Cómo desarrollar un *GameEngine Entity-Component-System* en C++20

Laureano Cantó Berná

Revisado por Francisco José Gallego Durán

Mayo, 2023

Contenidos

1 Introducción

Hacer un videojuego para muchas personas es algo desconocido. Esta guía va dirigida a personas con una mínima experiencia en programación, dirigida a todos aquellos desarrolladores con ganas de aprender como funciona la arquitectura de software Entity Component System (ECS) y la base de los videojuegos. Está enfocada en ir resolviendo problemas, desde los más sencillos a los más complejos, obteniendo como resultados pequeños proyectos y videojuegos.

Considero este libro novedoso, y lo ofrezco a nuevos desarrolladores como herramienta de apoyo para sus futuros proyectos.

Será un documento progresivo desde 0, donde iremos avanzando paso a paso en el que se da por hecho que ya se conoce la base del lenguaje C++. A su vez, el objetivo no es hacer un videojuego enfocado en el aspecto gráficos, aunque usaremos gráficos para representar nuestros avances. Como ya se ha mencionado no serán el foco principal ya que éste reside en comprender la arquitectura ECS en un motor de juegos y en saber cómo manejarlo de forma correcta.

En esta primera introducción hablamos del mundo de los videojuegos en general y otros conceptos base para poner en contexto todo lo que realizaremos posteriormente, como estructuras de programación complejas y otros detalles. Si tu voluntad es empezar directamente con proyectos y los primeros pasos de esta guía puedes empezar leyendo el Capítulo 3.

1.1 El mundo de los videojuegos

Hoy en día, todos sabemos lo importante que han llegado a ser los videojuegos en nuestra sociedad. Han evolucionado desde pequeños juegos donde solo podíamos mover píxeles, a grandes obras de arte donde hasta el último detalle es importante.

Pero no solo han evolucionado respecto al arte o a lo bonito que podemos llegar a representar datos en nuestras pantallas, si no que han aparecido videojuegos con grandes historias por ejemplo, donde hace falta una narrativa, o donde el jugador puede incluso manejar el flujo de la historia con sus acciones. Otro ejemplo de avance lo podemos encontrar en la educación, ya que muchos de ellos pueden llegar a usarse como método de enseñanza para niñas y niños pequeños e incluso para llevar a cabo terapias psicológicas.
.

Si hablamos de avance tecnológico y artístico, el avance desde los primeros videojuegos a la actualidad, es demasiado grande. Por ejemplo uno de los juegos más importantes a nivel histórico fue el denominado "Pong", el cual se muestra en la figura 1. Un videojuego sencillo donde había poco contenido visual, pero el necesario para poder tener un objetivo a seguir y unas mecánicas sencillas. En un juego de este estilo, a lo mejor no es tan

necesaria una estructura o patrón de diseño, por que el máximo de objetos que puedes tener son pocos más de dos raquetas y una pelota.

Por otro lado, ya muy evolucionados, tenemos videojuegos más avanzados que requieren de estructuras y patrones de diseño, ya que disponen de muchos objetos y muchos factores distintos que modifican lo que vemos constantemente en la pantalla. Cuando el tamaño de objetos no se puede contar con las manos, puede volverse muy difícil el desarrollo del videojuego, además de sufrir tirones o errores por mala optimización. Es por eso que más adelante hablaremos de patrones de diseño y en concreto este proyecto trata sobre uno de ellos.

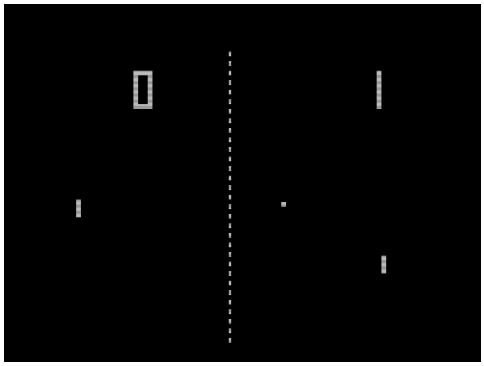


Figura 1. Imagen del videojuego "Pong"

En resumen, el mundo de los videojuegos ha evolucionado desde sus inicios en una industria global que ofrece diversión y aprendizaje. Con el avance de la tecnología, el futuro de los videojuegos promete ser aún más emocionante.

1.2 Desarrollo de videojuegos

Figura 2. Un evento de videojuegos

Los videojuegos para muchos son la forma de escapar de la realidad, o emplear su tiempo libre en divertirse con ellos. Pero para gran cantidad de personas, ha llegado a ser incluso su trabajo. Se utilizan en la enseñanza, y otros campos, algunos han dado pie a eventos y competiciones (figura 2). Detrás de todo esto hay grandes equipos de desarrollo, los cuales se dividen el trabajo para conseguir grandes resultados en el menor tiempo posible.

El desarrollo de videojuegos es un proceso complejo y multidisciplinario que involucra la creación de un juego desde su concepto hasta su lanzamiento y distribución. Esta industria es muy competitiva y exigente, pero también es muy gratificante y llena de oportunidades.

El equipo de desarrollo de un juego puede incluir diseñadores de juegos, artistas, programadores, compositores y otros profesionales. Cada uno de ellos tiene un papel importante en la creación de un juego y trabaja en estrecha colaboración para producir un producto final de alta calidad.

Para crear un juego, primero se debe tener una idea clara y definir su concepto. Luego, se crean bocetos (figura 3) y prototipos para probar la jugabilidad y las mecánicas del juego. Una vez que se tiene una idea sólida, el equipo comienza a desarrollar el arte, la música y el código que conforman el juego.

La programación es una parte importante y esencial del desarrollo de videojuegos. Los programadores utilizan lenguajes de programación como C# o C++ para crear la lógica y el comportamiento del juego, siendo el primero de estos el más utilizado. Además, también se utilizan motores de juegos con entornos gráficos como Unreal Engine y Unity para simplificar y acelerar el proceso de desarrollo.

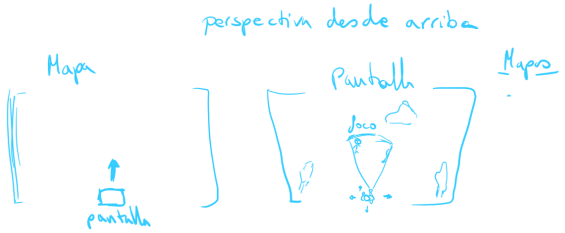


Figura 3. Bocetos de mecánicas.

Una vez que el juego está en un estado avanzado, se lleva a cabo una fase de pruebas y ajustes para resolver cualquier problema o errores. Después de la fase de pruebas, el juego está listo para su lanzamiento y distribución.

En esta guía nos centramos y puliremos la parte del desarrollo del motor que gestiona nuestro juego y usaremos el lenguaje de programación C++ para lograrlo.

1.2.1 C++20 como lenguaje de programación

C++ es un lenguaje de programación de nivel medio multiparadigma que se utiliza en una amplia variedad de aplicaciones, incluyendo el desarrollo de videojuegos, aplicaciones de escritorio, sistemas operativos, aplicaciones móviles y más. Fue desarrollado por Bjarne Stroustrup a finales de los años 70 y es un lenguaje construido para ser compatible con C.

Es conocido por su velocidad y eficiencia, lo que lo hace ideal para aplicaciones que requieren un rendimiento rápido y confiable. Es muy versátil y tiene compiladores para la mayoría de sistemas operativos. También cuenta con una gran comunidad de desarrolladores y una amplia variedad de herramientas y bibliotecas disponibles para ayudar en el desarrollo de aplicaciones.

A pesar de su complejidad, C++ es un lenguaje de programación muy popular y ampliamente utilizado. Muchos programadores encuentran que es un lenguaje de programación poderoso y muy manejable, lo que lo convierte en una excelente opción para llevar a cabo proyectos y videojuegos.

El desarrollo de videojuegos en C++ ha sido una de las formas más populares de crear juegos desde hace décadas. El estándar 20 de C++ ha introducido una gran cantidad

de mejoras y características nuevas, haciéndolo aún más fácil y eficiente para los desarrolladores de videojuegos.

1.2.2 Motores

Un motor de videojuegos no es más que unas rutinas de programación que permiten crear y diseñar el funcionamiento de un videojuego. Suelen proporcionar un conjunto de herramientas que pueden ser reutilizables. Esto te permite crear diversos videojuegos de forma rápida. Hay motores, los cuales incluyen una interfaz incorporada, como Unity (figura 4) o Unreal Engine.

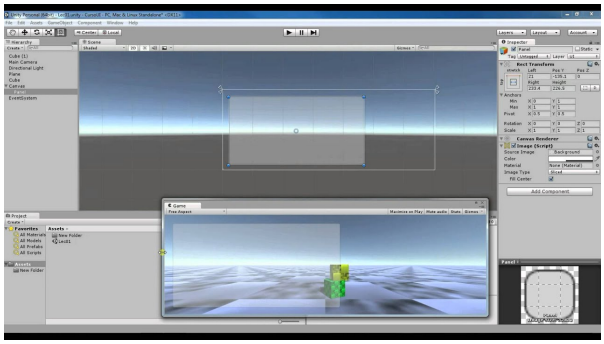


Figura 4. Interfaz de Unity

Los motores representan funcionalidades como mencionamos anteriormente. Hay motores que representan el apartado gráfico de un juego, los motores gráficos, los cuales disponen de funcionalidades de dibujo y animación. También existen motores de físicas, donde el objetivo es recopilar y ofrecer funciones de movimiento, caídas, gravedad, etc.

Pero, hay motores a su vez que se encargan del manejo de la memoria, los datos y de su gestión y procesado. Este tipo de motores de juego, podemos desarrollarlos basándonos en una arquitectura de software, lo que nos va a permitir manejar datos complejos y la memoria de forma sencilla y comprensible, y es el objetivo de esta guía, conseguir uno de estos motores usando la arquitectura ECS que veremos posteriormente.

Antes de la existencia de los motores en videojuegos, se desarrollaban en conjunto todas las ramas de un videojuego, es decir, no había diferencias entre motor gráfico, físico, etc. Todo debía ser planeado desde cero, intentando no complicar demasiado el código. Este avance tecnológico, que supuso la separación en bloques de un videojuego, marcó una gran diferencia por ejemplo en juegos como Doom o Wolfenstein 3D, que alcanzaron mucha popularidad.

Nosotros nos centraremos en el desarrollo del ECS que hemos mencionado anteriormente, y para ello, hablaremos a continuación de los distintos modelos para manejar la información y estructurar los datos.

1.2.3 Modelo orientado a objetos

En este modelo disponemos de estructuras llamadas objetos, que contienen los datos y las relaciones entre ellos. Nos permite reutilizar código, que esté organizado y sea fácil de mantener.

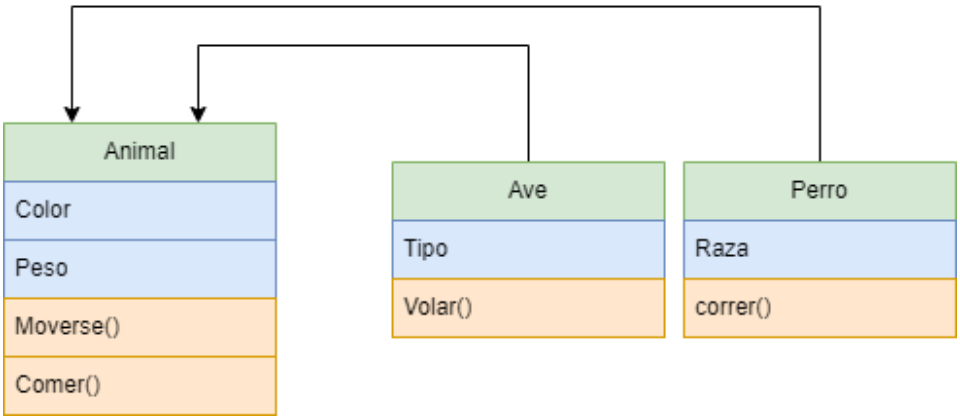


Figura 5. Ejemplo del modelo orientado a objetos

El primero de sus conceptos es la distinción entre clase y objeto. La clase es una plantilla. Define de manera genérica cómo van a ser los objetos de un determinado tipo.

Por ejemplo si hablamos de animales, podemos tener una clase que sea Animal, en la que habrán atributos tales como vida, número de patas, etc, y luego un objeto que hereda de esta clase que puede ser por ejemplo un Perro o un Pájaro. Los dos objetos tendrán vida o número de patas por ejemplo, pero el pájaro tendrá funciones de volar o planear. De esta forma podemos definir cualquier tipo de animal, partiendo de una base conjunta, que todos poseen. En la figura 5, se aprecia un ejemplo.

1.2.4 Modelo orientado a datos o por componentes

El modelo orientado a datos, tiene como objetivo optimizar y obtener un procesado de datos más eficiente.El enfoque principal se basa en la reutilización de entidades llamadas componentes.

Uno de los problemas del modelo orientado a objetos es que este modelo tiene demasiados detalles y elementos específicos en cada proyecto diferente. La ventaja principal es que los componentes son más abstractos que las clases de objetos y pueden considerarse más independientes.

Estos componentes proporcionan un servicio sin importar dónde se ejecute este, ya que podemos definirlo como una entidad ejecutable independiente que a su vez puede estar compuesta por otros objetos ejecutables.

Por lo tanto, en lugar de tener objetos, tendremos entidades que contendrán componentes, y cada uno de estos componentes representará los datos que necesitamos para llevar a cabo una funcionalidad.

De esta forma podemos facilitar los accesos y comprobaciones solamente teniendo en cuenta los datos o podemos acceder solo a los componentes que deseemos con un coste

menor. En un sistema orientado a datos, el código y los datos están separados.

De la misma forma que hemos propuesto en el apartado anterior, esta vez tendremos Entidades que serán los distintos animales, las cuales contendrán componentes que serán las funcionalidades de vida, o cantidad de patas.

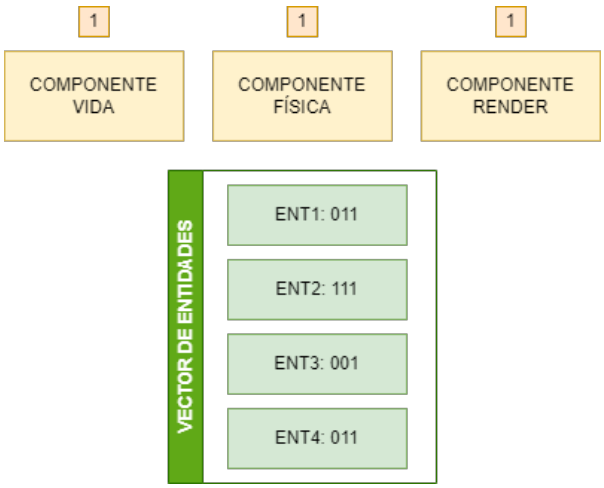


Figura 6. Ejemplo del modelo orientado a datos

De esta forma también podemos representarlos con bits. Podemos tener la funcionalidad de vida, donde necesitaremos datos que representen la vida, como un número. Si queremos una entidad que tenga la funcionalidad de visualizarse en la pantalla, tendremos que añadirle el componente con los datos que formen esa funcionalidad, por ejemplo una textura para el "sprite".

Esto también lo podemos representar con máscaras de bits, como se muestra en el diagrama de la figura 6. Una entidad que tenga los bits 0011 por ejemplo, que dispondrá de dos componentes y otra entidad con 0010 la cual será estática ya que no tiene componente de físicas pero si tendrá vida. De esta forma podemos tener numerosas entidades y solo modificando los componentes necesarios en cada momento tendremos diferentes funcionalidades para cada una.

Es este modelo de programación orientada a datos el que llevaremos a cabo para realizar nuestro motor Entity Component System, del cual hablaremos más en profundidad en el siguiente capítulo, (2).

1.3 Cómo seguir este libro

Este libro está pensado para enseñar a los programadores junior y estudiantes de C++ interesados en el mundo de los videojuegos a desarrollar un motor con el que poder crearlos. Para ello seguiremos unos pasos a la hora de iterar sobre nuestro motor ECS.

Esta guía está basada en prototipos de juegos, que he elegido para ir evolucionando dicho motor proyecto tras proyecto. Por lo tanto los consejos siguientes te vendrán muy bien para entender el libro y los conocimientos que contiene.

En primer lugar os encontraréis con un apartado de teoría donde introducimos el patrón Entity Component System, que si ya conocéis recomiendo ir directos al siguiente apartado, donde comenzaremos a trabajar, el capítulo 3.

El libro consta de cinco proyectos, empezaremos con uno muy sencillo y acabaremos con un proyecto final con mucho contenido nuevo y un motor preparado para hacer juegos a un nivel más alto.

Mi consejo es recorrer en orden cada uno de los proyectos, de esta forma entenderás todas las partes del motor y podrás crear el tuyo propio entendiendo por qué usar unas u otras cosas y sabiendo como optimizar al máximo su rendimiento.

Durante los diferentes proyectos hay apartados de teoría para introducir elementos los cuales requieren de una base previa, además de apartados para introducir tecnologías que usaremos.

En la siguiente lista, puedes observar los proyectos que la guía incluye, para hacerte una idea principal de estos proyectos.

- Nivel bajo: Los dos primeros proyectos están hechos usando la terminal del sistema para mostrar los resultados y su objetivo es aprender la estructura del patrón. Esto incluye una explicación de la librería de Gallego-Durán (2021), que utilizaremos para que sea más fácil y más bonito.
- Introducción a Raylib, Santamaria (2013): El siguiente proyecto, el tercero, será un proyecto intermedio entre los dos anteriores y los siguientes. Esto se debe a que introducimos gráficos y reharemos el segundo proyecto para aprender a usar las nuevas tecnologías.
- Nivel alto: Los dos últimos proyectos avanzan de forma más contundente en la creación de entidades, componentes y sistemas. Esto se debe a que el objetivo, a parte de mejorar nuestro motor, es crear juegos más completos, explorando nuevos sistemas y formas de crearlos, y mucho más.

Finalmente, espero que disfrutes de este libro y puedas desarrollar tus juegos entendiendo qué estas haciendo en cada momento. Recomiendo seguir el flujo del libro, ya que en cada uno de estos proyectos mejoramos nuestro motor y en todos aparecen conceptos importantes y novedades.

2 Entity Component System

En el mundo de los videojuegos, la optimización del código, los datos y la memoria son muy importantes, de esto depende la potencia de tu creación. Uno de los problemas a la hora de hacer un juego es la flexibilidad de tu código, ya que si no se sigue ninguna estructura u orden será un desastre el resultado.

Necesitaremos unas rutinas de programación que permitan el diseño, creación y funcionamiento de un videojuego. Esta rutina la conseguiremos desarrollando un motor Entity Component System (ECS), el cual nos permite tener entidades relacionadas con sus componentes y asegura un correcto flujo entre los sistemas.

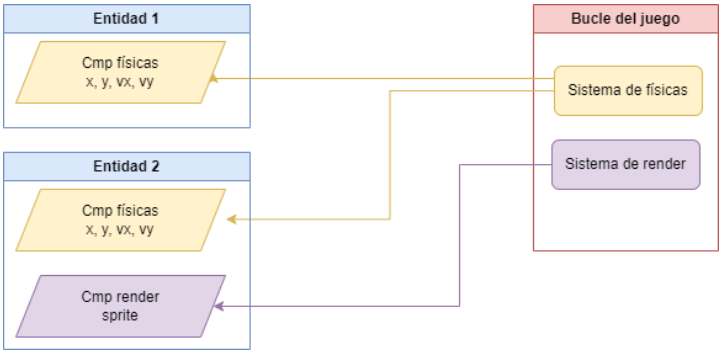


Figura 7. Diagrama simple del Entity Component System

2.1 ¿Qué es un ECS?

El Entity Component System(ECS), es una arquitectura de software muy usada en el mundo de los videojuegos que tiene un objetivo claro, mejorar la organización y eficiencia del código. Esto se puede llegar a conseguir descomponiendo los elementos del juego en entidades, componentes y sistemas, lo que genera una separación de responsabilidades, haciendo que al aplicar esta arquitectura de software el código pueda ser muy escalable y reutilizable.

Sigue el principio de composición sobre herencia, esto significa que cada entidad se define por sus componentes asociados. Los sistemas actúan globalmente sobre todas las entidades que tengan los componentes requeridos. En el ejemplo de la figura 7 se puede apreciar de forma gráfica la estructura que sigue esta arquitectura.

Las características del Entity Component System nombradas en varias ocasiones anteriormente, podemos definirlas de la siguiente forma:

2.1.1 Entidades

Las Entidades representan cada uno de los objetos visibles y no visibles del videojuego, cada personaje, cada objeto puede ser una entidad. La entidad es simplemente un

identificador único que se asocia con los componentes que describen sus características y comportamientos. Muchas veces, cuando hablamos de entidades en código, las vemos representadas por componentes en su interior o simplemente con una máscara de bits, que representa el conjunto de componentes que la forman.

Las entidades en el ECS son simples y no tienen lógica propia, ya que esta lógica se encuentra en los componentes y los sistemas. En su lugar, las entidades se utilizan para agrupar componentes relacionados y permitir una gestión más eficiente de la lógica y los datos en el juego o aplicación.

En el siguiente código hay un ejemplo de Entidad, de forma rudimentaria, ya que es poco óptimo almacenar dentro de esta los componentes directamente.

Entidad

```
struct Entity{
     int id{};
     HealthComponent health{};
     RenderComponent render{};
};
```

Pero teniendo esto en cuenta podemos escalar en complejidad. Por ejemplo ahora podemos usar "std::optional<Componente>" para poder saber de forma rápida si la entidad tiene estos componentes, simplemente sabiendo si se han inicializado o no.

Entidad con std::optional

```
struct Entity{
     int id{};
     std::optional<HealthComponent> health{};
     std::optional<RenderComponent> render{};
};
```

Ya avanzando en complejidad y optimizando mucho lo anterior, podemos incluso representar los componentes con bits y simplemente tener en la entidad una mascara junto al identificador, cosa que haremos en futuros ejemplos.

Entidad con máscara de componentes

```
struct Entity{
     int id{};
     std::size_t maskCmp{};
};
```

2.1.2 Componentes

Los componentes son datos que representan las funcionalidades del juego. Por ejemplo un componente de vida, otorga esta capacidad a una entidad. Si nos fijamos en el código siguiente, un componente puede ser este.

Ejemplo: Componente de vida

```
struct HealthComponent{
     int health;
     int healthIncrement;
}
```

En el se guardarán dos valores, el número de vida y el incremento que aumentará o decrementará la vida según la situación dada.

Esto es solo un ejemplo de para qué sirven los componentes e implementaremos diferentes tipos a lo largo de la guía.

2.1.3 Sistemas

Los sistemas son códigos que procesan los componentes y actualizan el estado del juego o aplicación. Trabajan con los componentes de las entidades y realizan tareas específicas, como el dibujado de gráficos, la gestión de la física, el control de los personajes, la detección de colisiones, etc.

Además, son los responsables de la actualización del juego. Pueden acceder a los componentes que necesitan para realizar sus tareas sin tener que conocer la estructura interna de las entidades.

Estos componentes que los sistemas necesitan suelen verse especificados de alguna forma, y el motor como ya hemos mencionado se encarga de revisar las entidades que disponen de estos componentes, las envía al sistema y este las modifica.

2.1.4 Manejador de entidades

Las entidades de nuestro videojuego serán modificadas durante la ejecución con acciones que el jugador realice, se crearán nuevas y se eliminarán a su debido momento.

Hemos nombrado con anterioridad al “Entity Manager”, como la parte base y de unión de toda la estructura. Es la clase que contiene el espacio para almacenar todos nuestros datos. Es el lugar donde encontraremos por ejemplo un "array" de entidades, o estructuras para almacenar componentes. En el caso de que estos almacenamientos sucedan en otro lugar, siempre habrá un acceso a esa memoria en esta interfaz.

Además de contener la gran cantidad de información de nuestro motor, contiene funcionalidades base del videojuego. Posteriormente los sistemas se apoyan en estas funciones propias del Entity Manager para obtener los componentes a modificar. Estas funciones por ejemplo serán funciones para seleccionar a que componentes aplicar ciertas acciones o si aplicarlas a todos.

Entity Manager Base

```
struct EntityManager{
   EntityManager(){}
   void forall ( auto && function );

   private:
   std::vector<Entity>  entities;
}
```

El manejador de entidades es esencial para la implementación eficiente y escalable del ECS, ya que permite una organización clara y estructurada de las entidades y componentes en el juego o aplicación. De esta forma, se pueden crear, gestionar y procesar grandes cantidades de entidades de manera eficiente, lo que permite construir juegos y aplicaciones complejos y escalables.

3 Estructura principal de un ECS

Los primeros pasos se verán reflejados en dos proyectos principales, donde aplicaremos la arquitectura Entity Component System al motor que vamos a crear. No se tomará mucho en cuenta el tema optimización en ambos proyectos, ya que el primer objetivo es tener clara la estructura y familiarizarse con esta. Aun así, comentaremos pequeños detalles a tener en cuenta para optimizar lo máximo posible.

Además en la siguiente tabla (1), se podrán observar los proyectos que realizaremos en este apartado.

	Título	Descripción corta
Proyecto 1	Starfield	Un ECS sencillo, observamos su estructura.
Proyecto 2	Firefighter Game	ECS con diferenciación de entidades.

Tabla 1. Proyectos sobre la estructura del ECS

La importancia de estructurar nuestros códigos, reside principalmente en tener un orden de trabajo y disponer de una buena organización. Si las dos características anteriores se cumplen, obtendremos una optimización del código, y del trabajo a realizar posteriormente.

La explicación anterior acerca del Entity Component System, nos servirá para tener claro el objetivo de cada elemento que usemos en los proyectos futuros.

A continuación, se proponen dos proyectos similares, escalables en dificultad, donde comenzaremos por un simple efecto visual, que tendrá detrás un pequeño ECS, y posteriormente un minijuego donde podremos hacer movimientos como jugador.

En ambos usaremos Entidades para representar a cada uno de los objetos que tendremos que tener bajo control, y componentes que construirán nuestras entidades. Como primera medida, todas nuestras entidades tendrán todos los componentes, lo que no lo hace del todo óptimo, pero nos permitirá obtener la primera idea de lo que representa.

3.1 Introducción a "Basic Linux Terminal Library"

Antes de comenzar con el primer proyecto, en este apartado introduciremos la librería "Basic Linux Terminal" de Gallego-Durán (2021).

Esta librería nos servirá para imprimir en la terminal o consola de nuestra máquina (en sistemas Linux), donde podremos modificar el color, el estilo de letra, etc.

En cuanto al control por teclado de nuestros juegos, usaremos también esta librería, ya que dispone de funciones muy sencillas y útiles para esto.

Voy a poner dos ejemplos de los usos más comunes que tiene, los dos referenciados anteriormente:

- Pintamos en el terminal un elemento. Para esto, se pueden concatenar los valores con el comando "\033[" con elementos de esta librería como el estilo de la letra "At_Bold" o el color de los elementos que dibujamos "FG_Red". También podemos establecer el color del fondo sobre el que escribimos con "BG_Default" y otros parámetros similares, y mucho más.

  Uso de Basic Linux Terminal para estilizar nuestros elementos

  ```
  std::cout << "\033["<< TERM::AT_Bold <<";"<< TERM::FG_White
  <<"m------FIREFIGHTER GAME------\033["<< TERM::BG_Default
  <<"m\n"; std::cout<<"@\n";
  ```

  Pintamos el símbolo "@" en negrita, de color blanco sobre el fondo por defecto de nuestra terminal. Podemos cambiar el color de la letra y todos los parámetros antes de dibujar algo nuevo y así obtener elementos de diferentes colores.
- Controlamos las teclas. Para saber que tecla ha pulsado el usuario, "Basic Linux Terminal" tiene una función que nos facilitará el trabajo. Esta función es la siguiente:

  Uso de Basic Linux Terminal para controlar las teclas

  ```
      TERM::Terminal_t drawer{};
      int tecla = drawer.wait4NextKeyPress();
  ```

  De esta forma, podemos guardar en una variable el valor de la tecla pulsada, y comprobar si se han pulsado unas teclas u otras.

Podemos hacer muchas más cosas con la librería, pero nos basta con estas dos funcionalidades para el desarrollo de los siguientes proyectos.

Por último, aunque no pertenece a esta librería, quiero también hacer incapié en el siguiente comando:

Estabilización con el borrado de la terminal

```
    std::cout<< "\033[H\033[J";
```

El comando anterior, se encarga de limpiar la terminal y estabilizarla, para que cada elemento que pintemos de la sensación de que está en la misma posición que el anterior en cada iteración del juego. Si no usamos esto, imprimiríamos hacia abajo, como si de una lista se tratase, y no queremos eso.

Esto es todo lo que respecta a la librería usada, y a continuación, comencemos con los proyectos.

3.2 Primer Proyecto: Starfield

Para poner en práctica toda la teoría, haremos un primer proyecto que llamaremos "Starfield". Este proyecto consta de numerosas estrellas recorriendo la pantalla de un extremo a otro. Desaparecen por un lado y aparecen por otro. El resultado, tiene que tener estrellas moviéndose a diferentes velocidades, dando una impresión de profundidad y movimiento adecuados.

En primer lugar, para conseguir este, como para conseguir cualquier otro proyecto, tendremos que tener muy en cuenta antes de empezar todo lo que necesitamos crear y cómo lo vamos a representar.

3.2.1 Presentación del proyecto

Tendremos que representar estrellas. Cada una de estas estrellas serán entidades, las cuales tendrán un componente de renderizado, ya que queremos que se vean por pantalla, y un componente de físicas, para que se muevan según su velocidad de movimiento. Este último componente estará compuesto por posición x e y para saber donde colocarla y una velocidad para saber cómo de rápido se mueven nuestras estrellas. No nos hará falta tener un entorno donde colocar nuestra entidad ya que en este caso tendrá la terminal como salida y podemos dibujar en posición x e y el sprite correspondiente en cada caso. En la figura 8 se aprecia el esquema a conseguir.

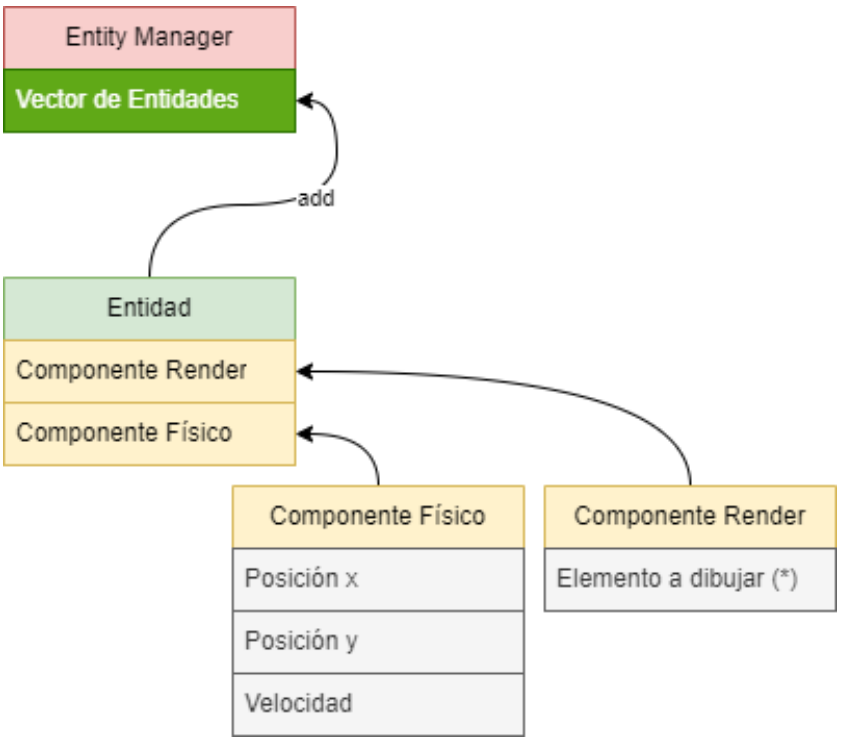


Figura 8. Diagrama estructural del proyecto Starfield.

Una vez tratada toda la introducción al problema, comenzaremos con el desarrollo, el cual comenzará definiendo los componentes de nuestro proyecto, siguiendo el esquema anterior.

3.2.2 Creación de componentes y composición de Entidades

Harán falta dos componentes: el componente de físicas y el componente de renderizado.

El componente de físicas necesita dos posiciones, x e y, para representar la posición donde más tarde dibujaremos esa estrella. Además contiene la velocidad en el eje x ya que no necesitamos otra velocidad.

Por otro lado el componente de render simplemente tendrá un carácter que guardaremos, y lo llamaremos sprite, representará con asteriscos a las estrellas de nuestro proyecto.

Componentes

```
//COMPONENTES----------------------
struct PhysicsCMP{
   int x, y;
   int vx;
};
struct RenderCMP{ char sprite{'*'}; };
```

Antes de continuar, y para comprender posteriores métodos, es conveniente saber cómo van a funcionar las físicas de nuestras entidades. Es sencillo, principalmente las declararemos como aleatorias, es decir, la 'x' será un numero aleatorio entre 0 y 4, así tendremos unas que salen antes y otras que salen después. La posición 'y' variará entre 0 y 9 ya que hemos decidido que haya 10 filas de tamaño por donde circular, y por último la velocidad, también aleatoria entre 1 y 4 para tener diferentes velocidades y crear el efecto que queremos. El componente de físicas quedará de las siguiente forma, tras aplicar estos cambios a su constructor:

Componente Físico

```
struct PhysicsCMP{
   int x, y;
   int vx;
   PhysicsCMP(){
        x = rand()%4;          //entre 0 y 3
        y = rand()%10;         //entre 0 y 9
        vx = 1 + (rand()%5); //entre 1 y 4
```

```
    }
}
```

Ya tenemos los componentes definidos, en el diagrama 9 tenemos iluminada la parte que tenemos programada hasta ahora. A continuación necesitaremos seguir definiendo las entidades a partir de los componentes que hemos definido.

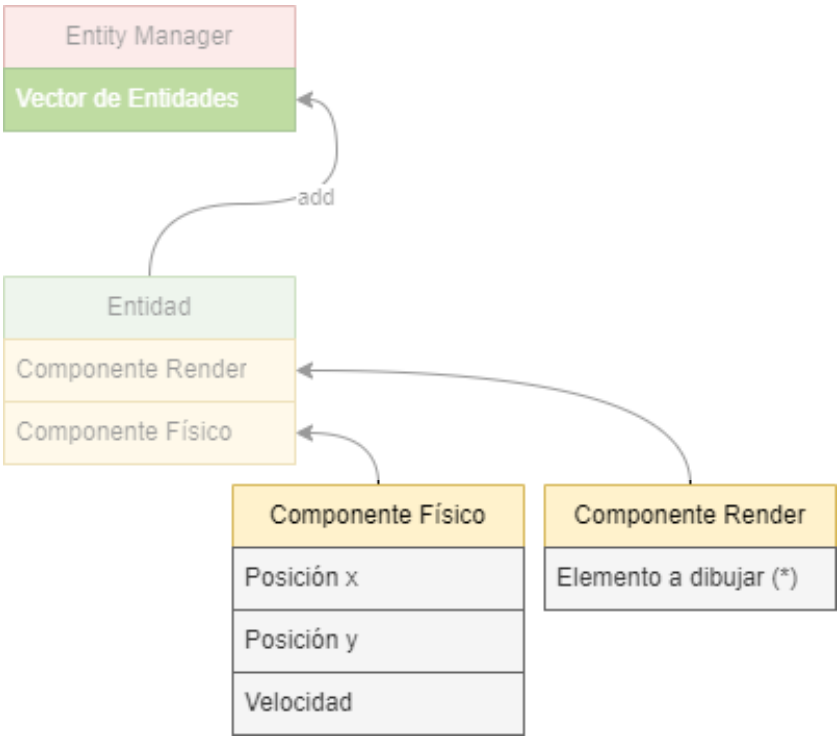


Figura 9. Componentes creados

Las entidades estarán compuestas en este caso tanto por componente físico como por componente de render. Normalmente pueden llevar id, para diferenciar a cada una de ellas, pero es algo que incluiremos más adelante. Por ahora simplemente la compondremos de nuestros componentes.

Entidades

```
//COMPONENTES--------------------
//ENTIDAD-------------------------
struct Entity{
    PhysicsCMP phy{};
    RenderCMP rend{};
}
```

3.2.3 Manejador de entidades

Podemos continuar construyendo la clase manejadora de entidades de nuestro motor, la cual contendrá el vector de entidades de nuestro proyecto, y un constructor principalmente:

Entity Manager

```
struct EntityManager{
   EntityManager(){}

   private:
   std::vector<Entity>  entities;
}
```

El Entity Manager tiene como único propósito la gestión de nuestras entidades. Esto quiere decir que todas las entidades que creemos en nuestro juego o programa, irán almacenadas en el vector "entities" que hemos creado. El constructor de esta clase, simplemente será una reserva de memoria, ya que usamos vectores y podemos pedir que de primeras nos guarde una cantidad de memoria. Por defecto la podemos establecer en 10 entidades.

Entity Manager::EntityManager()

```
EntityManagerP1(int size_for_entities = 10){
    entities.reserve(size_for_entities);
}
```

Seguimos creando las entidades, las cuales todavía no se pueden crear, ya que el vector de entidades es privado y necesitamos crearlas en este vector. Para conseguir esto, vamos

a programar la función de crear entidades en el Entity Manager. También vamos a programar un método auxiliar para obtener las entidades en cualquier momento:

Creación de entidades y métodos get

```
auto& createEntity(){ return entities.emplace_back(); }

std::vector<Entity>& getEntityVector(){ return entities; }
```

Con estos métodos hay que llevar cuidado y entenderlos bien. El 'auto' del método de creación de entidades junto a la referencia, nos devolverá justo la entidad que acabamos de crear. Tal vez no sea muy útil en este proyecto pero más adelante puede ser de gran utilidad, ya que una vez creada, podremos configurarla. A su vez, usaremos "emplace_back()", método de 'vector', para insertar directamente la entidad al final del vector. También con el método "get", al devolver por referencia, nos referimos a ese mismo objeto, y esto nos será útil para editar el vector de entidades y para consumir mucho menos memoria, ya que no creamos copias enteras del vector.

Si repasamos el diagrama del inicio 8, podemos observar que lo hemos completado. Simplemente era un diagrama estructural, para comprender la estructura de los componentes y ver como se componen las entidades. Sin embargo, tenemos que pensar y programar los sistemas.

3.2.4 Teoría: Lvalue y Rvalue

En C++, un "lvalue" se refiere a un objeto que tiene una identidad y que persiste en el tiempo, lo que significa que tiene una dirección en memoria y puede ser asignado a otra variable. Por ejemplo, una variable, un elemento de un array o un miembro de una estructura.

Por otro lado, un "rvalue" se refiere a un objeto que no tiene una identidad y que no persiste en el tiempo, lo que significa que no tiene una dirección en memoria y no puede ser asignado a otra variable. Un ejemplo común de rvalue es un número literal o una cadena de caracteres.

La diferencia entre estos, es que los lvalues son objetos que tienen una dirección en memoria y persisten en el tiempo, mientras que los rvalues son objetos que no tienen una dirección en memoria y no persisten en el tiempo. Esta distinción es importante porque los lvalues pueden ser modificados, mientras que los rvalues no. Además, la utilización de referencias rvalue en C++11 y versiones posteriores ha permitido el uso de valores temporales en operaciones de manera más segura y eficiente.

En C++, '&', se utiliza para definir una referencia a un objeto existente, permitiendo acceder y modificar el objeto original. Por ejemplo, en una función, se puede pasar un

objeto por referencia utilizando la sintaxis "void function(Type& obj)", lo que permite a la función modificar el objeto original pasado como argumento.

Por otro lado, la referencia '&&' se utiliza para definir una "referencia universal"o "forwarding reference". Esta referencia es capaz de vincular tanto a valores lvalue como rvalue, y se utiliza a menudo en plantillas y en funciones que deben aceptar cualquier tipo de argumento.

Esto lo tenemos que tener claro para el siguiente apartado, ya que usaremos la doble referencia para crear una función que reciba otra función como parámetro, a la cual le pasaremos las entidades para modificarlas.

3.2.5 Teoría: Expresiones lambda

Una expresión lambda es una forma de crear funciones anónimas, es decir, funciones que no tienen un nombre identificativo y que pueden ser usadas de manera inmediata en el ámbito en el que se crean. Las expresiones lambda son muy útiles para simplificar y hacer más legible el código, ya que permiten definir funciones pequeñas y específicas en el contexto que se requiera.

La sintaxis básica de una expresión lambda en C++ es la siguiente:

Estructura de una expresión lambda

```
[captura] (parámetros) -> tipo_retorno { cuerpo_función }
```

- "captura" indica qué variables externas (de la función que contiene la expresión lambda) se pueden utilizar dentro de la expresión lambda.
- "parámetros" son los argumentos que se le pasan a la función.
- "tipo_retorno" es el tipo de dato que devuelve la función.
- "cuerpo_función" es el código de la función.

Por ejemplo, si quisiéramos definir una función lambda que calcule el cuadrado de un número entero, podríamos hacerlo de la siguiente forma:

Expresión lambda que calcula el cuadrado de un número

```
auto cuadrado = [](int x) -> int { return x * x; };
```

La expresión lambda "cuadrado" toma un parámetro entero "x", y devuelve el valor de "x" multiplicado por sí mismo (el cuadrado de "x"). Devuelve un "int".

Las expresiones lambda también pueden tener captura, es decir, pueden utilizar variables externas. Por ejemplo:

Expresión lambda con captura

```
int a = 5;
auto suma_a = [a](int x) -> int { return x + a; };
```

La expresión lambda "suma_a" toma un parámetro entero "x", y devuelve el valor de "x" sumado con la variable "a". En este caso, la variable "a" se ha capturado mediante "[a]".

Si en lugar de pasarle a la captura "[a]", le pasamos "[&]", dentro de la expresión lambda podremos usar todas las variables y métodos de el ámbito exterior al lambda.

Las expresiones lambda son una herramienta muy útil para definir funciones pequeñas y específicas en el contexto en el que se necesitan. Permiten crear código más legible y simplificado, y pueden ser utilizadas en muchos contextos en los que se necesitan funciones anónimas.

3.2.6 Creamos los sistemas

A continuación hacen falta sistemas, la tercera parte del Entity Component System, y en este proyecto inicial, vamos a programar dos. El primero será un sistema de físicas, cuyo objetivo será que para cada entidad que le llegue, actualice su posición, siempre comprobando si se sale de los límites para reposicionarla al principio de la pantalla, haciendo el efecto de que las estrellas aparecen por el principio y desaparecen por el final. Para lograr esto, necesitaremos un método extra y uno de los más importantes en el Entity Manager, el método "forall". Este método recorrerá todas las entidades del vector ejecutando una función para cada una de ellas.

Método forall

```
    void forall(auto&& function){
        for(auto&e:entities){
            function(e);
        }
    }
```

De esta forma, podemos pasar por parámetro funciones que serán ejecutadas para cada entidad del vector "entities". El "auto&&" permite hacer referencia a la función que posteriormente pasaremos por parámetro, y una vez se compile el código se evaluará para cada caso. Esto lo hemos visto en el apartado anterior, 3.2.4.

Otra forma de verlo, será usando un tipo de función para que en lugar de dejarlo para que se evalúe posteriormente, nosotros le declaramos el tipo de las funciones que se le van a pasar por parámetro al método. Es el código del siguiente ejemplo:

Otro ejemplo de método forall

```
    // El tipo que definimos a continuación recibe
    // una referencia a Entidad y devuelve void
    using PointerToFunction = void(*)(Entity&);
    void forall(PointerToFunction function){
        for(auto&e:entities){
            function(e);
        }
    }
```

Una de las diferencias entre ambos "forall", se verá reflejada en los sistemas que programaremos a continuación, ya que la que usa la doble referencia '&&', puede usar un lambda en su sistema y nos será más eficiente posteriormente, ya que no requerirá de una función estática para usarla y podremos referirnos a todas las variables del ámbito dentro del mismo sistema.

Teniendo el "forall" programado, es hora de pasar a los sistemas, primero el de físicas. En el sistema de físicas, realizaremos un método que denominaremos "update()". Todos los métodos que estén dentro de un sistema, cuyo objetivo sea actualizar los datos de los componentes, lo llamaremos "update()". En este método, haremos una llamada al Entity Manager el cual se nos pasará por parámetro y tendremos dos modalidades para crear sistemas:

Sistema de físicas con miembro Update One Entity

```
struct PhysicsSystem{
    void update(EntityManagerP1& EM){
        EM.forall(updateOneEntity);
    }
    static void updateOneEntity(Entity& e){
        e.phy.x += e.phy.vx;
        if(e.phy.x > 99){
            e.phy.x = 0;
        }
    }
};
```

Sistema de físicas con lambda

```
struct PhysicsSystem{
    void update(EntityManagerP1& EM){
        // El [&] del parámetro del forall, recoge
```

```
        // todas las variables por referencia a las que se
        // puede acceder desde fuera del lambda
        EM.forall([&](Entity&e){
            e.phy.x += e.phy.vx;
            if(e.phy.x > 99){
                e.phy.x = 0;
            }
        });
    }
};
```

En ambos ejemplos el funcionamiento es el mismo, sumamos la velocidad a la posición x y comprobamos que esté entre 0 y 99 que será la longitud horizontal de nuestra pantalla. De esta forma, en cada iteración veremos como avanzan nuestras estrellas.

Por último, antes de concluir, nos hace falta otro sistema, el sistema de renderizado. Para este sistema he utilizado la librería "TerminalHelper"de Gallego-Durán (2021), la cual hace posible colorear de forma sencilla y rápida en la terminal, y poder introducir parámetros por pantalla sin complicaciones.

El sistema de renderizado cuenta con su particular método "update", el cual simplemente dibuja el "sprite" de la entidad, almacenado en el componente de render, en la posición de la pantalla según lo indique su componente de físicas (x,y).

Sistema de renderizado

```
struct RenderSystem{
    void update(EntityManagerP1& EM){
        EM.forall([&](Entity&e){
            std::cout <<"\033["<< e.phy.y <<";"<< e.phy.x<<"f" ;
            std::cout << "\033["<< TERM::AT_Bold <<";"
                      << TERM::FG_Cyan <<"m";
            std::cout << e.rend.sprite;
        });
    }
};
```

Lo único que tenemos que saber del interior del lamda utilizado en el sistema anterior es lo siguiente:

- La primera línea, establece la posición del cursor donde se va a pintar posteriormente.
- La segunda línea, establece el tamaño y color del texto a dibujar.

- La tercera línea, simplemente dibuja según los anteriores parámetros, el sprite de esa entidad en su lugar.

3.2.7 Clase Game: Unión de todas las partes

Ya tenemos todas las partes programadas, falta darle funcionalidad y unión a todo, para ello vamos a crear la clase "Game", clase que creará los elementos que necesitemos y ejecutará el bucle del programa. Esta clase principalmente contendrá lo siguiente:

- Tendremos dos sistemas, el de render y el de físicas.

Clase Game

```
struct Game {
    EntityManagerP1 manager;
    void run_game(){
        PhysicsSystem system_phy{};
        RenderSystem system_rend{};
        ...
    }
};
```

- El objeto de "terminalhelper" que nos ayudará más adelante con los colores y las teclas.

Clase Game

```
struct Game {
    EntityManagerP1 manager;
    void run_game(){
        ...
        TERM::Terminal_t drawer{};
        ...
        };
    }
};
```

- El bucle del juego, el cual parará cuando "running" sea falso.

Bucle del juego

```
void run_game(){
    ...
    bool running = true;
    int timer = 100;
    while(running){
```

```
        if(timer == 0){
            system_phy.update(manager);
            timer = 100;

            if(manager.getEntityVector().size()<40){
                manager.createEntity();
            }
        }else{
            timer--;
        }
        system_rend.update(manager);
        pressKey(running, drawer);
        std::cout<< "\033[H\033[J";
    }
}
```

He añadido dentro del bucle del juego un condicional con la variable "timer", haciendo así más lento el número de veces que se ejecuta el sistema de físicas, con el objetivo de que no se ejecute en cada iteración, si no unas pocas veces, modificando así la velocidad de movimiento de las estrellas.
Dentro de este condicional también iremos creando entidades hasta un máximo de 40, es decir, 40 estrellas. Modificando este número, ajustamos la cantidad de estrellas y si cambiamos la variable "timer", cambiaremos la frecuencia con la que se ejecuta nuestro sistema de físicas.
"PressKey(running, drawer)" es una función en la clase "Game", la cual permite cerrar el juego con el escape (modifica la variable "running" y la hace falsa), y además controlar las pulsaciones del teclado de forma correcta, aunque solo detectamos el escape, más adelante será de más utilidad en otros proyectos.
Por último, el comando final en el método "run_game()" de la clase "Game", sirve para establecer el terminal fijo, haciendo que siempre se dibuje sobre el mismo fragmento de la terminal, similar a ejecutar el comando "clear" en cada iteración.

Clase Game completada

```
struct Game {
    EntityManagerP1 manager;
    void run_game(){
        PhysicsSystem system_phy{};
        RenderSystem system_rend{};
        TERM::Terminal_t drawer{};
        bool running = true;
        int timer = 100;
        while(running){
```

```
            system_rend.update(manager);
            if(timer == 0){
                system_phy.update(manager);
                timer = 100;

                if(manager.getEntityVector().size()<40){
                    manager.createEntity();
                }
            }else{
                timer--;
            }

            pressKey(running, drawer);
            std::cout<< "\033[H\033[J";
        }
    }
};
```

Con esto nuestro primer proyecto está acabado y podemos dejar la función main, limpia de la siguiente forma:

Main del proyecto

```
int main(){
    Game game{};

    game.run_game();
    return 0;
}
```

3.2.8 Resumen de lo aprendido

A continuación, la figura 10, representa una captura del funcionamiento final de nuestro "Starfield", aunque el movimiento que hemos programado no se puede apreciar en una imagen, pero servirá para visualizar el resultado.

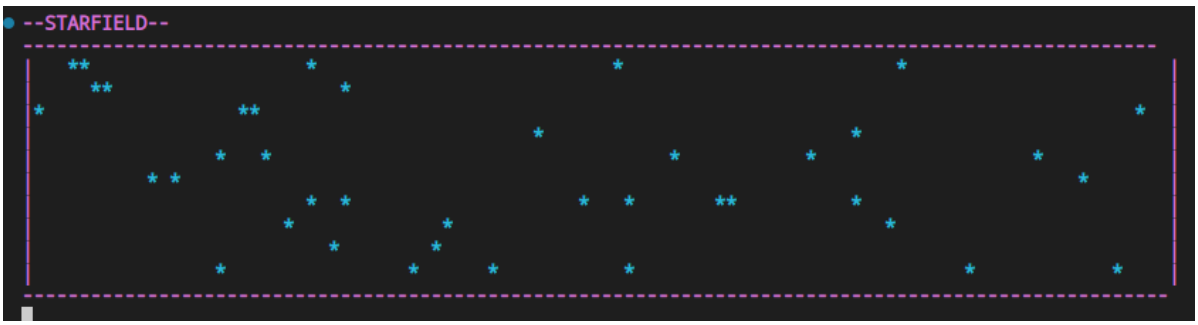


Figura 10. Starfield terminado

En este primer proyecto de la guía hemos obtenido una primera impresión de la estructura del motor ECS para conseguir un objetivo concreto, hacer que nuestras estrellas se muevan independientes una de otra. Aunque hemos creado un motor básico, hemos conseguido entender la unión entre todas las partes del ECS, hemos comprendido el concepto de las entidades y la idea de composición mediante los componentes.

Hemos aprendido a crear la clase que manejará los datos y entidades de nuestro juego, el manejador de entidades, el cual iremos reutilizando en proyectos futuros y alq ue añadiremos funcionalidades. Iteraremos sobre este motor hasta que consigamos uno completo.

En el siguiente proyecto, intentaremos avanzar en los conocimientos sobre el ECS, intentando ahondar más en la estructura de las entidades, y configurando nuevos sistemas.

El código de este proyecto se puede recopilar en la siguiente referencia: (Cantó-Berná (2023a)).

3.3 Segundo Proyecto: Firefighter Game

Si has dedicado un tiempo al primer proyecto, este segundo te resultará mucho más sencillo, ya que partimos con una base sobre cómo estructurar un motor Entity Component System.

Lo que necesitamos conseguir con este proyecto es entender de mejor manera como funcionan las entidades y componentes entre sí. Es otro proyecto básico que nos servirá para entender la estructura global del ECS y observar como se puede usar con diferentes objetivos y en diferentes ámbitos.

3.3.1 Presentación del proyecto

Este proyecto trata sobre el juego del bombero. Es un juego donde tendremos diferentes habitaciones con fuego, y dispondremos de un bombero (el jugador) que mediante movimientos hacia arriba y hacia abajo tendrá que apagar las habitaciones incendiadas en el edificio.

Este proyecto va a ser presentado de forma distinta al primero, ya que en primer lugar mostraremos el resultado, e iremos construyendo todo lo que necesitemos para obtenerlo.

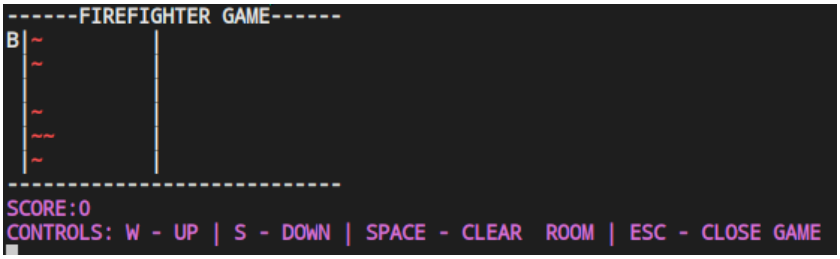


Figura 11. Juego del Bombero al ejecutar

En la primera figura, 11, podemos observar que tendremos distintas alturas como hemos mencionado anteriormente, tendremos fuegos dentro de cada habitación (las habitaciones se representarán con barras a cada lado en cada altura),y tenemos una 'B' que representará a nuestro bombero.

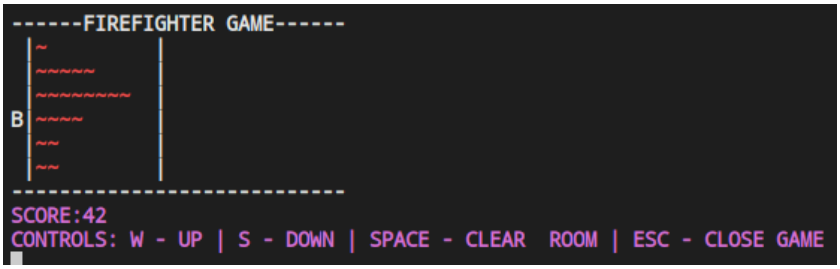


Figura 12. Juego del Bombero a mitad de partida

En la figura 12, podemos observar también el fuego en aumento según el movimiento

de nuestro bombero y una puntuación que equivale a la cantidad de fuego que hemos apagado.

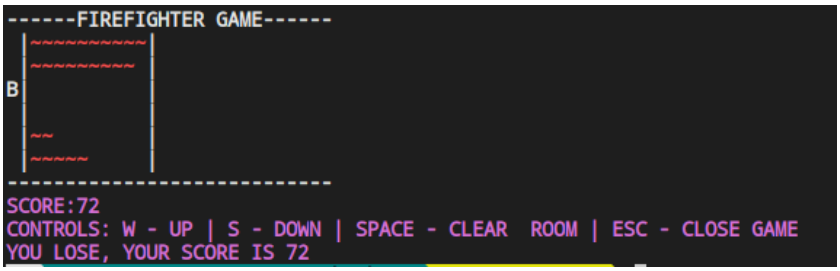


Figura 13. Juego del Bombero terminado

Por último, en la figura 13, podemos observar como el fuego ha llegado a la máxima capacidad y por lo tanto hemos perdido, obteniendo un mensaje final.

Ahora bien, una vez hemos visto el problema que proponemos, tendremos que comenzar el desarrollo.

En primer lugar, tomaremos como punto de partida el inicio del proyecto anterior, es decir, hacernos un pequeño esquema de las entidades que tendremos en este juego (figura 14), definiendo cada uno de los componentes que estas requieran.

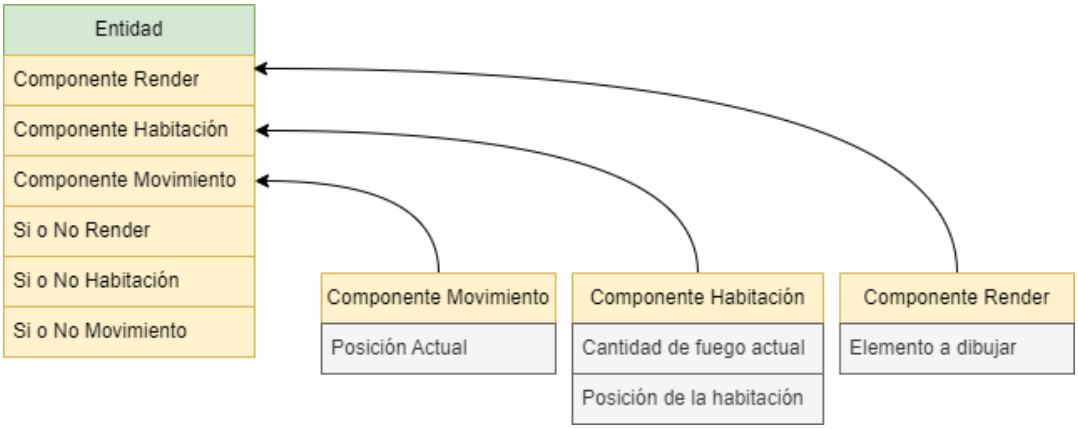


Figura 14. Diagrama de formación de entidades del Juego del Bombero

Posteriormente, podremos hacer cambios en los componentes, según nuestros objetivos cambien, pero por ahora este es un buen punto de partida para programar tanto estas entidades, como estos componentes, y los sistemas que necesitemos.

3.3.2 Creación de componentes y composición de entidades

En primer lugar definiremos componentes y entidades. Para ello, en el componente de movimiento, tomaremos la posición como un entero

Componente de movimiento

```
struct MovementComponent{
    int yPos;
};
```

En el componente de render, necesitaremos pintar tanto el fuego de las habitaciones como al personaje, para ello declararemos la variable "sprite" que será del tipo "const char*", ya que nos permite guardar caracteres, y elementos especiales.

Componente de render

```
struct RenderComponent{
    const char* sprite;
};
```

Por último, en el componente de habitación, la cantidad de fuego actual será otro entero principalmente, y la posición de la habitación también.

Componente de habitación

```
struct RoomComponent{
    int position;
    int fire;
};
```

En cuanto a las entidades, crearemos un "bool" para cada componente que nos permita saber si esa entidad tiene o no ese elemento. Con este primer gran cambio, aunque no es el más eficiente, ya que tendremos igualmente almacenado en la entidad el componente vacío, podemos diferenciarlas. Esto nos ofrece un juego muy grande a la hora de decidir que entidades quiero modificar y cuales no. En otros proyectos futuros, podremos hacer esto de diferentes formas, cada vez más óptimas y más cómodas. A continuación, todo esto quedará programado de la siguiente forma:

Composición de entidades

```
struct Entity{
    bool has_render{};
    RenderComponent render{};

    bool has_room{};
    RoomComponent room{};
```

```
    bool has_movement{};
    MovementComponent movement{};
};
```

Esta forma de declarar las entidades las hace aún más pesadas, ya que se pierden bytes entre variables, ya que al crearlas se guardan bytes de más debido a la alineación de la memoria.

3.3.3 Teoría: Alineamiento de la memoria

Los objetos en C++ tienen un tamaño y una alineación determinados por el compilador y la arquitectura de la computadora.

Es un tema importante ya que algunos procesadores requieren que los datos estén alineados en ciertas posiciones de memoria para acceder a ellos de manera eficiente. Si los datos no están alineados correctamente, se pueden producir errores de rendimiento y pérdidas de tiempo.

Por lo tanto, el compilador de C++ intenta colocar los objetos en posiciones de memoria alineadas para evitar estos errores. Por ejemplo, en una arquitectura de 32 bits, los objetos pueden requerir una alineación de 4 bytes, lo que significa que el compilador intentará colocar los objetos en una dirección de memoria múltiplo de 4.

Para lograr esto, el compilador puede insertar bytes adicionales entre los miembros de una estructura o clase para que la siguiente miembro comience en una dirección de memoria alineada. Esto se conoce como "relleno" o "padding". Por ejemplo, si una estructura tiene un miembro de 2 bytes seguido de un miembro de 4 bytes en una arquitectura de 32 bits, el compilador puede insertar 2 bytes adicionales de "relleno" para que el miembro de 4 bytes comience en una dirección de memoria múltiplo de 4.

En el caso de nuestro proyecto, tal y como hemos colocado los elementos dentro de la entidad, fuerza un alineamiento de memoria lo que tendremos bytes de más, sin embargo, si colocamos los 3 booleanos al final de la estructura juntos, en lugar de salteados y agrupados con sus respectivos componentes, implicará ahorrarnos un alineamiento por cada uno de los booleanos.

Aunque no esté tan optimizado, el objetivo es comprender la estructura y pienso que colocados intercalados y unidos a su componente es más visual, entendiéndola mejor.

3.3.4 Manejador de entidades

Para nuestro manejador de entidades, no haremos gran cosa, pues vamos a reutilizar el que hemos hecho en el ejercicio anterior, así demostraremos también que se nos permite reutilizar código, gracias a la flexibilidad que esta arquitectura de software tiene, y

aunque aún sea una clase simple, posteriormente podremos seguir reutilizándola sin problemas. Por lo tanto, la clase manejadora de entidades será:

Entity Manager

```
struct EntityManager{
    EntityManager(std::size_t size_for_entities = 10){
        entities.reserve(size_for_entities);
    }
    auto& createEntity(){ return entities.emplace_back();}
    void forall(auto&& function){
        for(auto&e:entities){
            function(e);
        }
    }
    std::vector<Entity>& getEntityVector(){
        return entities;
    }
    private:
    std::vector<Entity>  entities;
};
```

3.3.5 Creación de los sistemas

Es el momento de empezar a preparar los sistemas que usaremos. Serán un total de dos sistemas, el de renderizado y el de movimiento.

Comenzaremos hablando del sistema de movimiento, o podríamos llamarlo también de input. Este trata de detectar las pulsaciones de teclado y realizar una acción para cada tecla permitida. Tendremos que realizar un método de actualización (el método "update()") donde tendremos que decidir que entidades disponen de movimiento. Gracias a los tipo "bool" que hemos colocado en nuestra entidad, ahora podemos saber cual de ellas tiene movimiento.

Teniendo en cuenta esto, ahora en el "forall" podemos solo realizar la acción pertinente con la entidad que posea el booleano de movimiento como verdadero.

Sistema de movimiento

```
struct MovementSystem{
    void update(EntityManager& EM, bool& running,
                TERM::Terminal_t& drawer, int& score){
        EM.forall([&](Entity&e){
```

```
            if(e.has_movement == true){
                PressKey(EM, running, drawer, score);
            }
        });

    }
};
```

En este "update" recibiremos el "Entity Manager", como en todos los sistemas y en este caso recibiremos tres elementos más. En primer lugar la variable "running" ya que podremos hacer que el juego pare con una tecla como puede ser el "escape". La otra variable es un objeto de la librería que hemos usado también anteriormente, la librería "terminalhelper" de Gallego-Durán (2021), y como último parámetro el entero que corresponde a la puntuación, para cambiar la puntuación desde este sistema cuando apaguemos el fuego. Y finalmente, en el condicional que hemos colocado anteriormente, llamaremos a la función "PressKey" pasándole como parámetro todos los valores que el método "update()" recibe.

La función "PressKey" como su nombre indica, detectará el teclado y según la tecla pulsada (gracias al método de la librería "terminalhelper") realizará diferentes acciones que vamos a enumerar a continuación.

- W - Realizará el movimiento del jugador hacia arriba, siempre comprobando que la posición siguiente almacenada en el componente de movimiento no sea 0. En cada paso que el jugador realice la cantidad de fuego almacenada en el componente de habitaciones, aumentará en una habitación aleatoria sumándole a esta variable 1.

  MovementSystem::PressKey (Pulsamos W)

  ```
  ...
  case 119: // up w
  if(player.movement.yPos != 0){
      player.movement.yPos -=1;
      for(int i =0 ;i < 2; i++){
          int roomRand = rand()%6;
          EM.getEntityVector()[1+roomRand].room.fire += 1;
      }
  }
  break;
  ...
  ```

- S - La tecla S realizará exactamente lo mismo que la tecla W, pero en la dirección opuesta.

MovementSystem::PressKey (Pulsamos S)

```
...
case 115: // down s
if(player.movement.yPos != 5){
    player.movement.yPos +=1;
    for(int i =0 ;i < 2; i++){
        int roomRand = rand()%6;
        EM.getEntityVector([1+roomRand].room.fire += 1;
    }
}
break;
...
```

- Space - La tecla "space", borrará la cantidad actual de fuego en una habitación y lo sumara a la puntuación final.

MovementSystem::PressKey (Pulsamos Space)

```
...
case 32:  // remove fire space
auto& ent=EM.getEntityVector([1+player.movement.yPos];
score += ent.room.fire;
ent.room.fire = 0;
break;
...
```

- ESC - La tecla "escape" automáticamente cerrará el juego.

MovementSystem::PressKey (Pulsamos ESC)

```
...
case 27: //close game ESC
running = false;
break;
```

La función completa de "PressKey" es la siguiente:

Pulsación de tecla en el sistema de movimiento

```
void PressKey(EntityManager& EM, bool& running,
                TERM::Terminal_t& drawer, int& score){
    int tecla = drawer.wait4NextKeyPress();
    auto& player = EM.getEntityVector()[0];
```

```
        switch(tecla){
          case 119: // up w
             if(player.movement.yPos != 0){
                 player.movement.yPos -=1;
                 for(int i =0 ;i < 2; i++){
                     int roomRand = rand()%6;
                     EM.getEntityVector()[1+roomRand].room.fire += 1;
                 }
             }
             break;
          case 115: // down s
             if(player.movement.yPos != 5){
                 player.movement.yPos +=1;
                 for(int i =0 ;i < 2; i++){
                     int roomRand = rand()%6;
                     EM.getEntityVector()[1+roomRand].room.fire += 1;
                 }
             }
             break;
          case 32:  // remove fire space
             auto& ent=EM.getEntityVector()[1+player.movement.yPos];
             score += ent.room.fire;
             ent.room.fire = 0;
             break;
          case 27: //close game ESC
             running = false;
             break;
        }
    }
```

En las teclas de movimiento (W y S), el numero aleatorio oscila entre 0 y 5, ya que de forma predefinida, el número de habitaciones es 6. Esto podemos hacerlo configurable para que el jugador antes de empezar elija cuantas habitaciones quiere incluso el tamaño que quiere que tengan.

La función anterior estará incluida en el sistema de movimiento de forma privada, ya que nadie debería llamarla salvo el método "update()" de este mismo sistema.

Teniendo completo el "MovementSystem", vamos a seguir hablando del sistema de renderizado.

He colocado salidas por pantalla que mostrarán los diferentes títulos, controles, e interfaz de nuestro juego. Usando la librería nombrada anteriormente, podemos establecer diferentes grosores y colores para la salida por pantalla, escribiendo comandos de salida

como se muestra en el siguiente código.

Además recogemos la entidad del jugador ("player"), ya que la vamos a crear como primera entidad del vector de entidades como más adelante mostraré. El primer bucle recorrerá cada una de las habitaciones, donde comprobaremos si está el "player" en esta habitación, y de ser así, lo dibujamos en esa posición, si no, dibujaremos un hueco.

En los siguientes bucles recorreremos para cada habitación la cantidad de fuegos que hay, y los iremos dibujando, guardándonos las posiciones restantes sin fuego, para rellenarlas posteriormente con huecos. De esta forma, obtendremos la habitación llena con la cantidad de fuego correspondiente. Finalmente restablecemos los valores de la variable "room_size", para la siguiente habitación.

Sistema de renderizado

```
struct RenderSystem{

    void update(EntityManager& EM, int score){
        //PINTAMOS INTERFAZ
        std::cout << "\033["<< TERM::AT_Bold <<";"<< TERM::FG_White
            <<"m------FIREFIGHTER GAME------\033["<< TERM::BG_Default
            <<"m\n";

        auto& player = EM.getEntityVector()[0];
        for(int i=0; i < EM.getEntityVector().size() - 1 ; i++){
            //DIBUJAMOS EL PLAYER
            if(player.movement.yPos==i){
                std::cout<< player.render.sprite;
            }else{
                std::cout<< " ";
            }
            //DIBUJO PRIMERA BARRA DE LA HABITACION
            std::cout<< "\033["<< TERM::AT_Bold <<";"
                << TERM::FG_White <<"m|";
            // EL SIZE DE LA HABITACION ES 10
            int room_size =10;
            //PINTAMOS FUEGO DENTRO DE CADA HABITACION
            for(int j=0; j<EM.getEntityVector()[i+1].room.fire;j++){
                std::cout<< "\033["<< TERM::AT_Bold <<";"
                    << TERM::FG_Red <<"m"
                    << EM.getEntityVector()[i+1].render.sprite;
                room_size--;
            }
            //RELLENAMOS DE HUECOS LO QUE NO SE HA PINTADO DE FUEGO
```

```
            for(int j=0; j<room_size;j++){
                std::cout<< " ";
            }
            room_size = 10;
            //PINTAMOS FINAL DE LA HABITACION
            std::cout<< "\033["<< TERM::AT_Bold <<";"
                << TERM::FG_White <<"m|\n";
        }
        //PINTAMOS INTERFAZ FINAL Y RESULTADOS
        std::cout<< "\033["<< TERM::AT_Bold <<";"
            << TERM::FG_White <<"m----------------------------\n";
        std::cout << "\033["<< TERM::AT_Default <<";"
            << TERM::FG_Magenta<<"mSCORE:" << score <<  "\n";

    }
};
```

Para concluir con el código del programa, tendremos que unir todos los cabos sueltos que tenemos, y estructurar bien una clase "Game" o similar donde tengamos los principales objetos, donde se creen nuestras entidades y se ejecuten nuestros "updates".

En primer lugar crearemos la clase y un método "run" dentro, al cual llamaremos desde nuestro "main" para ejecutar este juego. Este método creará un objeto para cada sistema, y el objeto de la librería de dibujado "TERM::Terminal_t". Además crearemos la variable "score", igualada a 0, ya que estará presente durante todo el juego. Esta clase es el lugar donde crearemos también el "Entity Manager".

3.3.6 Creamos la clase Game

El siguiente paso será crear nuestras entidades, y como en este caso, no son iguales todas, que tenemos diferentes tipos, crearemos otro método a parte para este propósito. Este método creará la primera entidad, que será el player, ajustando su "sprite" a 'B' y los booleanos "rendCMP" y "movementCMP" verdaderos. Se crearán 6 más que serán las habitaciones, ajustando el fuego inicial como aleatorio entre 0 y 3, la posición que tendrá en el "edificio" digamos y el "sprite" de fuego "~". Por último, los booleanos "roomCMP" y "rendCMP" de la entidad los pondremos en verdadero.

Creación de entidades

```
    void createEntitiesForFirefighterGame(){
        auto& player = manager.createEntity();
        player.render.sprite="B";
        player.has_render = true;
        player.has_movement = true;
```

```
        for(int i=0; i < 6; i++){
            auto& room_ENTITY = manager.createEntity();
            room_ENTITY.room.fire= rand()%3;
            room_ENTITY.room.position = i;
            room_ENTITY.render.sprite= "~";
            room_ENTITY.has_room = true;
            room_ENTITY.has_render = true;
        }

    }
```

Simplemente con estos booleanos, hemos creado entidades distintas, que aunque tengan todos los componentes en su interior, hemos configurado y elegido cuales usar.

Seguiremos hablando de este método "run" de la clase "Game", donde lo siguiente que encontramos es el bucle del juego. El primer paso dentro de este bucle es comprobar si hay alguna sala llena de fuego, lo que supondría perder la partida y salir de este bucle. Esto lo comprobamos con otra función auxiliar, la cual he llamado "checkRooms", donde podemos comprobar si el número de fuego en todas las habitaciones supera a 9. Si hay una que lo supera eso quiere decir que se ha quemado y el juego acaba. Es la siguiente función, dentro de la clase "Game".

Comprobación para seguir jugando

```
    bool checkRooms(bool& running){
        for(int i =0 ; i<manager.getEntityVector().size()-1;i++){
            if(manager.getEntityVector()[i+1].room.fire > 9){
                running = false;
                return true;
            }
        }
        return false;
    }
```

Por último, los métodos "update" de ambos sistemas, colocando primero el de render para dibujar y luego mover. El comando para que la pantalla sea estática y no avance hacia abajo todo lo que pintamos en cada iteración ("clear"), y si el bucle termina, es decir el juego ha terminado, ejecutamos una última vez el sistema de render fuera del bucle del juego para lograr terminar la ejecución con el resultado visual en pantalla.

En el método "run" hablaremos del bucle del juego. En primer lugar comprobaremos si las habitaciones son correctas como hemos mencionado anteriormente, y después actu-

alizaremos render y sistema de movimiento. Una vez se acaba el bucle, actualizamos el sistema de render una última vez para exponer el resultado.

Este método "run" que hemos descrito anteriormente, qudará de la siguiente forma:

Game::run(EntityManager& manager)

```
    void run(){
        MovementSystem movSys;
        RenderSystem rendSys;
        TERM::Terminal_t terminal_drawer{};
        int score=0;
        bool running = true;
        createEntitiesForFirefighterGame();
        while(running){
            if(checkRooms(running)){
                break;
            }
            rendSys.update(manager, score);
            movSys.update(manager, running, terminal_drawer, score);
            std::cout<< "\033[H\033[J";
        }
        rendSys.update(manager, score);
        std::cout<<"YOU LOSE, YOUR SCORE IS "<< score <<"\n";
    }
```

Hemos terminado nuestro prototipo, ahora ya podemos disfrutar de nuestro juego, pero no antes sin colocar en el "main" del proyecto el objeto "Game" que ejecutará nuestro juego mediante el método "run()".

Método Main del programa

```
int main(){
    Game game;
    game.run();
    return 0;
}
```

3.3.7 Resumen de lo aprendido

Hemos terminado el proyecto. La figura 15, representa el juego terminado, en el cual hemos aprendido conceptos similares a los conceptos del proyecto anterior. La diferencia más importante a remarcar es la aparición de elementos para comprobar los componentes que una entidad posee.

Entendiendo estos conceptos, representar entidades y componentes, será mucho más sencillo de entender posteriormente. Esta estructura, la podemos simplificar, pero antes de las optimizaciones y dejar bonito nuestro código, hay que comprender el por qué de las cosas.

En resumen, en este proyecto hemos aprendido:

- Entidades con elementos de diferenciación para los componentes.
- La estructura y unión de las partes de un ECS.
- Nuevos sistemas, como el de movimiento.
- Manejador de entidades, que crea entidades y las maneja.

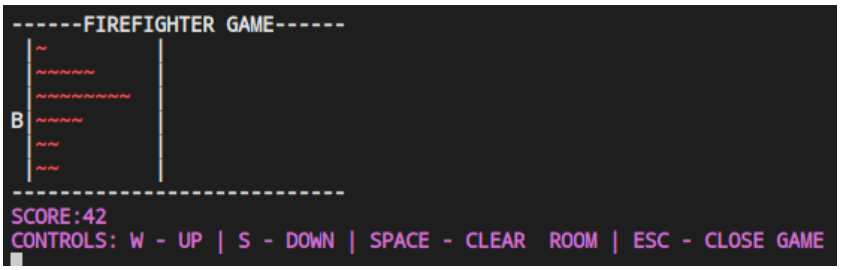


Figura 15. Proyecto Finalizado

El código de este proyecto se puede encontrar en la siguiente referencia: Cantó-Berná (2023b).

4 Introducción a RayLib para nuestros proyectos

En este apartado, vamos a hacer una pequeña pausa respecto al avance de nuestro objetivo principal, y vamos a dedicarlo a explicar cómo se utiliza la librería RayLib, de Santamaria (2013), la cual usaremos en proyectos futuros para el diseño gráfico de estos.

Después de la introducción a la librería, nos basaremos en proyectos para que sea fácil comprender su uso, en este caso, reharemos el juego del bombero, pero esta vez con gráficos, este será nuestro tercer proyecto.

En la tabla 2, se recogen los proyectos de este apartado que en este caso será solo uno.

	Título	Descripción corta
Proyecto 3	Firefighter Game RayLib	Introducción a RayLib y flexibilidad del ECS

Tabla 2. Proyecto que abarcaremos en este apartado

Hay que tener en cuenta, que el código del juego será el mismo y lo único que cambiará será el componente de render, el sistema de renderizado y la forma en la que detectamos las pulsaciones del teclado, por lo tanto este apartado será un poco diferente al resto, ya que primero hablaremos de la función o funciones de RayLib y a continuación expondremos como hemos modificado el código principal.

Si se desea ir directamente a la implementación, ir al apartado 4.2.

Figura 16. Icono RayLib

4.1 ¿Qué es RayLib?

RayLib es una biblioteca de programación de software libre y multiplataforma diseñada para facilitar la creación de videojuegos y aplicaciones multimedia. Es una herramienta de desarrollo de software que proporciona una amplia gama de funciones para la creación de gráficos, sonidos y entradas de usuario, lo que permite a los desarrolladores centrarse en la lógica de su juego en lugar de preocuparse por los detalles de bajo nivel de la programación.

Está escrita en lenguaje C y es compatible con varios lenguajes de programación como C++, C#, Java y Python entre otros, lo que la hace accesible para una amplia gama de desarrolladores. Además, RayLib es compatible con varias plataformas, incluyendo Windows, macOS, Linux, Android, y iOS, lo que la hace ideal para crear juegos y aplicaciones que se pueden ejecutar en múltiples dispositivos.

Entre las características más notables de esta librería se incluyen la gestión de ventanas y pantallas, el dibujo de formas básicas y texturas, la manipulación de imágenes, la reproducción de sonidos, el soporte de teclado, ratón y gamepad, y la detección de colisiones. También es fácil de usar, con una documentación completa y una comunidad de desarrolladores activa que puede proporcionar ayuda y soporte.

4.1.1 Módulos que la componen

Esta librería está compuesta por distintos módulos que hacen de ella una librería completa:

- Core: Este módulo es el corazón de RayLib y proporciona funciones básicas como la gestión de ventanas, el control de tiempo y el procesamiento de entradas de usuario.
- Shapes:Proporciona funciones para el dibujo de formas geométricas básicas, como líneas, rectángulos y círculos, así como para el dibujo de polígonos personalizados.
- Textures: Sirve para cargar y manipular texturas, incluyendo la creación de texturas a partir de imágenes y la generación de texturas procedurales.
- Audio: Este módulo proporciona funciones para la carga y reproducción de sonidos y música, así como para el control del volumen y la posición espacial de los sonidos.
- Models: Este módulo proporciona funciones para la carga y manipulación de modelos 3D, incluyendo la creación de modelos a partir de archivos y la generación de modelos procedurales.
- Shaders: Este módulo proporciona funciones para la creación y uso de shaders, que son programas especiales que se ejecutan en la tarjeta gráfica para realizar cálculos complejos en los gráficos.
- Raymath: Este módulo proporciona funciones matemáticas avanzadas para la realización de cálculos vectoriales y matriciales, que son necesarios para muchas operaciones en gráficos 3D.

4.2 Creamos una aplicación

En primer lugar, al no usar la terminal para mostrar nuestros juegos, necesitaremos una ventana gráfica donde mostrar todo el contenido. Para eso usamos la función "InitWindow()":

RayLib - InitWindow()

```
void InitWindow(int width, int height, const char *title);
```

Esta función crea una ventana a partir del ancho y alto, y un título que se mostrará en la parte superior de la aplicación.

Al igual que abrimos la ventana, necesitamos poder cerrarla cuando los procesos terminen, es por esta razón que al final del programa, o cuando deseemos cerrar la ventana, tendremos que ejecutar la función "CloseWindow()":

RayLib - CloseWindow()

```
void CloseWindow(void);
```

Esta función no recibe parámetro y simplemente cerrará la ventana del proyecto.

Disponemos de diversas funciones para cambiar el flujo y opciones de la ventana o aplicación, pero las dos anteriores son las que utilizaremos junto a una tercera que decidirá si la ventana tiene que cerrarse. Esta función será "WindowShouldClose":

RayLib - WindowShouldClose()

```
bool WindowShouldClose(void);
```

La función anterior, no recibe parámetros pero devuelve un valor verdadero si ha ocurrido algo en el programa que requiere del cierre de la ventana y falso en cualquier otro caso, por lo que no se cerrará la ventana. En un juego, esta función puede estar en el condicional del bucle del juego, haciendo que este se ejecute mientras queramos.

Un detalle para finalizar es que podemos establecer a cuantos frames por segundo como máximo se ejecute nuestro programa. Esto lo podemos hacer con la función "SetTargetFPS()":

RayLib - SetTargetFPS()

```
void SetTargetFPS(int fps);
```

Nos permite pasar un parámetro que corresponderá al número de frames por segundo que queramos.

4.2.1 Creamos la ventana: Firefighter Game

Para pasar a ventana nuestro juego el primer paso es crear la ventana y ejecutar entre las dos funciones "InitWindow" y "CloseWindow" el bucle del juego.

main.cpp

```
int main(){
    Game game{};
    InitWindow(700, 400, "FIREFIGHTER GAME");
```

```
    game.run();
    CloseWindow();
    return 0;
}
```

El método "run()" de el objeto Game será exactamente igual que teníamos en el proyecto del bombero principal, simplemente añadiremos "SetTargetFPS(60)" antes del bucle para configurar los frames por segundo de nuestro juego.Otro detalle a cambiar es la forma de dibujado y el sistema de renderizado. Esto lo veremos en el siguiente apartado.

Antes de continuar, otra función que podemos utilizar para obtener un código más completo es comprobar si la ventana sigue abierta antes de cerrarla, ya que podríamos tener problemas si no lo comprobamos. Esto se hace mediante la función "IsWindowReady()". Por lo tanto quedará de la siguiente forma:

main.cpp

```
int main(){
    Game game{};
    InitWindow(700, 400, "FIREFIGHTER GAME");
    game.run();
    if(IsWindowReady())
        CloseWindow();
    return 0;
}
```

4.3 Dibujamos un Sprite

Para dibujar algo en la pantalla que hemos creado anteriormente hacen falta principalmente 3 funciones clave de RayLib y además crear las texturas pertinentes en cada situación.

Comenzaremos aprendiendo a crear Texturas o Sprites, para ello necesitaremos crear 2 objetos distintos, los cuales irán vinculados el uno con el otro. Estos objetos son "Texture2D" y "Rectangle".

Con Texture2D podemos definir una textura y necesitaremos la función "LoadTexture()" para definirla y crearla. El siguiente código muestra como podemos crear un sprite:

Creando nuestro primer sprite

```
//Texture2D LoadTexture(const char *fileName);
Texture2D sprite = LoadTexture("img/bombero.png");
```

La función "LoadTexture" devuelve una "Texture2D" que guardaremos en nuestro sprite.

A continuación, teniendo preparado el sprite, podemos definirle un tamaño y posición, esto lo podemos hacer mediante el objeto "Rectangle".

Creando nuestro primer sprite. Parte 2

```
Rectangle box = {0,0,static_cast(sprite.width),
                        static_cast(sprite.height)};
```

Esto creará una caja invisible del tamaño de nuestro sprite, para posicionarlo, escalarlo o hacer cualquier transformación sobre este. Otra utilidad es definir que sprite coger de una imagen donde existan varios. Con "Rectangle" además podemos calcular colisiones, lo que veremos en el apartado 5.1.7.

Para cada textura que leamos o carguemos al programa, tendremos que descargarla o quitarla del programa, esto se hace con la función "UnloadTexture":

Eliminando texturas

```
    void UnloadTexture(Texture2D texture);
```

Ahora tenemos que ver como dibujar en la pantalla sprites, y para ello hace falta hacerlo en un ámbito de dibujado. Este espacio se crea y se destruye con dos funciones, "BeginDrawing" y "EndDrawing".

"BeginDrawing" lo colocaremos antes de ejecutar el sistema de renderizado, que es el espacio donde dibujaremos en la pantalla y después de que esto suceda, colocaremos "EndDrawing"

Bucle del juego (sección de dibujado)

```
    while (!WindowShouldClose()){
        BeginDrawing();
            //RENDER UPDATE
        EndDrawing();
    }
```

Entre estas llaves, podemos dibujar cualquier cosa, pero podemos tener problemas si antes de redibujar no borramos lo que anteriormente había. Esto lo podemos hacer con una nueva función que se denomina "ClearBackground":

Borramos el fondo de la aplicación

```
void ClearBackground(Color color);
```

RayLib tiene definidos un montón de colores, aun así podemos definir un color de forma muy sencilla, en este caso podemos usar "RAYWHITE" como parámetro de esta función y borrar el fondo de la aplicación. Por lo tanto, en el interior del bucle del juego nos quedará una estructura similar a esta:

Bucle del juego: sección de dibujado

```
while (!WindowShouldClose()){
    BeginDrawing();
        ClearBackground(RAYWHITE);
        //RENDER UPDATE
    EndDrawing();
}
```

Por último antes de continuar implementando todos estos conceptos en nuestro juego, tenemos que aprender a dibujar en nuestra ventana. Para ello usaremos la función "DrawTextureRec":

Dibujamos en la pantalla

```
void DrawTextureRec(Texture2D texture, Rectangle source,
                    Vector2 position, Color tint);
```

Esta función dibujará la textura pasada por el primer parámetro cuyo rectángulo de tamaño es el pasado como segundo parámetro. También deberemos proporcionarle la posición en el tercer parámetro y un tinte que aplicar sobre la textura, el cual si es blanco no modificará el color original.

4.3.1 Creando nuestros propios Sprites y dibujando en nuestro juego

En primer lugar tendremos que modificar el componente de render que tenemos en el juego del bombero que simplemente era un caracter:

Sprite anticuado

```
struct RenderComponent{
    const char* sprite{};
};
```

Usaremos el material explicado en el apartado enterior, y este componente quedará de la siguiente forma:

Creando nuestro primer sprite

```
struct RenderComponent{
   RenderComponent();
   Texture2D sprite;
   Rectangle box;
};
RenderComponent::RenderComponent(){
   sprite = LoadTexture("img/bombero.png");
   box = {0,0, static_cast(sprite.width), static_cast(sprite.height)};
}
```

Ahora ya podemos proceder para dibujarlo en la pantalla.Para lograrlo tenemos que ir al "update" de nuestro sistema de render el cual renderizará nuestro juego modificando cada impresión de texto por terminal por la llamada a la función de dibujado "DrawTextureRec".

He realizado una estructura auxiliar para crear los distintos sprites que tendrán las paredes, el fuego y demás objetos del juego y los he colocado en un "struct Map". Cada elemento en el interior de este "Map" a su vez es una struct que contiene un sprite y un rectángulo, como hemos visto para el caso del componente de renderizado.

Este Mapa será el siguiente:

Mapa de sprites

```
struct Map{
    Background background{};
    Wall1 wall1{};
    Wall2 wall2{};
    NoFire nofire{};
    Rope rope{};

    ~Map(){
        UnloadTexture(background.sprite);
        UnloadTexture(wall2.sprite);
        UnloadTexture(wall1.sprite);
        UnloadTexture(nofire.sprite);
        UnloadTexture(rope.sprite);
```

```
    }
};
```

Una vez definido el mapa, finalmente podremos dibujar nuestras entidades y elementos extra. Para ello simplemente ejecutaremos la función "DrawTextureRec" para cada elemento.

Remplazando cada salida por pantalla, por el siguiente código, conseguiremos el sistema de render actualizado y listo para dibuajar.

Dibujando al player

```
DrawTextureRec(player.render.sprite, player.render.box,
            (Vector2){0,player.movement.yPos*row_size},WHITE);
```

Aquí está el ejemplo del jugador. Pasamos el sprite al primer parámetro y su "Rectangle" que hemos llamado "box" al segundo. Creamos un Vector2 y recogemos la posición del jugador del componente pertinente, en este caso el de movimiento ya que guarda en que posición estamos. Por último el filtro blanco para no modificar el sprite original.

Este puede ser otro ejemplo, el dibujado del fuego, el cual se realiza tantas veces como fuegos haya en el componente "room".

Dibujando fuego

```
DrawTextureRec(e.render.sprite, e.render.box,
            (Vector2){col_size*(j+2),e.room.position*row_size},WHITE);
```

Los elementos "col_size" y "row_size" son elementos creados para medir el espacio de la ventana, el cual he dividido en filas y columnas para que sea más fácil el posicionamiento de sprites a la hora del dibujado. Esta función se encuentra dentro del bucle de dibujado de fuego.

4.4 Dibujado de texto

Para dibujar texto contamos con diferentes funciones según el objetivo que tengamos. A continuación vamos a explicar un ejemplo de dibujado de texto en la pantalla, con el código mediante el cual lo hago en el juego.

Necesitaremos usar la función de RayLib "DrawText":

Método que dibuja texto

```
void DrawText(const char *text, int posX, int posY,
              int fontSize, Color color);
```

Este método dibuja lo que le pasemos como primer parámetro en la posición 'X' e 'Y' que coloquemos en el segundo y tercer parámetro. Además el cuarto parámetro es el tamaño de la fuente con la que vamos a dibujar y por último el color que usaremos.

En primer lugar conseguiremos pasar el texto que queramos a "const char*" y una vez lo tengamos dibujaremos. Para ello, conseguiremos esto usando el elemento "ostringstream" de la siguiente forma:

Conseguimos la cadena a dibujar

```
std::ostringstream stream;
stream << std::fixed << "YOU LOSE --- ESC TO EXIT";
std::string str = stream.str();
const char* pointsString = str.c_str();
```

A continuación definiremos el color que queremos utilizar:

Creamos el color a emplear en el dibujado

```
Color color{100,111,100,255};
```

Por último, dibujamos el texto escrito anteriormente, usando todos los parámetros definidos para componer la función de dibujado:

Dibujamos el texto

```
DrawText(pointsString, 150,190, 30, color);
```

4.5 Otras funciones interesantes

RayLib está compuesta por una cantidad de funciones inmensa, a continuación voy a exponer unas cuantas que aunque no las he nombrado pueden sernos de gran utilidad.

En primer lugar tenemos funciones que nos permiten conocer en cualquier momento cuál es el estado de la ventana:

Estado de la ventana

```
bool IsWindowFullscreen(void);
bool IsWindowHidden(void);
bool IsWindowMinimized(void);
bool IsWindowMaximized(void);
bool IsWindowFocused(void);
bool IsWindowResized(void);
void MaximizeWindow(void);
void MinimizeWindow(void);
```

Además de estas disponemos de gran variedad de teclas para controlar la entrada por teclado, lo cual es muy útil ya que disponemos de numerosos estados de las teclas:

Teclado

```
bool IsKeyPressed(int key);
bool IsKeyDown(int key);
bool IsKeyReleased(int key);
bool IsKeyUp(int key);
int GetKeyPressed(void);
```

Algunas de estas funciones han sido incluidas en el sistema de input para controlar el numero de veces que pulsamos una tecla. Por ejemplo, "IsKeyDown", la cual usamos en todo momento para saber si ejecutar el código de la tecla pulsada.

RayLib también cuenta con todas las teclas con su valor numérico ASCII agrupadas en un Enum que se llama "KeyboardKey".

Teclas

```
KeyboardKey::KEY_SPACE        = 32
KeyboardKey::KEY_S            = 83
KeyboardKey::KEY_W            = 87
...
```

También dispone de funciones para el manejo de audio. Aunque no las he utilizado, se puede definir, cerrar, controlar el estado y configurar el volumen de una pista de sonido.

Sonido

```
void InitAudioDevice(void);
void CloseAudioDevice(void);
```

```
bool IsAudioDeviceReady(void);
void SetMasterVolume(float volume);
```

Todas estas funciones y muchas otras no incluidas, se pueden encontrar en la página oficial de la librería RayLib, de Santamaria (2013), en el apartado "CheatSheet" donde están todas los métodos de la librería explicados.

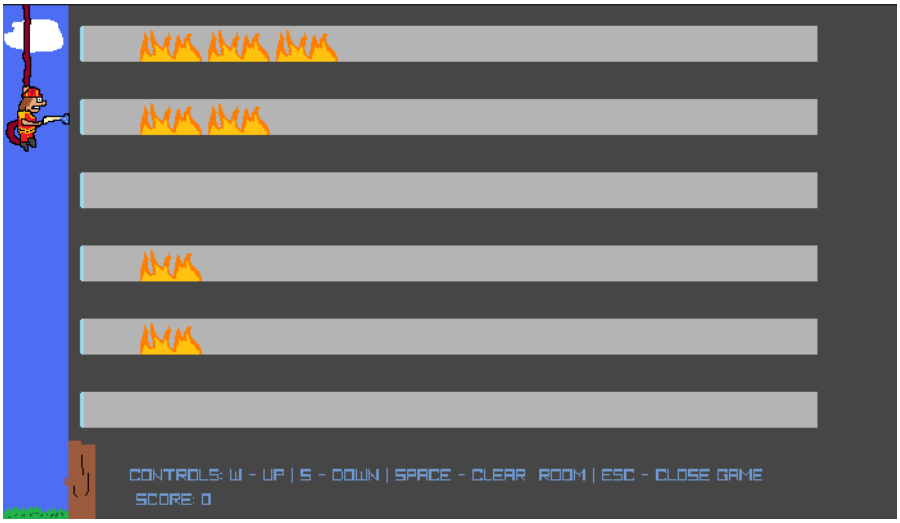
4.6 Resumen de lo aprendido

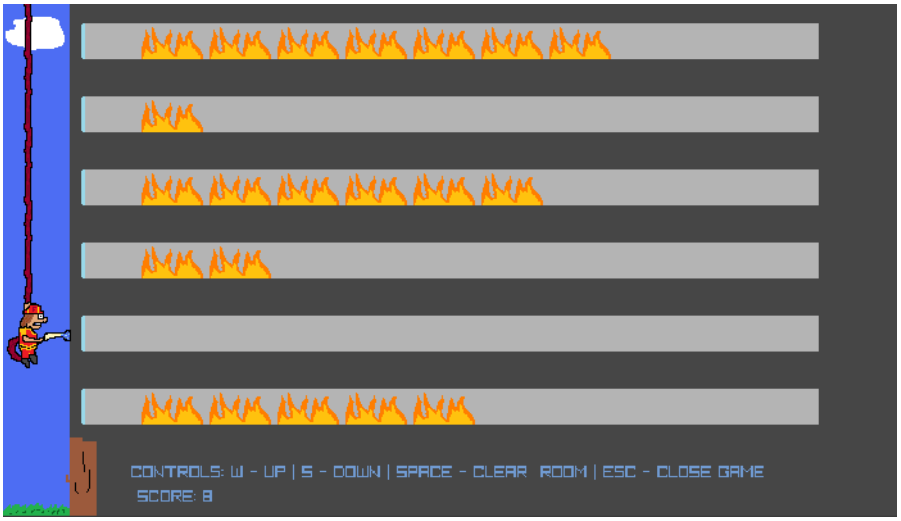
Finalmente, este apartado termina con el proyecto del juego del bombero el cual ya hemos realizado anteriormente, implementado usando la librería RayLib, principalmente para el dibujado de texturas y sprites.

La curva de aprendizaje en este proyecto ha sido grande, ya que aunque nos hayamos desmarcado del objetivo principal para introducir la librería gráfica que utilizaremos, hemos aprendido una inmensidad de funciones de esta, y a la vez hemos observado las facilidades de las que disponemos para cambiar elementos de nuestro juego si usamos un motor ECS.

Todos los sprites utilizados en este proyecto son hechos por mi, y su único objetivo es representar de forma sencilla el juego para poder dibujarlo en la ventana.

Estas imágenes que se muestran a continuación representan el resultado final de este proyecto, el cual ha servido como explicación de uso de RayLib. El código de este proyecto se puede encontrar en la referencia, Cantó-Berná (2023c).




CONTROLS: W - UP | S - DOWN | SPACE - CLEAR ROOM | ESC - CLOSE GAME
SCORE: 8


YOU LOSE -- ESC TO EXIT
CONTROLS: W - UP | S - DOWN | SPACE - CLEAR ROOM | ESC - CLOSE GAME
SCORE: 8

5 Diferenciación de componentes

Una vez llegado a este punto vamos a repasar cómo tenemos actualmente la estructura de nuestro ECS, tal cual la hemos dejado en los apartados anteriores, a la que añadiremos y cambiaremos funcionalidades durante este y los siguientes apartados.

En primer lugar tenemos entidades las cuales contienen componentes que las forman. Es decir, todas las entidades tienen todos los componentes y además tenemos unas variables booleanas con cada componente para saber si la entidad lo usa o no. Luego tenemos los sistemas, que mediante el manejador de entidades, reciben las entidades necesarias para el cambio de los datos en cada una de estas estructuras. En el diagrama 17 se puede observar esta estructura de forma gráfica.

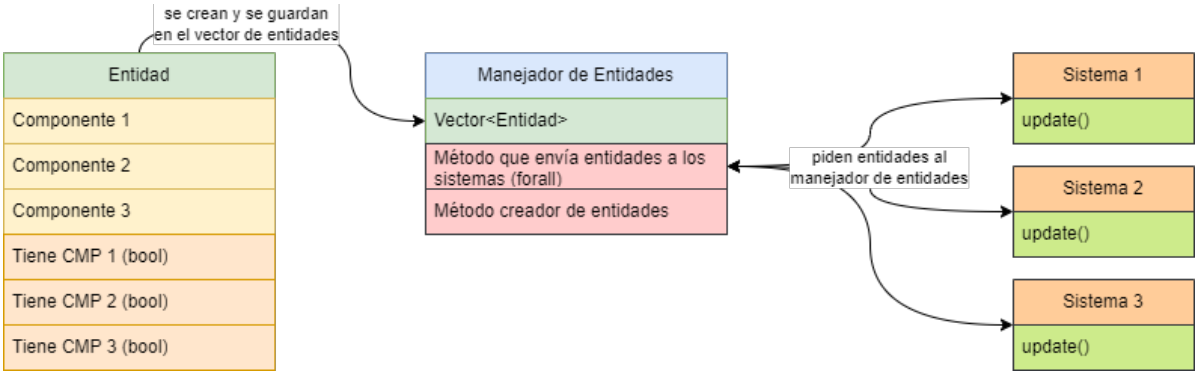


Figura 17. Diagrama estructural del ECS hasta el momento.

Como se observa en el diagrama, el manejador de entidades guarda las entidades que se crean, las cuales incluyen los datos de los componentes (por ahora), y los sistemas a su vez piden entidades y reciben entidades para cambiar los datos de los componentes.

Una vez hemos recordado el contexto actual, vamos a proceder con el siguiente proyecto, en el cual ampliaremos la funcionalidad de nuestro manejador de entidades, ya que haremos que las elimine. A su vez, mejoraremos o veremos de otra forma la creación de entidades y sus componentes, usando "std::optional". Además lo representaremos de forma gráfica, así tendremos una primera visión real de posibles componentes.

A continuación, se muestra una tabla (3) que representa los datos del siguiente proyecto.

	Título	Descripción corta
Proyecto 4	Save the OVNI	Eliminamos entidades y diferenciamos componentes

Tabla 3. Proyecto que abarcaremos en este apartado

Este proyecto incluirá contenido gráfico, mediante la librería RayLib, Santamaria (2013).

5.1 Cuarto proyecto: Save the OVNI

"Save the OVNI" es el típico juego de "scroll lateral", donde iremos moviéndonos para esquivar obstáculos y enemigos. Nuestra puntuación depende de la cantidad de distancia que consigamos recorrer sin morir. Para plantear este proyecto, podemos hablar de otro juego muy conocido como es el minijuego del dinosaurio de Google cuando perdemos la conexión a internet; el cual cuenta con un jugador que tendrá que ir esquivando obstáculos, figura 18. Es el mismo concepto, pero en nuestro caso podremos movernos por toda la pantalla mientras esta avanza.

Figura 18. Famoso minijuego de Google

A continuación mostraré una imagen para hacernos una idea de la composición que le he dado al juego y entender el diagrama de la estructura de datos de nuestro motor ECS, donde definiremos los componentes necesarios.

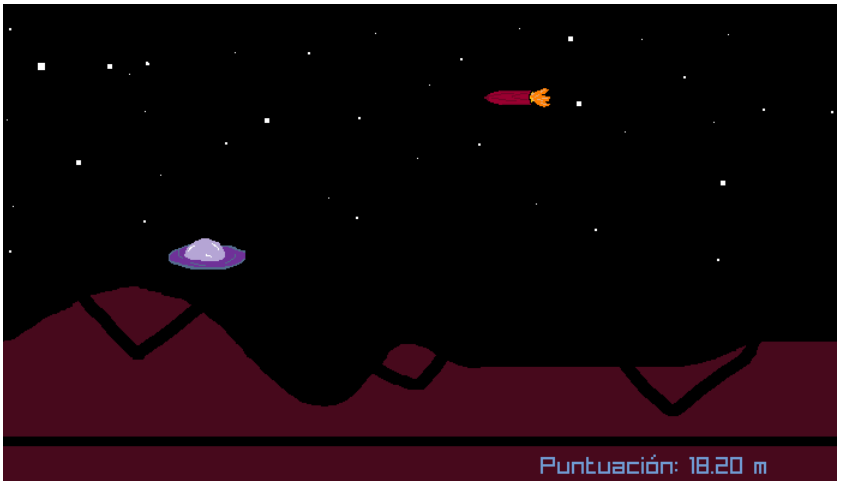


Figura 19. Primera imagen del ejempolo del apartado 4.

En la figura 19, se puede apreciar el juego en ejecución y principalmente tenemos que fijarnos en los elementos que aparecen. Aparece un OVNI, que será nuestro jugador. También observamos un misil por la parte superior. Por último, la puntuación, aunque menos importante a la hora de programar el funcionamiento de nuestro ECS.

Con estos elementos podemos empezar a diseñar nuestra estructura de entidades y componentes, teniendo en cuenta que la pantalla se mueve de derecha a izquierda y los misiles siguen esta dirección y sentido. También tendremos en cuenta que los enemigos aparecerán de forma aleatoria desde el lado derecho. Empecemos hablando de que

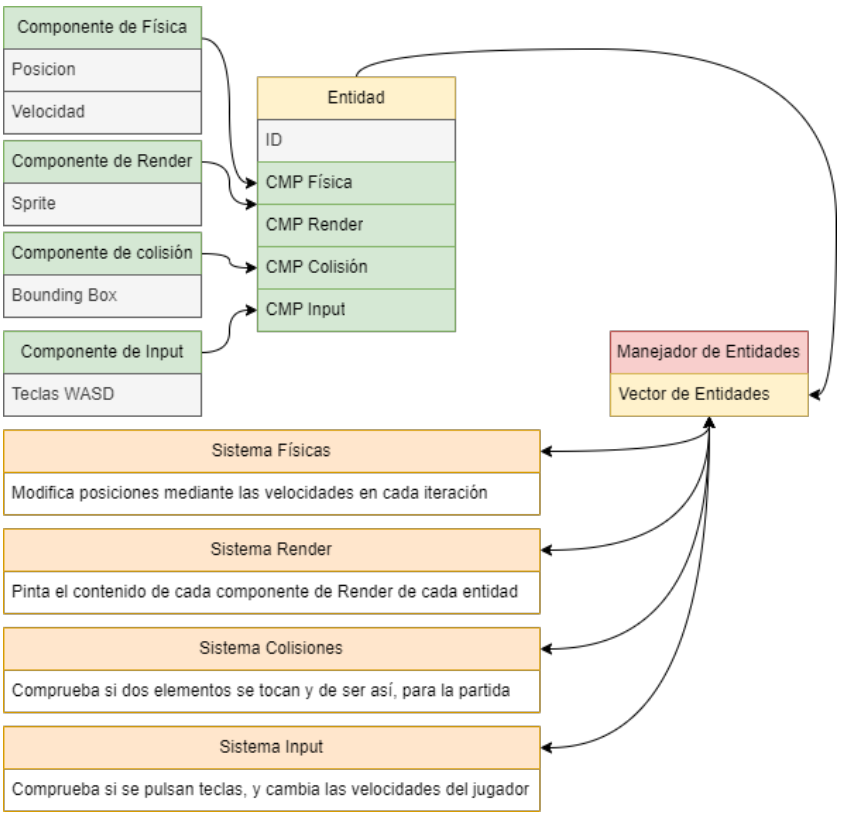


Figura 20. Diagrama de estructura de Entidades, Componentes y Sistemas.

necesitan tener nuestras entidades.

- Componente de física: necesitaremos posicionar elementos y hacer que estos se muevan.
- Componente de render: necesitaremos pintar en pantalla, y como hemos observado, sprites.
- Componente de movimiento o input: para mover a nuestro jugador.
- Componente de colisiones: Si chocamos con un enemigo, necesitaremos que algo ocurra.

A parte de todos estos componentes, necesitaremos también cuatro sistemas:

- Sistema de físicas: actualizaremos datos de posiciones y velocidades.
- Sistema de renderizado: pintaremos en pantalla los sprites que se guarden en los componentes de render.
- Sistema de input: según las teclas pulsadas, cambiará las velocidades de nuestro jugador.
- Sistema de colisiones: comprobaremos si hemos colisionado con algo y si es así, moriremos, es decir, pararemos la partida.

Ahora que hemos nombrado todo el material principal con el que tenemos que trabajar, vamos a hacer un diagrama 20 donde quede todo plasmado. Además, en el diagrama comenzaremos a definir qué contiene cada elemento.

En este punto del proyecto, ya podemos ir viendo como con diferentes proyectos, simplemente definimos y dedicamos un pequeño tiempo a los componentes, entidades y sistemas, y conseguimos el funcionamiento de forma rápida y eficaz.

5.1.1 Creación de componentes y composición de entidades

Comenzaremos con el código. En primer lugar los componentes. Vamos a definirlos.

El componente de físicas tiene una modificación respecto a los componentes de proyectos anteriores, simplemente a la hora de programar. Definiremos posición y velocidad con "std::pair<float,float>", simplemente por tener todavía más ordenados nuestros datos. Con este tipo de datos, tendremos que tener en cuenta que el primer valor será 'x', tanto de velocidad como de posición, y el segundo será 'y'.

Componente de físicas

```
struct PhysicCMP{
    PhysicCMP();
    std::pair<float,float> position;
    std::pair<float,float> velocity;
};
```

El componente de render, en este proyecto se basa en la librería RayLib de Santamaria (2013), ya que será nuestro motor gráfico para este y futuros proyectos. No dedicaremos demasiado tiempo a saber como usarlo, porque hemos dedicado el capítulo 4 para introducirlo, pero para dibujar necesitaremos definir una textura y un tamaño, y eso es lo que hacemos en este componente.

Componente de render

```
struct RenderCMP{
    RenderCMP();
    Texture2D sprite;
    Rectangle box;
};
// Podemos definir por defecto un componente de render de
// la siguiente forma
RenderCMP::RenderCMP(){
    sprite = LoadTexture("img/player.png");
    box    = {0,0, static_cast<float>(sprite.width),
```

```
                static_cast<float>(sprite.height)};
}
```

Seguimos con el componente de Input, guardando el código ASCII de la tecla en su variable. Estotambién nos lo ofrece la librería RayLib.

Componente de input

```
struct InputCMP{
    InputCMP();
    int KeyW, KeyA, KeyD, KeyS;
};

InputCMP::InputCMP(){
    KeyA = KEY_A; // KEY_A = 65 (ASCII)
    KeyD = KEY_D; // KEY_D = 68 (ASCII)
    KeyS = KEY_S; // KEY_S = 83 (ASCII)
    KeyW = KEY_W; // KEY_W = 87 (ASCII)
}
```

Y nos falta el último de los componentes de nuestro proyecto, el de colisiones, que tratará de un rectángulo, que define la "bounding box" de la entidad.

Componente de colisiones

```
struct CollisionCMP{
    CollisionCMP();
    Rectangle boundingBox;
};
CollisionCMP::CollisionCMP(){
    boundingBox = {0,0,0,0}; // por defecto no tiene dimension
}
```

Con todos los componentes preparados, tendremos que empezar a pensar en las entidades, las cuales cambiaremos un poco respecto al proyecto 3.3, y podremos diferenciar que entidades contienen un componente u otro de forma muy sencilla, pero aún lejos de una optimización total.

Usaremos el tipo de datos "std::optional<Componente>", que para el que no lo conozca, se asemeja a un puntero que puede o no tener un valor válido. La diferencia es que "std::optional" es un objeto en sí mismo y no necesita ser alojado en la memoria de manera dinámica como lo hacen los punteros. Si recordamos el proyecto anterior, creamos una variable bool para saber si tenemos un componente o no, lo que hace poco manejable al

código y voluminoso. Sin embargo, ahora ese "bool" digamos que está contenido en el tipo de datos "std::optional", el cual dispone de un método para comprobar si tenemos o no objeto inicializado en él.

El código de las entidades quedará de la siguiente forma:

Entidad

```
struct Entity{
    Entity();
    int id;
    std::optional<CollisionCMP> coll;
    std::optional<InputCMP> inp;
    std::optional<RenderCMP> rend;
    std::optional<PhysicCMP> phy;
    private:
    inline static int nextID{1};
};
Entity::Entity(){
    id = nextID++;
}
```

En el constructor simplemente aumentamos en uno la variable estática "nextID", lo que hace que cada vez que creamos una entidad tenga id+1 como ID. Esto es posible a que la variable nextID es inline y static, por lo tanto será la misma para todas las entidades e irá aumentando con cada entidad.

Si volvemos al diagrama anterior, podemos dar por finalizada la parte de creación de componentes y composición de la entidad, por lo tanto quedará por construir los sistemas y el manejador de entidades, empezando por este último en el apartado siguiente.

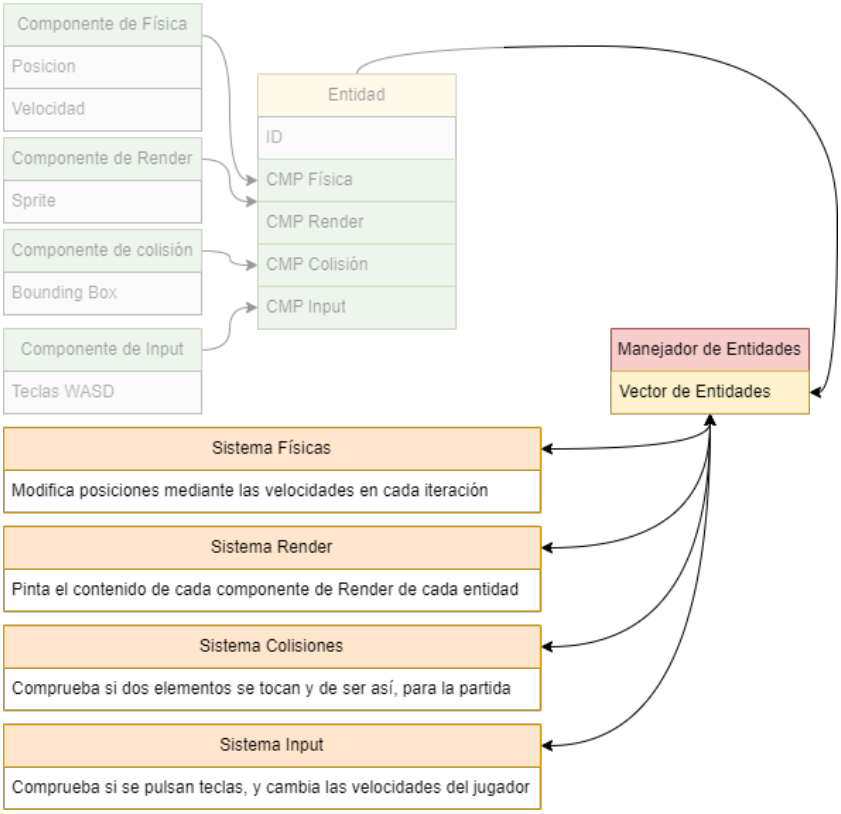


Figura 21. Elementos restantes

5.1.2 Manejador de entidades

Ya tenemos las entidades montadas con nuestros componentes. Es hora de comenzar a montar el manejador de entidades. Próximamente, añadiremos nuevas funcionalidades a este, pero por ahora seguimos con el del proyecto anterior como base.

EntityManager – Manejador de entidades

```
struct EntityManager{

    EntityManager(std::size_t size_for_entities = 10){
        entities.reserve(size_for_entities);
    }
    auto& createEntity(){ return entities.emplace_back();}
    void forall(auto&& function){
        for(auto&e:entities){
            function(e);
        }
    }
    std::vector<Entity>& getEntityVector(){
```

```
        return entities;
    }
    private:
    std::vector<Entity>  entities;
};
```

No podemos decir que hemos terminado el manejador de entidades hasta que completemos las funcionalidades que veremos en los siguientes apartados.

Llegado a este punto, lo siguiente es preparar nuestros sistemas para poder agruparlos en la clase "Game", clase que ejecutará el bucle del juego, y poder decir que tenemos un prototipo al que jugar.

5.1.3 Sistema de físicas

Comencemos con el sistema de físicas, desde mi punto de vista es el más sencillo, ya que su objetivo es actualizar la posición de las entidades, teniendo en cuenta la velocidad de estas.

Método update de PhysicSystem

```
void PhysicSystem::update(EntityManager& EM){
    auto player = EM.getEntityVector()[0];
    EM.forall([&](Entity&e){
        if(e.phy.has_value()){
            // entity pos x += entity velocity x
            e.phy.value().position.first +=
            e.phy.value().velocity.first;
            // entity pos y += entity velocity y
            e.phy.value().position.second +=
            e.phy.value().velocity.second;
            if(e.id == player.id){
                e.phy.value().velocity.first = 0;
                e.phy.value().velocity.second = 0;
            }
        }
        checkPositions(EM,e);
    });
}
```

Si nos fijamos en el código, al llamar al método "forall" del "EntityManager", seguidamente comprobaremos si el componente de físicas tiene valor o no. El método "has_value()", es el método de "std::optional" del cual hablaba anteriormente. Nos devuelve verdadero si el componente se ha definido. Por lo tanto, todas las entidades con físicas, realizarán

este método. Además, si es la entidad jugador, que lo sabemos mediante una comprobación de "id" como se muestra en el código, hacemos cero su velocidad, para que no se esté moviendo hacia ningún lado si no lo decide el sistema de input.

Después de haber hecho esta actualización de posiciones, tenemos que tener en cuenta un detalle importante. Los enemigos, aunque aún no los hayamos implementado, se mueven de derecha a izquierda, por lo tanto llegará un momento en el que salgan por el lado izquierdo de la pantalla y dejemos de verlos. Si no hacemos nada al respecto, cada vez tendremos más y más entidades, las cuales se irán acumulando hasta que se haga un juego muy lento. Con esto quiero decir que tenemos que tener esta situación controlada, y es por eso por lo que crearemos la función "checkPositions()" que hay justo debajo de la actualización de posiciones.

Esta función comprobará si la entidad de cada momento, está más a la izquierda que la pantalla, es decir si la posición 'x' es menor que el límite de la pantalla 0. Si es así esta entidad deberá ser eliminada, para que nuestro sistema no la tenga más en cuenta a la hora de actualizar los datos.

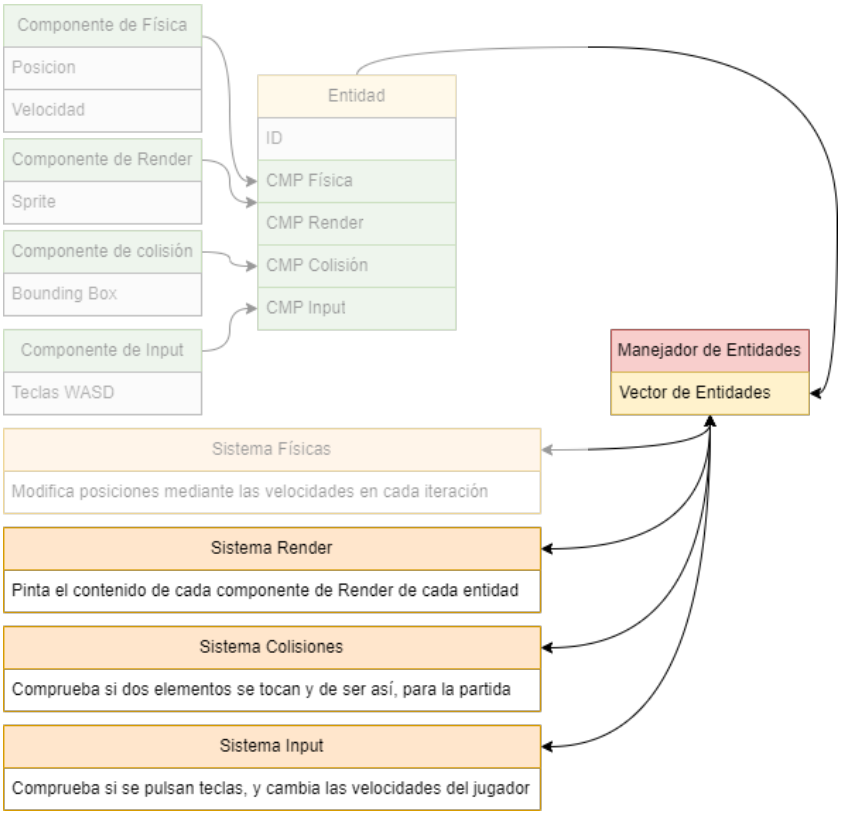


Figura 22. Elementos restantes

Hemos completado el sistema de físicas. Observamos en el diagrama 23 lo que aún queda por hacer para tener un juego. A continuación proseguiremos con el manejador de entidades.

5.1.4 Manejador de entidades: Eliminamos entidades

Llegado a la conclusión del apartado anterior, programaremos un nuevo método en el "EntityManager" el cual eliminará una entidad según la identificación (id) de esta. Lo primero que tiene que saber el método que vamos a crear es la posición de la entidad que vamos a borrar. Esto lo podemos conseguir mediante la función "std::remove_if()" la cual recibe tres parámetros. En primer y segundo lugar, los punteros "begin()" y "end()" de un vector, y como último parámetro la comprobación, para la cual nosotros haremos un lambda que devuelva "true" si se cumple que la id de cada entidad es igual a la pasada por parámetro. De esta forma obtendremos en la variable "it" el iterador exacto de la entidad que queremos borrar, solo falta eliminarla con el método "erase" del vector.

EntityManager::removeEntity(int id)

```
void removeEntity(int id) {
    auto it = std::remove_if(entities.begin(), entities.end(),
        [id](const Entity &e) {
            return e.id == id;
        });
    entities.erase(it, entities.end());
}
```

También podemos crear una función que pueda servirnos más tarde que sirva para eliminar todas las entidades del juego:

EntityManager::removeAllEntities()

```
void removeAllEntities(){
    for(std::size_t i = 0; i < entities.size();i++){
        UnloadTexture(entities[i].rend.value().sprite);
        removeEntity(entities[i].id);
    }entities.clear();
}
```

Ahora si podemos concluir con nuestro sistema de físicas que habíamos dejado inacabado, para terminar programando el método "checkPositions()". En lugar de 0 en la comprobación, que es el límite izquierdo de la pantalla, usaremos -20, para que no desaparezca al tocar el borde, si no que suceda cuando ya lo haya traspasado, para dar un efecto visual menos brusco.

PhysicSystem::checkPositions(EntityManager& EM, Entity& e)

```
void PhysicSystem::checkPositions(EntityManager& EM, Entity& e){
```

```
    if (e.phy.value().position.first < -20){
        EM.removeEntity(e.id);
    }
}
```

Ahora si que hemos completado el manejador de entidades para este proyecto, por lo tanto volvamos a observar el diagrama y observemos que el siguiente paso es construir los otros tres sistemas restantes.

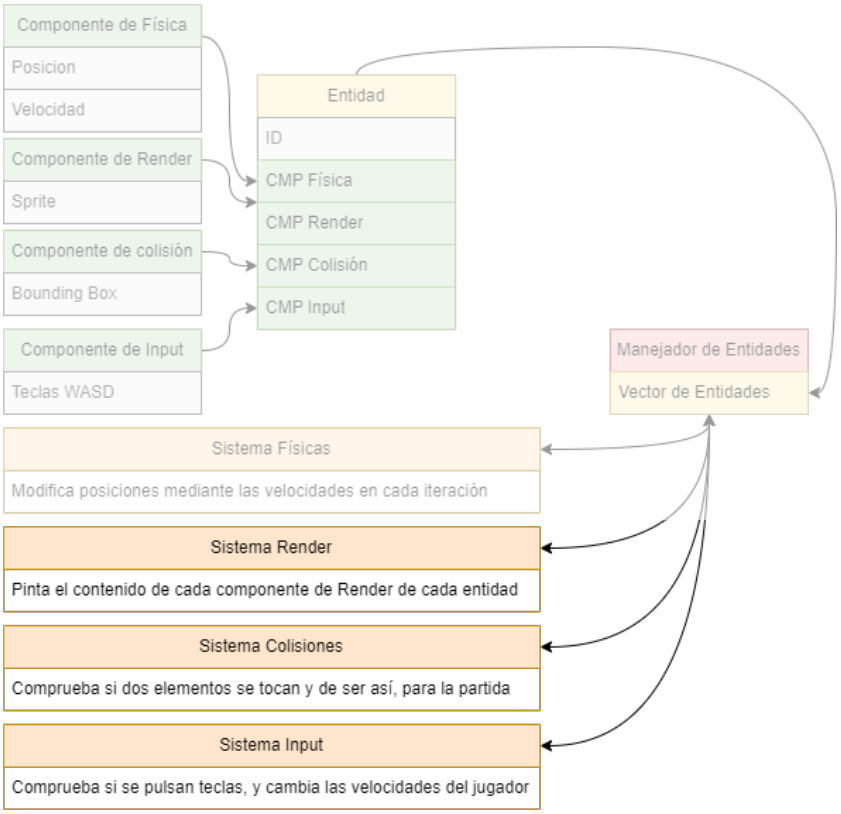


Figura 23. Componentes creados

5.1.5 Sistema de renderizado

Es la hora de dibujar elementos en la pantalla, y ahí es donde entra principalmente el sistema de renderizado. En este sistema, el objetivo principal es dibujar cada una de las entidades que llegan al sistema, teniendo en cuenta la posición en la que se encuentran. Además de dibujar las entidades, podemos dibujar texto en la pantalla, lo que es muy útil para representar puntuaciones, mensajes, etc.

Sistema de render: update

```
void RenderSystem::update(EntityManager& EM, float score){

    //draw entities
    EM.forall([&](Entity&e){
        if(e.rend.has_value()){
            float posx = e.phy.value().position.first;
            float posy = e.phy.value().position.second;
            DrawTextureRec(e.rend.value().sprite,
                           e.rend.value().box,
                           (Vector2){posx,posy},
                           WHITE);
        }
    });

    //draw text
    printScore(score);
}
```

En el código anterior hemos programado el sistema de render, el cual hace justo lo que hemos planteado. Recoge las posiciones x e y de la entidad, y seguidamente dibuja el sprite, en la posición indicada. La función "DrawTextureRec(sprite, dimensión, posiciones, filtro)" se encarga de dibujarlo. No tenemos que pasar por alto el detalle de que se comprueba antes de hacer todo esto si la entidad tiene componente de render definido en el "std::optional".

Obviaré el código de las funciones de dibujado de texto, ya que está explicado en la introducción a RayLib del capítulo 4. Para más información sobre esto, siempre se puede consultar el código que estará disponible (Cantó-Berná (2023c)).

Pero si que comentaré el objetivo de la función "paintScore()", dibujar en la pantalla los puntos actuales.

5.1.6 Sistema de input

Seguimos con el sistema de input, que tiene un objetivo claro y conciso, cambiar la velocidad de las entidades con componente de input según la tecla pulsada.

Sistema de input: update

```
void InputSystem::update(EntityManager& EM){
    EM.forall([&](Entity&e){
        if(e.inp.has_value()){
            if (IsKeyDown(e.inp.value().KeyW) &&
            e.phy.value().position.second > 10){
                e.phy.value().velocity.second =
                -200.0f * GetFrameTime();
            }
            if (IsKeyDown(e.inp.value().KeyA) &&
            e.phy.value().position.first > 10){
                e.phy.value().velocity.first =
                -400.0f * GetFrameTime();
            }
            if (IsKeyDown(e.inp.value().KeyS) &&
            e.phy.value().position.second < 360){
                e.phy.value().velocity.second =
                200.0f * GetFrameTime();
            }
            if (IsKeyDown(e.inp.value().KeyD) &&
            e.phy.value().position.first < 620){
                e.phy.value().velocity.first =
                200.0f * GetFrameTime();
            }
        }
    });
}
```

Como podemos apreciar, podemos obtener las teclas del componente y mediante la función de RayLib "IsKeyDown()" detectar si esa es la tecla pulsada, y en ese caso, cambiamos la velocidad. Este sistema es así de sencillo, ya podemos ponernos con el siguiente sistema.

5.1.7 Teoría: Colisiones

En C++, hay varios tipos de colisiones que se pueden implementar en un programa de simulación o juego, entre ellos se encuentran:

- Colisión AABB (Axis-Aligned Bounding Box): Este tipo de colisión se calcula utilizando cajas delimitadoras alineadas con los ejes (AABB) que rodean a cada objeto en la simulación. La colisión se produce cuando las cajas se superponen.

Para calcular la colisión de AABB, se utiliza la geometría de la caja para determinar si dos objetos se superponen en cualquier dirección.

- Colisión de Esferas: Este tipo de colisión se calcula utilizando la geometría de la esfera que rodea a cada objeto en la simulación. La colisión se produce cuando las esferas se superponen. Para calcular la colisión de esferas, se utiliza la distancia entre los centros de las esferas y su radio para determinar si dos objetos se superponen.
- Colisión de Polígonos: Este tipo de colisión se utiliza para objetos con formas irregulares que se definen por polígonos. La colisión se produce cuando dos polígonos se superponen. Para calcular la colisión de polígonos, se utiliza la geometría de los polígonos para determinar si dos objetos se superponen.
- Colisión de Rayos: Este tipo de colisión se calcula mediante el trazado de rayos que salen de un objeto e intersectan con otros objetos en la simulación. La colisión se produce cuando el rayo intersecta con otro objeto. Para calcular la colisión de rayos, se utiliza la geometría del objeto y el algoritmo de intersección de rayos para determinar si un objeto se encuentra en el camino del rayo.
- Colisión de Mallas: Este tipo de colisión se utiliza para objetos con formas más complejas que se definen por mallas de triángulos. La colisión se produce cuando dos mallas se superponen. Para calcular la colisión de mallas, se utiliza la geometría de los triángulos y se comprueba si alguno de los vértices de un triángulo está dentro de otro triángulo.

El cálculo de las colisiones en C++ puede ser complejo y requiere conocimientos avanzados de geometría y programación. Los algoritmos utilizados para cada tipo de colisión pueden variar dependiendo de la implementación y las necesidades del programa.

En nuestro caso, usaremos colisiones AABB que aunque nos las proporciona RayLib, voy a mostrar como podemos hacerlo de una forma manual.

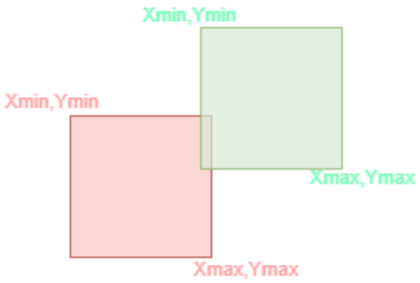


Figura 24. Colisión del tipo AABB

En primer lugar visualizaremos con la figura 24, cómo funciona la colisión y de qué parámetros disponemos. Observamos que tenemos punto verde 1 que será (Xmin, Ymin) y el punto verde 2 que será (Xmax, Ymax), y otros dos puntos de la figura roja, similares y en la posición exacta de la caja respecto a la otra figura.

En primer lugar, definiremos estas cajas de colisión como un struct con 4 puntos.

Objeto de colisión

```
struct CajaDeColision {
```

```
    float x_min, x_max, y_min, y_max;
};
```

De esta forma tenemos en una estructura las coordenadas de los dos puntos, con las que trabajaremos para construir una función que compare dos de estas estructuras. Medimos las posiciones de estos puntos para comprobar si hay colisión o no entre esas dos cajas.

Esa función la podemos llamar "checkCollision()" la cual además recibirá dos parámetros del tipo de la estructura anterior.

Comprobamos si existe colisión

```
bool check_collision(const CajaDeColision& roja,
                     const CajaDeColision& verde) {
    // Comprobamos si hay superposicion en el eje X
    bool x_collision = (roja.x_min <= verde.x_max &&
                        roja.x_max >= verde.x_min);

    // Comprobamos si hay superposicion en el eje Y
    bool y_collision = (roja.y_min <= verde.y_max &&
                        roja.y_max >= verde.y_min);

    // Si hay superposicion en ambos ejes, hay colision
    return (x_collision && y_collision);
}
```

Comprobamos el eje X. Si se cumple que la X mínima de la caja roja es menor o igual que la X máxima de la caja verde, y por otro lado que la X máxima de la caja roja, es mayor o igual que el mínimo punto en X de la verde, significa que en el eje X habrá colisión.

Lo mismo ocurrirá para el eje Y, y la composición de ambas comprobaciones nos definirá si hay o no colisión.

5.1.8 Sistema de colisiones

El sistema de colisiones tiene como objetivo detectar choques entre entidades. Para ello, lo primero que hay que tener en cuenta es la bounding box de cada entidad. Esta necesita ser actualizada, si no veremos como se mueve por la pantalla el objeto, pero no se moverá su caja de colisiones, por lo tanto no habrá colisión. Una vez actualizada, tendremos que comprobar para todas las entidades, menos para la entidad jugador, si hay colisión. Dicho de otra forma, no hay que comprobar si hay colisión entre el jugador y el jugador mismo, por que siempre habría colisión. Entonces, nos guardamos el jugador y vamos comprobando con todas las entidades que tengan componente de colisión si hemos chocado.

Sistema de colisiones: update

```
void CollisionSystem::update(EntityManager& EM, Map& map,
States& state){
    auto& player = EM.getEntityVector()[0];
    EM.forall([&](Entity& e){
        if (e.coll.has_value()){
            e.coll.value().boundingBox = {
            e.phy.value().position.first,
            e.phy.value().position.second,
            static_cast<float>(e.rend.value().sprite.width),
            static_cast<float>(e.rend.value().sprite.height)
            };
            if(e.id != player.id){
                bool collision = CheckCollisionRecs(
                player.coll.value().boundingBox,
                e.coll.value().boundingBox);
                if(collision){
                    collisionWithEnemy(EM, map, state);
                }
            }
        }
    });
}
```

La colisión se calcula mediante la librería Raylib de nuevo, que tiene soporte de colisiones, mediante la función "CheckCollisionRecs()". En caso de haber colisionado con un enemigo, ya que es la única colisión que tenemos, habrá que parar la partida. Es en este punto en el que tendremos que pensar en el diseño de nuestro juego. Haremos un pequeño sistema de estados básico para salir al paso. Es por eso que este método recibe por parámetro el estado actual para modificarlo en caso de colisión.

Si colisionamos con un enemigo borraremos todas las entidades del juego, ya que se habrá acabado la partida, cambiaremos el estado a un estado de fin de juego, y cambiaremos a la pantalla final como se muestra en el ejemplo del código siguiente (no te preocupes si no sabes cómo preparar los estados o el 'map' que uso, se explican en siguientes apartados).

Sistema de colisiones: Colisión con enemigo

```
void CollisionSystem::collisionWithEnemy(EntityManager& EM, Map& map,
                                         States& state){
    EM.removeAllEntities();
    state = States::end;
    map.setMapSpeed( 0.0f );
    map.setMapBackground( "img/end.png" );
    map.setMapPositions((Vector2){0,0});

}
```

5.1.9 Estados y Mapa

Para entender a la perfección todo lo que tenemos hasta ahora, hablaremos de los estados que he creado para el paso entre pantallas, y que la colisión con el enemigo no cierre el juego de golpe, lo que sería una forma brusca de terminar esto.

Representaremos los estados como un enum con tres diferentes valores, menu, play, end, 0, 1 y 2 respectivamente.

Estados del juego

```
enum States{
    menu,
    play,
    end
};
```

Lo siguiente es el mapa, que es algo más complejo que los estados, y están muy basados en la librería de apollo RayLib, ya que es un apartado bastante gráfico.

La clase mapa tiene diferentes variables, como serán la posición X e Y, la velocidad y la textura que tendrá nuestro fondo. También dispone de métodos que ejecutan las siguientes acciones:

- Un constructor que inicializa todo a valores por defecto.
- Una función que he llamado "drawAndMoveMap()", la cual se encarga de ir cambiando la posición de la textura, según la velocidad y de dibujar dos veces la textura, la primera en su posición actual, y la siguiente seguida a la actual. El resultado de la ejecución de esta función hace parecer al jugador que la pantalla está avanzando, pero en realidad solo se mueve la textura.

Map::drawAndMoveMap()

```
void Map::drawAndMoveMap(){
    map_x -= speed;
    if (map_x <= -background.width) map_x = 0;
    DrawTextureEx(background, Vector2{map_x,map_y},
                  0.0f, 1.0f, WHITE);
    DrawTextureEx(background,
                  Vector2{map_x + background.width, map_y},
                  0.0f, 1.0f, WHITE);
}
```

- Una función que hemos llamado "textureCleaner()", que elimina la textura de la memoria, liberando espacio.

Map::textureCleaner(Texture2D& tex)

```
void Map::textureCleaner(Texture2D& tex){
    UnloadTexture(tex); //Metodo de RayLib
}
```

- Los métodos get, para obtener las variables de mapa en los momentos necesarios.

Métodos get del mapa

```
//devuelve una copia de la pos actual del mapa
Vector2 Map::getMapPositions(){
    return (Vector2){map_x, map_y};
}
//devuelve una copia de la velocidad actual del mapa
float Map::getScreenSpeed(){
    return speed;
}
//devuelve la textura del mapa por referencia
Texture2D& Map::getMapBackground(){
    return background;
}
```

- Métodos set, los cuales permiten modificar las variables de mapa desde el exterior de la clase.

Métodos set del mapa

```
//coloca nuevas posiciones al mapa
```

```
void Map::setMapPositions(Vector2 positions){
    map_x = positions.x;
    map_y = positions.y;
}
//coloca nueva velocidad al mapa
void Map::setMapSpeed(float sp){
    speed = sp;
}
//cambia la textura del mapa, eliminando la anterior
void Map::setMapBackground(const char* route){
    textureCleaner(background);
    background = LoadTexture(route);
}
```

Ahora, si volvemos al código de la colisión con los enemigos, podemos entender todos los conceptos que comenté anteriormente.

Tenemos todo programado, pero ahora falta la última parte, la clase "Game" del juego, es decir, donde creamos nuestros objetos y entidades y unimos todas las funcionalidades que hemos programado, pero antes de ir a esta clase final, quiero mostrar los diferentes estados que tendrá el juego y algún que otro detalle extra.

Figura 25. Primera pantalla del juego, en el estado 'menu'

En primer lugar tendremos un estado "menu" que será nuestra pantalla de inicio, ya que no queremos ejecutar el juego y que directamente tengamos que esquivar misiles.

En este momento del juego, tendremos la opción de aumentar la dificultad, que se verá reflejado en la velocidad a la que aparecen los misiles. A más dificultad más rápido irán apareciendo. He diseñado un prototipo de esta primera pantalla, es el de la figura 25

En esta pantalla además hay que configurar la dificultad para el juego, y como se aprecia en mi dibujo, será con las teclas 1, 2 y 3. Con el "Intro" comenzará la partida y el "Escape" cierra el juego. Los últimos dos detalles son reflejar el récord encima del titulo y la dificultad abajo de todas las opciones.

En la siguiente pantalla entra la partida, el contador se inicializa con valor cero y cada iteración del bucle del juego aumentará esta puntuación. Los misiles salen en posición 'Y' aleatoria, de derecha a izquierda. El "OVNI" es decir el jugador, tendrá que esquivar los misiles, las teclas que permiten esto son "W", "A", "S" y "D". En el caso de morir, tendremos que ir a otra pantalla, esta vez una final. También he realizado un boceto de pantalla final 26

Figura 26. Pantalla final del juego

En esta última, la tecla "space" nos hace regresar al estado de menú y el "escape" cerrará el juego. Estos son los detalles que hay que conocer antes de desarrollar la clase "Game".

5.1.10 Teoría: Orden de ejecución de sistemas

El orden en el que se ejecuten los sistemas en un Entity Component System puede tener un impacto significativo en el rendimiento y la experiencia de juego. Estos son los sistemas de los que disponemos en el juego que estamos programando:

- Input: Este sistema debe ejecutarse antes que los demás, ya que se encarga de procesar la entrada del usuario y actualizar los componentes correspondientes en consecuencia.
- Colisiones: El sistema de colisiones debe ejecutarse después del sistema de entrada, ya que necesita verificar las colisiones basadas en los componentes que se han actualizado en la entrada.
- Físicas: El sistema de físicas debe ejecutarse después del sistema de colisiones, ya que necesita actualizar las posiciones de los objetos en función de las colisiones y aplicar fuerzas en el caso de que existan.
- Render: El sistema de renderizado debe ejecutarse después del sistema de físicas, ya que necesita actualizar la visualización de los objetos en función de sus nuevas posiciones.

Sin embargo, vale la pena mencionar que el orden de ejecución puede variar en función de la implementación específica y las necesidades del juego en cuestión.

5.1.11 Desarrollo de la clase Game

Empezaré hablando sobre todos los objetos que hay definidos como variables privadas en esta clase. Crearemos un manejador de entidades, junto a los cuatro sistemas que hemos programado anteriormente. También crearemos el objeto del mapa y el objeto "States" para el estado actual del juego. El resto de variables las conoceremos más adelante, pero hay comentarios sobre ellas en el código que indican para qué las hemos utilizado.

struct Game

```
struct Game{
    private:
    //Manejador de entidades
    EntityManager EM;
    //Sistemas
    CollisionSystem coll_sys;
    PhysicSystem phy_sys;
    RenderSystem rend_sys;
    InputSystem inp_sys;
    //Mapa
    Map map;
    //Estado actual del juego
    States state;
    //Variables para controlar el spawn de enemigos
    int spawnRatio;
    int spawn;
    //Puntuaciones
    float score;
```

```
    float record;
    //Dificultad
    int difficulty;
};
```

El primer paso en la clase "Game" de este y otros proyectos anteriores es definir todos los objetos que hemos programado anteriormente ya que es aquí el ámbito en el que se crean y utilizan. Todas estas variables las inicializamos en el constructor.

Game::Game()

```
    Game(){
        state = States::menu;
        EM =  EntityManager();
        coll_sys = CollisionSystem();
        phy_sys = PhysicSystem();
        rend_sys = RenderSystem();
        inp_sys = InputSystem();
        score = 0.0f;
        record=0.0f;
        difficulty = 1;
        spawnRatio=SPAWN_NUMBER - DIFFICULTY_INCREMENT * difficulty;
        spawn = spawnRatio;
    }
```

SPAWN_NUMBER y DIFFICULTY_INCREMENT son dos constantes que definimos fuera de la clase Game. Modificando estas obtenemos diferentes sistuaciones respecto al spawn de enemigos, que se pueden ver reflejadas en el constructor de "Game".

Constantes numéricas

```
#define SPAWN_NUMBER 60
#define DIFFICULTY_INCREMENT 10
```

El siguiente paso es plantear la otra función pública de la clase. Este es el método "run()", el cual ejecuta el bucle del juego. A continuación se muestra el código de este método, en el que iremos comentando y definiendo los diferentes métodos auxiliares que necesita.

Game::run()

```
    void run(){
        map.setMapBackground("img/menu.png");
```

```
        SetTargetFPS(60);
        while (!WindowShouldClose()){
            BeginDrawing();
            ClearBackground(RAYWHITE);
            map.drawAndMoveMap();
            switch (state) {
            case States::menu:
                chooseDifficulty();
                rend_sys.printDifficultySelected(difficulty);
                rend_sys.printRecord(record);
                goToPlay();
                break;
            case States::play:
                inp_sys.update(EM);
                coll_sys.update(EM, map, state);
                phy_sys.update(EM);
                rend_sys.update(EM, score);

                enemySpawn();
                score+= 0.1f;
                break;
            case States::end:
                rend_sys.printScore(score);
                goToMenu();
                break;
            }
           EndDrawing();
        }map.textureCleaner(map.getMapBackground());
    }
```

En primer lugar iniciaremos el mapa, colocando la primera pantalla de fondo, y establecemos los frames a 60 con la función "SetTargetFPS(valor)" de RayLib.

En RayLib el bucle lo podemos controlar como hemos aprendido en el apartado de introducción a RayLib. Mientras esto se cumpla el bucle sigue. Entre los métodos "BeginDrawing()" y "EndDrawing()" vamos a dibujar todo el contenido. "ClearBackground(RAYWHITE)" limpia la pantalla en cada iteración y finalmente antes de decidir en que estado estamos, dibujamos el mapa. Esto lo podemos hacer mediante la función de la clase "Map" que hemos compuesto anteriormente "drawAndMoveMap()". Con esto el mapa se moverá si tiene velocidad en cada iteración del bucle.

Lo siguiente será decidir en que estado estamos. Para eso programamos la estructura "switch" que nos permite ejecutar un caso para cada estado. Por lo tanto colocamos la variable state en el switch y tendremos 3 casos:

- States::menu: en este caso ejecutaremos la función "chooseDifficulty()" que simplemente cambiará la variable "difficulty" y el spawnRatio, según la tecla pulsada.

Game::chooseDificulty()

```
    void chooseDifficulty(){
        if (IsKeyDown(KEY_ONE)){
            difficulty = 1;
            spawnRatio = SPAWN_NUMBER -
                         DIFFICULTY_INCREMENT* difficulty;
        }
        if (IsKeyDown(KEY_TWO)){
            difficulty = 2;
            spawnRatio = SPAWN_NUMBER-
                         DIFFICULTY_INCREMENT* difficulty;
        }
        if (IsKeyDown(KEY_THREE)){
            difficulty = 3;
            spawnRatio = SPAWN_NUMBER-
                         DIFFICULTY_INCREMENT* difficulty;
        }

    }
```

Seguidamente pintamos texto como hemos visto anteriormente, con los métodos auxiliares de dibujado de texto de RayLib. Con esto dibujamos la variable "difficulty" en su lugar en la pantalla inicial, y el récord en la parte superior del título. Por último, en este estado tendremos que controlar cuándo pasar al siguiente estado. Para ello cuento con una función que he llamado "goToPlay()" la cual si pulsamos "Enter", cambiamos el estado a "play", además del fondo de pantalla, y la velocidad del mapa. Por último creamos el personaje en la pantalla con la función "createPlayer()".

Game::goToPlay() y Game::createPlayer()

```
void goToPlay(){
    if (IsKeyDown(KEY_ENTER)){
        score = 0.0f;
        createPlayer();
        map.setMapSpeed(4.0f);
        map.setMapBackground("img/background1.png");
        state = States::play;
    }
}
```

```
void createPlayer(){
    auto& player = EM.createEntity();
    player.phy = PhysicCMP();
    player.rend= RenderCMP();
    player.inp= InputCMP();
    player.coll=CollisionCMP();
}
```

Creamos la entidad, y le inicializamos los componentes de los que dispondrá el jugador.

- States::play : En este estado, ya creado el jugador, lo siguiente es llamar al método "update" de todos los sistemas que hemos creado. Como hemos observado en el apartado teórico acerca del orden de los sistemas, 5.1.10, el orden que seguiremos para actualizarlos será el siguiente: input, colisión, físicas y renderizado finalmente. Otro detalle en este estado trata de incrementar la variable "score", dando parecer que conforme avanzamos en la pantalla obtenemos distancia, o puntos de recompensa. Por último, generaremos los enemigos con la función "enemySpawn()".

Game::enemySpawn() y Game::createEnemy()

```
    }
void enemySpawn(){
    if (spawn == 0){
        createEnemy();
        spawn = spawnRatio;
    }spawn--;
}
void createEnemy(){
    auto& enemy= EM.createEntity();
    enemy.phy = PhysicCMP();
    enemy.rend= RenderCMP();
    enemy.coll= CollisionCMP();
    enemy.phy.value().position.first = 700;
    enemy.phy.value().position.second = 20 + (rand()%370);
    enemy.phy.value().velocity.first = (-1) * (8 + rand() % 15);
    enemy.rend.value().sprite = LoadTexture("img/enemy.png");
    enemy.rend.value().box = {0,0,
        static_cast<float>(enemy.rend.value().sprite.width),
        static_cast<float>(enemy.rend.value().sprite.height)};
}
```

El generador de enemigos funciona creando un enemigo cada vez que "spawn" llega a 0. Por otro lado, en la función "createEnemy()" creamos la entidad, inicializamos los componentes que tienen los enemigos que serán todos menos el de input. La posición de los enemigos, comenzará siendo 700 para que salgan desde la derecha

del todo, la 'Y' será un valor aleatorio desde 20 a 370, lo que hará que puedan salir por cualquier punto de la pantalla. La velocidad también será aleatoria entre diferentes valores para dar dificultad al juego y por último, definiremos las texturas y tamaños de los sprites de los enemigos.

- States::end : En esta pantalla dibujaremos la puntuación final de la partida, en cada iteración, y todo lo demás habrá cambiado en el evento de colisión. Finalmente, tenemos el método "goToMenu()" el cual guarda en la variable "record" el contenido de la variable "score" si es mayor a lo que hay en ese instante en la primera. Si la tecla "space" es pulsada en esta pantalla, reseteamos la puntuación, cambiamos al primer estado, cambiamos el fondo a la pantalla inicial y reseteamos el "spawnRatio" para futuras partidas.

Game::goToMenu()

```
void goToMenu(){
    if(record<score){
        record = score;
    }
    if (IsKeyDown(KEY_SPACE)){
        score = 0.0f;
        state = States::menu;
        map.setMapBackground("img/menu.png");
        spawnRatio = SPAWN_NUMBER;
    }

}
```

Con todo esto tenemos el proyecto terminado. Pero antes de pasar al siguiente proyecto, así queda el método "main", el cuál ejecutará nuestra aplicación.

main()

```
#include "game.hpp"
int main(){
    Game game;
    InitWindow(700, 400, "OVNI");
    // Bucle del juego
    game.run();
    // Cerrar la ventana y liberar los recursos
    CloseWindow();
    return 0;
}
```

5.1.12 Ejercicio: Proponemos una mejora visual

Para que el jugador tenga una mejor experiencia, vamos a proponer una mejora. La experiencia de juego es uno de los factores más importantes de este, ya que de esto dependerá que el jugador quiera seguir jugando o abandone el juego.

Una de las cosas a mejorar es ver como el jugador muere, y es lo que proponemos cambiar en este apartado. Actualmente al colisionar con un misil, automáticamente el juego cambia de estado a una pantalla final y no se aprecia el choque del todo.

Es por eso que nuestro objetivo a continuación es hacer que cuando muera el jugador se siga viendo esa colisión en la pantalla con el juego como si estuviese pausado, y salga el mensaje final en esa pantalla con la colisión de fondo. De esta forma el jugador siempre podrá ver cómo ha perdido.

Implementación. Comenzaremos observando que ocurre si el misil impacta y se produce la colisión. Para ello vamos a ver cómo lo teníamos implementado.

Colisión con enemigo actual

```
void CollisionSystem::collisionWithEnemy(EntityManager& EM, Map& map,
                                         States& state){
    EM.removeAllEntities();
    state = States::end;
    map.setMapSpeed( 0.0f );
    map.setMapBackground( "img/end.png" );
    map.setMapPositions((Vector2){0,0});
}
```

Podemos observar que eliminamos todas las entidades, cambiamos de estado, colocamos la velocidad del mapa a 0, cambiamos su fondo y restablecemos su posición a 0 también.

Ahora en esta función no se eliminarán las entidades, ni modificaremos el fondo del juego, ni tampoco la posición del mapa. Si que modificaremos la velocidad del mapa a 0 para parar el scroll lateral y el estado seguirá cambiando a un estado final. Además de estos cambios, vamos a modificar a todas las entidades para que sus velocidades sean 0, así dará sensación de que al chocar todo para y obtenemos esta pantalla como en pausa mostrando el final del juego.

Colisión con enemigo modificada

```
void CollisionSystem::collisionWithEnemy(EntityManager& EM, Map& map,
                                         States& state){
    state = States::end;
```

```
    map.setMapSpeed( 0.0f );
    for(std::size_t i=0; i<EM.getEntityVector().size();i++){
        EM.getEntityVector()[i].phy.value().velocity.first = 0;
    }
}
```

Además de este cambio, añadiremos el mensaje final a la pantalla. Para ello en game tendremos que llamar a una función que haga esto. Esa función la he creado en el sistema de renderizado. Y simplemente dibuja el mensaje en el centro de la pantalla.

Pintamos mensaje final

```
void RenderSystem::printExit(){
    std::ostringstream stream;
    stream << std::fixed << std::setprecision(2) <<
        "YOU LOSE - PRESS SPACE TO RETURN TO MENU" ;
    std::string str = stream.str();
    const char* pointsString = str.c_str();
    DrawText(pointsString, 85,200,fontSize, WHITE);
}
```

Por último tenemos que modificar el estado "end" en el bucle del juego en la clase "Game". Añadiremos la función anterior y actualizaremos el sistema de renderizado, ya que ahora si que nos interesa que se sigan dibujando las entidades aunque el juego haya terminado.

Estado end en clase Game

```
    ...

    case States::end:
        rend_sys.printScore(score);
        rend_sys.update(EM,score);
        rend_sys.printExit();
        goToMenu();
        break;
    }

    ...
```

Antes de terminar, tendremos que eliminar las entidades en el caso de que el jugador quiera volver a jugar, para ello sin modificar lo que actualmente hay en la función "goToMenu()", añadiremos "EM.removeAllEntities()".

5.1.13 Resumen de lo aprendido

Hemos completado el proyecto. Ahora toca recapitular y fijarnos en lo que hemos aprendido de este juego durante el desarrollo.

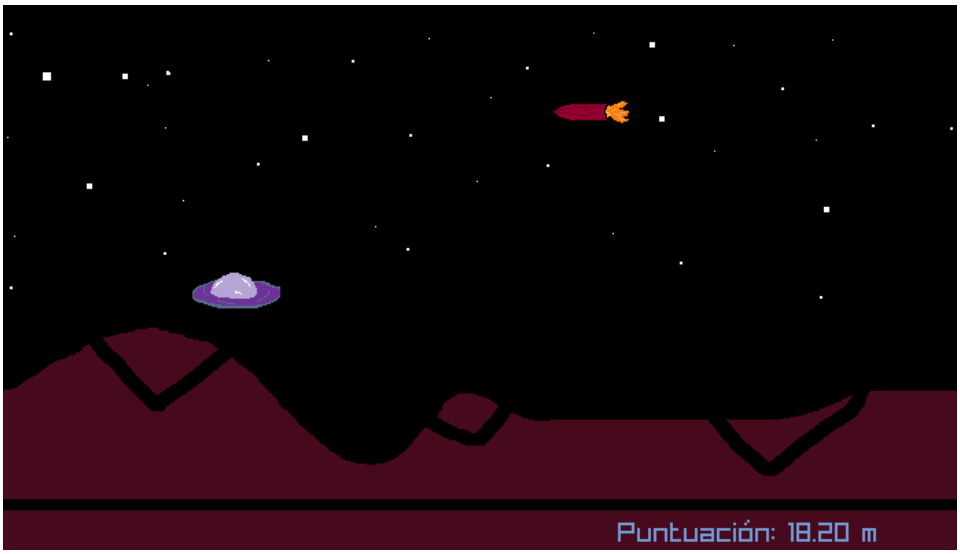
En primer lugar, cambiamos la forma de recoger los componentes en las entidades, lo que fue el primer y principal objetivo de este proyecto. Usamos "std::optional", para obtener una visión aún más clara de la composición de las entidades, basándonos en cómo las construimos en el proyecto anterior, el del bombero.

Además construimos sistemas novedosos hasta el momento, como el de input y colisiones, que aunque hemos usado el input anteriormente, en este caso usamos más opciones de entrada. Por otro lado explicamos los tipos de colisiones que existen, elegimos las del tipo AABB e implementamos un pequeño ejemplo de colisiones.

Respecto al manejador de entidades, hemos introducido nuevos métodos para eliminar entidades. También creamos y destruimos entidades constantemente durante el flujo del juego.

El código se puede encontrar en la referencia Cantó-Berná (2023d), y a continuación muestro algunas imágenes del resultado final.


DISTANCE RECORD> 45.80 m
SAVE THE OVNI
NUEVA PARTIDA
ESC
SALIR DEL JUEGO
1 2 3
ELEGIR DIFICULTAD
ACTUAL > 3


Puntuación: 18.20 m


YOU LOSE
SPACE
VOLVER AL MENÚ
ESC
SALIR DEL JUEGO

6 Estructuración y optimización de los datos

La optimización de la memoria es un tema importante a la hora de crear nuestros juegos, ya que cuanto más optimizada esté, más fluidos y con mejor rendimiento funcionarán.

En este apartado vamos a llevar a cabo uno de los pasos más importantes para la optimización de nuestro motor ECS. Cambiaremos la forma de almacenar los componentes y de utilizarlos. Venimos manteniendo todos los componentes en el interior de las entidades, y a ahora cambiaremos la forma de guardarlos usando un gestor de componentes que crearemos más adelante. Ahora las entidades contendrán la llave que tendremos que usar para obtener los componentes que la entidad tenga. Esto implica menos consumo de memoria, por lo tanto habremos optimizado el motor.

Utilizaremos los Slotmaps, estructura de datos que se explica en el apartado 6.2.1, que permiten un acceso rápido y eficiente a los componentes de las entidades, ya que se puede acceder a estos directamente mediante una clave o índice. Además, los Slotmaps también pueden mejorar el rendimiento de la búsqueda de componentes, ya que se almacenan en un "array" contiguo en memoria, reduciendo la cantidad de accesos y pérdidas.

Actualmente, como ya hemos mencionado, tenemos toda la memoria de los componentes metida en todas las entidades, esto en un proyecto pequeño no tendrá gran impacto, pero en proyectos a gran escala o más complejos que los que mostramos en la guía, será un problema de optimización enorme.

El proyecto que crearemos para desarrollar un Slotmap y ponerlo en funcionamiento en nuestro ECS será "Final Room". En la tabla 4 queda plasmado antes de comenzar.

	Título	Descripción corta
Proyecto 5	Final Room	Utilizamos Slotmaps para optimizar el juego y la memoria.

Tabla 4. Proyecto que abarcaremos en este apartado

Las novedades que nos encontraremos en este proyecto serán:

- Entidades sin componentes, si no con llaves a esos componentes y máscaras para tenerlos controlados.
- Nuevo manejador de componentes para crearlos, añadirlos y eliminarlos.
- Slotmaps para almacenamiento de datos.
- Flujo del juego mejorado, gestion de las entidades en ejecución, y nuevo gestor del juego "Game Manager".
- Etiquetas para diferenciar entre tipos de entidades. (Objetos, enemigos, muros, etc.)

Como se puede observar, el avance de este proyecto es grande pero avanzaremos dando pequeños pasos en cada uno de los objetivos, no te frustres si te abrumas con el nuevo

contenido, repito que iremos poco a poco y no dejaremos atrás ningún detalle. También podemos tomárnoslo como un sprint final para conseguir aquello que queremos, un primer motor optimizado completo y haber completado los conocimientos de esta guía.

Todos los sprites utilizados para este último proyecto, han sido creados para este juego por Alicia (2023).

6.1 Quinto proyecto: The Final Room

En este último proyecto, nos centraremos en encontrar una forma para que en nuestro motor, las entidades no tengan en su interior todos los componentes, ya que de la forma que lo venimos haciendo, cada entidad creada, aunque solo necesite un componente, tendrá en su memoria una instancia de cada uno.

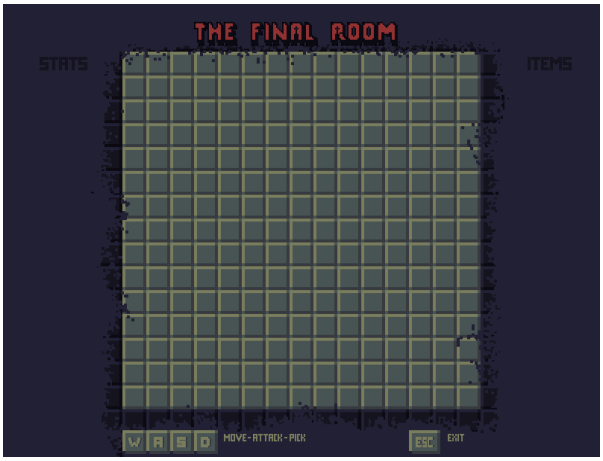


Figura 27. Tablero del juego.

Para ello necesitaremos buscar nuevas estructuras para almacenar estos componentes, que podrían ser vectores o arrays, pero en este caso usaremos los Slotmaps comentados en el apartado anterior. Gracias a estas matrices dispersas, conseguiremos optimizar las operaciones de búsqueda de componentes en nuestra estructura de datos, así como la creación y destrucción.

Antes de comenzar con el desarrollo del quinto y último proyecto de la guía, tenemos que decidir que proyecto haremos y que contendrá nuestro juego.

Para hacernos una idea, el juego contendrá un menú inicial que al acceder al juego cambiará a una pantalla con una especie de tablero, de 15x15 casillas 27, donde el personaje, el cual controlaremos nosotros, podrá moverse, atacar, recoger objetos e incluso romper obstáculos. El objetivo del juego es avanzar tantas rondas como puedas, recogiendo objetos que mejoren tus estadísticas.

Habrá enemigos que harán una patrulla hasta que detecten al personaje y decidan perseguirle. El jugador, en cada ronda, deberá llegar hasta la llave, para así poder escapar de ese mapa por la puerta.

En la interfaz, dispondremos de las estadísticas del jugador y de los objetos recogidos equipados en cada momento.

Esta es la idea inicial del juego, e iremos desarrollando todo lo que necesitemos para

reproducirla.

El siguiente diagrama, la figura 28, representa los componentes que tendremos, y los manejadores que crearemos, tanto el de entidades, como el de componentes, que más adelante indagaremos en por qué son necesarios. Como en proyectos anteriores, iremos avanzando creando esta estructura para nuestras entidades y componentes.

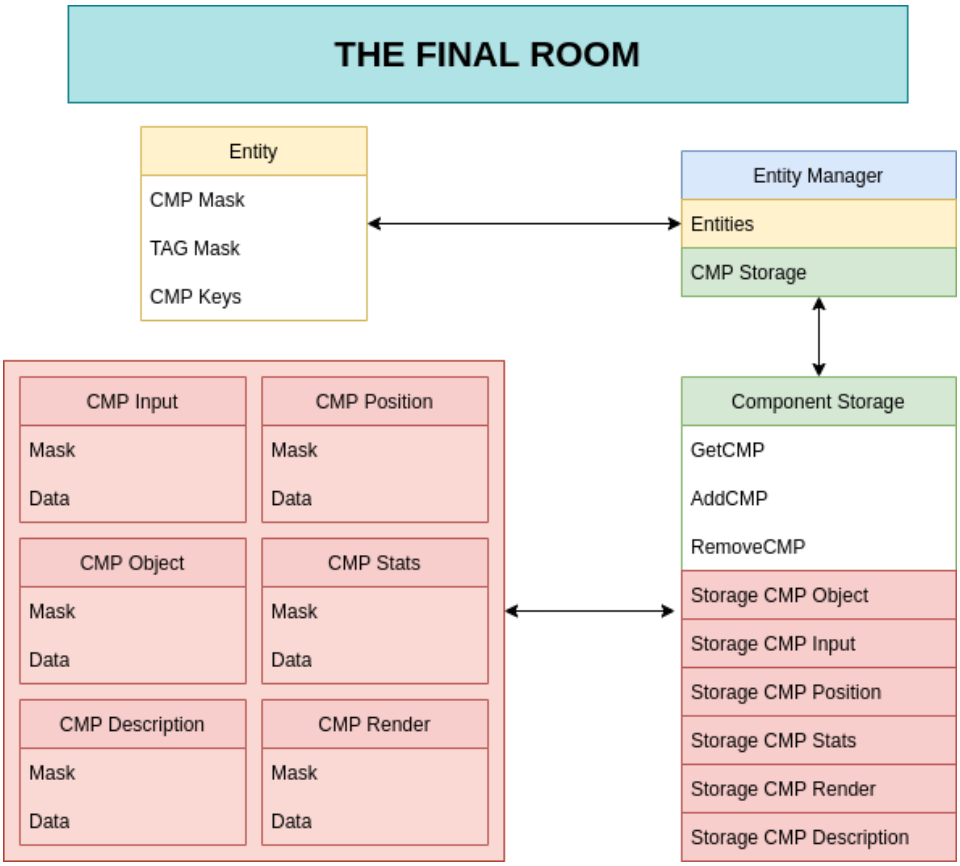


Figura 28. Diagrama de la estructura de entidades del Juego "The Final Room"

Dispondremos de componente de Input para mover al personaje, de posición, para saber hacia donde se mueven y en que posición se encuentran nuestras entidades, de render, para poder pintarlos en la pantalla, de estadísticas, para tener todos los datos de esa entidad, como el daño, la vida..., de descripción, para guardar que hace cada objeto, como ayuda al usuario y por último un componente de objetos, que guardará el tipo de cada objeto.

Tendremos una estructura "Component Storage", que almacenará nuestros componentes en estructuras de datos y al que tendremos que llamar para crear nuevos componentes, eliminarlos o recogerlos para usarlos en los sistemas. De esta forma las entidades se quedarán con una máscara de bits, que representará los componentes que tiene.También guardará las llaves a sus componentes, que estudiaremos en el siguiente apartado.

6.2 Implementación y estructuración del motor de entidades

En esta primera parte, vamos a implementar un motor que nos permita crear un juego, más amplio que los anteriores, más optimizado y más estructurado. Para ello vamos a remodelar las entidades, crearemos estructuras para almacenar nuestros componentes y vincularemos ambas partes. Además gestionaremos todo con el manejador de entidades.

Los sistemas del juego los veremos en el siguiente apartado, donde implementaremos el juego propuesto más a fondo.

6.2.1 Teoría: Estructuras de datos, los Slotmaps

Los datos de todas las instancias de un componente se almacenan normalmente en la memoria física (la memoria que se almacena temporalmente mientras se ejecutan los programas, la RAM), lo que permite un acceso eficiente a la memoria para los sistemas que operan con las entidades.

Aun así, la técnica de optimización que utilizaremos no se basa en esto ya que todavía es más óptima, los "Slotmaps".

Los Slotmaps son estructuras de datos muy utilizadas en videojuegos (las denominadas matrices dispersas), y especialmente son estructuras muy útiles para almacenar componentes, y optimizar la memoria, haciendo que nuestros componentes estén guardados en la caché que es hasta cien veces más rápida que la memoria RAM estándar.

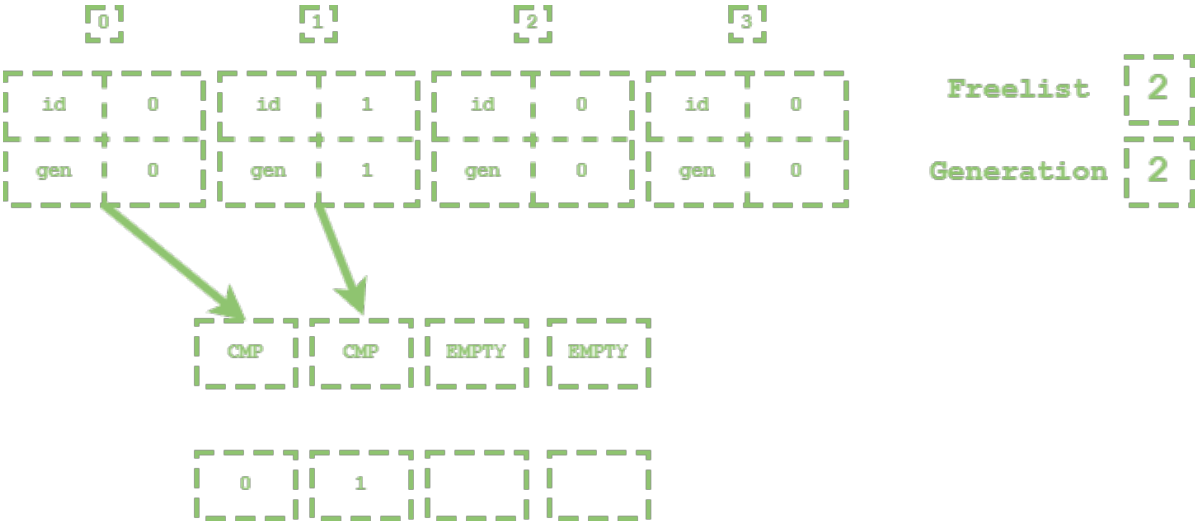


Figura 29. Estructura de un Slotmap.

Estas estructuras de datos están compuestas por 3 arrays y 2 variables numéricas. Estos tres arrays se dedican a almacenar distintos parámetros y datos. El primer array es el array "índices" el cual almacenará pares de dos valores en cada posición. Uno de estos

valores será el id, que guardará el lugar donde está almacenado el componente en el array de "data". El siguiente valor es el generador, que cada vez que tengamos un nuevo elemento en el vector de índices incrementa en uno. De esta forma nos aseguramos de que en el vector de índices nunca haya dos pares iguales, ya que cada par de valores representa a un componente.

Los "índices" nunca los moveremos de lugar. Esto quiere decir que puede haber huecos entre pares de valores. Para eso tenemos la variable "freelist", que almacena la posición del primer hueco libre en este array.

El siguiente array será el "data" donde almacenaremos el componente con todos sus parámetros. En este array no habrá huecos entre los componentes, ya que, si se elimina uno, el último elemento del array pasa a la posición libre y esto también se verá reflejado en el array de índices.

El último será el array "erase" que contendrá el índice del componente en su misma posición en el array "índices". Esto nos servirá para modificar el id en el array "índices" al hacer un movimiento, y para saber en el momento que eliminamos un componente, qué índice ocupaba, para eliminarlo también.

Las variables de esta estructura son dos como hemos comentado, una la freelist, la posición libre del array índices, y la otra es el generator, que es el valor que aumenta en 1 cada vez que hay una operación de borrado o creación en el array de índices. Ambas comentadas anteriormente.

En la figura 29, se puede apreciar la estructura de un Slotmap, con varios componentes guardados.

6.2.2 Separación de entidades y componentes

A continuación vamos a crear el primer componente de nuestro juego, con el que iremos iterando hasta conseguir nuestro objetivo de mejorar nuestra estructura de entidades-componentes y almacenamiento de los mismos de una forma optimizada.

Para ello crearemos el componente de render que nos permitirá pintar en la ventana del juego el sprite de la entidad pertinente, e incluso saber cual de los sprites pintar, en el caso de tener más de uno, dando lugar a modificaciones, por ejemplo al abrir una puerta o romper una roca.

Componente de Render

```
struct RenderCMP {
    const char* route {}; // Ruta del sprite
    Texture2D sprite { LoadTexture(route) }; // Leemos la textura
    int actual_frame = 0; // Frame a leer
```

```
    Rectangle  frame = {static_cast<float>(actual_frame*32), 0,
        32.0f, static_cast<float>(sprite.height)}; // Tamaño
    static const int mask{0b1}; // Máscara del componente, el bit 1.
};
```

La máscara de nuestro componente será muy útil y lo veremos cuando creemos las entidades, las cuales tendrán una máscara con la que podremos comprobar si contiene un componente o no, para que de una comprobación lógica sencilla, tengamos el resultado, y no haya que buscar un componente en ningún tipo de estructura. Será "static" por que de esta forma la podremos llamar aunque el componente no esté instanciado, y será muy útil para referirnos a componentes de ese tipo concretamente. Además será constante, ya que no puede cambiar en ningún momento, porque será la identificación de los componentes.

Crearemos a continuación la entidad, tal y como la teníamos hasta la fecha.

Entidad antigua

```
struct Entity {
    int id;
    std::optional<RenderCMP> render;
};
```

Si nos paramos a pensar en lo que estamos haciendo, nos daremos cuenta que cada entidad contendrá un tamaño reservado para el componente de render, y puede que haya entidades que no lo requieran. Estamos usando mucha memoria que no necesitamos y a la larga, al tener cientos de entidades, puede ser un gran problema de rendimiento en nuestro juego. Además ahora solo tenemos un componente, pero imagina si tuviésemos diez diferentes.

Es por esto que necesitamos una solución, y la implementaremos a continuación, ya que queremos optimizar al máximo nuestro motor, y darle más flexibilidad. Para ello, crearemos el "Component Storage", una clase que contendrá los componentes almacenados en estructuras de datos, y métodos para el manejo de estos. Además, esas estructuras de datos serán los Slotmaps, de los que venimos hablando en apartados anteriores. El primer paso será implementar un Slotmap.

6.2.3 Implementamos los Slotmaps

Llegado a este punto necesitaremos crear una estructura de datos que cumpla nuestros requisitos de optimizar y almacenar cada componente, e implementar alguna forma de recogerlos de forma rápida y óptima.

Para ello iremos creando cada una de las partes que forman estos Slotmaps, comenzando desde cero, una estructura vacía.

Slotmap

```
struct Slotmap { };
```

Comenzaremos añadiendo las estructuras que en el apartado teórico hemos explicado, el array de data, erase e índices, y las dos variables "freelist" y "generation"

Slotmap

```
struct Slotmap {
    std::uint32_t                       freelist_{};
    std::uint32_t                       generation_{};
    std::array< ? , 10 >                indices_{};
    std::array< std::uint32_t, 10 >     erase_{};
    std::array< RenderCMP, 10 >         data_{};
};
```

La elección de estructura para "índices", "data" y "erase" es "array", aunque esto nos limita a que no podamos redimensionar estas estructuras dependiendo de los datos que tengamos, que lo podemos conseguir con vector, pero usar vector requiere implementar aún más detalles en los Slotmaps, para controlar cuando cambian de tamaño. Por lo tanto, comenzaremos usando "array", lo que hace que esta versión de los Slotmaps tengan almacenamiento fijo , y los objetos almacenados sean trivialmente destruibles. Esto simplifica la estructura, y simplemente colocaremos un tamaño fijo para estos arrays asequible y suficiente para la memoria que nuestro juego pide.

Tenemos que pensar que tipo de datos usará nuestro array de índices ya que si recordamos, funciona almacenando el índice de el componente del array data, y una variable "generation" que hace que este par de valores nunca coincida con otros componentes, por lo tanto los hace únicos. Además es lo que necesitaremos para recoger un componente, la llave que nos dice donde se encuentra y si existe. Este par de valores lo tendremos que guardar en nuestra entidad para recoger el componente posteriormente. Este par de valores puede estar formado con una estructura de la siguiente forma:

Estructura llave de un componente

```
struct { std::uint32_t id; std::uint32_t gen; };
```

Por lo tanto necesitamos un tipo que represente esta estructura, y para eso usamos "using" que nos servirá para crear un tipo de un objeto concreto.

Estructura llave de un componente

```
using   key_type = struct { std::uint32_t id; std::uint32_t gen; };
```

De esta forma "key_type" será el tipo de la estructura con dos valores dentro, y lo podremos usar para crear nuestro vector de índices, ya que cada índice será un "struct" de este par de valores. Por lo tanto, ahora mismo nuestro Slotmap tendrá la siguiente forma, antes de comenzar a implementar todas las funcionalidades:

Slotmap: key_type

```
#define Capacity 10
struct Slotmap {
    using key_type = struct { std::uint32_t id; std::uint32_t gen; };
    private:
    std::uint32_t                       freelist_{};
    std::uint32_t                       generation_{};
    std::array< key_type , Capacity >       indices_{};
    std::array< std::uint32_t, Capacity >   erase_{};
    std::array< RenderCMP, Capacity >       data_{};
};
```

Comenzaremos desarrollando el constructor de esta estructura, el cual simplemente tiene que inicializar la "freelist" a la primera posición libre del "array" de índices y establecer que no hay ningún componente dentro del "array data". Justo lo mismo que haría una función para borrar todo el contenido. Como son trivialmente destruibles, bastaría con olvidarnos de ese componente, y el siguiente que vaya en esa posición lo sobrescribirá. Entonces, el primer paso es crear una función de borrado que inicialice la "freelist", y deje claro que el tamaño actual es cero. En el constructor, llamaremos a esta función, que hará lo mismo que un borrado inicial.

Antes de continuar, hay un detalle que se nos pasa por alto, y es que el tamaño de nuestro Slotmap no está definido, es decir, habrá que definir también una variable "size", la cual nos muestre en cada momento cuantos componentes hay almacenados, aunque la capacidad sea de diez por ejemplo, si hay un solo componente almacenado, "size" será uno.

Slotmap: Size Inicialización y Clear

```
struct Slotmap {
public:

    ...
```

```
    explicit Slotmap() { clear(); }
    std::size_t size() { return size_; }
    bool clear() { freelist_init(); size_ = 0; return true;}
    private:
    void freelist_init() noexcept {
        for(std::uint32_t i{}; i < indices_.size(); ++i){
            indices_[i].id = i+1;
        }
        freelist_ = 0;
    }
    std::uint32_t size_{};

    ...

};
```

Para inicializar la "freelist", y a su vez el Slotmap, tendremos que asegurarnos de que todos los índices de este array, están inicializados, por eso recorremos el array inicializando cada uno de ellos.

Antes de seguir, podemos definir los get que aún no hemos definido:

Slotmap: Métodos get

```
struct Slotmap {
public:

    ...

        std::size_t capacity()  const noexcept { return Capacity; }
        std::array<RenderCMP, Capacity>& getData(){ return data_; }
        auto getFreelist(){ return freelist_; }

    ...

};
```

El "getData" nos devolverá una referencia al array data, con todos los componentes, que la podremos necesitar en algún momento proximamente y "getFreelist" que simplemente nos devolverá la freelist. Usamos auto, por que en un futuro puede que el tipo de la freelist sea otro diferente al actual, y de esta forma lo automatizamos. Por último, "capacity" nos devolverá la constante definida anteriormente.

Vamos a proceder a añadir un componente a nuestro Slotmap. Para ello tendremos que tener en cuenta una serie de pasos previos que tendremos que dar para conseguir este

objetivo.

- En primer lugar tendremos que reservar un "slot" del "array índices", el cual va a referenciar a la posición donde está guardado nuestro componente en el "array data". Para reservar el "slot" deberemos comprobar si existe. El proceso de reserva de "slot" funciona creando uno que tiene como id la "freelist". La "freelist" a continuación pasa a ser el id de ese "slot", que apunta al siguiente. En otras palabras, se cambian los valores, el índice del "slot" apuntará a la posición de "índices" donde estará guardado nuestro componente y la "freelist" apuntará al siguiente libre. Seguimos inicializando el índice, haciendo que la id sea el "size" actual y "gen" pasa a ser el valor de "generation". Por último, aumentamos el "size" del Slotmap y el "generation", para que ningún otro componente sea igual.

Slotmap: Reserva de slot

```
struct Slotmap {
...
private:
std::uint32_t allocate() {
  if (size_ >= Capacity)
    throw std::runtime_error("No space left in the slotmap");
  assert(freelist_ < Capacity);
  //Reservamos el slot
  auto slotid = freelist_;
  freelist_ = indices_[slotid].id;//freelist al 1er slot libre
  //Inicializamos el slot
  auto& slot = indices_[slotid];
  slot.id = size_;
  slot.gen = generation_;
  //Actualizamos
  ++size_;
  ++generation_;
  return slotid;
}
...
};
```

  Esta función nos devuelve el índice del "slot" reservado en el "array índices".
- Ahora que tenemos una función que reserva un "slot" en caso de que exista el espacio necesario, crearemos la función para añadir un componente al "array data". Para ello tenemos que llamar a la función "allocate()" para reservar el id, obtener el "slot", mover el valor o componente que nos llega por parámetro al "array data" en la posición que marca el "slot", y generar una llave para guardarla en nuestra entidad. De esta forma, podremos recoger el componente o eliminarlo en otro momento.

Slotmap: Push_back() - Añadimos un componente

```
struct Slotmap {
public:
...
    key_type push_back(RenderCMP&& newVal) {
        auto reservedid = allocate();
        auto& slot = indices_[reservedid];
        //Movemos el componente recibido
        data_[slot.id]  = std::move(newVal);
        erase_[slot.id] = reservedid;
        //Key para la entidad
        auto key { slot };
        key.id = reservedid;
        return key;
    }
...
};
```

En este punto tenemos una función que reserva un "slot", mueve el componente a su espacio en la memoria, y devuelve la llave para almacenarla y poder recuperar los datos almacenados.

- Nos falta el broche final para este "push_back" y es crear un nuevo "push_back" para cuando nos llega por parámetro un componente por referencia. Actualmente nos llega un objeto que ha sido creado en la misma llamada a la función, es decir se ha creado el componente al llamar a la función:

Ejemplo de llamada a push_back(RenderCMP newVal)

```
    push_back(RenderCMP{});
```

Para finalizar añadiremos una función "push_back" que reciba por referencia y llame a la función que ya tenemos para añadirlo de la forma más sencilla posible:

Slotmap: Función para añadir componentes por referencia constante

```
struct Slotmap{
public:
...
    key_type push_back(RenderCMP const& newVal) {
        return push_back( RenderCMP{ newVal } );
    }
...
```

```
};
```

Finalmente, tenemos forma de agregar elementos al Slotmap. Para recapitular, llamamos a la función "push_back", que reserva un "slot", con la función "allocate", guarda el componente en el "slot" reservado si hay hueco, y nos devuelve una llave, que más tarde nos encargaremos de controlar y manejar.

Ya podemos decir que podemos añadir componentes, solo tendremos que hacer un "push_back" de la misma forma que haríamos en un vector.

Seguiremos añadiendo funcionalidad a esta nueva estructura que estamos creando. Es momento de implementar un borrado, pero antes de proceder con la implementación vamos a hacer una lista con todos los pasos que tenemos que dar para efectuarlo, donde manejemos de forma correcta nuestra memoria.

- En primer lugar tendremos que comprobar que la "key" que nos pasa el usuario sea correcta y coincida con algun slot de los que tenemos en funcionamiento actualmente. De lo contrario tendremos un problema que provocará que el método de borrado nos devuelva "false", dando a entender que no ha sido posible el borrado.
- Después de esta comprobación, tendremos que liberar el "slot" que reservamos al añadir el componente al Slotmap, para ello necesitaremos una función "free()".
- Por último devolveremos "true" ya que si hemos liberado un "slot", significa que hemos borrado un componente.

Nos centraremos primero en la función de comprobación, que nos devolverá "true" si la llave tiene un "slot" asociado y "false" en otro caso. Para esto, tendremos que comprobar que el id de la llave no sea mayor que el "Capacity", para asegurarnos que es un "slot" real. También comprobaremos que la variable "gen" de la llave sea exactamente el mismo que el del "slot" del "array índices" que corresponde con la "id" de la llave.

Slotmap: Comprobamos si la "key" es válida

```
struct Slotmap{
public:

...

    bool is_valid(key_type key){
        if ( key.id >= Capacity || indices_[key.id].gen != key.gen ){
            return false;
        }
        return true;
    }

...
```

```
};
```

Podemos proceder, ahora si al borrado de un elemento del Slotmap. Para ello nos pasaran una llave y comprobaremos si es válida con la función "is_valid". Posteriormente liberaremos el "slot" correspondiente a la llave. Para esto, crearemos la función "free(key_type key)", que recibe la llave del "slot" a borrar.

En primer lugar colocaremos un "assert(key)", que nos servirá a nosotros como programadores para darnos cuenta de errores cometidos, ya que si llega una llave errónea, significará que algo estamos haciendo mal. Procedemos guardándonos el "slot" correspondiente al "id" de la llave para un mejor manejo. También guardamos donde está guardado nuestro componente en el "array data", cuya posición está guardada en la variable "id" del "slot" que hemos guardado anteriormente. Actualizamos la "freelist", y el "slot" que estamos liberando.

Una vez logrados estos pasos, hace falta una comprobación para saber si hay que hacer algo más o no, ya que si el elemento a borrar es el último del "array data", no tendremos que hacer nada. Por otro lado, si no es el último, tendremos que mover el último al espacio del "array data" que acabamos de borrar.

El "array data" y "erase", no tienen que tener huecos de por medio. El único que los puede tener es el "array índices", que a su vez, una vez se ha reservado un "slot", ese "slot" no cambiará de posición nunca, ni será modificado hasta ser liberado.

Si nos toca mover el último elemento de data al nuevo hueco, lo haremos y posteriormente decrementaremos el "size" del Slotmap. Seguidamente aumentaremos el "generation" para evitar errores, ya que el "slot" liberado tendrá el valor que había actualmente. A continuación esta función "free", queda plasmada de la siguiente forma:

Slotmap: Liberamos un slot - free(key_type key)

```
struct Slotmap{

...

private:
    void free(key_type key) noexcept {
        assert(is_valid(key));
        auto& slot = indices_[key.id];
        auto dataid = slot.id; //Guardar el id del array data
        //para comprobar si es el último componente del array
        //Actualizamos freelist y el slot liberado
        slot.id = freelist_;
        slot.gen = generation_;
```

```
        freelist_ = key.id;
        //Comprobamos si es el último cmp
        if ( dataid != size_ - 1 ){
            //Si es el último, lo movemos al hueco liberado
            data_[dataid]   = data_[size_ - 1];
            erase_[dataid]  = erase_[size_ - 1];
            indices_[erase_[dataid]].id = dataid;
        }
        //Actualizamos variables
        --size_;
        ++generation_;
    }

...

};
```

Teniendo lista esta función, podemos crear la función que utilizará el usuario para borrar un componente, la función "erase", que recibirá la llave del componente a borrar y devolverá verdadero en caso de haber borrado el componente y falso en todos los demás casos.

Slotmap: Borramos un componente

```
struct Slotmap{
public:

...

    bool erase(key_type key) noexcept {
        if ( !is_valid(key) ){ return false;};
        //Borramos
        free(key);
        return true;
    }

...

};
```

Con esta implementación del Slotmap podemos funcionar creando y borrando componentes para nuestras entidades a nuestro placer, simplemente recogiendo y manejando las llaves, que lo veremos próximamente. Pero antes de terminar, nos falta alguna forma de recoger un componente del Slotmap utilizando la llave que nos devolvió al crearlo. Para

esto, no implementaremos un "get" o algo similar, si no que sobrecargaremos el operador corchete "[]" para que pasándole esa llave, nos devuelva dicho componente si existe.

Para ello, simplemente comprobaremos si es válida la llave en un "assert", ya que si falla es que algo hemos hecho mal, por que sería un "segmentation fault", accederíamos de forma errónea a la memoria. Seguidamente, recogeremos el "id" del "slot" que nos marca el id de nuestra llave, y con ese id obtendremos del "array data" nuestro componente. Esta sobrecarga de operador queda de la siguiente forma, devolviéndonos o una referencia de un componente o una referencia constante a un componente, según el parámetro:

Slotmap: Operador corchete para obtener un componente

```
struct Slotmap{
public:

...

    RenderCMP& operator[](key_type const& key) {
        assert(is_valid(key));
        auto dataID = indices_[key.id].id;
        return data_[dataID];
    }
    RenderCMP const& operator[](key_type const key){
        assert(is_valid(key));
        auto dataID = indices_[key.id].id;
        return data_[dataID];
    }

...

};
```

También como extra, podemos añadir unos iteradores, uno normal y otro constante, para poder hacer cosas como la que se muestra en el comentario del siguiente código:

Iteradores

```
struct Slotmap{

...

    using   iterator = RenderCMP*;
    using  citerator = RenderCMP const*;

...
```

```
};
/* Con iteradores podemos recorrer el Slotmap de la siguiente forma:
for(auto&cmp : slotmap){...}
*/
```

Ya tenemos un Slotmap para un tipo de componentes, el de Render. Ahora tendremos que realizar un Slotmap distinto para cada tipo de componente que hagamos en el juego. Esto es muy costoso, tanto de esfuerzo como de código, y es por eso que podemos usar las plantillas, tal y como las usan las estructuras más conocidas como los vectores o arrays. Para ello definiremos un tipo en la plantilla y crearemos un Slotmap configurable para cada componente, de esta forma optimizamos muchísimo el código, el juego y el trabajo.

Para llevar a cabo esta última actualización, simplemente habrá que añadir antes de la declaración del Slotmap la cabecera de plantilla y modificar cada lugar de nuestro Slotmap donde ponga RenderCMP por el tipo definido en esa plantilla, aquí te dejo un ejemplo:

Slotmap: Template para configurar el Slotmap

```
template <typename DataType, std::size_t Capacity = 300>
struct Slotmap{

...

    //Usando este tipo donde antes ponía RenderCMP ahora value_type
    using value_type = DataType;

...

};
```

Además podemos configurar el "Capacity" que queremos para cada Slotmap de un componente, es decir, si en nuestro juego todas las entidades tienen render, podemos configurar el Slotmap de componentes de render con un "Capacity" más alto que un Slotmap para componentes de otro tipo donde hagan falta la mitad. De esta forma también optimizamos la memoria.

6.2.4 Teoría: Máscaras y operaciones lógicas

Las máscaras son elementos útiles que nos servirán en el caso de los componentes, para saber si tenemos o no ese componente, o en caso de las "tags" o etiquetas, para marcar a nuestras entidades con un bit que represente un tipo de entidad.

Estas máscaras se entenderán mejor a lo largo del desarrollo, cuando las vayamos utilizando, pero van a tomar un papel muy importante en el manejo de nuestras entidades.

Vamos a ver las tres operaciones lógicas que más se usan cuando trabajamos con máscaras y para qué nos servirán concretamente. Comenzaremos con las operaciones lógicas "AND". Supongamos que tenemos una entidad con máscara "0b0110" ("0b" al principio se usa para hacer saber que es un número binario). Y tenemos diferentes componentes con máscaras tal y como se aprecian en el código siguiente.

```
Entidad -> 110       CMP1 -> 001 CMP2 -> 010 CMP3 -> 100
```

Si queremos comprobar por ejemplo si la entidad tiene componente 1 haremos una operación "AND" y el resultado, tendrá que ser igual a la máscara del componente 1. Pues como sabemos, este tipo de operación lógica solo devuelve 1 si ambos valores son iguales.

```
(Entidad AND CMP1) == CMP1
Entidad -> 110
CMP1    -> 001
AND     -> 000 != CMP1
```

En este caso la entidad no dispone de CMP1. Ahora vamos a ver si contiene a CMP2.

```
(Entidad AND CMP2) == CMP2
Entidad -> 110
CMP2    -> 010
AND     -> 010 == CMP2
```

Si que dispone de componente dos, ya que la operación es igual que el mismo. De esta forma podremos hacer comprobaciones y ver si las entidades tienen o no componentes. Además, podemos comprobar agrupaciones de componentes, y comprobar varios a la vez. Esto será útil en los sistemas. Por ejemplo:

```
(Entidad AND (CMP2 | CMP3)) == (CMP2|CMP3)
Entidad   -> 110
CMP2|CMP3 -> 110
AND       -> 110 == (CMP2|CMP3)
```

Y esto me da paso a hablar de las operaciones "OR" que nos servirán para añadir y concatenar diferentes bits, como hemos visto en el ejemplo anterior. Una operación "OR" solo será uno si hay un uno ya en cualquiera de las máscaras. Solo será 0 si ambos valores son 0.

Por último, tenemos las operaciones "XOR" que funcionan al revés de las OR. Estas operaciones nos servirán para eliminar de la máscara un bit, por ejemplo cuando un componente es eliminado. Si en la operación ambos bits son 1, pasará a ser 0.

Por ejemplo, si queremos borrar el "CMP3" de la entidad, haremos una operación "XOR" sobre la entidad con la máscara del componente 3.

```
Entidad = Entidad XOR CMP3
Entidad   -> 110
CMP3      -> 100
XOR       -> 010 <- nueva máscara de entidad
```

6.2.5 Component Storage: Entidad y componentes ordenados

En primer lugar, vamos a retomar la Entidad. La tenemos actualmente con el componente en su interior, pero ahora tenemos estos Slotmaps que almacenarán los componentes, por lo tanto, en la entidad solo tendremos que tener un método para recuperar esos componentes. Para lograr este objetivo, solo necesitaremos la "key" que nos devolvió al crearlo. Es por eso que necesitaremos un objeto de ese tipo en nuestra entidad.

Cada programador ordena sus estructuras y códigos a su gusto, y esto tiene que ver con la forma de recoger nuestras llaves o "keys". En este caso tenemos que almacenar todas estas llaves en la entidad, de los componentes que existan, para que en el caso de tener componente guardemos la llave. No importa tener todas las llaves dentro de la entidad, ya que ocupan mucho menos espacio que un componente entero. En mi caso, estas llaves estarán dentro de la entidad sin más, pero se pueden hacer diferentes estructuras para almacenarlas, como una tupla (ya que podemos introducir elementos de tipos diferentes).

Para obtener el tipo de la llave de un componente, simplemente tendremos que acceder a la definición de ese componente. Por ejemplo para obtener el tipo de la llave del componente de Render, tendremos que definir el Slotmap de render igual que lo tendremos en el "Component Storage" más adelante, y después acceder a su tipo definido "key_type" sin necesidad de crear el Slotmap.

Las entidades quedan de la siguiente forma:

Entity: Nuestra primera key

```
struct Entity{
    int cmpMask{0b0};
    Slotmap<RenderCMP>::key_type renderKey;
}
```

También definimos la máscara de componentes vacía por defecto, ya que de primeras la entidad no tendrá componentes, y esto nos da pie a hacer una función para acomprobar

si un componente está en la máscara y así en la entidad. Haremos una función "hasComponent" que reciba una máscara de un tipo de componente y nos devuelva "true" si está contenido en la máscara y "false" en otro caso.

Entity: comprobamos si tiene un cmp

```
struct Entity{

...

    bool hasComponent(int m){
        if((cmpMask & m) == m){ return true;}
        return false;
    }

...

};
```

Esta función recibe una máscara como hemos programado y hace una operación lógica "AND" con la máscara de la entidad. Si el resultado es igual a la máscara del componente del parámetro devolverá "true", significando que la entidad dispone de ese componente.

Ahora tendremos que definir el "Component Storage", la clase que manejará los componentes, y donde tendremos definido el almacenamiento de los mismos. Comenzaremos simplemente con un solo componente, como con los Slotmaps. Necesitamos crear el Slotmap para este componente y diferentes métodos que lo gestionen de cara al usuario.

Entity: Nuestra primera key

```
struct ComponentStorage{
    //Método para añadir un CMP
    //Método para eliminar un CMP
    //Método para recoger un CMP
    private:
    Slotmap<RenderCMP> renderStorage{};
};
```

Llegados a este punto, tendremos un Slotmap para los componentes "RenderCMP", pero no tenemos forma de manejarlo desde la parte del usuario, ya que ese Slotmap será privado. Definiremos las tres acciones a realizar sobre el Slotmap que serán públicas y utilizadas por el usuario.

Comenzaremos añadiendo un componente al Slotmap. Tendremos dos sobrecargas de la

misma función, al igual que en Slotmap. Son las dos formas de añadir un componente, por lo tanto, tendremos que representar con dos sobrecargas esas funcionalidades.

En la primera de ellas recibiremos un componente creado en el parámetro, lo que significa que será una referencia a rvalue o a un valor en movimiento. Estas referencias son útiles cuando se quiere mover el valor a otro lugar, para no tener que copiar grandes cantidades de datos.

Component Storage: Añadimos un componente

```
struct ComponentStorage{

...

    void addRenderCMP(RenderCMP&& cmp, Entity& e){
        //Comprobamos que esa entidad no tenga ese componente ya
        if(!e.hasComponent(RenderCMP::mask)){
            //Añadimos el componente al slotmap y
            //guardamos la llave en la entidad
            e.renderKey = renderStorage.push_back(cmp);
            //actualizamos la máscara de la entidad
            e.cmpMask = e.cmpMask | cmp.mask;
        }
    }

...

};
```

Recibiremos el componente a añadir y la entidad a la que se añadirá dicho componente. En primer lugar comprobamos si la entidad ya tiene ese componente. Si no lo tiene, lo añadiremos, y para ello tendremos que guardar en la llave de la entidad correspondiente al componente de render, lo que nos devuelve la función "push_back" del Slotmap. Por último actualizaremos la máscara de la entidad haciendo una operación lógica "OR" a la entidad con la máscara del componente.

Pero, ¿qué ocurre si la entidad tiene ya componente? Cada programador puede decidir en estos casos lo que quiera hacer con su código y sus funcionalidades, pero en mi caso, añadiremos una mejora a este método haciendo que en caso de tener ya el componente, se cambie por el nuevo del parámetro. Recogeremos el componente actual por referencia, y lo cambiaremos por el nuevo.

Component Storage: Añadimos un componente

```
struct ComponentStorage{
```

```
...

    void addRenderCMP(RenderCMP&& cmp, Entity& e){
        //Comprobamos que esa entidad no tenga ese componente
        if(!e.hasComponent(RenderCMP::mask)){

        ...

        }else{
            //Si tiene ese componente lo sobrescribimos
            auto& cmpRender = renderStorage[e.renderKey];
            cmpRender = cmp;
        }
    }

...

};
```

Con esta función ya podemos añadir o sobrescribir componentes en nuestras entidades. Pero nos falta otra que reciba un componente por referencia. Para ello crearemos un componente como parámetro de la función que ya tenemos y lo construiremos con el del parámetro de esta.

Component Storage: Añadimos un componente

```
...

void addRenderCMP(RenderCMP& cmp, Entity& e){
    addRenderCMP(RenderCMP{ cmp }, e);
}

...
```

Ya tenemos la funcionalidad de añadir componentes a un Slotmap y guardarnos la "key" en la entidad, modificando la máscara de componentes. Ahora que hemos conseguido esto, haremos lo mismo para el borrado de un componente.

Para desarrollar una función de borrado, pediremos por parámetro la entidad de la cual queremos borrar el componente de render (recordamos que esto es solo para un tipo de componente) y comprobaremos si tiene ese componente. De no tenerlo, devolveremos "false" y no haremos nada más, pero si lo tenemos, llamaremos a la función "erase" del Slotmap, que eliminará el componente y actualizará el Slotmap de la forma que vimos al crearla. El siguiente paso, será modificar la máscara de la entidad, ya que no

dispondrá ya de ese componente, para ello realizaremos una operación lógica, en este caso "XOR", que pasándole la máscara del componente de render, modificara a cero el bit correspondiente. Finalmente, actualizaremos y haremos nula la llave al componente de render que guardamos en nuestra entidad.

Component Storage: Añadimos un componente

```
struct ComponentStorage{

...

    bool removeRenderCMP(Entity& e){
        if(e.hasComponent(RenderCMP::mask)){
            //Eliminamos el componente de render
            renderStorage.erase(e.renderKey);
            //Actualizamos la mascara de la entidad
            e.cmpMask = e.cmpMask xor RenderCMP::mask;
            //ELiminamos la key de la entidad
            e.renderKey.id = 0;
            e.renderKey.gen= 0;
            return true;
        }
        return false;
    }

...

};
```

El último método necesario para tener la funcionalidad completa sobre el manejo desde el "Component Storage" de nuestros Slotmaps, es la función que nos devuelva el render de una entidad, pasándole como parámetro dicha entidad.

Esta función es la más sencilla de todas las anteriores, pues simplemente comprobaremos si existe el componente de render en la entidad pasada por parámetro y usaremos la sobrecarga que hicimos anteriormente de el operador corchete del Slotmap para recoger ese valor. Esto nos devolverá una referencia al componente de render. De fallar la comprobación, deberemos lanzar una excepción o algo similar, porque tendremos errores de programación.

Component Storage: Añadimos un componente

```
struct ComponentStorage{
```

```
...

    RenderCMP& getRenderCMP(Entity& e){
        if(!e.hasComponent(RenderCMP::mask))
            throw std::runtime_error(" RenderCMP no existe ");
        return renderStorage[e.renderKey];
    }

...

};
```

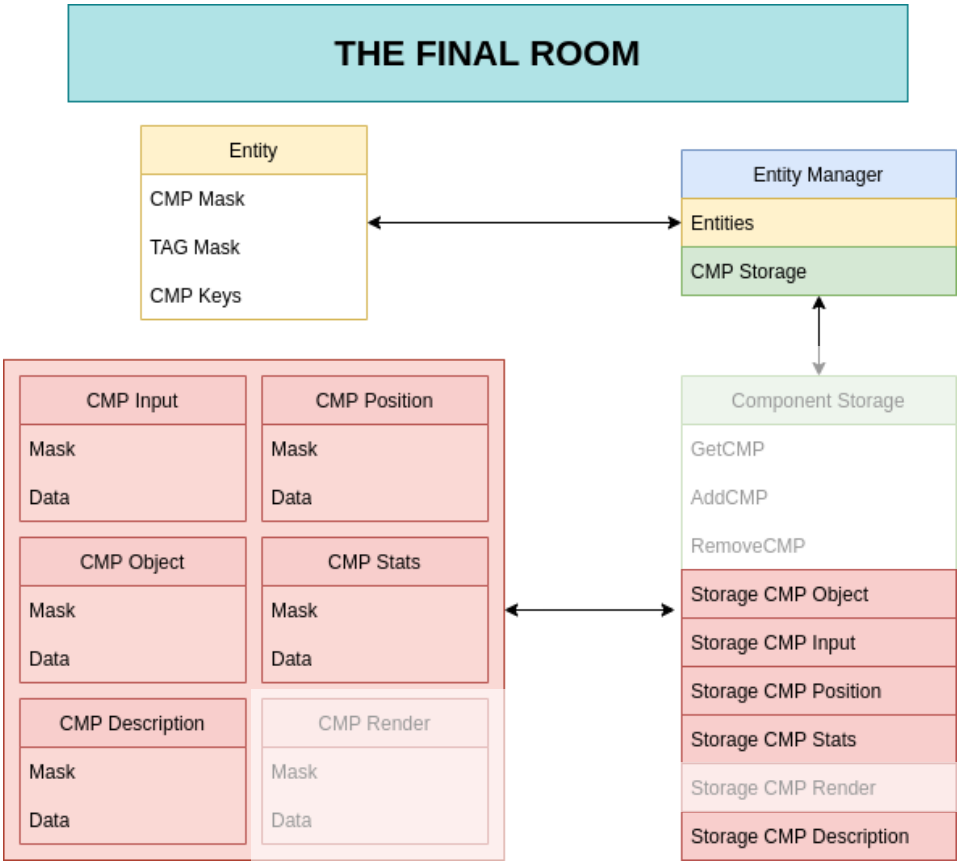


Figura 30. Diagrama de estructura de datos de "The Final Room" actual

Actualmente tenemos la funcionalidad completa para añadir, eliminar y recoger un componente de render del Slotmap en nuestro "Component Storage". Pero, ¿qué hacemos si tenemos diez componentes distintos?. Bien, tendremos que repetir este código completo para cada uno de esos componentes y puede llegar a ser muy ineficiente. Es por eso, que existen formas de hacer una función única que se configure mediante una plantilla, y de esta forma solo tener una para todos los componentes.

Para este proyecto, es un paso gigante llegar a hacer ese avance, y como no tendremos tantos componentes, replicaremos este código para cada componente. En futuros proyectos, abarcaremos este problema, que no es tan simple de resolver como usar una plantilla, si no que necesitaremos otros detalles más complejos de la programación en C++. Esto queda pendiente, pero ahora seguiremos con el desarrollo de este motor.

En el diagrama 30, queda plasmado lo que llevamos hecho hasta ahora. Hemos creado la clase "Component Storage" con sus métodos para el componente de render, y el mismo componente de render.

6.2.6 Componentes del juego y etiquetas

Ahora teniendo todo esto en cuenta, podemos crear todos los componentes del juego, y todas las estructuras necesarias para manejarlos.

Teniendo el componente de render creado, proseguiremos con el de posicionamiento, para posteriormente en el sistema de render, saber donde dibujar el sprite.

Componente de posicionamiento

```
struct PositionCMP {
    int posX{}, posY{}, velX{}, velY{};
    static const int mask{0b10};
};
```

Este componente tiene dos variables que guardarán su posición y velocidad, que en este caso la velocidad representará a la casilla que nos moveremos, ya que es un tablero. Le añadiremos una máscara con el siguiente bit al que tenía el componente de render que era el primer bit.

Además de la creación del componente, tenemos que acordarnos de crear todo lo relacionado, como un Slotmap, sus funciones en el "Component Storage" y la "key" en la entidad.

Key de PositionCMP en la entidad

```
//Añadimos la nueva key a la entidad
Slotmap<PositionCMP>::key_type positionKey;
```

Funciones para gestionar los componentes en Component Storage

```
//Creamos las funciones del Component Storage
void addPositionCMP(PositionCMP& cmp, Entity& e){...}
void addPositionCMP(PositionCMP&& cmp, Entity& e){...}
```

```
PositionCMP& getPositionCMP(Entity& e){...}
bool removePositionCMP(Entity& e){...}
```

Almacenamiento del componente de posición

```
//Almacenamiento del componente en Component Storage
Slotmap<PositionCMP> positionStorage{};
```

Además de estos dos componentes anteriores, necesitamos un componente que recoja todos los parámetros estadísticos de nuestro jugador, como de los enemigos u otros objetos. Para esto crearemos el "statsCMP" y antes de crearlo, entendamos su utilidad y que contiene exactamente.

- Vida actual y máxima: Las entidades tendrán vida actual y vida máxima, esto nos servirá para entidades como el jugador o los enemigos, a quienes podremos modificar la vida durante la ejecución, con objetos, golpes, etc. También puede servir para estructuras destruibles.
- Pico: Únicamente el jugador usará este valor, ya que como hemos dicho anteriormente los obstáculos serán destruibles y esto lo conseguiremos aplicando el daño de la variable "pickaxe" a la vida del enemigo.
- Steps: Los pasos servirán para saber el numero de casillas que moveremos nuestra entidad. Podremos conseguir mejoras para avanzar más rápido.
- Critical: El golpe crítico, se aplicará teniendo en cuenta el porcentaje de esta variable, si es 5, aleatoriamente entre 100 posibilidades se aplicará 5 veces.

Además de todos estos valores, al ser el tercer componente que creamos, tendrá un bit activo en la tercera posición de su máscara, que será el bit que represente a este componente.

Componente de Estadísticas

```
struct StatsCMP {
    int health{};
    int maxhealth{};
    int damage{};
    int step{};
    int critical_hit{};
    int pickaxe{};
    static const int mask{0b100};
};
```

Los objetos tendrán estadísticas al igual que el personaje, y tendremos que sumarle estas estadísticas al recogerlo. Para ello, crearemos una función para que recibiendo este componente, se apliquen esas estadísticas a las del personaje. Creamos la función "addStats(StatsCMP)"

Componente de Estadísticas: Añadir estadísticas

```
...

void addStats(StatsCMP obj){
    health += obj.health;
    maxhealth += obj.maxhealth;
    damage += obj.damage;
    step += obj.step;
    critical_hit += obj.critical_hit;
    pickaxe += obj.pickaxe;
}

...
```

Al igual que con los componentes anteriores, también tendrán sus respectivas funciones en el "Component Storage" y la "key" en la entidad, pero obviaremos esa parte ya que es exactamente igual para el resto de componentes. Así que, podemos proseguir con el resto de componentes.

Tenemos componente de render, para dibujar a nuestro personaje, tenemos componente de posición y velocidad, para saber donde pintarlo, tenemos componente de estadísticas para saber cuanta vida o daño entre otros valores tiene, pero, ¿cómo controlamos hacia dónde moverlo y cuándo se mueve?. Crearemos el componente de input.

Este componente controlará los input de teclado para que nuestro personaje pueda moverse por el mapa o tablero. Este componente contendrá las teclas permitidas para movernos por nuestro juego. Serán "W", "A", "S" y "D", y como el resto de componentes, contendrá una máscara, siendo el cuarto componente que creamos, con únicamente el cuarto bit activo.

El componente de input queda plasmado de la siguiente forma:

Componente de Input

```
struct InputCMP {
    int KeyW = KEY_W;
    int KeyA = KEY_A;
    int KeyD = KEY_D;
```

```
    int KeyS = KEY_S;
    static const int mask{0b1000};
};
```

Con los componentes que tenemos actualmente, podemos hacer un juego, y no necesitaremos mucho más. Aun así, vamos a añadir dos componentes más, que facilitarán el trabajo más tarde, a la hora de conocer de que tipo es cada objeto que creamos. Aunque las estadísticas de ese objeto están guardadas en el "StatsCMP", no sabemos que tipo de objeto es, que nos vendrá bien en un futuro. Para esto creamos el "ObjectCMP".

Este componente esta basado en un enumerado que contiene los tipos de objetos que existen en nuestro juego. De esta forma podemos crear un objeto más tarde simplemente aplicándole un valor del enumerado. También posee la máscara de componente, con el bit quinto marcado.

Componente de objetos y enumerado

```
struct ObjectCMP{
    int obj;
    static const int mask {0b10000};
};
enum Objects{
    Health, Damage, Steps, Critical, Pickaxe
};
```

Ya hemos creado todos nuestros componentes, y recuerdo que "Component Storage" tiene que tener una instancia de Slotmap para cada uno de ellos y sus funciones para manejarlo. Además la entidad tiene que tener todas las llaves de los diferentes componentes listas por si esa entidad tuviese ese componente.

Tomate un respiro si has llegado hasta aquí, pues tienes un gran avance y una estructura base para proseguir con la creación de este juego.

Seguiremos hablando de las etiquetas, que como ya las he mencionado en apartados anteriores, simplemente serán más máscaras de bits, diferentes a las de los componentes, que agruparan tags descriptivos permitiéndonos diferenciar entidades simplemente con bits.

Por ejemplo podemos definir una que sea "player" y otra "enemy" y de un vistazo, sabremos que entidad es el jugador y cual el enemigo en los sistemas. O una que defina diferentes tipos de rocas, destruible o no destruible.

Para definirlas crearemos una estructura que defina todas y cada una de las que necesitaremos en nuestro juego, comenzando por las que he mencionado anteriormente, "player", "enemy", y una para los muros, "walls".

Componente descripción

```
struct Tags{
    static const int player  {0b001};
    static const int enemy   {0b010};
    static const int wall    {0b100};
};
```

Todas las máscaras serán "static const", ya que de esta forma las podremos usar aun sin componente creado y no se podrán modificar.

Teniendo creadas estas etiquetas, podemos terminar de crear la que será nuestra entidad finalizada.

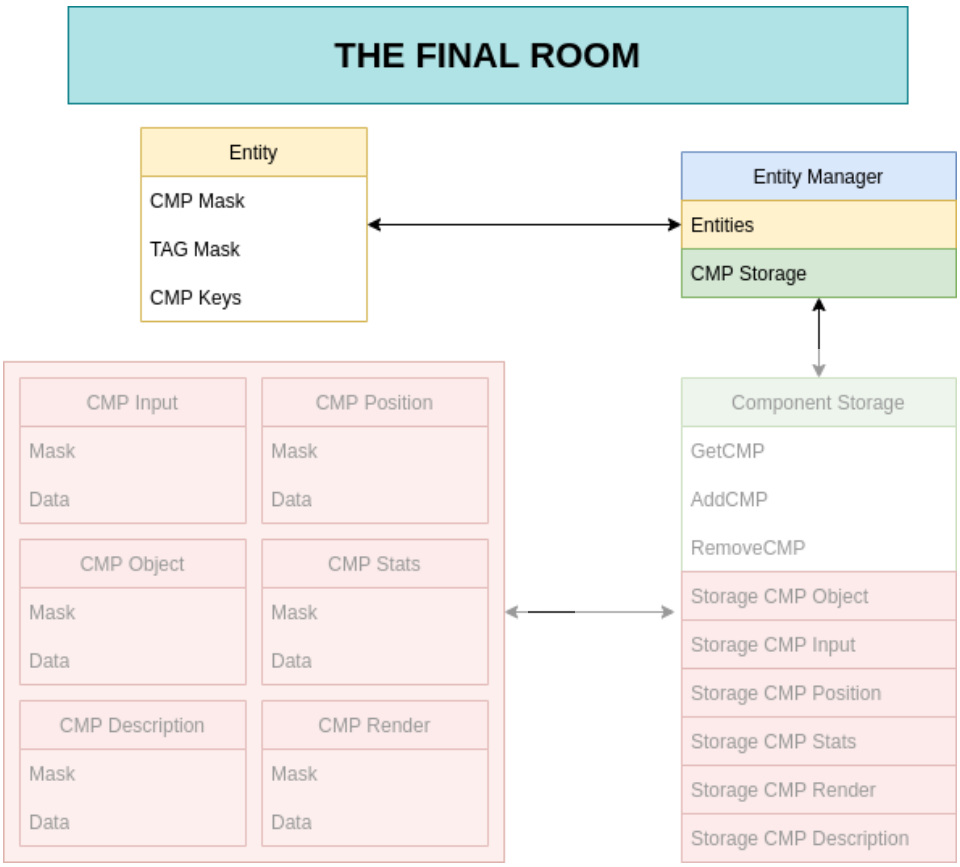


Figura 31. Diagrama de estructura de datos de "The Final Room" actual

Hemos avanzado mucho hasta el momento, manejando nuestros datos. En el diagrama de la figura 31 se puede ver el avance y el trabajo restante para conseguir nuestro motor de entidades preparado para crear nuestro juego.

6.2.7 Entidad final

A continuación vamos a terminar de definir nuestra entidad ya que disponemos de todos los componentes y de las etiquetas, que tendrán un uso muy importante en el flujo de la ejecución.

Entidad actual:

Entidad actual

```
struct Entity{
    int cmpMask{0b0};
    Slotmap<RenderCMP>::key_type renderKey;
    //MÉTODOS
    bool hasComponent(int m){...}
}
```

En la entidad que tenemos usamos una máscara de componentes para saber en todo momento cuales tenemos y una llave al componente de render. A continuación, podemos añadir el resto de llaves que hemos creado en el apartado anterior con cada uno de los componentes y además la nueva máscara que contendrá los tags que contiene dicha entidad.

Nueva entidad actual

```
struct Entity{
    int cmpMask{0b0};
    int tagMask{0b0};
    //KEYS
    Slotmap<RenderCMP>::key_type renderKey;
    Slotmap<PositionCMP>::key_type positionKey;
    Slotmap<StatsCMP>::key_type statsKey;
    Slotmap<InputCMP,1>::key_type inputKey;
    Slotmap<ObjectCMP,50>::key_type objKey;
    Slotmap<DescriptionCMP>::key_type descriptionKey;
    //MÉTODOS
    bool hasComponent(int m){...}
}
```

Tenemos una entidad casi completa, y si os preguntáis por qué hay llaves a las que le defino el tamaño del "Capacity" del Slotmap, es por que al definir ese Slotmap en el "Component Storage", también le definí ese tamaño. El por qué de esto, simplemente es que por ejemplo, de input solo habrá un componente así que solo me hace falta espacio para uno. Al igual que para los objetos, que requeriremos de menos espacio que el predeterminado.

Crearemos diferentes métodos para el manejo de las máscaras, ya que iremos añadiendo y quitando bits en cualquier momento. Un método para añadir una etiqueta, otro para comprobar si tiene esa etiqueta como hicimos con los componentes. Estos son los que necesitaremos.

Métodos para el manejo de máscaras

```
struct Entity{
    bool hasTag(int t){
        if((tagMask & t) == t){ return true; }
        return false;
    }
    void addTag(int tag){
        tagMask = tagMask | tag;
    }

    ...

};
```

Esta es nuestra entidad final, con la que podremos trabajar, sin tener todos los componentes en su interior, optimizándolas al máximo y dándoles una complejidad extra a como veníamos haciéndolas.

Más adelante, las usaremos y se podrán ver en funcionamiento.

Antes de terminar y dejar las entidades, sobrecargaremos el operador "==". De esta forma podremos comparar entidades. Para ello tendremos que igualar todas las llaves de componentes con las del parámetro e igual con las máscaras.

Operador "==" de Entity

```
struct Entity{
    bool operator==(const Entity& e2) {
        return cmpMask == e2.cmpMask
            && tagMask == e2.tagMask
            && renderKey.gen == e2.renderKey.gen
            && renderKey.id == e2.renderKey.id
            && positionKey.gen == e2.positionKey.gen
            && positionKey.id == e2.positionKey.id
            && statsKey.gen == e2.statsKey.gen
            && statsKey.id == e2.statsKey.id
            && inputKey.gen == e2.inputKey.gen
            && inputKey.id == e2.inputKey.id
```

```
            && objKey.gen == e2.objKey.gen
            && objKey.id == e2.objKey.id
            && descriptionKey.gen == e2.descriptionKey.gen
            && descriptionKey.id == e2.descriptionKey.id;
    }

    ...

};
```

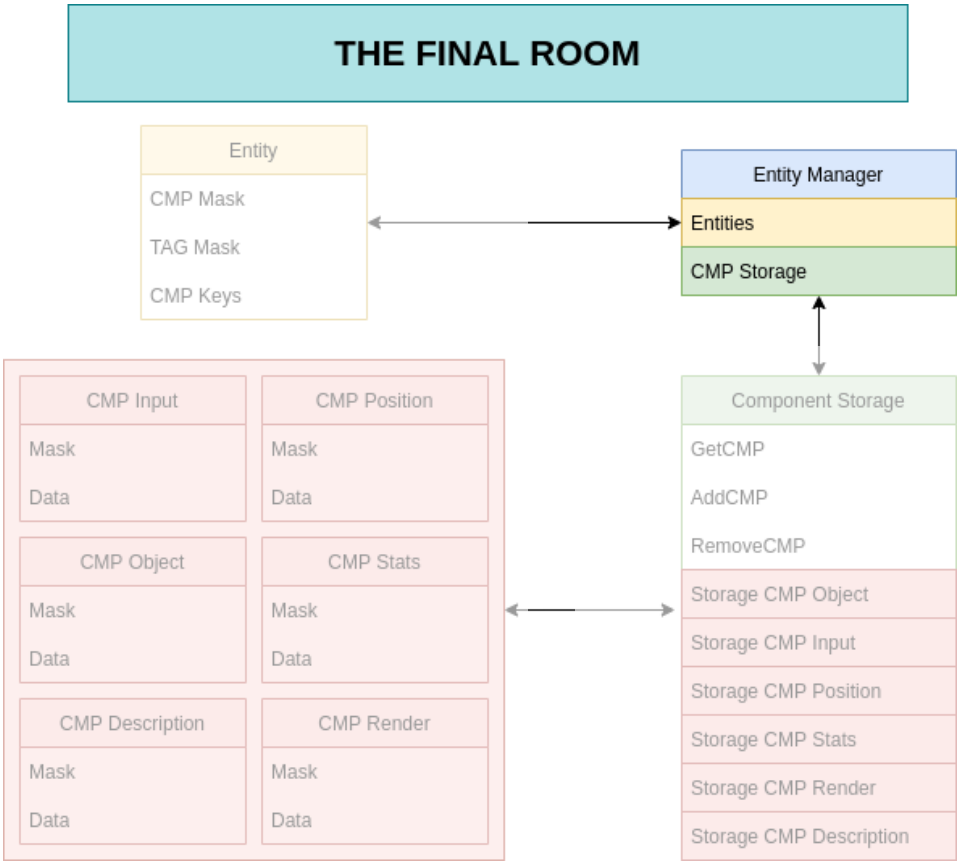


Figura 32. Diagrama de estructura de datos de "The Final Room" actual

Ya tenemos las entidades (32), prosigamos con la última parte, el manejador de entidades.

6.2.8 Manejador de entidades

Vamos a mejorar el manejador de entidades que tenemos de apartados anteriores, para ello reharemos desde cero el "EntityManager" implementado en otros proyectos y le añadiremos lo que necesitemos para este.

Comenzaremos definiéndolo, con un vector de entidades, donde se guardarán todas y cada una de ellas y el objeto "Component Storage" que a partir de ahora lo llamaremos "CS".

Manejador de entidades

```
struct EntityManager{
    EntityManager(std::size_t size_for_entities = 10){
        entities.reserve(size_for_entities);
    }
    private:
    std::vector<Entity>     entities;
    ComponentStorage CS{};
};
```

El constructor recibe al crear el "EntityManager" un número que será el número de entidades a reservar, y si no recibe nada será 10 por ejemplo.

Seguiremos creando una función básica, crear una entidad. Para crear una entidad, lo único que tendremos que hacer es añadirla al vector de entidades. Tan sencillo como eso. Por lo tanto añadamos la siguiente función pública, que nos devolverá dicha entidad para añadirle componentes o tags, en el momento de su creación.

Crear una entidad

```
struct EntityManager{
    public:
    auto& createEntity(){ return entities.emplace_back();}

    ...

};
```

Ya podemos crear una entidad, pero igual que creamos, destruimos, por lo tanto necesitaremos un método para eliminar entidades, lo que supone eliminar también todos los componentes que tenga.

Para ello haremos una función que elimine todos los componentes de una entidad, que consistirá en eliminar todos los que tenemos. Si alguno de los componentes que intentamos eliminar no lo tiene la entidad, no pasará nada, ya que antes de eliminar un componente, comprobamos si existe en el método del CS.

Eliminar todos los componentes de una entidad

```
struct EntityManager{
    public:
    void removeAllComponents(Entity& e){
        CS.removeInputCMP(e);
        CS.removePositionCMP(e);
        CS.removeRenderCMP(e);
        CS.removeStatsCMP(e);
        CS.removeObjectCMP(e);
        CS.removeDescriptionCMP(e);
    }

    ...

};
```

Ahora podemos eliminar una entidad, ya que podemos destruir el espacio que tiene reservado. Para ello, llamaremos a la función anterior y, al igual que borrábamos una entidad en proyectos anteriores, ahora la eliminaremos obteniendo su id, usando "std::remove_if". Y usando el "==" de las entidades, sobrecargado anteriormente.

Eliminar una entidad

```
struct EntityManager{
    public:
    void removeEntity(Entity& e) {
        //Elimina todos los componentes de una entidad
        removeAllComponents(e);
        //Eliminar entidad
        auto it = std::remove_if(entities.begin(), entities.end(),
                    [e](Entity &ent) {
                        return e == ent;
                    });
        entities.erase(it, entities.end());
    }

    ...

};
```

De esta misma forma podremos eliminar a todas las entidades recorriendo el vector de entidades y llamando a esta función para que todo quede limpio de forma correcta y eliminar todos los componentes.

Ahora que podemos crear entidades, pero también eliminarlas, necesitaremos métodos que las gestionen, y para eso crearemos los métodos "forall" y "forallMatching", que ejecutarán una función para cada entidad seleccionada.

El método forall lo hemos implementado en anteriores proyectos y en este funcionará de la misma forma. Recibe una referencia a una función que ejecuta para todas y cada una de las entidades del vector.

Método "forall"

```
struct EntityManager{
    public:
    void forall(auto&& function){
        for(auto&e:entities){
            function(e);
        }
    }

    ...

};
```

A continuación implementaremos una gran novedad, y es que este "forall" que tenemos implementado, nos sirve si queremos ejecutar algo para todas las entidades del juego. Sin embargo podemos parametrizar este forall, para que además de recorrer el vector de entidades y ejecutar una función para cada una, decida a que entidades aplicarles dicha función. Esto es posible gracias a las máscaras de la entidad, tanto de componentes como de etiquetas. Crearemos una comprobación entre las máscaras de la entidad con una pasada por parámetro que decidirá el sistema que use este método.

El sistema de render necesita entidades que tengan componente de render. En juegos anteriores, todas las entidades podían tener componente de render pero ahora es diferente, pueden haber entidades que no necesiten de ese componente y por lo tanto no ejecutarle la función de renderizado.

Una vez hechas las comprobaciones ejecutaremos la función de la misma forma que en el método previo, pero así nos aseguramos también de que todas las entidades que se ejecuten cumplan un requisito. Esto nos abre muchas puertas a la hora de hacer un juego y es uno de los factores más importantes de este. Además optimiza operaciones y facilita el trabajo para el usuario o programador que la use.

A continuación implementaremos el "forallMatching" y para ver su puesta en marcha habrá que esperar a los sistemas, que veremos proximamente.

Método "forallMatching"

```
struct EntityManager{
    public:
    void forallMatching(auto&& function, int systemCMPmask,
                        int systemTAGmask){
        for(auto&e:entities){
            if(((e.cmpMask & systemCMPmask) == systemCMPmask) &&
               ((e.tagMask & systemTAGmask) == systemTAGmask)){
                function(e);
            }
        }
    }

    ...

};
```

Con esto el manejador de entidades está completo, a gusto del programador se pueden incluir otras funciones necesarias como algun get para el vector de entidades, o un "getPlayer()" que nos será muy útil al no tener patrón singleton implementado, y que hara que podamos recoger la entidad que pertenezca al jugador en cualquier momento.

El método get que sí vamos a necesitar es el que nos devuelve al "Component Storage" que lo necesitaremos para crear y hacer todas las operaciones pertinentes. Otra opción es implementar un "addCMP", "removeCMP" y "getCMP" en el manejador de entidades, pero tendriamos que rehacer uno de cada para cada componente, ya que aun no hemos aprendido como mejorar esa parte. En futuros proyectos veremos como hacerlo. Por lo tanto, lo más inteligente es crear un método que nos devuelva nuestro "CS".

Get Component Storage

```
struct EntityManager{
    public:
    ComponentStorage& getCMPStorage(){
        return CS;
    }

    ...

};
```

Ya tenemos todo lo necesario para comenzar a crear nuestro juego, y lo haremos en el siguiente capítulo. El diagrama de la figura 33 queda implementado.

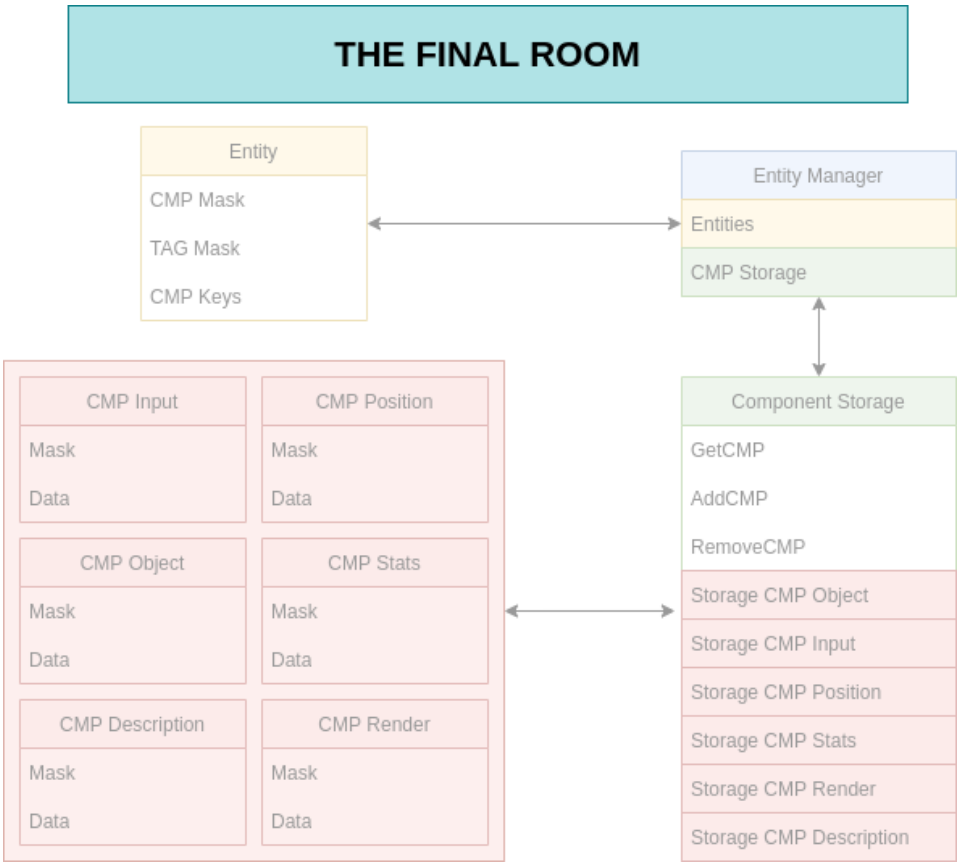


Figura 33. Diagrama de estructura de datos de "The Final Room" actual

7 Implementamos The Final Room

Hemos creado un motor completo, tenemos forma de crear entidades, tenemos forma de destruirlas, tenemos forma de crear componentes y añadirlos a las entidades. Podemos destruirlos también. Manejamos tanto entidades como componentes y tenemos métodos para seleccionar cuales queremos modificar de entre todos los existentes.

El juego que vamos a crear se propone en el apartado 6.1, ya que hemos tenido que explicarlo anteriormente para el desarrollo de estructuras de almacenamiento y de los componentes de los que disponemos.

El diagrama 34, es el diagrama de la estructura de almacenamiento de datos que tiene nuestro motor en este momento, que hemos implementado en el apartado 6.2, y vamos a basarnos en esta estructura para desarrollar las mecánicas y sistemas de nuestro proyecto.

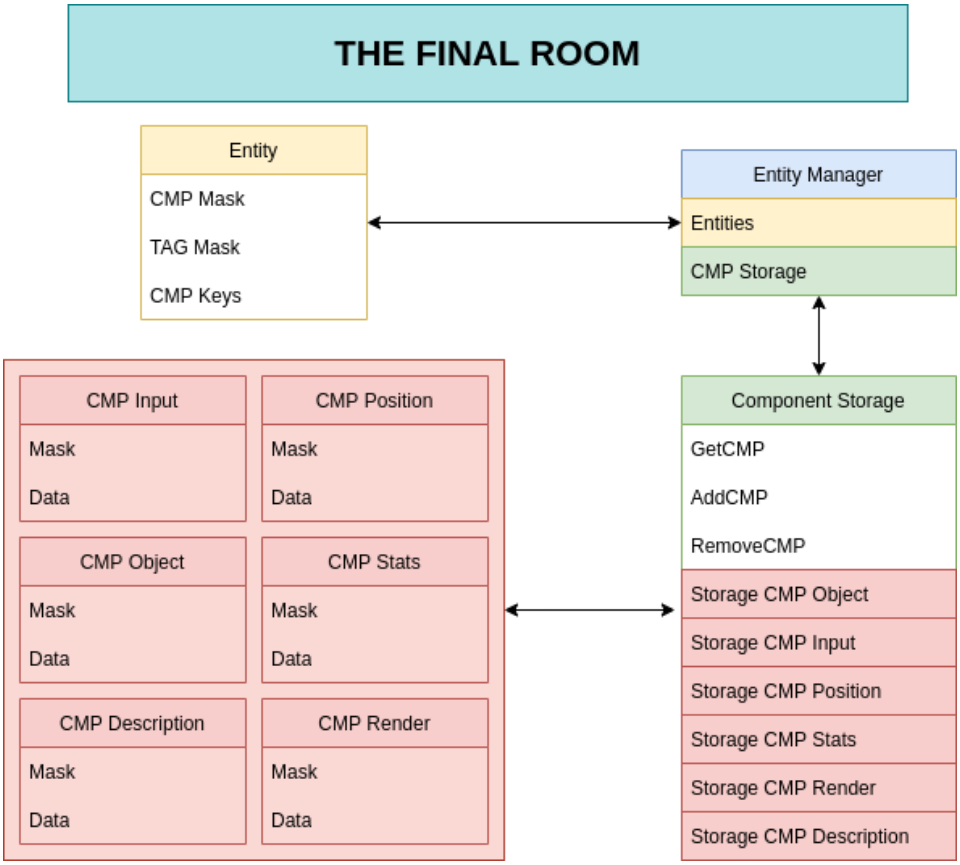


Figura 34. Estructura de datos de nuestro juego.

En primer lugar, crearemos una ventana, donde colocar nuestros objetos y manejadores para comenzar teniendo al menos una pantalla vacía. Para ello vamos a comenzar a implementar la clase "Game" de nuestro juego, aunque la iremos construyendo a medida que avancemos.

Tendremos una clase donde crearemos un manejador de entidades y un método que ejecute el bucle del juego. De esta forma tendremos una primera ventana, aunque nada implementado.

Game inicial

```
struct Game{
    EntityManager EM{100};
    bool  playing_lvl   = false;
    //Métodos
    void run();
};
```

Game "run()"

```
void Game::run(){
    while (playing_lvl){
        BeginDrawing();
        ClearBackground(RAYWHITE);
        //pintar
        EndDrawing();
    }
}
```

Usaremos una variable para decidir cuando el juego finaliza, mientras que "playing_lvl" sea verdadera, el juego seguirá su curso. De esta forma podremos modificar este valor al morir, pasándolo por referencia, y más adelante, con la llegada de los menús, podremos controlar el cierre del juego sencillamente.

A partir de aquí, seguiremos una rutina. Hablaremos de que es lo que queremos conseguir, y seguidamente lo programaremos, usando el motor que tenemos, y RayLib como apoyo para el dibujado.

7.1 Sistema de renderizado y Mapa de sprites

Vamos a avanzar definiendo nuestro primer sistema, el sistema de renderizado, para ello tendremos que recorrer todas las entidades que haya creadas y ejecutar el sistema solo para aquellas que dispongan de por ejemplo, componente de render y de posición, ya que ambos son necesarios.

Antes de esta implementación, hagamos una pequeña pausa, porque no todo es renderizar las entidades. Crearemos una interfaz gráfica para el juego donde estarán elementos como los objetos que llevamos en ese momento o las estadísticas del jugador en cada momento. También pintaremos el mapa de juego.

Vamos a crear una clase "Map" que represente todos estos valores, y simplemente tengamos una instancia con los sprites pertinentes para dibujarlos manualmente. Conseguiremos así la interfaz para el jugador. Antes de pasar a programar esta estructura de sprites, observaremos la idea de interfaz que propongo.

Esta interfaz de la figura 35 representa el tablero del juego, dejando espacio en la izquierda para las estadísticas del jugador y en la derecha para los objetos del momento. Además tendremos que representar el número de rondas que llevamos.

Para esto tenemos un sprite que representa a la perfección esta interfaz, figura 27, pero tendremos que detallar cómo crearlo y cuándo mostrarlo.

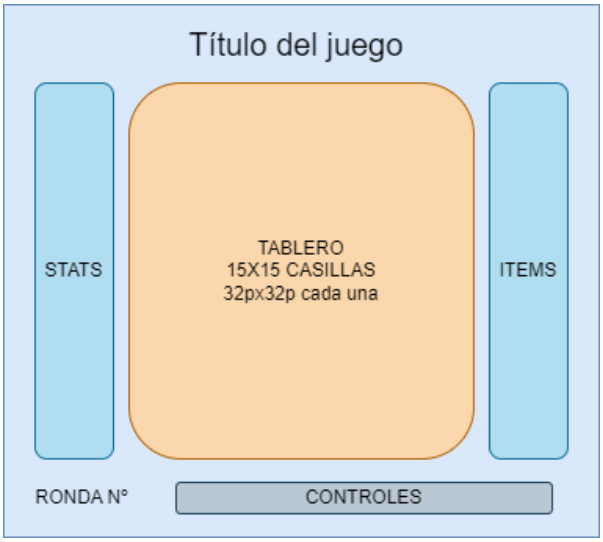


Figura 35. Interfaz del juego

Para ello crearemos una estructura interfaz que llamaremos "Map" como hemos dicho anteriormente. En ella tendremos dos objetos de otras dos estructuras extra llamadas "Menu" e "Interface" y en ellas recogeremos todos los sprites a mostrar. En la clase "Map" tendremos un constructor para inicializar todos esos sprites y un destructor para borrar todas las texturas para evitar problemas de memoria.

Esta clase es externa a nuestro motor de entidades, así que no la voy a mostrar. Aunque recuerdo, que solo contiene los sprites de la interfaz, el menú que posteriormente crearemos y de las diferentes estadísticas que existen, los diferentes sprites para representar todo en la interfaz del juego.

Una vez tenemos la clase "Map" lista con todos los sprites de HUD o interfaz, comenzaremos a crearemos una instancia de esta en el mismo "Game", ya que solo la necesitamos mientras jugamos.

Después de esto proseguiremos con el "render system" el cual nos llevará un poco de trabajo gráfico.

Creamos la estructura del sistema, que todos los sistemas seguirán como patrón, para que nos sea más fácil de visualizar y modificar. Crearemos una estructura con el típico método "update()", donde llamaremos a la función que hemos creado en el "Entity Manager", "forallMatching" y mediante un método lambda, como en ejemplos anteriores, ejecutaremos el código que queremos que las entidades que dispongan de render reproduzcan.

Render System (.h)

```
struct RenderSystem{
    void update(EntityManager& EM, Map& map);
    int cmpMaskToCheck = RenderCMP::mask | PositionCMP::mask;
    int tagMaskToCheck = 0;
}
```

Definimos las máscaras que utilizaremos, como hemos mencionado anteriormente la de render y posición. De esta forma lo que ejecutemos en el "forallMatching" de "update" solo se ejecutará para las entidades que dispongan de ambos componentes.

Los componentes de posición serán valores entre 0 y 14, representando las casillas del tablero, por lo tanto necesitamos unas constantes que definan el tamaño de este, para saber la posición (0,0) del tablero a que píxel correspondería.

Para ello, he implementado las siguientes constantes "define" para representar esas medidas:

Render System (.h): constantes

```
#define HORIZONTAL_BORDER 160
#define HORIZONTAL_MIDDLE 480
#define VERTICAL_BORDER 64
#define VERTICAL_MIDDLE 480
#define SPRITE_DIMENSIONS 32
```

Con ellas podemos implementar de forma correcta el método update de render:

Render System (.cpp): update

```
void RenderSystem::update(EntityManager& EM, GameManager& GM,
                                                    Map& map){
    EM.forallMatching([&](Entity&e){
        auto& render    = EM.getCMPStorage().getRenderCMP(e);
        auto& pos       = EM.getCMPStorage().getPositionCMP(e);
        float pos_X= HORIZONTAL_BORDER + pos.posX * SPRITE_DIMENSIONS;
        float pos_Y= VERTICAL_BORDER + pos.posY * SPRITE_DIMENSIONS;
        DrawTextureRec(render.sprite, render.frame,
                       (Vector2){pos_X, pos_Y},WHITE);
    }, cmpMaskToCheck, tagMaskToCheck);
}
```

Recogemos la posición y el render del slotmap, llamando al "component storage" y recogiendo cada componente. Posteriormente pintamos el sprite guardado en el componente

de render de la entidad en la posición que marca el componente de posición.

Hemos pintado todas las entidades que tenemos en el vector con esas características, pero no hemos pintado el mapa. Para pintar el mapa, necesitaremos ejecutar el dibujado antes del método "forallMatching" para que solo ocurra una sola vez y antes de pintar las entidades, si no, pintaremos encima de las entidades y tendremos problemas.

Para ello, crearemos una función "renderMap" que recibiendo el mapa, lo dibuje en la posición (0,0) de la pantalla.

Render System (.cpp): renderMap

```
void RenderSystem::renderMap(Map& map){
    DrawTexture(map.map,0,0,WHITE);
}
```

Esta función la llamaremos antes del "forallMatching" como bien hemos comentado anteriormente.

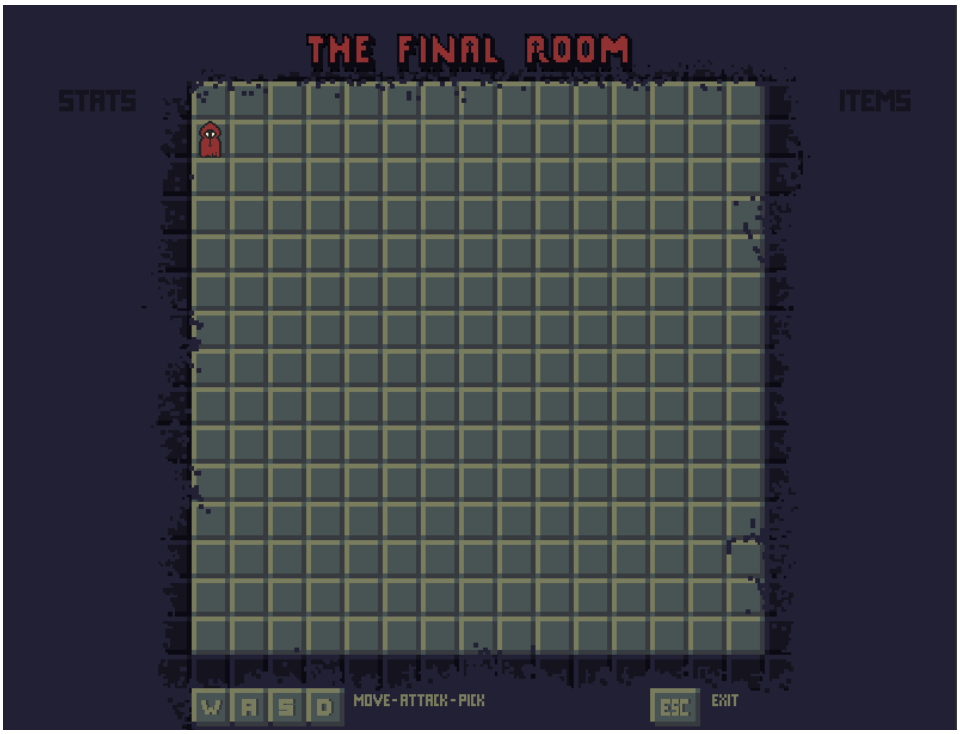


Figura 36. Esta figura representa lo que llevamos hasta ahora visualmente

En la figura 36, podemos apreciar el mapa vacío, simplemente con la única entidad que he creado, a la que he asociado un "sprite" para el jugador principal.

Empezamos a ver resultados, y a continuación como tenemos la entidad jugador, queremos saber las estadísticas del mismo, y las queremos ver a la izquierda. La forma de

colocarlas es personal, yo las colocaré individualmente en vertical, con un "sprite" de puntos en horizontal, que mostrarán la cantidad.

Esta función pintará usando "Map.interface", que es donde están almacenados todos los sprites de las estadísticas. Usamos las dimensiones constantes que hemos definido anteriormente, y pintamos un sprite encima de otro.

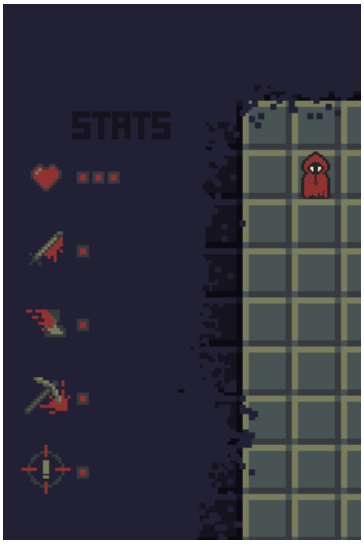


Figura 37. Interfaz del juego

Lo más importante de este método es comprobar si hay entidades creadas, y en concreto, el jugador o "player", ya que si no existe, estaremos accediendo donde no debemos y esto no puede ser. Después de comprobar esto, recorremos por ejemplo la vida, si tenemos 3 de vida pintamos 3 puntitos. Pero si tenemos 2 de vida y 1 de vida máxima, pintaremos 2 puntitos rojos y uno negro (otro sprite diferente, que representa que te han quitado vida).

No es necesario ver la implementación, pues simplemente es recorrer este componente del "player" e ir pintando puntos donde toque. Este sería el resultado, en la figura 37, después de añadir esta función a "update".

Para dibujar los objetos en la otra parte de la pantalla, crearemos una función concretamente para eso, donde ejecutaremos otro forallMatching para revisar que objetos hay creados, pasándole una etiqueta o "tag" definida en la estructura "Tags", la llamaremos "object".

De esta forma renderizaremos todos los objetos apilados a la derecha de la pantalla.

Figura 38. Objetos renderizados

Todo esto está creado usando RayLib y no es el objetivo de este libro, por lo tanto es el motivo de el avance rápido en el aspecto gráfico. Aun así, iré mostrando los resultados para poder observar los avances. En la figura 38, se aprecia la representación de objetos, una vez los tenemos en el inventario.

Con esto, completamos por ahora el sistema de renderizado, aunque habrá más cambios próximamente, a medida que implementemos menús y otros parámetros como los niveles o rondas.

7.2 Game Manager: Flujo del juego

Ahora que tenemos un primer sistema, vamos a crear un manejador para nuestro juego, ya que la creación de entidades, componentes, etc, lo implementaremos en esta clase, que tendrá como función principal, manejar los bloques de entidades que se crean y destruyen para cada nivel o ronda.

Esta clase tendrá diferentes parámetros del juego, como por ejemplo, la ronda actual, la matriz actualizada del nivel actual, las diferentes matrices de niveles implementados manualmente, y otros elementos como una matriz para niveles aleatorios y el manejo del inventario del juego.

Comenzaremos por las funciones básicas de creación de entidades. Los valores de cada parámetro, dependerán de las características que querramos que tenga el juego.

En primer lugar, crearemos el jugador. Que por defecto quedará de la siguiente forma:

Game Manager: createPlayer

```
void createPlayer(EntityManager& EM, int x, int y){
   auto& player = EM.createEntity();
   player.addTag(Tags::player | Tags::movement |
                 Tags::collider | Tags::collisionable);
   EM.getCMPStorage().addRenderCMP(RenderCMP{"player.png"},player);
   EM.getCMPStorage().addPositionCMP(PositionCMP{x,y}, player);
   EM.getCMPStorage().addInputCMP(InputCMP{}, player);
   EM.getCMPStorage().addStatsCMP(StatsCMP{3,3,1,1,5,1}, player);
   player_alive=true;
}
bool player_alive=false;
```

Además de crearlo, usaremos una variable booleana para saber si el player esta creado en cualquier momento, para saber si podemos realizar diferentes acciones posteriormente.

Seguiremos creando enemigos, y en este caso, crearemos enemigos de dos tipos. En primer lugar, unos fantasmas que tendrán 1 unidad de vida y lo mismo de daño y otro tipo de enemigos que serán unos soldados, con 2 de vida y 2 de daño. Podemos hacer tantos enemigos como queramos en esta función y usando tags, programarles diferentes comportamientos a cada uno de ellos.

Los enemigos se crearán de la siguiente forma, recibiendo por parámetro el tipo, que será o 3 o 4, respectivamente. Si es 3 aplicamos a sus componentes los elementos del fantasma y si es 4 los del soldado.

Game Manager: createEnemy

```
Entity& createEnemy(EntityManager& EM, int type){
    auto& enemy = EM.createEntity();
    enemy.addTag(Tags::collider | Tags::collisionable | Tags::enemy |
                                                        Tags::movement);
    EM.getCMPStorage().addPositionCMP(PositionCMP{0,0,0,0}, enemy);
    switch (type)
    {
    case 3: // FANTASMA
        enemy.addTag(Tags::ghost);
        EM.getCMPStorage().addRenderCMP(RenderCMP{"enemigo_1.png"},
        enemy);
        EM.getCMPStorage().addStatsCMP(StatsCMP{1,1,1,1,1,0}, enemy);
        break;
    case 4: // SOLDADO
        enemy.addTag(Tags::soldier);
        EM.getCMPStorage().addRenderCMP(RenderCMP{"enemigo_2.png"},
        enemy);
        EM.getCMPStorage().addStatsCMP(StatsCMP{2,2,2,1,1,0}, enemy);
        break;
    default:
        break;
    }
    return enemy;
}
```

Podemos fijarnos en los tags que tiene cada enemigo o cada jugador, ya que hay nuevos, en concreto lo siguientes:

- Collider: Lo usaremos cuando una entidad tenga colisiones y sea la que las detecta.
- Collisionable: Lo tendrán las entidades que sean colisionables, y que habrá que tener en cuenta.
- Movement: Lo tendrán todas las entidades que se muevan. De esta forma podemos hacer que las propiedades de movimiento se apliquen a jugador, enemigos e incluso a algún que otro obstáculo.

Los tags enemy y player, ya los conocemos.

A continuación, seguiremos creando los diferentes objetos, y crearemos tanto la llave, como la puerta. Estas dos entidades servirán para acceder al siguiente nivel. Si tenemos la llave, se abrirá la puerta. Estos dos tipos de entidades se crearán con sus respectivas funciones.

Game Manager: createKey and createDoor

```
Entity& createKey(EntityManager& EM, int x, int y){
    auto& key = EM.createEntity();
    key.addTag(Tags::collisionable | Tags::key);
    EM.getCMPStorage().addRenderCMP(RenderCMP{"llave.png"}, key);
    EM.getCMPStorage().addPositionCMP(PositionCMP{y,x,0,0}, key);
    return key;
}
Entity& createDoor(EntityManager& EM, int x, int y){
    auto& door = EM.createEntity();
    door.addTag(Tags::collisionable | Tags::door);
    EM.getCMPStorage().addRenderCMP(RenderCMP{"puerta.png"}, door);
    EM.getCMPStorage().addPositionCMP(PositionCMP{y,x,0,0}, door);
    return door;
}
```

También podemos crear un cofre, que contendrá un objeto aleatorio. Para crear este cofre tendremos que añadirle el "tag colisionable", y un nuevo "tag" llamaod "chest".

Game Manager: createChest

```
Entity& createChest(EntityManager& EM, int x, int y){
    auto& chest = EM.createEntity();
    chest.addTag(Tags::chest | Tags::collisionable);
    EM.getCMPStorage().addRenderCMP(RenderCMP{"chest.png"}, chest);
    EM.getCMPStorage().addPositionCMP(PositionCMP{y,x,0,0}, chest);
    return chest;
}
```

El último tipo de entidad a crear serán los objetos, de los que hay cierta variedad de tipos. Cada tipo incrementa una estadística. Cada tipo tiene el mismo sprite que la estadística que incrementa. Para crear un objeto, tendremos que pasarle el tipo de objeto a la función de creación de objetos y esta se encargará de configurar cada componente.

Game Manager: Creamos diferentes objetos

```
Entity& createObject(EntityManager& EM, int obj){
  Entity& entity = EM.createEntity();
  entity.addTag(Tags::object);

  switch (obj)
  {
  case Objects::Health :
   EM.getCMPStorage().addRenderCMP(RenderCMP{"vida.png"}, entity);
```

```
    EM.getCMPStorage().addStatsCMP(StatsCMP{1,1,0,0,0,0}, entity);
    EM.getCMPStorage().addObjectCMP(ObjectCMP{Objects::Health},entity);
    break;
  case Objects::Damage :
    EM.getCMPStorage().addRenderCMP(RenderCMP{"ataque.png"}, entity);
    EM.getCMPStorage().addStatsCMP(StatsCMP{0,0,1,0,0,0}, entity);
    EM.getCMPStorage().addObjectCMP(ObjectCMP{Objects::Damage},entity);
    break;

    ...

  default:
      break;
  }
  return entity;
}
```

Hay más tipos de objetos que no se muestran por la longitud del código, se pueden observar en el código completo, pero es simple. Habrá un objeto por cada estadística definida. Aquí hay una lista de los objetos que existen, la cual usamos para definir los tipos de objeto.

Game Manager: Tipos de objeto

```
enum Objects{
    Health,
    Damage,
    Steps,
    Critical,
    Pickaxe
};
```

Con todas estas entidades diferentes, podemos crear un juego, a falta de crear obstáculos. Vamos a crear un primer nivel. Para ello, haremos lo siguiente:

- Crearemos un "enum" con los diferentes niveles,
- Crearemos la matriz de forma manual del juego.
  - Si el valor es 0, no hay nada en esa casilla.
  - Si el valor es 1, habrá un obstáculo de tipo 1.
  - Si el valor es 2, habrá un obstáculo de tipo 2.
  - Si el valor es 3, habrá un enemigo de tipo 3.
  - Si el valor es 4, habrá un enemigo de tipo 4.
  - Si el valor es 5, habrá un cofre.

  - Si el valor es 6, habrá una llave.
  - Si el valor es 7, habrá una puerta.
- Después de crear esa matriz, la tendremos que convertir a entidades. Y las posiciones de esas entidades, serán las del componente de posición.
- Por último guardaremos la matriz cargada actualmente en otra matriz de 15x15, como matriz actual, la cual iremos actualizando, para que la IA de los enemigos detecte todos los obstáculos de forma correcta.

Por lo tanto vamos a crear la primera matriz, por ejemplo la siguiente, que será la matriz de nivel 0. Añadiremos al "enum" el nivel 0.

Game Manager: Lvl0 Matriz

```
const int map_lvl0[15][15]{{0,0,0,0,0,0,0,2,2,2,2,0,0,0,0},
                           {1,2,1,1,0,0,0,0,0,0,0,0,0,0,0},
                           {1,3,0,0,0,0,0,0,0,0,2,2,0,0,0},
                           {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
                           {1,0,1,2,2,2,0,0,0,0,0,0,1,2,6},
                           {1,0,1,0,0,0,0,0,0,0,0,0,0,0,1},
                           {0,0,0,0,0,0,0,0,1,1,1,0,1,2,3},
                           {0,0,0,0,0,2,2,2,2,0,0,0,0,0,0},
                           {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
                           {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
                           {1,1,1,0,0,1,2,2,1,0,0,0,0,0,0},
                           {1,2,2,0,0,1,2,1,1,0,0,0,0,0,0},
                           {0,1,0,0,0,0,0,0,0,0,0,0,2,2,0},
                           {0,0,0,0,0,0,0,0,0,0,0,0,2,7,0},
                           {1,1,1,0,0,1,1,1,0,0,1,1,1,0,0}};
```

Ya tenemos un nivel preparado, pero nuestro juego tiene que interpretarlo. Crearemos un generador de niveles. Este generador, recibe por parámetro la matriz a representar y para cada posición llama a una función, dependiendo el número de esa posición. Si es un 5 llamará a la función para crear un cofre y además guardará un 5 en la matriz actual, en la misma posición. Esta función es un poco extensa por los diferentes casos que existen.

Además de esto, si es 1 o 2 crearemos un obstáculo, que hasta ahora no lo hemos implementado. Simplemente creando una entidad y añadiéndole componentes como render (tendrá forma de piedra), posición, y estadísticas, teniendo 4 de vida, lo que significa que serán destruibles. La idea es hacerlos de diferentes cantidades de vida, para tener destruibles diferentes.

Game Manager: Generar un nivel

```
void generateLvl(EntityManager& EM, const int matrix_lvl[15][15]){
```

```
    for (int i = 0; i < 15;i++){
        for(int j = 0; j < 15; j++){
            if (matrix_lvl[i][j] == 1){
                auto& rock_wall = EM.createEntity();
                rock_wall.addTag(Tags::wall | Tags::collisionable);
                EM.getCMPStorage().addRenderCMP(
                            RenderCMP{"piedra_2-sheet.png"},rock_wall);
                EM.getCMPStorage().addPositionCMP(
                            PositionCMP{j,i,0,0}, rock_wall);
                EM.getCMPStorage().addStatsCMP(
                            StatsCMP{4,4,0,0,0,0}, rock_wall);
                actual_lvl[i][j]=1;
            }else if ( ... )

        ...

        }
    }
    chargeInventory(EM);
}
```

Este método es mucho más largo, pero hacemos lo mismo. Para el valor 2 se crea otro obstáculo similar, para el 3 un enemigo con el tipo 3, etc. De esta forma conseguimos crear todas las entidades de un nivel y una matriz actual que podremos actualizar.

A continuación subiremos de escalón y crearemos otra función. Esta vez para elegir que nivel crear de los que disponemos. Para ello, crearemos otro nivel, añadimos al "enum", creamos su matriz e implementamos el siguiente método.

Game Manager: Elegimos el nivel a generar

```
void selectLvlAndGenerate(EntityManager& EM, int lvl){
    clearMatrix();
    switch (lvl){
    case LvlNumber::Lvl0:
        generateLvl(EM, map_lvl0);
        break;
    case LvlNumber::Lvl1:
        generateLvl(EM, map_lvl1);
        break;
    case LvlNumber::Random:
        generateRandomMatrix(lvl);
        generateLvl(EM, random_lvl);
        break;
```

```
    default:
        generateRandomMatrix(lvl);
        generateLvl(EM, random_lvl);
        break;
    }
}
```

Esta función tiene elementos nuevos, y los vamos a explicar sin entrar mucho en detalle. Este método recibe un nivel, y con el valor del "enum" que coincida, generará el nivel correspondiente. Si es "random" o mayor, crearemos un nivel "random" con la función "generateRandomMatrix(int)" con el número del nivel como parámetro. Esta función guarda en la matriz "random_lvl" valores del 0 al 7, con un criterio. Dependiendo del nivel, más enemigos, más obstáculos, y menos cofres. Siempre se generará una puerta y una llave.

No mostraré el código de esta función pero es fácil de implementar. Lo podéis consultar en el código del juego que está adjunto. Solo queda un detalle importante, y es la función "clearMatrix()" del principio del generador de niveles. Esta función limpia la matriz actual y la aleatoria, para que se puedan crear nuevos niveles desde cero en cada iteración o ronda.

Antes de terminar con esta clase, nos ocuparemos del "player". Con esto quiero decir que tendremos que hacer una función que ponga a "false" la variable que permite saber si está vivo el "player". Para ello simplemente creamos una función que mate al "player". Yo la he llamado "killPlayer()".

Tenemos que manejar también el inventario del jugador, ya que de eso dependen sus estadísticas. Si borramos todas las entidades al pasar de ronda, tendremos que tener apuntados que componentes tenía el jugador antes del cambio de nivel. Para eso tenemos tres funciones y un vector. El vector "inventory" que es un vector de enteros que almacenará el tipo del objeto, que lo podemos recoger del "ObjectCMP".

- La primera función es "clearInventory()". Esta función simplemente limpia el inventario y elimina los objetos.

  Game Manager: Manejo de inventario

  ```
  void clearInventory(){
      inventory.clear();
  }
  private:
  std::vector<int> inventory{};
  ```

- La segunda es "saveInventory()". Esta función repasará las entidades existentes y guardará todas las que sean objetos, pero no guardará la entidad, si no el tipo del objeto.

Game Manager: Manejo de inventario

```
void saveInventory(EntityManager& EM){
    clearInventory();
    EM.forallMatching([&](Entity& e){
        int type_of_cmp = EM.getCMPStorage().getObjectCMP(e).obj;
        inventory.push_back(type_of_cmp);
    }, ObjectCMP::mask, Tags::object);
}
```

- Por último tenemos la función "chargeInventory()". Esta recoge el vector y crea las entidades "objeto", según los tipos que hay guardados en el inventario.

Game Manager: Manejo de inventario

```
void chargeInventory(EntityManager& EM){
    for(std::size_t i = 0; i < inventory.size(); i++){
        auto& player = EM.getPlayer();
        auto& player_stats=EM.getCMPStorage().getStatsCMP(player);
        auto& obj = createObject(EM, inventory[i]);
        auto& obj_stats = EM.getCMPStorage().getStatsCMP(obj);
        player_stats.addStats(obj_stats);
    }
}
```

Por último podemos crear funciones auxiliares para recoger o modificar las variables del juego, por ejemplo para recoger la matriz o mapa actual, o para aumentar el nivel o resetearlo.

Game Manager: Gets y sets

```
auto& getActualMap(){
    return actual_lvl;
}
int getActualLvl(){
    return num_lvl_actual;
}
void nextLvl(){
    num_lvl_actual++;
}
void resetLvl(){
    num_lvl_actual = 0;
}
```

Esta es la clase "Game Manager", que como has podido observar su utilidad es elevada, ya que se encarga de dirigir que entidades crear y en que posiciones.

Teniendo esta clase en funcionamiento, podemos seguir implementando los diferentes sistemas, pero antes de avanzar con el de posicionamiento, vamos a añadir un elemento extra al de render. Este elemento será la ronda actual. Para ello tendremos que pasar por parámetro una referencia del "Game Manager", cuya localización es la clase "Game". Una vez en el "update" de render, habrá que llamar a una función que renderice texto, para dibujar el número de rondas actuales. Esa función es la siguiente:

Render System: Ronda actual con Game Manager

```
void RenderSystem::renderRound(GameManager& GM){
    std::ostringstream stream;
    stream << "Round: "<< GM.getActualLvl()+1;
    std::string str = stream.str();
    const char* text = str.c_str();
    Color color{20,19,30,255};
    Font font = GetFontDefault();
    font.baseSize *= 2.0f; // aumentar el tamaño en un 100%
    DrawTextEx(font, text, Vector2{15.0f, 576.0f}, 50, 1, color);
}
```

Simplemente usamos RayLib para pintar el texto, recogiendo la variable de nivel actual.

7.3 Sistema de posición o físicas

El sistema de posición es muy simple, contiene lo siguiente:

- Máscara de componentes y tags requeridos.
  - Componentes: PositionCMP
  - Tags: Movement
- Método "update" para actualizar las posiciones de las entidades.

No necesitaremos más. Comencemos creando la función "update".

Esta función ejecuta el siguiente flujo. En primer lugar obtendremos una referencia a la matriz del mapa actual y el componente de posición de la entidad actual. Para el jugador, comprobaremos si la entidad tiene el "tag player". Como vamos a intentar movernos, la posición en la matriz del nivel que pertenece al "player" pasa a tener el valor 0. Después de este cambio intentamos movernos. Para ello comprobamos si la velocidad en x es 0 o la y es 0. En el caso de que la x no sea 0, significa que queremos movernos en horizontal y antes de ejecutar el movimiento, comprobamos si el valor de la posición del mapa actual a la que nos queremos mover es 0 y que esté contenida dentro del tablero. Si es así nos movemos y la posición cambia. Lo mismo ocurre para el eje Y.

Para los enemigos (si la entidad no tiene el tag "player"), colocamos un 3 en la posición del mapa donde está.

Como último paso, la velocidad de cada entidad que recibe el sistema es 0 en el eje horizontal y 0 en el eje vertical.

Position System: update

```
void PositionSystem::update(EntityManager& EM, GameManager& GM){
    EM.forallMatching([&](Entity& e){
        auto& map = GM.getActualMap();
        auto& posCMP = EM.getCMPStorage().getPositionCMP(e);
        GM.getActualMap()[posCMP.posY][posCMP.posX] = 0;
        if(e.hasTag(Tags::player)){
            int directionX=0;
            if(posCMP.velX<0){ directionX= -1; }else{ directionX = 1;}
            int directionY=0;
            if(posCMP.velY<0){ directionY= -1; }else{ directionY = 1;}

            if(posCMP.velX!=0){
                for (int i=0; i<abs(posCMP.velX); i++){
                    if(map[posCMP.posY][posCMP.posX+directionX]==0 &&
                    posCMP.posX+directionX >=0 &&
                    posCMP.posX+directionX<15){
                        posCMP.posX += directionX;
                    }else{
                        break;
                    }
                }
            }else if(posCMP.velY!=0){
                for (int i=0; i<abs(posCMP.velY); i++){
                    if(map[posCMP.posY+directionY][posCMP.posX]==0 &&
                    posCMP.posY+directionY >=0 &&
                    posCMP.posY+directionY<15){
                        posCMP.posY += directionY;
                    }else{
                        break;
                    }
                }
            }
        }else{
            posCMP.posX += posCMP.velX;
            posCMP.posY += posCMP.velY;
        }
        posCMP.velX = posCMP.velY = 0;

        //Matriz del mapa actualizada
```

```
        if(e.hasTag(Tags::enemy)){
            GM.getActualMap()[posCMP.posY][posCMP.posX] = 3;
        }
    }, cmpMaskToCheck, tagMaskToCheck);
}
```

7.4 Sistema de Input o movimiento

El sistema de input sirve para que mediante pulsaciones en el teclado, se ejecuten acciones en el juego. Las únicas teclas que necesitaremos serán "W", "A", "S", "D" y "Escape". Estas teclas están definidas en el componente de input. Por lo tanto en el sistema tendremos que recuperar las teclas y ejecutar alguna acción con ellas.

En primer lugar, crearemos una función para comprobar si hay teclas pulsadas, y dependiendo de cual sea cambiaremos la velocidad a una u otra, dependiendo de la dirección y la cantidad de velocidad que tengamos en el componente de estadísticas. Por lo tanto, recogemos una referencia a los componentes de input, posición y estadísticas.

Esta función recibe por parámetro lo siguiente:

- El manejador de entidades, ya que se usa para recoger los componentes
- La entidad que contiene "InputCMP", el turno que aún no lo hemos implementado, pero es una variable de "Game" que se encarga de decidir de quien es el turno. Primero mueve el jugador, y luego es el turno de los enemigos. Lo veremos próximamente, pero si pulsamos una tecla, significa que el turno ya no nos pertenece, ya que habremos ejecutado una acción.
- La variable "playing_lvl" que de pulsar ESC, pasará a ser falsa y por lo tanto el juego terminará.
- Recibe un estado que aún no hemos implementado tampoco. Pero simplemente cambiará esta variable al salir del juego al valor que tendrá el estado menú. No hace falta que esté implementado todavía, pero es bueno ir teniendo en cuenta donde cambiaríamos de pantalla.

Vamos a proceder a programa esta función:

Input System: keyPressed

```
void InputSystem::keyPressed(EntityManager& EM,Entity& e,
            bool& turn, bool& playing_lvl, States& state){
    auto& input = EM.getCMPStorage().getInputCMP(e);
    auto& pos   = EM.getCMPStorage().getPositionCMP(e);
    auto& stats = EM.getCMPStorage().getStatsCMP(e);
    if ((IsKeyDown(input.KeyW) && pos.posY > 0) ){
        pos.velY = -stats.step;
        turn = false;
```

```
    }else if ((IsKeyDown(input.KeyA) && pos.posX > 0) ){
        pos.velX = -stats.step;
        turn = false;
    }else if ((IsKeyDown(input.KeyS) && pos.posY < 14) ){
        pos.velY = stats.step;
        turn = false;
    }else if ((IsKeyDown(input.KeyD) && pos.posX < 14) ){
        pos.velX = stats.step;
        turn = false;
    }else if (IsKeyDown(KEY_ESCAPE)){
        playing_lvl = false;
        state = States::MENU;
    }
    checkKey();
}
```

El último detalle de esta método a tener en cuenta es otro método denominado "checkKey()". Este método se encarga de controlar que solo se pulse una sola vez la tecla ESC. Esto ocurre por que cada segundo se ejecuta 60 veces el bucle del juego, por lo tanto de una pulsación a ESC, pulsaríamos muchas más veces y en lugar de salir al menú, saldríamos del juego totalmente.

El resto de teclas no necesita comprobación, por que la primera vez que se pulsa, cambia el turno, por lo tanto el sistema de input queda deshabilitado. Esto lo veremos próximamente en la clase "Game".

La función que controla el valor del Escape es la siguiente:

Input System: checkKey

```
void InputSystem::checkKey(){
    esc_pressed = IsKeyDown(KEY_ESCAPE);
}
```

En el menú tendremos otra función que veremos más adelante, y es donde usaremos "esc_pressed", entonces allí tendrá que ser falsa para poder ser pulsado el escape. Si es verdadera, significa que la tecla ya viene pulsada y no ejecutará ninguna acción.

Ahora que disponemos de estas dos funciones, implementaremos el "update" de este sistema, que simplemente llamará a la función "keyPressed" con cada una de las entidades que reciba, que en nuestro caso será la entidad jugador únicamente.

Input System: update

```
void InputSystem::update(EntityManager& EM, bool& turn,
                    bool& playing_lvl, States& state){
    EM.forallMatching([&](Entity& e){
        keyPressed(EM, e, turn, playing_lvl, state);
    }, cmpMaskToCheck, tagMaskToCheck);
}
```

7.5 Sistema de IA: Movimiento de enemigos

El sistema de IA permite que los enemigos decidan donde moverse o que patrón de movimiento seguir en cada momento. Vamos a comenzar a definir las diferentes funciones que hay en este sistema. Sabemos que necesitamos el "update" pero necesitamos muchas más cosas, como por ejemplo:

- Una función que detecte si el jugador está cerca del enemigo, con una distancia de visión.
- Una función que ejecute el movimiento del enemigo, ya sea en patrón o persiguiendo al jugador.
- Una función que haga una comprobación al intento de movimiento, para saber si se puede o no mover a donde ha elegido.

Comenzaremos detectando si el "player" esta cerca de la entidad enemigo. Para ello, implementaremos "isPlayerNearby", la cual recibe por parámetro la referencia a ambas posiciones, jugador y enemigo, y la distancia de detección.

Calcula el valor absoluto de la distancia en horizontal y vertical, si ambas son menores o iguales que la distancia concretada, devolverá verdadero, si no, falso.

IA System: Comprobamos si el jugador está en rango

```
bool IASystem::isPlayerNearby(PositionCMP& player_pos,
                        PositionCMP& enemy_pos, int distance){
    int deltaX = abs(enemy_pos.posX - player_pos.posX);
    int deltaY = abs(enemy_pos.posY - player_pos.posY);
    return (deltaX <= distance && deltaY <= distance);
}
```

Una vez tenemos esto, podemos plantear lo que será el update de nuestro sistema. Ejecutaremos esta función, si es verdadera tendremos que perseguir al jugador, si no, haremos un patrón aleatorio por el mapa.

Recuerdo, que las máscaras del sistema comprobarán si tiene componente de posición, y que además tenga dos tags, "enemy" y "movement".

IA System: update

```
void IASystem::update(EntityManager& EM, GameManager& GM){
  EM.forallMatching([&](Entity& e){
   auto& ene_pos = EM.getCMPStorage().getPositionCMP(e);
   auto& pla_pos =  EM.getCMPStorage().getPositionCMP(EM.getPlayer());
   if (isPlayerNearby(pla_pos, ene_pos, 5)){
       //PLAYER CERCA - PERSEGUIR
   }else{
       //PLAYER LEJOS - PATRÓN ALEATORIO
   }
  }, cmpMaskToCheck, tagMaskToCheck);
}
```

Esta será la estructura del sistema. A continuación crearemos una función para comprobar si la posición a la que se intenta mover el enemigo será correcta o no. Para ello creamos "tryMoving". Este método recibe por parámetro 4 enteros, posición en el eje X, en el Y, velocidad X e Y, y por último una referencia a "Game Manager", ya que necesitaremos la matriz del nivel actual.

Obtenemos la posición a la que intenta moverse mediante la posición actual de la entidad y su velocidad. En primer lugar comprobamos que esa posición esté dentro de los límites del tablero. Si eso devuelve verdadero significa que es una posición errónea.

La segunda comprobación se basará en el mapa actual. Si la posición a la que intentamos movernos es diferente de 0, no podremos movernos, por lo tanto devolverá verdadero.

IA System: Intentamos movernos

```
bool IASystem::tryMoving(int posX, int posY, int velX, int velY,
                                                    GameManager& GM) {
    auto& map = GM.getActualMap();
    int nextPosX = posX + velX;
    int nextPosY = posY + velY;
    if (nextPosX < 0 || nextPosX > 14 ||
                nextPosY < 0 || nextPosY > 14) {
        // La siguiente posición esta fuera de rango
        return true;
    }
    if (map[nextPosY][nextPosX] != 0) {
        // La siguiente posición está ocupada
        return true;
    }
    return false;
```

```
}
```

Teniendo esta función lista, falta un último método, que será muy extenso, cuyo objetivo es mover al enemigo teniendo en cuenta las casillas disponibles mediante la función que acabamos de crear, decidiendo que versión de la función ejecutaremos según la proximidad.

Crearemos la función "enemyMovement", que recibe como parámetro el "Game Manager", la posición del personaje, la posición del enemigo y el modo que queremos ejecutar. Hay dos modos posibles, que nos persiga, o que haga un patrón aleatorio.

Esta es la estructura que tendrá:

IA System: Función para decidir hacia donde se mueve el enemigo

```
void IASystem::enemyMovement(GameManager& GM, PositionCMP& player_pos,
                             PositionCMP& enemy_pos, int mode){
    if( mode==0 ){
        //PERSECUCIÓN
    }else{
        //PATRÓN ALEATORIO
    }
}
```

Comenzaremos pasándole el número 1 como parámetro modo. Por lo tanto comenzaremos por el patrón aleatorio. Calcularemos la dirección aleatoriamente, dependiendo hacia donde nos movemos, colocaremos una velocidad u otra. Si la decisión del enemigo no es válida, lo comprobaremos también, entonces no se moverá.

IA System: Patrón de movimiento aleatorio

```
...

//PATRÓN ALEATORIO
int direction = (rand() % 3 == 1) ? -1 : 1;
if (rand() % 3 == 1){
    if(direction<0 &&
    !tryMoving(enemy_pos.posX, enemy_pos.posY, direction, 0, GM)){
        enemy_pos.velX += 1 * direction;
    }else if(direction>0 &&
    !tryMoving(enemy_pos.posX, enemy_pos.posY, direction, 0, GM)){
        enemy_pos.velX += 1 * direction;
    }
}else{
    if(direction<0 &&
```

```
    !tryMoving(enemy_pos.posX, enemy_pos.posY, 0, direction, GM)){
        enemy_pos.velY += 1 * direction;
    }else if(direction>0 &&
    !tryMoving(enemy_pos.posX, enemy_pos.posY, 0, direction, GM)){
        enemy_pos.velY += 1 * direction;
    }
}

...
```

De esta forma el enemigo moverá aleatoriamente por el mapa. Hasta que detecte al enemigo, entonces tendremos que ejecutar unas instrucciones diferentes las cuales dividiremos en tres partes.

- En primer lugar intentará moverse a la posición más cercana al jugador
- Si no se ha podido mover en ese turno porque tiene un obstáculo delante, se moverá a otra que esté libre alrededor suya. De esta forma se desbloqueará si ha quedado atrapado en obstáculos.
- Por último, si ha decidido moverse en varias direcciones, tendremos que decidir hacia cual se moverá, ya que no permitimos el movimiento diagonal.

Para intentar movernos a la posición más cercana al jugador comprobaremos la posición horizontal y vertical, si está en la misma X entonces moverá verticalmente y si está en la misma Y moverá horizontalmente. Si no coincide ni en X ni en Y, tomará ambas direcciones como válidas.

IA System: Persecución - Decidimos hacia donde movernos

```
...

//PERSECUCIÓN
bool choose= false;
//Intenta moverse a la posicion mas cercana al jugador
if (player_pos.posX > enemy_pos.posX &&
        !tryMoving(enemy_pos.posX, enemy_pos.posY, 1, 0, GM)) {
    enemy_pos.velX = 1;
    choose = true;
}else if (player_pos.posX < enemy_pos.posX &&
        !tryMoving(enemy_pos.posX, enemy_pos.posY, -1, 0, GM)) {
    enemy_pos.velX = -1;
    choose = true;
}
if (player_pos.posY > enemy_pos.posY &&
        !tryMoving(enemy_pos.posX, enemy_pos.posY, 0, 1, GM)) {
    enemy_pos.velY = 1;
```

```
    choose = true;
} else if (player_pos.posY < enemy_pos.posY &&
        !tryMoving(enemy_pos.posX, enemy_pos.posY, 0, -1, GM)) {
    enemy_pos.velY = -1;
    choose = true;}

...
```

Ahora, siguiendo el flujo definido, tenemos que comprobar si el enemigo ha decidido hacia que dirección moverse, para ello comprobamos la variable "choose" que hemos definido al principio del método.

Mediante el uso de la función "tryMoving", intentamos movernos a otras posiciones. Si no fuera posible, por que tiene obstáculos en todas, quedará atrapado.

IA System: Persecución - Elegimos otra dirección si no se decidió

```
...

//Si no se ha podido mover a la posicion que eligió, intenta otra
if(!choose){
    if (!tryMoving(enemy_pos.posX, enemy_pos.posY, 1, 0, GM)) {
        enemy_pos.velX = 1;
    }else if (!tryMoving(enemy_pos.posX, enemy_pos.posY, -1, 0, GM)) {
        enemy_pos.velX = -1;
    }
    if (!tryMoving(enemy_pos.posX, enemy_pos.posY, 0, 1, GM)) {
        enemy_pos.velY = 1;
    }else if (!tryMoving(enemy_pos.posX, enemy_pos.posY, 0, -1, GM)) {
        enemy_pos.velY = -1;
    }
}

...
```

Si ha conseguido moverse, en este punto de la función, ya tendrá velocidad asociada. Ahora, como último paso de la persecución, tendremos que comprobar que si tiene velocidad asociada, no la tenga en ambas direcciones, ya que no queremos movimientos diagonales de los enemigos.

Podríamos tener otro tipo de enemigo que si las permita, entonces podríamos crear una etiqueta para ellos, y este código solo lo ejecutarían los enemigos que no la tengan. Las "tags" nos dan mucho juego a la hora de crear diferentes inteligencias o comportamientos.

Sin embargo, no estarán implementados, y tendremos que quedarnos con la velocidad

teniendo en cuenta la distancia más cercana, es decir, si la resta de posición X entre jugador y enemigo es mayor o igual a la vertical, haremos la velocidad vertical cero. En el caso de que sea al revés, la horizontal será cero.

IA System: Persecución - Control de movimiento diagonal

```
...

// No permitimos la diagonal
if (enemy_pos.velX!=0 && enemy_pos.velY!=0){
    if (abs(player_pos.posX - enemy_pos.posX) >=
            abs(player_pos.posY - enemy_pos.posY)) {
        enemy_pos.velY = 0;
    }else{
        enemy_pos.velX = 0;
    }
}

...
```

Ya tenemos el sistema con todo el material necesario para actualizar la velocidad del enemigo en cada movimiento. Nos falta completar el método "update". Simplemente llamaremos a la función que hemos creado, con 0 o 1 como modo.

IA System: update

```
void IASystem::update(EntityManager& EM, GameManager& GM){
  EM.forallMatching([&](Entity& e){
   auto& ene_pos = EM.getCMPStorage().getPositionCMP(e);
   auto& pla_pos =  EM.getCMPStorage().getPositionCMP(EM.getPlayer());
   if (isPlayerNearby(pla_pos, ene_pos, 5)){
       //PLAYER CERCA - PERSEGUIR
       enemyMovement(GM, pla_pos, ene_pos, 0);
   }else{
       //PLAYER LEJOS - PATRÓN ALEATORIO
       enemyMovement(GM, pla_pos, ene_pos, 1);
   }
  }, cmpMaskToCheck, tagMaskToCheck);
}
```

Estos son los últimos detalles de este sistema. Ahora nuestros enemigos podrán moverse en cada turno del que dispongan (Los turnos los implementaremos en la clase "Game" más adelante).

7.6 Sistema de colisiones: Interactuamos con el entorno

El sistema de colisiones es el más extenso y el último de los que nos quedan por implementar.

La jugabilidad casi completa depende de este sistema, ya que cada vez que choquemos, cojamos un objeto, recibamos daño, ataquemos, accedamos a una puerta, etc, tendrá lugar aquí.

Comenzaremos definiendo que ocurrirá en el "update" de este sistema.

En primer lugar, configuraremos las máscaras, aunque este sistema es un poco especial, ya que usará cuatro, dos de componentes y dos de etiquetas. Esto se debe a que hay que lanzar a la vez dos "forallMatching". La primera buscará con la máscara de componentes con "PositionCMP" y como tag "Collider". El siguiente "forallMatching", lo lanzaremos en el interior de este otro, porque cada entidad "collider", tendrá que revisar si existe colisión con el resto de entidades que son "collisionables".

La colisión se produce cuando una entidad "collider" se mueve, por lo tanto, puede ser que en ese movimiento exista colisión con algún elemento que tendremos que comprobar.

Las diferentes colisiones que se pueden dar son:

- Colisión con un muro. Este tipo de colisiones consisten en que no te puedas mover hacia esa dirección, ya que está ocupada por un obstáculo. Sin embargo, si aun así te mueves en esa dirección, le harás daño (si eres el jugador, los enemigos no harán daño a obstáculos). Si acabas con la resistencia de un obstáculo, este desaparece. Para ello necesitaremos las referencias a las entidades "collider" y "collisionable" por parámetro.
  Obtenemos las estadísticas del player, y el render, stats y posición de los muros. Si nos movemos hacia el obstáculo le hacemos daño, es decir, restamos a la vida del muro, el daño de pico del "player". Actualizaremos el sprite de la roca, que pasará a otro con una grieta, para dar feedback al usuario. Por último si es menor o igual a 0 la vida, desaparece el obstáculo, eliminando la entidad y actualizando el mapa.

Collision System: Colisión con obstáculo

```
void CollisionSystem::collisionWithWall(EntityManager& EM,
GameManager& GM,Entity& ent,Entity& wall,PositionCMP& collider){
    //Check Collision
    collider.velX = 0; collider.velY = 0;
    if(ent.hasTag(Tags::player)){    // Hit Wall
        auto& statsPlayer = EM.getCMPStorage().getStatsCMP(ent);
        auto& statsWall = EM.getCMPStorage().getStatsCMP(wall);
```

```
        auto& rendWall = EM.getCMPStorage().getRenderCMP(wall);
        auto& posWall = EM.getCMPStorage().getPositionCMP(wall);
        statsWall.health -= statsPlayer.pickaxe;
        rendWall.actual_frame++;
        rendWall.frame = {
            static_cast<float>(rendWall.actual_frame*32), 0,
            static_cast<float>(32),
            static_cast<float>(rendWall.sprite.height)};
        if(statsWall.health <= 0){
            //Eliminar Entidad
            GM.getActualMap()[posWall.posY][posWall.posX] = 0;
            EM.removeEntity(wall);
        }
    }
}
```

- Colisión con un enemigo. La condición es que si la entidad "collider" tiene además "player" ejecutaremos esta función. Al colisionar con un enemigo, le haremos daño. Para ello tendremos que recoger las estadísticas de ambas entidades, y también la posición de ambas. La velocidad pasa a ser 0 ya que no nos moveremos a su posición, si no que golpearemos.
  Después pediremos un número aleatorio entre el valor crítico que tenemos guardado en "statsCMP" y 100. Si sale un numero menor o igual, el golpe será critico y aplicaremos el doble de daño. Si no es crítico simplemente restamos a la vida del enemigo el daño del jugador.
  Si el enemigo tiene 0 de vida, muere y eliminamos la entidad, a parte de actualizar el mapa.

Collision System: Colisión con enemigo

```
void CollisionSystem::collisionWithEnemy(EntityManager& EM,
GameManager& GM, Entity& player, Entity& enemy){
    auto& player_stats = EM.getCMPStorage().getStatsCMP(player);
    auto& player_pos  = EM.getCMPStorage().getPositionCMP(player);
    auto& enemy_stats = EM.getCMPStorage().getStatsCMP(enemy);
    auto& enemy_pos   = EM.getCMPStorage().getPositionCMP(enemy);
    player_pos.velX = 0; player_pos.velY = 0;
    if(rand() % 100 <= player_stats.critical_hit){
        enemy_stats.health = enemy_stats.health -
                                    player_stats.damage*2;
    }else{
        enemy_stats.health = enemy_stats.health -
                                    player_stats.damage;
    }
```

```
    if (enemy_stats.health <= 0){
        GM.getActualMap()[enemy_pos.posY][enemy_pos.posX] = 0;
        EM.removeEntity(enemy);
    }
}
```

- Enemigo ataca a jugador. Esta es al revés que la anterior, solo que sin tener en cuenta el golpe crítico, y si morimos cambiaremos de estado y habrá acabado el juego. El enemigo tampoco se moverá, si no que atacará.

Collision System: Colisión de enemigo con jugador

```
void CollisionSystem::collisionWithPlayer(EntityManager& EM,
Entity& player, Entity& enemy, States& state, bool& playing_lvl){
    auto& player_stats = EM.getCMPStorage().getStatsCMP(player);
    auto& enemy_pos  = EM.getCMPStorage().getPositionCMP(enemy);
    auto& enemy_stats = EM.getCMPStorage().getStatsCMP(enemy);
    player_stats.health -= enemy_stats.damage;
    enemy_pos.velX = 0; enemy_pos.velY = 0;
    if(player_stats.health <= 0){
        //Morimos
        std::cout<< "Has muerto\n";
        state = States::ENDGAME;
        playing_lvl = false;
    }
}
```

- Colisionamos con un cofre. Al colisionar con un cofre, no nos moveremos, ya que es un obstáculo al igual que otros objetos. Generaremos un objeto entre los existentes de forma aleatoria, y después le añadimos las estadísticas del objeto al jugador. Finalmente, cambiamos el sprite del cofre para que aparezca abierto, y le añadimos un tag nuevo "object_picked" para que no podamos colisionar más con este cofre.

Collision System: Colisión con cofre

```
void CollisionSystem::collisionWithChest(EntityManager& EM,
GameManager& GM, Entity& player, Entity& chest){
    //COLISIONAMOS CON EL OBJETO
    auto& posPlayer = EM.getCMPStorage().getPositionCMP(player);
    posPlayer.velX = 0;
    posPlayer.velY = 0;
    //GENERAR OBJ ALEATORIAMENTE
    auto& obj = GM.createObject(EM, rand()%5);
    //APLICAR SUS ESTADISTICAS AL PLAYER
    auto& statsPlayer = EM.getCMPStorage().getStatsCMP(player);
```

```
    auto& statsObj    = EM.getCMPStorage().getStatsCMP(obj);
    statsPlayer.addStats(statsObj);
    //CAMBIAR SPRITE DEL COFRE Y HACERLO INACCESIBLE
    auto& rendChest = EM.getCMPStorage().getRenderCMP(chest);
    rendChest.actual_frame++;
    rendChest.frame = {
        static_cast<float>(rendChest.actual_frame*32),
        0,32.0f, static_cast<float>(rendChest.sprite.height)};
    chest.addTag(Tags::object_picked);
}
```

- Colisión con llave. Obtenemos la posición de la llave y al colisionar con ella la movemos a otro lado de la interfaz (esto dará la sensación de que ha sido recogida). Además, añadimos al jugador la etiqueta "has_key", que es el comprobante para poder abrir la puerta próximamente. Actualizamos el "sprite" de la puerta para que aparezca abierta y listo.

Collision System: Colisión con llave

```
void CollisionSystem::collisionWithKey(EntityManager& EM,
GameManager& GM, Entity& player, Entity& key){
    auto& posKey = EM.getCMPStorage().getPositionCMP(key);
    GM.getActualMap()[posKey.posY][posKey.posX] = 0;
    posKey.posX =  16;
    posKey.posY =   0;
    player.addTag(Tags::has_key);
    EM.forallMatching([&](Entity& door){
        auto& rend_door = EM.getCMPStorage().getRenderCMP(door);
        rend_door.actual_frame++;
        rend_door.frame = {
            static_cast<float>(rend_door.actual_frame*32), 0,
            32.0f, static_cast<float>(rend_door.sprite.height)};
    }, 0, Tags::door);
}
```

- Colisión con puerta. Para la colisión con la puerta necesitaremos simplemente modificar la variable de "Game Manager" que representa el nivel en el que estamos, al nivel siguiente. Para ello usamos la función "nextLvl()" y terminamos con el bucle del juego, ya que al acabar el nivel lo recreamos con el siguiente.

Collision System: Colisión con puerta

```
void CollisionSystem::collisionWithDoor(GameManager& GM,
                                        bool& playing_lvl){
    GM.nextLvl();
```

```
        playing_lvl = false;
    }
```

Una vez tenemos todas las posibles colisiones programadas, tenemos que controlar cuando se produce una colisión. Para ello tendremos que comprobar si el siguiente movimiento de una entidad "collider" es una "collisionable".

Comprobaremos hacia qué dirección nos estamos moviendo. Si una de las dos comprobaciones (la del eje X o la del eje Y) ha coincidido con una entidad "collisionable", habrá colisión.

Collision System: Comprobamos colisión

```
bool CollisionSystem::checkCollision(PositionCMP& pos1,
PositionCMP& pos2){
    bool removedHorizontal = false;
    bool removedVertical = false;
    int directionX=0;
    if(pos1.velX<0){ directionX= -1; }else{ directionX = 1; }
    int directionY=0;
    if(pos1.velY<0){ directionY= -1; }else{ directionY = 1; }
    if((pos1.velX != 0) &&
        (pos2.posY == pos1.posY &&
        ((pos1.posX + directionX) == pos2.posX))){
        //Try moving HORIZONTAL (Key A OR D)
        removedHorizontal = true;
        pos1.velX = 0;
    }
    if((pos1.velY != 0) &&
        (pos2.posX == pos1.posX &&
        ((pos1.posY + directionY) == pos2.posY))){
        //Try moving VERTICAL (Key W OR S)
        removedVertical = true;
        pos1.velY = 0;
    }
    if(removedHorizontal || removedVertical){
        return true;
    }
    return false;
}
```

Teniendo todas las cartas sobre la mesa, podemos construir un método "update" que compruebe si hemos colisionado con alguna entidad. De ser así, comprobaremos los tags de esa entidad, para ver de que elemento se trata. Por lo tanto ahora tenemos un método "update" muy limpio y ordenado.

Este método queda de la siguiente forma:

Collision System: update

```
void CollisionSystem::update(EntityManager& EM, GameManager&
GM, bool& playing_lvl, States& state){
    EM.forallMatching([&](Entity& e){
        auto& posCMPOfCollider = EM.getCMPStorage().getPositionCMP(e);
        if(posCMPOfCollider.velX!=0 || posCMPOfCollider.velY!=0){
            EM.forallMatching([&](Entity& coll){
            auto& pCMPofColl = EM.getCMPStorage().getPositionCMP(coll);
            if(checkCollision(posCMPOfCollider,
                                pCMPofColl)){
                if(coll.hasTag(Tags::wall)){
                  collisionWithWall(EM, GM, e, coll, posCMPOfCollider);
                }else if(!e.hasTag(Tags::enemy) &&
                coll.hasTag(Tags::enemy)){
                    collisionWithEnemy(EM, GM, e, coll);
                }else if(e.hasTag(Tags::enemy) &&
                coll.hasTag(Tags::player)){
                  collisionWithPlayer(EM, coll, e, state, playing_lvl);
                }else if(e.hasTag(Tags::player) &&
                coll.hasTag(Tags::chest) &&
                !coll.hasTag(Tags::object_picked)){
                    collisionWithChest(EM, GM, e, coll);
                }else if(e.hasTag(Tags::player) &&
                coll.hasTag(Tags::key)){
                    collisionWithKey(EM, GM, e, coll);
                }else if(e.hasTag(Tags::player | Tags::has_key) &&
                coll.hasTag(Tags::door)){
                    collisionWithDoor(GM,playing_lvl);
                }
                }
            }, cmpMaskToCheck, Tags::collisionable);
        }
    },cmpMaskToCheck, tagMaskToCheck);
}
```

Esta función "update" del sistema de colisión es muy extensa, pero comprobamos todas las colisiones individualmente. Simplemente elegimos cual ejecutar según el tipo de objeto con el que chocamos.

Con este, terminamos todos los sistemas del juego, y vamos a proceder a crear un sistema de estados, para representar las diferentes pantallas disponibles.

7.7 Montaje del juego: Estados

Para ofrecer un buen juego tendremos que añadir más que una única pantalla donde jugar. Crearemos cuatro estados que recorreremos durante la ejecución.

En primer lugar tendremos un estado menú, donde presentaremos el juego y nos sirva de pantalla principal.

Tendremos el estado juego también. En el se desarrollaran los diferentes niveles.

Por último, dispondremos de un estado final "endgame", para mostrar una pantalla final. El objetivo de este estado es comunicar al jugador que ha terminado el juego y ha muerto. Es al estado al que cambiamos cuando un enemigo acaba con la vida de la entidad "player".

En este enumerado se recogen los estados propuestos:

States: Estados del juego

```
enum States{
    MENU, GAME, ENDGAME
};
```

Estos estados se despliegan en el bucle del método "run" de "Game". Para el su control disponemos de una variable que ha aparecido anteriormente de tipo "States" que llamaremos "state" y es la encargada de guardar el estado actual.

El método "run" tendrá un bucle que se ejecutará mientras "running" esté activo. Dentro de este bucle es donde encontramos los diferentes ámbitos de cada estado.

Game: Despliegue de estados

```
void Game::run(){
    bool running = true;
    while (running){
        switch (state) {
            case States::MENU:
                break;
            case States::GAME:
                break;
            case States::ENDGAME:
                break;
        }
    }
}
```

Ahora que tenemos los estados listos, comencemos a desarrollar cada uno de sus ámbitos. Comenzaremos por el estado "MENU".

7.8 Montaje del juego: Menú

Aunque no es esencial para poder jugar, hemos implementado este estado, que añade una mejora visual y de flujo de pantallas en nuestro juego.

En caso de que la variable "state" contenga el 0, corresponderá al menú del juego. Vamos a crear una función que ejecute el bucle de este menú, que es muy sencillo.

Necesitaremos pasarle una referencia de la variable "map", que contiene todos los fondos de pantalla del menú y demás pantallas. Además de esta, le pasaremos también por referencia la variable que hemos creado en "Game", "running". Esta podrá cambiar en el menú, con la intención de cerrar el juego.

Cada vez que el flujo del juego llegue a la pantalla del menú, lo primero que tenemos que hacer es un reset. Eliminaremos todas las entidades y reiniciaremos las variables del juego. Para ello crearemos un método auxiliar en la clase "Game" que llamaremos "reset" y recibe por parámetro un entero, correspondiente a la configuración.

- En primer lugar, usaremos "killPlayer", que recuerdo que cambia a falsa la variable que nos dice si está o no vivo el jugador en el "Game Manager".
- Seguiremos guardando el inventario, para poder mantenerlo, ya que esta función también la llamaremos para resetear el nivel con la configuración "1".
- Eliminamos posteriormente todas las entidades del juego.
- Ahora crearemos una diferencia para el tipo de configuración que nos llegue.
- La variable "playing_lvl" pasará a ser falsa, y el nivel, en caso de que estuviese activo, terminará. Y el objeto "state", lo configuramos a "MENU".
- Por último llamaremos a la función de "Game Manager" denominada "resetLvl" que creamos anteriormente. Recuerdo que este método se encarga de poner el número de nivel a 0.

El reset del juego queda de la siguiente forma:

Game: Reset

```
void Game::reset(int config){
    GM.killPlayer();
    GM.saveInventory(EM);
    EM.removeAllEntities();
    EM.getCMPStorage().clearAllStorage();
    if (config == 0){
        GM.resetLvl();
        playing_lvl = false;
```

```
        state = MENU;
    }
}
```

En este método podemos añadir una especie de comprobantes, mediante salida por pantalla, antes y después de vaciar los almacenamientos tanto de entidades como de componentes, para comprobar que se están eliminando de forma correcta.

Necesitaremos otras funciones que aún no están implementadas, como un input específico para el menú, o un "renderMenu" que añadiremos a los respectivos sistemas como función extra. En este estado, no llamaremos a los sistemas ya que no disponemos de entidades, pero podemos llamar a estas funciones auxiliares que harán algo similar.

La función del sistema de Input que hemos llamado "menuKeys", comprobará en cada momento que estemos en este estado, que teclas permitidas estarán pulsadas.

Para ello creamos la función y comprobamos si las teclas "ESC" y "Space" están pulsadas. De estar pulsado el "ESC", hará que la variable "running" pase a ser falsa, saliendo del juego. A parte de comprobar la tecla, en este caso también comprobaremos si la variable "esc_pressed" que hemos implementado en el sistema de Input sea falsa. Esto significa que no está la tecla pulsada al entrar al menú, evitando una salida directa desde el estado de juego.

Al pulsar el espacio, simplemente cambiará el estado a "GAME".

Finalmente, comprobaremos mediante la función "checkKey()", el estado de la tecla "ESC", para evitar problemas.

Input System: Teclas para el menú

```
void InputSystem::menuKeys(int config, States& state, bool& running){
    if (IsKeyDown(KEY_SPACE)) {
        state = States::GAME;
    }else if (IsKeyDown(KEY_ESCAPE) && esc_pressed == false){
        running = false;
    }
    checkKey();
}
```

Por último, pintaremos el fondo del menú con la función propuesta anteriormente "renderMenu", que recibe el mapa y pinta el menú cada iteración del bucle.

Render System: Pintamos el menú

```
void RenderSystem::renderEndGame(Map& map){
```

```
    DrawTexture(map.menu.endgame,0,0,WHITE);
}
```

Teniendo todas estas funciones listas, podemos implementar el menú, con un orden, primero borramos todo y seguidamente pintamos el menú. Al final controlamos las teclas por si tenemos que cambiar a otro estado o salir del juego.

Game: Estado "MENU"

```
void Game::menuScreen(Map& map, bool& running){
    reset(0);
    BeginDrawing();
    ClearBackground(RAYWHITE);
    rendSys.renderMenu(map);
    inpSys.menuKeys(States::MENU, state, running);
    EndDrawing();
}
```

7.9 Montaje del juego: Partidas

Para acceder al juego tendremos que haber pulsado la tecla "Space" desde el menú. Entonces cambiará el estado a "GAME".

En este estado, se ejecutarán todos los sistemas anteriores siguiendo un orden, lo que es muy importante, y cada vez más ya que tenemos proyectos más grandes.

En primer lugar, al entrar al estado del juego, llamaremos a la función "reset", pero esta vez con la configuración por parámetro como 1. Seguidamente generamos el nivel actual. De esta forma, cada vez que accedamos a "GAME" crearemos un nivel. La variable de "Game" "playing_lvl", pasará a ser verdadera, y ejecutaremos el bucle del juego.

Game: Estado "GAME"

```
...

case States::GAME:
    reset(1);
    GM.selectLvlAndGenerate(EM , GM.getActualLvl());
    playing_lvl = true;
    startNormalGame(map);
    break;

...
```

La función "startNormalGame" recibe un mapa, y es la encargada de ejecutar el bucle del juego. Vamos a construirla.

El bucle del juego se ejecutará mientras "playing_lvl" sea verdadera.

Ahora es el momento de implementar los turnos entre enemigo y personaje, es lo primero que tendremos que tener en cuenta al ejecutar cada vez el bucle. ¿De quién es el turno?

Esto lo haremos de una forma sencilla y es creando tres variables en la clase "Game". La primera decidirá de quien es el turno y será un "bool", las otras dos controlarán el tiempo. La variable "time", establecerá en cuanto tiempo se ejecutará el movimiento del enemigo, y "turn_seconds" contará el tiempo.

Game: Turnos

```
...

    bool  turn          = true;
    float time          = 0.25;
    float turn_seconds  = 0.0;

...
```

Al acceder al bucle, siempre contamos el tiempo transcurrido mediante una función llamada "GetFrameTime()" de la librería RayLib. Seguidamente tendremos que ordenar los sistemas.

- En primer lugar comprobaremos si "turn" es verdadero. De ser así ejecutaremos el sistema de input, para que se mueva el jugador, si es falsa por que el jugador ya se ha movido o ha realizado alguna acción, será turno del enemigo y por lo tanto lanzaremos el sistema de IA.
- Seguido a estos sistemas, lanzaremos el de colisiones, ya que para comprobar si hay colisión, ya sea por parte de enemigos o jugador, hay que moverse primero.
- A continuación, el sistema de posiciones es el adecuado. Nos hemos movido, hemos comprobado las colisiones, y si aun así tenemos velocidad tendremos que actualizar las posiciones.
- Finalmente, renderizaremos todas las entidades en su posición adecuada gracias al orden seguido. El sistema de renderizado será el último en ejecutarse.

El bucle del juego queda de la siguiente forma:

Game: Bucle del juego

```
void Game::startNormalGame(Map& map){
    while (playing_lvl){
        turn_seconds += GetFrameTime();
        BeginDrawing();
        ClearBackground(RAYWHITE);
        if (turn){
```

```
            inpSys.update(EM, turn, playing_lvl, state);
            turn_seconds = 0;
        }else{
            if (turn_seconds >= time){
                iaSys.update(EM, GM);
                turn=true;
                turn_seconds = 0;
            }
        }
        collSys.update(EM, GM, playing_lvl, state);
        posSys.update(EM, GM);
        rendSys.update(EM,GM, map);
        EndDrawing();
    }
}
```

Recordamos que solo salimos del bucle del juego si nos mata un enemigo, si accedemos a una puerta, o si pulsamos "ESC".

Si pulsamos la tecla "escape", salimos directamente a la interfaz del menú. Si atravesamos la puerta, rompemos este bucle e incrementamos el número del nivel, subiendo un escalón en el código y generando el mapa y entidades del siguiente nivel. Si nos mata un enemigo, cambiamos el estado a un estado final llamado "ENDGAME".

El estado "ENDGAME", funcionará similar al resto, pero será temporal. Con esto quiero decir que a los 5 segundos se quitará y volveremos al menú. También podremos quitarlo pulsando la tecla "ESC".

Los sistemas que se ejecutan en este estado tienen una pequeña diferencia. No se eliminan las entidades al cambiar de estado y esto ocurre por que queremos seguir renderizando el juego bajo la pantalla de final del juego.

Para ello lanzaremos el sistema de render seguido a una función auxiliar para renderizar la pantalla final. Esta función es la siguiente:

Render System: Pintamos la pantalla de muerte

```
void RenderSystem::renderEndGame(Map& map){
    DrawTexture(map.menu.endgame, 0, 0, WHITE);
}
```

No tiene mucha más dificultad este estado, ya que no hay jugabilidad. Por lo tanto el bucle del estado "ENDGAME" queda plasmado en la siguiente función:

Game: Bucle final

```
void Game::endGame(Map& map){
    turn_seconds = 0;
    while (turn_seconds < 5 && !IsKeyDown(KEY_ESCAPE) ){
        turn_seconds += GetFrameTime();
        BeginDrawing();
        ClearBackground(RAYWHITE);
        rendSys.update(EM,GM, map);
        rendSys.renderEndGame(map);
        EndDrawing();
        inpSys.checkKey();
    }
    state = States::MENU;
}
```

Hemos terminado la implementación del juego. A falta de ver como ha quedado el método "run()" de la clase "Game". Se puede observar el resultado a continuación:

Game: run

```
void Game::run(){
    Map map{};
    bool running = true;
    SetTargetFPS(60);
    while (running){
        switch (state) {
            case States::MENU:
                menuScreen(map, running);
                break;
            case States::GAME:
                reset(1);
                GM.selectLvlAndGenerate(EM , GM.getActualLvl());
                playing_lvl = true;
                startNormalGame(map);
                break;
            case States::ENDGAME:
                endGame(map);
                break;
        }
    }
}
```

Antes de ir a los resultados del proyecto, propondré dos ejercicios finales.

7.10 Ejercicios

En este apartado, propongo dos ejercicios y su implementación. El objetivo es que apliques lo aprendido a la hora de crear componentes y como usarlos para implementar otras funcionalidades.

En primer lugar habrá un ejercicio donde tendremos que crear un componente y todo lo necesario para que funcione en nuestro motor. Posteriormente tendremos que usarlo para crear una página "biblioteca", donde exponer las diferentes entidades. Esto servirá de guía para el usuario, ya que le mostramos que hace cada elemento.

En segundo lugar, implementaremos el uso de un "sprite", cuando el golpe que demos al enemigo sea crítico. De esta forma, la persona que juegue, sabrá lo que ha pasado.

7.10.1 Ejercicio: Biblioteca o WIKI

En este apartado, es conveniente que trabajes tú primero. El objetivo es crear un nuevo método para que en el menú, podamos acceder a una biblioteca o "wiki", donde encontremos todas las diferentes entidades que hay en el juego con una descripción de lo que hacen.

Es conveniente crear un nuevo componente, para almacenar la descripción de cada entidad. Además, es recomendable crear un nuevo estado "WIKI".

¿Cómo lo harías? El objetivo es intentarlo antes de seguir.

Implementación. Comenzaremos creando el componente:

Este componente tiene una función sencilla y es crear una descripción para los objetos, personaje, enemigos y resto de entidades que la requieran. El objetivo es tener un apartado donde explicar al usuario que es ese objeto, para que sirve o cualquier otra cosa que se nos ocurra. La máscara correspondiente a este objeto es el sexto bit activo y todo lo demás a cero.

Componente descripción

```
struct DescriptionCMP{
    const char* description{""};
    static const int mask {0b100000};
};
```

Crearemos otro estado, que es el estado biblioteca o "WIKI", donde recogeremos una entidad de cada tipo de objetos que hemos creado, y le adjuntaremos una descripción con el componente "DescriptionCMP" que acabamos de crear.

States: Estado "WIKI" añadido

```
enum States{
    MENU, GAME, WIKI, ENDGAME
};
```

Para este menú también necesitaremos métodos auxiliares específicos para esta pantalla. En primer lugar, recordamos que en este punto no tenemos entidades, así que vamos a crear una función que nos cree una de cada elemento del juego, con su componente descripción y llamando a las diferentes funciones que hemos implementado, la del "player", la de los enemigos, la del cofre, la de la llave, etc.

Es una función extensa, y solo creamos entidades y le asociamos su componente descripción, de la misma forma que hemos ido haciendo anteriormente cuando generábamos el nivel.

Lo siguiente que hay que preparar es un método que funcione como "update" para este estado. Simplemente dibujará cada sprite del componente de render de cada entidad bajo el de la anterior, para ello controlamos las posiciones cada iteración. Además llamamos a un método auxiliar para pintar una descripción, que pasandole la entidad y la posición de esta, la pinta en ese punto de la pantalla.

De esta forma pintamos la descripción de cada objeto:

Render System: Pintamos las entidades con sus descripciones

```
void RenderSystem::renderWiki(EntityManager& EM, Map& map){
    float y{1.2f};
    DrawTexture(map.menu.wiki, 0, 0, WHITE);
    EM.forall([&](Entity&e){
        auto& render = EM.getCMPStorage().getRenderCMP(e);
        float pos_X  = HORIZONTAL_BORDER/3 + SPRITE_DIMENSIONS;
        float pos_Y  = (VERTICAL_BORDER ) + y * SPRITE_DIMENSIONS;
        DrawTextureRec(render.sprite, render.frame,
                       (Vector2){pos_X, pos_Y},WHITE);
        printDescription(EM, e, pos_X, pos_Y);
        y+=1.1f;
    });
}
```

Ya tenemos todo lo necesario, por lo tanto el bucle de este etado queda plasmado de esta forma:

Game: Estado "WIKI"

```
void Game::wikiScreen(Map& map){
    GM.createWikiEntities(EM);
    while (!IsKeyDown(KEY_ESCAPE) ){
        BeginDrawing();
        ClearBackground(RAYWHITE);
        rendSys.renderWiki(EM, map);
        EndDrawing();
        inpSys.checkKey();
    }
    state = States::MENU;
}
```

Creamos las entidades, y pintamos el fondo de pantalla, que encima tendrá todas las entidades con sus descripciones. Este bucle termina cuando pulsemos la tecla "ESC", por lo tanto tendremos que llamar a la función "checkKey()" del sistema de Input, para que controle ese comportamiento.

Si se pulsa la tecla "ESC" el estado cambia y pasa de nuevo a ser el estado "MENU". Esto implica controlar de nuevo las entidades, borrándolas ya que en el estado menú no existen entidades.

Añadiendo este estado, el método "run" de la clase "Game" se verá modificado, ya que tendrá otro caso más.

Game: run() case "WIKI"

```
void run(){

    ...

    case States::WIKI:
        wikiScreen(map);
        break;

    ...

}
```

Ahora tendremos un menú extra, con una interfaz para poder observar y aprender todo lo que el juego contiene, siendo el resultado el de la figura siguiente (figura 39).

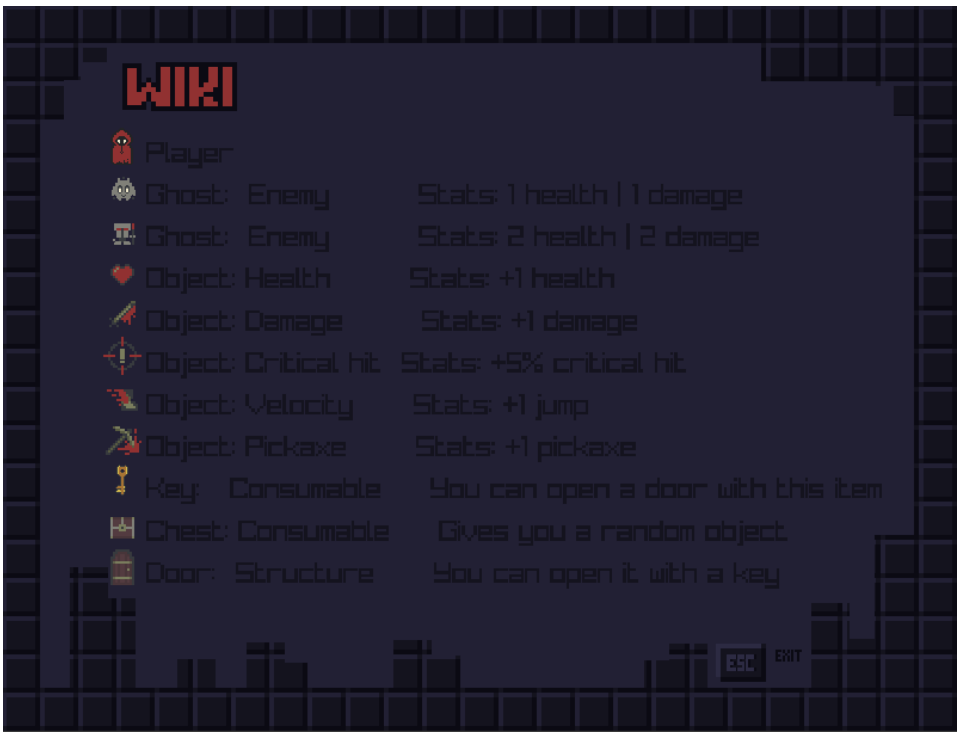


Figura 39. Esta figura representa el estado "WIKI".

7.10.2 Ejercicio: Visualización de golpe crítico

Este ejercicio, propone que al aplicar golpe crítico contra un enemigo, aparezca un sprite encima que lo represente, para que el jugador aprecie estos detalles.

Para ello recomiendo guardar la posición del enemigo que ha recibido golpe crítico y activar una variable para que el sistema de renderizado sepa cuando pintar y cuando no este sprite.

El sprite lo guardaremos en la clase Map.

Implementación. Creamos las variables en el "Game Manager"

Añadiremos dos variables para la posición y una para activar o desactivar el renderizado del golpe crítico.

Game Manager: Golpe crítico

```
struct GameManager{

    ...

    //GOLPE CRÍTICO
```

```
    int critHit{};
    int posx_crit{},posy_crit{};

    ...

};
```

A continuación nos dirigimos a donde se produce el golpe crítico, en el sistema de colisiones, específicamente a la colisión con un enemigo.

Una vez allí añadiremos al producirse golpe crítico la actualización de las posiciones creadas con las del enemigo y un incremento de la variable "critHit".

Sistema de colisiones: Guardamos la posición del golpeo

```
void CollisionSystem::collisionWithEnemy(EntityManager&
EM,GameManager& GM, Entity& player, Entity& enemy){

    ...

    if(rand() % 100 <= player_stats.critical_hit){
        enemy_stats.health = enemy_stats.health-player_stats.damage*2;
        GM.critHit++;
        GM.posx_crit = enemy_pos.posX;
        GM.posy_crit = enemy_pos.posY;

    ...

}
```

Por último, en el sistema de renderizado, añadimos una comprobación al final del método "update()" que compruebe si "critHit" es diferente a cero, y de ser así, renderizamos el golpe crítico.

Sistema de renderizado: Golpe crítico

```
void RenderSystem::update(EntityManager& EM,GameManager& GM,Map& map){

    ...

    if(GM.critHit!=0){
        renderCritHit(map, GM);
    }

}
```

Sistema de renderizado: Golpe crítico

```
void RenderSystem::renderCritHit(Map& map, GameManager& GM){
    float pos_X = (HORIZONTAL_BORDER - 16)+
                GM.posx_crit * SPRITE_DIMENSIONS;
    float pos_Y = VERTICAL_BORDER + GM.posy_crit * SPRITE_DIMENSIONS;
    DrawTexture(map.interface.crit_text,
    static_cast<float>(pos_X),
    static_cast<float>(pos_Y),
    WHITE);
    GM.critHit++;
    if (GM.critHit>25){
        GM.critHit=0;
    }
}
```

Añadiremos una comprobación final para que se renderice en más de una iteración por golpe crítico, para que al usuario le de tiempo a ver lo sucedido.

Calculamos la posición, pintamos la textura en dicha posición y comprobamos si ha acabado el tiempo de renderizado. Si ha acabado aplicamos el valor 0 a la variable y no se pintará más.

7.11 Resumen de lo aprendido

Hemos llegado al final del proyecto. Se hizo extenso, pero lo hemos conseguido. Vamos a ver que hemos aprendido en este proyecto y posteriormente visualizaremos el resultado final.

Este proyecto lo hemos separado en dos partes bien diferenciadas. Por un lado el motor de entidades y las diferentes estructuras para los datos. Por otro lado hemos desarrollado el juego con el motor creado.

Respecto al motor de entidades hemos conseguido los siguientes objetivos:

- Hemos creado un Slotmap, con el objetivo de sacar los componentes de las entidades y almacenarlos por separado en estructuras creadas para ellos. Con los Slotmaps hemos optimizado mucho la cantidad de datos que almacenamos y manejamos.
- Hemos creado etiquetas y componentes con máscaras que nos permiten trabajar de una forma más rápida y con comprobaciones sencillas a lo largo de todo el desarrollo del juego.
- Hemos implementado un manejador de componentes a la vez que su almacenamiento en la clase "Component Storage", que nos facilita la vida a la hora de crear componentes y asociarlos a las entidades.

- Nuestras entidades ahora contienen sus máscaras y sus llaves para acceder a los componentes que tengan asociados.
- Por último un manejador de entidades ampliado a como veníamos haciendo, con un método que selecciona entidades según los componentes o tags que tienen.

Un resultado muy bueno, que optimiza y mejora los motores que hemos ido construyendo con el paso de los proyectos. En un futuro tomo de este libro, habrá mejoras incluso para este último motor, añadiendo elementos que se ejecuten en tiempo de compilación y metaprogramación para lograr más optimización.

En cuanto al juego, hemos realizado diferentes mejoras respecto a otros anteriores.

- Hemos creado una estructura "Game Manager" para tener el juego organizado, con funciones para crear niveles tanto de forma manual como aleatoria.
- Hemos controlado el flujo de entidades que creamos y destruimos en cada instante, optimizando el espacio.
- Hemos creado sistemas con un funcionamiento diferente, usando las nuevas funciones del manejador de entidades y las máscaras de entidades y componentes.
- Hemos creado objetos que modifican las estadísticas de nuestro personaje mediante componentes, de forma sencilla, y un inventario de objetos.

Un juego completo que es mejorable ya que el objetivo era compartir el proceso y los pasos para llegar hasta aquí, un punto en el que tener un proyecto jugable y entretenido. Esto no quiere decir que sean perfectos, ya que puedes pensar en mejoras para nuestros juegos y prototipos.

Podemos mejorar añadiendo animaciones, o alguna técnica o interfaz para ver la vida y el daño de los enemigos en cada momento, o de los objetos que golpeamos como las rocas. También podemos añadir objetos nuevos que combinen estadísticas u objetos malditos que las resten, y mucho más. El objetivo lo hemos cumplido, y ahora puedes mejorar el resultado.

Este proyecto lo puedes encontrar en la referencia Cantó-Berná (2023e).

7.12 Resultado visual de "The Final Room"

Ha quedado un juego bonito, con un personaje principal, enemigos, objetos, estructuras y más. Las figuras siguientes, muestran el resultado visual y final de este proyecto.

En esta lista, quedan expuestas las diferentes fases del juego.

- La figura 40, muestra la pantalla inicial y menú del juego.
- La figura 41, muestra la pantalla de WIKI o biblioteca, donde podemos consultar datos de cualquier elemento que aparece en el juego.
- La figura 42, muestra el nivel 1 del juego, generado manualmente.

- La figura 43, muestra el nivel 4 del juego, generado aleatoriamente.
- La figura 44, muestra la pantalla final o de muerte.

Figura 40. "The Final Room": Menú del juego

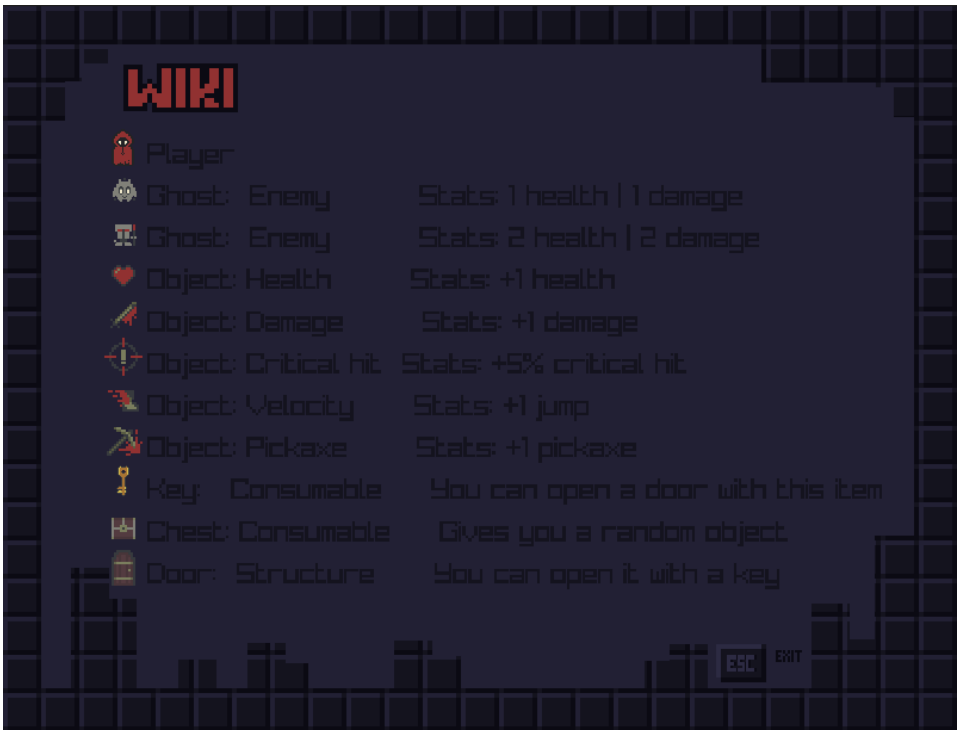


Figura 41. "The Final Room": WIKI o biblioteca

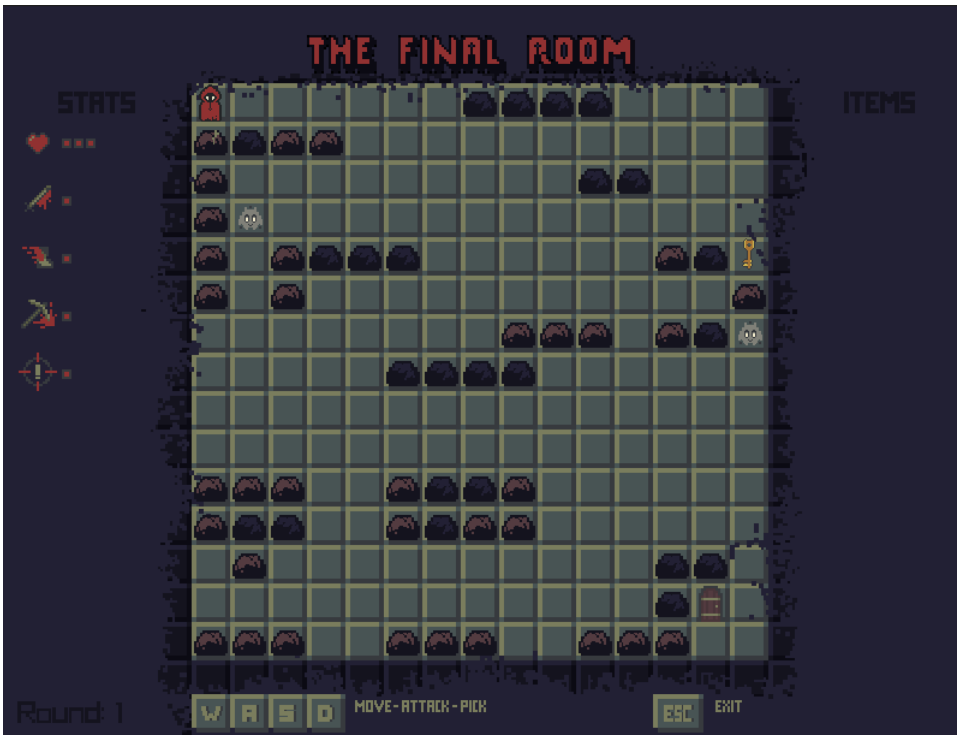


Figura 42. "The Final Room": Nivel 1 creado a mano

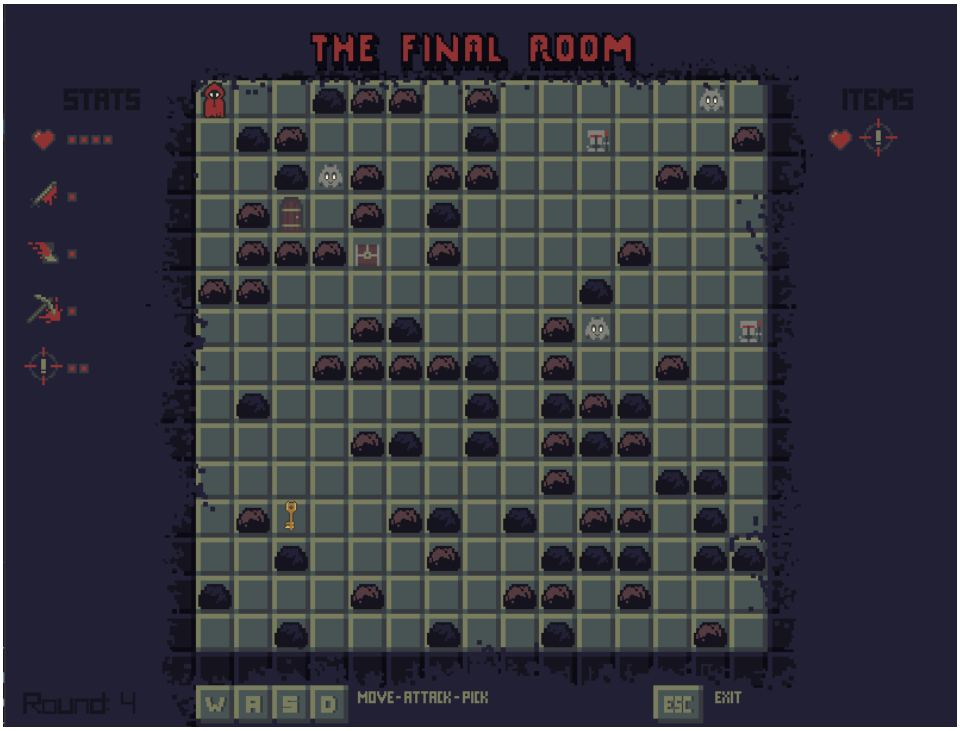


Figura 43. "The Final Room": Nivel 4 creado de forma automática

Figura 44. "The Final Room": Pantalla final

8 Recapitulamos el viaje

¡Has completado la guía! Si has llegado hasta aquí, probablemente significa que has obtenido todos los conocimientos de este libro.

En este punto tienes que poder hacer tu propio motor ECS y tus propios juegos, optimizando en gran medida el flujo de datos y su manejo.

Han sido cinco proyectos, unos más largos que otros, pero en los que hemos aprendido a hacer muchas cosas y vamos a mencionar las más importantes de cada uno de los proyectos, para recapitular.

- Primer proyecto: En el primer proyecto, "Starfield", nos pusimos en contacto con el patrón ECS y creamos nuestras primeras entidades, compuestas de componentes y nuestros primeros sistemas, con un manejador de entidades sencillo, donde solo había un solo tipo de entidades.
- Segundo proyecto: En el segundo proyecto, "Firefighter Game", teníamos la primera diferenciación de entidades, añadiendo un booleano que nos permitía saber que entidad elegir en ciertos momentos. Disponíamos de entrada por teclado, aunque aún estábamos comprendiendo cómo funcionaba la arquitectura Entity Component System.
- Tercer proyecto: El tercer proyecto, fue el mismo que el segundo pero donde implementamos RayLib, y nos sirvió de toma de contacto con esta librería. Por otro lado, observamos como podemos usar el mismo código y solo cambiar un componente y un sistema, para conseguir un juego con gráficos. Esto gracias a la flexibilidad y facilidad para cambiar y crear nuevos elementos con este motor.
- Cuarto proyecto: El cuarto proyecto, "Save the Ovni", fue un proyecto donde necesitábamos diferenciar componentes con más frecuencia que en el juego anterior, por lo que usamos "std::optional" para solucionar nuestro problema. Avanzamos un poco más en la optimización del código y creamos nuevos sistemas como el de colisiones, y estados para manejar nuestras pantallas. También creamos la funcionalidad de eliminar entidades.
- Quinto proyecto: El último proyecto, "The Final Room", un juego completo donde creamos nuevas estructuras para tener entidades libres, y componentes agrupados por tipo en Slotmaps, para optimizar el espacio y el manejo del juego. Creamos un gestor para el almacenamiento y nuevas funcionalidades para el manejador de entidades, como un buscador por componentes y etiquetas. Además, creamos un manejador para el juego, optimizando el flujo de entidades y el manejo y reutilización del espacio en memoria. Hemos obtenido un juego con un avance en optimización muy grande respecto a proyectos anteriores.

Ahora eres capaz de dar el siguiente paso, y crear estos motores para tus juegos. Pero, aún podemos mejorarlos.

8.1 Aprendizaje futuro

Este ha sido el resultado del primer tomo de "Cómo desarrollar un GameEngine Entity-Component-System en C++20". La idea es avanzar y desarrollar un segundo tomo, donde mejoremos la optimización, y nos metamos de lleno en otras técnicas para optimizar las diferentes partes del motor.

La idea principal era llegar mucho más lejos, pero hemos avanzado poco a poco para entender y saber qué es lo que hacemos en cada momento. Por lo tanto, el final obtenido es muy bueno para cerrar el primer libro.

Todos los juegos realizados en este proyecto son mejorables y os animo a que os lancéis a intentarlo, mientras se cocina el siguiente nivel.

Además en un futuro será posible llevar a cabo adaptaciones a otros lenguajes, o a otros estándares, etc. También es posible desarrollar este libro con focos mas centrados en optimizar, con usos más avanzados de "templates", todo esto podremos incluirlo en este segundo tomo mencionado anteriormente.

Bibliografía

Alicia, A. V. (2023). *Sprites for "the final room".* Retrieved from `https://archive.org/details/frsprites`

Cantó-Berná, L. (2023a, February). *Ecs project: Starfield.* Retrieved from `https://archive.org/details/starfield_202302`

Cantó-Berná, L. (2023b, February). *Entity component system project: Firefighter game.* Retrieved from `https://archive.org/details/firefighter-game`

Cantó-Berná, L. (2023c, March). *Entity component system project: Firefighter game with raylib.* Retrieved from `https://archive.org/details/firefighter-rl`

Cantó-Berná, L. (2023d, March). *Entity component system project: Save the ovni.* Retrieved from `https://archive.org/details/save-the-ovni`

Cantó-Berná, L. (2023e, May). *The final room game.* Retrieved from `https://archive.org/details/froom`

Gallego-Durán, F. J. (2021, May). *Basic linux terminal library.* Retrieved from `https://archive.org/details/bltl_20210507`

Santamaria, R. (2013). *Raylib: A simple and easy-to-use library to enjoy videogames programming.* Retrieved from `https://www.raylib.com/index.html`