



Grupo 3 y 4
Ingeniería informática - Eina
Videojuegos 2024/2025
21 de junio de 2025

Jorge Fernández Marín (839113)
Javier Julve Yubero (840710)
Marcos García Ortega (844419)
Saul Caballero Luca (848431)
Aroa Redondo Zamora (851769)

ÍNDICE

1. INTRODUCCIÓN.....	4
2.MOTOR.....	4
2.1 ESTRUCTURA DEL MOTOR.....	4
2.2 FUNCIONAMIENTO INTERNO.....	5
2.2.1 CORE.....	5
2.2.2 MANAGERS.....	7
2.2.3 COMPONENTS.....	10
2.2.4 UTILS.....	16
2.3 DEPENDENCIAS.....	16
2.4 DOCUMENTACIÓN.....	17
3.RESOURCES.....	18
4.PERSONAJE.....	19
5.ENEMIGOS.....	20
5.1 AXE MAX.....	20
5.2 BATTON BONE.....	20
5.3 BOMB BEEN.....	21
5.4 DIG LABOUR.....	21
5.5 FLAMMINGLE Y SNOW SHOOTER.....	22
5.6 SPIKY.....	22
5.7 RAYBIT.....	22
5.8 TOMBOT.....	23
5.9 SKY CLAW.....	23
5.10 SCRAP ROBO.....	23
5.11 METAL.....	24
5.12 JAMMINGER.....	24
5.13 HOGANMER.....	25
6. JEFES.....	26
6.1 CHILL PENGUIN.....	26
6.2 FLAME MAMMOTH.....	28
7. PROBLEMAS Y SOLUCIONES.....	30
7.1 MANEJO DE SPRITES.....	30
7.2 PROBLEMAS CON LA ALINEACIÓN DE SPRITES.....	30
7.3 PROBLEMA CON EL WALL JUMP.....	31
7.4 PROBLEMAS DE COLISIONES.....	32
8. DECISIONES FINALES.....	32
9.DESARROLLO DEL JUEGO.....	33
9.1 TIEMPO EMPLEADO:.....	33
9.2 TAREAS REALIZADAS:.....	33

10. ANEXO.....	34
1. DESCRIPCIÓN GENERAL.....	34
1.1 RESUMEN.....	34
1.2 GÉNERO.....	34
1.3 JUGADORES.....	34
1.4 AUDIENCIA OBJETIVO.....	34
1.5 LOOK AND FEEL.....	34
2. HISTORIA.....	36
3. GAMEPLAY.....	37
3.1 OBJETIVOS Y LÓGICA DEL JUEGO.....	37
3.2 MECÁNICAS.....	37
3.2.1 MOVIMIENTO.....	37
3.2.2 ATAQUE.....	37
3.2.3 OBSTÁCULOS.....	37
3.2.4 GESTIÓN DE RECURSOS (DROPS).....	37
3.3 REGLAS.....	38
3.3.1 REGLAS MEGAMAN.....	38
3.3.2 REGLAS DEL JUEGO.....	38
3.3.3 REGLAS DE JUGABILIDAD.....	38
3.4 PROGRESIÓN.....	39
3.4.1 NIVEL 1 - CHILL PENGUIN.....	39
3.4.2 NIVEL 2 - FLAME MAMMOTH.....	39
4.4.3 CÓMO TERMINA EL JUEGO.....	39
4. PERSONAJES.....	40
4.1 PERSONAJE JUGABLE.....	40
4.1.1 ARMA.....	40
4.2 ENEMIGOS.....	40
4.2.1 ENEMIGOS NORMALES.....	40
4.2.2 BOSS FINAL.....	42
4.3 DROPS.....	43
5. INTERFACES.....	44
5.1 MENÚ PRINCIPAL.....	44
5.2 MENÚ DE OPCIONES.....	44
5.3 MENÚ ELECCIÓN BOSS.....	44
5.4 MENÚ PAUSA.....	45
6. MÚSICA.....	46
7. REQUERIMIENTOS DE DESARROLLO.....	46
8. MARKETING Y PRESUPUESTO.....	47
8.1 MARKETING.....	47
8.2 PRESUPUESTO.....	47

9.BIBLIOGRAFÍA.....	48
---------------------	----

TECHNICAL DESIGN DOCUMENT

1. INTRODUCCIÓN

El clon de mega man ha sido desarrollado en su totalidad en c++, con la ayuda de la biblioteca SDL(Simple DirectMedia Layer), una biblioteca que facilita el desarrollo de aplicaciones multimedia en c++. Esta biblioteca ofrece herramientas muy variadas, entre ellas herramientas para la creación de ventanas, renderizado de sprites, también permite tener control sobre los diferentes inputs del sistema y otro tipos de eventos como redimensiones de la ventana. De esta biblioteca se ha elegido su versión 2.0 que es la versión moderna y la más potente.

2.MOTOR

Ahora se hablará de cómo se ha desarrollado el motor del juego. Para el desarrollo del clon de megaman se ha creado un motor totalmente desde 0 separado de la implementación de las mecánicas y entidades del juego. Al igual que el resto del proyecto el motor está completamente escrito en c++. Este hace uso de la herramienta Cmake para compilar y linkear dependencias del proyecto.

2.1 ESTRUCTURA DEL MOTOR

La estructura del motor se basa en la arquitectura **ECS**(Entity-Component-System), aunque se le han añadido unas ligeras modificaciones. **ECS** es un patrón arquitectural muy usado en el mundo de los videojuegos y sirve para representar los objetos del juego. Esta arquitectura se divide en tres partes fundamentales: entidades, componentes y sistemas.

- **Entidades:** son las encargadas de contener los componentes, no tienen ni datos ni comportamiento. Son diferenciadas por un identificador único, que va incrementando cada vez que se crea una entidad. Ejemplo: el personaje principal.
- **Componentes:** son contenedores de datos que se usan para modelar propiedades de las entidades. Cada componente describe una característica de una entidad. Ejemplo: el componente que se encarga de guardar la vida restante del jugador.
- **Sistemas:** son procesos que actúan sobre las entidades que tienen cierto conjunto de componentes. Se encargan de ejecutar lógica usando la información de los componentes. Ejemplo: el sistema de físicas.

Esta arquitectura ha permitido tener gran modularidad en el motor y ha facilitado la reutilización de los módulos. Además aumenta la eficiencia en memoria ya que **ECS** permite crear una tabla donde se enlacen las entidades con los componentes de manera ordenada y secuencial, lo que permite que los sistemas tengan un acceso rápido a estos componentes.

	Entity 1	Entity 2	Entity 3
Component 1	[Data]		
Component 2	[Data]	[Data]	[Data]
Component 3		[Data]	[Data]

En el caso del motor desarrollado se ha permitido implementar lógica en los componentes para facilitar el desarrollo.

Para el control de la creación de entidades y ejecución de la lógica del juego se ha creado un sistema de escenas. Una escena es una estructura que contiene sus propios sistemas y un conjunto de entidades. Los sistemas de una escena operan únicamente con las entidades de la misma, no operan con entidades de otras escenas, las entidades de una escena no pueden colisionar con entidades de otra escena distinta. El hecho de trabajar con estas escenas permite tener control sobre las entidades que se cargan y descargan en tiempo de ejecución.

2.2 FUNCIONAMIENTO INTERNO

El motor se divide en 4 grandes partes: core, managers, components y utils.

2.2.1 CORE

Define las bases de la ejecución de la lógica del juego, se divide en dos módulos: Engine y Systems.

- **Engine:** es el núcleo del motor, se encarga de inicializar la ventana donde se renderizan los sprites, gestionar el bucle principal del juego, así como la gestión de eventos y el cierre de los sistemas del juego al terminar la ejecución. También se encarga de gestionar las escenas guardando una instancia de SceneManager, y permite un acceso global al motor mediante un patrón singleton.

```
while (running) {
    int currentTime = SDL_GetTicks();
    float deltaTime = ((currentTime - lastTime) - delayInputs) / 1000.0f;
    lastTime = currentTime;

    handleEvents();
    update(deltaTime);
    render();
    InputManager::resetInputState();

    frameCount++;

    if (currentTime - fpsTimer >= 1000) {
        int fps = frameCount;
        frameCount = 0;
        fpsTimer = currentTime;

        Debug::Log("FPS: " + std::to_string(fps));
    }
}
```

- **Systems:** define los principales sistemas con los que cuenta el motor, estos son: el sistema de físicas, el sistema de colisiones y el sistema de renderizado.

El sistema de físicas se encarga de aplicar físicas las entidades del juego que tengan los componentes rigidbody y transform. Para aplicar las físicas este sistema aplica una fuerza de gravedad a todas las entidades que no sean kinematic, además se encarga de actualizar la posición de las entidades según su velocidad y calcula las fuerzas que se aplicarán cada entidad una vez por ciclo. El sistema de colisiones se encarga de detectar y resolver las colisiones entre varias entidades que tengan el componente collider. Este sistema diferencia entre entidades dinámicas, se pueden mover, y entidades estáticas, no se moverán en el transcurso de la partida. Por cada frame se recogen todos los colliders de la escena y se organizan en una estructura BVH, posteriormente se comprueba para cada collider dinámico si colisiona con otro collider con ayuda del BVH. Si se encuentra una colisión se encuentra el eje que sigue esta, se corrige la entidad para que no se produzca solapamiento y se anula la velocidad en el eje de la colisión, además se actualizan los puntos de colisión de ambos colliders. Por último el sistema de renderizado se encarga de dibujar todas las entidades y elementos visuales en pantalla usando SDL, respetando la cámara, el escalado y el orden de renderizado. En cada frame obtiene la cámara activa y renderiza los fondos con parallax, luego las entidades y posteriormente la interfaz del usuario. Para cada entidad que no sea un UI component se calcula la posición en pantalla relativa a la cámara y el

escalado, se ajusta la posición del sprite según el punto de anclaje de este que puede ser el centro o esquina superior izquierda y por último se dibuja el sprite con la textura, rotación y transparencia configurados

2.2.2 MANAGERS

Esta parte del motor contiene los módulos responsables de la gestión y coordinación de aspectos clave del juego como las escenas, audio o las entidades. Su propósito general es desacoplar la lógica de gestión del resto del motor facilitando la organización de los elementos comentados anteriormente. Los managers implementados son:

- **Audio Manager:** gestiona todo lo relacionado con el audio del juego (efectos de sonido y música). Permite cargar, reproducir, pausar, reanudar y detener los sonidos, así como ajustar el volumen. Usa `SDL_mixer`, que es una biblioteca complementaria de `SDL2` y permite gestionar el audio. Sigue un patrón singleton para dar acceso global.
- **Input Manager:** se encarga de gestionar toda entrada proporcionada por el usuario, tanto del teclado como de el ratón. Además controla el estado de los inputs mediante la detección de eventos `SDL`.
- **Dynamic input Manager:** este módulo permite asociar acciones a diferentes inputs del usuario, de este modo las acciones no están hardcodeadas como inputs predeterminados. Este manager permite cambiar qué inputs están asociados a según qué acción, dando más libertad al usuario a la hora de elegir con qué teclas jugar, mejorando así la experiencia de juego. Por poner un ejemplo de su uso, en vez de hacer que la tecla espacio ejecute la lógica de saltar, lo cual haría que la tecla espacio este hardcodeada, puedes crear el código "Saltar" enlazado a la tecla espacio, y mediante esta clase, comprobar el input "Saltar", lo que hace más claro el código, y permite al jugador cambiar la tecla espacio por la tecla W, por ejemplo.
- **Entity Manager:** se encarga de gestionar las entidades, las crea, las elimina y permite el acceso a ellas. También se encarga de añadir y eliminar componentes de una entidad.
- **Prefab Manager:** se encarga de la lectura de los prefabs y haciendo uso del Entity Manager crea una instancia de ellos. Un prefab contiene los datos iniciales de los componentes que pertenecen a una entidad. En el caso del motor se modelan como ficheros json, un ejemplo es:


```
{
  "components": {
    "Transform": {
      "posX": 0,
      "posY": 0,
      "rotZ": 0,
      "sizeX": 2.5,
      "sizeY": 2.5,
      "anchor": "CENTER"
    },
    "Collider": {
      "tag": "Enemy",
      "scaleX": 20,
      "scaleY": 20,
      "isTrigger": true
    },
    "Rigidbody": {
      "velocityX": 0,
      "velocityY": 0,
      "mass": 0,
      "gravity": 0,
      "drag": 0.00,
      "bounceFactor": 0.00,
      "kinematic": false
    },
    "Sprite": {
      "texturePath": "../sprites/enemies/pipeturn/rotate/rotate_0.png",
      "anchor": "CENTER"
    },
    "Animator": {
      "parameters": {
      },
      "states": {
        "rotate": {
          "looping": true,
          "folder": "../sprites/enemies/pipeturn/rotate",
          "duration": 0.2
        }
      },
      "transitions": {
      },
      "initialState": "rotate"
    },
    "PipeturnController": {}
  }
}
```

En este ejemplo se observan los diferentes componentes que tendrá la entidad, así como sus datos.

- **Raycast Manager:** un raycast es una técnica usada en el mundo de los videojuegos que consiste en lanzar un rayo desde un punto para ver si ese rayo colisiona con alguna entidad del entorno. Un rayo está formado por su punto de origen y su dirección, a la hora de detectar una colisión se tiene en cuenta la distancia que debe recorrer ese rayo. Este manager se encarga de gestionar estos rayos en el motor desarrollado, lanzar los rayos, comprobar si colisionan y devolver datos de la colisión como son el collider con el que ha colisionado, el punto de colisión la distancia desde el origen del rayo hasta el punto de colisión y la entidad propietaria del collider.
- **Render Text Manager:** se encarga de mostrar texto por pantalla, lo primero es cargar la fuente, en el caso de este motor las fuentes están almacenadas en una carpeta con imágenes de cada carácter, cada fuente tiene su propia carpeta. Una vez elegida la fuente según el texto que se quiera mostrar su posición y tamaño se coloca y redimensiona las imágenes ligadas a cada carácter de dicha fuente.
- **Escenas:** estas son algo especiales, están gestionadas por un manager, pero están divididas en un submódulo. Este submódulo consta de tres partes: Scene, SceneLoader y SceneManager. **Scene** representa una escena individual, con su propio Entity Manager y su sistema de físicas, colisiones y renderizado. También se encarga de ejecutar la lógica de todos los componentes de las entidades que forman dicha escena, esta ejecución se divide en 3 funciones: **start()** se encarga de inicializar los componentes y sistemas, **update()** que se encarga de ejecutar la lógica de los componentes y sistemas una vez por frame y **render()** que ejecuta el

código encargado de pintar las sprites en la pantalla una vez por frame. Luego está el **SceneLoader** que se encarga de cargar las diferentes escenas, esta carga puede ser síncrona a la hora de inicializar el juego, o asíncrona para cargar escenas en tiempo de ejecución. Al igual que los prefabs las escenas también se cargan en fichero json, los cuales contienen las entidades y sus posiciones en el espacio 2d, además de la cámara. Las entidades pueden estar hardcodedas en este json o se le puede hacer referencia al prefab de la entidad, lo cual permite un json más limpio y legible.

```
{
  "name": "test",
  "entities": [
    {
      "components": {
        "Transform": {
          "posX": 100,
          "posY": 100,
          "rotZ": 0,
          "sizeX": 1,
          "sizeY": 1,
          "anchor": "CENTER"
        },
        "Camera": {
          "zoom": 1,
          "windowWidth": 800,
          "windowHeight": 600
        }
      }
    },
    {
      "prefab": "megaman"
    },
    {
      "components": {
        "Transform": {
          "posX": 0,
          "posY": 300,
          "rotZ": 0,
          "sizeX": 1,
          "sizeY": 1,
          "anchor": "CENTER"
        },
        "Collider": {
          "tag": "Terrain",
          "scaleX": 750,
          "scaleY": 100,
          "isTrigger": false,
          "tag": "Terrain"
        },
        "Rigidbody": {
          "velocityX": 0,
          "velocityY": 0,
          "mass": 10000,
          "gravity": 9.8,
          "drag": 0.01,
          "bounceFactor": 0.01,
          "kinematic": true
        },
        "Sprite": {
          "texturePath": "suelo.png",
          "anchor": "CENTER"
        }
      }
    }
  ]
}
```

Por último está el **Scene Manager** que es el módulo que se encarga de gestionar las escenas. Solo hay una única instancia de este módulo en el motor, y almacena la escenas, hace uso del Scene Loader para cargarlas y es el encargado de ejecutar las funciones **start()**, **update()** y **render()** de las escenas.

2.2.3 COMPONENTS

Describe cómo deben de crearse los componentes, para la gestión de estos tanto en ejecución como en carga dinámica desde ficheros, estos implementan la siguiente estructura:

```
struct TParameters {
    //Parametros necesarios para pasar a T
    TParameters(Transform* t) : transform(t) {}
};

struct T : public Component {
    //Parametros necesarios para el componente
    T(TParameters t) {}

    void start() override { }

    void update(float deltaTime) override { };
};

class TLoader {
public:

    static TParameters fromJSON(const nlohmann::json& j, EntityManager& entityManager) {
        //Si el componente requiere el Transform de la entidad en la que esta
        Transform* transform = entityManager.getComponent<Transform>(entityManager.getLast());
        if (!transform) {
            throw std::runtime_error("T requires a Transform component");
        }

        TParameters params();

        return params;
    }

    static T createFromJSON( nlohmann::json& j, EntityManager& entityManager) {
        return T(fromJSON(j, entityManager));
    }
};
```

El **Loader** será la clase encargada de leer de un fichero en formato json todos los componentes que se van a cargar, estos son almacenados en la estructura Parameters para luego que puedan ser usados por la estructura principal.

Además de esto se han declarado unos componentes base, los cuales son:

- **Transform:** gestiona la posición, rotación y escala. Es un componente obligatorio en cualquier entidad, ya que se usa para situar las entidades en un espacio 2D.
- **RigidBody:** contiene la información necesaria para que se puedan aplicar las físicas al componente. Entre esta información se encuentra la velocidad, masa, gravedad, aceleración o si al componente le afectan las físicas o no.
- **Collider:** define el área de colisión de una entidad, permitiendo así detectar y solucionar colisiones entre diferentes entidades. El motor permite tanto colisiones AABB (cajas alineadas con los ejes) como colisiones con triángulos.
- **Sprite Renderer:** este componente contiene toda la información necesaria para renderizar un sprite, como por ejemplo la imagen que este tiene. Tanto la posición como el tamaño han sido previamente definidos en el componente transform.
- **Camera:** define las propiedades necesarias para renderizar una escena por pantalla. Haciendo uso de datos como el tamaño de la ventana o los datos del Transform de su entidad se puede gestionar el área visible por pantalla. Este

componente permite definir que la cámara siga a una entidad en concreto o se mueva libremente por el mundo del juego.

- **Animator:** este componente funciona como una máquina de estados. Cada estado corresponde a una animación, que es un conjunto de sprites que se muestran continuamente para dar sensación de movimiento. Como en una máquina de estados convencional cada estado cuenta con sus transiciones, estas transiciones definen el estado al que toca transicionar y la condición para hacerlo. Estas condiciones se definen mediante atributos generados dinámicamente en función de cada transición. Este componente se define como un objeto json de la siguiente forma:

```
"Animator": {
  "parameters": {
    "die": {"type": "bool", "value": false}
  },
  "states": {
    "spin": {
      "looping": true,
      "folder": "./sprites/enemies/spiky/spin",
      "duration": 0.1
    },
    "die": {
      "looping": false,
      "folder": "./sprites/enemies/spiky/die",
      "duration": 0.08
    }
  },
  "transitions": [
    {
      "from": "spin",
      "to": "die",
      "conditions": [
        { "parameter": "die", "mode": "EQ", "value": 1 }
      ]
    }
  ],
  "initialState": "spin"
},
```

Los **parameters** son los atributos que se usan en las condiciones de las transiciones. Los **states** representan los estados del autómata, cada estado representa una animación que cuenta con diferentes atributos, entre ellos si la animación se debe repetir en bucle, en que carpeta se encuentra la animación y el tiempo en segundos que dura la animación. En **transitions** se encuentran definidas las transiciones, el estado inicial, al que se va a transicionar y las condiciones necesarias para hacerlo, las cuales se evalúan como comparaciones booleanas. Por último se encuentra el estado inicial que tendrá la entidad.

- **UI Component:** este componente permite definir que la entidad que lo contiene no es un elemento del mundo del juego. Esto hace que no se vea afectada por el transform de la cámara sino por el espacio en pantalla.

```
"components":{
  "Transform": {
    "posX": 0.5,
    "posY": 0.5,
    "rotZ": 0,
    "sizeX": 2.64,
    "sizeY": 2.64,
    "anchor": "CENTER"
  },
  "UIComponent": {
  },
  "Sprite": {
    "texturePath": "sprites/flamemammoth/start/start_0.png"
    "anchor": "CENTER"
  },
}
```

El transform de un componente de este tipo funciona diferente, ya que las posiciones no se toman como absolutas, sino que indican el porcentaje de la pantalla en el que se coloca el objeto. Es decir, el componente de la imagen anterior se renderiza en el punto que coincide con el 50% de la x de la pantalla y el 50% de la y de la pantalla, en medio. En el caso de ser posX: 0 y posY:0 el objeto se renderiza en la esquina superior izquierda.

- **Parallax Layer:** gestiona capas de fondo con efecto parallax, creando profundidad visual moviendo fondos a diferentes velocidades.
- **IA:** es un submódulo que permite la implementación de dos tipos de IAs diferentes.
 - **Classic IA:** se basa en un máquina de estados cuyos estados son las acciones y tiene transiciones entre ellos. A diferencia del animator, donde todos los estados y transiciones se definen desde json, aquí se marcan los estados y transiciones a nivel esquemático desde json, pero luego las acciones a ejecutar y la lógica de transiciones se definen desde el código de un componente.

```
"ClassicIA": {
  "states": [
    {
      "name": "Idle",
      "transitions": [
        {"nextState": "Jump"},
        {"nextState": "ShootLava"},
        {"nextState": "ShootFire"},
        {"nextState": "ChangeDirection"},
        {"nextState": "Death"}
      ]
    },
    {
      "name": "Jump",
      "transitions": [
        {"nextState": "Idle"},
        {"nextState": "Death"}
      ]
    },
    {
      "name": "ShootLava",
      "transitions": [
        {"nextState": "Idle"},
        {"nextState": "Death"}
      ]
    },
    {
      "name": "ShootFire",
      "transitions": [
        {"nextState": "Idle"},
        {"nextState": "Death"}
      ]
    },
    {
      "name": "ChangeDirection",
      "transitions": [
        {"nextState": "Idle"},
        {"nextState": "Death"}
      ]
    },
    {
      "name": "Death",
      "transitions": []
    }
  ]
}
```

Esta es la definición de una IA clásica en formato json.

```
iaComponent->registerAction("Death", [this](EntityManager *entityManager, float deltaTime)
{
    Debug::Log("Death");
    death(); });
```

De este modo se crea la lógica para ejecutar en cada estado.

```
iaComponent->registerCondition("Jump", "Death", [this](EntityManager *entityManager, float deltaTime)
{ return healthComponent->isDead(); });
```

Este es un ejemplo de código para definir cómo se va a transicionar entre dos estados, se transiciona si la función devuelve un valor booleano igual a true.

- **Reinforcement IA:** se basa en una IA que tiene aprendizaje por refuerzo, el objetivo es que una entidad pueda aprender y tomar decisiones mediante la técnica del Q-Learning. Esta técnica se basa en aprender una función Q que estima la recompensa a largo plazo al realizar una acción determinada en un estado determinado. Esta IA se define en los prefabs de la entidad de la siguiente manera:

```
"ReinforcementIA": {  
  "states": [  
    { "name": "Idle", "actions": ["Jump", "ShootFire", "ShootLava", "Stay"] }  
  ],  
  "learning_rate": 0.1,  
  "discount_factor": 0.9,  
  "epsilon": 0.2  
}
```

Consta de varias partes, **states** se encarga de definir los diferentes estados que va a tener la entidad, además de las diferentes acciones que se pueden realizar en ese estado. **Learning rate** es la velocidad a la que aprende la IA, para valores altos aprende más rápido y para valores bajos más lento. **Discount factor** contiene el valor que le va dar la IA a recompensas inmediatas, para valores altos la IA no solo tendrá en cuenta las recompensas inmediatas, también tendrá las futuras, para valores bajo la IA premiará sobre todo las recompensas inmediatas. Por último **epsilon** determina el porcentaje de veces que la IA realizará una acción aleatoria para explorar nuevas posibilidades, si el valor es 0,2 habrá un 20% de probabilidad de que la IA elija una acción al azar.

Para implementarla en las entidades se ha hecho uso de las siguientes funciones ubicadas en el módulo de IA avanzada del motor:

```
void registerAction(const std::string& state, const std::string&  
actionName, std::function<void(EntityManager*, float)> callback);
```

Esta función se encarga ejecutar la función callback cuando desde el estado state se transiciona a la acción actionName.

```
void registerRewardFunction(std::function<float(EntityManager*,  
const std::string&, const std::string&, float)> callback);
```

A la función callback se le pasa tanto el estado como la acción, y devuelve el valor de la recompensa que se le da a la IA.

```
void saveQTable(const std::string& filename);
```

Esta función se usa para guardar el aprendizaje en una tabla, que tiene la siguiente pinta:

```
"q_table": [  
  {  
    "action": "Stay",  
    "q_value": -299.998291015625,  
    "state": "Idle"  
  },  
  {  
    "action": "Blizzard",  
    "q_value": -299.998291015625,  
    "state": "Idle"  
  },  
  {  
    "action": "Ice",  
    "q_value": -299.998291015625,  
    "state": "Idle"  
  },  
  {  
    "action": "Shoot",  
    "q_value": -299.998291015625,  
    "state": "Idle"  
  },  
  {  
    "action": "Dash",  
    "q_value": -299.998291015625,  
    "state": "Idle"  
  },  
  {  
    "action": "Jump",  
    "q_value": -299.998291015625,  
    "state": "Idle"  
  }  
]
```

Aquí se observa tanto las acciones que tiene por estado como el valor de cada acción.

2.2.4 UTILS

Este apartado del motor define una serie de módulos que no son estrictamente necesarios para el funcionamiento de este, pero mejoran tanto la usabilidad como el rendimiento de este. Los módulos son:

- **BVH:** proporciona una implementación de la estructura BVH. BVH (Bounding volume hierarchy) es una estructura en árbol que se usa para acelerar la intersección entre colliders y rayos en la escena. En esta implementación el árbol es binario. En cada frame, se construye el BVH con los colliders de la escena, haciendo que cada nivel del árbol englobe 2 nodos, teniendo cada uno la mitad de los colliders en ese nodo. El nodo raíz contiene englobado todos los colliders de la escena. Además, cada nodo contiene el AABB resultante que engloba todos los colliders de su nodo. A la hora de comprobar colisiones, en vez de testear con todos los colliders, se testean con los AABBs de los nodos. Si intersecciona, se profundiza por el nodo, sino, se descarta todos los subnodos de ese nodo, lo que permite ir descartando la mitad de los nodos en cada nivel del árbol, pasando de una complejidad $O(n^2)$ a $O(n \cdot \log(n))$.

Este BVH lo usa el sistema de colisiones, haciendo 2 instancias: uno para elementos dinámicos(su posición cambia con el tiempo), que se construye en cada frame, y otro para elementos estáticos(su posición es fija), que sólo se construye al inicio, lo que optimiza aún más el sistema de colisiones.

- **Debug:** este módulo implementa una serie de métodos para la escritura de mensajes en ficheros de texto. Esto permite la visualización del estado del juego y del motor a posteriori para la detección de fallos en el desarrollo del proyecto y facilita la depuración del código.

2.3 DEPENDENCIAS

Para el desarrollo del motor, se han usado 4 dependencias externas, 3 de las cuales pertenecen a la misma librería:

- **SDL2:** Proporciona herramientas para la creación de ventanas y renderizado de sprites en diferentes plataformas, así como la gestión del tiempo transcurrido en el juego. También proporciona herramientas para la detección de inputs del sistema, y algunos eventos que se puedan producir, como cambios en el tamaño de ventana, o el cierre de esta.
- **SDL2_Image:** Proporciona herramientas para la carga de distintos formatos de imagen como sprites en memoria.
- **SDL2_Mixer:** Proporciona herramientas para la carga y reproducción de distintos formatos de ficheros de audio.
- **nlohmann_json:** Permite la lectura y escritura de ficheros en formato json.

2.4 DOCUMENTACIÓN

Para facilitar el desarrollo del juego con el motor se ha creado una documentación de este usando Doxygen. Doxygen es una herramienta que permite generar documentos HTML a partir de comentarios en el código. Esto ha permitido generar de manera rápida y sencilla una documentación del motor con detalles necesarios para su uso.

```
/**
 * @brief Realiza un raycast y devuelve el primer Collider impactado con etiquetas especificadas.
 * @param ray El rayo a lanzar.
 * @param entityManager Referencia al EntityManager para acceder a los Colliders.
 * @param maxDistance Distancia máxima que el rayo debe recorrer (predeterminado: infinito).
 * @param tags Vector de etiquetas para filtrar impactos (predeterminado: vacío, impacta todo).
 * @return Un std::optional<RaycastHit> si se detecta una colisión con una etiqueta coincidente, std::nullopt en caso contrario.
 */
static std::optional<RaycastHit> raycast(const Ray& ray, EntityManager* entityManager,
                                         float maxDistance = std::numeric_limits<float>::infinity(),
                                         const std::vector<std::string>& tags = {});
```

Esto es un ejemplo de comentario necesario para generar la documentación con Doxygen.



Y a partir del comentario anterior se genera este documento en formato HTML visible desde el navegador.

3.RESOURCES

En esta sección se va a explicar cómo se han organizado todos aquellos elementos del juego que no son código, cómo los sprites, sonidos o los diferentes prefabs.

Tanto los sprites que se dibujan en pantalla como los sonidos se han sacado de dos páginas que han proporcionado estos recursos:

- <https://www.sprites-inc.co.uk/sprite.php?local=/X/>
- <https://www.sprisers-resource.com/snes/mmx/>

Estas dos páginas proporcionaban los sprites en formato spriteSheet, por lo que hubo que recortarlos. Una vez recortados se almacenan en carpetas teniendo en cuenta la entidad que se está recortando y sus animaciones, por ejemplo para el megaman existe una carpeta de megaman que contiene todos sus sprites, pero estos están divididos en las posibles animaciones que este tiene, cada animación se encuentra en una carpeta y los sprites están ordenados en el orden en el que se deben visualizar. Estos sprites están en formato png.

Los sonidos están en una carpeta llamada sound_effects separados de la música la cual se encuentra en la carpeta music. Los sonidos se han organizado por carpetas dependiendo de la entidad que los utiliza y están en formato wav, por otro lado la música está en formato mp3.

A su vez en los resources también se encuentran los prefabs anteriormente mencionados. Se ha creado un prefab para cada entidad del juego y así tener más control sobre cada una de ellas por separado. Además se ha creado un prefab para la cámara de cada nivel.

Por último se han definido las escenas que va a tener el juego, estas escenas se componen del nombre de la misma y todas las entidades que aparecen en ella.

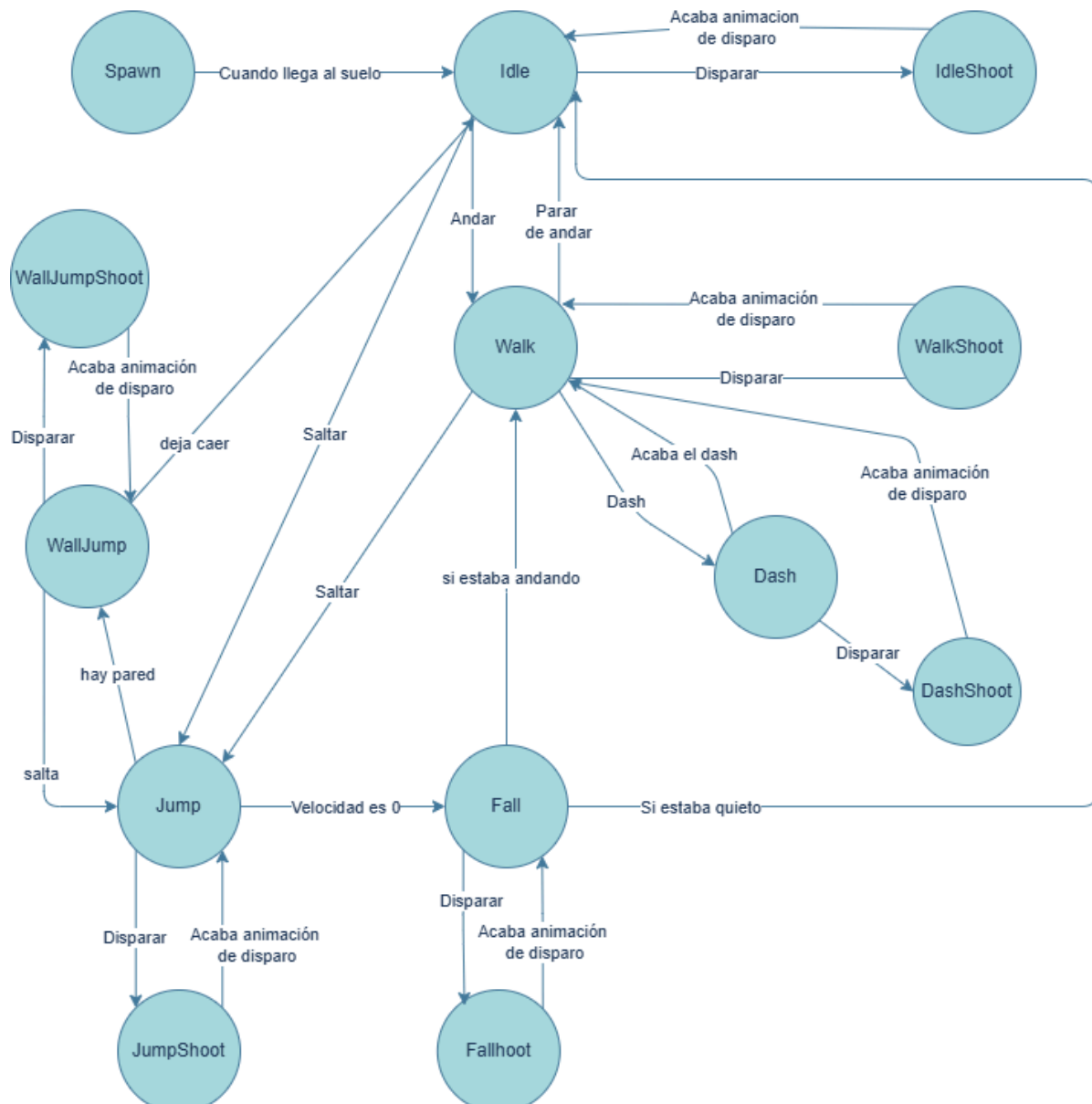
4.PERSONAJE

En esta sección se muestra la máquina de estados del personaje principal. Donde el personaje empieza en el estado *Spawn* (animación de entrada), y al tocar el suelo transita a *Idle* donde permanece quieto. Desde *Idle*, puede pasar a *Walk* si el jugador empieza a moverse horizontalmente, o disparar entrando a *IdleShoot*.

Mientras camina, puede disparar (estado *Shoot*), saltar (estado *Jump*) o hacer dash (estado *Dash*). Si dispara durante un dash, pasa a *DashShoot* y vuelve a caminar.

Durante un salto, Megaman entra al estado *Jump*, desde el cual puede disparar (*JumpShoot*) o pasar a *Fall* una vez la velocidad vertical es 0. Durante la caída, también puede disparar. Todas las acciones de disparo están acompañadas por una transición de regreso a su estado anterior una vez se completa la animación de disparar y el disparo.

La máquina de estados también contempla las situaciones de impulso en la pared (*WallJump*), qué ocurre si el personaje está saltando y se pega a una pared. Desde este estado, puede volver a saltar, dejarse caer o disparar.

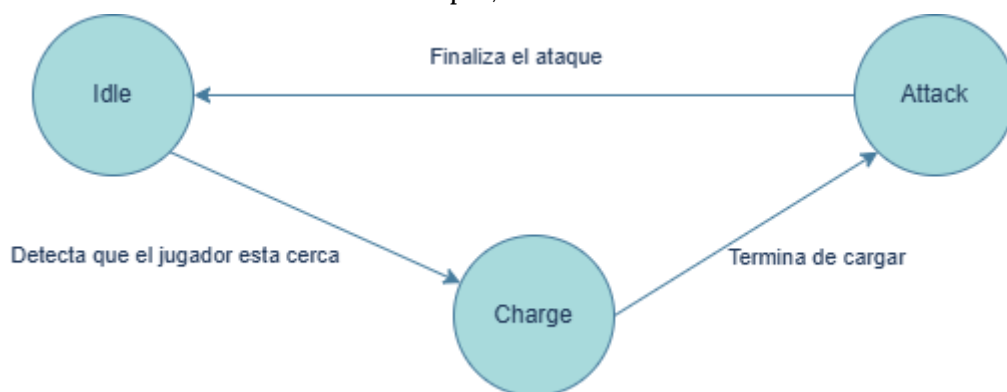


5.ENEMIGOS

Todos estos enemigos pueden morir en cualquier estado. No es obligatorio eliminar a estos enemigos para pasarse el nivel. Cada uno de estos enemigos además de su ataque, que obviamente hace daño al jugador, producen daño al jugador al contacto con ellos. Todos ellos están implementados con el módulo de IA clásica anteriormente mencionado.

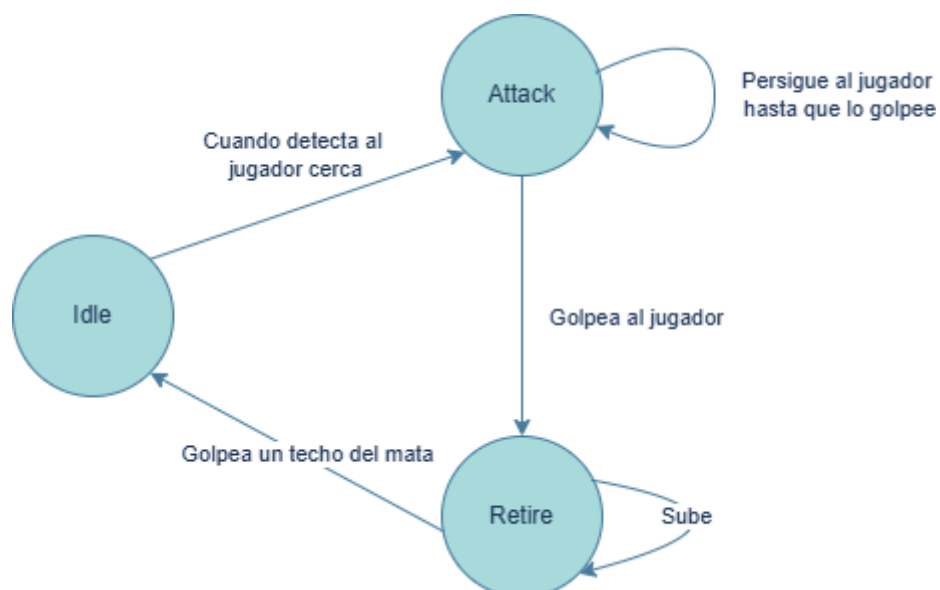
5.1 AXE MAX

Axe Max es un enemigo estático, que no se desplaza por el escenario. En cuanto el jugador se encuentra en su área de ataque, este carga dos bloques. Una vez ha terminado de cargarlos, pasa al estado de ataque. Donde le lanza un bloque y seguidamente el otro. Al finalizar el ataque, vuelve al estado inicial.



5.2 BATTON BONE

Batton Bone es un enemigo que se encuentra quieto en el techo. Cuando el jugador entra en su área de ataque, este le persigue hasta golpearlo. Una vez le haya golpeado, el enemigo se retira volando hacia arriba. En el caso de encontrarse con un techo, se vuelve a posar y pasa al estado inicial. Mientras que si no encuentra ninguno, desaparece.

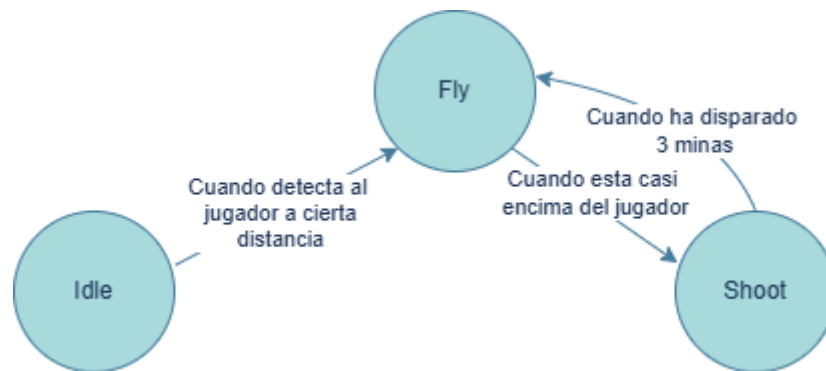


5.3 BOMB BEEN

Bomb Been es un enemigo que se encuentra volando por el mapa. Sin embargo, se le ha añadido un estado *idle* para que permanezca quieto, ya que de lo contrario podría detectar al jugador en partes de la escena donde aún no debería estar. Esto sucedía porque comenzaba a acercarse al jugador antes de tiempo.

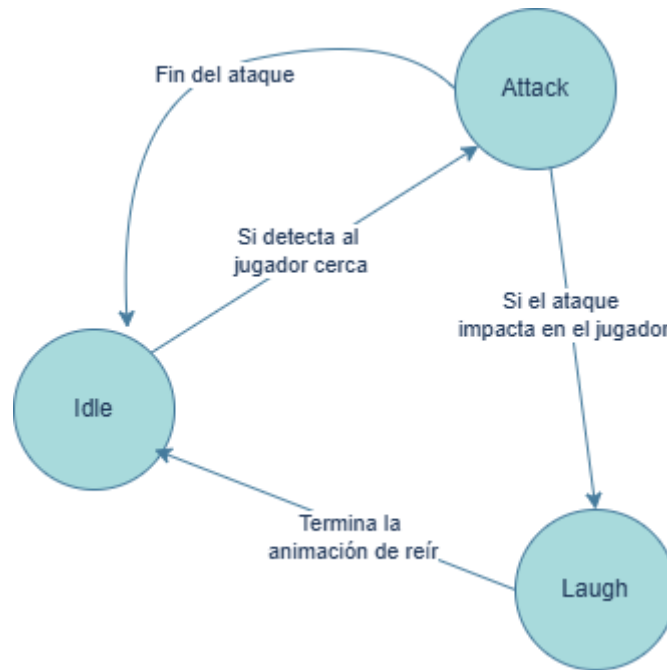
Para solucionarlo, se ha configurado para que empiece a volar hacia el jugador únicamente cuando este entre en su zona de ataque. Dicha zona de ataque es mayor a la de otros enemigos para que el jugador no aprecie que el enemigo ha permanecido quieto.

Una vez el jugador entre en su zona de ataque el Bomb Been vuela hasta colocarse encima del jugador. Cuando lo ha conseguido, este le lanza 3 minas de ataque y vuelve al estado de volar. Para que este enemigo pueda volver a atacar tiene que pasar 10 segundos.



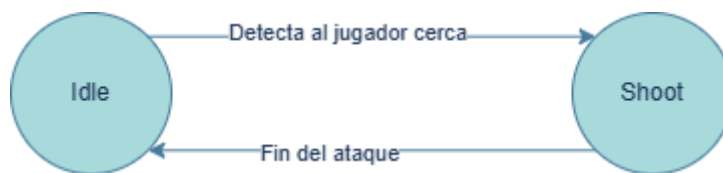
5.4 DIG LABOUR

Dig Labour es un enemigo estático. Cuando el jugador entra en su área de detección, Dig Labour cambia al estado de ataque. En este estado, el enemigo lanza un pico con la intención de dañar al jugador. Si el pico impacta al jugador, el enemigo se ríe. En caso contrario, o después de reírse, vuelve a su estado inicial.



5.5 FLAMMINGLE Y SNOW SHOOTER

Estos dos enemigos son también enemigos estáticos. Funcionan muy similar al anterior, cuando el jugador entra en su área de ataque, cambian al estado de ataque. En este estado, el enemigo le lanza una sierra en el caso del flammingle o una bola de nieve en el caso del snow shooter. A continuación, vuelven a su estado inicial.



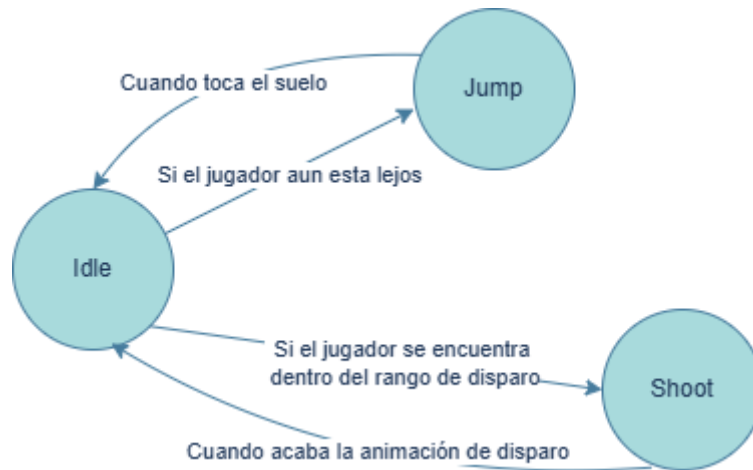
5.6 SPIKY

Spiky es un enemigo que se desplaza rodando por el escenario. Solo ha sido necesario comprobar si colisiona con alguna pared para que cambie su dirección de rodamiento. Este enemigo hace daño al jugador si colisiona con él.



5.7 RAYBIT

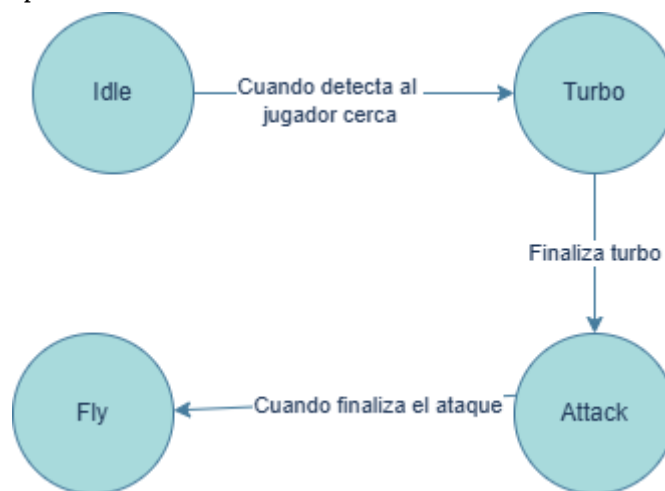
Raybit es un enemigo que se desplaza saltando por la escena. Cuando se encuentra con el jugador, le dispara y luego vuelve a su estado inicial. Para poder volver a atacar, deben transcurrir tres segundos desde su último disparo.



5.8 TOMBOT

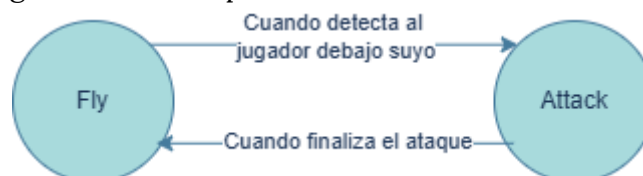
Tombot es un enemigo que permanece quieto en la escena. Cuando el jugador entra en su zona de ataque, realiza un pequeño impulso (turbo) para posicionarse justo encima de él. Al finalizar ese movimiento, entra en el estado de ataque, en el que lanza sus zapatillas con la intención de infligir daño si impactan contra Megaman.

Tras completar este ataque, Tombot comienza a perseguir al jugador, intentando chocar con él directamente para causarle daño.



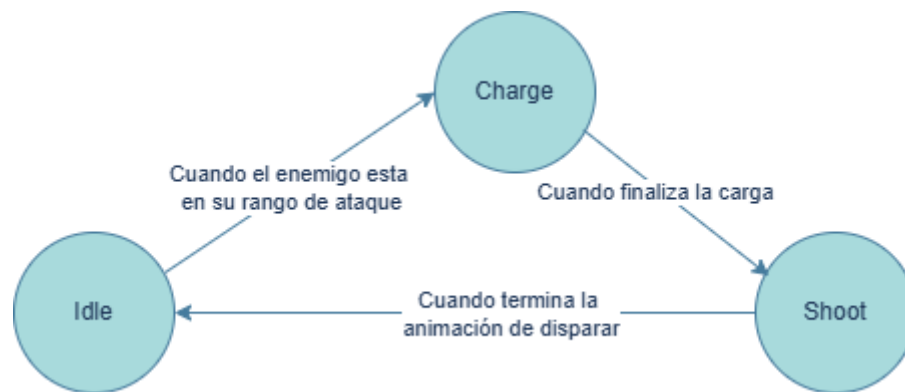
5.9 SKY CLAW

Sky Claw es un enemigo volador que patrulla por la escena. Cuando detecta al jugador justo debajo de él, se lanza en picado para golpearlo e infligir daño. Tras realizar un ataque, espera un segundo antes de poder volver a atacar.



5.10 SCRAP ROBO

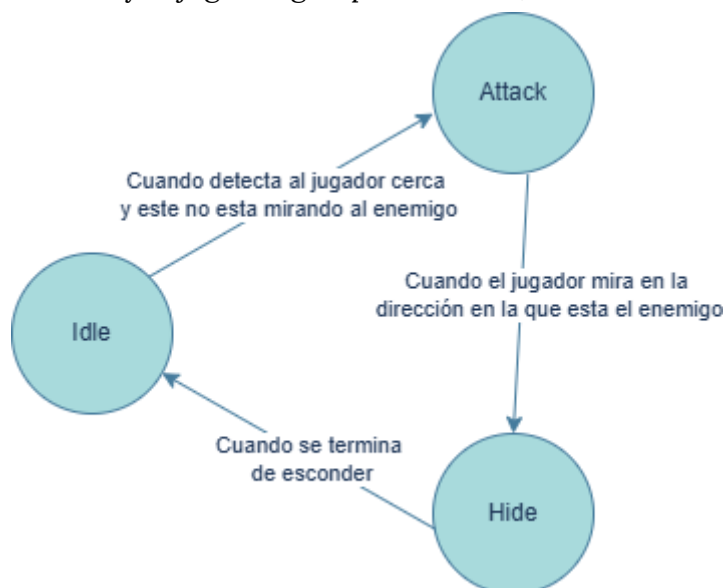
Scrap robo es un enemigo estático. Cuando el jugador entra en su zona de ataque, carga su disparo. Una vez completada la carga, dispara y vuelve a su estado inicial.



5.11 METAL

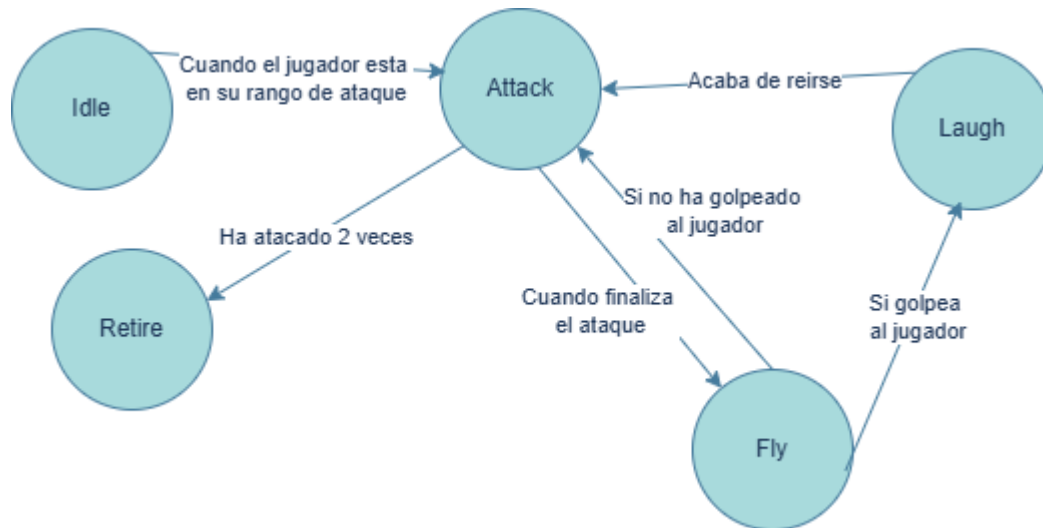
Metal es un enemigo que permanece quieto y oculto dentro de su casco. Cuando el jugador entra en su zona de ataque, si no está mirando hacia la posición de Metal, este sale de su escondite y lo persigue para intentar chocar y causarle daño.

Sin embargo, si el jugador está mirando en su dirección, permanece oculto. Además, si ya ha comenzado a atacar y el jugador gira para mirarlo, Metal se esconde nuevamente.



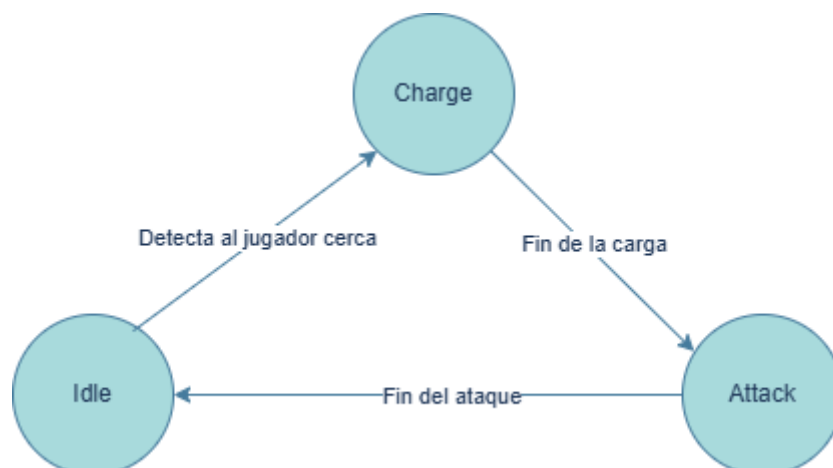
5.12 JAMMINGER

Jamminger es un enemigo que se encuentra volando por la escena. Cuando el jugador entra en su zona de ataque, cambia al estado de ataque. Su ataque consiste en descender para golpear al jugador. Si logra impactar al jugador, se desplaza verticalmente un poco hacia arriba y realiza una animación de risa. Si no golpea, vuelve a intentar atacar. También repite el ataque después de terminar la animación de risa. Cuando ha atacado dos veces, cambia al estado de retirada, en el que se aleja volando fuera de la escena.



5.13 HOGANMER

Hoganmer es un enemigo estático. Cuando el jugador entra en su zona de ataque, pasa al estado de carga (*charge*), en el que comienza a cargar su bola de pinchos y retira su escudo defensivo. En ese momento, o mientras está atacando, es cuando puede ser derrotado. Una vez finalizada la carga, lanza la bola de pinchos. Tras recogerla, vuelve a su estado *idle*.

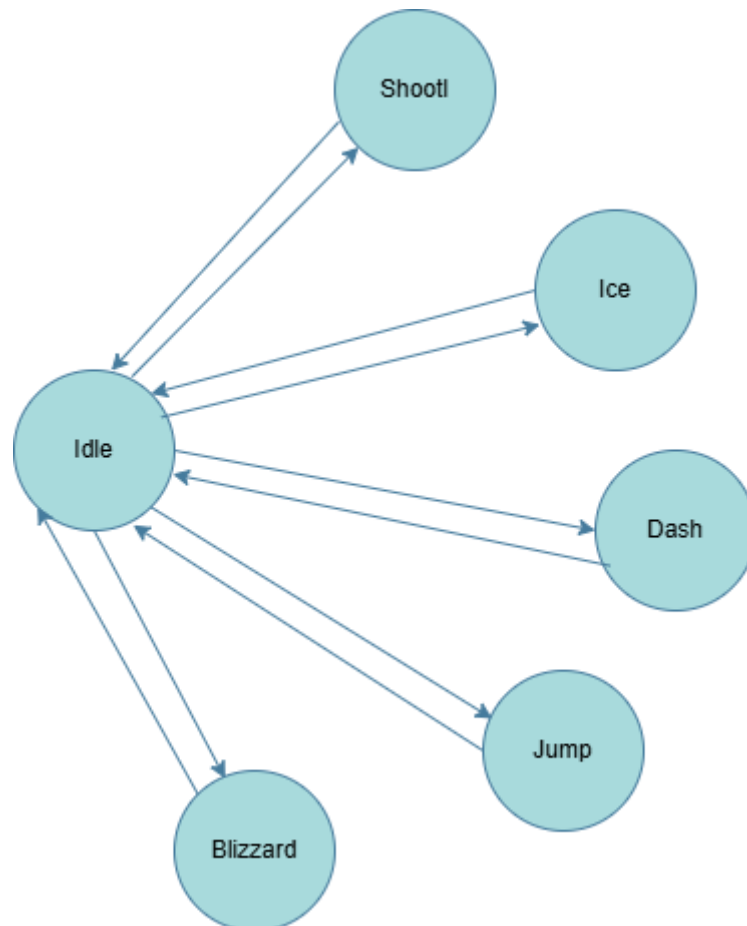


6. JEFES

Los jefes son los enemigos que se encuentran al final de cada nivel, estos están localizados en una sala especial donde el jugador se enfrentará a ellos. Estos tienen más vida que un enemigo básico y ataques y movimientos más complejos, dificultando la pelea. Esta pelea es obligatoria para superar el nivel y una vez terminada el jugador será redirigido al menú de selección de nivel. Los jefes permanecen inactivos hasta que el jugador entra en la sala de la pelea, una vez esto ocurre el jefe hará una animación que indicará el comienzo de la batalla.

6.1 CHILL PENGUIN

El Chill Penguin es el jefe final de la Montaña Nevada. Este jefe intenta derrotar a megaman con ataques basados en frío o hielo, como bolas de hielo, o ventiscas, además de golpear al jugador físicamente con un salto y un dash. No se han incluido las transiciones en la siguiente máquina de estados ya que son más complejas que las de los enemigos básicos y se detallarán más adelante.



Esta máquina de estados es determinista pero a través de la aleatoriedad y transiciones complejas se ha intentado obtener un comportamiento impredecible que simule el comportamiento del juego real. Todos los estados vuelven a Idle para permitir decidir cuál será la siguiente acción del jefe.

Los ataques son los siguientes:

- Shoot: dispara hacia el jugador una bola de hielo que si impacta con él le hará daño.
- Ice: el jefe dispara su aliento helado, el cual genera dos pingüinos de hielo, los cuales no hacen daño al jugador.
- Dash: el jefe se lanza deslizándose hacia el jugador, si lo impacta le hace daño.
- Jump: el jefe salta hacia el jugador intentando aplastarlo.
- Blizzard: el jefe salta, y se queda colgando del techo, activando una ventisca que empuja los pingüinos de hielo anteriormente creados hacia el jugador, haciéndole daño si alguno impacta con él

Las transiciones se realizan de la siguiente manera:

Idle->shoot: el jugador debe estar en el rango del jefe, ya sea a la izquierda como a la derecha de este. Además tiene en cuenta si el jefe ha disparado la bola de hielo recientemente.

Idle->Ice: el jugador debe estar en el rango del jefe, ya sea a la izquierda como a la derecha de este. Además tiene en cuenta si el jefe ha disparado el aliento helado recientemente.

Estos dos ataques son idénticos, por lo que si el jefe cumple las condiciones de ambos se decidirá aleatoriamente cuál de los dos realizará.

Idle->Dash: el jugador se encuentra a una distancia media del jefe y el ataque no tiene el cooldown activo, el cooldown son 5 segundos.

Idle->Jump: el jugador debe estar a una distancia más lejana que los ataques anteriores, y el ataque no puede estar el cooldown, este son 3 segundos.

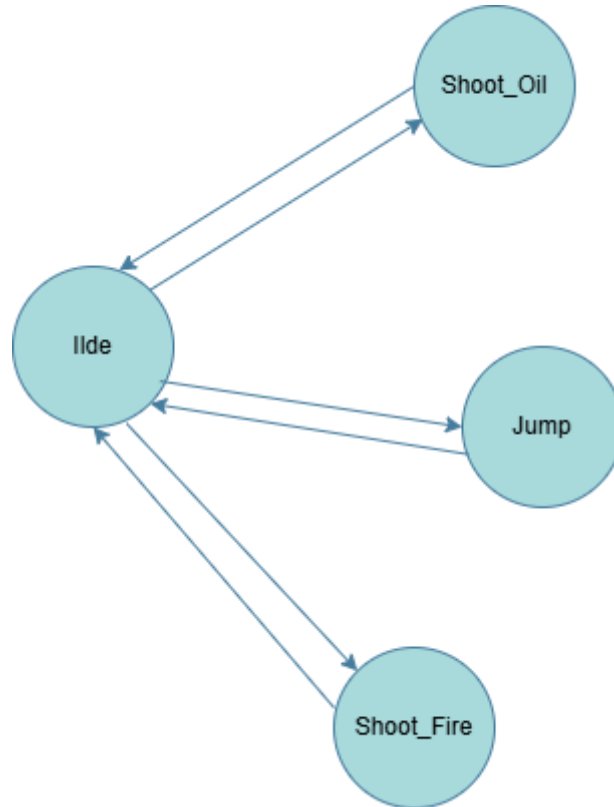
Idle->Blizzard: el jugador debe estar a una distancia más lejana que los ataques anteriores, y el ataque no puede estar el cooldown, este son 3 segundos.

Estos dos últimos ataques son idénticos, por lo que para elegir cuál se hará aleatoriamente.

También se ha implementado una IA más avanzada con ayuda del módulo ReinforcementIAComponent, este módulo ha permitido establecer una IA más compleja y con aprendizaje automático. Según las acciones del jefe y el resultado que estas tengan este se verá recompensado o castigado, se ha decidido que se vería recompensado cuando la decisión tomada daña al jugador, en caso en el que este no resulte dañado el jefe resultará castigado. Además si después de realizar una acción la vida del jefe se ha visto reducida también se le castigará. Las acciones que puede realizar son las mismas que en la máquina de estados anterior, saltar hacia el jugador, realizar un dash, lanzar un aliento helado que crea pingüinos de hielo, lanzar una bola de hielo e invocar una ventisca.

6.2 FLAME MAMMOTH

El Flame Mammoth es el enemigo del nivel de la fábrica. Este jefe te ataca con sus ataques de fuego y aceite, y arremete contra el jugador con un salto que pretende aplastar a Megaman. Como se observa en la siguiente máquina de estados no se han incluido las transiciones para las acciones, ya que se explican a continuación.



Esta máquina de estados representa la IA del jefe implementada con el módulo de IA clásica mencionado en la parte del motor. Es una máquina completamente determinista, pero que pretende simular un comportamiento complejo eligiendo las acciones que realizará de manera aleatoria. Cada acción tiene un porcentaje ligado a ella que dependiendo de la distancia a la que se encuentre el jugador varía, esto se ha hecho así para que al jugador se le dificulte encontrar patrones en los ataques. Además sumado a esto cada ataque puede repetirse un número de 1 a **N** veces, variando **N** según el ataque y siendo elegido el número de repeticiones de manera aleatoria. Con todos estos aspectos se ha conseguido obtener un comportamiento difícil de predecir intentando simular el comportamiento del juego original.

Los ataques que puede hacer este jefe son:

- Jump: el jefe salta sobre el jugador, al estar en una zona cerrada el jugador puede verse arrinconado en según qué situación.
- Shoot Oil: este ataque no hace daño al jugador. El jefe lanza una bola de aceite que se queda impregnado en el suelo formando un charco, este luego puede prender si entra en contacto con la llama que lanza el jefe, si esto ocurre creará una llama mucho más potente que dañará al jugador.

- Shoot Fire: este ataque lanza una llama al jugador, esta le hace daño si impacta con él y activará el aceite si impacta con uno de los charcos de aceite generados por el ataque anterior. Si la llama toca cualquier superficie esta se extinguirá.

En este jefe también se ha programado una IA con aprendizaje automático. Los pesos se reparten igual que en el chillPenguin, si el megaman es dañado con una acción la recompensa es positiva, si el megaman no es dañado o el jefe pierde vida la recompensa será negativa. Los estados son los mismos que se encuentran en la máquina de estados anterior, saltar hacia el jugador, lanzar aceite y lanzar fuego.

7. PROBLEMAS Y SOLUCIONES

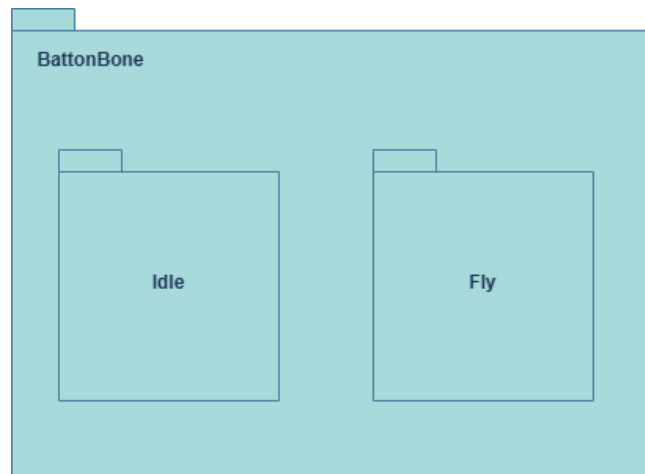
En esta sección se describen los principales problemas encontrados durante el desarrollo, junto con las soluciones aplicadas en los casos en que fue posible resolverlos.

7.1 MANEJO DE SPRITES

Inicialmente, se pensaba que cada sprite venía en archivos separados. Sin embargo, se descubrió que todas las animaciones de un personaje estaban agrupadas en una imagen. Para poder extraer las animaciones individualmente, se utilizó una herramienta que permite seleccionar y recortar manualmente los sprites desde una imagen. Esta herramienta se encontró en GitHub, a continuación se muestra el link:

<https://github.com/ForkandBeard/Alferd-Spritesheet-Unpacker>

Una vez recortados los sprites, se procedió a renombrarlos y organizarlos en carpetas según la animación correspondiente: Por ejemplo, para el personaje del BattonBone, que solo cuenta con dos animaciones (Idle y Fly), la estructura en carpetas es de la siguiente manera:

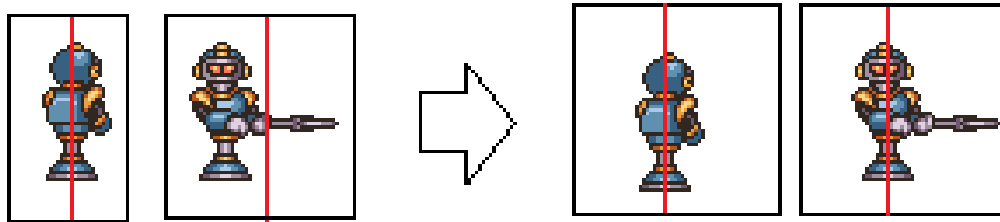


7.2 PROBLEMAS CON LA ALINEACIÓN DE SPRITES

Otro problema que se encontró tuvo que ver con la alineación de los sprites durante las animaciones.

El motor del juego permite elegir entre utilizar como punto de referencia la esquina superior izquierda del sprite o su centro. Debido a las diferencias en las dimensiones de algunos sprites, al animarlos estos parecían moverse o desajustarse, lo que daba la impresión de que la animación estaba mal hecha.

Para solucionar esto, se utilizó GIMP y se ajustó manualmente el tamaño del lienzo de los sprites afectados, asegurándose de que todos tuvieran las mismas dimensiones y que el punto de referencia fuera consistente. Un caso concreto fue el Axe MAX: durante su animación de ataque, el sprite se desplazaba debido a la rotación con el hacha, que alteraba visualmente el centro de la imagen.

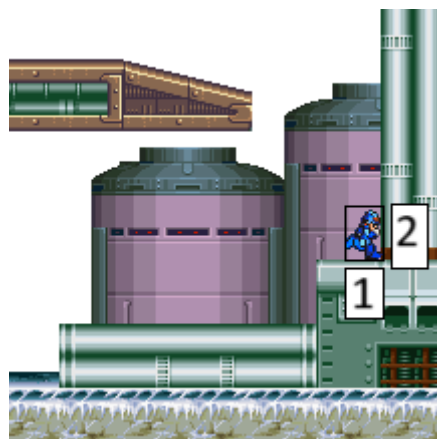


En la ilustración se muestra en la parte de la izquierda el problema que se encontró y en la parte de la derecha como quedó al solucionarlo.

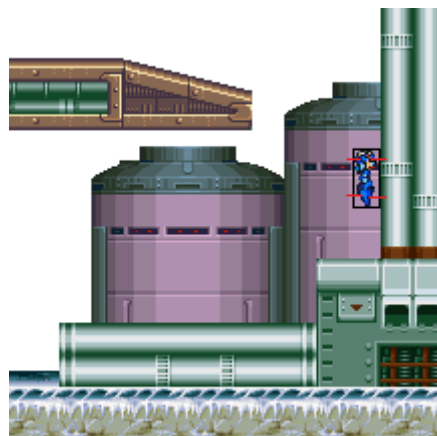
7.3 PROBLEMA CON EL WALL JUMP

Otro de los problemas que se encontraron fue con la implementación del Wall jump. En un primer intento, se optó por utilizar la función *OnCollisionEnter*, la cual se ejecuta automáticamente cuando el objeto colisiona con otro objeto que tenga collider.

El inconveniente principal de este enfoque fue que *onCollisionEnter* comparaba constantemente las colisiones con cualquier parte del terreno. Esto generaba inconsistencias, por ejemplo, si el jugador saltaba desde una esquina, el sistema detectaba primero la colisión con el suelo (1) en lugar de con la pared(2), impidiendo así ejecutar correctamente el wall jump.



Para resolver este problema, se decidió implementar una alternativa basada en el uso de Raycasts. Esta técnica consiste en lanzar rayos desde el personaje hacia los lados (dos por cada lateral) cada vez que se intenta realizar un salto. De este modo, es posible verificar con mayor precisión si el personaje está realmente en contacto con una pared, permitiendo que el wall jump se ejecute sólo cuando las condiciones son correctas.



7.4 PROBLEMAS DE COLISIONES

El motor de colisiones trabaja con 2 tipos: AABBs y triángulos. Las colisiones AABBs no han dado problemas, ya que son colisiones muy simples y fáciles de gestionar. El problema es que el Megaman consiste de niveles con rampas inclinadas, algo que con AABBs no se pueden recrear. La solución elegida en el desarrollo para representar estas rampas fue el uso de colisiones triangulares.

El problema de los triángulos respecto a los AABBs es que sus cálculos de intersección no son tan baratos computacionalmente (Esto se arregla con la estructura BVH mencionada anteriormente), y el gran problema son los cálculos para reposicionar una colisión en un triángulo, ya que hay muchos casos a comprobar no tan simples para determinar la posición de una colisión que está chocando con un triángulo.

Al final, poniendo parches al código relacionado con los triángulos se ha conseguido que no den tantos fallos, pero esta decisión de diseño ha aportado más errores que soluciones.

8. DECISIONES FINALES

Durante el desarrollo del proyecto, el equipo tuvo que tomar varias decisiones importantes.

Aunque inicialmente se propusieron 4 niveles, debido a problemas técnicos con el sistema de colisiones y dificultades en la organización del equipo, se decidió centrarse únicamente en dos niveles completos. Esta decisión, aunque difícil, ha permitido ofrecer una experiencia más pulida, en lugar de presentar una mayor cantidad de niveles con poca jugabilidad.

9.DESARROLLO DEL JUEGO

9.1 TIEMPO EMPLEADO:

Alumno	Tiempo
Jorge Fernández Marín	70 Horas
Javier Julve Yubero	110 Horas
Marcos García Ortega	115 Horas
Saul Caballero Luca	170 Horas
Aroa Redondo Zamora	100 Horas

9.2 TAREAS REALIZADAS:

Alumno	Tareas
Jorge Fernández Marín	Desarrollo del personaje principal, desarrollo del menú de configuración, corrección de bugs del juego y del engine.
Javier Julve Yubero	Realizados 3 enemigos y a chill penguin, bug tester, redacción de requisitos, recorte y edición de 2 enemigos, corrección de bugs del engine. Elaboración del tráiler y de la presentación de power point.
Marcos García Ortega	Implementación de enemigos, recorte de sprites, desarrollo del flame mammoth y otros objetos interactivos del mapa, implementación del el sistema de drops y redacción de la documentación.
Saul Caballero Luca	Elaboración de todo el engine y construcción de niveles, coordinador de las tareas del proyecto.
Aroa Redondo Zamora	Recortar sprites, cuadrar sprites con gimp, elaboración de enemigos, implementación de todos los menús, barra de vida de megaman, redacción de la documentación.

10. ANEXO

GAME DESIGN DOCUMENT FINAL

1. DESCRIPCIÓN GENERAL

1.1 RESUMEN

“Megaman” es un clon clásico del juego Megaman X que combina acción de plataformas 2D con elementos shooter. Donde, el jugador asume el papel del Megaman X, un avanzado robot con capacidades de correr, saltar, disparar y realizar otros movimientos especiales como wall jumps y dash. A través de dos niveles temáticos, el jugador deberá enfrentarse a diversos enemigos en entornos peligrosos. El juego mantiene la esencia y dificultad del Megaman original con gráficos del estilo pixel art de 16 bits.

1.2 GÉNERO

El género del videojuego pertenece a una combinación de plataformas y shooter en 2D, donde aparecen desplazamientos laterales, combate con armas a distancia y saltos precisos.

Debido a que el objetivo es crear un clon del clásico Megaman X, se hereda la estructura de niveles y mecánicas principales. El jugador avanza por los escenarios linealmente evitando obstáculos y plataformas, además de enfrentarse a enemigos con ataques a media distancia y cuerpo a cuerpo.

1.3 JUGADORES

El juego cuenta con solo un modo de juego, **single player**. El jugador tiene el control total del personaje principal, donde su objetivo es superar una serie de niveles en solitario, enfrentados a enemigos, jefes y plataformas.

1.4 AUDIENCIA OBJETIVO

El juego va dirigido a aquellos jugadores que disfrutan de juegos clásicos de acción y plataformas en 2D. Su dificultad, ritmo intenso y mecánicas lo hacen más atractivo para aquellos que cuentan con cierta experiencia en el mundo de los videojuegos, aunque también va destinado aquellos jugadores que desean disfrutar y descubrir videojuegos del estilo retro.

1.5 LOOK AND FEEL

La estética está replicada del videojuego Megaman X. La cual muestra una estética de pixel art del estilo 16-bit. Los gráficos están diseñados para ofrecer una experiencia visual de efectos fluidos y detallados, que dan vida a los personajes y escenarios, además

de un ambiente futurista con un tono más serio que otros juegos de plataformas. Los niveles están diseñados para reflejar una amplia gama de climas y entornos, como paisajes nevados o minas con lava, que aportan diversidad visual.

2. HISTORIA

En el año 21XX, la humanidad hace un descubrimiento revolucionario: un robot con la capacidad de pensar, razonar y tomar decisiones por sí mismo. Este hallazgo marca un antes y un después en el desarrollo tecnológico. El científico responsable del descubrimiento decide estudiar su diseño en profundidad, realizando ingeniería inversa para crear una nueva generación de robots inteligentes llamados *Reploids*, concebidos para convivir y colaborar con los humanos en diversas tareas.

Con el tiempo, algunos de estos *Reploids* comienzan a comportarse de forma anómala debido a un virus desconocido que altera su programación y los vuelve hostiles. A estos individuos se les conoce como *Mavericks*. La amenaza crece rápidamente y pone en peligro a toda la civilización, liderados por un antiguo *Reploid* de élite llamado *Sigma*, el más poderoso y peligroso de todos.

Para combatir esta amenaza, se funda una organización especial conocida como los *Maverick Hunters*, un grupo de élite formado por *Reploids* diseñados para identificar y neutralizar a los *Mavericks*. El protagonista del juego forma parte de esta unidad, enfrentándose a las fuerzas de *Sigma* con el objetivo de restaurar la paz.

3. GAMEPLAY

3.1 OBJETIVOS Y LÓGICA DEL JUEGO

El objetivo principal del juego es completar los dos niveles disponibles, derrotando a los jefes finales de cada uno: Chill Penguin y Flame Mammoth. El jugador debe avanzar a través de las plataformas, evitar obstáculos y eliminar enemigos mientras gestiona su salud.

El juego sigue una estructura lineal donde cada nivel presenta desafíos únicos relacionados con su temática. El jugador puede elegir el orden en el que desea enfrentarse a los niveles, ya que no hay un orden predeterminado.

3.2 MECÁNICAS

3.2.1 MOVIMIENTO

El movimiento del megaman consta de un movimiento lateral que permite desplazarse de izquierda a derecha. Además del movimiento horizontal básico, puede realizar saltos, que permiten al personaje alcanzar plataformas elevadas y evitar obstáculos del mapa. Incluye también una mecánica de dash, que permite desplazarse rápidamente hacia delante. Finalmente, el personaje puede hacer wall jumps, permitiendo impulsarse desde paredes para alcanzar zonas más altas.

3.2.2 ATAQUE

El personaje cuenta con un sistema de disparo cargado que permite atacar a los enemigos a media distancia. Este sistema cuenta con 3 niveles de carga. El primer nivel es el básico, sin carga, se ejecuta al presionar rápidamente el botón de ataque y lanza un proyectil pequeño. Al mantener presionado un segundo el botón de ataque aparece el segundo nivel, que lanza un proyectil más grande, más potente y de color verde. Por último, el tercer nivel es el más potente y el que genera más daño. Se muestra con un proyectil de color azul.

3.2.3 OBSTÁCULOS

El mundo cuenta con varios obstáculos. Algunos de ellos son, plataformas, rampas y paredes a escalar. Además, cuenta con espacios vacíos los cuales son bastante peligrosos ya que causan la derrota instantánea del personaje si cae en ellos. Estos combinados con los ataques de los enemigos será todo un desafío para el jugador.

3.2.4 GESTIÓN DE RECURSOS (DROPS)

La gestión de recursos se basa en un sistema de drops obtenidos al derrotar al enemigo. En cambio, no se obtienen de forma garantizada, ya que solo algunos enemigos dejarán caer objetos al ser eliminados. Entre los recursos disponibles hay: vida pequeña, que recupera una porción de la salud del personaje; vida grande, que recupera el doble de

salud que la vida pequeña ; y el corazón, que aumenta en uno las vidas restantes del personaje. Estos drops son clave para la supervivencia del jugador en los niveles.

3.3 REGLAS

3.3.1 REGLAS MEGAMAN

- El megaman muere si la salud del mismo se acaba.
 - ◆ Respawnea en un checkpoint si tiene vidas restantes.
 - ◆ Si no cuenta con vidas restantes te lleva al menú principal.
- El megaman muere si se cae al vacío.
- El megaman restaura vidas al recoger el drop con forma de cabeza de megaman.
- Si un enemigo golpea al jugador, éste pierde salud, se queda inmovil en el momento del impacto, pero gana unos segundos de inmunidad.

3.3.2 REGLAS DEL JUEGO

- El jugador pasa de nivel si y sólo si mata al boss del nivel.
- Los ataques del jugador atraviesan los obstáculos.
- Los enemigos sueltan drops aleatoriamente.
- Algunos enemigos pueden atravesar los obstáculos.

3.3.3 REGLAS DE JUGABILIDAD

- Los saltos tienen una altura fija.
- El megaman no puede volver a saltar si aún está en el aire.
- Solo puede ejecutar un wall jump si entra en contacto con la pared.
- El tiempo que dura el dash es fijo y predeterminado.
- El megaman no puede volver a realizar un dash si este aun esta en el estado de dash o está saltando.

3.4 PROGRESIÓN

El juego cuenta con 2 niveles.

3.4.1 NIVEL 1 - CHILL PENGUIN

El nivel de chill penguin se desarrolla en una zona montañosa y nevada, con una temática ártica. El nivel culmina con el combate a chill penguin, el jefe utiliza ataques de hielo, proyectiles congelantes y ráfagas de viento.

3.4.2 NIVEL 2 - FLAME MAMMOTH


El nivel del Flame Mammoth está ambientado en una fábrica con temática de fuego y fundición. Está lleno de plataformas móviles, lava, y enemigos mecánicos relacionados con el entorno industrial. El jefe final, utiliza ataques de fuego y proyectiles de gran alcance lo que obliga al jugador a mantener la movilidad y precisión de los ataques.

4.4.3 CÓMO TERMINA EL JUEGO




El objetivo del juegos es explorar ambos niveles y derrotar a el boss final de cada uno de ellos.

4. PERSONAJES

4.1 PERSONAJE JUGABLE




Personaje	Resumen	Ataque	Salud	Vidas	Imagen
Megaman	Mega Man X es un Cazador de Maverick, y dedica su vida a la erradicación de Replodes delincuentes, denominados como "Mavericks", y a desbaratar las confabulaciones de Sigma.	3 ataques	500	2	








4.1.1 ARMA






Nombre	Descripción	Color	Ataque	Imagen
Ataque básico	se activa presionando rápidamente el botón de ataque	Amarillo	35	
Ataque normal	se activa presionando un segundo el botón de ataque	Verde	50	
Ataque cargado	se activa presionando tres segundos el botón de ataque	Azul	100	

4.2 ENEMIGOS



4.2.1 ENEMIGOS NORMALES

Nombre	Descripción	Niveles	Vida	Ataque	Imagen
Spiky	Enemigo que rueda hacia el personaje.	1	100	50	
Ray Bit	Enemigo que persigue al enemigo saltando. Dispara proyectiles.	1	100	50	
Tombot	Enemigo que se encuentra volando. Cuando te	1	100	50	

	acercas a él, te persigue y lanza sus botas				
Bomb Been	Enemigo que se encuentra volando en una dirección y cuando el megaman está cerca le lanza 3 minas	1	100	50	
Snow shooter	Enemigo que se encuentra quieto en el nivel. Cuando el personaje entra en su área de ataque, te dispara bolas de nieve.	1	100	50	
Flammingle	Enemigo que se encuentra quieto en el nivel. Cuando el personaje entra en su área de ataque, te dispara la sierra que se encuentra en su cabeza	1	100	50	
Axe Max	Enemigo que se encuentra quieto en el enemigo. Cuando entras en su área de ataque carga los bloques que más tarde te va a arrollar	1	100	50	
Batton Bone	Enemigo que se encuentra en el techo. En cuanto el personaje pasa por debajo de él, le persigue volando para golpearte.	1	100	50	
Jamminger	Enemigo que se encuentra volando. Cuando el personaje se acerca a jamminger, este le ataca	1	100	50	
Scrap Robo	Enemigo que se mantiene quieto y dispara un rayo láser al personaje	2	100	50	
Dig Labour	Enemigo que permanece quieto, cuando el megaman entra en su área de ataque, este le lanza su pico	2	100	50	




Hoganmer	Enemigo que permanece quieto, cuando el megaman entra en su área de ataque, lanza su bola de pinchos para golpearla. Una vez lanzada la retrae. Mientras no lanza la bola se cubre con el escudo, el cual para todo el daño	2	100	50	
Pipeturn	Enemigo se encuentra rotando por tuberías, es inmune	2	-	50	
Sky claw	Enemigo que se encuentra volando. Cuando el personaje se acerca a sky claw, este le ataca	2	100	50	
Metal	Enemigo que te persigue cuando entras en su rango de ataque.	2	100	50	
Scrap	Enemigo que se encuentra quieto en el mapa, si entras en contacto con el te baja la salud	2	100	50	

4.2.2 BOSS FINAL

Nombre	Descripción	Niveles	Vida	Ataque	Imagen
Chill Penguin	Puede disparar bolas de hielo, lanzar aliento gélido para crear dos señuelos de hielo, saltar hacia ti. También puede usar un gancho en el techo para lanzarte una ventisca.	1	1500	50	
Flame Mammoth	Este enemigo salta hacia ti y sacude el suelo, dispara aceite y bolas de fuego. Si una bola de fuego impacta un charco de aceite,	2	1500	50	

	creará una Ola de Fuego.				
--	--------------------------	--	--	--	--

4.3 DROPS

Nombre	Recupera Vida/salud	Porcentaje de aparición	Imagen
Vida pequeña	Recupera 30 salud	35%	
Vida grande	Recupera 75 salud	20%	
Corazón	Recupera 1 vida	5%	

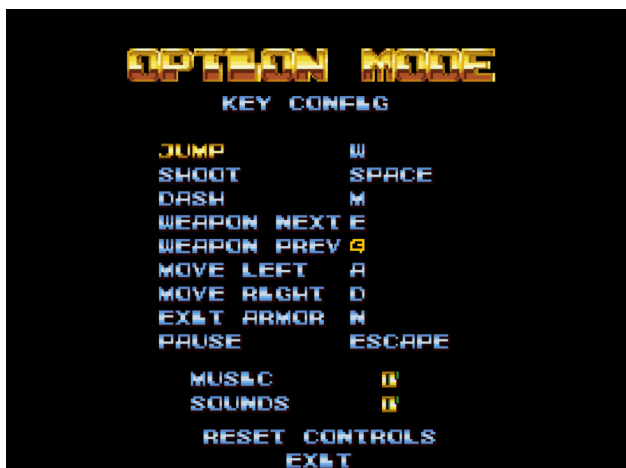
5. INTERFACES

5.1 MENÚ PRINCIPAL

A continuación, se muestra una ilustración del menú principal, el cual cuenta con 3 opciones. La opción de exit sirve para salir del juego y que este se cierre, la opción de *option mode* y *game start* redirigen al menú de opciones y menú de selección de nivel respectivamente.



5.2 MENÚ DE OPCIONES



El menú de opciones permite cambiar los controles del jugador, así como resetearlos y poner los controles por default. Además, cuenta con dos controles para el sonido. El primero, *music*, da la posibilidad de controlar el volumen de la música del juego. Mientras que *sounds*, permite regular el volumen de los sonidos del juego.

5.3 MENÚ ELECCIÓN BOSS

El menú de selección cuenta con tres opciones, *stage*, conforme el jugador va moviendo el selector de nivel va mostrando una imagen relacionada con el nivel, *map*, muestra con un círculo parpadeando la ubicación de nivel en el mapa, y por último, *spec*, muestra información sobre el boss del nivel.



5.4 MENÚ PAUSA



El menú de pausa muestra, en la parte superior, el combustible restante para los ataques de los power ups.

En la parte central inferior, se puede observar la figura del megaman junto con la barra de vida. Esta muestra la salud restante del mismo. Por último, en la parte de la derecha se encuentra el número de vidas restantes del jugador.

6. MÚSICA

El apartado sonoro del juego está inspirado en el estilo retro de 16 bits. Cada fase de plataformas contará con un tema musical único, adaptado a la ambientación del nivel. Por otra parte, los jefes compartirán un tema musical común.

Además, se integrarán efectos de sonido específicos para acciones clave como disparos, explosiones y daño recibido, que contribuyen a una mejor experiencia del juego.

7. REQUERIMIENTOS DE DESARROLLO

El videojuego requiere un procesador y un sistema operativo de 64 bits.

Elemento	Descripción
SO	WINDOWS 7 (64bit)
Procesador	Intel Core i3 550 3.2GHz ó AMD equivalent ó superior
Memoria	2 GB de RAM
Gráficos	Sprites 2D
Lenguaje de programación	c++ con MingGW
Control de versiones	Git

8. MARKETING Y PRESUPUESTO

8.1 MARKETING

Se ha planificado una fase de pruebas por etapas para asegurar la calidad del producto antes de su lanzamiento oficial. El 26 de marzo se lanzará una versión Alfa, destinada a ver los errores graves del juego y del sistema base. Posteriormente, el 28 de abril se publicará la versión Beta, que contendrá el contenido casi final , y estará destinada a que los usuarios puedan probar una versión del juego y ofrecer su retroalimentación inicial sobre la jugabilidad. Finalmente, el 14 de mayo se llevará a cabo el lanzamiento oficial, junto con la publicación de un tráiler, que mostrará las características claves del proyecto.

8.2 PRESUPUESTO

Dado que el proyecto no corresponde con un lanzamiento de un juego comercial oficial, el presupuesto se ha estimado en función del esfuerzo invertido por el equipo en términos de horas de trabajo. Se ha calculado un total de 510 horas de desarrollo.

9.BIBLIOGRAFÍA

Arte: <https://www.sprisers-resource.com/snes/mmx/>

Mas arte: <https://www.sprites-inc.co.uk/sprite.php?local=X/X1>