**INFORME DE PROYECTO FINAL**

**Autores:** *Cristian Flores Grisales, Natalia Polo Peña*

*Informática 2*

*Departamento de Ingeniería Electrónica y de Telecomunicaciones*

*Universidad de Antioquia*

# Resumen

En este informe se detalla la implementación del videojuego Fiesta Multiversal en el lenguaje de programación C++ utilizando el enfoque de Programación Orientada a Objetos (POO). Para ello, fue esencial la inspiración en el capítulo 11 de la temporada 1 de Rick and Morty.

**Palabras clave:** atributos, clases, colisión, constructor, métodos.

# Introducción

El siguiente juego está inspirado en el capítulo 11 – temporada 1 de la serie de televisión Rick and Morty.

La serie se basa en las aventuras de un abuelo y su nieto, quienes viajan atravesando portales multidimensionales que los llevan a otros universos atravesando diversos desafíos extremos.

En el capítulo seleccionado Rick and Morty se transportan a otro universo, en el cual, deben recolectar diamantes intergalácticos que les permitirán regresar a casa.

A partir de esto, surge la inspiración de este videojuego, en el cual se crean dos niveles. El primero, Rick debe recolectar cierta cantidad de diamantes esquivando los enemigos alienígenas, para defenderse, el personaje puede realizar tiros para eliminarlos, sin embargo, si se deja alcanzar por un alienígena, automáticamente, termina el juego.

# Contextualización del problema

El desafío que se aborda consiste en la implementación de un videojuego inspirado en la serie de televisión Rick and Morty en el lenguaje de programación C++, con un enfoque principal en la programación orientada a objetos (POO) para resolverlo. Para lograrlo, es esencial que se divida el análisis del problema en varios pasos, el primero de ellos es estudiar y analizar las reglas y la lógica del juego, lo que permitirá realizar una implementación exitosa en C++.

Análisis de la solución

Para la implementación del videojuego fue necesario realizar un diagrama de clases que permitiera un bosquejo para la estructura del juego. Con ello, se tuvo en cuenta, la cantidad de personajes y elementos a utilizar, así como también la división del juego por niveles.

Nivel 1: El objetivo es que Rick recolecte cierta cantidad de diamantes intergalácticos evitando chocarse con los alienígenas.

Nivel 2: Morty debe aniquilar cierta cantidad de alienígenas.

# Procedimiento experimental y resultados

Para el diseño de la solución implementamos la siguiente estructura de datos

**Clase NivelesJump:**

**Constructor:** Se establece la interfaz de usuario (**ui**) y se crean instancias de las clases.

Se configura la interfaz gráfica, obteniendo las dimensiones de la pantalla principal y creando una escena gráfica para la vista **graphicsView**.

**Objeto**: Se añade el objeto **Rick** a la escena y se configura su posición inicial.

Se configuran dos temporizadores (**enemigoTimer** e **ItemTimer**) para llamar a las funciones **agregarEnemigo** y **agregarDiamante** periódicamente.

**Funciones para el manejo de eventos:**

**keyPressEvent**: Maneja eventos de teclas para mover el personaje Rick y verifica colisiones con enemigos e ítems.

**mousePressEvent**: Maneja eventos del mouse para crear un proyectil (Proyectil) cuando se hace clic izquierdo.

**enemigoAlcanzadoPorProyectil**: Elimina proyectiles y enemigos al colisionar.

**ColisionRickEnemigo**: Elimina a Rick y al enemigo al colisionar, muestra la pantalla de Game Over y cierra el nivel.

**colisionRickItem**: Maneja colisiones entre Rick y ítems (diamantes), actualiza el contador y verifica si se alcanzó la cantidad necesaria para ganar.

**QMessageBox** para mostrar cuadros de diálogo con mensajes de Game Over o Ganador.

**Delete ui:** Libera la memoria asignada al objeto de la interfaz de usuario (**ui**).

**Clase NivelesPersecución:**

**Constructor:** Se establece la interfaz de usuario (ui) y se crean instancias de las clases Morty y **QGraphicsScen.**

**Objeto:** Morty.

**Manejo de eventos:**

**mousePressEvent:** Maneja eventos del mouse para crear un proyectil (ProyectilParabolico) cuando se hace clic izquierdo.

**keyPressEvent:** Maneja eventos de teclas para mover el personaje Morty.

**Función del manejo del mouse:**

Calcula la posición y ángulo para el disparo del proyectil y crea un nuevo proyectil parabólico (ProyectilParabolico) con velocidad y posición inicial. La función actualizarTiempo se llama para iniciar el movimiento parabólico.

**Función para el manejo del teclado:**

Maneja eventos de teclas para cambiar la dirección de Morty hacia la derecha o hacia arriba.

**Destructor:**

Delete Morty, Delete, scene, Delete ui. Libera la memoria asignada a los objetos Morty, scene, y ui.

**Clase Enemigos:**

**Constructor:** se inicia el constructor de la clase Enemigos que hereda de **QObject**.

Se crea un temporizador para controlar la actualización de la posición del objeto.

Se establecen las dimensiones (ancho y alto) y se carga una imagen (pixmap) para representar al enemigo.

Se inicia el temporizador y se conecta su señal **timeout** a la función **Actualizacion**.

Se establece el tipo de usuario usando **setData** con el valor **UserType + 1**.

Se crea un temporizador (**timer**) que se conecta a la función **Actualizacion**.

**Funciones de representación**:

**boundingRect**: Define el rectángulo delimitador del objeto para la detección de colisiones.

**paint**: Dibuja la imagen del enemigo en el rectángulo delimitador.

**Funciones de movimiento**:

**setVelocidad**: Permite configurar la velocidad del enemigo.

**Actualizacion**: Actualiza la posición del enemigo en función de su velocidad, moviéndolo hacia abajo

**Clase Items**:

**Constructor**: Inicia el constructor de la clase Items, que hereda de **QGraphicsObject**.

Carga una imagen (en este caso, un diamante representado por un objeto QPixmap (pixmap).

Se establecen las dimensiones (ancho y alto) para la detección de colisiones y la representación visual del objeto.

**Funciones de representación:**

**boundingRect**: Define el rectángulo delimitador del objeto para la detección de colisiones.

**paint**: Dibuja la imagen del ítem en el rectángulo delimitador utilizando la clase QPainter.

**Clase Morty**:

**Constructor**: Inicia el constructor de la clase Morty, que hereda de **QGraphicsObject**.

Configura un temporizador (timer) para manejar la animación del personaje.

Carga un sprite sheet que contiene las imágenes de animación de Morty.

Se establece las dimensiones del personaje (ancho y alto).

**Función de la actualización de la animación:**

La función actualizacion se llama cada vez que el temporizador (timer) emite la señal timeout.

Incrementa la columna actual del sprite sheet, avanzando a la siguiente imagen en la animación.

Si la columna alcanza el límite del sprite sheet, se reinicia a cero.

La función **cambiarDirrecion** actualiza la posición del objeto Morty según la dirección proporcionada.

La función **paint** dibuja la imagen actual del sprite sheet en el rectángulo delimitador del objeto.

**Clase Rick:**

**Constructor:** Inicia el constructor de la clase Rick, que hereda de QGraphicsObject.

Se configura un temporizador (timer) para manejar la animación del personaje.

Se carga un sprite sheet que contiene las imágenes de animación de Rick.

Se establece las dimensiones del personaje (ancho y alto).

La función **actualizacion** se llama cada vez que el temporizador (**timer**) emite la señal **timeout**.

Incrementa la columna actual del sprite sheet, avanzando a la siguiente imagen en la animación.

Si la columna alcanza el límite del sprite sheet, se reinicia a cero.

La función **cambiarDirrecion** actualiza la posición del objeto Rick según la dirección proporcionada.

verificarColisionConEnemigo y verificarColisionConItem verifican las colisiones de Rick con enemigos y objetos (Items) respectivamente.

Emiten señales (ColisionRickEnemigo y colisionRickItem) cuando se detecta una colisión.

La función **paint** dibuja la imagen actual del sprite sheet en el rectángulo delimitador del objeto Rick.

**Clase Proyectil:**

**Constructor:** Inicia el constructor de la clase Proyectil, que hereda de QObject.

Habilita la selección del proyectil (**ItemIsSelectable**).

Configura un temporizador para llamar a la función mover periódicamente.

**boundingRect** define el rectángulo delimitador del objeto para la detección de colisiones.

La función **paint** dibuja el proyectil como un círculo con un color específico.

La función **mover** se llama cada vez que el temporizador emite la señal **timeout**.

Calcula las nuevas coordenadas del proyectil en función de su dirección y velocidad.

Verifica si el proyectil colisiona con algún elemento en la escena.

Si colisiona con un elemento de tipo **UserType + 1** (posiblemente un enemigo), emite la señal **proyectilAlcanzoEnemigo** pasando el proyectil y el elemento colisionado como parámetros.

**Clase ProyectilParabolico**

El constructor de la clase **ProyectilParabolico** establece la bandera **ItemIsSelectable** para permitir la selección del proyectil.

**movimientoParabolico** Esta función crea un temporizador (**QTimer**) y lo conecta a la función **moverParabolico**. Este temporizador controla el movimiento del proyectil con una frecuencia de 50 milisegundos.

**moverParabolico** Esta función es llamada por el temporizador y actualiza la posición del proyectil utilizando las ecuaciones del movimiento parabólico.

**Clase MainWindow:**

Constructor: inicializa la interfaz de usuario (**ui**).

El destructor Delete ui se encarga de liberar la memoria asignada dinámicamente para la interfaz de usuario.

**on\_pushButtonJugarNivelesJump\_clicked** este método es un "slot" conectado al evento de clic del botón llamado **pushButtonJugarNivelesJump**. Cuando se hace clic en el botón, se crea una instancia de la clase **NivelesJump**, se establece como modal (bloqueando la interacción con otras ventanas), y se ejecuta.

**on\_pushButtonJugarNivelesPersecucion\_clicked** similar al método anterior, este es un "slot" conectado al evento de clic del botón pushButtonJugarNivelesPersecucion. Cuando se hace clic en el botón, se crea una instancia de la clase NivelesPersecución, se establece como modal, y se ejecuta.

# Conclusiones

En definitiva, es importante realizar un buen diseño para el manejo de las clases cuando se requiere hacer uso de la programación orientada a objetos. Se enfrentaron muchos desafíos a la hora de implementar sistemas físicos para la mecánica del juego. En definitiva, la elaboración de un videojuego requiere de un buen análisis y diseño de estructura de datos para facilitar su implementación.