

UNIVERSIDAD DE ANTIOQUIA  
DEPARTAMENTO DE ELECTRÓNICA Y TELECOMUNICACIONES  
2598521 - INFORMATICA II



# Poyecto Final - Infierno en Okinawa

## Documentación del Juego usando POO

**Rigoberto Berrio Berrio**  
Estudiante de Ingeniería Electrónica  
Universidad de Antioquia  
rigoberto.berrio1@udea.edu.co

**Jeisson Stevens Martinez Arevalo**  
Estudiante de Ingeniería en Telecomunicaciones  
Universidad de Antioquia  
jeisson.martinez1@udea.edu.co

3 de diciembre de 2025

## Índice

<b>1. Introducción</b>	<b>3</b>
1.1. Contexto del proyecto, motivación y público objetivo . . . . .	3
<b>2. Planteamiento del Problema</b>	<b>3</b>
2.1. Necesidad que cubre el software y justificación del desarrollo . . . . .	3
<b>3. Definición General y Objetivos</b>	<b>3</b>
3.1. Alcance funcional, metas y requisitos principales . . . . .	3
<b>4. Especificación de Requerimientos</b>	<b>3</b>
4.1. Requisitos funcionales . . . . .	3
4.2. Requisitos no funcionales . . . . .	4
4.3. Restricciones, dependencias y usuarios destinatarios . . . . .	4
<b>5. Metodología y Planificación</b>	<b>4</b>
5.1. Metodología empleada . . . . .	4
5.2. Fases del desarrollo y cronograma . . . . .	4
<b>6. Diseño y Arquitectura del Sistema</b>	<b>5</b>
6.1. Explicación jerárquica de los componentes . . . . .	5
6.2. Diagrama de clases . . . . .	6
6.3. Dependencias externas y bibliotecas utilizadas . . . . .	6
<b>7. Desarrollo e Implementación</b>	<b>7</b>
7.1. Breve descripción de la lógica interna, principales algoritmos y decisiones técnicas . . . . .	7
<b>8. Procedimientos de Prueba</b>	<b>8</b>
8.1. Métodos de verificación, casos de prueba, resultados obtenidos y revisión de calidad . . . . .	8
8.2. Procedimientos de prueba . . . . .	8
8.3. Pruebas funcionales por nivel . . . . .	8
8.4. Casos de prueba representativos . . . . .	8
8.5. Resultados y observaciones . . . . .	8
<b>9. Guía de Instalación y Uso</b>	<b>9</b>
9.1. Instrucciones para instalar y ejecutar el software . . . . .	9
9.2. Requisitos técnicos y recomendaciones . . . . .	9
<b>10. Resultados y Discusión</b>	<b>10</b>
10.1. Funcionamiento logrado, limitaciones actuales, dificultades enfrentadas y mejoras planeadas . . . . .	10
<b>11. Conclusiones</b>	<b>10</b>
11.1. Reflexión sobre el proceso, el aprendizaje obtenido y recomendaciones . . . . .	10
<b>12. Referencias</b>	<b>11</b>
12.1. Bibliografía, enlaces y documentación consultada . . . . .	11
<b>13. Anexos/Apéndices</b>	<b>11</b>
13.1. Fragmentos de Código Importantes Comentados . . . . .	11
13.2. Capturas de Pantalla (Ejemplos Visuales) . . . . .	12

## 1. Introducción

### 1.1. Contexto del proyecto, motivación y público objetivo

El proyecto **Infierno en Okinawa** es un videojuego de acción y disparos desarrollado como requisito para el curso de Informática II, haciendo uso de los principios de la **Programación Orientada a Objetos (POO)** y la biblioteca **Qt** para la interfaz gráfica. El juego se inspira en el género de acción de la Segunda Guerra Mundial, ofreciendo una experiencia que combina la jugabilidad de plataformas 2D con niveles de vista superior (Top-Down). La motivación principal es aplicar conceptos avanzados de C++, como la gestión de memoria dinámica, la implementación de modelos físicos, el diseño de agentes de Inteligencia Artificial (IA) simple, y el manejo de concurrencia a través de timers y señales/slots.

El público objetivo del juego son **jugadores con interés en juegos clásicos de acción** y, principalmente, **estudiantes de ingeniería** que busquen entender cómo se aplican los fundamentos de POO y C++ en el desarrollo de software interactivo y de entretenimiento.

## 2. Planteamiento del Problema

### 2.1. Necesidad que cubre el software y justificación del desarrollo

El software cubre la necesidad de **validar y demostrar la capacidad de aplicar los principios de la Programación Orientada a Objetos y el manejo de recursos gráficos complejos** en un entorno real. El desarrollo de un videojuego en C++ y Qt justifica la aplicación práctica de varios temas académicos obligatorios:

**Gestión de la Memoria:** Uso intensivo de memoria dinámica (`new` y `delete`) para la creación de objetos de juego (jugadores, enemigos, proyectiles). **Arquitectura de Eventos:** Implementación del sistema de señales y slots de Qt para la comunicación asíncrona entre objetos (ej. muerte del jugador, disparo del enemigo). **Diseño de Software:** Creación de una jerarquía de clases funcional que separa la lógica del juego (físicas, IA) de la representación gráfica (`QGraphicsPixmapItem`).

El videojuego sirve como una plataforma de aprendizaje robusta que va más allá de un simple ejercicio de consola.

## 3. Definición General y Objetivos

### 3.1. Alcance funcional, metas y requisitos principales

El alcance funcional del juego es ofrecer una experiencia de dos niveles interconectados, cada uno con una mecánica distinta.

#### Metas del Proyecto:

Lograr un **sistema de juego funcional** que alterne entre mecánicas de Plataforma (Nivel 1) y Top-Down Nivel 2. Implementar al menos **tres modelos físicos** distintos (gravedad/salto, movimiento balístico y colisiones). Demostrar una **abstracción de dificultad** tangible mediante la alteración de parámetros.

## 4. Especificación de Requerimientos

### 4.1. Requisitos funcionales

El juego debe permitir al jugador moverse, saltar, agacharse y disparar en el Nivel 1. El juego debe permitir el movimiento libre en 2D y el disparo en el Nivel 2. El juego debe presentar enemigos con **Inteligencia Artificial simple** (ej. persecución y reacción a eventos). El Nivel 1 debe finalizar al derrotar al jefe final (**BunkerBoss**). El Nivel 2 debe finalizar al sobrevivir un tiempo límite establecido. El juego debe reproducir **música de fondo y sonidos de eventos** (disparos, daño, muerte).

## 4.2. Requisitos no funcionales

**Usabilidad:** La interfaz de `GameWindow` debe ser clara, mostrando la vida, el arma y el progreso del nivel. **Rendimiento:** El ciclo de juego principal (`onTick`) debe ejecutarse a una tasa de al menos 60 FPS (1000/60 ms). **Mantenibilidad:** El código debe ser modular y documentado.

## 4.3. Restricciones, dependencias y usuarios destinatarios

### Restricciones Obligatorias de Implementación:

El desarrollo debe utilizar el lenguaje C++ y la interfaz gráfica Qt. **POO y Memoria Dinámica:** Uso obligatorio de clases, herencia y gestión de punteros (`new`, `delete`, `QPointer`). **Herencia Propia (no de Qt):** Se debe implementar una clase base de dominio para cumplir con el requisito de abstracción. **Excepciones:** La implementación debe incluir el uso de excepciones para manejar fallos críticos.

**Dependencias:** Qt Core, Qt GUI, Qt Widgets y Qt Multimedia. **Usuarios Destinatarios:** Estudiantes y evaluadores del curso Informática II.

## 5. Metodología y Planificación

### 5.1. Metodología empleada

El desarrollo del videojuego se realizó siguiendo una metodología sencilla pero iterativa. En lugar de intentar construir todas las funcionalidades desde el comienzo, se avanzó por etapas pequeñas: primero se obtuvo una versión mínima jugable y, a partir de ella, se fueron añadiendo y refinando las características del juego.

En una primera etapa se implementó un prototipo básico con el personaje principal y su movimiento dentro de la escena. Una vez verificado que el control era estable, se incorporaron elementos adicionales como los disparos, las colisiones con el escenario y la gestión de la vida del jugador. Posteriormente se añadieron los enemigos, sus patrones de comportamiento y las condiciones de victoria y derrota en los niveles.

El trabajo también se repartió de manera práctica entre los integrantes del equipo. Una parte se enfocó principalmente en la lógica del juego (manejo de colisiones, actualización de objetos, condiciones de paso entre escenas), mientras que la otra se concentró en los aspectos visuales y de presentación (sprites, fondos, sonidos ambiente, organización de las escenas y pantallas de interfaz). Al finalizar cada iteración se realizaban pruebas de juego informales para detectar errores, ajustar parámetros de dificultad y decidir qué mejoras abordar a continuación.

Esta forma de trabajo, basada en iteraciones cortas de implementación y prueba, permitió ir corrigiendo problemas desde etapas tempranas del desarrollo y mantener siempre una versión funcional del proyecto.

### 5.2. Fases del desarrollo y cronograma

A efectos de documentación, el proceso de desarrollo puede organizarse en varias fases principales:

- **Fase de conceptualización:** definición de la temática del juego, su ambientación, las mecánicas básicas (movimiento, disparo, enemigos) y el número de niveles a implementar.
- **Fase de diseño técnico:** elaboración del diagrama de clases, selección de las clases de Qt a utilizar (`QGraphicsScene`, `QGraphicsItem`, `QTimer`, etc.) y diseño general de las escenas y pantallas.
- **Fase de prototipo inicial:** implementación de un prototipo con el jugador, su movimiento y una escena sencilla, con el objetivo de validar el control y la estructura básica del proyecto.
- **Fase de implementación del Nivel 1:** incorporación de los elementos propios del primer nivel, incluyendo enemigos, colisiones, gestión de vida y condición de finalización del nivel.

- **Fase de implementación del Nivel 2:** desarrollo del segundo nivel, reutilizando la estructura anterior y adaptando las mecánicas a su diseño específico (tipo de vista, distribución de enemigos, ritmo de juego).
- **Fase de pruebas y documentación:** realización de pruebas de juego, corrección de errores, ajuste de parámetros de dificultad y redacción del informe técnico correspondiente.

De manera resumida, el cronograma aproximado de trabajo puede expresarse como se muestra en la Tabla 1.

Tabla 1: Resumen de las fases de desarrollo del videojuego.

Periodo	Actividades principales
Semana 1–2	Conceptualización del juego, definición de mecánicas y diseño del diagrama de clases.
Semana 3–4	Implementación del prototipo inicial y del Nivel 1 (jugador, enemigos, colisiones básicas).
Semana 5–6	Implementación del Nivel 2, ajustes de comportamiento de enemigos y pulido de la jugabilidad.
Semana 7	Pruebas de juego, corrección de <i>bugs</i> y elaboración de la documentación.

## 6. Diseño y Arquitectura del Sistema

### 6.1. Explicación jerárquica de los componentes

La arquitectura del juego se implementó siguiendo un diseño basado en el patrón **Modelo-Vista-Controlador (MVC)** adaptado al entorno de **QGraphicsView** de Qt, lo que facilita la separación de responsabilidades:

- Capa de Control/Vista (GameWindow):** Actúa como el controlador principal y la ventana de visualización. Es responsable de:
- 1. Inicializar el juego, la escena (**QGraphicsScene**) y la vista (**QGraphicsView**).
  - 2. Gestionar el flujo del juego (cambio de nivel, estado *Game Over*).
  - 3. Contener las referencias a todos los objetos críticos (`player_`, `tdPlayer_`, listas de enemigos) y gestionar su ciclo de vida.
- Contener el **QTimer** principal (`timer_`) que impulsa el ciclo de juego (`onTick`).

**Capa Lógica y Visual (QGraphicsItems):** Compuesta por todas las clases que representan entidades del juego (`*Item`). Estas clases implementan la lógica de negocio (físicas, IA, vida) y son simultáneamente la "Vistaza que heredan de **QGraphicsPixmapItem** para renderizarse.

**Capa de Recursos:** Constituida por los archivos de imágenes, sonidos y el motor de señales y slots de Qt para la comunicación interna.

## 6.2. Diagrama de clases

El diseño de clases es la columna vertebral del proyecto, utilizando la herencia múltiple de C++ en conjunto con Qt para dotar a los objetos de funcionalidad de señalización (QObject) y representación gráfica (QGraphicsPixmapItem).

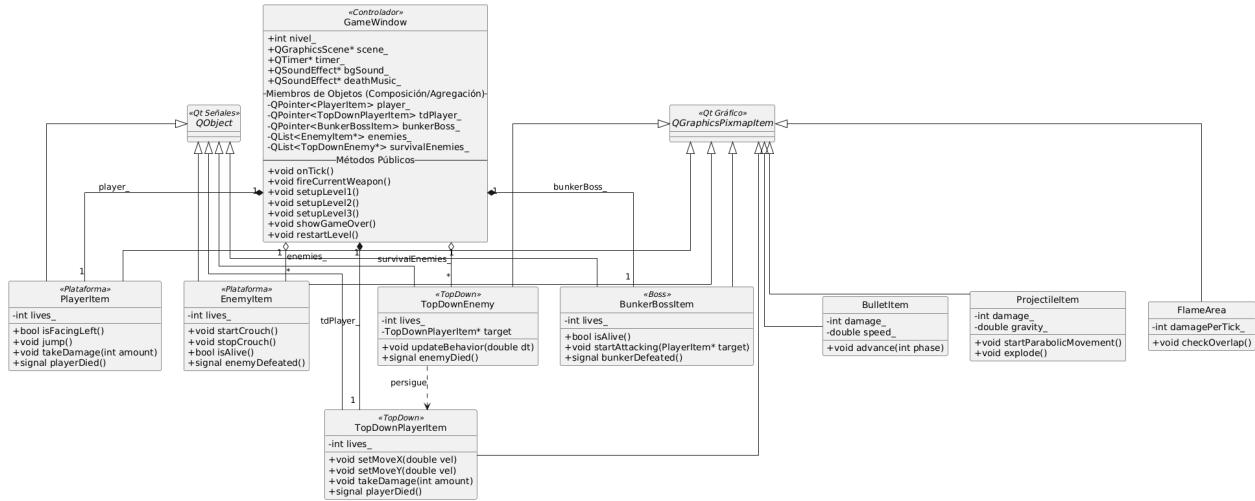


Figura 1: Diagrama de Clases del Sistema de Juego.

### Descripción de Clases Clave

**GameWindow:** (*Controlador*). Clase principal que maneja el estado global (`nivel_`) y el bucle de juego. Muestra una relación de **Composición** con los objetos críticos (`player_`, `bunkerBoss_`) y **Agregación** con las listas de enemigos. **PlayerItem / TopDownPlayerItem:** (*Plataforma/Top-Down Héroe*). Representan al personaje principal. Contienen atributos críticos como `lives_` e implementan la lógica de movimiento y daño, emitiendo `playerDied()` al ser derrotados. **EnemyItem / TopDownEnemy:** (*Enemigos*). Implementan la **IA simple**. El `TopDownEnemy` usa el miembro `target` para perseguir al jugador, estableciendo una relación de **Dependencia**. **BunkerBossItem:** (*Jefe*). Subclase de enemigo con lógica de ataque específica, emitiendo `bunkerDefeated()` al ser destruido. **BulletItem, ProjectileItem, FlameArea:** (*Projectiles*). Clases ligeras que gestionan la lógica balística (`ProjectileItem`), el movimiento rectilíneo (`BulletItem`) y el área de efecto (`FlameArea`), todas diseñadas para ser creadas y destruidas dinámicamente.

## 6.3. Dependencias externas y bibliotecas utilizadas

El proyecto se basa exclusivamente en las siguientes bibliotecas del marco de Qt:

**Qt Core:** Para la gestión de hilos, timers (QTimer), estructuras de datos (QList), y el sistema de señales y slots (QObject). **Qt Widgets:** Usado para la ventana principal (QMainWindow) y los elementos HUD como QLabels para mostrar la vida y mensajes. **Qt GraphicsView:** Esencial para el motor de renderizado 2D, incluyendo QGraphicsScene, QGraphicsView, y QGraphicsPixmapItem. **Qt Multimedia:** Utilizado para la reproducción de audio (QSndEffect) para la música de fondo y los sonidos de eventos.

## 7. Desarrollo e Implementación

### 7.1. Breve descripción de la lógica interna, principales algoritmos y decisiones técnicas

El motor del juego se basa en el ciclo de actualización de la función `onTick()` del `QTimer`, donde se aplican las reglas físicas y se gestionan las interacciones. La implementación se centró en cumplir con el requisito de incluir al menos tres modelos físicos y un agente inteligente.

#### Modelos Físicos Implementados

Se incluyeron tres modelos físicos distintos al movimiento rectilíneo:

- 1. **Gravedad y Salto (Movimiento Parabólico Vértical):** Implementado en `PlayerItem` para el Nivel 1. La posición vertical (`y`) del jugador se actualiza en cada `onTick()` utilizando la velocidad vertical (`vy`) y la aceleración constante de la gravedad (`g`). El salto establece una velocidad vertical inicial negativa (hacia arriba).
- 2. **Movimiento Balístico (Proyectiles):** Implementado en `ProjectileItem` (granadas). Utiliza una lógica similar a la gravedad pero aplicada a un proyectil disparado con ángulo, creando una trayectoria parabólica.
- 3. **Detección de Colisiones:** Algoritmo de chequeo de colisiones basado en `collidingItems()` de Qt, esencial para: (1) detener la caída del jugador al tocar el suelo o plataformas (*World Collision*); (2) aplicar daño al impactar un enemigo o proyectil (*Hitbox Collision*).

#### Implementación de Inteligencia Artificial (IA)

Se implementó un agente de IA simple a través de la clase `TopDownEnemy` y el `BunkerBossItem`:

- **Persecución Simple (Seek):** El método `updateBehavior(dt)` del `TopDownEnemy` calcula la diferencia de posición entre el enemigo y el objetivo (`tdPlayer_`). Utiliza el vector resultante para mover al enemigo directamente hacia el jugador, manteniendo una distancia de ataque.
- **IA Reactiva (Plataforma):** Los enemigos del Nivel 1 (`EnemyItem`) tienen la capacidad de reaccionar a un evento específico (ej. el jugador disparando) agachándose (`startCrouch()`) con un 50% de probabilidades (parametro ajustable) de evadir el impacto, mostrando una respuesta basada en el estado del juego.

#### Abstracción de la Dificultad

El requisito de la noción de dificultad se cumple mediante la implementación de **dos niveles**, donde el Nivel 2 (Supervivencia Top-Down). La dificultad se abstrae y se ajusta mediante la modificación de tres parámetros clave en el método `setupLevel3()`:

1. **Vida Inicial del Héroe:** Reducida (ej. de 6 a 4 puntos de vida).
2. **Tiempo de Supervivencia:** Reducido (ej. de 60 a 40 segundos).
3. **Cantidad de Generación de Enemigos:** Incrementada para saturar la pantalla.

Esta abstracción permite un ajuste rápido y escalable de la dificultad sin modificar la lógica interna de los enemigos.

## 8. Procedimientos de Prueba

### 8.1. Métodos de verificación, casos de prueba, resultados obtenidos y revisión de calidad

### 8.2. Procedimientos de prueba

Con el fin de verificar el correcto funcionamiento del videojuego, se realizaron pruebas funcionales, pruebas de juego (*playtesting*) y pruebas orientadas a la estabilidad. Más allá de comprobar que el código compilara sin errores, se buscó garantizar que cada nivel fuera jugable de principio a fin y que las mecánicas respondieran de forma coherente para el usuario.

### 8.3. Pruebas funcionales por nivel

Las pruebas funcionales se centraron en verificar que las principales características del juego se comportaran como se esperaba:

- **Control del jugador:** se probó el movimiento en todas las direcciones permitidas, el salto (en el nivel de desplazamiento lateral) y el disparo. Se verificó que no existieran bloqueos al chocar contra elementos del escenario.
- **Sistema de disparos y daño:** se comprobó que los proyectiles generados por el jugador colisionaran correctamente con los enemigos y redujeran su vida, y que los disparos enemigos afectaran la barra de vida del jugador.
- **Gestión de enemigos:** se probó la aparición de enemigos en cada nivel, su comportamiento básico (movimiento hacia el jugador, seguimiento o patrones definidos) y su eliminación cuando su vida llegaba a cero.
- **Transiciones entre niveles:** se verificó que, tras cumplir la condición de victoria de cada nivel (derrotar al jefe o sobrevivir el tiempo establecido), el juego avanzara correctamente al siguiente nivel o mostrara la pantalla final correspondiente.

### 8.4. Casos de prueba representativos

Además de las pruebas generales, se definieron algunos casos de prueba representativos:

- **Caso 1: Derrota del jugador.** Se dejó que el personaje recibiera daño hasta agotar su barra de vida. El juego debía mostrar la pantalla de *game over* y ofrecer la opción de reiniciar.
- **Caso 2: Victoria en el primer nivel.** Se derrotó al jefe del primer nivel para comprobar que la transición al siguiente nivel se realizara sin errores y que los recursos del nivel anterior se liberaran correctamente.
- **Caso 3: Supervivencia en nivel 2.** En el nivel con vista cenital, se cumplió el tiempo de supervivencia definido para verificar la aparición del mensaje de victoria y el cierre adecuado de la escena.

En cada caso se registraron posibles comportamientos inesperados, como colisiones que no se detectaban, enemigos que quedaban inmóviles o inconsistencias en la barra de vida. Los errores detectados durante estas pruebas sirvieron para ajustar posiciones de colisión, tiempos de aparición de enemigos y parámetros de daño.

### 8.5. Resultados y observaciones

Las pruebas permitieron confirmar que el juego es jugable de principio a fin y que las mecánicas principales responden de forma estable. Los problemas encontrados estuvieron relacionados principalmente con detalles de colisión, liberación de memoria en ciertos objetos (se trababa el juego al reiniciar muchas veces) y ajustes de dificultad (por ejemplo, niveles demasiado fáciles o difíciles en las primeras versiones).

Tras varias iteraciones de corrección, se consiguió una experiencia de juego más equilibrada, con transiciones más fluidas y sin errores críticos que impidieran completar el videojuego. No obstante, aún se identifican posibles mejoras en el refinamiento de la IA de algunos enemigos (sobretodo en el segundo nivel) y en la presentación de mensajes de retroalimentación al usuario.

## 9. Guía de Instalación y Uso

### 9.1. Instrucciones para instalar y ejecutar el software

El proyecto fue desarrollado utilizando Qt Creator y el framework Qt para C++. Para compilar y ejecutar el juego a partir del código fuente, se siguieron los siguientes pasos:

1. Abrir Qt Creator y seleccionar la opción *Open Project*.
2. Cargar el archivo de proyecto principal (por ejemplo, `mi_primer_interfaz.pro` o equivalente).
3. Seleccionar un kit de compilación compatible, como *Desktop Qt 6.x MinGW 64-bit*.
4. Configurar el modo de compilación en *Debug* o *Release*, según se requiera.
5. Compilar el proyecto y, una vez finalizado el proceso, ejecutar el juego desde Qt Creator o desde el ejecutable generado en la carpeta de *build*.

En caso de distribuir únicamente el ejecutable, el usuario debe descargar la carpeta de *build* completa (incluyendo las bibliotecas de Qt necesarias) y ejecutar el archivo `.exe` correspondiente al juego en un sistema Windows compatible.

### 9.2. Requisitos técnicos y recomendaciones

Los requisitos técnicos mínimos estimados para ejecutar el juego son:

- Sistema operativo: Windows 10 o superior.
- Procesador: CPU de dos núcleos.
- Memoria RAM: al menos 2 GB.
- Tarjeta gráfica: integrada, con soporte básico para aplicaciones 2D.
- Espacio en disco: suficiente para almacenar el ejecutable y las librerías de Qt asociadas.

Se recomienda utilizar una resolución de pantalla de, al menos,  $1366 \times 768$  píxeles para visualizar correctamente las escenas del juego.

En cuanto a los controles, el videojuego utiliza el teclado para el manejo del personaje. De forma general:

- Flechas de dirección o teclas WASD: movimiento del personaje (izquierda, derecha, arriba, abajo, dependiendo del nivel).
- Tecla dedicada al salto (en el nivel de desplazamiento lateral).
- Tecla de disparo: generación de proyectiles hacia la dirección de ataque definida.

La combinación exacta de teclas puede ajustarse en el código o en la configuración del proyecto, pero se ha buscado mantener un esquema de control estándar para facilitar el aprendizaje del jugador.

## 10. Resultados y Discusión

### 10.1. Funcionamiento logrado, limitaciones actuales, dificultades enfrentadas y mejoras planeadas

#### Funcionamiento Logrado y Requisitos Cumplidos

El proyecto **Infierno en Okinawa** logró cumplir satisfactoriamente con la mayor parte de los requisitos funcionales y de diseño de POO impuestos, incluyendo:

- **POO y Arquitectura:** Implementación completa de la jerarquía de clases de juego con herencia de `QGraphicsPixmapItem` y `QObject`, demostrando el uso de **Herencia Múltiple** y **Polimorfismo** (ej. diferentes comportamientos en `PlayerItem` y `TopDownPlayerItem`).
- **Mecánicas de Juego Múltiples:** Se logró implementar con éxito 2 niveles con mecánicas distintas (Plataforma, Top-Down de Supervivencia Simple), demostrando la capacidad del controlador `GameWindow` para gestionar transiciones complejas.
- **Físicas y Algoritmos:** Se implementaron los tres modelos físicos requeridos: gravedad y salto parabólico, movimiento rectilíneo para balas, y la IA de persecución (*Seek*) simple en el `TopDownEnemy`.
- **Dificultad:** Se cumple la noción de dificultad mediante la abstracción de parámetros críticos (`lives_`, tiempo, tasa de aparición), haciendo que la dificultad sea controlada y programable.

#### Limitaciones y Requisitos Pendientes

A pesar de los logros funcionales, dos requisitos obligatorios de implementación no se integraron de manera nativa en el diseño final o se pospusieron, representando las principales limitaciones del proyecto:

- **Herencia Propia (No-Qt):** No se implementó una clase base puramente C++ (ej. `GameEntity`) de la cual heredaran las clases `*Item` para cumplir el requisito de abstracción de dominio no dependiente de Qt. La lógica de vida (`lives_`) se mantuvo dentro de las clases que heredan directamente de `QObject` y `QGraphicsPixmapItem`.
- **Uso de Excepciones:** El manejo de errores críticos (ej. fallo al cargar un recurso esencial, puntero nulo en el ciclo principal) se manejó mediante el sistema de advertencias de Qt (`qWarning` o `qCritical`) y retornos de función, en lugar de utilizar la estructura `try-catch` y la familia `std::exception`, lo cual es un requisito pendiente.

#### Dificultades Enfrentadas

La principal dificultad técnica fue la **gestión de la memoria y el desreferenciamiento de punteros** en el bucle principal de juego. El uso de `QPointer` en `GameWindow` fue una decisión técnica clave para mitigar los fallos por punteros colgantes (dangling pointers) que ocurren cuando un objeto es eliminado de la escena sin ser explícitamente puesto a `nullptr` esto nos generaba un crash en el juego algunas veces si y otras veces no.

## 11. Conclusiones

### 11.1. Reflexión sobre el proceso, el aprendizaje obtenido y recomendaciones

El desarrollo de *Infierno en Okinawa* permitió aplicar de forma práctica conceptos fundamentales de programación orientada a objetos, manejo de escenas gráficas y gestión de eventos en tiempo real. A lo largo del proyecto se pasó de una idea inicial relativamente abstracta a un producto jugable, lo que obligó a tomar decisiones concretas sobre la estructura del código, la organización de las clases y la distribución de responsabilidades entre los distintos módulos del juego.

Una de las principales lecciones aprendidas fue la importancia de diseñar una arquitectura clara desde el principio. La necesidad de reutilizar comportamientos comunes entre distintos tipos de enemigos y objetos puso en evidencia las ventajas de utilizar herencia y clases base, así como los límites de ciertas decisiones iniciales. Algunas de las limitaciones identificadas, como la falta de una clase de dominio verdaderamente genérica para todas las entidades del juego, abren la puerta a futuras refactorizaciones que podrían simplificar el mantenimiento y la extensión del código.

Desde el punto de vista técnico, trabajar con Qt y sus componentes gráficos permitió comprender mejor el manejo de escenas, la actualización periódica de objetos mediante temporizadores y la coordinación entre interfaz y lógica de juego. Asimismo, el proceso de depuración de errores relacionados con colisiones y memoria mostró la relevancia de las pruebas tempranas y de la instrumentación del código para detectar comportamientos inesperados.

Finalmente, el proyecto deja varias líneas claras de mejora. Entre ellas se encuentran la incorporación de una IA más elaborada para los enemigos, la inclusión de más variedad de niveles y mecánicas, y la implementación de opciones de configuración para el jugador (controles, volumen, dificultad, entre otros). A pesar de estas posibles extensiones, el resultado obtenido cumple con el objetivo de construir un videojuego funcional que integra diferentes conceptos de programación vistos durante el curso, y constituye una base sólida para desarrollos futuros más complejos.

## 12. Referencias

### 12.1. Bibliografía, enlaces y documentación consultada

El desarrollo del proyecto se apoyó en la documentación oficial de los frameworks y bibliotecas utilizadas, así como en recursos académicos especializados en programación orientada a objetos en C++.

1. **Documentación Oficial de Qt:** Referencias extensivas sobre el sistema de señales y slots, la gestión de la memoria con `QObject` y `QPointer`, y el uso de `QGraphicsView` para el motor 2D.
2. C++. Documentación de la librería estándar (`std::`), el uso de memoria dinámica (`new`, `delete`) y las estructuras de excepción (`std::exception`).
3. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. (Consultado para la aplicación de patrones como MVC en el diseño de `GameWindow`).
4. Material de Clase y Guías de Laboratorio del curso 2598521 - Informática II, Universidad de Antioquia. (Consultado para la validación de los requisitos obligatorios de herencia y memoria).
5. Recursos externos y tutoriales sobre programación de videojuegos en 2D con C++ para la implementación de algoritmos de gravedad y detección de colisiones.

## 13. Anexos/Apéndices

### 13.1. Fragmentos de Código Importantes Comentados

Se incluyen fragmentos de código que demuestran la aplicación de los principios POO y las decisiones técnicas clave, particularmente en la gestión del bucle de juego y la abstracción de la dificultad.

#### Bucle de Juego y Aplicación de Físicas (`GameWindow::onTick`)

Este método es el corazón del sistema, ejecutado continuamente por `QTimer`, donde se aplican las físicas, se procesan las colisiones y se actualiza la lógica de los agentes inteligentes.

```

void GameWindow::onTick()
{
    // 1. Lógica del jugador (gravedad, salto)
    if (player_) {
        player_->updatePhysics();
        player_->checkPlatformCollisions(); // Detiene caída al tocar suelo
    }

    // 2. Lógica del jefe
    if (bunkerBoss_ && bunkerBoss_->isAlive()) {
        // La IA del jefe es puramente reactiva, esperando el objetivo.
        bunkerBoss_->startAttacking(player_);
    }

    // 3. Lógica de enemigos (IA de persecución)
    for (auto *e : survivalEnemies_) {
        // Aplica IA 'Seek' basada en la posición actual del jugador.
        e->updateBehavior(tickDeltaTime_);
    }

    // 4. Gestión de proyectiles y colisiones dinámicas
    // ... Código para gestionar el ciclo de vida de BulletItem y ProjectileItem ...

    // 5. Verificación de victoria/derrota
    checkGameStatus();
}

```

### Abstracción de Dificultad

Este método ilustra cómo se cumple el requisito de la abstracción de dificultad mediante la modificación de atributos críticos del juego, en lugar de cambiar la lógica base de las clases.

```

{
    tdPlayer_ = new TopDownPlayerItem();
    // AJUSTE CRÍTICO DE DIFICULTAD #1: Reducir vida del jugador.
    tdPlayer_->resetLives(4);

    // AJUSTE CRÍTICO DE DIFICULTAD #2: Reducir tiempo de supervivencia.
    survivalSecondsLeft_ = 40;

    // AJUSTE CRÍTICO DE DIFICULTAD #3: Aumentar la vida del bunker
    int health_ = 18;
}

```

### 13.2. Capturas de Pantalla (Ejemplos Visuales)

Se presentan capturas de pantalla de la interfaz principal y las dos mecánicas de juego para validar el cumplimiento de los requisitos funcionales de la Interfaz Gráfica (RF 1, 2 y RNF 1).

- Menú Principal:



■ Nivel 1 (lateral con scroll):



■ Nivel 2 (Top-Down fija):

