CENTRO DE ENSEÑANZA TÉCNICA Y SUPERIOR

Pepe Pepinillo

Grupo: 2C

Profesor: Vanessa Miranda Lopez

Ivan Fernandez

Angel Ramirez

Diego Varela

Esteban Colautti

Jose Jimenez

Tijuana, B.C., 30 de Mayo de 2024

Introducción

En este documento, presentamos nuestro proyecto "Pepe el Pepinillo". El objetivo principal es crear un videojuego de plataformas 2D utilizando SFML. Nuestra idea es un pepinillo llamado Pepe, este se enfrenta a una serie de enemigos y obstáculos durante el transcurso del juego, al finalizar el nivel, se enfrentará a un jefe que tendrá que derrotar, En este proyecto intentamos darle un énfasis enfocamos principalmente a la herencia entre clases, para intentar no repetir código que pudiera llegar a ser no útil,

Justificación del uso de la librería SFML

Para este proyecto, utilizamos la librería SFML debido a sus capacidades para manejar gráficos, ventanas y eventos de manera eficiente. Las principales funciones que aplicamos en el código son:

- **Gráficos**: Para dibujar formas geométricas y manejar texturas.
- **Ventana**: Para crear la ventana donde se proyectará el juego.
- **Eventos**: Para registrar la entrada del usuario, como el teclado y el mouse.

Descripción del problema

El principal desafío en el desarrollo de nuestro videojuego de plataformas 2D es el diseño y la implementación de los niveles. Aquí hay algunos puntos clave que consideramos:

- **Diseño de niveles**: Crear niveles interesantes y desafiantes que mantengan a los jugadores comprometidos.
- **Equilibrio de dificultad**: Garantizar una dificultad progresiva y justa.
- **Mecánicas de juego**: Definir controles de movimiento y salto fluidos y responsivos.

- **Variedad de obstáculos y enemigos**: Incluir una variedad de obstáculos y enemigos para mantener el interés del jugador.
- **Ambientación**: Mejorar la experiencia del jugador con una ambientación atractiva.

Requerimientos

- Librería SFML
- Documentación de SFML
- Editor de código (Visual Studio 2022)
- Reuniones grupales (presenciales o por Discord)

Diseño preliminar

El diseño preliminar del juego incluye las siguientes clases:

- **Clase Juego**: Gestión general del juego.
- **Clase Menú**: Implementación del menú principal.
- **Clase Nivel**: Manejo de múltiples niveles.
- **Clase Objeto**: Representación de objetos en el juego.
- **Clase Mecánica**: Lógica de movimiento y física del juego.
- **Clase Personaje**: Movimientos y detección de teclado para el personaje.
- **Clase Estructuras**: Interacción del personaje con las estructuras del juego.

Implementación(final)

Durante el transcurso de la creación del proyecto, estuvimos actualizando las clases, ya que iban surgiendo nuevas, pero finalmente estas son las clases que finalmente quedaron:

Clase AdministradorDeTexturas

```
class AdministradorDeTexturas {
private:
   std::map<std::string, std::shared_ptr<sf::Texture>> texturas;

public:
   bool cargarTextura(const std::string& nombre, const std::string& rutaArchivo) {
      std::shared_ptr<sf::Texture> textura = std::make_shared<sf::Texture>();
      if (!textura->loadFromFile(rutaArchivo)) {
         std::cerr << "Error al cargar la textura: " << rutaArchivo << std::endl;
         return false;
      }
      texturas[nombre] = textura;
      return true;
   }

   std::shared_ptr<sf::Texture> obtenerTextura(const std::string& nombre) {
      return texturas[nombre];
   }

};
```

El AdministradorDeTexturas es muy importante para manejar la carga y el almacenamiento eficiente de texturas en el juego. Utilizamos un puntero para asociar nombres de texturas con objetos sf::Texture. Gracias a este nos permitío reutilizar texturas sin necesidad de cargarlas repetidamente desde el disco, lo que hizo que mejorara mucho el rendimiento. El método cargarTextura intenta cargar una textura desde un archivo y, si tiene éxito, la almacena en el mapa bajo el nombre especificado. Si al cargar la textura falla, se registra un mensaje de error. El método obtenerTextura devuelve un puntero compartido a la textura almacenada, permitiendo acceder fácilmente a cualquier textura que hayamos cargado con anterioridad.

Clase Menu

```
class Menu {
private:
    int seleccionado;
    Font fuente;
    std::vector<Text> menuPrincipal;

public:
    Menu(float ancho, float alto, AdministradorDeTexturas& administradorDeTexturas) : seleccionado(1) {
        fuente.loadFromFile("C:/PepePepinillo/fuente/Sign Rover Layered.ttf");

        std::vector<std::string> opcionesMenu = { "PEPE EL PEPINILLO", "Jugar", "Salir" };
        std::vector<int> tamanosCaracter = { 65, 45, 45, 45 };
        std::vector<Vector2f> posiciones = {
            {ancho / 28, alto / 4},
            {ancho / 8, alto / 4 + 225},
            {ancho / 8, alto / 4 + 325},
            {ancho / 8, alto / 4 + 425}
        };

        for (size_t i = 0; i < opcionesMenu.size(); ++i) {
            Text texto;
            texto.setFont(fuente);
            texto.setString(opcionesMenu[i]);
            texto.setCharacterSize(tamanosCaracter[i]);
            texto.setPosition(posiciones[i]);
            texto.setFillColor(i == seleccionado ? Color(65, 119, 27) : Color(0, 0, 0));
            menuPrincipal.push_back(texto);
        }
    }

    void dibujar(RenderWindow& ventana) {
        for (const auto& item : menuPrincipal) {
            ventana.draw(item);
        }
    }

    void moverArriba() {
        if (seleccionado > 1) {
            menuPrincipal[seleccionado].setFillColor(Color(0, 0, 0));
            seleccionado--;
            menuPrincipal[seleccionado].setFillColor(Color(65, 119, 27));
        }
```

```
    }

    void moverAbajo() {
        if (seleccionado < menuPrincipal.size() - 1) {
            menuPrincipal[seleccionado].setFillColor(Color(0, 0, 0));
            seleccionado++;
            menuPrincipal[seleccionado].setFillColor(Color(65, 119, 27));
        }
    }

    int obtenerSeleccionado() const {
        return seleccionado;
    }
};
```

La clase Menu se encarga de crear la primera ventana con la opción de jugar y de salir. El menú se inicia con una fuente específica y una serie de opciones de texto que se almacenan en un arreglo. Cada caso tiene un tamaño y posición predeterminada en la ventana del juego. El índice seleccionado se utiliza para mantener un registro de cuál opción está actualmente resaltada. La función draw dibuja todas las opciones en la ventana del juego, y moveUp y moveDown permiten al jugador cambiar la selección utilizando las teclas de flecha, cuando el jugador presiona la tecla Enter, el caso seleccionado se define mediante opciónSeleccionada

Clase Mecanicas

```
class Mecanicas {
private:
    float velocidad = 200.0f;
    float velocidadY = 0.f;
    float velocidadSalto = 400.0f;
    float gravedad = 980.0f;
    float aceleracion = 2000.0f;
    float friccion = 2000.0f;
    bool saltando = false;

public:
    bool debajoDeBloque = false;
    float velocidadX = 0.0f;

    float obtenerVelocidadY() const {
        return velocidadY;
```

```
    }

    float obtenerVelocidad() const {
        return velocidad;
    }

    Mecanicas(float gravedad) : gravedad(gravedad) {}

    void movimientoBasico(sf::Sprite& sprite, float deltaTime) {
        int direccionDeseada = 0;

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::A) || sf::Keyboard::isKeyPressed(sf::Keyboard::Left))
{
            direccionDeseada = -1;
        }
        else if (sf::Keyboard::isKeyPressed(sf::Keyboard::D) ||
sf::Keyboard::isKeyPressed(sf::Keyboard::Right)) {
            direccionDeseada = 1;
        }

        if (direccionDeseada != 0) {
            velocidadX += direccionDeseada * aceleracion * deltaTime;
            if (velocidadX > velocidad) velocidadX = velocidad;
            if (velocidadX < -velocidad) velocidadX = -velocidad;
        }
        else {
            if (velocidadX > 0) {
                velocidadX -= friccion * deltaTime;
                if (velocidadX < 0) velocidadX = 0;
            }
            else if (velocidadX < 0) {
                velocidadX += friccion * deltaTime;
                if (velocidadX > 0) velocidadX = 0;
            }
        }

        sprite.move(velocidadX * deltaTime, 0.f);

        if ((sf::Keyboard::isKeyPressed(sf::Keyboard::W) || sf::Keyboard::isKeyPressed(sf::Keyboard::Up))
&& !saltando) {
            saltar();
            saltando = true;
        }
```

```
        if (!debajoDeBloque) {
            velocidadY += gravedad * deltaTime;
        }
        else {
            velocidadY = 0.f;
        }

        sprite.move(0.f, velocidadY * deltaTime);
    }

    void resetearVelocidadY() {
        velocidadY = 0.0f;
    }

    void saltar() {
        velocidadY = -velocidadSalto;
    }

    void setSaltando(bool value) {
        saltando = value;
    }
};
```

Mecánicas controla las físicas y movimientos del personaje principal. Define varios atributos para manejar la velocidad horizontal (velocidad), la velocidad en Y, la velocidad de salto, la gravedad y la fricción. La clase incluye métodos para manejar el movimiento básico del personaje, haciendo que la entrada del teclado se haga a la izquierda, derecha y permitirle saltar. El método movimientoBasico se encarga de actualizar la posición del sprite del personaje según la entrada del jugador y el tiempo transcurrido (deltaTime). Además, la clase maneja la aceleración y desaceleración del personaje y aplica la gravedad cuando no está en el suelo. Los métodos resetearVelocidadY y saltar permiten controlar la velocidad vertical del personaje, la cual es muy importante para realizar los saltos y manejar las caídas.

Clase Objeto

```
class Objeto {
public:
    sf::Sprite sprite;
    std::unique_ptr<Mecanicas> mecanicas;
    bool movible;
```

```
    Objeto(const std::string& rutaImagen, std::unique_ptr<Mecanicas> movimientos, bool movible,
AdministradorDeTexturas& administradorDeTexturas)
        : mecanicas(std::move(movimientos)), movible(movible) {
        sprite.setTexture(*administradorDeTexturas.obtenerTextura(rutaImagen));
    }

    Objeto(sf::Texture& textura, std::unique_ptr<Mecanicas> movimientos, bool movible)
        : mecanicas(std::move(movimientos)), movible(movible) {
        sprite.setTexture(textura);
    }

    virtual void ejecutarMovimiento(sf::RenderWindow& ventana, float deltaTime) {
        if (movible && mecanicas != nullptr) {
            mecanicas->movimientoBasico(sprite, deltaTime);

            sf::Vector2u tamanoVentana = ventana.getSize();

            if (sprite.getPosition().x < 0)
                sprite.setPosition(0, sprite.getPosition().y);
            else if (sprite.getPosition().x + sprite.getGlobalBounds().width > tamanoVentana.x * 3) // Ajustar
tamano de ventana
                sprite.setPosition(tamanoVentana.x * 3 - sprite.getGlobalBounds().width,
sprite.getPosition().y);

            if (sprite.getPosition().y < 0)
                sprite.setPosition(sprite.getPosition().x, 0);
            else if (sprite.getPosition().y + sprite.getGlobalBounds().height > tamanoVentana.y)
                sprite.setPosition(sprite.getPosition().x, tamanoVentana.y - sprite.getGlobalBounds().height);
        }
    }

    virtual void dibujar(sf::RenderWindow& ventana) {
        ventana.draw(sprite);
    }
};
```

La clase Objeto es la clase base para todos los elementos del juego que tienen un sprite y pueden moverse. Cada Objeto tiene un sprite para su representación visual y puede tener un conjunto de mecánicas de movimiento asociadas mediante un puntero único a Mecánicas. El método ejecutarMovimiento aplica las mecánicas de movimiento al sprite del objeto, y dibujar el sprite en la ventana del juego. Esta clase crea una estructura básica para otros objetos más específicos en el juego, permitiendo una gestionar de forma uniforme el movimiento y la visualización.

Clase Personaje

```
class Personaje : public Objeto {
public:
    bool enElSuelo;

    Personaje(const std::string& rutaImagen, std::unique_ptr<Mecanicas> movimientos, bool movible,
AdministradorDeTexturas& administradorDeTexturas)
        : Objeto(rutaImagen, std::move(movimientos), movible, administradorDeTexturas), enElSuelo(false)
{
        sprite.setPosition(50.f, 350.f);
    }

    Personaje(sf::Texture& textura, std::unique_ptr<Mecanicas> movimientos, bool movible)
        : Objeto(textura, std::move(movimientos), movible), enElSuelo(false) {
        sprite.setPosition(50.f, 350.f);
    }

    void setEnElSuelo(bool value) {
        enElSuelo = value;
        if (value) {
            mecanicas->resetearVelocidadY();
            mecanicas->setSaltando(false);
        }
    }

    void ejecutarMovimiento(sf::RenderWindow& ventana, float deltaTime) override {
        mecanicas->movimientoBasico(sprite, deltaTime);

        sf::Vector2u tamanoVentana = ventana.getSize();

        if (sprite.getPosition().x < 0)
            sprite.setPosition(0, sprite.getPosition().y);
        else if (sprite.getPosition().x + sprite.getGlobalBounds().width > tamanoVentana.x * 3) // Ajustar
tamano de ventana
            sprite.setPosition(tamanoVentana.x * 3 - sprite.getGlobalBounds().width, sprite.getPosition().y);

        if (sprite.getPosition().y < 0)
            sprite.setPosition(sprite.getPosition().x, 0);
        else if (sprite.getPosition().y + sprite.getGlobalBounds().height > tamanoVentana.y)
            sprite.setPosition(sprite.getPosition().x, tamanoVentana.y - sprite.getGlobalBounds().height);
    }

    void verificarColisionesConJefe(Boss& jefe);
```

```
};
```

Personaje se hereda de Objeto y representa un personaje jugable. Además de los atributos y métodos de Objeto, Personaje tiene el atributo enElSuelo para conocer si el personaje está en el suelo, esto es muy importante para permitir o prevenir saltos adicionales. setEnElSuelo y ejecutarMovimiento se encargan de ajustar el estado del bool y controlan el movimiento del personaje. Personaje también incluye una función llamada verificarColisionesConJefe para manejar las interacciones con el jefe final del nivel, asegurando que el personaje responda adecuadamente a estas colisiones.

Clase Enemigo

```
class Enemigo : public Personaje {
protected:
    float velocidad;
    float posicionInicialX;
    float posicionInicialY;
    bool eliminado = false;

public:
    Enemigo(const std::string& rutaImagen, std::unique_ptr<Mecanicas> movimientos, bool movible, float
posicionInicialX, float posicionInicialY, AdministradorDeTexturas& administradorDeTexturas, float
velocidad)
        : Personaje(rutaImagen, std::move(movimientos), movible, administradorDeTexturas),
velocidad(velocidad), posicionInicialX(posicionInicialX), posicionInicialY(posicionInicialY) {
        sprite.setPosition(posicionInicialX, posicionInicialY);
    }

    void marcarEliminado() {
        eliminado = true;
    }

    bool estaEliminado() const {
        return eliminado;
    }

    virtual void mover(float deltaTime) = 0;

    void setEscala(float escalaX, float escalaY) {
        sprite.setScale(escalaX, escalaY);
    }
};
```

Enemigo está conectada Personaje para representar a los enemigos en el juego. Cada Enemigo tiene sus atributos adicionales como velocidad, posicionInicialX, posicionInicialY y una función eliminado para determinar si el hemos matado al enemigo. En la clase hay una función virtual mover, que debe ser implementada por las subclases para especificar cómo se mueve cada tipo de enemigo. Las funciones marcarEliminado y estaEliminado permiten gestionar el estado del enemigo, marcándolo como eliminado y verificando el estado.

Clase EnemigoTerrestre

```
class EnemigoTerrestre : public Enemigo {
private:
   bool moverDerecha = true;

public:
   EnemigoTerrestre(const std::string& rutaImagen, std::unique_ptr<Mecanicas> movimientos, bool
movible, float posicionInicialX, float posicionInicialY, AdministradorDeTexturas&
administradorDeTexturas, float velocidad)
      : Enemigo(rutaImagen, std::move(movimientos), movible, posicionInicialX, posicionInicialY,
administradorDeTexturas, velocidad) {}

   void mover(float deltaTime) override {
      if (moverDerecha) {
         sprite.move(velocidad * deltaTime, 0.f);
         if (sprite.getPosition().x >= posicionInicialX + 100.f) {
            moverDerecha = false;
         }
      }
      else {
         sprite.move(-velocidad * deltaTime, 0.f);
         if (sprite.getPosition().x <= posicionInicialX) {
            moverDerecha = true;
         }
      }
   }
};
```

EnemigoTerrestre es una subclase de Enemigo que representa a todos los enemigos que se mueven sobre el suelo. Se mueven de derecha a izquierda dentro de un rango definido, alternando la dirección del movimiento cuando se alcanza los límites de su rango. Este comportamiento se controla mediante un indicador moverDerecha. El método mover implementa

este patrón de movimiento, asegurando que el enemigo se mueva hacia adelante y hacia atrás de manera predecible.

Clase EnemigoSaltarin

```
class EnemigoSaltarin : public Enemigo {
private:
   bool moverArriba = true;

public:
   EnemigoSaltarin(const std::string& rutaImagen, std::unique_ptr<Mecanicas> movimientos, bool
movible, float posicionInicialX, float posicionInicialY, AdministradorDeTexturas&
administradorDeTexturas, float velocidad)
      : Enemigo(rutaImagen, std::move(movimientos), movible, posicionInicialX, posicionInicialY,
administradorDeTexturas, velocidad) {}

   void mover(float deltaTime) override {
      if (moverArriba) {
         sprite.move(0.f, -velocidad * deltaTime);
         if (sprite.getPosition().y <= posicionInicialY - 50.f) {
            moverArriba = false;
         }
      }
      else {
         sprite.move(0.f, velocidad * deltaTime);
         if (sprite.getPosition().y >= posicionInicialY + 50.f) {
            moverArriba = true;
         }
      }
   }
};
```

EnemigoSaltarin es otra subclase de Enemigo la cual representa a los enemigos que saltan verticalmente. Estos enemigos alternan entre moverse hacia arriba y hacia abajo dentro de un rango específico, utilizando la función moverArriba para controlar la dirección del movimiento. La función mover implementa este comportamiento, proporcionando una dificultad extra para el jugador que debe evitar o saltar sobre estos enemigos en el momento adecuado.

Clase EnemigoVolador

```
class EnemigoVolador : public Enemigo {
private:
    bool moverDerecha = true;
    bool moverArriba = true;

public:
    EnemigoVolador(const std::string& rutaImagen, std::unique_ptr<Mecanicas> movimientos, bool
movible, float posicionInicialX, float posicionInicialY, AdministradorDeTexturas&
administradorDeTexturas, float velocidad)
        : Enemigo(rutaImagen, std::move(movimientos), movible, posicionInicialX, posicionInicialY,
administradorDeTexturas, velocidad) {}

    void mover(float deltaTime) override {
        if (moverDerecha) {
            sprite.move(velocidad * deltaTime, 0.f);
            if (sprite.getPosition().x >= posicionInicialX + 100.f) {
                moverDerecha = false;
            }
        }
        else {
            sprite.move(-velocidad * deltaTime, 0.f);
            if (sprite.getPosition().x <= posicionInicialX) {
                moverDerecha = true;
            }
        }

        if (moverArriba) {
            sprite.move(0.f, -velocidad * deltaTime);
            if (sprite.getPosition().y <= posicionInicialY - 50.f) {
                moverArriba = false;
            }
        }
        else {
            sprite.move(0.f, velocidad * deltaTime);
            if (sprite.getPosition().y >= posicionInicialY + 50.f) {
                moverArriba = true;
            }
        }
    }
};
```

EnemigoVolador es la última clase de enemigos comunes que pertenece a enemigo , esta se encarga del comportamiento de los enemigos que se mueven tanto horizontal como vertical. Estos enemigos tienen un patrón de movimiento algo más complejo, combinando desplazamiento lateral, ascendente y descendente. Las funciones moverDerecha y moverArriba se encargan controlar las direcciones de movimiento, y mover asegura que el enemigo se desplace dentro de su rango definido.

Clase Boss

```
class Boss {
private:
    sf::Clock clock;
    float speed;
    int attackPattern;
    bool isImmobilized;
    sf::Clock immobilizedClock;
    int health;
    float limiteEscalerasIzq;
    float limiteEscalerasDer;
    bool moveRight = true;
    bool jumping = false;
    float initialPosY;
    float rangoVision;
    bool eliminado;
    sf::Sprite spriteperro;

public:
    Boss(const std::string& nombre, const std::string& rutaImagen, float posX, float posY,
AdministradorDeTexturas& administradorDeTexturas, float limiteIzq, float limiteDer)
        : limiteEscalerasIzq(limiteIzq), limiteEscalerasDer(limiteDer), initialPosY(posY),
rangoVision(300.0f), health(6), eliminado(false) {
        if (!administradorDeTexturas.cargarTextura(nombre, rutaImagen)) {
            std::cerr << "Error al cargar la textura del jefe: " << rutaImagen << std::endl;
            exit(1);
        }
        spriteperro.setTexture(*administradorDeTexturas.obtenerTextura(nombre));
        spriteperro.setPosition(posX, posY);
        spriteperro.setScale(0.7f, 0.7f); // Ajustar la escala del jefe para hacerlo más pequeño
        speed = 110.0f;
        attackPattern = 0;
        isImmobilized = false;
```

```
    health = 6;
}

void update(float deltaTime, sf::Vector2f playerPos) {
    if (isImmobilized) {
        if (immobilizedClock.getElapsedTime().asSeconds() > 3.0f) {
            isImmobilized = false;
            attackPattern++;
            if (attackPattern >= 6) {
                attackPattern = 0;
            }
        }
        return;
    }

    sf::Vector2f direction = playerPos - spriteperro.getPosition();
    float length = sqrt(direction.x * direction.x + direction.y * direction.y);
    direction /= length;

    switch (attackPattern % 3) {
    case 0:
        if (length < rangoVision) {
            spriteperro.move(direction.x * speed * deltaTime, 0.f);
        }
        else {
            if (moveRight) {
                spriteperro.move(speed * deltaTime, 0.f);
                if (spriteperro.getPosition().x > limiteEscalerasDer) moveRight = false;
            }
            else {
                spriteperro.move(-speed * deltaTime, 0.f);
                if (spriteperro.getPosition().x < limiteEscalerasIzq) moveRight = true;
            }
        }
        break;
    case 1:
        if (!jumping) {
            spriteperro.move(direction.x * speed * deltaTime, -speed * 2 * deltaTime);
            if (spriteperro.getPosition().y <= initialPosY - 100.f) {
                jumping = true;
            }
        }
        else {
            spriteperro.move(direction.x * speed * deltaTime, speed * 2 * deltaTime);
```

```
            if (spriteperro.getPosition().y >= initialPosY) {
                jumping = false;
            }
        }
        break;
    case 2:
        spriteperro.move(direction * speed * deltaTime * 1.5f);
        break;
    }

    if (spriteperro.getGlobalBounds().intersects(sf::FloatRect(playerPos, sf::Vector2f(50.0f, 50.0f)))) {
        if (playerPos.y + 50.0f <= spriteperro.getPosition().y + 10.0f) {
            perderVida();
            isImmobilized = true;
            immobilizedClock.restart();
            if (health <= 0) {
                spriteperro.setPosition(-100, -100);
            }
        }
    }
}

void draw(sf::RenderWindow& window) {
    window.draw(spriteperro);
}

void reset() {
    spriteperro.setPosition(2500.0f, 300.0f); // Reemplaza estos valores con la posición inicial adecuada
    health = 6;
    isImmobilized = false;
    attackPattern = 0;
}

void perderVida() {
    health--;
}

int getVidas() {
    return health;
}

bool estaEliminado() const {
    return health <= 0;
}
```

```
    void manejarColisiones(Personaje& personaje) {
        if (spriteperro.getGlobalBounds().intersects(personaje.sprite.getGlobalBounds())) {
            // Colisión desde arriba
            if (personaje.sprite.getPosition().y + personaje.sprite.getGlobalBounds().height <=
spriteperro.getPosition().y + 10.f) {
                personaje.mecanicas->saltar();
                personaje.setEnElSuelo(false);
                perderVida();
                isImmobilized = true;
                immobilizedClock.restart();
                if (health <= 0) {
                    spriteperro.setPosition(-100, -100); // Jefe eliminado
                }
            }
            else {
                // Reiniciar la posición del personaje
                personaje.sprite.setPosition(50.f, 316.f); // Ajustar la posición de reinicio de Pepe
                // Reiniciar la posición del jefe y el patrón de ataque
                reiniciarPosicion();
                attackPattern = 0; // Reiniciar el patrón de ataque
                isImmobilized = false; // Reiniciar la inmovilización
                immobilizedClock.restart(); // Reiniciar el reloj de inmovilización
            }
        }
    }

    void reiniciarPosicion() {
        spriteperro.setPosition(2500.0f, 300.0f); // Reemplaza estos valores con la posición inicial adecuada
    }
};
```

La clase Boss representa al jefe final del nivel, un enemigo más grande y complicado con varias fases de ataque. Los atributos incluyen relojes para manejar el tiempo y la inmovilización, speed para la velocidad, attackPattern para el patrón de ataque actual, y health para la vida del jefe. Durante la pelea, el jefe puede cambiar su patrón de ataque en función de su estado y la proximidad del jugador. Las funciones update y draw manejan la actualización y dibujo del jefe, mientras que perderVida y manejarColisiones controlan el daño recibido y las interacciones con el jugador. Reset reinicia el estado del jefe, esto lo pensamos por que el personaje no te dejaba de seguir incluso si te había matado entonces nos encargamos de crearla.

Clase Bloque

```
class Bloque : public Objeto {
public:
   Bloque(const std::string& rutaImagen, float x, float y, bool movible, AdministradorDeTexturas&
administradorDeTexturas)
       : Objeto(rutaImagen, nullptr, movible, administradorDeTexturas) {
       sprite.setPosition(x, y);
       sprite.setScale(34.f / administradorDeTexturas.obtenerTextura(rutaImagen)->getSize().x, 34.f /
administradorDeTexturas.obtenerTextura(rutaImagen)->getSize().y);
   }

   Bloque(Bloque&&) = default; // Constructor de movimiento
   Bloque& operator=(Bloque&&) = default; // Operador de asignación por movimiento

   sf::FloatRect obtenerLimitesGlobales() {
       return sprite.getGlobalBounds();
   }
};
```

Bloque se hereda de Objeto y esta plasma los elementos estáticos del entorno del juego, básicamente los bloques. Estos bloques forman el piso del nivel y algunas superficies sobre las que el jugador puede moverse. La función obtenerLimitesGlobales devuelve los límites del bloque en coordenadas globales, lo que es bastante importante ya que maneja las colisiones con otros objetos en el juego. Esta clase es importante para definir el terreno del juego y establecer límites físicos para los personajes y enemigos.

Clase PlataformaMovil

```
class PlataformaMovil : public Bloque {
private:
   float velocidad;
   bool moverDerecha = true;
   float limiteIzquierdo, limiteDerecho;

public:
   PlataformaMovil(const std::string& rutaImagen, float x, float y, float velocidad, float
limiteIzquierdo, float limiteDerecho, AdministradorDeTexturas& administradorDeTexturas)
       : Bloque(rutaImagen, x, y, false, administradorDeTexturas), velocidad(velocidad),
limiteIzquierdo(limiteIzquierdo), limiteDerecho(limiteDerecho) {}
```

```
    void mover(float deltaTime) {
        if (moverDerecha) {
            sprite.move(velocidad * deltaTime, 0.f);
            if (sprite.getPosition().x >= limiteDerecho) {
                moverDerecha = false;
            }
        }
        else {
            sprite.move(-velocidad * deltaTime, 0.f);
            if (sprite.getPosition().x <= limiteIzquierdo) {
                moverDerecha = true;
            }
        }
    }

    sf::Vector2f obtenerVelocidad() const {
        return moverDerecha ? sf::Vector2f(velocidad, 0.f) : sf::Vector2f(-velocidad, 0.f);
    }
};
```

PlataformaMovil es una subclase de Bloque que básicamente crea el comportamiento de las plataformas. Estas plataformas se desplazan entre dos límites definidos, moviendo su dirección cuando alcanzan estos límites. Los atributos incluyen velocidad para la velocidad de movimiento y limiteIzquierdo y limiteDerecho para los límites del movimiento. La función mover ajusta la posición de la plataforma dentro de estos límites, proporcionando el reto más difícil para el jugador dentro del juego, este debe sincronizar sus movimientos con las plataformas móviles, para no caerse al vacío y reiniciar desde el spawn.

Clase Nivel

```
class Nivel {

private:
    sf::Texture texturaFondo;
    sf::Sprite spriteFondo;
    sf::RenderWindow& ventana;
    std::vector<std::unique_ptr<Bloque>> bloques; // Cambiar a vector de unique_ptr
    std::vector<std::unique_ptr<PlataformaMovil>> plataformasMoviles; // Vector para plataformas móviles
    View& camara;
```

```
    std::unique_ptr<Boss> jefeFinal; // Alex: Uso de unique_ptr para el jefe final

public:
    Nivel(const std::string& rutaImagenFondo, sf::RenderWindow& ventana, View& camara,
AdministradorDeTexturas& administradorDeTexturas)
        : ventana(ventana), camara(camara) {

        if (!texturaFondo.loadFromFile(rutaImagenFondo)) {
            std::cerr << "Error cargando la textura del fondo desde: " << rutaImagenFondo << std::endl;
            exit(1);
        }

        spriteFondo.setTexture(texturaFondo);

        sf::Vector2u tamanoTextura = texturaFondo.getSize();
        sf::Vector2u tamanoVentana = ventana.getSize();

        float escalaX = static_cast<float>(tamanoVentana.x * 3) / tamanoTextura.x; // Ajuste de nivel más
largo
        float escalaY = static_cast<float>(tamanoVentana.y) / tamanoTextura.y;

        float escala = std::max(escalaX, escalaY);
        spriteFondo.setScale(escala, escala);

        // Alex: Inicialización del jefe final
        jefeFinal = std::make_unique<Boss>("Boss1", "C:/PepePepinillo/jefe.png", 2500.0f, 300.0f,
administradorDeTexturas, 1900.0f, 2600.0f);
    }

    void dibujar Fondo(sf::RenderWindow& ventana) {
        ventana.draw(spriteFondo);
    }

    void dibujarBloques(sf::RenderWindow& ventana) {
        for (auto& bloque : bloques) {
            bloque->dibujar(ventana);
        }
    }

    void dibujarPlataformasMoviles(sf::RenderWindow& ventana) {
        for (auto& plataforma : plataformasMoviles) {
            plataforma->dibujar(ventana);
        }
    }

    void moverPlataformasMoviles(float deltaTime) {
        for (auto& plataforma : plataformasMoviles) {
            plataforma->mover(deltaTime);
```

```
        }
    }

    void dibujarJefeFinal(sf::RenderWindow& ventana); // Alex: Método para dibujar el jefe final
    void moverJefeFinal(float deltaTime, sf::Vector2f posicionJugador); // Alex: Método para mover el jefe
final


    std::vector<std::unique_ptr<Bloque>>& obtenerBloques() {
        return bloques;
    }

    std::vector<std::unique_ptr<PlataformaMovil>>& obtenerPlataformasMoviles() {
        return plataformasMoviles;
    }

    std::unique_ptr<Boss>& obtenerJefeFinal() {
        return jefeFinal;
    }

    void crearBloques(AdministradorDeTexturas& administradorDeTexturas, int nivel,
std::vector<std::unique_ptr<Enemigo>>& enemigos) {
        bloques.clear();  // Limpiar bloques existentes
        plataformasMoviles.clear(); // Limpiar plataformas móviles existentes
        enemigos.clear(); // Limpiar enemigos existentes


        switch (nivel) {
            //Primer nivel
        case 1: {
            // Lógica para crear los bloques del nivel 1
            float anchoBloque = 34.0f;
            float altoBloque = 34.0f;
            float alturaSuelo = 350.0f;
            int numBloques = static_cast<int>(ventana.getSize().x / anchoBloque) + 1;

            //Bloques piso
            for (int i = 0; i < numBloques * 3; ++i) { // Aumentar el número de bloques 3 veces
                cout << "Cantidad de bloques: " << i << endl;
                int contadorPiso = i;

                //Agregar huecos
                if (contadorPiso > 9 && contadorPiso < 12 || contadorPiso > 20 && contadorPiso < 23 ||
contadorPiso > 27 && contadorPiso < 30 || contadorPiso > 40 && contadorPiso < 50) {
                    continue;

                }
                float posXBloque = i * anchoBloque;
```

```
            bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png",
posXBloque, alturaSuelo, false, administradorDeTexturas));
        }

        //Agregar bloques flotantes
        bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png", 250.0f,
280.0f, false, administradorDeTexturas));
        bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png", 318.0f,
212.0f, false, administradorDeTexturas));

        bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png", 488.0f,
246.0f, false, administradorDeTexturas));
        bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png", 520.0f,
246.0f, false, administradorDeTexturas));
        bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png", 554.0f,
246.0f, false, administradorDeTexturas));
        bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png", 1500.0f,
300.0f, false, administradorDeTexturas));
        bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png", 1550.0f,
300.0f, false, administradorDeTexturas));
        bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png", 1300.f,
150.0f, false, administradorDeTexturas));
        bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png", 1843.f,
284.f, false, administradorDeTexturas));

        //Tres bloques arriba de otro
        bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png", 1986.f,
318.f, false, administradorDeTexturas));
        bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png", 1986.f,
284.f, false, administradorDeTexturas));
        bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png", 1986.f,
250.f, false, administradorDeTexturas));

        bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png", 2736.f,
318.f, false, administradorDeTexturas));
        bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png", 2736.f,
284.f, false, administradorDeTexturas));
        bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png", 2736.f,
250.f, false, administradorDeTexturas));




        float posXEscalera = 680.0f;
        float posYEscalera = alturaSuelo + altoBloque;
        int numEscalones = 5;

        //Agregar escaleras
```

```
        for (int i = 0; i < numEscalones; i++) {
            bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png",
posXEscalera, posYEscalera, false, administradorDeTexturas));
            posXEscalera += anchoBloque;
            posYEscalera -= altoBloque;
        }

        bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png", 986.0f,
212.0f, false, administradorDeTexturas));


        // Agregar plataformas móviles

plataformasMoviles.push_back(std::make_unique<PlataformaMovil>("C:/PepePepinillo/plataforma.png",
1050.f, 150.f, 100.f, 1360.f, 1700.f, administradorDeTexturas));

//plataformasMoviles.push_back(std::make_unique<PlataformaMovil>("C:/PepePepinillo/plataforma.png",
800.f, 150.f, 120.f, 750.f, 850.f, administradorDeTexturas));

        // Agregar enemigos
        float posYEnemigo = alturaSuelo - 34.f;
        float velocidadEnemigos = 100.0f;

        //Enemigos terrestres
        enemigos.push_back(std::make_unique<EnemigoTerrestre>("C:/PepePepinillo/malo.png",
std::make_unique<Mecanicas>(980.0f), true, 100.f, posYEnemigo, administradorDeTexturas,
velocidadEnemigos));
        enemigos.push_back(std::make_unique<EnemigoTerrestre>("C:/PepePepinillo/malo.png",
std::make_unique<Mecanicas>(980.0f), true, 400.f, posYEnemigo, administradorDeTexturas,
velocidadEnemigos));
        enemigos.push_back(std::make_unique<EnemigoTerrestre>("C:/PepePepinillo/malo.png",
std::make_unique<Mecanicas>(980.0f), true, 620.f, posYEnemigo, administradorDeTexturas,
velocidadEnemigos));
        enemigos.push_back(std::make_unique<EnemigoTerrestre>("C:/PepePepinillo/malo.png",
std::make_unique<Mecanicas>(980.0f), true, 1100.f, posYEnemigo, administradorDeTexturas,
velocidadEnemigos));

        enemigos.push_back(std::make_unique<EnemigoTerrestre>("C:/PepePepinillo/malo.png",
std::make_unique<Mecanicas>(980.0f), true, 1200.f, posYEnemigo, administradorDeTexturas,
velocidadEnemigos));
        enemigos.push_back(std::make_unique<EnemigoTerrestre>("C:/PepePepinillo/malo.png",
std::make_unique<Mecanicas>(980.0f), true, 1350.f, posYEnemigo, administradorDeTexturas,
velocidadEnemigos));
        enemigos.push_back(std::make_unique<EnemigoTerrestre>("C:/PepePepinillo/malo.png",
std::make_unique<Mecanicas>(980.0f), true, 2111.f, posYEnemigo, administradorDeTexturas,
velocidadEnemigos));
        enemigos.push_back(std::make_unique<EnemigoTerrestre>("C:/PepePepinillo/malo.png",
std::make_unique<Mecanicas>(980.0f), true, 2211.f, posYEnemigo, administradorDeTexturas,
velocidadEnemigos + 20));
```

```
            enemigos.push_back(std::make_unique<EnemigoTerrestre>("C:/PepePepinillo/malo.png",
std::make_unique<Mecanicas>(980.0f), true, 2311.f, posYEnemigo, administradorDeTexturas,
velocidadEnemigos - 20));
            enemigos.push_back(std::make_unique<EnemigoTerrestre>("C:/PepePepinillo/malo.png",
std::make_unique<Mecanicas>(980.0f), true, 2500.f, posYEnemigo, administradorDeTexturas,
velocidadEnemigos + 30));

            //Enemigos voladores
            enemigos.push_back(std::make_unique<EnemigoVolador>("C:/PepePepinillo/maloVolador.png",
std::make_unique<Mecanicas>(980.0f), true, 1200.f, 200.f, administradorDeTexturas,
velocidadEnemigos));
            enemigos.push_back(std::make_unique<EnemigoVolador>("C:/PepePepinillo/maloVolador.png",
std::make_unique<Mecanicas>(980.0f), true, 1530.f, 200.f, administradorDeTexturas,
velocidadEnemigos));
            //enemigos.push_back(std::make_unique<EnemigoVolador>("C:/PepePepinillo/maloVolador.png",
std::make_unique<Mecanicas>(980.0f), true, 400.f, 100.f, administradorDeTexturas,
velocidadEnemigos));

            //Enemigos saltarines
            enemigos.push_back(std::make_unique<EnemigoSaltarin>("C:/PepePepinillo/malo.png",
std::make_unique<Mecanicas>(980.0f), true, 660.f, 240.f, administradorDeTexturas,
velocidadEnemigos));
            enemigos.back()->setEscala(2.0f, 2.0f);
            enemigos.push_back(std::make_unique<EnemigoSaltarin>("C:/PepePepinillo/maloVolador.png",
std::make_unique<Mecanicas>(980.0f), true, 1050.f, 240.f, administradorDeTexturas,
velocidadEnemigos));
            enemigos.back()->setEscala(2.0f, 2.0f);
            enemigos.push_back(std::make_unique<EnemigoSaltarin>("C:/PepePepinillo/maloVolador.png",
std::make_unique<Mecanicas>(980.0f), true, 2650.f, 240.f, administradorDeTexturas,
velocidadEnemigos));
            enemigos.back()->setEscala(2.0f, 2.0f);

            //Jefe final
            jefeFinal = std::make_unique<Boss>("Boss1", "C:/PepePepinillo/jefe.png", 2500.0f, 300.0f,
administradorDeTexturas, 2000.0f, 2600.0f);

            break;
        }
        case 2: {
            // Lógica para crear los bloques del nivel 2 con muchos huecos grandes y más enemigos aéreos
            float anchoBloque = 34.0f;
            float altoBloque = 34.0f;
            float alturaSuelo = 350.0f;
            int numBloques = static_cast<int>(ventana.getSize().x / anchoBloque) + 1;

            for (int i = 0; i < numBloques * 3; ++i) { // Aumentar el número de bloques 3 veces
                int contadorPiso = i;

                //Agregar huecos
```

```
            if (contadorPiso > 10 && contadorPiso < 20 || contadorPiso > 36 && contadorPiso < 90) {
                continue;

            }
            float posXBloque = i * anchoBloque;
            bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png",
posXBloque, alturaSuelo, false, administradorDeTexturas));
        }

        //Para agregar bloques en el aire
        //bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png", 150.0f,
250.0f, false, administradorDeTexturas));
        //bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png", 300.0f,
200.0f, false, administradorDeTexturas));

        //Para crear piso en el aire

        float posXPlataforma = 400.0f;
        float posYPlataforma = 300.0f;
        int numPlataformas = 4;

        for (int i = 0; i < numPlataformas; i++) {
            bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png",
posXPlataforma, posYPlataforma, false, administradorDeTexturas));
            posXPlataforma += anchoBloque;
        }

        posXPlataforma = 1750.0f;
        posYPlataforma = 180.0f;
        numPlataformas = 15;

        for (int i = 0; i < numPlataformas; i++) {
            bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png",
posXPlataforma, posYPlataforma, false, administradorDeTexturas));
            posXPlataforma += anchoBloque;
        }


        float posXEscalera = 1000.0f;
        float posYEscalera = alturaSuelo + altoBloque;
        int numEscalones = 5;

        //Agregar escaleras
        for (int i = 0; i < numEscalones; i++) {
            bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png",
posXEscalera, posYEscalera, false, administradorDeTexturas));
            posXEscalera += anchoBloque;
            posYEscalera -= altoBloque;
        }
```

```
        posXEscalera = 2240.0f;
        posYEscalera = 180;
        numEscalones = 3;

        for (int i = 0; i < numEscalones; i++) {
            bloques.push_back(std::make_unique<Bloque>("C:/PepePepinillo/bloqueChido.png",
posXEscalera, posYEscalera, false, administradorDeTexturas));
            posXEscalera += anchoBloque;
            posYEscalera -= altoBloque;
        }

        // Ivan: Agregar plataformas móviles al nivel 2

plataformasMoviles.push_back(std::make_unique<PlataformaMovil>("C:/PepePepinillo/plataforma.png",
600.f, 250.f, 90.f, 550.f, 650.f, administradorDeTexturas));

plataformasMoviles.push_back(std::make_unique<PlataformaMovil>("C:/PepePepinillo/plataforma.png",
1200.f, 200.f, 110.f, 1150.f, 1250.f, administradorDeTexturas));

plataformasMoviles.push_back(std::make_unique<PlataformaMovil>("C:/PepePepinillo/plataforma.png",
1350.f, 200.f, 110.f, 1300.f, 1400.f, administradorDeTexturas));

plataformasMoviles.push_back(std::make_unique<PlataformaMovil>("C:/PepePepinillo/plataforma.png",
1550.f, 200.f, 110.f, 1500.f, 1700.f, administradorDeTexturas));

plataformasMoviles.push_back(std::make_unique<PlataformaMovil>("C:/PepePepinillo/plataforma.png",
2500.f, 200.f, 110.f, 2342.f, 2600.f, administradorDeTexturas));

plataformasMoviles.push_back(std::make_unique<PlataformaMovil>("C:/PepePepinillo/plataforma.png",
2800.f, 250.f, 150.f, 2700.f, 3000.f, administradorDeTexturas));

plataformasMoviles.push_back(std::make_unique<PlataformaMovil>("C:/PepePepinillo/plataforma.png",
3200.f, 250.f, 110.f, 3100.f, 3300.f, administradorDeTexturas));

        float posYEnemigo = alturaSuelo - 34.f;
        float velocidadEnemigos = 100.0f;
        //Enemigos voladores
        enemigos.push_back(std::make_unique<EnemigoVolador>("C:/PepePepinillo/maloVolador.png",
std::make_unique<Mecanicas>(980.0f), true, 200.f, 200.f, administradorDeTexturas,
velocidadEnemigos));
        enemigos.push_back(std::make_unique<EnemigoVolador>("C:/PepePepinillo/maloVolador.png",
std::make_unique<Mecanicas>(980.0f), true, 600.f, 150.f, administradorDeTexturas,
velocidadEnemigos));
        enemigos.push_back(std::make_unique<EnemigoVolador>("C:/PepePepinillo/maloVolador.png",
std::make_unique<Mecanicas>(980.0f), true, 1000.f, 100.f, administradorDeTexturas,
velocidadEnemigos));
```

```
            enemigos.push_back(std::make_unique<EnemigoVolador>("C:/PepePepinillo/maloVolador.png",
std::make_unique<Mecanicas>(980.0f), true, 1400.f, 50.f, administradorDeTexturas,
velocidadEnemigos));
            enemigos.push_back(std::make_unique<EnemigoVolador>("C:/PepePepinillo/maloVolador.png",
std::make_unique<Mecanicas>(980.0f), true, 1800.f, 0.f, administradorDeTexturas, velocidadEnemigos));
            enemigos.push_back(std::make_unique<EnemigoVolador>("C:/PepePepinillo/maloVolador.png",
std::make_unique<Mecanicas>(980.0f), true, 2500.f, 100.f, administradorDeTexturas,
velocidadEnemigos));
            enemigos.push_back(std::make_unique<EnemigoVolador>("C:/PepePepinillo/maloVolador.png",
std::make_unique<Mecanicas>(980.0f), true, 2850.f, 150.f, administradorDeTexturas,
velocidadEnemigos));
            enemigos.push_back(std::make_unique<EnemigoVolador>("C:/PepePepinillo/maloVolador.png",
std::make_unique<Mecanicas>(980.0f), true, 3000.f, 100.f, administradorDeTexturas,
velocidadEnemigos));

            //Enemigos terrestres
            enemigos.push_back(std::make_unique<EnemigoTerrestre>("C:/PepePepinillo/malo.png",
std::make_unique<Mecanicas>(980.0f), true, 100.f, posYEnemigo, administradorDeTexturas,
velocidadEnemigos));
            enemigos.push_back(std::make_unique<EnemigoTerrestre>("C:/PepePepinillo/malo.png",
std::make_unique<Mecanicas>(980.0f), true, 400.f, posYEnemigo - 64, administradorDeTexturas,
velocidadEnemigos));
            enemigos.push_back(std::make_unique<EnemigoTerrestre>("C:/PepePepinillo/malo.png",
std::make_unique<Mecanicas>(980.0f), true, 2005.f, 145, administradorDeTexturas,
velocidadEnemigos));
            enemigos.push_back(std::make_unique<EnemigoTerrestre>("C:/PepePepinillo/malo.png",
std::make_unique<Mecanicas>(980.0f), true, 2105.f, 145, administradorDeTexturas, velocidadEnemigos
+ 25));
            //Enemigos saltarines
            enemigos.push_back(std::make_unique<EnemigoSaltarin>("C:/PepePepinillo/malo.png",
std::make_unique<Mecanicas>(980.0f), true, 900.f, 240.f, administradorDeTexturas,
velocidadEnemigos));
            enemigos.back()->setEscala(2.0f, 2.0f);
            enemigos.push_back(std::make_unique<EnemigoSaltarin>("C:/PepePepinillo/malo.png",
std::make_unique<Mecanicas>(980.0f), true, 3120.f, 240.f, administradorDeTexturas,
velocidadEnemigos));
            enemigos.back()->setEscala(2.0f, 2.0f);


            break;
        }
        default:
            break;
        }
    }
};
```

Esta clase es la segunda clase más importante del juego, por que tiene muchas funciones, la principal, es la de cargar los niveles estos funcionan mediante un vector y un switch, por falta de tiempo solamente pudimos hacer dos casos, pero estos básicamente se almacenan en un vector y a partir de este se despliega el nivel, pero también se encarga de guardar en un vector a los enemigos para poder darles las texturas, otra de las funciones es el acomodo del fullscreen, cuando utilizas full screen se reajusta todo, esto lo hace jugable de igual manera.

Clase Juego

```
class Juego {
private:
    View camara;
    int puntaje;
    int enemigosDerrotados;
    int vecesMuerto;
    bool pantallaCompleta = false;
    sf::Clock reloj;
    std::vector<std::unique_ptr<Enemigo>> enemigos;
    AdministradorDeTexturas administradorDeTexturas;
    sf::Text textoPuntaje;
    sf::Font fuentePuntaje;

public:
    Juego();
    sf::RenderWindow ventana;

    void alternarPantallaCompleta() {
        pantallaCompleta = !pantallaCompleta;
        ventana.close();
        if (pantallaCompleta) {
            ventana.create(VideoMode::getDesktopMode(), "Pepe el pepinillo", Style::Fullscreen);
        }
        else {
            ventana.create(VideoMode(1200, 720), "Pepe el pepinillo", Style::Default);
        }
        ventana.setView(camara);

        ventana.setFramerateLimit(60);
        ventana.setVerticalSyncEnabled(true);
    }

    void ejecutar() {
        RenderWindow ventana(VideoMode(1200, 720), "Juego");
        Menu menu(1200, 720, administradorDeTexturas);
```

```
Texture texturaFondoMenu;
texturaFondoMenu.loadFromFile("C:/PepePepinillo/fondoMenuSimple.jpg");
Sprite fondoMenu;
fondoMenu.setTexture(texturaFondoMenu);

while (true) {
    if (numPagina == 1000) {
        while (ventana.isOpen()) {
            Event evento;
            while (ventana.pollEvent(evento)) {
                if (evento.type == Event::Closed) {
                    ventana.close();
                    break;
                }
                if (evento.type == Event::KeyPressed) {
                    if (evento.key.code == Keyboard::W || evento.key.code == Keyboard::Up) {
                        menu.moverArriba();
                    }
                    if (evento.key.code == Keyboard::S || evento.key.code == Keyboard::Down) {
                        menu.moverAbajo();
                    }
                    if (evento.key.code == Keyboard::Return) {
                        if (menu.obtenerSeleccionado() == 1) {
                            numPagina = 1;
                        }
                        if (menu.obtenerSeleccionado() == 2) {
                            numPagina = 2;
                        }
                        if (menu.obtenerSeleccionado() == 3) {
                            numPagina = -1;
                        }
                    }
                }
            }
            ventana.clear();
            if (numPagina != 1000) {
                break;
            }
            ventana.draw(fondoMenu);
            menu.dibujar(ventana);
            ventana.display();
        }
        if (numPagina == -1) {
            ventana.close();
            break;
        }
        if (numPagina == 1) {
            ventana.close();
            iniciarJuego();
```

```
                break;
            }
            if (numPagina == 2) {
                // Instrucciones
            }
        }
    }
}

void iniciarJuego() {
    iniciarNivel(1);  // Comenzar con el nivel 1
}

void iniciarNivel(int nivel) {
    bool enPausa = false;
    Nivel nivelJuego("C:/PepePepinillo/fondo.png", ventana, camara, administradorDeTexturas);
    nivelJuego.crearBloques(administradorDeTexturas, nivel, enemigos);

    auto mecanicasPepe = std::make_unique<Mecanicas>(980.0f); // Establecer gravedad
correctamente
    Personaje pepe("C:/PepePepinillo/pepe.png", std::move(mecanicasPepe), true,
administradorDeTexturas);
    pepe.sprite.setPosition(50.f, 316.f); // Ajustar la posición inicial de Pepe

    while (ventana.isOpen()) {
        sf::Event evento;
        while (ventana.pollEvent(evento)) {
            if (evento.type == Event::Closed)
                ventana.close();
            if (evento.key.code == Keyboard::Escape && evento.type == Event::KeyPressed) {
                ventana.close();
            }
            if (evento.key.code == Keyboard::F11 && evento.type == Event::KeyPressed) {
                alternate Pantalla Completa();
            }
            if (evento.key.code == Keyboard::N && evento.type == Event::KeyPressed) {
                nivelActual = (nivelActual % 2) + 1;  // Cambiar de nivel
                iniciarNivel(nivelActual);
                return;  // Salir del bucle y reiniciar el nivel
            }
        }

        float deltaTime = reloj.restart().asSeconds();

        Vector2f pepePosicion = pepe.sprite.getPosition();
        float limiteDerechoCamara = pepePosicion.x > 300.f ? pepePosicion.x : 300.f;
        float limiteIzquierdoCamara = 3 * ventana.getSize().x - camara.getSize().x / 2; // Ajuste de nivel
más largo
```

```
        if (limiteDerechoCamara < limiteIzquierdoCamara) {
            camara.setCenter(limiteDerechoCamara, camara.getCenter().y);
        }
        else {
            camara.setCenter(limiteIzquierdoCamara, camara.getCenter().y);
        }

        if (pepePosicion.y > 350) {
            pepe.sprite.setPosition(50.f, 316.f); // Ajustar la posición de reinicio de Pepe
            camara.setCenter(300, camara.getCenter().y);
        }

        // Cambiar de nivel cuando Pepe llega al borde derecho de la ventana
        if (pepePosicion.x + pepe.sprite.getGlobalBounds().width >= 3 * ventana.getSize().x) { // Ajuste de
nivel más largo
            nivelActual++;
            iniciarNivel(nivelActual);
            return; // Salir del bucle y reiniciar el nivel
        }

        actualizarPosicionTextoPuntaje();

        ventana.clear();
        ventana.setView(camara);

        nivelJuego.dibujarFondo(ventana);
        nivelJuego.dibujarBloques(ventana);
        nivelJuego.dibujarPlataformasMoviles(ventana); // Dibujar plataformas móviles

        nivelJuego.moverPlataformasMoviles(deltaTime); // Mover plataformas móviles

        // Mover y dibujar el jefe final
        nivelJuego.moverJefeFinal(deltaTime, pepe.sprite.getPosition());
        nivelJuego.dibujarJefeFinal(ventana);

        // Verificar colisiones con el jefe
        pepe.verificarColisionesConJefe(*nivelJuego.obtenerJefeFinal());

        pepe.ejecutarMovimiento(ventana, deltaTime);
        pepe.dibujar(ventana);

        for (auto& enemigo : enemigos) {
            enemigo->mover(deltaTime);
            enemigo->dibujar(ventana);
        }

        for (auto& enemigo : enemigos) {
            if (pepe.sprite.getGlobalBounds().intersects(enemigo->sprite.getGlobalBounds())) {
```

```
                if (pepe.sprite.getPosition().y + pepe.sprite.getGlobalBounds().height >=
enemigo->sprite.getPosition().y &&
                    pepe.sprite.getPosition().y + pepe.sprite.getGlobalBounds().height <=
enemigo->sprite.getPosition().y + enemigo->sprite.getGlobalBounds().height &&
                    pepe.sprite.getPosition().x + pepe.sprite.getGlobalBounds().width >=
enemigo->sprite.getPosition().x &&
                    pepe.sprite.getPosition().x <= enemigo->sprite.getPosition().x +
enemigo->sprite.getGlobalBounds().width) {
                    enemigo->marcarEliminado();
                    puntaje += 1000;
                    actualizarPuntaje();
                    pepe.mecanicas->saltar();
                    pepe.setEnElSuelo(false); // Asegurarse de que Pepe pueda seguir saltando después de
eliminar un enemigo
                }
                else {
                    pepe.sprite.setPosition(50.f, 316.f); // Ajustar la posición de reinicio de Pepe
                    camara.setCenter(300, camara.getCenter().y);
                }
            }
        }

        auto it = enemigos.begin();
        while (it != enemigos.end()) {
            if ((*it)->estaEliminado()) {
                it = enemigos.erase(it);
            }
            else {
                ++it;
            }
        }

        verificarColisiones(nivelJuego.obtenerBloques(), nivelJuego.obtenerPlataformasMoviles(), pepe,
deltaTime);

        // Dibujar el puntaje
        ventana.draw(textoPuntaje);

        ventana.display();
    }
}

void verificarColisiones(std::vector<std::unique_ptr<Bloque>>& bloques,
std::vector<std::unique_ptr<PlataformaMovil>>& plataformasMoviles, Personaje& pepe, float deltaTime) {
    bool enElSuelo = false;
    bool debajoDeBloque = false;
    bool sobrePlataforma = false;
    sf::Vector2f velocidadPlataforma(0.f, 0.f);
```

```
        for (auto& bloque : bloques) {
            if (pepe.sprite.getGlobalBounds().intersects(bloque->obtenerLimitesGlobales())) {
                sf::FloatRect limitesPepe = pepe.sprite.getGlobalBounds();
                sf::FloatRect limitesBloque = bloque->obtenerLimitesGlobales();

                bool colisionSuperior = limitesPepe.top + limitesPepe.height <= limitesBloque.top + 10.f;
                bool colisionInferior = limitesPepe.top >= limitesBloque.top + limitesBloque.height - 10.f;
                bool colisionIzquierda = limitesPepe.left + limitesPepe.width <= limitesBloque.left + 10.f;
                bool colisionDerecha = limitesPepe.left >= limitesBloque.left + limitesBloque.width - 10.f;

                if (colisionSuperior) {
                    pepe.sprite.setPosition(pepe.sprite.getPosition().x, bloque->sprite.getPosition().y -
limitesPepe.height);
                    pepe.mecanicas->resetearVelocidadY();
                    enElSuelo = true;
                    pepe.setEnElSuelo(true);
                }
                else if (colisionInferior) {
                    pepe.mecanicas->resetearVelocidadY();
                    pepe.sprite.setPosition(pepe.sprite.getPosition().x, bloque->sprite.getPosition().y +
limitesBloque.height);
                    debajoDeBloque = true;
                }
                else if (colisionIzquierda) {
                    pepe.sprite.setPosition(limitesBloque.left - limitesPepe.width, pepe.sprite.getPosition().y);
                }
                else if (colisionDerecha) {
                    pepe.sprite.setPosition(limitesBloque.left + limitesBloque.width, pepe.sprite.getPosition().y);
                }
            }
        }

        for (auto& plataforma : plataformasMoviles) {
            if (pepe.sprite.getGlobalBounds().intersects(plataforma->obtenerLimitesGlobales())) {
                sf::FloatRect limitesPepe = pepe.sprite.getGlobalBounds();
                sf::FloatRect limitesPlataforma = plataforma->obtenerLimitesGlobales();

                bool colisionSuperior = limitesPepe.top + limitesPepe.height <= limitesPlataforma.top + 10.f;
                bool colisionInferior = limitesPepe.top >= limitesPlataforma.top + limitesPlataforma.height - 10.f;
                bool colisionIzquierda = limitesPepe.left + limitesPepe.width <= limitesPlataforma.left + 10.f;
                bool colisionDerecha = limitesPepe.left >= limitesPlataforma.left + limitesPepe.width - 10.f;

                if (colisionSuperior) {
                    pepe.sprite.setPosition(pepe.sprite.getPosition().x, plataforma->sprite.getPosition().y -
limitesPepe.height);
                    pepe.mecanicas->resetearVelocidadY();
                    enElSuelo = true;
                    pepe.setEnElSuelo(true);
                    sobrePlataforma = true;
```

```
                velocidadPlataforma = plataforma->obtenerVelocidad();
            }
            else if (colisionInferior) {
                pepe.mecanicas->resetearVelocidadY();
                pepe.sprite.setPosition(pepe.sprite.getPosition().x, plataforma->sprite.getPosition().y +
limitesPlataforma.height);
                debajoDeBloque = true;
            }
            else if (colisionIzquierda) {
                pepe.sprite.setPosition(limitesPlataforma.left - limitesPepe.width, pepe.sprite.getPosition().y);
            }
            else if (colisionDerecha) {
                pepe.sprite.setPosition(limitesPlataforma.left + limitesPlataforma.width,
pepe.sprite.getPosition().y);
            }
        }
    }

    // Permitir movimiento lateral del personaje sobre la plataforma móvil
    if (sobrePlataforma && !sf::Keyboard::isKeyPressed(sf::Keyboard::A) &&
!sf::Keyboard::isKeyPressed(sf::Keyboard::D) &&
        !sf::Keyboard::isKeyPressed(sf::Keyboard::Left) &&
!sf::Keyboard::isKeyPressed(sf::Keyboard::Right)) {
        pepe.sprite.move(velocidadPlataforma * deltaTime);
    }

    if (!enElSuelo) {
        pepe.setEnElSuelo(false);
    }

    pepe.mecanicas->debajoDeBloque = debajoDeBloque;
}

void cargarTexturas() {
    administradorDeTexturas.cargarTextura("C:/PepePepinillo/pepe.png", "C:/PepePepinillo/pepe.png");
    administradorDeTexturas.cargarTextura("C:/PepePepinillo/malo.png", "C:/PepePepinillo/malo.png");
    administradorDeTexturas.cargarTextura("C:/PepePepinillo/maloVolador.png",
"C:/PepePepinillo/maloVolador.png");
    administradorDeTexturas.cargarTextura("C:/PepePepinillo/bloqueChido.png",
"C:/PepePepinillo/bloqueChido.png");
    administradorDeTexturas.cargarTextura("C:/PepePepinillo/fondo.png",
"C:/PepePepinillo/fondo.png");
    administradorDeTexturas.cargarTextura("C:/PepePepinillo/plataforma.png",
"C:/PepePepinillo/plataforma.png"); // Cargar textura de la plataforma móvil
}

void actualizarPuntaje() {
    textoPuntaje.setString("Score: " + std::to_string(puntaje));
}
```

```
    void actualizarPosicionTextoPuntaje() {
        textoPuntaje.setPosition(camara.getCenter().x - camara.getSize().x / 2 + 10, camara.getCenter().y -
camara.getSize().y / 2 + 10);
    }
};

Juego::Juego() : ventana(VideoMode(1200, 720), "Pepe el pepinillo", Style::Default),
camara(Vector2f(300.f, 185.f), Vector2f(600.f, 367.f)),
puntaje(0), enemigosDerrotados(0), vecesMuerto(0) {
    ventana.setView(camara);
    cargarTexturas();

    ventana.setFramerateLimit(60);
    ventana.setVerticalSyncEnabled(true);

    // Cargar la fuente y configurar el texto de puntaje
    fuentePuntaje.loadFromFile("C:/PepePepinillo/fuente/Sign Rover Layered.ttf");
    textoPuntaje.setFont(fuentePuntaje);
    textoPuntaje.setCharacterSize(30);
    textoPuntaje.setFillColor(sf::Color::Black);
    actualizarPuntaje();
}
```

Juego es la clase más importante del juego esta es la que coordina todos los elementos del juego. Esta inicia la ventana del juego, la cámara, y estadísticas del juego como el puntaje, enemigos derrotados y las veces que mueres. La clase tiene el ciclo principal del juego, controlando la lógica del menú, la transición entre los dos niveles y la actualización del estado del juego. Ejecutar gestiona el ciclo principal del juego, incluyendo la inicialización del menú y la respuesta a la entrada del usuario. La clase Juego también se encarga de cargar todas las texturas necesarias, verificar colisiones entre el jugador, los enemigos y el entorno, y actualizar el puntaje del jugador. La función iniciarNivel configura y lanza un nivel específico, y verificarColisiones es la que maneja las interacciones físicas entre el jugador y otros elementos del juego.

Conclusiones

Conclusiones del equipo

Durante el desarrollo del proyecto, enfrentamos varios retos y dificultades, incluyendo la integración de FreeType para manejar las fuentes de texto y la correcta implementación del menú en diferentes máquinas. Sin embargo, logramos alcanzar nuestros objetivos principales, creando un juego funcional con varios niveles y un jefe final pasable, pero quitando a un lado eso, este proyecto nos ayudó sobre todo a entender los conceptos básicos de POO, aparte de que es nuestra (casi)primera experiencia como tal entregando un proyecto por avances, haciendo meetings, explicando cada avance, aparte de que al ser cinco integrantes, .

Conclusiones individuales

Diego:

El implementar el menu me enseñó las diferencias que puede haber entre entornos de programación y tambien me enseño la forma en la que funcionan los eventos y cómo se pueden implementar inputs, otra cosa que aprendí es que no todos los tipos de fuentes o imágenes son compatibles con las librerías gráficas de SFML ya que por ejemplo con las fuentes el problema era el tipo de fuente ya que si no pertenecía al grupo de fuentes TRUE TYPE no funcionaba y marcaba error lo cual hacía que fuera muy complicado distinguir si era error de compatibilidad o si era un error al descargar e implementar el SFML, podría decir que fue divertido el implementar las cosas y hacer que el menú ya pueda abrir el juego.

Alex:

Implementar el jefe final fue algo que me tomó demasiado tiempo, pero aprendí mucho sobre el manejo de colisiones y patrones de ataque. Aunque logramos que sirviera más o menos, me hubiera gustado hacer que este fuera más complicado para la gente que jugó el juego. En cuanto a los enemigos que fue otra de las cosas que me encargue fue más tranquilo y aunque tiene su lógica, disfrute bastante hacer algunos sprites(Aunque no nos quedamos con ninguno :(.)

Esteban:

Aprendí sobre el uso de la librería SFML para manejar gráficos, ventanas y eventos. Implementar funciones como el menú y las mecánicas del personaje y enemigos fue un reto que me enseñó a gestionar y solucionar problemas de compatibilidad. Aunque fue complicado, especialmente la integración de enemigos con mayor inteligencia como el jefe, o el crear un movimiento que se sienta bien al jugar y no se rompa con cambios, este proyecto me permitió aplicar conocimientos teóricos, trabajar en equipo, gestionar el tiempo y resolver problemas técnicos, preparándome mejor para futuros proyectos y consiguiendo alguna experiencia dentro de lo que sería el uso de herramientas para un trabajo de equipo real.

Ivan:

Aunque podría haber implementado más funcionalidad al juego, en particular más niveles, me quedé satisfecho con el rendimiento de nuestro juego. Enfrentamos muchos retos desde la implementación de la librería SFML hasta el último minuto. A comparación con mi proyecto anterior de métodos de programación, interactúe más con las partes principales del juego como el bucle principal y los niveles. Al finalizar esta experiencia veo bastantes áreas de mejora, por ejemplo me hubiera gustado haber planeado mejor la implementación de los niveles desde un principio, debido a que llegué a un punto en el que para facilitar este proceso, tenía que cambiar la mayor parte del código. Dado al tiempo limitado termine creando los niveles de una forma muy ineficaz. Al final de cuentas disfruté de la experiencia y estoy emocionado para los siguientes proyectos que vienen en el futuro.

Jose:

El implementar las mecánicas principales del juego al principio fue una tarea fácil ya que no ocupamos mucho movimiento del personaje principal, pero después el irle agregando las mecánicas a los enemigos y al jefe final ya fue algo que llegó a ser complicado por que queríamos que cada uno tuviera su mecánica de movimiento y como estos reaccionan entre sí, pero con el paso de las semanas fui aprendiendo con guías y leyendo libros de SFML, otra cosa que me llegó a frustrar fue la implementación de la música para el menú, los niveles, el personaje principal y los enemigos ya que llegaba a romper el código o en su caso a romper mi

computadora por completo y es una espina clavada que tendré y espero que me sirva como enseñanza.

Logros y aspectos por mejorar

Logros:

Como equipo tuvimos varios logros, ya que logramos hacer que un juego multiplataforma con varios elementos interesantes, pero lo más importante es que el juego género interés, lo cual era algo que buscábamos, aparte de que logramos extender nuestro conocimiento en un proyecto real.

Aspectos por mejorar:

Hay bastantes cosas que pudimos haber mejorado, pero lo más notorio que consideramos fue la falta de música y monedas, lo cual era algo que teníamos planeado, aparte de más niveles, solamente hicimos dos y creemos que pudimos haber hecho uno o dos más con un poco más de tiempo, también algo que nos faltó fue hacer que la gente tuviese que matar al jefe para pasar al siguiente nivel, ya que lo podía saltar sin mucha dificultad, otra cosa que nos faltó mucho fue la organización, ya que al ser novatos en este ámbito nos costó un poco lograr, los objetivos que nos proponemos durante los lapsos de las revisiones.