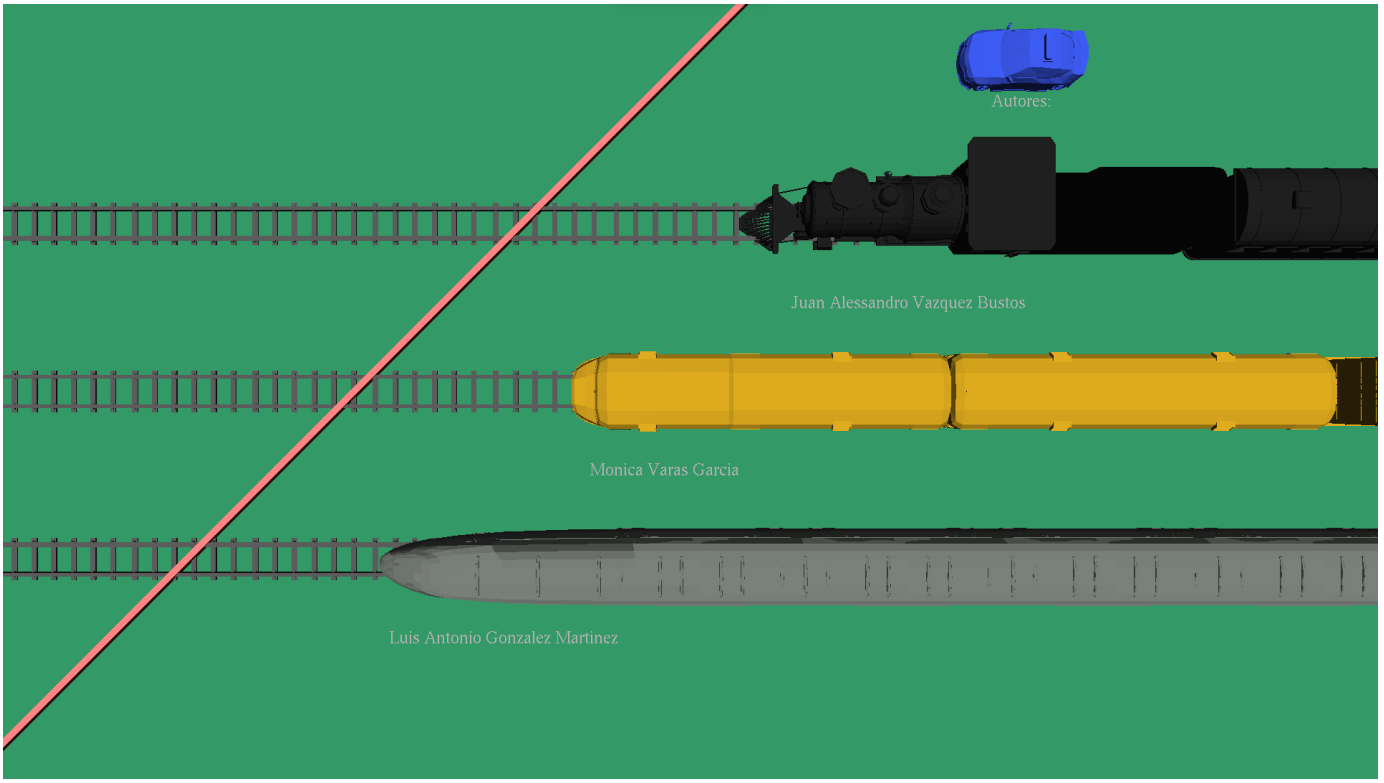
Programación Avanzada

Curso 2022 - 2023



PRÁCTICA FINAL

Mónica Varas Garcia
Juan Alessandro Vázquez Bustos
Luis Antonio González Martínez

19/06/2023

ÍNDICE

1. INTRODUCCIÓN

Esta memoria contiene la explicación del código del juego creado, el juego titulado "Scape the Train", consiste en una aventura en la que el jugador debe controlar a un personaje que se encuentra en un coche y tiene que esquivar los trenes que se encuentra en su camino para llegar a la meta. Este está diseñado utilizando la biblioteca gráfica OpenGL y programado en C++. La práctica está desarrollada a partir de la parte guiada realizada en clase.

2. EXPLICACIÓN DEL CÓDIGO

Se ha añadido código para adaptar el estado inicial de la práctica hasta convertirla en el videojuego final que conforma esta práctica final. Además de añadir las siguientes clases, se ha modificado parte de las ya incluyentes para aportar las características del proyecto.

2.1. Clase Game

La implementación de los niveles y las diferentes escenas se realiza desde el el Game::Init(). Se crean objetos de las clases derivadas de Scene para diferenciar los diferentes tipos de escenas. Se añaden las escenas al vector de escenas:

```
SceneMenu *sceneMenu = new SceneMenu();
sceneMenu->Init();
this->scenes.push_back(sceneMenu);

SceneLevel *sceneLevel;
for (int i = 1; i <= 3; i++)
{
    sceneLevel = new SceneLevel(i);
    sceneLevel->Init();
    this->scenes.push_back(sceneLevel);
}
```

Se activa la primera escena del array scenes al iniciar el juego:

```
    this->activeScene = this->scenes[0];
```

En su método Render() se renderiza la escena activa en caso de no ser un puntero vacío. En el método Update() se actualiza la escena activa y, además, se llama al método de la escena activa CheckStatus() para conocer el estado del juego. Este método es virtual y tan solo retorna 0 en las escenas que no sean de tipo SceneLevel (los propios niveles del juego). Si se devuelve algo diferente a 0 (un 1 o un 2) se activan las escenas de ganar o de perder respectivamente.

```
if (activeScene->CheckStatus() == 1)
{
    // Ir a pantalla de ganar.
    this->activeScene->Reset();
    this->activeScene = this->scenes[scenes.size() - 2];
}
else if (activeScene->CheckStatus() == 2)
{
```

```
        // Ir a pantalla de perder.
        this->activeScene->Reset();
        this->activeScene = this->scenes[scenes.size() - 1];
    }
```

2.2. Clase padre - Scene

La clase Scene, padre de todas las clases hijas de escena (SceneMenu, SceneLevel, SceneOver y SceneCreditos), contiene gran parte de propiedades y métodos para las clases derivadas: la cámara, límites de la escena, el vector de los objetos de la escena, los cargadores de modelos, el modelo de la vía de tren, un objeto de la clase Player (el jugador), un objeto de la clase Enemy (el enemigo), un objeto de la clase Obstacle y un vector de obstáculos (los trenes de la escena).

Los métodos propios de la clase Scene son:

```
// Metodos propios de la clase
    inline void AddGameObject(Solid *gameObject) {
this->gameObjects.push_back(gameObject); }
    inline void AddObstaculo(Obstacle *obstaculo) {
this->obstaculos.push_back(obstaculo); AddGameObject(obstaculo); }
    void LoadModelNvl1();
    void LoadModelNvl2();
    void LoadModelNvl3();
```

Se define el método virtual CheckStatus() para comprobar el estado de la partida usando polimorfismo:

```
int Scene::CheckStatus()
{
    return 0;
}
```

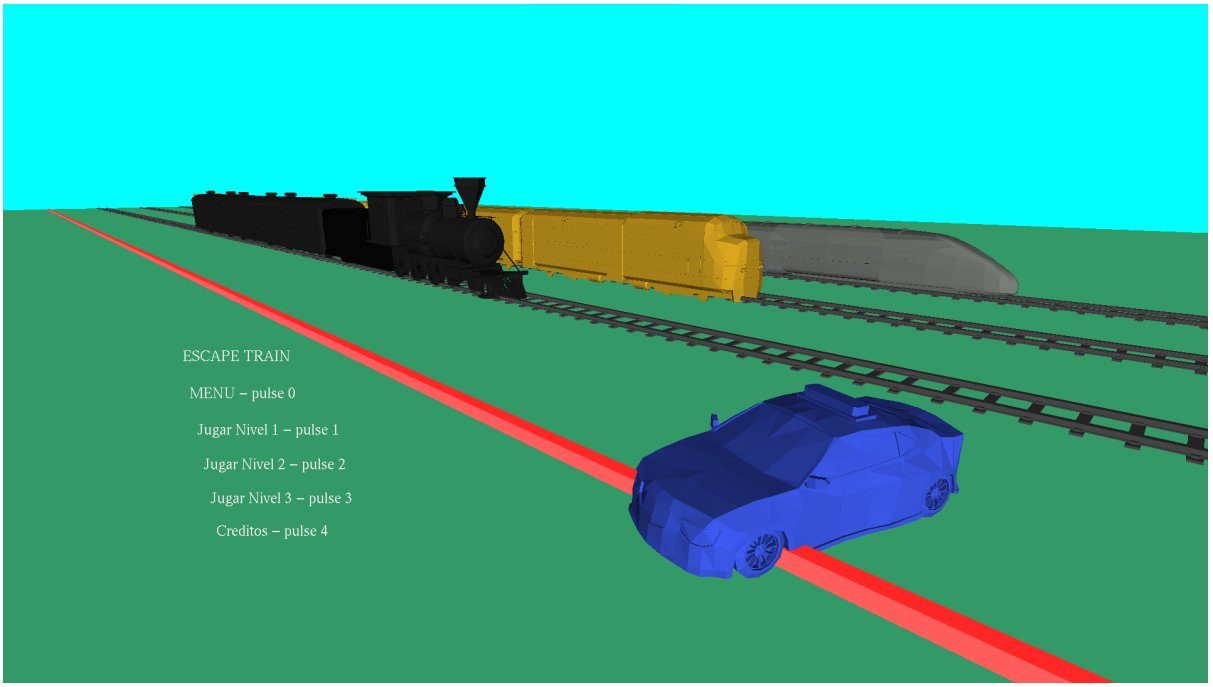
En los métodos LoadModel se cargan los distintos modelos de tren para cada nivel. Los modelos del personaje (el coche de policía) y el de las vías del tren se cargan en Init() teniendo en cuenta que son comunes a todas las escenas (y a todos los niveles del juego).

```
void Scene::LoadModelNvl1()
{
    // Carga del modelo 'TrenLvl1' (tren del nivel 1)
    loaderTren = new ModelLoader();
    loaderTren->LoadModel("..\\..\\3dModels\\TrenLvl1.obj");
    std::cout << "- Carga de modelo - Tren de nivel 1" << std::endl;
}
```

En el Render() y el Update() se la clase Scene se llama a su vez al Render() y Update() de cada objeto (gameObject) de la escena.

2.2.1. Clases hijas - SceneMenu, SceneGameOver y SceneCreditos

Las clases SceneMenu, SceneGameOver y SceneCreditos son las escenas en las que no se desarrolla la partida. Las tres clases contienen un vector de textos para almacenar los títulos, además de propiedades y métodos propios a cada clase individual.




```
vector<Text *> titulos;
```

Cada clase extiende el método Init() de Scene para añadir textos y crear una composición con los modelos. Se puede ver con la clase SceneMenu:

```
Scene::Init();
```

Se llama al método Init() de la clase base para extender el método original. Se añaden los textos mediante un objeto Text y el vector de textos:

```
Text *texto = new Text();
texto->SetText("ESCAPE THE TRAIN");
titulos.push_back(texto);



for (int i = 0; i < titulos.size(); i++) // Añadimos los textos
```

```
    {
        titulos[i]->SetPosition(Vector3D(0, desplazamiento, -8));
        titulos[i]->SetColor(Color(1, 1, 1));
        AddGameObject(titulos[i]);
        desplazamiento--;
    }
```

En la clase SceneGameOver se hace una elección del texto mostrado según el valor de una variable de tipo boolean llamada status:

```
    // Eleccion de texto
    string estado;
    if (status)
    {
        estado = "Ha ganado";
    }
    else
    {
        estado = "Ha perdido";
    }
```

2.2.2. Clases hijas - SceneLevel

La clase SceneLevel es la encargada de gestionar las partidas y los diferentes niveles. Contiene propiedades propias a la clase, tales como el nivel de la partida, el nivel máximo, la figura de la meta a la que tiene que llegar el jugador, la propiedad activo para permitir el movimiento del personaje o no y un vector de los PowerUps del nivel.

En Init() se compone la escena del nivel. Se crea la cámara:

```
// CAMARA EN CENITAL (60 grados)
    Camera *camara = new Camera();
    camara->SetPosition(Vector3D(0, 2 + (3.75 * (nivel - 1)), 20 + (7.5 * (nivel -
1))));
    camara->SetOrientation(Vector3D(60.0, 0.0, 0.0));
    SetCamera(*camara);
```

Se selecciona el modelo de tren dependiendo del número de nivel:

```
```

```
if (nivel == 1)
   {
       LoadModelNvl1();
   }
   else if (nivel == 2)
   {
       LoadModelNvl2();
   }
   else
   {
       LoadModelNvl3();
   }
```

También se crean los trenes (objetos de la clase Obstáculo) en el método Init():

```
obstaculo = new Obstacle(
               loaderTren->GetModel(),                          // Modelo
               Vector3D(randomX, 1.7, -incremento * i),         // Posicion
               Vector3D(0.0, (i % 2 == 0) ? 180.0 : 0.0, 0.0), // Rotacion
               Vector3D(37.1, 3.58, 2.87),                      // Dimensiones
               Color(0.1, 0.1, 0.1),                            // Color
               (i % 2 == 0) ? 0.1 : -0.1);                      // Velocidad
```

Se crean el jugador y el enemigo:

```
// Personaje
   jugador = new Player(loaderPersonaje->GetModel(),
                        Vector3D(0.0, 0.5, 5.0));
   AddGameObject(jugador);

   // Enemigo
   randomX = (rand() % int(GetBoundary().GetX()));
   enemigo = new Enemy(loaderPersonaje->GetModel(),
                       jugador,
                       Vector3D(randomX, 0.5, GetBoundary().GetZ() / 4));
   AddGameObject(enemigo);
```

En cuanto a los PowerUps, se pueden introducir nuevos power ups mediante el método AddPowerUp():

```
```

```
inline void AddPowerUp(PowerUp *powerup) { this->powerups.push_back(powerup);
AddGameObject(powerup); };
```

```
   fastUp = new Accelere(loaderPower->GetModel(), jugador, Vector3D(randomX, 0.0,
randomZ));
   AddPowerUp(fastUp);
```

Otro método importante es el de comprobar los límites, CheckBoundary(), en el que se comparan las dimensiones de los objetos del juego con las de los límites (boundaries).

```
if (this->gameObjects[idx]->GetPosition().GetX() < -(this->boundary.GetX()) ||
           this->gameObjects[idx]->GetPosition().GetX() > this->boundary.GetX())
       {
           this->gameObjects[idx]->SetPosition(
               Vector3D(-1 * this->gameObjects[idx]->GetPosition().GetX(),
                        this->gameObjects[idx]->GetPosition().GetY(),
                        this->gameObjects[idx]->GetPosition().GetZ()));
       }
```

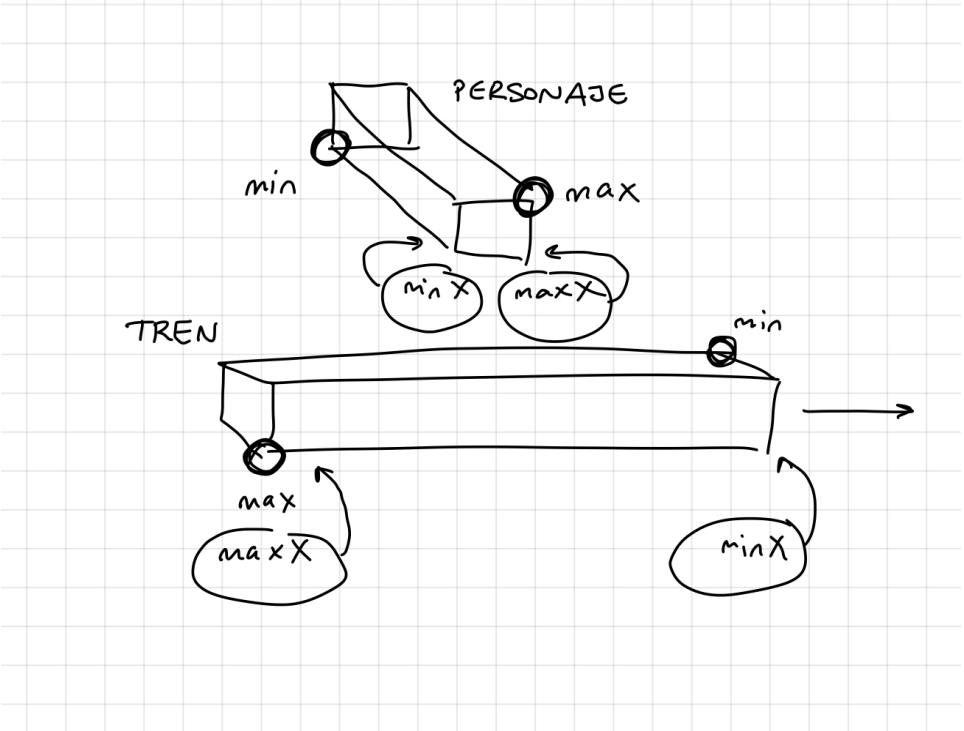
Se usa un método diferente (CheckColisiones()) para comprobar los diferentes tipos de colisiones que tienen lugar en la partida. Se usan las dimensiones y posiciones del personaje, el enemigo, los power ups y los obstáculos (los trenes) para determinar si ha habido colisión.
Este código de ejemplo, comprueba las colisiones con los obstáculos:

```
// // // Colisiones con los obstaculos
   for (int i = 0; i < this->obstaculos.size(); i++)
   {
       // Dimensiones del obstaculo.
       Vector3D dimensionesTren = this->obstaculos[i]->GetDimensions();
       float minXTren, maxXTren, minYTren, maxYTren, minZTren, maxZTren;
       minXTren = this->obstaculos[i]->GetPosition().GetX() -
dimensionesTren.GetX() / 2;
       maxXTren = this->obstaculos[i]->GetPosition().GetX() +
dimensionesTren.GetX() / 2;
       minYTren = this->obstaculos[i]->GetPosition().GetY();
       maxYTren = this->obstaculos[i]->GetPosition().GetY() +
dimensionesTren.GetY();
```

```
        minZTren = this->obstaculos[i]->GetPosition().GetZ() -
dimensionesTren.GetZ() / 2;
        maxZTren = this->obstaculos[i]->GetPosition().GetZ() +
dimensionesTren.GetZ() / 2;

        if (((minXPersonaje <= maxXTren && minXPersonaje >= minXTren) ||
             (maxXPersonaje <= maxXTren && maxXPersonaje >= minXTren)) &&
            ((minZPersonaje <= maxZTren && minZPersonaje >= minZTren) ||
             (maxZPersonaje <= maxZTren && maxZPersonaje >= minZTren)))
        {
            jugador->SetStatus(false);
            this->activo = false;
            std::cout << "Ha colisionado." << std::endl;
        }
    }
```

Se puede ver a continuación el esquema utilizado para desarrollar el código de las colisiones.



Se llama a los métodos CheckBoundary() y CheckColisiones() desde Update(), por lo que se comprueba constantemente.

Otro método propio es el CheckStatus(), del que hemos hablado previamente. Su cometido es devolver diferentes enteros para indicar el estado de la partida. Devuelve 0 si no ha ocurrido nada, 1 si se ha ganado la partida, y 2 si

se ha perdido. De esa forma se puede pasar en la clase Game a las determinadas escenas de clase SceneGameOver.

```
if (jugador->GetStatus())
{
    if (estadoPartida)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
else
{
    return 2;
}
```

Desde el método ProcessKeyPressed() se llama al método del mismo nombre del jugador si la partida se encuentra activa. De esa forma se puede mover el jugador con el teclado.

```
if (activo)
{
    jugador->ProcessKeyPressed(key, px, py);
}
```

2.3. Clase padre - Vehicle

"Vehicle" es la clase padre de las clases "Player", "Enemy", "Obstacle" y esta hereda de la clase "Solid".
Los métodos propios de la clase Vehicle son:

```
// Metodos propios de la clase
void ActivarRalenti();  // Ralentizar la velocidad al modelo
void ActivarAccelere(); // Acelerar la velocidad al modelo
void ActivarNormal();   // Normalizar la velocidad al modelo
```

El método ActivarRalenti() reduce la velocidad a la mitad del vehículo.
El método ActivarAccelere() aumenta la velocidad en una razón de dos.
El método ActivarNormal() la velocidad vuelve al estado inicial.

```
    // // // Colisiones con los powerups
    for (int i = 0; i < this->powerups.size(); i++)
    {
        if (minXPersonaje <= this->powerups[i]->GetPosition().GetX() &&
            maxXPersonaje >= this->powerups[i]->GetPosition().GetX() &&
            minZPersonaje <= this->powerups[i]->GetPosition().GetZ() &&
            maxZPersonaje >= this->powerups[i]->GetPosition().GetZ())
        {
            powerups[i]->ApplyEffect();
        }
    }
```

Cuando el jugador colisiona con los power ups, se llama al método ApplyEffect(). Dependiendo del power up con el que colisione, se aplicará una acción u otra.

2.3.1. Clases hijas - Player

La clase "Player" es una clase hija de "Vehicle" y es el encargado de controlar las acciones del usuario mediante el método processKeyPressed().

2.3.2. Clases hijas - Enemy

La clase "Enemy" es una clase hija de "Verhicle" y contiene el siguiente método propio :

```
    // Metodos propios de la clase
    void TrackPlayer(); // Seguir al jugador objetivo.
```

El método TrackPlayer() obtiene las coordenadas del personaje y calcula una ruta hacia ellas priorizando el eje y ante el x, así conseguimos que persiga al jugador. Este no tiene colisiones con los trenes pero sí con el player.

2.3.3. Clases hijas - Obstacle

La clase "Obstaculo" al igual que las dos anteriores es una clase hija de "Vehicle" y contiene el siguiente método:

```
// Metodos propios de la clase
    inline void Repositioning(const Vector3D &repositionToSet) {
this->SetPosition(repositionToSet); }
```

Debido a que una sobrecarga del método Setposition (heredada anteriormente de solid) daba problemas, se planteó un nuevo método que solucionase estos errores. Por lo que el método Repositioning(), reubica la posición del obstáculo de un lado del nivel al otro y viceversa.

2.4. Clase padre - PowerUp

PowerUp es hija de la clase "Solid" y proporciona los elementos necesarios para la creación de potenciadores. Sus clases hijas son "Ralenti" y "Accelere".

2.4.1. Clases hijas - Ralentí

Esta clase es la hija de la clase "PowerUp" y se encarga de llamar al método ActivarRalenti() de la clase Vehicle.

```
void Ralenti::ApplyEffect()
{
    if (GetStatus())
    {
        SetStatus(false);
        for (int idx = 0; idx < obstacles.size(); idx++)
        {
            obstacles[idx]->ActivarRalenti();
        }
    }
}
```

2.4.2. Clases hijas - Accelere

Esta clase al igual que la anterior es la hija de "PowerUp" y se encarga de llamar al método ActivarAccelere() de la clase vehicle.

```
void Accelere::ApplyEffect()
{
    if (GetStatus())
    {
        SetStatus(false);
        player->ActivarAccelere();
    }
}
```

3. CONCLUSIONES

En conclusión, la realización de esta práctica ha permitido adquirir y profundizar en el conocimiento de desarrollo de videojuegos en un entorno 3D. Se ha implementado un juego cuyo objetivo es evitar obstáculos incluyendo niveles de dificultad, movimiento, colisiones... Para ello se ha aprendido a utilizar correctamente la programación orientada a objetos.

En general, se puede decir que esta práctica ha sido muy enriquecedora y ha permitido poner en práctica y asentar conocimientos adquiridos en la asignatura.

4. BIBLIOGRAFÍA

Se han usado el material aportado en el aula virtual de la asignatura "Programación avanzada"
https://www.aulavirtual.urjc.es/moodle/course/view.php?id=193370

Con motivo de consulta para algunas funciones, también se ha usado la página oficial de OpenGL.
https://www.opengl.org

Para la realización del video, se ha usado OBS Studio para grabar la pantalla.
https://obsproject.com/es

Para la realización del diagrama de clases UML se ha utilizado la siguiente página.
https://www.diagrameditor.com