Desarrollo de un Videojuego 2D de Aventura basado en Físicas con C++ y Qt

Carlos Daniel Rua Gutiérrez
Facultad de Ingeniería
Universidad de Antioquia
Medellín, Colombia
cdaniel.rua1@udea.edu.co

Tomás Mesa Sánchez
Facultad de Ingeniería
Universidad de Antioquia
Medellín, Colombia
tomas.mesa@udea.edu.co

***Abstract*—Este documento detalla el proceso de diseño, desarrollo e implementación de un videojuego 2D de acción y aventura, inspirado en el capítulo 105 del manga Dragon Ball. El proyecto fue desarrollado en C++ utilizando el framework Qt, con un enfoque en la aplicación de diferentes modelos físicos para la creación de mecánicas de juego variadas. Se describe el análisis del problema, la arquitectura de software propuesta, la lógica de los algoritmos no triviales, los desafíos técnicos afrontados durante el desarrollo y la evolución del proyecto desde su concepción inicial hasta su versión final.**

***Index Terms*—desarrollo de videojuegos, C++, Qt, QGraphicsScene, físicas de juego, diagrama de clases, diseño de software.**

I. Análisis del Problema y Solución Propuesta

A. Análisis del Problema

El objetivo principal del proyecto fue desarrollar un videojuego que recreara la narrativa y los desafíos presentados en el capítulo 105 de Dragon Ball, "El Gran Yayirobe". La trama involucra combates, obstáculos y la interacción entre personajes con habilidades distintivas. Para traducir esto a una experiencia jugable, se identificaron los siguientes requerimientos:

- Un personaje principal controlable por el jugador en un entorno con vista cenital (top-down).
- Múltiples niveles con temáticas y desafíos crecientes.
- Implementación de enemigos con comportamientos y patrones de movimiento diversos, basados en físicas específicas:
  - Movimiento rectilíneo y disparo predecible.
  - Movimiento parabólico (proyectiles de cañón).
  - Movimiento pendular (trampas de entorno).
  - Movimiento circular y orbital (escudos de jefe).
  - Movimiento de persecución simple.
- Un sistema de colisiones para gestionar las interacciones entre el jugador, los enemigos, los proyectiles y los límites del escenario.
- Un sistema de persistencia de datos para guardar y cargar el progreso del jugador.

Proyecto realizado para el curso de Informática II, Facultad de Ingeniería, Universidad de Antioquia, 2024.

B. Alternativa de Solución Propuesta

Para afrontar estos requerimientos, se eligió una solución tecnológica basada en el lenguaje **C++** y el framework **Qt**.

C++ fue seleccionado por su alto rendimiento, fundamental para videojuegos que requieren cálculos físicos y gestión de múltiples objetos en tiempo real. Su paradigma de programación orientada a objetos (POO) facilitó la creación de una estructura de clases modular y escalable.

El **framework Qt** se eligió por su robusto módulo *Graphics View Framework* (‘QGraphicsScene‘, ‘QGraphicsItem‘). Esta herramienta es ideal para videojuegos 2D basados en objetos, ya que simplifica enormemente la gestión de la escena, el renderizado, las transformaciones de objetos y la detección de colisiones. Además, Qt ofrece un sistema de señales y slots (‘signals‘ and ‘slots‘) que permite una comunicación desacoplada y eficiente entre los objetos del juego, así como herramientas para crear interfaces de usuario y gestionar eventos de entrada (teclado, ratón).

Esta combinación permitió construir una base sólida donde cada entidad del juego (personaje, enemigo, proyectil) es un objeto independiente con su propio comportamiento y estado, todos orquestados por una clase principal (‘MainWindow‘) que actúa como el motor del juego.

II. Estructura y Diseño de Clases

La arquitectura de la solución se fundamenta en un diseño orientado a objetos para encapsular la lógica y el estado de cada componente del juego. La estructura de clases se diseñó para promover la reutilización de código y la modularidad.

La clase base principal es ‘personaje‘, que define los atributos y métodos comunes a todas las entidades móviles, como la vida, la posición y las funciones de movimiento básicas. De esta clase heredan directamente las clases de enemigos (‘enemigo1‘, ‘enemigo2‘, ‘enemigo3‘), lo que permite tratarlos de forma polimórfica en el bucle principal del juego.

De manera similar, se definió una clase ‘muros‘ para representar los obstáculos invisibles del escenario, de la cual heredan otros objetos estáticos como ‘cannon‘ y ‘municion‘ para compartir propiedades de posicionamiento y tamaño.

La clase ‘MainWindow‘ actúa como el componente central y orquestador. Es responsable de inicializar la escena del juego

(‘QGraphicsScene‘), crear y gestionar las instancias de todos los objetos (conteniéndolos en listas como ‘QList‘ y ‘std::list‘), manejar el bucle de juego a través de ‘QTimer‘, procesar las entradas del usuario y aplicar la lógica de colisiones entre las distintas entidades. Esta clase centraliza el control del flujo del juego y el estado de los niveles.

III. Descripción Lógica de Subprogramas No Triviales

A. Física de Movimiento Parabólico (‘bolaCannon‘)

Para simular el lanzamiento de un proyectil de cañón, se implementó un modelo de tiro parabólico. La lógica no se basa en una animación predefinida, sino en el cálculo iterativo de la posición usando ecuaciones cinemáticas.

- **Estado inicial:** La bola de cañón se instancia con una posición inicial (x_0, y_0), una velocidad escalar inicial v_0 y un ángulo de lanzamiento θ.
- **Cálculo de velocidad:** En cada paso de tiempo ‘DT‘, se actualizan las componentes de la velocidad (v_x, v_y). v_x permanece constante (ignorando la resistencia del aire), mientras que v_y se ve afectada por la aceleración de la gravedad g: $v_y(t) = v_{y0} - g \cdot t$.
- **Cálculo de posición:** Con las velocidades actualizadas, se calcula la nueva posición: $x(t) = x_0 + v_x \cdot t$ y $y(t) = y_0 + v_{y0} \cdot t - \frac{1}{2} g \cdot t^2$. El método ‘CalcularPosicion‘ implementa una versión discreta de estas ecuaciones.

B. Física de Movimiento Pendular (‘pendulo‘)

La simulación del péndulo se basa en la física del movimiento armónico simple. En lugar de calcular una posición cartesiana (x, y) directamente, se simula la variación del ángulo.

- **Estado:** El estado del péndulo se define por su ángulo actual θ y su velocidad angular ω.
- **Cálculo de aceleración:** La aceleración angular α se calcula en cada instante con la fórmula $\alpha = -\frac{g}{L} \sin(\theta)$, donde L es la longitud del péndulo.
- **Integración numérica:** A través de un ‘QTimer‘ con un pequeño intervalo de tiempo ‘dT‘, se actualiza el estado del péndulo. Se utiliza un método de integración numérica (similar a la integración de Euler) para encontrar la nueva velocidad angular y el nuevo ángulo: $\omega_{t+1} = \omega_t + \alpha_t \cdot dT$ y $\theta_{t+1} = \theta_t + \omega_t \cdot dT$.
- **Actualización de posición:** Finalmente, la posición cartesiana (x, y) del objeto en la escena se calcula a partir del nuevo ángulo θ y la longitud L.

C. Bucle Principal y Lógica de Colisiones (‘MainWindow‘)

El corazón del juego reside en los métodos ‘actualizar()‘ y ‘nivel1()‘, llamados por uno o más ‘QTimer‘.

- **Ciclo de actualización:** El ‘QTimer‘ principal dispara un evento a intervalos fijos (ej. 100ms). Este evento ejecuta los métodos de actualización.
- **Actualización de estado:** Dentro de estos métodos, se recorren las listas de objetos activos (enemigos, proyectiles). Para cada objeto, se invoca su respectivo método de movimiento (‘move()‘, ‘actualizar()‘, etc.).
- **Detección de colisiones:** Después de mover los objetos, se realiza la detección de colisiones. Se utiliza el método ‘collidesWithItem()‘ de ‘QGraphicsItem‘. Se implementan bucles anidados para verificar colisiones entre:
  - Proyectiles del jugador y enemigos.
  - Proyectiles de enemigos y el jugador.
  - El jugador y los enemigos/obstáculos.
  - Objetos móviles y los muros.
- **Resolución de colisiones:** Al detectar una colisión, se aplica una consecuencia: se reduce la vida de un personaje, se elimina un proyectil de la escena (y de la lista), o se revierte el último movimiento de un personaje para que no atraviese un muro.

IV. Algoritmos Implementados

A continuación se presentan fragmentos de código clave que implementan las lógicas descritas anteriormente.

A. Movimiento Parabólico

El Listado 1 muestra cómo se actualiza la velocidad y la posición de ‘bolaCannon‘ en cada paso de simulación.

```
#define DT 0.04  // Intervalo de tiempo discreto
#define g 9.8    // Aceleracion de la gravedad
#define cr 0.8   // Coeficiente de restitucion (para
     rebote)

void bolaCannon::CalcularVelocidad()
{
    // Calcula las componentes de la velocidad
    vel_x = vel * cos(ang);
    vel_y = vel * sin(ang) - g * DT; // Aplica la
    gravedad

    // Recalcula la magnitud y direccion de la
    velocidad
    vel = sqrt(pow(vel_x, 2) + pow(vel_y, 2));
    ang = atan2(vel_y, vel_x);

    // Simula un rebote con perdida de energia
    if (posy < 0) { // Asumiendo que 0 es el suelo
        vel_y = -vel_y * cr;
    }
}

void bolaCannon::CalcularPosicion()
{
    // Actualiza la posicion usando la velocidad
    actual
    // Se aplica una integracion de Euler simple
    posx = posx + vel_x * DT;
    posy = posy + vel_y * DT - 0.5 * g * DT * DT;
}
```

Listing 1. Cálculo de velocidad y posición para tiro parabólico.

B. Movimiento Pendular

El Listado 2 ilustra la actualización del estado del péndulo mediante integración numérica.

```
#define dT 0.005 // Intervalo de tiempo para el pendulo
#define g 9.8

void pendulo::calculo()
{
    // 1. Calcula la nueva velocidad angular
    vAngular = vAngulari + (aAngular * dT);

    // 2. Calcula el nuevo angulo
    angulo = anguloi + (vAngular * dT) + (aAngular * dT * dT) / 2;

    // 3. Calcula la nueva aceleracion angular para el proximo paso
    aAngular = -g * sin(angulo);
}

void pendulo::actualizar()
{
    calculo(); // Llama a la funcion de fisica

    // 4. Actualiza la posicion cartesiana en la escena
    posx = longitud * sin(anguloi);
    posy = longitud * cos(anguloi);
    setPos(posx + 150, posy + 100); // Se suma el offset del pivote

    // 5. Prepara los valores para la proxima iteracion
    vAngulari = vAngular;
    anguloi = angulo;
}
```

Listing 2. Cálculo del estado de un péndulo.

C. Bucle de Actualización de Enemigos

El Listado 3 muestra una versión simplificada del bucle principal en ‘MainWindow‘ que gestiona a los enemigos de tipo ‘enemigo1‘.

```
void MainWindow::actualizar()
{
    // Itera sobre la lista de enemigos 'goku' (enemigo1)
    for (itEnemigos1 = goku.begin(); itEnemigos1 != goku.end(); ) {
        enemigo1* punteroEnemigo = *itEnemigos1;

        // Si el enemigo colisiona con un muro, invierte su direccion
        if (EvaluarColision(punteroEnemigo) == true)
            punteroEnemigo->velocidad *= -1;

        punteroEnemigo->move(); // Mueve al enemigo

        // Logica de colision de balas del personaje con este enemigo
        for (it = balasPersonaje.begin(); it != balasPersonaje.end(); ) {
            if (punteroEnemigo->collidesWithItem(*it)) {
                punteroEnemigo->vida -= 10; // Reduce la vida

                mapaEscena->removeItem(*it);
                delete *it;
                it = balasPersonaje.erase(it); // Borrado seguro del iterador
                continue; // Evita incrementar un iterador invalido
            }
            ++it;
        }

        // Si el enemigo no tiene vida, se elimina
        if (punteroEnemigo->vida <= 0) {
            mapaEscena->removeItem(punteroEnemigo);
            delete punteroEnemigo;
            itEnemigos1 = goku.erase(itEnemigos1);
    // Borrado seguro
        } else {
            ++itEnemigos1; // Avanza al siguiente enemigo si no fue eliminado
        }
    }
}
```

Listing 3. Bucle de juego para enemigos y colisiones.

V. Problemas de Desarrollo Afrontados

Durante el ciclo de vida del proyecto, se presentaron varios desafíos técnicos que requirieron análisis y soluciones específicas.

1) Gestión de Memoria en C++: El reto más significativo fue la gestión manual de la memoria. Todos los objetos dinámicos del juego (personajes, enemigos, proyectiles) se crearon con ‘new‘. Esto obligó a implementar un control estricto sobre el ‘delete‘ de cada objeto para evitar fugas de memoria. El punto más crítico fue la eliminación de objetos de una ‘std::list‘ o ‘QList‘ mientras se iteraba sobre ella, ya que borrar un elemento invalida el iterador. Fue necesario adoptar un patrón de "borrado seguro" (ej. ‘it = miLista.erase(it);‘) para garantizar la estabilidad del programa.

2) Ineficiencia de Timers Múltiples: Un primer enfoque de diseño consideró que cada objeto animado (‘proyectil‘, ‘pendulo‘) gestionara su propio ‘QTimer‘. Este diseño, aunque conceptualmente simple, demostró ser muy ineficiente. Con decenas de proyectiles en pantalla, el sistema operativo debía gestionar un número igual de timers, consumiendo recursos y degradando el rendimiento. La solución fue refactorizar el código para centralizar toda la lógica de actualización en un único bucle de juego principal, controlado por un ‘QTimer‘ en ‘MainWindow‘.

3) Complejidad en la Detección de Colisiones: Implementar un sistema de colisiones robusto fue complejo. Se debía gestionar la interacción entre múltiples tipos de objetos. Un problema recurrente era que, tras una colisión con un muro, el personaje podía quedarse "atascado". Esto requirió implementar una lógica de "reversión de movimiento": si se detecta una colisión después de un movimiento, se deshace inmediatamente ese movimiento, devolviendo al personaje a su posición anterior válida.

4) Fragilidad de las Rutas de Archivos: El sistema de guardado y carga de partidas inicialmente utilizaba rutas de archivo relativas (ej. ‘../textos/usuarios.txt‘). Esto hacía que la aplicación fallara si se ejecutaba desde un directorio diferente al esperado. Fue necesario estandarizar las rutas de acceso a los archivos para que el programa fuera portable y robusto.

VI. Evolución de la Solución y Trabajo Futuro

A. Evolución del Proyecto

El proyecto evolucionó significativamente desde su concepción.

1) **De la Idea a la Estructura:** Se partió de una descripción narrativa y se tradujo a una arquitectura de software orientada a objetos, definiendo una jerarquía de clases clara con ‘personaje‘ como clase base.
2) **Implementación de Físicas:** Se implementaron de forma aislada los diferentes modelos físicos (parabólico, pendular), para luego integrarlos en el juego como mecánicas de enemigos y trampas.
3) **Centralización del Control:** El rol de ‘MainWindow‘ creció para convertirse en el núcleo del juego, manejando el bucle principal, la creación de niveles y la orquestación de todas las interacciones, lo que llevó al patrón de diseño "God Object".
4) **Cambios en la Narrativa:** El cambio más importante respecto al plan original fue el protagonista. Se planeó que el jugador controlara a Yayirobe, pero en la versión final se optó por un héroe genérico que se enfrenta a los antagonistas de la saga, incluido Goku. Este giro recontextualizó la experiencia, pasando de ser una recreación fiel a una aventura inspirada en el universo Dragon Ball.

B. Consideraciones para Trabajo Futuro

Para futuras versiones del proyecto, se recomiendan las siguientes mejoras:

- **Uso de Punteros Inteligentes:** Migrar el código de punteros crudos (‘new‘/‘delete‘) a punteros inteligentes de C++11. Esto automatizaría la gestión de memoria, prevendría fugas y haría el código más seguro y moderno.
- **Refactorización de ‘MainWindow‘:** Para combatir el anti-patrón "God Object", se debería refactorizar ‘MainWindow‘, delegando responsabilidades a clases más especializadas, como un ‘CollisionManager‘, un ‘LevelManager‘ o un ‘InputHandler‘.
- **Diseño Guiado por Datos:** Las propiedades de los niveles, enemigos y objetos están actualmente "hardcodeadas" (escritas directamente en el código). Una mejora sustancial sería adoptar un enfoque guiado por datos, leyendo estas configuraciones desde archivos externos (ej. JSON o XML). Esto permitiría diseñar y balancear niveles sin necesidad de recompilar el programa.
- **Optimización de Colisiones:** Para escenas con una gran cantidad de objetos, el sistema de colisiones actual (que compara cada objeto con todos los demás) podría volverse lento. Se podría implementar una estructura de datos de particionamiento espacial, como un Quadtree, para optimizar la detección de colisiones.

References

[1] Documentación Oficial de Qt. The Qt Company. Disponible en: https://doc.qt.io/

[2] Stroustrup, B. The C++ Programming Language. 4th ed. Addison-Wesley, 2013.

[3] Nystrom, R. Game Programming Patterns. Genever Benning, 2014.