**Proyecto Final**

Carlos Andrés Buelvas

Facultad de ingeniería, Universidad de Antioquia, Medellín

Informática II

Aníbal Guerra Soler y Augusto Salazar

13 de junio de 2025

Imagen que contiene Patrón de fondo

Descripción generada automáticamente

* **1. Descripción**

Goku y Bulma buscan una Esfera del Dragón en el fondo del océano, pero al no poder alcanzarla, deciden explorar una isla cercana. Mientras Goku intenta bucear, Bulma es atacada por soldados del Ejército del Listón Rojo, quienes la confunden con una amenaza. El jugador controlará a Goku, quien debe proteger a Bulma de los soldados mientras atraviesa la isla, evitando obstáculos y derrotando enemigos antes de llegar a un punto seguro.

**2. Estructura del Videojuego**

**Niveles Propuestos:**

* **Nivel 1:** *"Escape en la Isla"* (Estilo *Banana Kong* con scroll lateral automático).
* **Nivel 2:** *"Rescate de Bulma"* (Combate final para rescatar a Bulma).

**Nivel 1: "Huida en la Playa"**

**Dinámica:**

* **Scroll lateral automático:** Goku corre hacia la derecha mientras el fondo se mueve.
* **Objetivo:** Llegar al final del nivel sin chocar o colisionar.
* **Mecánicas:**
  + **Salto parabólico** (gravedad y velocidad inicial ajustable).
  + **Movimiento pendular** (balanceo en cuerdas).
  + **Movimiento rotacional/traslacional** (troncos giratorios)
  + **Plataformas móviles** (altura variable fija).

**Retos:**

* **Obstáculos:**
  + Troncos rotatorios.
  + Rocas y troncos estáticos.
* **Enemigos:** Soldados del Listón Rojo disparando proyectiles.

**Físicas Implementadas:**

1. **Movimiento Parabólico** (salto de Goku con aceleración gravitacional).
2. **Movimiento Pendular** (balanceo en cuerdas con ecuaciones de péndulo simple).
3. **Movimiento Rotacional/traslacional**.

**Nivel 2: "Encuentro con Bulma"**

**Dinámica:**

* **Objetivo:** Recolectar las llaves para abrir la puerta y rescatar a Bulma
* **Mecánicas:**
  + **Sistema de energía acumulable** (cápsulas aumentan la barra).
  + **Disparo rectilíneo con rebote** (Kamehameha).
  + **Soldados patrullando y disparando cuando detectan a Goku** (movimientos predecibles).

**Físicas Implementadas:**

1. **Trayectoria de proyectiles** (disparos enemigos con caída parabólica).
2. **Colisiones** (Kamehameha y disparo contra paredes)

**3. Controles**

* **Tecla W:** Salto (parabólico).
* **Tecla S:** Bajar de plataformas.
* **Tecla P:** Disparar Kamehameha (si la barra está llena).

**4. Dificultad**

* **Velocidad incremental** (el scroll acelera progresivamente).
* **Aparición aleatoria de enemigos y obstáculos.**
* **Sin vidas.**

**5. Sprites Necesarios**

* **Personajes:** Goku (corriendo, disparando, cayendo), Bulma, Soldados.
* **Escenario:** Playa, palmeras, lianas, plataformas, rocas.
* **Objetos:** Cápsulas de energía, Kamehameha, proyectiles enemigos.

**6. Tecnologías y Requisitos**

* **Lenguaje:** C++ con **Qt Creator**.
* **GUI:** Interfaz gráfica con **QGraphicsView**.
* **POO:** Clases para personajes, enemigos, físicas y niveles.
* **Memoria dinámica:** Uso de punteros y contenedores de Qt.

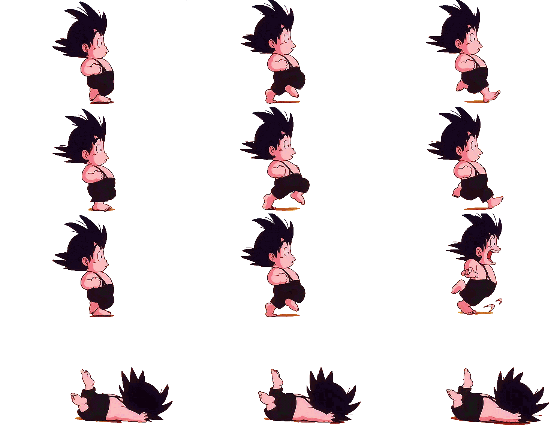
**Cambios respecto al Momento I**

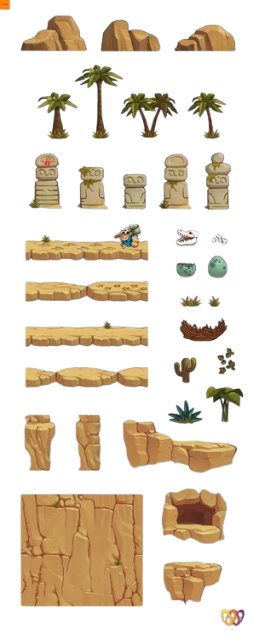
* Se definieron dos niveles: el primero basado en el estilo runner con scroll y dificultad media-alta; el segundo inspirado en *Bad Ice-Cream*, con vista superior.
* Se consolidó el enfoque del primer nivel: Goku corre en una isla con mecánicas de recolección de energía, enemigos que disparan, física pendular en lianas, salto parabólico y plataformas.
* Se agregó sistema de energía + barra para lanzar ataques tipo Kamehameha.
* Se estableció que Goku solo tendrá 1 vida, sin posibilidad de recuperarlas.

**3. Sprites por nivel**

**Nivel 1:**

* Sprite de Goku (correr, saltar, disparar)
* Sprite de soldado Patrulla Roja
* Sprite de cápsula de energía
* Sprite de tronco / obstáculo / plataforma





**Nivel 2:**

* Sprite de Goku (top-down)
* Sprite de enemigo patrullando
* Sprite de llaves (4 llaves)
* Textura de muros/obstáculos del mapa

**4. Descripción de vistas, interacciones y físicas**

**Nivel 1: Escape en la isla**

* Goku corre por un escenario playero con scroll lateral.
* Obstáculos: troncos (algunos rotatorios), piedras.
* Recolecta cápsulas para llenar barra de energía.
* Al llenarse, puede disparar Kamehameha con barra espaciadora.
* Enemigos aparecen aleatoriamente y disparan.
* Goku salta con la tecla W (salto parabólico). Si se mantiene oprimida, la caída es más lenta.
* Lianas: se activan al tocarlas e inician movimiento pendular (oscilatorio).
* Una vida sin recuperación. Termina al matar o aniquilar cierta cantidad de soldados.



(Ejemplo de vista lateral con scroll)

**Nivel 2: Exploración tipo Bad Ice-Cream**

* Vista superior.
* Goku debe recolectar 4 objetos (llaves) en un mapa cerrado.
* Soldados patrullan. Si colisionan con Goku, pierde vida.
* Se requiere esquivar enemigos y llegar al punto objetivo.
* Movimiento libre en 4 direcciones.



(Ejemplo de vista superior o cenital)

**Desarrollo**

Durante el desarrollo de mi videojuego en C++ con Qt, he ido avanzando en varios aspectos fundamentales del juego, tomando decisiones técnicas con base en los requisitos del proyecto, los principios de herencia y reutilización, y también la experiencia visual y jugable que quiero lograr. El proyecto está inspirado en el episodio 46 de Dragon Ball y está siendo desarrollado como parte del curso de Informática II.

Desde el inicio, decidí estructurar el proyecto separando la ventana de inicio (interfaz principal) de la ventana del juego (nivel). Para eso creé una clase MainWindow que se encarga únicamente de lanzar la pantalla inicial, donde eventualmente estará el menú, y una clase Juego que contiene toda la lógica del primer nivel. Esta decisión fue tomada para mantener un orden claro entre lo que es interfaz y lo que es la dinámica del juego. Por ahora no es necesario trasladar todo a una sola clase, ya que mantener esta separación permite trabajar con mayor claridad y posibilidad de escalar el proyecto más adelante (por ejemplo, agregando más niveles o pantallas).

Comencé implementando al personaje principal, Goku. Para eso creé una clase base abstracta llamada Personaje, de la cual heredan tanto Goku como los enemigos. Esta decisión fue importante porque el proyecto exige aplicar herencia de forma coherente, y además me permite tener métodos comunes como mover() y disparar(), que cada subclase puede redefinir con su propio comportamiento. Para Goku implementé varias animaciones: correr, disparar, caer, y colgarse de una cuerda, cada una con su sprite correspondiente. Una de las primeras dificultades fue lograr que todas las animaciones se vieran fluidas y con un tamaño coherente dentro de la escena, ya que los frames de los sprites venían en dimensiones distintas. Para solucionarlo, me aseguré de escalar los pixmaps al mismo tamaño final, ajustando también su posición con setOffset para que Goku se viera siempre alineado al suelo y no flotando ni cortado.

Otro punto clave fue el movimiento. Implementé un sistema en el que Goku avanza automáticamente hacia la derecha, como en un endless runner tipo Banana Kong. Para dar la sensación de desplazamiento continuo, creé un sistema de scroll en el fondo, donde dos imágenes de fondo se mueven hacia la izquierda y se reposicionan cuando salen de la escena. Este scroll se ajusta a una velocidad que incrementa progresivamente para generar tensión en el jugador. Al principio tuve problemas con esto, ya que el fondo no se movía correctamente o lo hacía de forma limitada. Terminé corrigiéndolo, cambiando la forma en que se controlaban las posiciones del fondo y utilizando QGraphicsPixmapItem con reposicionamiento dinámico.

Implementé una barra de energía que se llena cuando Goku recoge cápsulas que aparecen aleatoriamente. Una vez llena la barra, Goku puede disparar. El disparo se visualiza como una bola amarilla de energía. Inicialmente Goku solo disparaba una bola, pero decidí que disparara tres seguidas para hacerlo más vistoso y efectivo. Además, agregué una animación de disparo que se activa cuando se presiona espacio, y para que no se viera tan rápida o abrupta, decidí repetir los frames varias veces y temporizar con QTimer.

En cuanto a los enemigos, decidí implementar soldados que patrullan horizontalmente, alternando direcciones. Estos enemigos detectan a Goku cuando se encuentra a una distancia determinada, y si lo detectan, se detienen y comienzan a disparar proyectiles rojos hacia la izquierda. Al principio, el patrullaje era muy lento y no estaba alineado con el scroll, pero corregí la velocidad y sincronización para que los enemigos desaparecieran si se salían de la escena. También hubo un problema con la altura de los enemigos, ya que no estaban alineados al mismo nivel del suelo que Goku. Después de ajustar su setPos con respecto al alto del sprite y a la altura del suelo (Y = 375), logré que todos aparecieran correctamente posicionados.

Uno de los errores más persistentes fue que, cuando un enemigo moría (por colisión con un disparo de Goku), sus proyectiles quedaban congelados en pantalla. La raíz del problema era que no estaba llevando correctamente un control de los proyectiles activos por cada enemigo. Para solucionarlo, implementé una lista proyectilesActivos dentro de cada enemigo que guarda tanto el QGraphicsEllipseItem del proyectil como su QTimer correspondiente. Al momento de morir el enemigo, recorro esa lista, borro los proyectiles de la escena y detengo sus timers. Esto garantizó que no quedaran efectos visuales colgados en la escena, algo que visualmente dañaba bastante la experiencia.

Otro detalle que trabajé fue la detección de colisión entre Goku y los enemigos o sus disparos. Cuando esto ocurre, por ahora se activa una animación de caída para Goku, lo que simula daño. Más adelante esto será reemplazado por una transición de Game Over o pérdida de vida. También corregí detalles como hacer que los enemigos disparen solo hacia la izquierda, para mantener coherencia visual y lógica con la posición del personaje.

Con respecto a la eficiencia, comencé a revisar partes del código que podían optimizarse. Una de las sugerencias que tomé fue comenzar a reemplazar QList por QVector, ya que este último ofrece mejor rendimiento en operaciones de acceso y recorrido, que es justamente lo que más hago en actualizar(). También estoy evaluando si crear una clase Proyectil podría ayudar a reutilizar más código sin hacerlo complejo. Por ahora lo he dejado como método en Personaje porque eso me permite cumplir con el principio de herencia sin agregar más clases, pero lo estoy considerando para más adelante si noto redundancia.

En resumen, he avanzado considerablemente en la lógica de juego, animaciones, interacción entre personajes, y control del entorno. Me he enfocado en que la experiencia sea visualmente coherente, jugablemente fluida, y estructuralmente clara. Aún me falta implementar el nivel 2, el menú final, mejoras en colisiones y optimizaciones, pero la base del nivel 1 ya está bastante consolidada. Considero que las decisiones que he tomado están fundamentadas tanto en los requisitos del curso como en las necesidades prácticas de un juego en tiempo real.

**Dinámica del nivel 1**

El juego cuenta con movimiento automático, desplazamiento tipo scroll, animaciones personalizadas, enemigos que detectan al jugador, sistemas de disparo, colisiones, recolección de ítems y ahora, una nueva funcionalidad: las plataformas dinámicas.

Inicialmente, la idea de agregar plataformas surgió con el fin de aumentar la complejidad del desplazamiento de Goku, y permitir que el jugador tenga más decisiones en cuanto a movimiento vertical y rutas posibles. Implementar esto no fue tan simple como solo mostrar plataformas en pantalla; fue necesario diseñar un sistema que las hiciera aparecer de manera aleatoria, que respetaran ciertas posiciones y que interactuaran correctamente con Goku.

Uno de los primeros retos fue asegurarse de que Goku no atravesara las plataformas desde arriba cuando caía sobre ellas. Para eso, se añadió una verificación dentro del método actualizar() de la clase Juego, que compara la base de Goku con la posición Y de la plataforma, asegurando que sólo si viene cayendo y está por encima, se considere que ha aterrizado. Esto previene efectos visuales no deseados como que el personaje se "pegue" desde un costado o desde abajo.

Además, se estableció que Goku solo puede bajarse voluntariamente de una plataforma si presiona la tecla S. Para eso, dentro del método keyPressEvent, se invoca un método forzarCaida() que reestablece el estado de caída y permite abandonar la plataforma de forma controlada. Al principio, esto no funcionaba correctamente porque faltaban ciertos métodos en la clase Goku, como estaBajando(), detenerCaida(), activarCaida(), entre otros. Para solucionar este problema, fui añadiendo dichos métodos, encapsulando su lógica de manera eficiente para no romper la herencia con la clase base Personaje.

Otro detalle técnico que surgió fue que, visualmente, al caminar sobre las plataformas, Goku no se alineaba bien con la superficie. Esto hacía que pareciera flotando o mal posicionado. Lo solucioné ajustando manualmente la coordenada Y del personaje al colocarlo justo sobre la superficie de la plataforma (teniendo en cuenta la altura de su bounding box). Además, aumenté ligeramente el tamaño de las plataformas (más anchas) para mejorar la jugabilidad y dar más estabilidad visual.

Más adelante, quise mejorar aún más el dinamismo del juego haciendo que a veces aparezcan dos plataformas al mismo tiempo, pero cuidando que no se superpongan ni se vean amontonadas. Logré esto modificando el método generarPlataforma() para permitir que se generen una o dos plataformas de forma aleatoria, y en caso de que sean dos, se agregan con cierta separación horizontal y respetando un orden: la primera siempre es la más bajita y la segunda más alta. Esto además simula una especie de escalera natural, muy útil si el jugador desea subir niveles o esquivar enemigos y proyectiles desde las alturas.

Por supuesto, todo esto se fue integrando con el sistema general del juego, donde también ya había resuelto problemas importantes como la colisión entre proyectiles y enemigos, la desaparición correcta de proyectiles cuando un soldado muere, y la animación de caída de Goku cuando recibe daño, tanto de enemigos como de sus disparos.

En general, puedo decir que esta etapa del proyecto ha estado enfocada en darle más profundidad mecánica y verticalidad al escenario, combinando plataformas con enemigos y obstáculos, para fomentar una jugabilidad más rica y fluida. Además, se ha cuidado mucho la eficiencia del código, evitando punteros colgantes, borrado de objetos dentro del mismo contexto donde se usan, y estructuras como QVector para un manejo más rápido de colecciones. En ningún momento se ha presentado un cierre inesperado del programa, lo cual indica que la memoria y los ciclos de vida de los objetos están bien controlados.

Todavía faltan varias cosas por implementar, como el Game Over, vidas, reinicio de niveles y el nivel 2, pero con esta base sólida y eficiente ya se puede seguir avanzando con confianza y sin miedo a romper la lógica del programa. Todo ha sido construido paso a paso, resolviendo errores de forma progresiva y entendiendo bien cada decisión de diseño tanto a nivel visual como de código.

**Implementación del Sistema de Cuerdas en el Juego**

Durante el desarrollo del proyecto, se implementó un sistema de cuerdas/péndulos para el personaje principal (Goku), inspirado en mecánicas de juegos como \*Banana Kong\*. Este documento detalla el proceso, los problemas encontrados, las soluciones aplicadas y los resultados obtenidos.

**1. Objetivos Iniciales**

**- Movimiento pendular realista**: Que Goku se balancee al agarrarse a una cuerda.

- **Detección precisa de colisión**: Que la cuerda responda al contacto visualmente exacto.

- **Liberación automática/manual**: Que Goku se suelte al completar el balanceo o al presionar la tecla `S`.

- **Visual orgánico**: Cuerdas con curvatura y holgura natural.

**2. Problemas y Soluciones**

**Problema 1**: Movimiento Pendular Robótico

**Descripción**: Las cuerdas se movían de forma rígida, sin fluidez.

**Solución**:

- Se implementó **física basada en ángulos** usando:

double acceleration = -gravity / length \* sin(angle); // Ecuación del péndulo

angularVelocity += acceleration \* dt;

angle += angularVelocity \* dt;

```

- **Amortiguación** (`damping = 0.995`) para simular resistencia del aire.

**Problema 2**: Detección Imprecisa de Colisión

**Descripción**: Goku se agarraba a la cuerda sin tocarla visualmente.

**Solución**:

- Se ajustó el **área de colisión** usando `QRectF` alrededor del extremo de la cuerda:

QRectF(extremo.x() - 15, extremo.y() - 10, 30, 20);

- Se añadió **debug visual** con `qDebug()` para verificar las coordenadas.

**Problema 3**: Goku No Se Soltaba Automáticamente

**Descripción**: El personaje quedaba "pegado" a la cuerda indefinidamente.

**Solución:**

- Se añadió un **temporizador** y detección de fin de balanceo:

if (abs(angularVelocity) < 0.01 && abs(angle) < 0.1) {

soltarGokuDeCuerda();

}

**Problema 4**: Cuerdas Visualmente Planas

**Descripción**: Las cuerdas se veían como líneas rectas poco realistas.

**Solución**:

- Se implementaron **curvas Bézier** con `QGraphicsPathItem`:

QPainterPath path;

path.moveTo(origen);

path.quadTo(puntoMedio, extremo); // Curvatura dinámica

- **Punto medio dinámico** para simular holgura:

puntoMedio = (origen + extremo)/2 + QPointF(0, 25 + 15\*sin(angle));

```

**Problema 5**: Error de Compilación `no member named 'linea'`

**Descripción**: El código intentaba acceder a `cuerda.linea` después de cambiar a `QGraphicsPathItem`.

**Solución**:

- Se reemplazaron **todas las referencias** de `linea` por `cuerdaItem`.

- Se actualizó la lógica de colisión:

goku->collidesWithItem(cuerda.cuerdaItem); // En lugar de cuerda.linea

```

**3. Implementación Clave**

Estructura `Cuerda` (juego.h)

struct Cuerda {

QPointF origen; // Punto de anclaje

double largo = 220; // Longitud en píxeles

double angulo = -M\_PI/4; // Ángulo inicial (-45°)

QGraphicsPathItem\* cuerdaItem; // Ítem gráfico (curva Bézier)

QGraphicsPixmapItem\* gokuSprite; // Sprite de Goku agarrado

bool activa = false; // Si está en movimiento

QPointF puntoMedio; // Control de curvatura

};

```

Método `actualizarCuerda()`

void Juego::actualizarCuerda() {

for (Cuerda& cuerda : cuerdas) {

// 1. Actualizar física

if (cuerda.activa) {

cuerda.velocidadAngular += (-0.7 / cuerda.largo) \* sin(cuerda.angulo);

cuerda.angulo += cuerda.velocidadAngular;

}

// 2. Actualizar gráficos

QPointF extremo = calcularExtremo(cuerda);

cuerda.puntoMedio = /\* ... fórmula de curvatura ... \*/;

QPainterPath path;

path.moveTo(cuerda.origen);

path.quadTo(cuerda.puntoMedio, extremo);

cuerda.cuerdaItem->setPath(path);

}

}

```

**4. Resultados**

- **Movimiento realista**: Las cuerdas oscilan con física precisa y amortiguación.

- **Colisiones exactas**: Goku se agarra solo cuando toca visualmente la cuerda.

- **Liberación consistente**: Se suelta automáticamente al finalizar el balanceo o con la tecla `S`.

- **Visual atractivo**: Cuerdas con curvatura dinámica y grosor variable.

**5. Lecciones Aprendidas**

1. **Física > Gráficos**: Primero implementar la física correctamente y luego ajustar los gráficos.

2. **Debugging visual**: Herramientas como `qDebug()` y dibujar áreas de colisión son esenciales.

3. **Mantenimiento de código**: Actualizar todas las referencias al cambiar estructuras de datos.

**Desarrollo del Sistema de Game Over en el Juego Dragon Ball**

El primer intento de implementación mostró un problema fundamental: el mensaje de "GAME OVER" que se suponía debía aparecer en pantalla no era visible para el jugador. Esto ocurría porque el código emitía inmediatamente la señal para regresar al menú principal, lo que hacía que la ventana del juego se ocultara antes de que el mensaje tuviera oportunidad de mostrarse. Este comportamiento no solo frustraba la experiencia del jugador, que no recibía retroalimentación clara sobre su derrota, sino que también rompía la inmersión del juego.

Al analizar el problema en profundidad, identifiqué que necesitaba establecer un flujo de eventos más estructurado. La solución pasó por implementar un sistema de temporización que permitiera mostrar primero el mensaje de game over dentro del contexto del juego, con el estilo visual coherente con el resto de la interfaz (usando la fuente Dragon Ball, efectos de sombra y colores temáticos), y solo después de un breve período mostrar las opciones de reinicio o retorno al menú principal.

La implementación técnica requirió varias modificaciones importantes. En primer lugar, añadí un nuevo QTimer como miembro de la clase Juego, específicamente dedicado a manejar la secuencia del game over. Este temporizador se configuró como singleShot para asegurar que solo se disparara una vez. Luego creé un nuevo slot llamado mostrarMenuGameOver que se encargaría de emitir la señal gameOver después del retraso establecido.

El método mostrarGameOver fue rediseñado para cumplir con dos funciones principales: primero, mostrar el mensaje visual en pantalla con todos los efectos gráficos (texto rojo con sombra en la fuente Dragon Ball, centrado en la pantalla); y segundo, iniciar el temporizador que después de 2000 milisegundos activaría la transición al menú. Este tiempo de espera fue cuidadosamente elegido para dar al jugador suficiente oportunidad de leer el mensaje, sin hacer la espera demasiado larga como para resultar tediosa.

En la clase MainWindow, mantuve la lógica del QMessageBox personalizado que había implementado previamente, pero ajusté su comportamiento para que ahora apareciera después de que el jugador hubiera tenido tiempo de ver el mensaje in-game. El message box fue diseñado con una hoja de estilo que mantenía la coherencia visual con el tema Dragon Ball, usando tonos rojos y naranjas característicos de la serie.

Uno de los desafíos técnicos más interesantes fue asegurar la correcta liberación de recursos y el reinicio completo del estado del juego cuando el jugador elegía la opción "Reintentar". Esto requería no solo limpiar todos los elementos gráficos de la escena, sino también reinicializar todas las variables de estado del juego a sus valores por defecto. Implementé un método reiniciarJuego que se encargaba meticulosamente de esta tarea, eliminando enemigos, plataformas, obstáculos y cualquier otro elemento dinámico, antes de volver a crear los elementos iniciales del nivel.

El proceso de depuración reveló varios errores sutiles, como fugas de memoria cuando no se liberaban correctamente todos los QGraphicsItem, o problemas con los punteros a los elementos del juego durante el reinicio. Cada uno de estos problemas fue abordado con pruebas específicas y validaciones de punteros nulos.

La solución final logra un equilibrio efectivo entre retroalimentación inmediata al jugador (a través del mensaje de game over en pantalla) y usabilidad (con las opciones claras de continuar o salir). El flujo ahora es mucho más profesional y similar al de juegos comerciales: cuando Goku colisiona, aparece el mensaje de derrota en el contexto del juego, hay una breve pausa, y luego se presentan las opciones en un cuadro de diálogo modal que permite al jugador decidir si quiere intentarlo nuevamente o regresar al menú principal.

Un aspecto particularmente satisfactorio fue mantener la coherencia visual en todo el proceso, desde el mensaje de game over que usa la misma tipografía y efectos que el resto de la interfaz del juego, hasta el cuadro de diálogo que sigue la paleta de colores del estilo Dragon Ball. Esta atención al detalle contribuye significativamente a una experiencia de usuario pulida y profesional.

El sistema implementado no solo resuelve el problema inicial de las colisiones, sino que establece una base sólida para futuras expansiones, como diferentes tipos de mensajes de fin de juego (por ejemplo, para cuando el jugador complete el nivel), o la incorporación de efectos de sonido y animaciones durante la transición. La arquitectura modular permite que estos elementos adicionales se integren fácilmente sin requerir cambios significativos en el código existente.

Esta experiencia de desarrollo reforzó la importancia de considerar cuidadosamente el flujo de interacción en los juegos, especialmente en momentos críticos como el game over, donde una implementación descuidada puede frustrar al jugador. La solución final logra comunicar claramente el estado del juego mientras mantiene al jugador en control de sus opciones, todo ello dentro del marco visual y temático establecido por el universo Dragon Ball.

**Decisiones**

Desde el inicio, decidí dividir el juego en dos niveles bien diferenciados: el primero con una jugabilidad lateral al estilo Banana Kong, y el segundo con una vista cenital tipo Bomberman o Bad Ice-Cream. Esta decisión no solo obedeció a motivos creativos, sino también al cumplimiento de los requisitos de mi proyecto final para la asignatura de Informática II.

La estructura general del proyecto se basa en el uso de Qt y C++ utilizando herencia para los personajes, con una clase Personaje base y subclases como Goku y Enemigo. También cuento con una clase MainWindow que representa la interfaz principal, la cual tiene un tamaño fijo de 800x600, distinto del tamaño que se maneja en los niveles, que es de 1280x680.

Durante la implementación del primer nivel, una de las decisiones más importantes fue trabajar con un sistema de scroll continuo, donde el fondo se desplaza hacia la izquierda mientras Goku avanza automáticamente hacia la derecha. Esto permitió crear una experiencia visual dinámica, muy similar a juegos de plataformas de avance automático. También implementé enemigos, obstáculos, plataformas aleatorias, proyectiles, cápsulas y una cuerda colgante para balancearse.

Uno de los problemas que enfrenté fue que el juego se cerraba inesperadamente al disparar proyectiles. Este problema surgía porque al eliminar enemigos, el acceso al vector de proyectiles generaba una corrupción de memoria. Lo resolví asegurando que los proyectiles y enemigos se eliminen de forma segura y sincronizada con la escena, evitando llamadas peligrosas dentro de bucles de iteración.

Posteriormente, añadí la lógica para que, al eliminar una cantidad específica de enemigos, se mostrara un mensaje de “Nivel Completado” y se pasara automáticamente al segundo nivel. Esto trajo nuevos desafíos. Por ejemplo, inicialmente intenté detectar la clase MainWindow desde la clase Juego usando qobject\_cast, pero no funcionó porque el juego no estaba recibiendo correctamente al MainWindow como su padre. La solución fue emitir una señal nivelCompletado() desde Juego y conectarla en MainWindow, lo cual resultó ser una forma más elegante y segura de manejar la transición entre niveles.

Otro problema ocurrió al pasar al segundo nivel: el programa se cerraba repentinamente o la ventana quedaba en blanco. Esto pasaba porque no se estaba seteando correctamente el centralWidget ni ajustando el foco a la nueva escena del juego. Ajusté esto utilizando setCentralWidget y reestableciendo el foco con setFocus() y setFocusPolicy(Qt::StrongFocus) para asegurar que Goku pudiera recibir eventos de teclado desde el inicio.

Una dificultad adicional fue que, al pasar al segundo nivel, los controles con teclas WASD no respondían hasta presionar Tab. Esto lo solucioné estableciendo el foco explícitamente sobre Goku después de insertarlo a la escena. Aun así, noté que el QGraphicsView debía estar correctamente configurado como receptor de eventos, así que aseguré que Juego2 tuviera el foco activado inmediatamente con setFocus() tras el cambio de nivel.

Durante esta etapa final, una de mis necesidades fue poder seguir desarrollando y probando el segundo nivel sin tener que pasar por todo el nivel 1. Para lograrlo, creé una bandera bool modoDebugNivel2 = true; que, si está activada, salta directamente a iniciarNivel2() sin necesidad de completar el nivel 1. Esta funcionalidad me permite ahorrar tiempo de prueba mientras afino detalles del laberinto, los enemigos, la recolección de llaves, la cápsula de Bulma, y la puerta final que se abre cuando se recolectan todas las llaves.

En general, el proyecto ha sido un proceso de prueba y error constante, donde he aprendido a depurar errores lógicos, gestionar memoria, comunicar correctamente objetos entre sí mediante señales y slots, y asegurar que la experiencia del usuario sea fluida. Aunque hubo varios momentos de frustración cuando las cosas no funcionaban como esperaba, siempre traté de entender el porqué del fallo y corregirlo con la solución más limpia posible.

El siguiente paso será implementar la lógica completa de apertura de puertas, mostrar un mensaje de victoria al finalizar el nivel 2 y regresar al menú principal de forma ordenada. También deseo añadir efectos de sonido y música de fondo si el tiempo lo permite. Hasta el momento, el proyecto ha cumplido con los objetivos principales del curso, aplicando herencia, manejo de eventos, diseño modular y una interfaz visual atractiva.

**Nivel 2 del Videojuego (Laberinto - Estilo Cenital)**

El mapa lo representé mediante una matriz textual, usando caracteres específicos para indicar distintos objetos o entidades: # para muros, **.** para caminos, G para la posición inicial de Goku, K para llaves, E para enemigos, C para cápsulas, B para Bulma y D para la puerta. Este diseño textual me permite definir el mapa fácilmente y escalarlo si es necesario.

Uno de los primeros desafíos fue implementar correctamente la carga de ese mapa, asegurándome de que todos los elementos aparecieran en la escena con sus imágenes correspondientes, posicionados en las coordenadas correctas. Para esto, también necesité escalar previamente todas las imágenes según el tamaño de cada tile.

Una vez logrado eso, me enfoqué en que Goku se pudiera mover en cuatro direcciones (arriba, abajo, izquierda y derecha), usando W, S, A y D como teclas de control. Para que esto funcionara correctamente, creé animaciones separadas para cada dirección y definí un método mover2() heredado desde la clase base Personaje, el cual se encarga de mover a Goku siempre y cuando no colisione con un muro.

Luego, implementé el disparo para Goku. A diferencia del primer nivel, en este caso los disparos no son hacia una sola dirección sino hacia la última dirección en la que Goku se movió. Para ello, incorporé una variable direccionActual que se actualiza cada vez que se presiona una tecla de movimiento, y que luego se utiliza para determinar la dirección del disparo cuando el jugador presiona P. El proyectil solo puede ser disparado si la barra de energía está completamente llena.

La energía de Goku se recarga al recolectar cápsulas. Sin embargo, quise implementar una lógica especial: si Goku ya tiene la energía llena y toca otra cápsula, no pasa nada. Este comportamiento fue intencional, y lo controlé al verificar primero que energia < 100 antes de permitir absorber una cápsula. Una sola cápsula llena la barra por completo. Implementé también una barra de energía visible en pantalla que comienza vacía (gris) y se llena con color amarillo. Este cambio de color fue algo que inicialmente no ocurría correctamente, ya que la barra siempre empezaba amarilla. Para corregir eso, establecí su color inicial como gris y solo lo cambio a amarillo si la energía alcanza el valor máximo. También me aseguré de que, al disparar, la energía se reinicie a cero y la barra visual se actualice correctamente.

Durante este proceso surgieron varios errores y situaciones complicadas. Por ejemplo, al principio los proyectiles de Goku no colisionaban correctamente con los soldados enemigos. En teoría, debería eliminar tanto al enemigo como al proyectil si hay colisión. Sin embargo, el proyectil simplemente pasaba de largo. Esto se solucionó al verificar correctamente el data(0) del enemigo y asegurarme de que los enemigos tuvieran ese valor asignado. Más adelante, el juego incluso empezó a cerrarse repentinamente cuando un proyectil impactaba a un enemigo. Este bug se debía a que yo eliminaba el enemigo desde el lambda del QTimer, pero como el enemigo estaba aún siendo referenciado por otras partes del código (incluyendo la colección de enemigos), eliminarlo sin sacarlo antes del QVector causaba un fallo. La solución fue remover primero al enemigo del QVector y luego eliminarlo de la escena y liberar memoria con delete.

Otro detalle importante fue que inicialmente el atributo energia estaba definido en la clase Personaje, lo cual no tenía mucho sentido, ya que solo Goku debería tener energía y usarla. Por lo tanto, eliminé ese atributo de la clase base y lo coloqué en la clase Goku, manteniendo así una lógica más limpia y coherente.

También implementé un contador visual que muestra cuántas llaves ha recolectado Goku. Al tocar una llave, esta desaparece y el contador se incrementa. Cuando se alcanzan las 4 llaves, la puerta se abre automáticamente. Esa parte funcionó correctamente sin complicaciones mayores. Decidí implementar la puerta como un objeto con data(0) == "puerta", lo cual además me permite restringir el paso de Goku antes de que recolecte las llaves.

Después de que Goku recolecta todas las llaves y se abre la puerta, implementé la lógica para que Goku pueda avanzar hacia Bulma, que se encuentra en el mapa. Cuando Goku la toca, el juego muestra un mensaje de victoria con QMessageBox::information y se cierra la ventana. Esta parte la controlé mediante una bandera nivelCompletado para evitar que se repita el evento si por algún motivo Goku sigue colisionando con Bulma múltiples veces.

En cuanto al comportamiento de los enemigos, estos disparan automáticamente cuando detectan a Goku cerca. Sus proyectiles son de color rojo y si impactan a Goku, también terminan el juego (aunque por ahora solo se imprime un mensaje en consola y no se reinicia el nivel). Los enemigos también mueren de un solo disparo. Su animación se actualiza al disparar para que apunten hacia la dirección del proyectil.

Tuve algunos problemas con el acceso a atributos privados de la clase Enemigo, como frameActual, goku o disparandoNivel2. Para resolverlo correctamente, opté por crear métodos públicos getters y setters, en lugar de convertir todo en público, lo cual habría roto la encapsulación.

Por último, también encontré un error al tratar de usar directamente la instancia de Juego2 desde otros archivos como personaje.cpp, intentando invocar Juego2::obtenerInstancia(). Esto no tenía sentido en ese contexto porque no estaba usando un patrón singleton. La solución fue utilizar una señal desde Goku (emit actualizarBarraEnergiaSignal()) y conectarla a Juego2::actualizarBarraEnergia(), evitando así el acoplamiento entre clases innecesarias y respetando la lógica orientada a objetos de Qt.

En resumen, esta etapa del desarrollo fue una de las más complejas, pero también una de las más satisfactorias. Me permitió aplicar muchos conceptos avanzados de C++ y Qt como herencia, detección de colisiones, animaciones, señales y slots, timers y gestión de memoria. Aunque hubo varios errores y fallos inesperados, cada uno me ayudó a entender mejor cómo estructurar el código, aislar responsabilidades entre clases y evitar errores comunes como eliminar objetos aún referenciados. El nivel 2 ya está casi completamente funcional y me siento satisfecho con el avance. La siguiente etapa será probablemente mejorar los efectos visuales y trabajar en una transición más fluida hacia la pantalla final del juego.

**Mejoras funcionales**

Durante el desarrollo del videojuego, especialmente en la implementación del nivel 2, surgieron numerosos retos tanto técnicos como lógicos que requirieron ajustes importantes para lograr estabilidad, orden y funcionalidad adecuada en la interfaz gráfica y la dinámica del juego. El siguiente texto recopila todos los cambios, problemas encontrados, soluciones planteadas y decisiones de diseño que he venido trabajando durante la última parte del proyecto.

Al implementar el nivel 2, uno de los objetivos principales fue mantenerlo completamente separado del nivel 1 en términos de ejecución y comportamiento. Esta separación es necesaria porque ambos niveles utilizan lógicas distintas (scroll horizontal en el nivel 1 y movimiento estilo laberinto en el nivel 2), y por tanto deben ejecutarse de forma aislada. Esto fue logrado manteniendo dos clases diferentes: Juego y Juego2, donde cada una administra su propia escena, personajes, enemigos y comportamiento general.

Una de las primeras cosas que implementé en este nuevo nivel fue la funcionalidad de mostrar un título animado al iniciar el nivel. Para ello, dentro de la clase Juego2 creé el método mostrarTituloNivel(), que permite mostrar el nombre del nivel durante unos segundos y luego iniciar el juego. Durante ese tiempo, el juego debía estar pausado. Para lograr esto, decidí detener toda acción antes de que el título desaparezca, y solo tras su eliminación invocar iniciar().

Después enfoqué mis esfuerzos en integrar el botón de pausa al nivel 2, ya que en el nivel 1 esa funcionalidad ya estaba implementada y resultaba fundamental replicarla. Reutilicé gran parte del código del método togglePausa() y configurarBotonesPausa() desde Juego, pero lo adapté a Juego2 y lo asocié a un botón fijo con texto "PAUSA" ubicado en la parte superior derecha de la ventana. Para lograr una pausa real, tuve que asegurarme de que los enemigos, sus proyectiles y cualquier otra acción se detuvieran correctamente. Implementé un sistema de pausa dentro de la clase Enemigo, donde al llamar a setPausado(true) se detenían sus temporizadores de disparo y patrullaje, y también se pausaban todos los proyectiles activos. Este sistema fue complementado con métodos detenerProyectiles() y reanudarProyectiles() para tener un control completo.

Sin embargo, no todo fue tan sencillo. A lo largo de este proceso empecé a notar errores extraños y cierres inesperados del programa, especialmente al usar el botón “Salir al menú” desde el menú de pausa o al perder en el juego (cuando se muestra un QMessageBox). El programa se cerraba por completo sin dar errores claros en consola, lo cual me llevó a investigar profundamente qué estaba pasando.

Después de analizar el código y con la ayuda de varias pruebas, descubrí que el problema se originaba por el uso incorrecto del orden de destrucción y señales. En Juego2, cuando se ejecutaba emit salirAlMenu() y luego se llamaba togglePausa(), el programa intentaba ejecutar código sobre un objeto (Juego2) que ya había sido eliminado desde MainWindow, generando así un comportamiento indefinido que causaba el cierre del juego. Para solucionar esto, cambié el orden dentro de la conexión del botón “Salir al menú” en configurarBotonesPausa(), primero quitando el menú de pausa y solo después emitiendo la señal que lleva al menú principal.

Otro problema frecuente fue con los QMessageBox al finalizar el nivel, ya sea por derrota o victoria. En ciertos casos aparecían varios cuadros de mensaje al mismo tiempo, o el juego intentaba reiniciar el nivel y mostraba un QMessageBox mientras aún se estaban liberando recursos. Esto nuevamente estaba relacionado con llamadas y señales que se emitían múltiples veces o sin controlar adecuadamente los estados del juego. La solución fue asegurarme de que los connect() se hicieran con Qt::UniqueConnection, y también usé una variable de control (static bool mostrado) para evitar que el mensaje se mostrara más de una vez por error. Esto estabilizó el flujo y ya no aparecían múltiples ventanas inesperadas.

Además, uno de los detalles más importantes era que cuando Goku recibía un impacto de un proyectil enemigo, no se mostraba ningún mensaje de Game Over ni se reiniciaba el juego. Para resolver esto, integré una señal personalizada (gameOver) que se emite desde la clase Personaje o Juego2 cuando Goku recibe daño fatal. Pero hubo una dificultad: no podía emitir directamente un QMessageBox dentro del método disparar2() al detectar la colisión, ya que eso interrumpía abruptamente la lógica del juego (especialmente si estaba dentro de un QTimer). Para corregirlo, decidí emitir la señal gameOver desde Juego2 y dejar que MainWindow sea quien maneje el mensaje de derrota, lo cual permitió separar las responsabilidades y evitó los cierres inesperados.

A pesar de todas estas mejoras, sigo teniendo ciertos errores difíciles de reproducir, especialmente si alguna señal se dispara cuando un objeto ya fue eliminado. Por eso me aseguré de hacer disconnect(juego2, nullptr, this, nullptr); antes de conectar nuevamente señales en MainWindow::iniciarNivel2() y así evitar duplicaciones. También siempre verifico que juego2 o juego no sean nulos antes de usarlos, lo que ha contribuido a reducir los fallos silenciosos.

En resumen, estos últimos días de desarrollo se han centrado completamente en la estabilidad del juego, asegurando que las transiciones entre niveles, el menú principal, la pausa y los estados de derrota o victoria se manejen correctamente. He aprendido que, en una aplicación con Qt, el orden de destrucción, las señales, y el control del flujo de ejecución son clave para evitar errores que no se detectan a simple vista. Esta experiencia me ha permitido mejorar mi comprensión sobre programación de interfaces gráficas y gestión de objetos dinámicos, además de enseñarme a tener un mejor manejo de eventos asíncronos como QTimer y QMessageBox.

**Reflexiones**

Desde el principio, sabía que desarrollar un videojuego con C++ y Qt no iba a ser una tarea sencilla, pero decidí ir hasta el final, paso a paso, línea por línea. El concepto general del juego estaba inspirado en el universo de Dragon Ball, dividiendo la aventura en dos niveles diferenciados: el primero tipo plataforma dinámica, al estilo de Banana Kong, y el segundo tipo laberinto cenital, más estratégico y clásico, como Bomberman.

La mayoría del tiempo lo dediqué al desarrollo del primer nivel, ya que era más exigente en términos de físicas, movimiento y sincronización. Implementé sistemas personalizados de salto, detección de colisiones con plataformas y cuerdas, así como disparos condicionados al nivel de energía de Goku. Sin embargo, el verdadero reto comenzó cuando pasamos a construir el segundo nivel.

Desde que inicié la fase final del desarrollo del Nivel 2, comencé a encontrarme con una serie de problemas que, aunque esperados en un proyecto de esta magnitud, no dejaban de ser complejos y frustrantes. Uno de los primeros fue lograr que el botón de pausa del Nivel 2 funcionara igual al del Nivel 1. En el Nivel 1 todo estaba muy controlado, pero al intentar replicarlo en Juego2, muchos elementos no respondían como deberían, desde soldados que seguían patrullando, aunque el juego estuviera pausado, hasta los botones que simplemente no se conectaban bien a las señales.

Uno de los errores más molestos fue que al pulsar "Salir al menú" desde el menú de pausa, la aplicación se cerraba completamente. Analizando el código descubrí que las señales no estaban conectadas correctamente o el objeto MainWindow quedaba mal referenciado. Reorganizar esas conexiones dentro de main.cpp, MainWindow, y sobre todo asegurarme de que las instancias de Juego y Juego2 fueran bien destruidas o escondidas, fue lo que me permitió finalmente mantener el flujo normal del programa sin que colapsara.

También tuve que rehacer varias veces la forma en la que controlaba los estados del juego: cuándo debía pausar absolutamente todo, cuándo debía reproducirse una animación o sonido y cuándo debía mostrarse un QMessageBox. Me encontré con el problema de que el QMessageBox aparecía de golpe, sin mostrar bien el fondo de "Game Over" o sin dejar que se sintiera el impacto de la derrota o la victoria. Para esto, añadí retardos controlados por QTimer, pausas momentáneas y hasta efectos sonoros de fondo para crear una atmósfera más inmersiva.

A eso le sumé detalles que hicieron todo mucho más especial: una música de fondo distinta para cada nivel, efectos sonoros para disparos y saltos, un mensaje final que aparece cuando rescatas a Bulma con una imagen .gif y un QMessageBox estilizado con HTML y CSS para que todo mantuviera una estética Dragon Ball. Además, incorporé un botón llamado "Información" en el menú principal, que actúa como una pancarta para explicar cómo se juega, advertir errores conocidos y agradecer a quienes me apoyaron durante este camino. Incluso agregué una imagen llamativa que apunta al botón con una flecha como si fuera un aviso urgente tipo "¡Ey, léeme primero!"

Otro gran detalle técnico fue permitir el uso de enlaces clicables en las ventanas emergentes del juego. Esto lo logré usando QLabel con soporte para Qt::RichText, ya que QMessageBox por sí solo no los soporta. Este aspecto me permitió crear interfaces más modernas y con un poco más de vida.

Hubo errores en compilación por uso incorrecto de tipos incompletos (como QLabel y QGraphicsProxyWidget) y advertencias molestas como unused parameter, que resolví refinando la forma en la que escribía las lambdas en mis connect(). Uno de los problemas más rebeldes fue que los proyectiles de los enemigos impactaban a Goku pero no ocurría nada. Al final, agregué un sistema interno de verificación en Personaje::disparar2() para detectar el impacto correctamente, reproducir el sonido de derrota y mostrar un mensaje de “Has perdido”.

Y qué decir de la estética final. Generé imágenes con estilo Dragon Ball, como un título personalizado que dice “JUEGAZO” con colores explosivos, y otra que apunta al botón de información con una advertencia amistosa, todo para envolver al jugador en una experiencia coherente, divertida y con identidad visual fuerte.

La sensación de escuchar la música del menú al volver desde los niveles, de ver una pantalla que dice "¡Misión Cumplida!" tras rescatar a Bulma, y de notar que el menú es estático, inmersivo y lleno de personalidad, me confirma que todo este esfuerzo valió la pena.

**Conclusión del Proyecto**

Este videojuego no es simplemente un trabajo para la materia de Informática II. Es la representación más fiel del aprendizaje, la constancia, la frustración, los errores, los reinicios, las pruebas fallidas y, sobre todo, la pasión por crear algo con alma. A veces parecía que no iba a terminar, otras veces sentía que estaba creando algo mucho más grande de lo que imaginaba al principio. Pero al final, fue una mezcla perfecta de código, creatividad y determinación.

Gracias a este proyecto descubrí que puedo superar cualquier bug, incluso los que parecían imposibles. Descubrí también que un QMessageBox puede ser tan épico como una Genkidama si se diseña con cariño, que un botón puede fallar mil veces, pero seguir siendo mejorado, y que nada en la programación es más poderoso que el deseo de terminar algo con orgullo.

Y ahora que el juego está terminado, solo me queda decir lo siguiente:

Este no es un simple jueguito.

Es un JUEGAZO,

hecho a mano, a corazón y a puro teclado.

Gracias, Goku. Gracias a mí.

Gracias a todos los que saltaron, dispararon y se cayeron conmigo en cada prueba.

Ahora sí, proyecto completo.

Y como diría el maestro Roshi…

¡A entrenar para el siguiente nivel!