres para un mismo significado. Un sistema basado en entidades o componentes, es una parte de un programa (digo esto, porqué normalmente están dentro de un programa más grande que no sigue esta forma de diseño) que, en algún sentido, presenta una división en distintas componentes. Estas componentes interactúan entre ellas para llevar a cabo el funcionamiento del sistema, y los elementos fuera del sistema no tienen forma de comunicarse directamente con estas componentes, solo pueden comunicarse con el sistema en su totalidad. En cierta manera el Entity System es una interfaz que recibe peticiones de afuera y las resuelve dejando que sus componentes las lleven cabo, para luego devolver los resultados. Un ejemplo de Component System sería Component Object Model (COM), donde básicamente los programas que utilizan COM utilizan interfaces para comunicarse con éstos elementos mientras que sus implementaciones no son accesibles por el programa.
- Entity Component System: Forma parte la familia de los ES, abarcando lo que sería el juego en si. La división en componentes tiene dos formas:
  - Los datos de cada Entidad o elemento del juego son descompuestos en distintas Componentes.
  - Por un lado tenemos los datos, representados por Entidades y Componentes, y por otro lado tenemos el código, representado por los Procesadores o Sistemas (dos nombres para un mismo concepto).

Otra aclaración importante es que hay algunos desarrolladores que entienden ECS como algo que no excluye Object-Oriented Design (OOD), mientras otros si que lo excluyen. En mi caso, el proyecto se ha hecho evitando OOD en el ECS, pero para llegar a ese punto es buena idea explicar ECS como algo que no está totalmente apartado de OOD.

Para ello se explicará lo que es un juego basado en Agregación de Componentes, que básicamente se trata de una estructura de código de la familia ES que no se ha apartado totalmente de OOD. No es ECS, pero casi, así que ya va bien para explicar paso por paso.

El resto de esta sección está fuertemente basado en [4].

### 4.1.1. Agregación de componentes

Primero de todo, hay que introducir la idea de no usar OOD en el sentido tradicional. Esto se debe a que OOD presenta algunos problemas, sobre todo con algo como el desarrollo de videojuegos.

Para hacer una pequeña demostración o presentar el problema principal de OOD, se va a dar un ejemplo. Cuando se usa OOD al estilo tradicional para hacer un engine de juego, tenemos un árbol de clases tal que así:

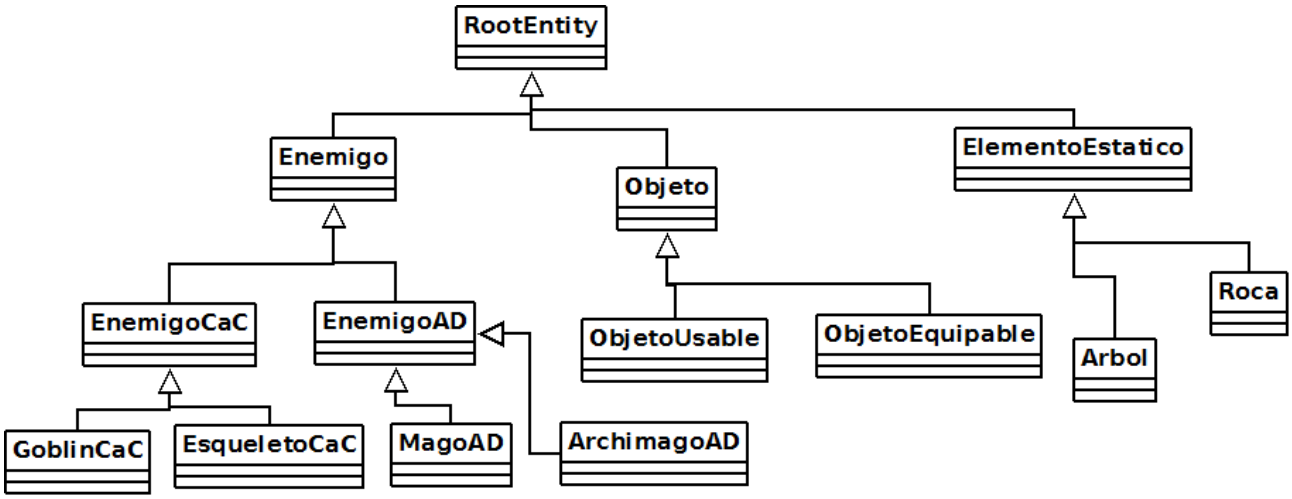


*Figura 2: Ejemplo de diagrama de clases en un diseño OOD para hacer un juego.*

Está bien como diagrama, y mientras las especificaciones de nuestro juego continúen siendo las mismas, este diagrama nos dará un buen soporte de diseño. El problema vendrá cuando haya un cambio en el tipo de juego que queramos ya que entonces se tendrá que modificar el diseño hasta cumpla con las nuevas especificaciones.

Hay que tener en cuenta que el problema no es que las especificaciones cambien, si no que cambien mucho. La aparición de cambios drásticos de manera reiterada que ocurren durante el desarrollo de un juego acaba en este tipo de problemas:

- Anti-patrón Blob: Si muchas clases empiezan a necesitar funcionalidades, se suele añadir estas funcionalidades a la clase RootEntity para que esté a disposición de todos los herederos. Esto es un problema ya que hay herederos que no necesitan esta funcionalidad pero la consiguen. También la clase que representa al jugador se sobrecarga fácilmente.
- Copia de código: Si algunas de las clases hoja (nodos hoja en el diagrama de clases) necesitan funcionalidades, se suele hacer la funcionalidad para una, y luego copiar y pegar el código para las demás.
- Solución más correcta: Las "soluciones" anteriores aparecen cuando no se hace un verdadero estudio de OOD, sobretodo la forma en que las clases heredan unas de otras. Entonces habría soluciones más correctas, aunque una solución más correcta puede implicar cambios más drásticos en el código y, teniendo en cuenta que las decisiones que acabamos de tomar pueden volver a cambiar en el futuro, es posible que ese trabajo extra no solo no sirva si no que nos aleje de la solución final. En realidad esto si es una solución, pero aun así da pie a otros problemas más adelante, dependiendo del camino que haya tomado el equipo de desarrollo del juego.

Esto ocurre debido a que estamos haciendo un juego; y los juegos durante su desarrollo (y en algunos casos desarrollo post lanzamiento) cambian, y mucho. Para entenderlo solo hace falta comparar el desarrollo de un engine para un juego con el desarrollo de una aplicación para apoyar a la administración de una biblioteca.

En una aplicación para la administración de una biblioteca, se podrían tomar algunas decisiones de diseño como por ejemplo :

- Poner una clase Libro.
- Que esta clase Libro tenga un parametro entero numeroDeSerie.

Y lo más seguro es que esto no cambie durante todo el proceso de desarrollo, ya que estamos tratando con libros, y los libros tiene números de serie, y esto paso porqué es una aplicación que interactúa con elementos del mundo real, que normalemente son bastante estáticos. Un juego en cambio no, no es la realidad ni planea ser-lo con lo cual de una iteración a otra pueden haber cambios muy drásticos.

Supongamos que empezamos con la Figura 2 y en una iteración de desarrollo se decide hacer un cambio, de repente ahora se requiere que haya esqueletos que disparen a distancia y que todos los tipos de esqueletos reciban daño extra al recibir daño de "luz". Actualmente tenemos nuestra clase EsqueletoCaC que hereda de EnemigoCaC.

Entonces vamos a presentar tres soluciones siguiendo los errores típicos explicados anteriormente:

- Solución Blob: La solución podría ser que toda instancia de Entidad pueda atacar cuerpo a cuerpo o atacar a distancia, absorbiendo las clases EnemigoCaC y EnemigoAD. Entonces se crearía una clase Esqueleto que heredase de Entidad, y luego una clase EsqueletoCaC y EsqueletoDA que heredasen de Esqueleto. La parte de que "un esqueleto recibe más daño de luz" se llevaría a cabo en Esqueleto, mientras que EsqueletoCaC tendría implementado el método ataqueCaC mientras que el método ataqueAD sería nulo o no haría nada. Para EsqueletoAD haríamos lo contrario con los métodos.

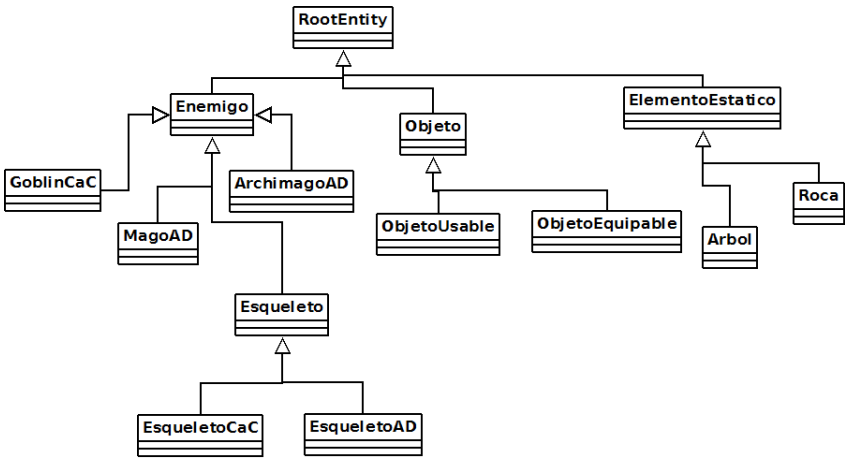


*Figura 3: Diagrama de clases de ejemplo de OOD, con solución Blob aplicada.*

- Solución copia de código: Crearíamos una clase EsqueletoAD que herede de EnemigoAD. Cada uno tendría implementado correctamente su método de ataque, pero ambos tendrían algo de código idéntico para conseguir el efecto de "defensa baja contra daño de luz".

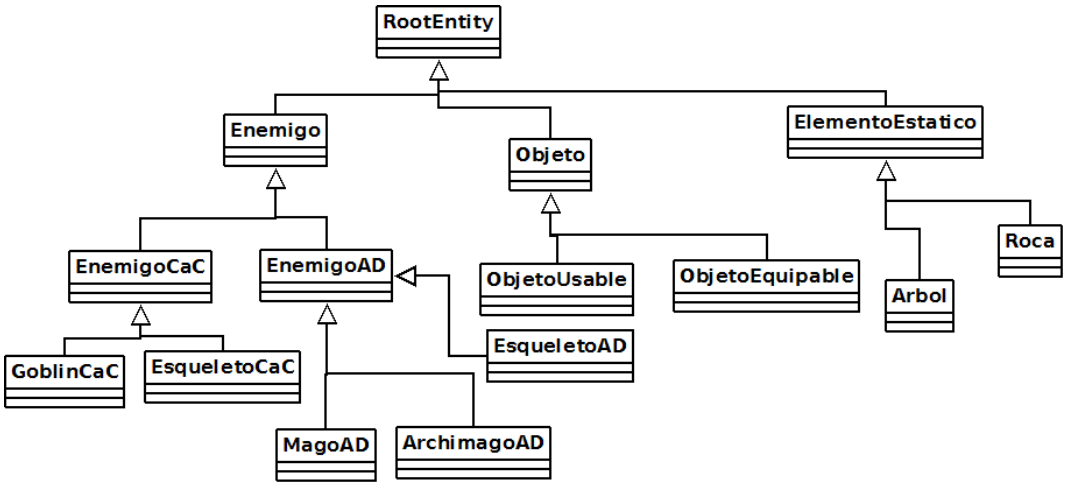


*Figura 4: Diagrama de clases de ejemplo de OOD, con solución copia de código.*

- Solución más correcta: Crear una interfaz Esqueleto y hacer que EsqueletoCaC y EsqueletoAD implementen Esqueleto. EsqueletoAD hereda de EnemigoAD. En cierta manera es la misma solución que la anterior, pero con la interfaz se puede obligar a implementar algún método para resolver el tema del daño extra, lo que está un poco mejor estructurado.

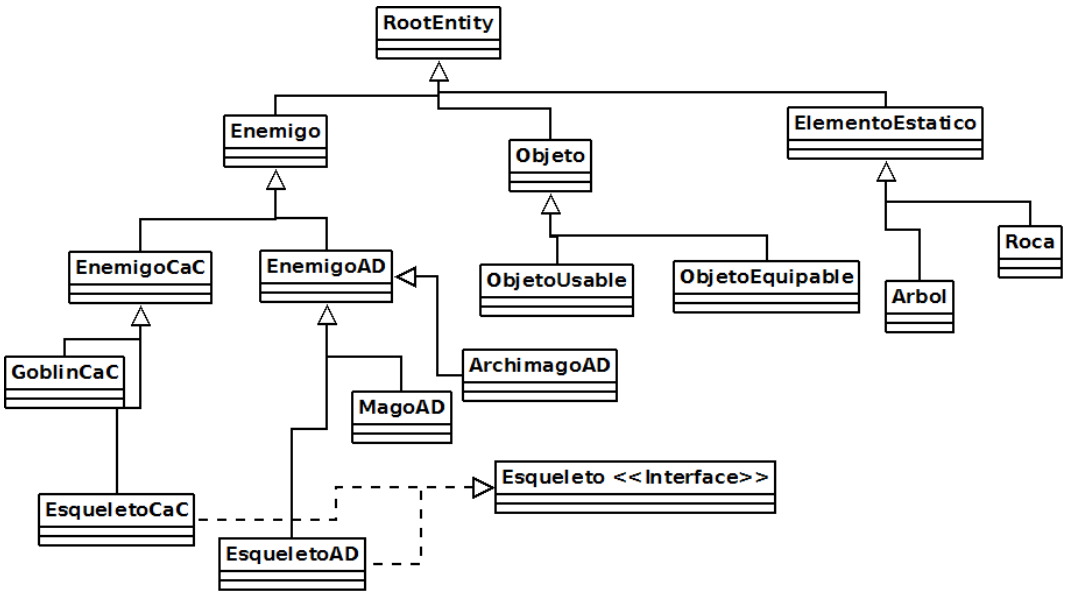


*Figura 5: Diagrama de clases de ejemplo de OOD, con solución correcta.*

El problema como se ha dicho antes no es que hayan cambios, si no que hayan muchos cambios. Ahora supongamos que aparece la idea de que un esqueleto pueda tener alas. El problema se ha complicado y si aparecen problemas de este tipo uno detrás de otro probablemente el equipo de desarrollo habrá acabado haciendo alguno de los errores mencionados arriba.

Se reitera, el problema son los cambios constantes en las especificaciones del juego. Si el juego estuviese diseñado bien desde el principio y no hubiese ningún cambio durante el proceso de desarrollo no aparecerían estos problemas. Además, si algún miembro o miembros del equipo son muy versados en OOD estos problemas descritos anteriormente aunque aparezcan serán resueltos con más rapidez, con lo cual el proceso de desarrollo será más ágil y , por lo tanto, viable.

Al principio se ha comentado que no se usaría OOD tradicional, de eso se trata Agregación de Componentes. Agregación de Componentes consiste en considerar una entidad del juego como un conjunto de componentes en lugar de ser un nodo hoja en un árbol de herencias (sucesión de herencias, al fin y al cabo un único elemento). Para explicar en que consiste esto, se van a pasar por tres fases, cada una más cercana a lo que se entiende como Agregación de Componentes.

#### 4.1.1.1. Objeto como un blob organizado

Antes se tenía una clase Entidad de la cual todas las otras clases heredaban, directa o indirectamente. Ahora se contará con una clase EntidadRaiz que solo poseerá referencias a otras clases.

Estas otras clases serán las Componentes, y cada una será un aspecto de nuestra EntidadRaiz. Para entenderlo un poco mejor, es buena idea hacer una comparación con OOD. Antes si en el diagrama de clases se contaba con la clase EnemigoCaC y una clase EsqueletoCaC que heredaba de esta última, si se expresara en una frase sería "EnemigoCaC tiene un 'ataque cuerpo a cuerpo' y EsqueletoCaC es un EnemigoCaC". Ahora en cambio tendremos una clase EntidadRaiz y una clase AtaqueCaC, y EntidadRaiz tendrá una referencia a una instancia de AtaqueCaC. Si una instancia de EntidadRaiz tiene uno de sus puntero a Null, significa que no posee esa Componente, en caso contrario si que la posee.

Por lo tanto esqueleto, que antes era una instancia de una clase Esqueleto, ahora es una instancia de EntidadRaiz con Componentes como ComponenteVida, ComponenteDañoCaC, ComponenteRender, ComponentePosición y ComponenteDebilVSLuz, todos los demás punteros de EntidadRaiz, a Null. Para hacer un Esqueleto que dispare a distancia, ComponenteDañoCaC será Null y ComponenteDañoAD tendrá un valor, mientras que todos los demás Componentes se quedarán igual.

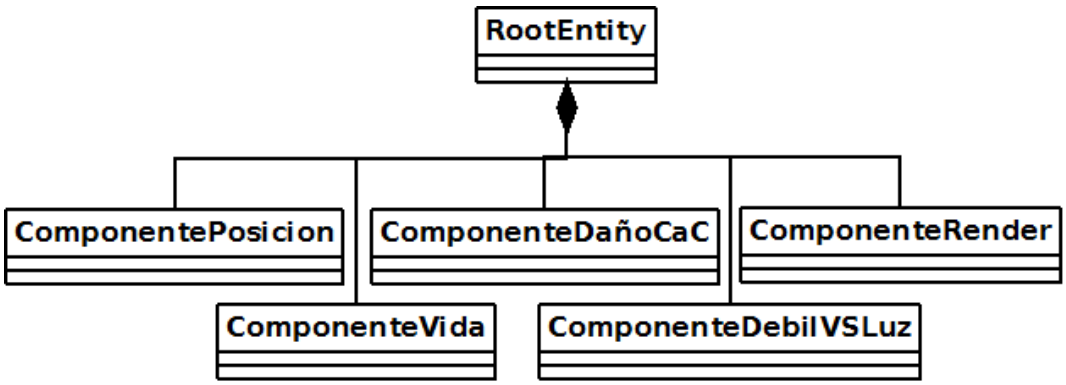

*Figura 6: Diagrama de clases para blob organizado, acorde al ejemplo dado.*

Además, se ha conseguido hacer el engine más ágil, ahora si se quiere crear una nueva Entidad solo hay que utilizar una combinación distinta de Componentes, crear nuevos Componentes,... En realidad es más complicado que eso, sobre todo cuando hay dependencias entre Componentes.

Aun se podría hacer mejor ya, en el estado actual, cada vez que se cree un nuevo tipo de Componente se tendrá que modificar EntidadRaiz. Además, se están utilizando referencias a NULL para describir la falta de un Componente, se puede mejorar.

#### 4.1.1.2. Objeto como contenedor de componentes

Ahora EntidadRaiz, en lugar de poseer una referencia a una instancia de una Componente por cada tipo de Componente, tendrá una lista de referencias. Para esta parte, si aun no se había hecho, se creará una interfaz Componente de la que todo tipo de Componente heredará, de esta manera el atributo de EntidadRaiz será una lista de referencias a cualquier instancia de clase que herede de Componente.

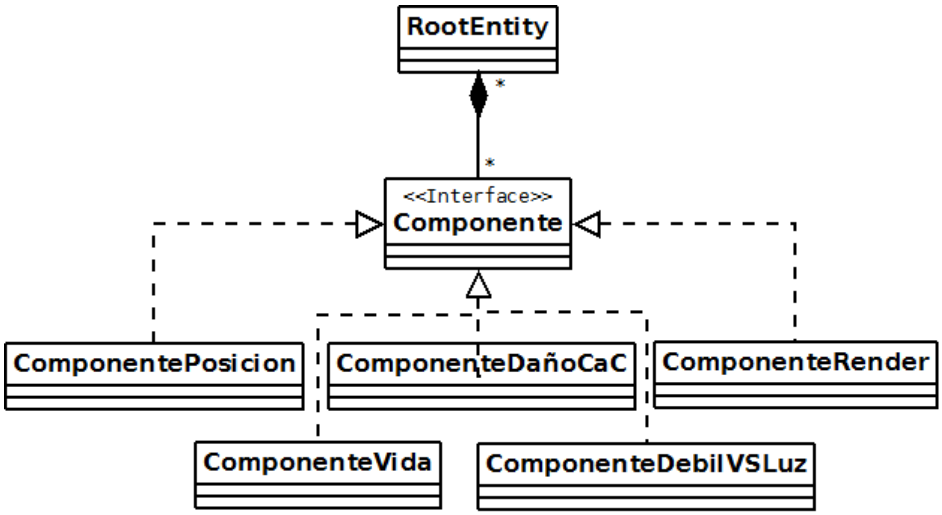


*Figura 7: Diagrama de clases de modelo de juego, versión contenedor de componentes.*

Ya no hay Nulls de los que preocuparse, y además ya no es necesario modificar nada más EntidadRaiz ya que todo la responsabilidad recae en sus componentes. Realmente EntidadRaiz es una como una interfaz que agrupa Componentes.

En este paso suelen quedarse muchos equipos, al menos la primera vez que se hace un proyecto ya que es suficientemente bueno. Además, el siguiente paso es el más complicado de asimilar, sobre todo si uno está acostumbrado a OOD clásico.

#### 4.1.1.3. Objeto como pura agregación

En esta etapa se destruye la clase EntidadRaiz y una Entidad dentro del juego pasa a ser su composición de Componentes. Todas las instancias de una misma Componente pasan a estar ubicadas en una misma lista, por lo tanto se tiene una lista por cada tipo de Componentes.

En este paso se pasa de trabajar con instancias de EntidadRaiz a trabajar directamente con los Componentes por separado. Entonces, a la hora de crear una Entidad se eligen los Componentes que la forman. A continuación un ejemplo de como podrían ser algunas Entidades siguiendo esta forma de programar:

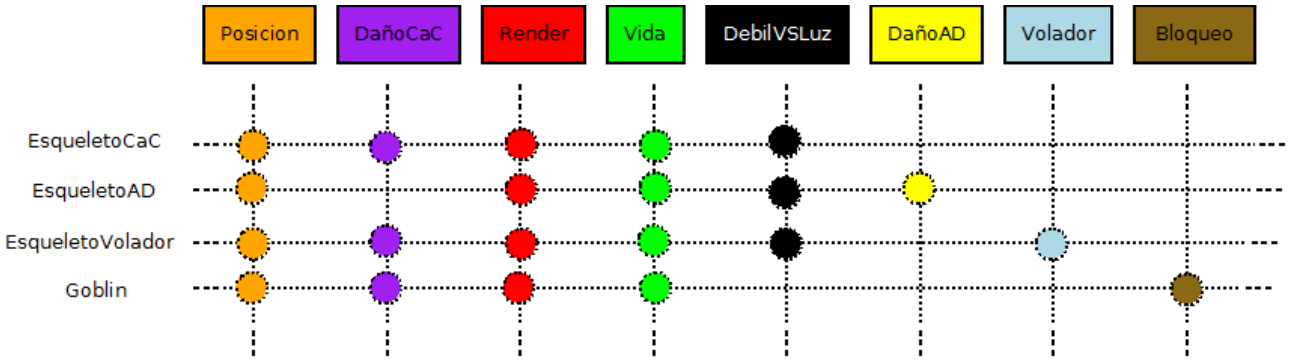


*Figura 8: Malla representando la composición de Entidades por Componentes.*

Esto tiene sentido recordando la estructura del bucle principal de un juego:

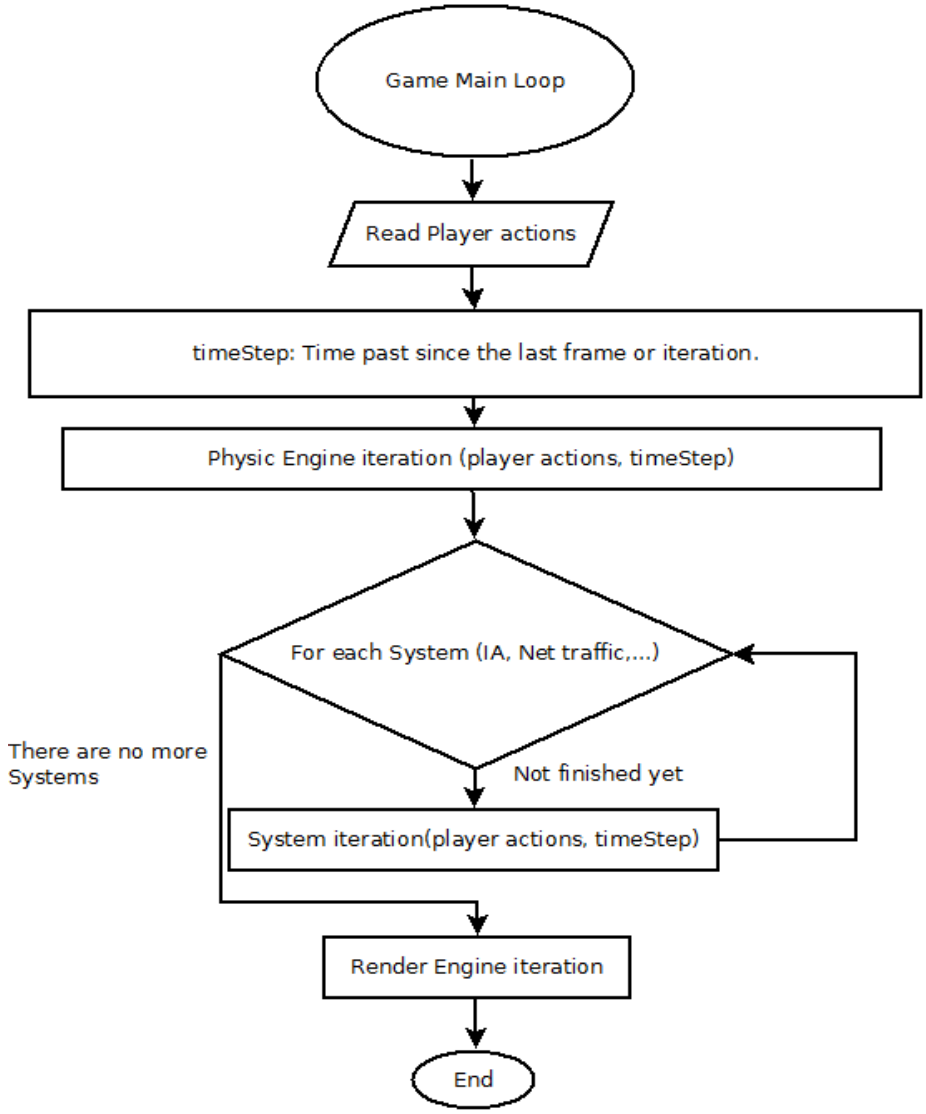


*Figura 9: Típico bucle principal de un juego.*

En los pasos de "Physic Engine iteration", "System iteration" y "Render Engine iteration" lo que se hace es trabajar con unas partes de una Entidad del juego. En el motor físico por ejemplo se trabajará con Componentes físicas de las Entidad; como las posiciones de las Entidades en el mundo, sus tamaños o incluso formas dependiendo del motor físico; en cada Sistema se tratará solo una parte de cada Entidad (dependiendo del Sistema) y en el motor gráfico solo la información necesaria para poder mostrar el estado actual del juego por pantalla. En resumen, en cada etapa de procesamiento no se trata con cada Entidad en su totalidad, si no con un fragmento de éstas.

Los beneficios de este tipo de diseño no son apreciables aun, por eso mismo muchos equipos no llegan aquí, pero cuando se explique en la sección '4.3.1. La cache y la ubicación de datos en memoria local' los problemas que hay con el funcionamiento de la cache y la arquitectura del GE entonces se entenderá mejor el porqué se pasa a trabajar solo con los Componentes destruyendo la EntidadRaiz.

Por otro lado, si que se puede hablar del problema que se ha ganado, ahora no se sabe a que entidad pertenece un Componente y eso es un gran problema, sobre todo cuando hay dependencia entre Componentes. Un buen ejemplo sería a la hora de mostrar por pantalla el estado actual del juego, necesitamos saber donde está una entidad (Componente de posición) para poder dibujar sus gráficos (Componente de renderización).

Para arreglarlo hay varios métodos, aunque ahora mismo solo interesa el más simple, que consiste en usar un identificador global único (globally unique identifier, GUID) y que cada instancia de Componente posea dicho identificador. De esta manera cuando se esté usando un Componente y se quiera usar otro Componente de la misma entidad, se mirará su identificador y a continuación se buscará su Componente hermano entre las instancias de otra Componente (poseerá el mismo GUID). Si en algún momento esto empieza a ser demasiado lento debido a las búsquedas en otras listas, se tendrá que recurrir al uso de una clase que sepa que Componentes pertenecen a una Entidad y cada Componente tendrá un puntero a dicha clase. Se llamará EntidadMetaData, aunque no se diferencia en su estructura respecto a la EntidadRaiz del apartado anterior. La diferencia en la nomenclatura se debe a que ahora no se están procesando Entidades, si no Componentes, esta nueva clase es auxiliar.

#### 4.1.1.4. Resumen de ES

En realidad este resumen es de agrupación de componentes, que forma parte de la familia ES. De momento lo que se ha conseguido es:

- Un engine, que sigue un diseño muy particular.
- En este engine, tenemos Entidades de forma implícita, existen como agrupación de Componentes.
- Tenemos dichos Componentes.
- El engine trabaja sobre todas las instancias de un tipo de Componente en cada paso, o en un conjunto de tipos de Componentes.
- Se tiene algún tipo de mecanismo de identificación de los Componentes para poder resolver las dependencias que hay entre ellos.

### 4.2. Que es ECS?

Si ES es una familia de diseños o arquitecturas que consisten en estructurar el código en componentes que interactúan entre ellos y que para acceder a ellos desde fuera del sistema hay que hacerlos a través de una interfaz, y Agregación de Componentens habla de dividir las Entidades del juego en Componentes y en trabajar estos Componentes, que es ECS? En [1.1] se puede confirmar lo que ahora se explicará.

ECS también forma parte de la familia ES y también divide las entidades en componentes. Lo que añade son los Sistemas o Procesadores. Una vez más, son dos nombres para el mismo concepto, aunque actualmente se suelen llamar Procesadores.

Hasta ahora aunque los diseños se alejaban del OOD tradicional, seguía siendolo en el fondo. Al fin y al cabo un Componente tenía sus métodos que eran llamados desde el bucle de juego principal. Con ECS esto cambia y se separa totalmente de OOD.

Para explicarlo, lo mejor es explicar que son cada uno de los elementos de un ECS:

- Entidad: No existe como tal, solo de manera implícita debido a la existencia de sus Componentes. Aquí no hay ningún cambio.
- Componente: Un Componente son los datos de un aspecto de una Entidad. Por ejemplo, se quiere representar "la posición de una Entidad dentro del mundo", entonces se crearía un Componente componentePosicion que tendría los datos necesarios para representar este aspecto. Por ejemplo: en un mundo 2D, la componentePosición contendría los datos de la posición en X e Y de la entidad.
- Procesadores: Un Procesador son las operaciones(métodos) de un aspecto de una Entidad. Su función principal es la de aplicar algoritmos sobre todas las instancias de un tipo o un conjunto de tipos de Componentes.

  Siguiendo el ejemplo anterior, teniendo un componentePosicion y también un componenteVelocidad, se crearía un Procesador procesadorFisico que, para cada Entidad que posea tanto componentePosicion como componenteVelocidad, convertiría la velocidad de componenteVelocidad a un valor de movimiento (para eso se usaría el tiempo pasado desde el último frame de ejecución o alguna medida de tiempo similar) y se añadiría a la posición de componentePosición pertinente (el de la misma Entidad).

En eso consiste principalmente ECS, hacer una división de los datos de un programa y de los métodos en Entidades/Componentes y Procesadores, respectivamente. Este tipo de diseño consigue dos cosas:

- En OOD se tendría una instancia por cada entidad, y a la hora de hacer el bucle de juego principal en cada paso se haría una iteración sobre cada entidad para utilizar una fracción de sus datos/métodos. Ahora, se tiene todo separado; para cada paso del bucle de juego solo se haría una llamada al Procesador o Procesadores pertinentes, donde a su vez cada uno de ellos iteraría solo sobre las componentes que trabajan. Así que lo que se consigue es una forma de diseño distinta a la habitual y aceptable.

  Además, en OOD tenemos entre mezclado código con datos en las clases y en los objetos, mientras que en ECS los datos están por una parte y el código por otra.

- Más adelante, una vez explicado el tema de la cache, se entenderá el otro beneficio de esta forma de diseño.

### 4.3. There be Dragons

Necesitaba un título para una serie de temas que són complicados en algún sentido, así que decidí dar un aviso.

En estas secciones se hablará sobre la cache y sobre como distintos paradigmas de programación la manejan.

#### 4.3.1. La cache y la ubicación de datos en memoria local

La información de esta sección se encuentra en [3], [5] y [6].

La cache de la CPU es una memoria hardware que es más pequeña que la random-access memory(RAM) pero también más rápida. En los procesadores actuales suelen haber hasta tres niveles de cache incorporados, cuyo tamaño disminuye pero cuya velocidad aumenta a medida que están más cercanos al procesador y a sus registros.

Si la RAM se usa para solucionar el problema de que los discos duros y otras memorias similares son demasiado lentas para pasar esos datos directamente al sistema procesador, entonces la memoria cache hace exactamente lo mismo con la RAM.

Desde los años 80 hasta ahora la diferencia entre la velocidad de los procesadores y las memorias RAM ha ido incrementándose hasta tal punto que enviar un dato de la RAM a los registros de un procesador tarda 600 ciclos de procesador (aproximadamente, depende del procesador y de la RAM). Durante este tiempo el procesador tiene que esperar a recibir los dados y si no tiene ningún dato a disposición para procesar se puede llegar a un estado de Stall. Durante ese estado, se está perdiendo tiempo de procesamiento con lo cual si esto ocurre con una aplicación "parece" que es lenta. En realidad tarda poco solo que hay periodos de inactividad en medio del procesamiento, haciendo que el tiempo real de ejecución sea lento.

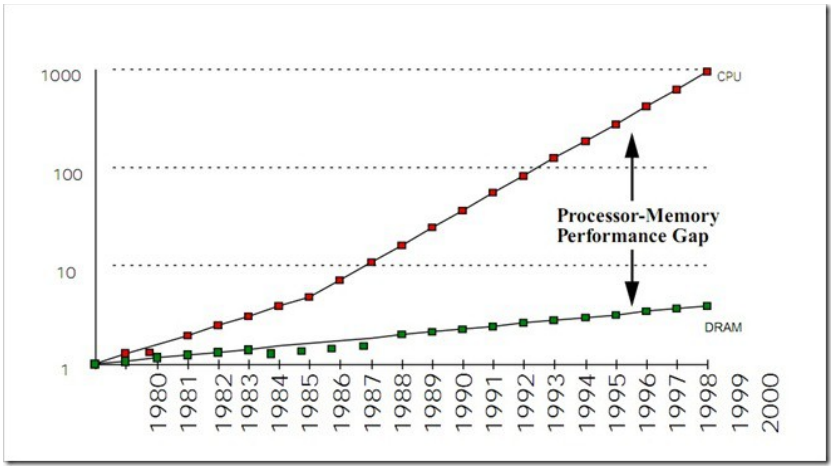


*Figura 10: Imagen mostrando la diferencia entre las velocidades de procesadores y RAM a lo largo de los años 1980-2000. Extraída de [8]*

La solución no vino solo en forma de una memoria más compacta y rápida si no también en el envío de datos desde la RAM hasta la cache en grupos. Es decir, en vez de enviar un solo dato se envía un conjunto de datos donde el dato deseado está incluido. De esa manera, la próxima vez que el procesador requiera un dato, quizás ya esté en la memoria cache y no hace falta ir hasta la RAM, ahorrándose mucho tiempo de procesamiento perdido.

En principio se puede diferenciar entre usar la cache para datos o para instrucciones. Se hablará principalmente de la primera parte.

Normalmente el procedimiento para abastecer a la cache consiste en:

- Cuando se pide un dato a la memoría RAM se mira si está en cache.
  - Si está, se coge de la memoria cache. (cache hit)
  - Si no está se recupera un conjunto de datos de la RAM (dato deseado incluido), se ponen en la cache y en el procesador. (cache miss)

Este bloque de datos que se envía desde la RAM hasta la cache, es un bloque de datos contiguos en RAM, es decir, que es físicamente compacto en una sola ubicación.

Ahora hablemos de OOD tradicional y de como se aprovecha de la existencia de la cache para hacer que los datos lleguen antes a la RAM y de esa manera consigamos un programa más "rápido".

No lo hace en absoluto. En OOD pensamos en clases, objetos,... son elementos abstractos de diseño, tan abstractos que temas como la cache no los tocan. Si se va a lo que es la implementación de un programa diseñado en OOD, con un lenguaje como Java o C++ que tiene una implementación de OOD, nos encontraremos que para conseguir el resultado deseado se están utilizando una cantidad de punteros bastante elevada. Antes de continuar, para que las afirmaciones a continuación fueran ciertas tendría que verme la implementación de algunos compiladores, cosa que no he hecho, pero a partir de comentarios en libros, papers y blogs creo que el modelo de datos presentado a continuación es una aproximación bastante buena.

Para cada instancia de una clase, un objeto, tenemos todos los atributos del objeto juntos en memoria (RAM) y suele haber un puntero hacia una tabla de punteros a métodos de la clase, en la implementación de la clase. Esta tabla suele llamarse virtual table, y posee un puntero por cada método virtual implementado por la clase, directamente a la implementación. Este tipo de información se puede encontrar en [7.1]

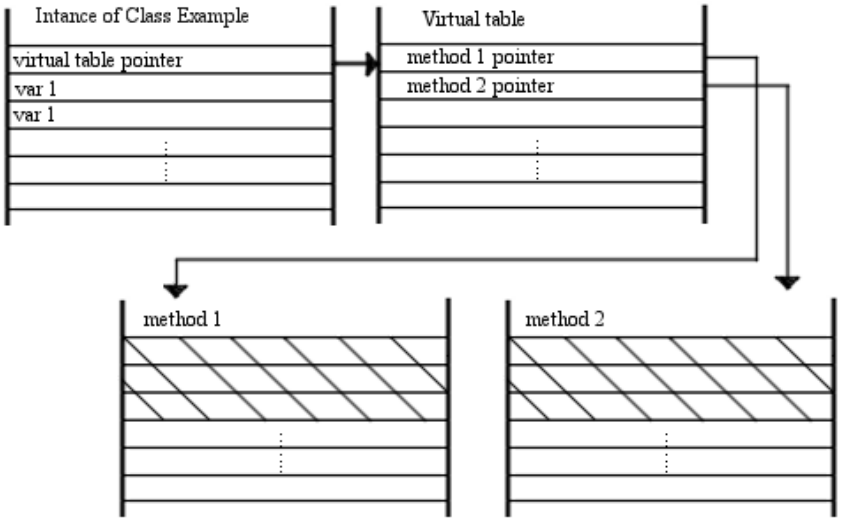


*Figura 11: Imagen representativa de un Objeto a nivel de memoria RAM.*

Además, debido al tema del polimorfismo no se puede tener realmente un array de objetos de distintos tipos, ya que pueden tener distintos tamaños. La realidad, lo que se hace es crear un array de punteros a objetos, ya que los punteros si que tienen un solo tamaño.

El problema viene a raíz de que la ubicación física de los objetos muchas veces no se tiene en cuenta en OOD tradicional. Tanto en Java como en C++ se suele llamar a una instrucción new que al fin y al cabo consiste en hacer una petición al Sistema Operativo (SO) para que le deje un espacio libre de la RAM para poder crear los datos del objeto. Esa posición la proporciona el SO.

Así que no es difícil darse cuenta de que lo que suele pasar es que los objetos están ubicados físicamente de manera errática en memoria, con lo cual a la hora de procesar todos los objetos uno a uno en algún paso de nuestro bucle principal de juego, estamos dando salto en memoria provocando que la cache no pueda hacer su trabajo de manera satisfactoria.

Además hay un problema más, en un bucle de juego principal en un engine diseñado con OOD lo normal y típico es que en cada paso de dicho bucle se accedan a todos los objetos secuencialmente. Antes se ha descrito como están ubicados los datos de los objetos en memoria, para cada objeto todos sus atributos están juntos en memoria. Pero en cada paso del main loop se está accediendo solo a unos datos de cada objeto, no a todos. Así que incluso aunque se tuviesen los objetos juntos en memoria, cada vez que se quisiese un dato de un objeto la caché se llenaría con todos los datos de cada objeto adyacente. Eso lleva a llenar mucho más rápido la cache, lo que implica que se tendrán muchos más misses que antes.

En '4.1.1.3. Objeto como pura agregación' se habla de que en cada paso se tratan solo algunos Componentes del conjunto entero, y aquí se puede ver como el tener todos los Componentes de un tipo juntos en memoria puede provocar un mejor uso de la caché. En '6.1. Implementación del ECS' se verá como estos dos temas se unen.

### 4.3.2. Data Oriented Design

Continuando con el tema de OOD explicado anteriormente, hay una explicación para el pobre uso de la cache por parte de un programa hecho en un lenguaje como Java o C++, usando OOD. C++ es uno de los lenguajes más importantes en la industria de los videojuegos, así que me centraré en este. C++ fue creado alrededor del año 85, y según la figura 10 la diferenciación de velocidades entre RAM y Procesador no era ni mucho menos el problema que es ahora en aquel entonces. Por lo tanto la razón por la cual OOD actúa pobre a la hora de manejar la cache es porqué en su época no había tanta necesidad de usar una caché, no era un problema con una gran prioridad.

Lo peor de todo es que la caché es una solución de hardware y el software muchas veces no hace los esfuerzos necesarios para que esta solución sea efectiva, además a medida que pasa el tiempo la diferencia de velocidades procesador-RAM continúa aumentando y, por lo tanto, el problema solo va a peor. Así que es algo a tener en cuenta a la hora de diseñar programas, sobre todo programas que tienen que ser rápidos, como por ejemplo un game engine.

Hay un paradigma distinto a OOD, conocido como Data Oriented Design (DOD). DoD piensa en datos, en como moverlos y en como procesarlos de manera eficiente. El objetivo de un buen DOD es reducir un programa a inputs, cadena de algoritmos y output.

No voy a entrar más en el tema de DOD debido a que no es el tema que se está tratando en este proyecto, pero se tiene que mencionar ya que en un paradigma de programación que trata bien con problemas como los de la cache. Además, si uno lo piensa bien, un bucle principal de juego es básicamente el resultado de un DOD, si encima se tiene en cuenta el uso de ECS, que divide el problema en datos y algoritmos no es muy difícil ver que hay una conexión aun más fuerte.

### 4.3.3. Evolución de ECS

El ECS que se ha explicado hasta ahora es una forma de diseño, solo explica como hacer la división de un programa entre datos y código, mediante Entidades (implícitas), Componentes y Procesadores.

Una vez visto todo el problema que hay con la cache y que un diseño tradicional no tiene los mecanismos necesarios para atacar dicho problema, se puede dar un paso en ECS y pasar de una forma de diseño a una forma de programar.

El esfuerzo del nuevo ECS consiste en como organizar de manera física los Componentes para aprovechar la cache todo lo posible. Es un tema muy amplio y llegado un punto entra en otros temas como el de fragmentación de memoria, debido a que las soluciones consisten principalmente en tener todas las Componentes de uno o varios tipos de Componente juntas de manera compacta en memoria, y en el momento en que una Entidad muere, hay que remover sus Componentes, dejando huecos en la memoria.

No se va a entrar en todas las formas de implementar un ECS, en lugar de eso solo se va a explicar una de las posibles implementaciones, la usada en este proyecto, en '6.1. Implementación del ECS'.

## 4.4. Especificaciones nuevas a ECS

ECS es relativamente joven. Por un lado empezó a hablarse en 2003 a raíz de una presentación hecha por Scott Bilas sobre la arquitectura de *Duengeon Siege*, donde ya hablaba de un ES muy

similar a ECS y de uso junto a Data-Driven Design (DDD) para agilizar el desarrollo de un juego. Entre 2003 y 2007 hubo un periodo de aprendizaje de la comunidad y se usaron ya en otros juegos como *Operation Flashpoint: Dragon Rising* o *Prototype*.

Otra figura muy importante es Adam Martin, de su blog se sacó gran parte de la información sobre ECS para este proyecto, es la fuente principal. Estos elementos que se van a comentar ya se habían estado usando pero la primera vez que aparece en su blog es en 2014. Por lo tanto aunque ECS ya tiene unos 11 años, en cierta manera aun es joven y requiere aun trabajo.

### 4.4.1. Ensamblajes

Si se compara OOD con ECS, se puede apreciar que los objetos de OOD están representados por las Entidades, los Componentes y los Procesadores de ECS. Hay un elemento de OOD que aun no tiene equivalencia en ECS, las clases.

En cierta manera una clase es un manual para hacer un objeto, es algo rígido, y ECS se caracteriza por ser fluido y permitir cualquier posible composición de Componentes. Aún así, en caso de que se quiera hacer 100 Entidades que sean totalmente iguales no se posee ningún mecanismo para ello.

Aquí es donde entran los Ensamblajes, explicados en [1.3]. Básicamente son métodos que inicializan una Entidad con los mismos Componentes siempre. Para evitar confusiones, no se está diciendo que dos Entidades que han sido creadas mediante el mismo Ensamblaje compartan las mismas instancias de Componentes, ambos tendrán sus propias instancias de Componentes, pero estas instancias serán de los mismos tipos. Por ejemplo: si se quisiesen crear 100 Esqueletos, se crearía un Ensamblaje que a la Entidad designada le añada lo Componentes de posición, tamaño, elementos de renderización, IA.... y la defensa baja contra Luz claro.

Un dato importante, una Entidad que haya sido creada por un Ensamblaje estará sujeta a cambios y se le podrá añadir o quitar Componentes, como con cualquier otra Entidad.

### 4.4.2. Relaciones entre Entidades

En ECS se especifica que se tiene que tener algún mecanismo para relacionar las Componentes de una misma Entidad entre ellas, pero no dice nada de relaciones entre Entidades o relaciones entre Componentes de distintas Entidades.

Esto es un problema, ya que habrá veces en que le pasará algo a una Entidad y se querrá hacer algo con otra Entidad que tiene relación con la primera. Un pequeño ejemplo de esto es el típico inventario en un Role-Playing Game (RPG) donde lo que queremos es una relación del tipo "La Entidad A tiene en su inventario la Entidad B".

Una forma de arreglarlo sería crear un Componente que entienda de relaciones entre Entidades, y también un Procesador que se encargue de esto.

Este tema lo pensé un poco debido a que en el engine hay un paso de bucle de juego principal en el que se resuelven colisiones entre Entidades. Pensé que tendría que utilizar algún mecanismo para resolver esto ya que una colisión es una relación entre unas Componentes de una Entidad (posición y tamaño) y las mismas Componentes de otra Entidad. No era necesario, porqué aunque si que es una relación entre entidades es de muy corta duración, se resuelve en el mismo Procesador y por lo

tanto no hace falta tener algún mecanismo para hacerlo patente a largo plazo.

Así que en realidad cuando en esta sección cuando hablamos de relaciones entre Entidades nos referimos a una relación a largo plazo.

## 5. Análisis y diseño

Primero de todo, este Engine será para juegos de plataformas 2D, al estilo *Super Mario Bros*.

Hubieron dos fases de análisis y diseño, en la primera fase se hizo un ECS simple, se especificó elementos como el tamaño del mundo de juego y como funciona el sistema de coordenadas, motor gráfico, un pequeño motor físico, el editor del engine y mecanismos para poder traspasar datos del editor al engine.

En la segunda fase se hizo un ECS y un motor físico más complejo.

La razón de porqué se explicarán las dos fases de diseño en lugar de solo la última es debido a que son en general complementarias. Aun así todo lo que se haya hecho durante la primera fase y se haya modificado en la segunda no será explicado a fondo, ya que se explicará en la segunda fase. Una excepción a eso sería el ECS, pero se debe a que lo explicado en la fase 1 sigue vigente en la fase 2.

Además antes de empezar, una mención: muchas veces se habla de "tiempo pasado entre frame anterior y actual", se trata de una variable que da esa información y se extrae de las clases auxiliares que vienen al crear un proyecto basado en una template para DirectX.

### 5.1. Fase I

#### 5.1.1. El mundo, el sistema de coordenadas y sus variables

Antes de empezar a hacer el ECS, pensé que quizás tendría más sentido empezar por definir el mundo del juego ya que cada Entidad tendrá una Componente posición, y para poder hacerlo se necesita algo de información de lo que es una posición. Además tuve en cuenta que el juego utiliza tiles para mostrar lo que sería el mundo estático.

Básicamente el mundo es un sistema cartesiano de coordenadas reales centrado en el 0.0, cuyo límite derecho es un valor real positivo, limite izquierdo un valor real negativo, limite superior un valor real positivo y limite inferior un valor real negativo. La posición de una entidad o un tile será la posición en este sistema de coordenadas la esquina inferior izquierda del cuadrado que ocupa, y el tamaño será el espacio entre la esquina inferior izquierda del cuadrado y la esquina derecha, también el espacio entre la esquina inferior izquierda y la esquina superior, width y height respectivamente. Los tiles además tendrán siempre una posición entera y una anchura y altura de 1, de esta manera si por ejemplo tenemos un mundo que va desde -20.0 a 20.0 en ejes x e y, entonces podemos tener hasta 40x40 = 1600 tiles; cualquiera de los tiles en las posiciones que estén en x o y en 20.0, no se cuentan ya que se salen de los bordes del mundo. Las entidades pueden estar en cualquier parte siempre que respeten los bordes del mapa, teniendo en cuenta su posición y tamaño.

La razón por la que se ha escogido un sistema de coordenadas centrado en el 0 es por un tema de implementación. Al implementarlo ya se tuvo en cuenta que se usarían valores float o double de C++, y un valor float (o double) cuando mayor es (tanto en negativo como en positivo) deja de representar algunos valores. Esto pasa porqué estamos usando el mismo numero de bits para representar el valor decimal sin importar el valor del exponente usado en los floats. Así que, si se centra el sistema de coordenadas en 0, el mundo tendrá que ser mucho más grande para llegar a ese punto en que los valores de posición tienen acceso a menos valores, al menos comparado con uno en que por ejemplo el 0,0 estuviese en la esquina inferior izquierda del mundo.

Además hay otras dos variables de mundo, la gravedad. Con ella en el motor físico se usará junto a la velocidad de cada Entidad para calcular la nueva posición. La segunda no es exactamente de mundo, se trata de *convWorldToScreen* y se trata de una variable que traduce de coordenas de mundo a coordenadas de pantalla. Actualmente vale 32, porqué los tiles, que ocupan 1 de tamaño en el mundo, ocupan 32 píxeles.

### 5.1.2. ECS simple

Los datos de una Entidad deberían ser:

- Posición en el mundo.
- Tamaño en el mundo.
- Imagen a mostrar por la pantalla.
- Velocidad en el mundo.
- Algún tipo de información para saber si se puede saltar más.
- Algún tipo de información para saber si esta Entidad acepta Inputs.
- Algún tipo de información para saber si esta Entidad es una cámara.

Y los Procesadores:

- Un Pocesador cuya entrada de datos son los inputs (teclas apretadas, click del ratón,... ) y que se dedica a cambiar parametros de las entidades que estén sujetas a inputs.
- Un Procesador que se dedica a modificar las posiciones de las Entidades en función de sus velocidades, colisiones entre ellas y colisiones con los límites del mundo.
- Un Procesador que se encargue de mostrar por pantalla todas las Entidades que estén dentro de la bounding box de una cámara (la bounding box se calcula con la posición y tamaño de dicha cámara).

Es posible que dos Entidades posean la misma imagen, así que es importante que no hayan dos copias de dicha imagen en memoria, sino solo una imagen y dos punteros o identificadores de la imagen. Esto es conocido como el patrón Fly-weight.

Esto sería un ECS simple y correcto, pero se puede ver que hay un exceso de información.

Por ejemplo, para este juego solo va a haber unas Entidades cámara y jugador y además los tiles son un tipo especial de Entidades, así que se han hecho las siguientes decisiones:

- Tanto Cámara como Jugador son Entidades especiales, así que se tratan de manera distinta a las demás Entidades.
- Cámara carece de "imagen a mostrar", "información de salto", "información de input" y, de hecho, ya que solo la Entidad Cámara es una cámara todas las entidades carecen de "información de cámara".
- Debido a que Jugador es la única Entidad que reacciona al input, ya no es necesario que cualquier otra Entidad tenga "información de input".
- Los Tiles también son un tipo distinto de Entidades, solo poseen "posición", "tamaño" y "imagen a renderizar". Nada más, ya que no se mueven ni saltan. Además, en realidad todos los Tiles tienen un único tamaño, algo que se tendrá en cuenta.

De esta manera acabamos con cuatro tipos de Entidades:

- Entidad Jugador:
  - Componentes:
    - renderPosition
      - posX
      - posY
    - renderSize
      - width
      - height
    - pyshicPosition
      - posX
      - posY
    - pyshicSize
      - width
      - height
    - render
      - imageId
    - speed
      - speedX
      - speedY
  - Atributos:
    - float playerDiffPRX
    - float playerDiffPRY
    - boolean playerIsOnLand
- Entidad Camara:
  - Componentes:
    - position
      - posX
      - posY
    - size
      - width
      - height
    - speed
      - speedX
      - speedY
- Entidades Tiles:
  - Componentes:
    - position
      - posX
      - posY
    - size (solo 1 para todos los Tiles)
      - width
      - height
    - render
      - imageId

- Entidades genéricas:
  - positionComponent
    - posX

        - posY
    - sizeComponent
        - width
        - height
    - renderComponent
        - imageId
    - speedComponent
        - speedX
        - speedY

Como se puede ver, Jugador ha ganado unas Componentes extras y algunas variables nuevas. Básicamente tiene una posición y tamaño físico, que son la que se usarán en el Procesador físico y otra posición y tamaño para el Procesador de renderizado. Esto es así porqué en un momento dado se vio que el Jugador parecía flotar. Además playerDiffPRX e Y sirven para saber la diferencia en la posición entre una y otra, de esta manera cuando se haya trabajado con la parte física del jugador se hará una actualización de la imagen a renderizar con estos valores. PlayerIsOnLand sirve para saber si el jugador está en el suelo o no, se actualiza en el procesador físico (physics) y se comprueba en el procesador de input para hacer efectivo un salto o no.

Ahora los Procesadores, teniendo en cuenta que hay distintos tipos de Entidades, funcionan de la siguiente manera:

- inputProcessor: Se encarga de actualizar la velocidad de la Entidad Jugador en función del input recibido. En caso de que playerIsOnLand está activa, no se añade ningún movimiento en caso de que haya habido un salto, en caso contrario si se ha producido un salto si que se modificaría el movimiento vertical del jugador.
- physicProcessor: Se encarga actualizar la posición de las Entidades Jugador y Cámara, comprobar las colisiones que tienen con los Tiles y con el mundo y rectificar la posición. Además debe actualizar la posición y el tamaño de render de Jugador utilizando playerDiffPRX e Y. La idea es que cuando se crea un Jugador la posición y el tamaño físicos se pueden entender como una bounding box centrada en el centro de la bounding box de la posición y tamaño de render pero más pequeña. Este "más pequeña" se guarda en playerDiffPRX e Y, y una vez se ha actualizado la parte física de la Entidad, solo hace falta usar playerDiffPRX e Y junto a esa parte física para actualizar la parte de render.
- RenderProcessor: Comprueba si una Entidad está contenida total o parcialmente dentro de la bounding box de la Entidad Cámara, bounding box creada con su posición y tamaño. Si una Entidad está dentro de la Cámara totalmente se dibuja toda la imagen, en caso de que esté contenido parcialmente solo se dibuja una parte y si no está contenida en absoluta no se dibuja.

En este punto los Ensamblajes existentes son el de Jugador y los de los Tiles. El del jugador solo necesita saber la posición del Jugador, todo los demás atributos se ponen en el Ensamblaje, los de los Tiles son el mismo y solo necesitan la posición y la imagen de Tile a poner.

### 5.1.3. Motor físico

El motor físico, ubicado en physicProcessor, tiene como objetivo actualizar las posiciones de todas las Entidades y de reposicionar según hayan colisiones.

Inicialmente lo que se hacía era:

- Cámara: Se actualiza la posición en eje x e y, reposicionando en caso de que sobrepase algún límite.
- Jugador: Se actualiza la posición en eje x e y, se reposiciona en caso de que sobrepase algún límite del mundo o en caso de que colisione contra un Tile.
- Tiles: No es necesario, son estáticos.

La parte en que se calculaban las colisiones pasó por diversas fases:

1. Teniendo la posición original y la final, se calculaba el área de colisión. Este área de colisión era de dos tipos, si el movimiento solo era en el eje x o y, entonces el área de colisión eran los cuadrados de la posición final más el espacio entre este cuadrado y el cuadrado de la posición original, sin contar el cuadrado de la posición original, es decir, un rectángulo. En caso de que el movimiento fuese en diagonal (movimiento tanto en el eje x como en el y), entonces el área de colisión era el cuadrado de la posición final más el área recorrida desde el cuadrado de la posición original hasta el cuadrado de la posición final, es decir, una especie de fecha o hexágono irregular. Si había otra Entidad en esas areas de colisión, para conseguir la nueva posición de la Entidad se tenía que utilizar ecuaciones de recta en el caso de movimientos en diagonal (para el caso totalmente horizontal o vertical era más sencillo) y reposicionar a una posición lo más cercana a la posición final hipotética (sin colisiones) pero que no provoque ninguna colisión.
2. No se llegó a implementarlo del todo el paso anterior, pero en una sesión con el tutor me dijo que  si es el sistema de juego es suficientemente rápido con mirar la posición final era suficiente. Así que el diseño de las colisiones pasó a consistir en utilizar solo la posición final  y la velocidad del Jugador. La velocidad del jugador se utiliza para saber como eran las colisiones en el sentido de que, como los Tiles son estáticos, si por ejemplo el Jugador se mueve en el eje x e y positivamente (arriba a la derecha), entonces todas las colisiones vendrán por esa dirección. Aquí aun se continuaba utilizando ecuaciones de la recta para conseguir la nueva posición válida.
3. Después de implementar la parte de colisiones de la segunda fase (el punto anterior a este) cuando el Jugador tocaba el suelo, no podía deslizarse. Esto se debía a que había una gravedad en el mundo así que, a no ser que el Jugador estuviese saltando, siempre tenía una velocidad en y negativa y por lo tanto, aunque se le añadiese una velocidad en x, no se movía porqué siempre colisionaba con el Tile que tenía debajo. Para solucionar esto se decidió calcular las colisiones solo en x y luego solo en y, para ello:
    1. Actualización de posición de Jugador utilizando la velocidad en x.
    2. Comprobaciones con los limites izquierdo y derecho del mundo.
    3. Comprobacion de colisión con todas las Entidades Tiles, solo se recalcula la posición en x no en y. Para ello se comprueba la bounding box del Jugador y del Tile,y si hay intersección y dependiendo de la dirección a la que vaya el Jugador se hará un tipo de reposición u otro. Por ejemplo, si el Jugador va hacia la derecha y tiene una colisión, entonces su nueva posición será la posición del Tile menos la anchura del Jugador, quedando el Jugador pegado a la izquierda del Tile.

Y luego hacer lo mismo para y, solo que con los límites superior e inferior del mundo y recalculando en el eje y. Este motor una vez implementado ya funcionaba correctamente.

### 5.1.4. Motor gráfico

El motor gráfico es muy simple y aunque no es muy ortodoxo funciona bien. La práctica más común es, una vez se tienen los datos del mundo como el tamaño y los Tiles, se pintan todos los Tiles en una imagen tan grande como el mundo. Luego en cada iteración se recorta una parte de dicha imagen (la que está dentro de la bounding box de la Cámara) y se dibujan las Entidades que estan dentro de la Camara en dicha imagen. Luego esa imagen se muestra por pantalla.

En lugar de usar ese algoritmo en que los Tiles se consideran de manera distinta, el algoritmo usado considera a las Entidades Tiles como si fuera cualquier tipo de Entidad.

El proceso es:

- Para cada Entidad, comprobamos la intersección con la bounding box de la Camara:
  - Si no está dentro de la Camara, no se dibuja.
  - Si está parcial o totalmente dentro, se dibuja.
- Para dibujar una Entidad no solo se necesitará la bounding box que ocupará en la pantalla, si no también la bounding box para usar solo un fragmento de la imagen original (es por Direct2D). Para ello el algoritmo funciona de la siguiente manera:
  - Primero en el eje y, se calcula a que altura intersecciona la Entidad con la Cámara, según el caso las bounding boxes serán::
    - Por arriba, hay inclusión parcial:
      - Bounding Box pantalla: El límite superior se encuentra en la parte superior de la pantalla. El limite inferior es la altura del fragmento de la Entidad que hay dentro de la Camara.
      - Bounding Box imagen Entidad: El limite superior se encuentra a una distancia igual a la altura del fragmento de la Entidad que hay fuera de la Camara. El limite inferior es la parte inferior de la imagen.
    - Centrado, hay inclusión total:
      - Bounding Box pantalla: Coincide con la posición de la Entidad dentro de la Cámara.
      - BoundingBox imagen Entidad: Toda la imagen.
    - Por abajo, hay inclusión parcial.
      - BoundingBox pantalla: El límite superior es la distancia que hay desde el limite superior de la Camara hasta el limite superior de la Entidad. El limite inferior es la parte inferior de la pantalla.
      - BoundingBox imagen Entidad: El límite superior es la parte superior de la imagen. El limite inferior es la altura del fragmento de la Entidad que hay dentro de la Camara.
  - Luego para el eje x es análogo y se puede ver en la figura 12, a continuación.
  - Todas las cajas conseguidas, las tenemos en coordenadas de mundo, solo hace falta multiplicar las posiciones por *convWorldToScreen* para tener coordenadas de pantalla.

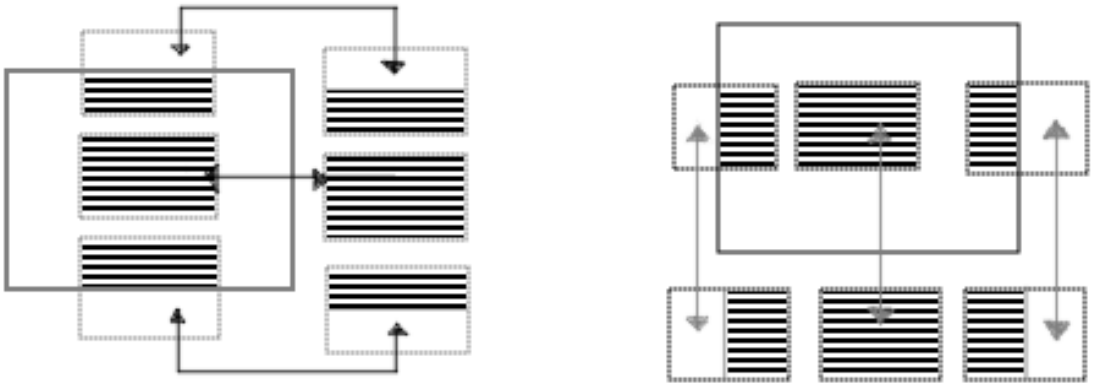

*Figura 12: Imagen mostrando como se generan las bounding boxes de pantalla y de imagen, primero para el eje y y luego para el eje x.*

El orden no obstante es, primero dibujar los Tiles, luego dibujar el Jugador y luego dibujar las otras Entidades (la Camara no se dibuja).

#### 5.1.5. El editor y los mecanismos necesarios para pasar datos del editor al engine

El editor trabaja sobre un modelo de ECS aun más simple, seguimos teniendo los tipos especiales de Entidad (Cámara, Tiles y Jugador) además de las Entidades normales. La diferencia está en que para cada Entidad solo se tienen las siguientes Componentes:

- positionComponent
- sizeComponent
- renderComponent (excepto en el caso de la Cámara)

La razón de usar tan pocas Componentes es que no se necesitan más para el funcionamiento del editor.

La interfaz gráfica del editor, construida con XAML, consiste en:

- VisorMundo: Esta parte de la interfaz se usará para ver lo que contiene la Cámara dentro del juego. Dependiendo del modo en que esté el editor se podrá:
  - Añadir/quitar cualquier tipo de Entidad.
  - Mover la Cámara.
- PanelEnsamblajes: Estará lleno de botones:
  - Un botón por cada Ensamblaje. Cada botón de este tipo tendrá una imagen del Ensamblaje que crea. Una vez se ha hecho click, al hacer click en el VisorMundo se creará una Entidad siguiendo la estructura de dicho Ensamblaje centrada en el punto clickado.
  - Un botón que sirve para ponerse en modo "mover cámara": Después de hacer click en este botón, al clickar y mantener en el VisorMundo podremos mover la cámara de posición según se mueva el ratón.
  - Un botón para eliminar Entidades: Después de hacer click en este botón, si se hace click en la pantalla, en una parte donde haya una Entidad, se destruirá.


*Figura 13: Imagen del editor con señalizaciones. En el caso del botón de ensamblaje, solo son algunos ejemplos, todos los recuadros que no sean ni "mover camara" ni "eliminar ensamblaje" (ni el botón "tiles")dentro de "panelEnsamblajes" es un "botón ensamblaje".*

Además habrán otros tres botones en la barra de comandos inferior (es un elemento de XAML, al hacer click derecho aparecerá por la parte de abajo de la pantalla):

- Uno para crear un nuevo mundo eligiendo su tamaño.
- Otro para guardar el mundo que se está trabajando en el editor.
- Otro para cargar un mundo desde un fichero de texto.


*Figura 14: Otra imagen del editor, esta vez mostrando los botones para crear, guardar y cargar un mundo.*

Guardar y cargar un mundo consiste en leer un fichero ASCII. Dicho fichero tiene la siguiente forma:

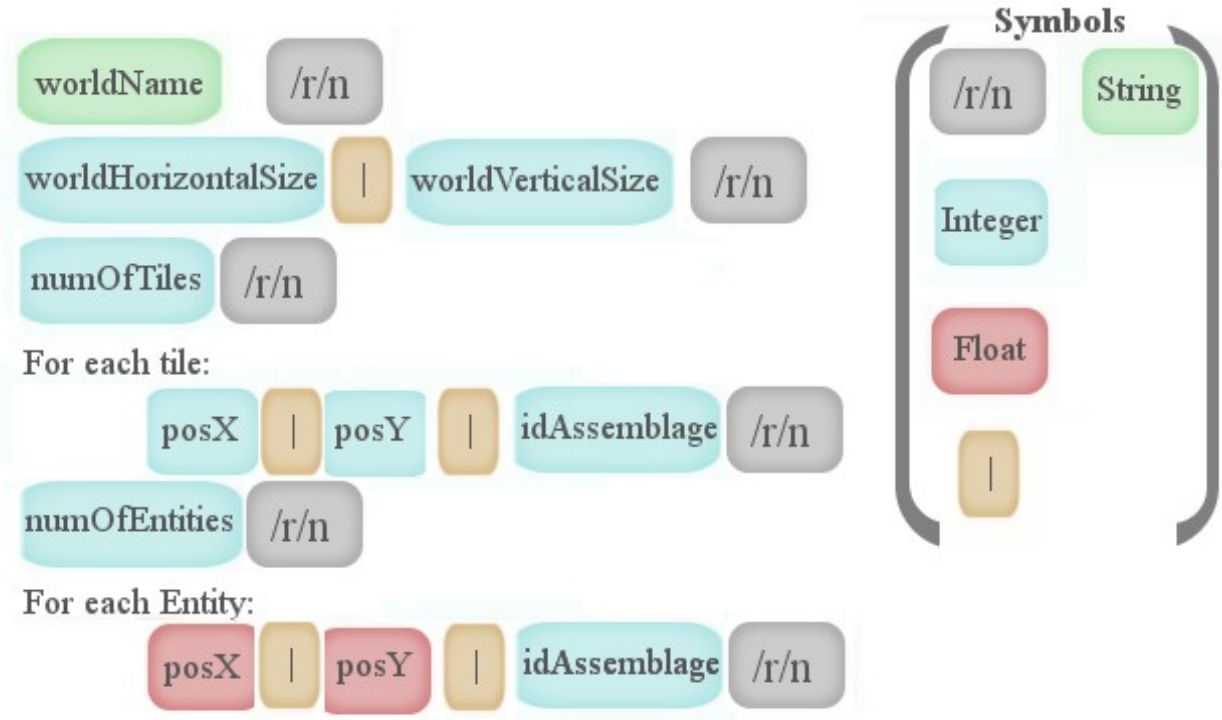


*Figura 15: Imagen representativa del formato de un mundo en un fichero .txt.*

Algunos aclaraciones:

- /r/n: Es un salto de linea en Windows, porqué /r es "retroceder el rodillo" y /n es "salto de linea". Tiene que ver con como funcionaban las máquinas de escribir.
- |: es ese mismo símbolo.
- En Tiles, posX y posY se refieren a las posiciones de los tiles si estuviesen en forma matricial. Por ejemplo, (0,0) sería el tile más arriba a la izquierda posible.
- En Entidades estamos contando al jugador como Entidad.
- En Entidades, posX y posY son valores de coordenadas de mundo.

Mediante este fichero, también se hace el traspaso de información del editor al engine, es una simple lectura de fichero, inicialización de Entidades y variables de mundo.

## 5.2. Fase II

### 5.2.1. ECS final

El ECS inicial era suficientemente bueno para un mundo muy simple, pero en el momento en que se empiezan a añadir Entidades que no son Jugador o Tiles uno se da cuenta que se requiere un análisis más profundo sobre el juego que se quiere hacer.

En esta fase busqué más sobre el tema de como hacer un análisis y diseño formal para hacer un ECS y solo encontré un ejemplo sobre como hacer un juego copia de Bomberman; bueno, en realidad eran dos ejemplos pero en el segundo, de una fuente más importante [9], se comenta que el primer ejemplo no es muy correcto así que solo seguí el segundo. El análisis y diseño explicados en el post son básicamente:

1. Recopilar comportamientos y datos del juego, siendo comportamientos del estilo "capacidad para crear una bola de fuego cada Y unidades de tiempo" y datos algo como "posición de la Entidad en el mundo".
2. Separar los requerimientos conseguidos en el paso anterior en Componentes. Para el comportamiento "capacidad para crear una bola de fuego cada Y unidades de tiempo" se puede crear el componente fireballGeneratorComponent que solo tenga 2 atributos, uno el tiempo desde el último lanzamiento y otro el tiempo máximo.
3. Disminuir número de Componentes en caso de que hayan duplicados. Esto ocurre si en el primer paso ya se tienen datos del juego como "posición de una entidad" y "posición de un tile", en este caso son el mismo Componente.

El análisis y diseño para esta fase del proyecto se han modificado un poco:

- En el primer punto se sigue teniendo comportamientos y datos pero en lugar de dividirse entre comportamientos y datos se dividen entre los Ensamblajes que se crearán. Para un juego sencillo en el que ya se tiene claro que Ensamblajes se quieren ya va bien, ya que al crear todos los Componentes que necesita un Ensamblaje ya se puede confirmar que funciona. Para un juego complejo sería más correcto el punto 1 del análisis original.
- El segundo y tercer punto se juntan en uno. Simplemente por cada dato o comportamiento de cada Ensamblaje se crean los Componentes necesarios para cumplirlo, y luego se repasa cada Ensamblaje para ver si estos nuevos Componentes cumplen con los datos o comportamientos de otros Ensamblajes.
- Esta forma de análisis y diseño además también tiene en cuenta los Procesadores. Los Componentes de por si solo son datos así que creo que es necesario que, a la vez que se están creando los Componentes, crear los Procesadores.

Ahora se explicarán estos pasos para este proyecto:

#### 5.2.1.1. Ensamblajes

Los Ensamblajes de este juego junto a sus propiedades son:

- Jugador (tipo especial de Entidad):
  - Puede ser controlada por el jugador (persona real).
  - Posición en el mundo.
  - Tamaño.
  - Puede mover más o menos rápido.
  - Puede saltar.
  - Al chocar contra otra Entidad puede reposicionarse (colisión), morir,...
  - Imagen a mostrar durante el render.

 *Figura 16: Imagen del jugador.*

- Tile (tipo especial de Entidad):
  - Posición en el mundo.
  - Tamaño único para todos los Tiles.
  - Es estático.
  - Al colisionar contra otras Entidades, en el caso de ser una Entidad Jugador el Jugador tendrá que reposicionarse pera evitar que haya una intersección en las bounding box.

- Imagen a mostrar durante el render.

*Figura 17: Imagen de tileset original, son los 9 tiles básicos.*

*Figura 18: Imagen de tile esquina abajo-derecha.*

*Figura 19: Imagen de tile esquina arriba-izquierda.*

*Figura 20: Imagen de tile esquina arriba-derecha.*

*Figura 21: Imagen de tile esquina abajo-izquierda.*

- Pincho
  - Posición en el mundo.
  - Tamaño.
  - Estático.
  - Al colisionar con un Tile o con los límites del mundo muere.
  - Al colisionar con el Jugador, el Jugador muere.
  - Imagen a mostrar durante el render.

*Figura 22: Imagen de pincho.*

- Llave
  - Posición en el mundo.
  - Tamaño.
  - Estático.
  - Al colisionar con el Jugador, la llave muere y el número de llaves recogidas por el Jugador aumenta.
  - Imagen a mostrar durante el render.

*Figura 23: Imagen de llave.*

- Puerta
  - Posición en el mundo.
  - Tamaño.
  - Estático.
  - Al colisionar con el Jugador:
    - Si el número de llaves recogidas es superior a 0:
      - Esta puerta muere.

        - Se reduce en 1 el número de llaves recogidas.
    - En caso contrario:
        - Se produce un reposicionamiento para que la bounding box del Jugador no interseccione a la de la Puerta.

*Figura 24: Imagen de puerta.*

- Bola de fuego
    - Posición en el mundo.
    - Tamaño.
    - Móvil, en un único sentido y dirección.
    - Al colisionar con el Jugador, el Jugador muere.
    - Al colisionar con un Tile, la bola de fuego muere.
    - Al colisionar con los límites del mundo, la bola de fuego muere.
    - Imágenes a mostrar durante el render, dependerá de en que sentido vaya.

*Figura 25: Imagen de bola de fuego hacia la izquierda.*

*Figura 26: Imagen de bola de fuego hacia la derecha.*

- Lanza Bolas de fuego
    - Posición en el mundo.
    - Tamaño.
    - Estático.
    - Crea bolas de fuego en la dirección que apunta este lanza bolas de fuego, con una cierta velocidad y de manera periódica.
    - Imágenes a mostrar durante el render, dependerá de en que sentido vaya.

*Figura 27: Imagen de lanza bolas de fuego hacia la izquierda.*

*Figura 28: Imagen de lanza bolas de fuego hacia la derecha.*

- Enemigo
    - Posición en el mundo.
    - Tamaño.
    - Móvil, se moverá en una misma dirección y sentido hasta que colisione contra un Tile, entonces cambiará de sentido.
    - Le afecta la gravedad.
    - Al colisionar con el Jugador, el Jugador muere.
    - Al colisionar con pincho o bola de fuego, morirá este enemigo.
    - Imágenes a mostrar durante el render, dependerá de en que sentido vaya.

*Figura 29: Imagen de enemigo hacia la izquierda.*

*Figura 30: Imagen de enemigo hacia la derecha.*

- Final

- Posición en el mundo.
- Tamaño.
- Estático.
- Al colisionar con Jugador, Jugador gana.
- Imagen a mostrar durante el render.

 *Figura 31: Imagen de final.*

#### 5.2.1.2. Componentes

Se pueden utilizar todos los Componentes desarrollados en el paso previo de análisis y desarrollo, así que solo es necesario crear unos nuevos que cumplan con los comportamientos y datos del paso anterior.

El Jugador y los Tiles ya se pueden construir con los Componentes actuales, así que se pasa a Pincho. Pincho se comporta de manera distinta al colisionar con un tipo u otro de Ensamblaje así que se necesita un Componente que responda ante esto.

- CollisionBehaviourComponent: tiene los siguientes valores posibles:
  - nothing: No hace nada al colisionar.
  - staticKiller: Mata al Jugador y a las Entidades con collisionBehaviourComponent enemy.
  - volatile: Mata al Jugador pero es matado por los Tiles, también mata a Entidad con collisionBehaviourComponent enemy.
  - CollectKey: Es recogida por el Jugador.
  - ClosedDoor: En caso de haber más de 0 llaves recogidas, esta Entidad muere y se decrementa en 1 el número de llaves recogidas, en caso contrario se produce un reposicionamiento en Jugadores y Entidades con collisionBehaviourComponent enemy y provoca la muerte en Entidades con collisionBehaviourComponent volatile.
  - Enemy: En caso de colisionar con Jugador, Jugador muerte. En caso de colisionar con Entity con collisionBehaviourComponent staticKiller o volatile, esta Entidad muere. En caso de colisionar con Tile, esta Entidad se reposiciona para que las bounding boxes no se intersequen.
  - End: Da la victoria al Jugador.

Con esto Pincho, Llave, Puerta y Final ya se cumplen. El siguiente es bola de fuego, en el que se especifica que dependiendo de en que dirección vaya tiene que mostrar una imagen u otra. Para resolver esto se necesitan dos nuevos Componentes:

- DirectionComponent:
  - xWay: Sentido de la Entidad en el eje x, solo tiene dos valores posibles: izquierda y derecha.
- AnimationComponent:
  - imageIdLeft: Imagen a mostrar al ir hacia la izquierda.
  - ImageIdRight: Imagen a mostrar al ir hacia la derecha.

Bola de fuego ya está completa. Con el lanza bolas de fuego solo se necesita añadir un Componente para poder generar bolas de fuego;

- GenerateFireBallComponent:
  - timer: cuenta el tiempo que ha pasado desde la última vez que se ha lanzado una bola de fuego.
  - MaxTime: es el tiempo necesario para generar una bola de fuego.

Lanza bolas de fuego completo. Para enemigo se necesitan dos componentes, una que se encargue de que la gravedad le afecte y otra que se encargue de cambiar de sentido cuando se reposicione debido a una colisión con un Tile.

- GravityComponent:
  - gravityEffect: La magnitud con la que afecta la gravedad a esta Entidad.

Y ahora una explicación previa al porqué de la segunda Componente. Se debe a que, que una Entidad cambie su comportamiento ante una cierta situación no parece algo que se puedan incluir , por ejemplo, en CollisionBehaviourComponent; pertenece más a algo del tipo Inteligencia Artificial.

- WalkerIAComponent:
  - leftSideObstructed: Indica si la Entidad ha sido obstruida por su lado izquierdo.
  - rightSideObstructed: Indica si la Entidad ha sido obstruida por su lado derecho.

Aunque ya se esté con los Componentes, debido a que estamos usando Entidades especiales tenemos algunos atributos que son propios solo de estas Entidades.

- Jugador:
  - playerIsOnLand: Indica si el Jugador está en el suelo o no, se utiliza para saber si puede saltar.
  - NumOfLosses: Indica el número de veces que ha muerto.
  - NumOfVictories: Indica el número de veces que ha ganado.
  - NumOfKeys: Número de llaves que el Jugador ha recogido.

#### 5.2.1.3. Procesadores

Queda por especificar los Procesadores usados y de que se encargarán.

- InputProcessor:
  - Recibe input (clicks de ratón, teclas presionadas,...) como entrada.
    - Si en el input la "tecla flecha arriba" ha sido apretada y playerIsOnLand indica que está en tierra, entonces se modifica speedComponent del Jugador para que aumente su velocidad vertical.
    - Si en el input la "tecla flecha derecha" ha sido apretada la velocidad horizontal de speedComponent adquiere un valor positivo.
    - Si en el input la "tecla flecha izquierda" ha sido apretada la velocidad horizontal de speedComponent adquiere un valor negativo.
- PhysicsProcessor y BoxOverlapingProcessor se explicarán en la siguiente sección.
- DirectionProcessor:
  - Para cada Entidad:
    - Si su velocidad es positiva entonces xWay de directionComponent cambia a derecha.
    - Si su velocidad es negativa entonces xWay de directionComponent cambia a la

izquierda.

- AnimationProcessor:
  - Para cada Entidad:
    - Si la directionComponent es derecha entonces se cambia el valor de imageId de renderComponent por el valor imageIdRight de AnimationComponent.
    - Si la directionComponent es izquierda entonces se cambia el valor de imageId de renderComponent por el valor imageIdLeft de AnimationComponent.
- FireballGeneratorProcessor:
  - Para cada Entidad:
    - Se incrementa el atributo timer de generateFireBallComponent con "el valor de tiempo transcrito entre el actual frame de ejecución y el actual". En caso de que se supere el valor de maxTime, entonces se le resta maxTime y se genera una nueva bola de fuego a una cierta distancia y utilizando de referencia la dirección de la actual Entidad. Por ejemplo, si la Entidad que genera la bola apunta hacia la izquierda la bola creada aparecerá a la izquierda de la Entidad y moviéndose hacia la izquierda.
- IAWalkerProcessor:
  - Para cada Entidad:
    - Si tiene leftSideObstructed de su WalkerIAComponent y va en dirección izquierda, o si tiene rightSideObstructed y va en dirección derecha, cambia su dirección, velocidad de movimiento a la contraria que la actual y desactiva los flags de aviso (componente IAWalker).

Se sigue utilizando el mismo RenderProcessor que en la fase 1.

### 5.2.2. Motor físico final y procesador de colisiones

Esta parte explica el diseño detrás de los Procesadores physicsProcessors y boxOverlapEffectProcessor. La razón de explicarlos a la vez es que en algunos momentos fueron un mismo Processor, pero al final acabaron siendo dos Procesadores distintos en los que básicamente:

- physicProcessor: Para toda Entidad contra toda Entidad, si hay colisión y es de tipo "reposicionamiento de Entidad", se resposicionan las Entidades.
- BoxOverlapEffectProcessor: Para toda Entidad con toda Entidad, si hay colisión y es un tipo distinto a "reposicionamiento de Entidad" entonces se resuelven los efectos.

La razón de porqué se ha hecho esta separaciñon es por temas de implementación, así que se explicarán en dicha parte '6.2. Implementación del motor físico'.

#### 5.2.2.1. PhysicProcessor

En esta sección, cada vez que se menciona una "variable temporal" se está refiriendo al tiempo que ha pasado entre el último frame de ejecución y el actual. Se ha utilizado un FlowChart para explicarlo, que en este caso es una gran ayuda debido a que simplifica mucho el código.

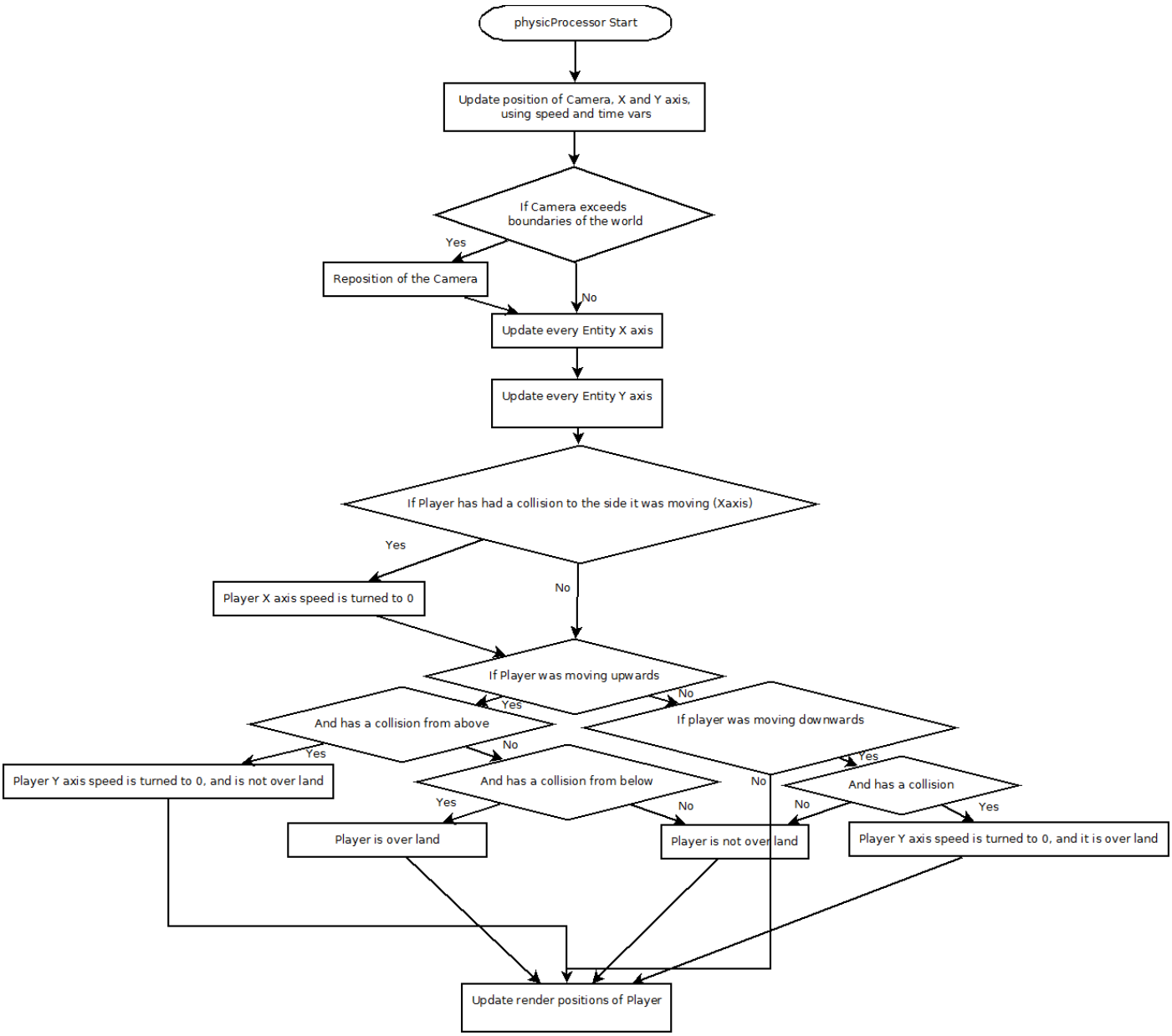


*Figura 32: Flowchart mostrando el algoritmo del PhysicProcessor.*

Este sería el diagrama básico, hay que tener en cuenta que cuando pone "Player is over land" se refiere a que se está modificando la variable playerIsOnLand.

Ahora queda por explicar "Update every Entity X axis" y "Update every Entity Y axis". Son lo mismo solo que uno lo hace todo para el eje x y el otro para el eje y:

- Se actualiza la posición del Jugador y de las demás Entidades con sus velocidades y la gravedad del mundo, solo en el eje x o y.
- Se hacen los reposicionamientos apropiados para que no se salgan fuera de los bordes del mundo.
- Para el Player:
  - Por cada Tile:

        - Si hay una colisión, hay un reposicionamiento del Player.
- Para cada Entidad con collisionBehaviourComponent enemy:
    - Por cada Tile:
        - Si hay una colisión, hay un reposicionamiento de la Entidad y además se pone uno de los dos atributos de IaWalkerComponent a verdadero, dependiendo de por donde venga la colisión.
- Para el Player, si no puede abrir puerta (el número de llaves es 0):
    - Por cada Entidad con collisionBehaviourComponent clossedDoor:
        - Si hay una colisión, hay un reposicionamiento del Player.
- Para cada Entidad con collisionBehaviourComponent enemy:
    - Por cada Entidad con collisionBehaviourComponent clossedDoor o enemy:
        - Si hay una colisión, hay un reposicionamiento de la primera Entidad".

Es importante mencionar algo, el algoritmo que se usa para las colisiones contra Tiles (elementos estáticos) es distinto al que se usa para colisiones contra Entidades (elementos móviles). Esto se debe a que si se conoce que uno de los dos miembros de una colisión es estático y el otro se está moviendo en una dirección, la colisión solo puede haber venido en esa dirección. En cambio cuando los dos miembros en una colisión son móviles, el algoritmo anterior no funciona.

#### 5.2.2.1.1. Algoritmo entre Entidades móviles y Entidades estáticas

El algoritmo consiste en calcular el rectángulo de colisión de la Entidad móvil. En este caso como las únicas Entidades estáticas eran Tiles, y todas tienen el mismo tamaño:

- rectX = posición de la Entidad en X  - width de Tile
- rectY = posición de la Entidad en Y - height de Tile
- rectWidth = width de la Entidad + width de Tile
- rectHeight = height de la Entidad + height de Tile

Ahora si estamos usando el algoritmo para las colisiones en X y además el movimiento de la Entidad móvil es hacia la derecha, las colisiones solo pueden aparecer por la derecha, así que por cada Tile:

- Calcular si hay colisión, para ello se coge la posición del Tile con el que se está comprobando si hay una colisión, si está entre rectX y rectX+rectWidth en el eje X, y entre rectY y rectY+rectHeight en el eje Y, hay colisión.
- Si hay colisión se cambia la posición de la Entidad en X por "la posición del Tile en X menos la anchura de la Entidad".
- Se actualiza rectX debido al cambio de la posición en X de la Entidad, su nuevo valor será "la posición de la Entidad en X menos la anchura del Tile".

Si es el caso de que el movimiento de la Entidad móvil es hacia la izquierda entonces lo único que cambia es la nueva posición que se da en caso de que haya colisión. En este caso la posición X de la Entidad (el atributo posX de su positionComponent) pasa a ser "la posición X del Tile más la anchura del Tile".

Para el caso del eje Y es análogo, solo que en vez de actualizar rectX se actualiza rectY.

#### 5.2.2.1.2. Algoritmo entre dos Entidades móviles

En estos casos, no se puede utilizar la dirección del movimiento para saber que clase de colisión es, al menos no como se hacia en el algoritmo anterior. Podrían usarse ambas velocidades para determinar el tipo de colisión, pero hay una forma mucho más sencilla de hacerlo.

Este algoritmo se aprovecha al máximo del hecho de que entre frame de ejecución y frame de ejecución ha pasado un corto periodo de tiempo y por lo tanto las diferencias de posiciones entre frame y frame son mínimas. Con aprovecharse, se refiere a que si el periodo de tiempo entre frame y frame es mínimo entonces las colisiones entre Entidades tendrán una zona de intersección mínima, con lo cual se puede dar por hecho que la colisión ha ocurrido por dicha zona de intersección y entonces lo único que queda por hacer es alejar las Entidades por dicha zona, hasta que no haya intersección de bounding boxes. La clave del algoritmo está en entender que a partir de unos datos insuficientes se está asumiendo una información: como es la colisión.

Para este ejemplo se utilizarán los términos Entidad A y Entidad B para denotar a dos Entidades. Para la fase en que se resuelven colisiones en el eje X el algoritmo sería:

- Se crea de nuevo el rectángulo de colisión:
  - rectX = posición de la Entidad A en X
  - rectY = posición de la Entidad A en Y
  - rectWidth = width de la Entidad A
  - rectHeight = height de la Entidad A
- Para cada Entidad B con la que se quiere comprobar si hay colisión:
  - Si posición X de Entidad B está entre (rectX – width de Entidad B) y (rectX + rectWidth) y la posición Y de Entidad B está entre (rectY – height de Entidad B) y (rectY + rectHeight), hay colisión.
  - Entonces se buscan las dos posiciones posibles de reposicionamiento:
    - A la izquierda de la Entidad B: posición en X de Entidad B – width de Entidad A.
    - A la derecha de la Entidad B: posición de X de Enditad B + width de Entidad B.
  - Una vez se tienen ambas, se calcula la distancia que hay desde la posición actual de Entidad A, rectX, a ambas nuevas posiciones. Entidad A se mueve a aquella posición que requiera menos movimiento.
  - Actualizamos rectX a la nueva posición de Entidad A en X.

Para Y sería igual, solo que se haría el reposicionamiento en Y, con las posiciones en Y.

### 5.2.2.2. BoxOverlapEffectProcessor

Similar a physicProcessor a la hora de encontrar colisiones, pero no hay reposicionamientos. En lugar de eso se aplican efectos. Además, aquí no es necesario calcular colisiones en X e Y, ya que no hay ningún tipo de reposicionamiento.

- Para cada Entidad con collisionBehaviourComponent volatile:
  - Para cada Tile:
    - Si hay colisión, la Entidad muere.
- Para el Jugador:
  - Para cada Entidad con collisionBehaviourComponent volatile, enemy o staticKiller:
    - Si hay colisión, Jugador muere.
  - Para cada Entidad con collisionBehaviourComponent collectKey:

        - Si hay colisión:
            - Se incrementa en uno el número de llaves recogidas.
            - Se destruye la Entidad con collisionBehaviourComponent collectKey.
    - Para cada Entidad con collisionBehaviourComponent closedDoor, mientras el número de llaves recogidas sea superior a 0:
        - Si hay colisión:
            - Se destruye la Entidad con collisionBehaviourComponent closedDoor.
            - Se decrementa el número de llaves recogidas en una.
    - Para cada Entidad con collision BehaviourComponent end:
        - Si hay colisión, el Jugador gana.
- Para cada Entidad con collisionBehaviourComponent enemy:
    - Para cada Entidad con collisionBehaviourComponent volatile o staticKiller:
        - Si hay colisión,  Entidad con collisionBehaviourComponent enemy muere.

## 6. Implementación

### 6.1. Implementación del ECS

La implementación del ECS, el nucleo del engine y del editor, se encuentra en los ficheros:

- Component.h : Aquí se encuentran las especificaciones de los Componentes.
- Scene2DModule.h y Scene2DModule.cpp: Aquí esta todo el sistema ECS.

Ya se ha explicado que es un ECS (Entidades, Componentes, Procesadores y Ensamblajes) y cual es el problema con la memoria cache, ya se puede hablar de como implementar un ECS.

Para comenzar, las Componentes son structs con variables de distintos tipos, los Procesadores son funciones que trabajan sobre las Componentes y los Ensamblajes son funciones que inicializan Componentes para una Entidad dada.

La implementación de un ECS consiste en como posicionar y reposicionar los datos (Componentes) a nivel de memoria RAM para que a la hora de ser procesados (por los Procesadores) se aproveche la cache al máximo y el sistema, en su totalidad, sea rápido. Ya hay algunas implementaciones más o menos complejas de ECS que intentan acercarse a ese objetivo, y para este proyecto se ha escogido una de ellas [2].

La implementación de ECS escogida para este proyecto consiste en tener todas las instancias de un tipo de Componente en una misma array, donde cada posición de esta array denota "esta es la instancia de la Componente X que forma parte de la Entidad Y", donde X es para todo tipo de Componente y Y es para la posición dentro del array. Es decir, la instancia de la Componente positionComponent en la posición del array 5 forma parte de la Entidad 5. Así que las Entidades en esta implementación son los identificadores de posición de los arrays.

Las ventajas de esta implementación son:

- Es la implementación más sencilla. Solo hay que usar arrays.
- No requiere que se hagan movimientos de datos en memoria para evitar los huecos. En cierta manera evitar fragmentación de memoria, pero aquí no es importante porqué las posiciones de las arrays que no se usan en un momento solo podrán ser ocupadas por algo del mismo tamaño y de la misma Entidad, así que no es un problema.

Las desventajas son:

- Actualmente se da por hecho que todas las Entidades tienen una instancia de cada tipo de Componente. Esto es así porqué no tenemos ningún mecanismo para saber si una Entidad que existe tiene o no una Componente. Para arreglarlo se tendría que usar una implementación más compleja de ECS.
- Debido a la desventaja anterior ocurren cosas como "actualizar la posición de una Entidad que no tiene velocidad". En este caso es lo mismo tener velocidad 0 que no tener. La solución que se ha tomado es la de considerar que todas las Entidades tienen todos los Componentes, por lo tanto Entidades con velocidades nulas siguen sumando movimiento, por ejemplo.
- Otra gran desventaja: como eliminar una Entidad. Para resolver esta parte se ha optado por crear un array de valores bool para las Entidades genéricas, exists. En este array hay un *true* en una posición cuando la Entidad con identificador dicha posición, existe. *False* en caso contrario. Eso llevo algunas modificaciones:

- Creación de Entidades: Ahora en lugar de incrementar el número de Entidades existentes y retornar su identificador, se busca si alguna Entidad ha dejado de existes, si es el caso se devuelve el identificador de dicha Entidad.
- Destrucción de Entidades: Se inicializan a 0 o false todas las variables de todas las Componentes de una Entidad, incluido exists.

  En realidad al principio con poner a 0 y false las Componentes de una Entidad era suficiente, sin utilizar exists. En unas pruebas que se hicieron se vió que si había algo que generaba nuevas Entidades in-game (lanza bolas de fuego por ej.) entonces el número de Entidades era demasiado elevado, hasta el punto que empezaban a haber fallos debido a que no cambian más.

### 6.2. Implementación del motor físico

Lo único que hay que comentar es que se hacen muchos bucles sobre las Entidades generales, que se podrían hacer únicamente con un bucle y con más operaciones condicionales. La razón de esto es que un menor uso de operaciones condicionales dentro de bucles puede mejorar el rendimiento de un bucle debido a la falta de "branch missprediction". Cuando se pasa por una operación condicional como un "if" en tiempo de ejecución se va a un bloque de código o a otro; branch missprediction es cuando se carga en cache uno de esos bloques antes de llegar a dicha operación condicional y una vez se pasa por dicha operación condicional se necesita el otro bloque de código, para cargar dicho bloque puede haber un delay. Esto depende mucho del procesador y de la forma en que llena la cache de instrucciones.

La otra gran razón es que hay unos fallos en el diseño que se explicarán en "8. Mejoras futuras", y en un futuro en que no hubiesen dichos fallos de diseño, estos diversos bucles serían mucho más correctos ya que se tendrían muchas más Componentes sobre las que iterar.

### 6.3. Implementación de los estados del engine

El engine tiene 4 estados:

- sTart: Estado inicial, aparece un imagen "Start" en la pantalla y las Entidades no se actualizan (todo está quieto).
- Playing: Estado usual del juego, las Entidades se actualizan.
- GameOver: Un estado final del juego, se muestra una imagen "GameOver" y las Entidades no se actualizan.
- Victory: Un estado final del juego, se muestra una imagen de "Victory" y las Entidades no se actualizan.

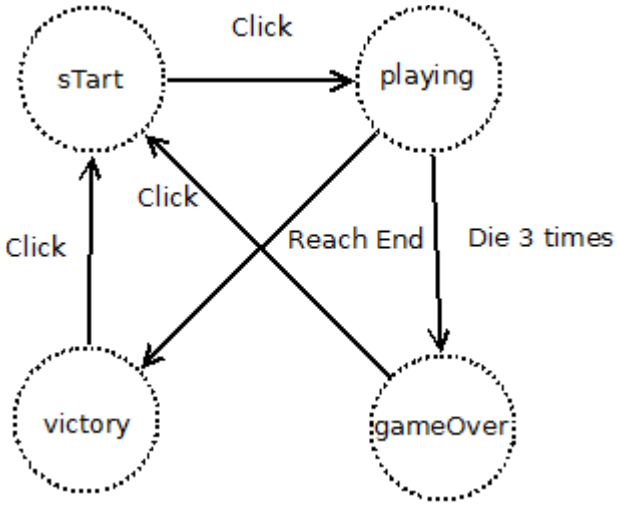


*Figura 33: Imagen de la maquina de estados del engine.*

Unas aclaraciones:

- Click: Un click del mouse del ratón.
- Reach End: Que el Jugador llegue a una Entidades Final (el letrero que pone End).
- Die 3 times: Cada vez que el Jugador colisiona con un pincho, un enemigo o una bola de fuego, retorna al punto de partida (muerte). Si muere una 3ª vez, se pierde la partida.

## 7. Resultados

Los resultados son:

- Un engine que:
  - Sigue una arquitectura ECS.
  - Puede leer y generar mundos.
  - Tiene un motor físico.
  - Tiene un motor gráfico.
  - Recibe input por parte del usuario y le permite interactuar con el mundo.
- Un editor que:
  - Permite crear, guardar y cargar mundos.
  - Permite añadir y eliminar Entidades de un mundo con el que se trabaja.
  - Se puede mover la Cámara del mundo con el ratón.
- Un juego:
  - En el que se controla una pelota.
  - Hay pinchos, bolas de fuego y enemigos que pueden matar a la pelota.
  - Hay cañones que lanzan bolas de fuego.
  - Hay llaves que se pueden recoger.
  - Hay puertas que se pueden abrir con llaves.
  - Hay metas a las que al colisionar con ellas se acaba la partida.

*Figura 34: Imagen de lo que se visualiza al jugar una partida.*

Además, el engine no tiene problemas de rendimiento.

## 8. Mejoras futuras

Hay muchas mejoras posibles para este proyecto, aquí algunas de ellas.

### 8.1. Mejora de implementación de ECS

La actual implementación de ECS es sencilla y rápida de implementar, pero tiene el gran problema de que no se puede designar que Componentes tiene una Entidad. Por defecto todas las Entidades tienen todas las Componentes, lo que provoca que el sistema se vea obligado a hacer operaciones de más o a tener que incluir comprobaciones extras para que todo funcione.

En principio hay dos caminos de mejora:

1. Algún mecanismo para saber que Entidades tienen que Componentes y que no suponga una perdida de velocidad de procesamiento.
2. Usar implementaciones más complejas de ECS.

La primera solución consistiría en usar un array por cada tipo de Componente, con lo cual ahora se contaría con una array que contendría todas las instancias de todas las Componentes, como antes, y otro array con otro tipo de información. Este segundo array tendría los identificadores de las Entidades que poseen una instancia del tipo de Componente, o lo que es lo mismo, serían las posiciones del otro array con Componentes inicializadas o válidas.

Esta solución en si ya introduce problemas del tipo:

- Que ocurre cuando se borra una Componente? Entonces supongo que se tiene que dejar un hueco en la array de índices. Para arreglar esto que se podría hacer?
  - Dejar un número negativo, será el identificador de una Entidad ficticia sobre la que se harán operaciones pero que no afectará al sistema. Entonces cuando se crea un nuevo Componente de un tipo donde sus array de índices tiene un hueco, en lugar de poner el nuevo índice al final podría ponerse donde se encuentre este identificador ficticio.
  - Correr todos los valores a la derecha del hueco una posición a la izquierda. Parece muy costoso, pero en algunos blogs de ECS, como el de Adam Martin [1], se dice que operaciones contrarias al sentido común como esta pueden ser una buena idea.
  - En lugar de usar un array, usar una estructura basada en punteros de manera que cuando creamos un hueco lo único que hay que hacer es hacer que el valor anterior apunte al valor siguiente al del hueco, de esta manera desaparece el hueco. En caso de que no haya quedado claro, una *linked list*. Podría ser una solución, se intentaba no usar estructuras de este tipo para guardar Componentes por el tema de la cache, pero aquí puede que no sea un gran problema.
- Y cuando se borra una Entidad? Bueno esta es fácil, es borrar todas sus Componentes así que la respuesta aquí depende de la respuesta a la pregunta anterior.

La segunda solución, la de usar una implementación distinta de ECS, es más compleja pero a la larga sale mucho más a cuenta ya que no solo se está arreglando el problema que hay con la implementación actual si no que además se está mejorando la manera en que se aprovecha el sistema de la memoria caché.

Una mejora de implementación posible, también en [2] junto a la implementación usada, consiste en:

- Se usa una especie de "mega array" con todas las instancias de todos los tipos de Componentes.
  - Cada elemento de esta array es una Componente de algún tipo.
  - Todas las Componentes de una misma Entidad están juntas.
- Un array para saber en que offset empieza cada Entidad dentro de ese "mega array".
- Para cada tipo de Componente, un array donde en cada posición está la información de "para la Entidad X su Componente de este tipo está a una distancia Y de donde está representada la Entidad en el 'mega array'".

Sobre esta mejora hay dos puntos a comentar:

- El problema de "que ocurre cuando se pierde una Componente?" vuelve a aparecer. En el blog donde se habla de esta solución no se describe como solucionarlo, así que se requeriría un estudio más profundo.
- Además, en cierta manera las Entidades vuelven a estar juntas a nivel de memoria, algo que se supone que se estaba evitando ya que no se van a necesitar todos los datos de una Entidad a la vez, si no a ir iterando sobre todos los Componentes de un tipo (o conjunto de tipos) para cada Entidad, para luego pasar a Componentes posteriores. Con lo cual considero que esta solución podría ser incluso peor que la actual. Por otro lado no hay arrays desperdiciando espacio como en esta solución.
- El "mega array" supuestamente contiene más de un tipo de Componente, pero cada tipo puede tener un tamaño distinto. Para hacer esto posible, sin utilizar ningún tipo de puntero o tecnología similar, sería necesario crear un array de bytes y asignar un número distinto de bytes a cada tipo de Componente. No se si esto es algo aceptable.

### 8.2. Mejora de diseño de ECS

Esta mejora es solo posible una vez se haya mejora el ECS, se debe a que hacer un buen diseño de ECS implica una cantidad bastante elevada de Componentes y actualmente cada tipo de Componente necesita de un array gigante, lo cual es un coste muy elevado.

El gran fallo de diseño de este ECS es collisionBehaviourComponent. De este fallo era consciente cuando estaba haciendo el diseño, pero debido a la implementación actual de ECS era un fallo casi favorable, por eso se dejó. El fallo consiste en que se está cargando un solo tipo de Componente con muchos comportamientos o datos, es decir, dependiendo del valor exacto de este Componente su comportamiento de colisión cambia mucho, incluso cambia el tipo de algoritmo.

Por poner un ejemplo, hay un valor de collisionBehaviourComponente que es volatile y otro que es enemy. Volatile solo tiene que detectar si hay una instersección entre las bounding boxes de dos Entidades y luego hacer algo, enemy según con que otra Entidad está colisionando en algunos casos con calcular si hay intersección es suficiente,  en otros necesita saber como es la colisión para hacer un reposicionamiento.

Así que una mejora substancial sería partir collisionBehaviourComponent en distintos Componentes, por ejemplo:

- collectKeyComponent: Pasa a ser un Componente con un solo valor booleano, true significaría que se recoge al haber colisionado con el Jugador.
- CollisionTypeComponent: Solo puede tener 3 valores:

    - Nothing: No hace nada.
    - Static: Produce colisiones, pero esta Entidad no se mueve.
    - Movable: Produce colisiones y se mueve.
- DamageOnTouchComponent: Para empezar podría ser un Componente booleano, donde si está a true entonces al haber una colisión entre dos Entidades donde una tiene este Componente, la otra Entidad muere.
- ClosedDoor: Pasa a ser un Componente booleano, si hay una colisión entre el Jugador y esa Componente entonces en caso de que haya más de 1 llave recogida, esta Entidad muere y se resta una unidad al número de llaves recogidas.

Se tendría que arreglar un poco más la Componente de daño,  y a la hora de hacer puertas habría que darle una Componente collisionTypeComponente: Static y una ClosedDoor: true, pero aparte de eso tiene mejor pinta. Otro problema sería en los casos en que una Entidad tuviese una Componente damageOnTouchComponente y una CollisionTypeComponente, si se maneja mal podría pasar el Jugador al tocar dicha Entidad primero se reposicionase primero (por la Componente de colisión), no llevándose a cabo la función de daño nunca.

Otro aspecto del diseño que creo que se debería de cambiar, no hacer el diseño enfocado a los Ensamblajes como se ha hecho, es mejor enfocarlo hacia los Componentes como se hacía originalmente. El problema que surje con el enfoque escogido es que se tiende a crear Componentes orientados "solo" hacia esos Ensamblajes, lo cual no tiene sentido en ECS ya que los Ensamblajes pueden variar fácilmente entre una iteración y otra, mientras que las Componentes son algo más sólidos.

Último tema importante, las Entidades especiales. Son una buena idea, sobre todo porqué te ahorras datos como la Componente de input (haciendo uso de la Entidad especial Jugador) y  el uso de más de una Componente de size para los Tiles, pero tienen sus inconvenientes: se estaban añadiendo funcionalidades extras solo con el uso de este tipo de Entidades. Por ejemplo, Jugador no tiene ningún tipo de collisionBehaviourComponente, aun así tiene colisiones; lo mismo va para Tiles. De hecho en un momento del diseño hubieron problemas por esto. Una buena práctica sería limitarse a usar Entidades normales, nada de Entidades especiales, hasta el final del proceso de desarrollo donde se hacen todas las mejoras posibles en el sistema.

### 8.3. Motor físico

No tengo suficiente experiencia en el tema de como hacer un motor físico, pero si que me doy cuenta que es uno de los puntos donde se puede mejorar más el engine de este proyecto. En este caso solo se está trabajando con rectángulos, pero en el momento en que se quisiese hacer un motor físico en el que se tuviese en cuenta las figuras del mundo a nivel de pixel entonces se tendría que trabajar en un motor físico más refinado.

### 8.4. Sonido

Un juego necesita sonido. Así que obviamente este aspecto es una necesidad. DirectX por ejemplo posee la API Xaudio 2 para el sonido, que trata hasta con fuentes de sonido para hacer un mundo con ambientación estéreo, sería un buen punto para empezar.

### 8.5. DDD

Actualmente tanto el engine como el editor conocen todos los Ensamblajes que existen. Esto actualmente no es Data Driven Design, y es un problema.

Una de las grandes ventajas de ECS es la de usarlo junto a DDD ya que es mucho más sencillo de implementar. Cada Ensamblaje es un una forma de estructurar unos Componentes de una misma Entidad, por lo tanto una forma de tener un diseño orientado a DDD sería la posibilidad de poder crear un Ensamblaje uno mismo.

La creación de un Ensamblaje consistiría en guardar unas elecciones de tipo de Componente más sus valores de inicialización. La creación de un Ensamblaje solo se encontraría en el editor, pero se tendría que crear un mecanismo para pasar la información de estos Ensamblajes creados al engine.

Un posible mecanismo sería la creación de otro fichero de texto donde se tuvieran todas las especificaciones de los Ensamblajes, y el fichero original que se usaba para poder transmitir como era un mundo utilizaría información del tipo "hay una Entidad siguiendo las pautas del Ensamblaje X en la posición...." además de información del tipo "hay una Entidad con Componentes...", ya que ECS es capaz de crear una Entidad con un Ensamblaje o con un conjunto de Componentes (ya que son lo mismo).

En el engine se tendrían que leer todos los ficheros de texto y saber como montar los nuevos Ensamblajes, y luego generar el mundo utilizando esa información.

### 8.6. Multithreading

ECS también es notable en su uso junto a multithreading. El método común de multhreading en ECS es que cada Procesador ocupa un thread distinto, aunque creo que requiere un estudios más profundo ya que en el caso de que los Componentes tengan algún tipo de dependencia entre ellos entonces podrían necesitarse semáforos o algún otro tipo de mecanismo para que no hubiesen problemas de paralelismo, provocando una ralentización. Un ejemplo de problema sería que en un thread se estuviese ejecutando boxOverlappingProcessor y en otro physicProcessor, ambos utilizan positionComponent de las Entidades y physicProcessor los actualiza, es necesario algún mecanismo de sincronización.

Supongo que en el caso en que los Componentes no tengan nada que ver entre ellas, no hay ningún problema.

## 9. Conclusiones

Siempre me había preguntado que era hacer el engine de un juego ya que, como he expresado antes, solo había usado algunos engines o programas similares para hacer juegos. Por un lado, me doy cuenta que si uno quiere hacer un engine sencillo, no hay ningún problema. Por otro, si se desea hacer un engine lo más perfecto hay muchos elementos a tener en cuenta y no es nada sencillo.

Relaciona elementos de hardware como el sistema procesador (me refiero a la CPU) y la cache con elementos de software como es la forma de programar, para este proyecto ECS. También se tienen en cuenta el paradigma de programación DOD que es muy importante para sacar al máximo la potencia de un programa.

Además mientras buscaba sobre ECS y temas similares, a juzgar por los comentarios de desarrolladores en compañías de juegos AAA (son aquellos juegos que tienen un gran presupuesto), las arquitecturas de sus engines son incluso más complejas y se utilizan técnicas como "hot & cold" data. Abreviando mucho, se dividen los datos en datos que se usan continuamente (hot data) y en datos que se usan poco (cold data).

También he podido ver que sería interesante hacer un sistema híbrido, uno donde en las partes en que se requiera una gran potencia de procesamiento se hiciese con ECS y otra parte donde no se necesitase tanta potencia pero fuera demasiado complejo como para hacer con un ECS, se hiciese con OOD. En este proyecto no ha hecho falta porqué la IA es muy sencilla, pero hay juegos donde se utilizan algoritmos de búsqueda como el A* para saber que camino tomar para llegar un punto A a un punto B.

La idea con la que me he quedado del proyecto es que el tema de hacer un engine para videojuegos es muy interesante, puede llegar a tener mucha profundidad y es muy complejo. Y eso que ha sido un juego en 2D.

## 10. Referencias y bibliografía

[1] Adam Martin, t-machine.org: Entity Systems are the future of mmog development, all parts
[1.1] Part 1 http://t-machine.org/index.php/2013/05/30/designing-bomberman-with-an-entity-system-which-components/
[1.2] Part 2 http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/
[1.3] Part 3 http://t-machine.org/index.php/2007/12/22/entity-systems-are-the-future-of-mmog-development-part-3/
[1.4] Part 4 http://t-machine.org/index.php/2008/03/13/entity-systems-are-the-future-of-mmos-part-4/
[1.5] Part 5 http://t-machine.org/index.php/2009/10/26/entity-systems-are-the-future-of-mmos-part-5/
[2] Adam Martin, t-machine.org: Entity Systems: Contiguous Memory, http://t-machine.org/index.php/2014/03/08/data-structures-for-entity-systems-contiguous-memory/
[3] Tony Albrecht, The latency Elephant, http://seven-degrees-of-freedom.blogspot.com.es/2009/10/latency-elephant.html
[4] Mick West, Evolve your Hierarchy, http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy
[5] DICE's investigation department, Introduction to Data-Oriented-Design, http://dice.se/wp-content/uploads/Introduction_to_Data-Oriented_Design.pdf
[6] Tony Albrecht, Pitfalls of Object Oriented Programming, http://research.scee.net/files/presentations/gcapaustralia09/Pitfalls_of_Object_Oriented_Programming_GCAP_09.pdf
[7] Jason Gregory. *Game Engine Architecture*. CRC Press
[7.1] - Page 126

[8] David Patterson, A Case for Intelligent RAM: IRAM
[9] Adam Martin, Designing Bomberman with an Entity System: Which Components? http://t-machine.org/index.php/2013/05/30/designing-bomberman-with-an-entity-system-which-components/

## 11. Apéndice A: Como utilizar el editor y el engine

Para poder utilizar el editor y el engine hace falta:

- El código fuente de ambos proyectos.
- Windows 8.
- Visual Studio Express 2013. Seguramente se puedan usar otras versiones.

Importar ambos proyectos a Visual Studio.
Para poder crear un mapa con el editor solo se necesitan seguir unos pocos pasos:

1. Ejecutar editor.
2. Utilizar el botón de "Crear Mundo" si el tamaño del actual no es el deseado.
3. Utilizar los botones de "Crear Ensamblaje" y luego clickar en el mundo para añadir contenido. Uno que en principio es obligatorio es el Jugador.
4. Clickar el botón de "Guardar Mundo" y guardarlo con un nombre.

Para poder utilizar este mapa en el engine es necesario seguir estos pasos:

1. Mover o 'copiar y pegar' el archivo generado en el paso anterior en una subcarpeta de la carpeta 'Assets' del proyecto del engine, preferiblemente la subcarpeta 'world'.
2. Abrir el proyecto del engine en Visual Studio.
3. Ir hasta donde debería estar el archivo con la ventana de navegación del proyecto, 'Solution Explorer'.
4. Si no aparece, hacer click dos veces en el botón 'Collapse All'.
5. Hacer click derecho en el archivo y darle a 'Include in Project'.

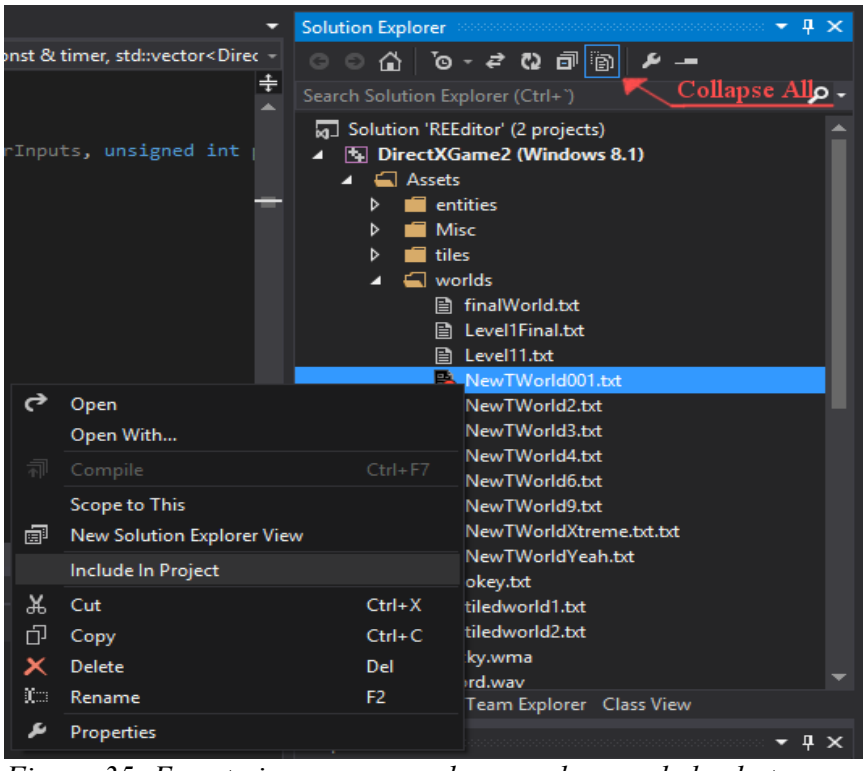


*Figura 35: En esta imagen se pueden ver algunos de los botones que se han mencionado, como 'Include in Project' o 'Collapse All'.*

6. Ir al archivo 'Scene2DModule.cpp', y cambiar el path del archivo en:
    1. Línea 64.
    2. Línea 371.

```
60      //Initialize world vars
61      initializeGameState();
62
63      //Load tile world
64      ReadTiledWorld(L"Assets\\worlds\\finalWorld.txt");
65
66      //gameState
67      state = sTart;
68      toInitWorld = false;
69
```

```
364 void Scene2DModule::Update(DX::StepTimer const& timer, std::vector<DirectXGame2::PlayerInputData>*
365     //First input
366     InputProcessor(timer, playerInputs, playersAttached);
367
368     //If the world has to be reinitialized
369     if (toInitWorld){
370         initializeGameState();
371         ReadTiledWorld(L"Assets\\worlds\\finalWorld.txt");
372         toInitWorld = false;
373     }
374
375     //Just while playing the world is updated
376     if (state == playing){
377         PhysicsProcessor(timer);
378         BoxOverlapEffectsProcessor();
```

*Figura 36: Los dos pedazos de texto a editar.*

7. Ejecutar el proyecto.

## 12. Apéndice B: Como jugar una partida

Una partida sigue siempre el mismo desarrollo:

- Start: Estado inicial, todo está quieto. Hay que hacer click con el ratón para continuar al estado Playing.
- Playing: En este estado ocurre la mayor parte del juego. En este estado:
  - Se puede controlar al Jugador, solo hay 3 acciones según la tecla que se aprete:
    - Tecla flecha izquierda: Mientras esté pulsada, mueve al Jugador hacia la izquierda.
    - Tecla flecha derecha: Mientras esté pulsada, mueve al Jugador hacia la derecha.
    - Tecla flecha arriba: Pulsarla hace que el Jugador salte (adquiere una gran velocidad vertical de repente). Mientras esté en el 'aire' no se podrá saltar de nuevo. Cuando aterrice, se podrá saltar de nuevo.
  - Hay una pequeña HUD donde se muestra:
    - El número de llaves recogidas, inicialmente 0.
    - El número de vidas, inicialmente 3.
  - Hay distintos mecanismos de juego:
    - Elementos que provocan la muerte:
      - Los pinchos, enemigos, bolas de fuego y tocar los bordes del mundo inferiores provocan la muerte. La muerte significa que el Jugador es movido a la posición inicial con la que apareció y perder una vida. Si se pierden todas las vidas el estado de juego pasa de Playing a GameOver.
      - Los pinchos son estáticos, pero los enemigos se mueven y cambian de dirección al chocar contra otros elementos y las bolas de fuego se mueven pero mueren al tocar algún tile.
    - Elementos que entorpecen al jugador:
      - Las puertas causan colisión con el Jugador. Eso puede entorpecer el progreso del jugador o hasta bloquearlo completamente.
      - Para poder pasar por una puerta solo es necesario encontrar una llave y luego ir hacia una puerta, que desaperecerá al entrar en contacto con el Jugador.
      - Los lanza bolas de fuego son inofensivos de por sí, pero crean bolas de fuego, que provocan la muerte.
    - Elementos que dan la victoria al jugador:
      - Solo hay uno, el letrero que pone "End", si el jugador lo toca gana y se pasa al estado Victoria.
- GameOver: Todo está quieto, al hacer click se pasa de nuevo al estado Start.
- Victoria: Es como GameOver, pero hay una imagen que pone "Victoria".

Solo queda decir que la estrategia para ganar es esquivar todos los elementos dañinos como enemigos o bolas de fuego, coger llaves y abrir puertas si es necesario, y llegar hasta los letreros "End" para ganar.