**Informe Técnico Final**

**Proyecto Final: Juego Dragon Ball**

**Autores:** Nikolas Ortega Suarez

Keiner Torres

**Asignatura:** Informática II

**Profesor:** Augusto Enrique Salazar Jimenez

**Programa:** Ingeniería de Telecomunicaciones

**Institución:** Universidad de Antioquia

**Fecha:** 13 de Julio de 2025

Contenido

[**Informe Técnico Final** 1](#_Toc203586895)

[**Proyecto Final: Juego Dragon Ball** 1](#_Toc203586896)

[**1. Introducción** 3](#_Toc203586897)

[**2. Arquitectura del Sistema** 4](#_Toc203586898)

[**3. Decisiones de Diseño e Implementación** 7](#_Toc203586899)

[**4. Desafíos de Desarrollo y Proceso de Depuración** 8](#_Toc203586900)

[**5. Plan de Desarrollo y División de Trabajo** 10](#_Toc203586901)

[**6. Conclusión** 11](#_Toc203586902)

[**7. Referencias** 11](#_Toc203586903)

[**8.Anexo A: Fragmentos de Código Clave** 12](#_Toc203586904)

**1. Introducción**

El presente informe documenta de manera exhaustiva el proceso de diseño, análisis y desarrollo de la arquitectura de software para el "Proyecto Final: Juego Dragon Ball". Este proyecto, desarrollado como requisito de la asignatura Informática II, representa la culminación de los conceptos teóricos aprendidos y su aplicación en un entorno práctico y complejo: la creación de un videojuego 2D funcional en C++ utilizando el framework Qt 6.

Nuestro objetivo principal fue construir una aplicación robusta, bien estructurada y escalable, aplicando rigurosamente los principios de la Programación Orientada a Objetos (POO). El juego se compone de tres niveles con mecánicas distintas: un primer nivel de esquivar obstáculos con vista cenital y dos niveles de combate cuerpo a cuerpo en 2D con vista lateral, enfrentando a un enemigo controlado por una Inteligencia Artificial (IA). Esta diversidad de mecánicas fue un desafío autoimpuesto para forzarnos a diseñar una arquitectura flexible y no monolítica.

Este informe detalla el proceso completo de desarrollo, desde la concepción inicial y el diseño de la arquitectura hasta la implementación de la lógica, la gestión de recursos gráficos y la resolución de desafíos técnicos. Se inicia con la descripción de la arquitectura de software elegida, justificando el uso de patrones de diseño como la separación de la lógica y la vista, y el uso extensivo de herencia y polimorfismo. Posteriormente, se profundiza en las decisiones de diseño clave, tanto a nivel de sistema como de jugabilidad, y se documenta de manera transparente el proceso de depuración, exponiendo los errores más significativos encontrados y las soluciones implementadas como parte fundamental de nuestro proceso de aprendizaje. Finalmente, se presenta el plan de trabajo colaborativo y se concluye con el estado final del proyecto.

**2. Arquitectura del Sistema**

**2.1. Filosofía de Diseño: Separación y Escalabilidad**

Desde el inicio, el principal desafío fue crear una base de código que pudiera soportar dos géneros de juego distintos (esquivar y pelear) sin caer en la trampa de un código "espagueti", difícil de mantener y extender. Para ello, adoptamos una filosofía de diseño basada en dos pilares fundamentales:

1. **Separación de Responsabilidades (Patrón MVC):** Se implementó una separación estricta entre la lógica del juego (el **Modelo**), su representación visual (la **Vista**) y la gestión de la entrada del usuario (el **Controlador**).
   * **Modelo:** Compuesto por clases de C++ puras que definen las reglas, los objetos, la física y los estados del juego. Es el núcleo del proyecto y es completamente agnóstico a cómo se dibuja en pantalla.
   * **Vista:** Responsabilidad de la clase MainWindow, que utiliza QPainter para renderizar el estado actual de los objetos del Modelo.
   * **Controlador:** La MainWindow captura los eventos de teclado y los delega a la clase Juego, que los traduce en acciones dentro del Modelo. Esta separación nos permitió desarrollar y probar la lógica de forma independiente a la interfaz gráfica, facilitando la depuración y el trabajo en paralelo.
2. **Abstracción y Polimorfismo:** Frente a la alternativa de un enfoque puramente procedural, se decidió usar la herencia como herramienta principal para modelar las relaciones entre los objetos. Clases base abstractas como Entidad y Nivel actúan como "contratos" que definen una interfaz común. Esto permite que el motor del juego, la clase Juego, los gestione de forma genérica a través de punteros a la clase base, sin conocer sus detalles específicos. Esta decisión, aunque requiere una planificación más cuidadosa, garantiza que el juego sea fácilmente escalable en el futuro (ej. añadir nuevos niveles o personajes con un impacto mínimo en la arquitectura existente).

**2.2. Diagrama de Clases Final**

La arquitectura final del proyecto se representa en el siguiente diagrama de clases UML. Este diagrama refleja la jerarquía de herencia, las relaciones de composición y agregación, y las clases de gestión que orquestan el flujo del juego.

Diagrama, Escala de tiempo

El contenido generado por IA puede ser incorrecto. [Imagen del diagrama de clases final]

(la generación de las flechas es rara por el modelo que entrega plantUML)

**2.3. Descripción Detallada de Clases y Dependencias**

La arquitectura se centra en una jerarquía de clases bien definida, cuyo flujo de control y dependencias se describe a continuación:

* **MainWindow**: Hereda de QMainWindow y es la única clase con conocimiento directo de la UI de Qt. Su responsabilidad es capturar eventos del sistema operativo (teclado, dibujado) y delegarlos a la capa lógica. Contiene una instancia de Juego y los QPushButton de la interfaz.
* **Juego**: Es el motor y director general. No hereda de ninguna clase de juego, sino que gestiona el estado global (MENU, JUGANDO, etc.) y el ciclo de vida de los niveles. Posee un puntero polimórfico Nivel\* nivelActual, lo que le permite controlar cualquier nivel que cumpla con la interfaz Nivel. Es el dueño del QTimer principal y del mapa de recursos std::map<std::string, QPixmap>.
* **Nivel**: Es una clase base abstracta que define el "contrato" para cualquier escena jugable. Obliga a sus clases derivadas a implementar métodos esenciales como inicializar(), actualizar(), dibujar() y los métodos de procesamiento de input.
* **Nivel\_1 y Nivel2\_3**: Son las implementaciones concretas de Nivel. Cada una contiene la lógica específica de su modo de juego y gestiona sus propios objetos a través de contenedores std::vector de punteros a Entidad. Por ejemplo, Nivel\_1 gestiona el temporizador de supervivencia y la generación de Obstaculo, mientras que Nivel2\_3 gestiona el combate y las Hitbox.
* **Entidad**: Es la clase base abstracta de todos los objetos visibles en el juego. Consolida las propiedades comunes (posición, tamaño, velocidad) y define una interfaz de actualización y dibujado. Sus atributos son protected para permitir un acceso eficiente desde las clases hijas en los cálculos de física, una decisión de diseño que prioriza el rendimiento sobre una encapsulación estricta. Su destructor virtual es crucial para la correcta gestión de la memoria.
* **Luchador**: Hereda de Entidad y añade la lógica de combate: vida, daño, gravedad y ataques. Es también una clase abstracta, ya que define ataques (atacarPuño, atacarPatada) que deben ser implementados por sus hijos.
* **PersonajeJugadorNivel1, PersonajeJugadorCombate, Enemigo**: Son las clases concretas que representan a los actores. Las clases de jugador se especializan para cada mecánica, mientras que Enemigo añade una capa de IA.
* **Hitbox y Obstaculo**: Heredan de Entidad y representan objetos con propósitos específicos, sin la complejidad de un Luchador.
* **GameTypes.h**: Un archivo de cabecera auxiliar creado para centralizar enum class compartidos (PersonajeSeleccionado, TipoAtaque), evitando errores de redefinición y mejorando la organización del código.

**3. Decisiones de Diseño e Implementación**

**3.1. Decisiones de Diseño de Sistema**

* **Gestión de Recursos con std::map:** Para evitar la latencia de cargar imágenes desde el disco repetidamente, se implementó un sistema de gestión de recursos en la clase Juego. Se utiliza un std::map<std::string, QPixmap> donde cada sprite se carga una sola vez al inicio del juego y se asocia a una clave de texto (ej. "goku\_patada\_derecha"). Esto optimiza el rendimiento y centraliza la gestión de los recursos gráficos, facilitando su uso en las funciones de dibujo.
* **Especialización de Clases de Jugador:** Se identificó que el comportamiento del personaje del jugador era drásticamente diferente entre el Nivel 1 (movimiento libre en 4 direcciones sin gravedad) y los niveles de combate (con gravedad, salto y ataques). En lugar de usar una única clase con múltiples condicionales (if (nivel == 1)), se tomó la decisión de crear dos clases distintas: **PersonajeJugadorNivel1** (hereda de Entidad) y **PersonajeJugadorCombate** (hereda de Luchador). Esta especialización mantiene cada clase con una única responsabilidad, adhiriéndose al Principio de Responsabilidad Única y resultando en un código más limpio y fácil de entender.
* **Gestión de Memoria y Punteros:** El proyecto hace un uso extensivo de la memoria dinámica (new/delete) para la creación de objetos cuyo ciclo de vida es variable (niveles, obstáculos, hitboxes). Se consideró el uso de punteros inteligentes (std::unique\_ptr) como una alternativa más moderna y segura para la gestión automática de la memoria. Sin embargo, se optó por el manejo manual con punteros crudos para tener un control explícito sobre el ciclo de vida de los objetos y para demostrar una comprensión clara de la gestión de memoria en C++, un aspecto fundamental del curso. La implementación de **destructores virtuales** en todas las clases base (Entidad, Nivel) fue una decisión crítica para prevenir fugas de memoria.

**3.2. Decisiones de Diseño Visual y Temático**

* **Estilo Artístico:** Se decidió unificar el estilo visual del juego basándose en el popular juego de fans **DBZ: Devolution**. Este estilo se caracteriza por proporciones semi "chibi", colores vivos y saturados, y un contorno negro nítido que define las siluetas. Esta elección busca evocar nostalgia y garantizar una coherencia visual a lo largo de toda la experiencia de juego.
* **Diseño de Escenarios:** Cada nivel fue diseñado con una temática específica para reforzar la narrativa. El Nivel 1 se ambienta en un **vuelo sobre la Capital del Oeste** en dirección al Torneo de Artes Marciales, utilizando una autopista aérea de 4 carriles como área de juego funcional. Los Niveles 2 y 3 utilizan como fondo la **arena del Torneo Mundial**, proporcionando un contexto reconocible para los combates.

**3.3. Decisiones de Diseño de Jugabilidad y Modelos Físicos**

* **Ajuste de la Curva de Dificultad (Nivel 1):** Tras las pruebas iniciales, se observó que el Nivel 1 era demasiado lento y los obstáculos se acumulaban. Se tomaron las siguientes decisiones para ajustar la jugabilidad:
  1. Se asignó una **velocidad de caída inicial** a los obstáculos en su creación para que representaran una amenaza inmediata.
  2. Se incrementó significativamente el factor de **aceleración global** para que la dificultad aumentara de forma perceptible con el tiempo.
  3. Se limitó el área de generación de obstáculos y los límites de movimiento del jugador a la **vía central**, definiendo un área de juego clara y funcional.
* **Sistema de Combate por Hitbox:** Para los niveles de pelea, se implementó un sistema de Hitbox para la detección de golpes. La clase Hitbox hereda de Entidad y representa un área de ataque invisible y temporal. Esta decisión desacopla la lógica del ataque del personaje, simplifica la detección de colisiones y es un patrón estándar en los juegos de lucha. Cada Hitbox almacena un puntero a su propietario para evitar que un personaje se dañe a sí mismo.
* **Inteligencia Artificial (IA) por FSM:** Para el enemigo, se diseñó una IA basada en una **Máquina de Estados Finitos (FSM)**. El enemigo alterna entre un conjunto definido de estados (ESPERAR, ATACAR, DEFENDER), gobernados por temporizadores y la proximidad con el jugador. Esto produce un comportamiento que no es ni completamente predecible ni caótico, ofreciendo un desafío justo y manejable de implementar.
* **Implementación de Modelos Físicos:**
  1. **MRUA:** Se aplica en el Nivel 1, donde una variable aceleracionGlobal en Nivel\_1 incrementa la velocidadY de todos los Obstaculo en cada fotograma.
  2. **Movimiento Sinusoidal:** Ciertos obstáculos en el Nivel 1 usan la función sin() en su método actualizar() para generar un movimiento lateral oscilatorio, creando patrones de esquive más complejos.
  3. **Gravedad:** En la clase Luchador, un método aplicarGravedad() añade constantemente una fuerza hacia abajo a la velocidadY, simulando la gravedad y permitiendo saltos parabólicos.
  4. **Colisiones:** Se utiliza el método QRectF::intersects() de Qt para la detección de colisiones rectangulares entre las entidades.

**4. Desafíos de Desarrollo y Proceso de Depuración**

La transición del diseño teórico a la implementación práctica presentó una serie de desafíos técnicos, cuya resolución fue una parte fundamental del proceso de aprendizaje.

**4.1. Errores de Compilación por Inconsistencias de Interfaz**

* **Problema:** Durante el desarrollo, nos encontramos con una cascada de errores de compilación como no declaration matches '...' y '...' marked 'override', but does not override.
* **Diagnóstico:** El análisis reveló que estos errores se originaban por una inconsistencia en la firma de un método virtual a lo largo de la jerarquía de herencia. Específicamente, cuando se modificó el método dibujar() en la clase base Nivel para aceptar nuevos parámetros (el rectángulo de la ventana y el mapa de sprites), este cambio no se propagó correctamente a todas las clases derivadas y a las clases que las invocaban.
* **Solución:** Se aplicó una refactorización sistemática, actualizando la firma del método dibujar() en toda la cadena de llamadas (Nivel.h, Nivel\_1.h, Nivel2\_3.h, Juego.h y sus respectivos archivos .cpp). Este proceso subrayó la importancia de mantener la consistencia de las interfaces en un diseño polimórfico.

**4.2. Errores de Enlazado (Linker Errors)**

* **Problema:** En una etapa del desarrollo, el proyecto compilaba pero no enlazaba, mostrando un error de undefined reference to MainWindow::keyReleaseEvent.
* **Diagnóstico:** Este tipo de error indica que una función fue declarada en un archivo de cabecera (.h) pero nunca se proporcionó su definición en el archivo de implementación (.cpp). El compilador confió en la declaración, pero el enlazador no pudo encontrar el código correspondiente para unirlo al programa final.
* **Solución:** La solución fue simple pero fundamental: añadir la implementación de la función keyReleaseEvent en mainwindow.cpp, aunque su cuerpo estuviera vacío inicialmente.

**4.3. Errores de Lógica en Tiempo de Ejecución**

* **Problema:** El personaje no se movía a pesar de que no había errores de compilación.
* **Diagnóstico:** Se implementó un protocolo de depuración sistemático utilizando qDebug() para trazar el flujo de la señal de input. Se colocaron mensajes en cada eslabón de la cadena de llamadas: MainWindow::keyPressEvent, Juego::procesarInput, Nivel\_1::procesarInput y PersonajeJugadorNivel1::procesarTeclaPresionada.
* **Solución:** La depuración reveló que el evento keyPressEvent en MainWindow no se estaba disparando, mientras que keyReleaseEvent sí. Esto apuntó a un error sutil en la declaración del método en mainwindow.h. Tras una revisión minuciosa, se confirmó y corrigió el error, y se forzó una recompilación limpia del proyecto (**Clean Project -> Run qmake -> Rebuild Project**), lo que finalmente solucionó el problema. Este caso demostró la importancia de un proceso de depuración estructurado para aislar problemas lógicos.

**4.4. Errores de Gestión de Recursos**

* **Problema:** El compilador fallaba con el error No rule to make target '...', indicando que no podía encontrar los archivos de sprites especificados en el archivo de recursos (.qrc).
* **Diagnóstico:** El análisis de los mensajes de error reveló que los nombres de archivo contenían caracteres no estándar (como la ñ en Puño) y espacios, que causaban problemas de codificación y de rutas en el sistema de compilación.
* **Solución:** Se estandarizaron todos los nombres de archivos y carpetas, eliminando espacios (reemplazados por guiones bajos) y caracteres especiales. Posteriormente, se actualizó el archivo .qrc eliminando las entradas antiguas y añadiendo de nuevo los archivos con sus nombres corregidos, asegurando la sincronización entre los recursos físicos y su registro en el proyecto.

**5. Plan de Desarrollo y División de Trabajo**

Para facilitar el desarrollo concurrente y asegurar una distribución equitativa de la carga de trabajo, el proyecto se ha dividido en dos módulos funcionales de complejidad similar pero enfocados en áreas distintas del desarrollo.

* **Módulo 1: Motor de Juego y Lógica del Nivel 1 (Asignado a Nikolas Ortega)**
  + **Responsabilidades:** Implementación de la arquitectura central, el gestor de juego global, las clases base abstractas y la lógica completa del primer nivel.
  + **Clases Involucradas:** Juego, Entidad, Nivel, Nivel\_1, Obstaculo, PersonajeJugadorNivel1, MainWindow.
  + **Desafío Principal:** Correcta gestión de la memoria, el estado global de la aplicación y el polimorfismo.
* **Módulo 2: Sistema de Combate y Lógica de Niveles 2 y 3 (Asignado a Keiner Torres)**
  + **Responsabilidades:** Implementación de toda la mecánica de combate, la física de los personajes, la inteligencia artificial del enemigo y la gestión de los niveles de pelea.
  + **Clases Involucradas:** Luchador, PersonajeJugadorCombate, Enemigo, Hitbox, Nivel2\_3.
  + **Desafío Principal:** Implementación de algoritmos complejos para la IA (FSM) y el sistema de Hitbox.

Se estableció como procedimiento de colaboración la definición conjunta de todos los archivos de cabecera (.h) antes de la implementación, para asegurar la compatibilidad entre los módulos.

**6. Conclusión**

El desarrollo del "Proyecto Final: Juego Dragon Ball" ha sido un ejercicio integral que nos ha permitido aplicar y consolidar nuestros conocimientos en Programación Orientada a Objetos, diseño de software y resolución de problemas en un entorno práctico como es el desarrollo de videojuegos con C++ y Qt.

Hemos logrado construir una arquitectura de software modular y escalable, capaz de soportar diferentes mecánicas de juego de forma organizada. Las decisiones de diseño, como la separación de la lógica y la vista, el uso de herencia y polimorfismo, y la implementación de patrones como la Máquina de Estados Finitos, han demostrado ser acertadas para gestionar la complejidad del proyecto.

El proceso de depuración de errores de compilación, enlace y lógica ha sido una de las etapas más formativas, enseñándonos la importancia de un enfoque sistemático para el diagnóstico de problemas y la necesidad de mantener la consistencia en toda la base de código.

El estado final del proyecto es una aplicación funcional que cumple con los requisitos establecidos, con un Nivel 1 completamente jugable y una base sólida para la finalización de los niveles de combate. Consideramos que los objetivos del desafío han sido alcanzados con éxito.

**7. Referencias**

* Documentación Oficial de Qt 6. (s.f.). Recuperado de <https://doc.qt.io/>
* Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley Professional.
* Nystrom, R. (2014). *Game Programming Patterns*. Genever Benning.

**8.Anexo A: Fragmentos de Código Clave**

A continuación, se presentan fragmentos de código que ilustran algunas de las decisiones de diseño más importantes implementadas en el proyecto.

**Fragmento 1: Juego::cambiarNivel() - Gestión Polimórfica de Niveles**

void Juego::cambiarNivel(int numeroNivel)

{

// Liberar memoria del nivel anterior si existe.

delete nivelActual;

nivelActual = nullptr;

switch (numeroNivel) {

case 1:

nivelActual = new Nivel\_1(personajeActual);

break;

case 2:

nivelActual = new Nivel2\_3(personajeActual, 2);

break;

// ... otros niveles ...

}

// Si se creó un nuevo nivel, se inicializa y se cambia el estado.

if (nivelActual != nullptr) {

nivelActual->inicializar();

estadoActual = GameState::JUGANDO;

}

}

*Este método demuestra el uso del polimorfismo. El puntero nivelActual puede apuntar a cualquier tipo de nivel, y la clase Juego lo gestiona sin necesidad de conocer su implementación específica.*

**Fragmento 2: Nivel\_1::generarObstaculos() - Lógica de Juego Procedural**

void Nivel\_1::generarObstaculos(float deltaTiempo)

{

tiempoParaSiguienteObstaculo -= deltaTiempo;

if (tiempoParaSiguienteObstaculo <= 0) {

float posXAleatoria = 225 + static\_cast<float>(rand() % 350);

TipoObstaculo tipo = static\_cast<TipoObstaculo>(rand() % 3);

bool esSinusoidal = (rand() % 2 == 0);

Obstaculo\* nuevoObs = new Obstaculo(posXAleatoria, -60.0f, tipo, esSinusoidal);

nuevoObs->setVelocidadY(150.0f);

obstaculos.push\_back(nuevoObs);

tiempoParaSiguienteObstaculo = INTERVALO\_GENERACION\_OBSTACULO;

}

}

*Este fragmento muestra la lógica procedural dentro del Nivel 1 para la creación aleatoria de obstáculos, controlando su posición, tipo y patrón de movimiento.*

**Fragmento 3: Nivel2\_3::dibujar() - Selección Dinámica de Sprites**

std::string Nivel2\_3::getClaveSpriteLuchador(const Luchador\* luchador, bool esJugador) const

{

// ... (lógica para determinar el nombre base, acción y dirección) ...

return base + accion + direccion;

}

void Nivel2\_3::dibujar(...)

{

// ...

if (jugador) {

std::string clave = getClaveSpriteLuchador(jugador, true);

auto it = sprites.find(clave);

if (it != sprites.end()) {

painter->drawPixmap(jugador->getBoundingRect(), it->second, it->second.rect());

}

}

// ...

}

*Este ejemplo ilustra cómo la función dibujar delega la responsabilidad de elegir el sprite a una función ayudante, manteniendo el código limpio y organizado.*