*María Angélica Osorio Rincón*

*Adriana Camila Erazo Mora*

**MOMENTO I**

**Capitulo seleccionado:** 77

**Nombre del capítulo:** El gran plan de Pilaf.

* **Sinopsis del capítulo base**

Goku está en búsqueda de la séptima esfera del dragón, él ya sabe quién la tiene y partirá en una misión para ir por ella. En el trayecto hasta encontrar a sus enemigos, deberá atravesar una serie de obstáculos hasta encontrarse con el auto que lleva dicha esfera. Cuando Goku logre conseguir la esfera se desplegará un mensaje que se muestra en el capítulo: “Tendremos que pelear y si tú resultas el ganador te daremos la esfera del dragón, pero si nosotros ganamos tú nos tendrás que dar las seis esferas del dragón que has reunido hasta ahora”. Luego, el jugador aceptará y avanzará hacia el siguiente nivel, donde Goku deberá reunir la suficiente energía para poder realizar su máximo poder (Kamehameha), de manera que pueda derrotar a su enemigo y salir victorioso con la esfera del dragón. A medida que Goku va acumulando pociones mientras debe esquivar los ataques del enemigo.

* **Dinámica del videojuego**

El objetivo central de Goku es recuperar la séptima esfera del dragón, que está en posesión del grupo de Pilaf. Para lograrlo, el jugador debe completar dos desafíos: una persecución aérea y una batalla final contra un robot enemigo.

* **Nivel I**
  + **Objetivo:** alcanzar el carro rojo donde se encuentran Pilaf y sus aliados, evitando obstáculos y preservando la vida de Goku.
  + **Dinámica general:** Goku se desplaza automáticamente hacia la derecha, montado sobre su nube voladora. El jugador puede controlarlo verticalmente con las teclas W (arriba) y S (abajo). El escenario se desplaza continuamente como si una cámara lo siguiera, revelando progresivamente obstáculos que se mueven en dirección contraria. Una barra de vida se despliega en la parte superior, junto con la distancia recorrida por Goku. Cuando Goku alcance al carrito, se detiene el movimiento horizontal automático, Goku da una patada contra el carro, el carro hace un movimiento de espiral, salen los 3 robots enemigos y se muestra el siguiente cuadro de diálogo: “Tendremos que pelear y si tú ganas, te daremos la esfera del dragón. Pero, si yo gano, tú me darás las seis esferas del dragón que has reunido hasta ahora.” El jugador acepta y se pasa al nivel 2.
  + **Físicas para implementar**
* MRU (Movimiento Rectilíneo Uniforme): Goku se desplaza automáticamente a la derecha con velocidad constante.
* Movimiento contrario en MRU: los obstáculos (aves, balas, montañas) se desplazan hacia la izquierda también con velocidad constante.
* Movimiento controlado en el eje Y (vertical): Goku puede desplazarse hacia arriba o abajo por decisión del jugador, también en un MRU controlado.
* Colisión sin rebote: si Goku colisiona con un obstáculo, se pierde vida sin retroceso ni rebote, simulando una absorción del impacto.
* Movimiento de espiral donde se emplean las siguientes físicas:

**Fase 0:** Subida

En esta fase, el carro sube a lo largo de una trayectoria vertical mientras se mueve horizontalmente. El movimiento en esta fase se describe mediante las siguientes ecuaciones:

*Posición horizontal (x):*

**𝑥(𝑡) = 127 + 𝑥0 + 𝑣𝑥⋅𝑡**

Donde:

**127** es el valor en pixeles que se le agrego para que cuando el carro entre en su fase 0 se aleje la distancia suficiente para evitar que en sus siguientes fases que el carro se vea sobre puesto en Goku.

**𝑥0** es la posición inicial en el eje x.

**𝑣𝑥** es la velocidad horizontal constante.

**𝑡** es el tiempo transcurrido desde el inicio de la fase.

*Posición vertical (y):*

**𝑦(𝑡) = 𝑦0 – 𝑣𝑦 ⋅ 𝑡 + ½ 𝑔 ⋅𝑡ˆ2**

Donde:

**𝑦0** es la posición inicial en el eje y.

**𝑣𝑦** es la velocidad inicial vertical.

**𝑔** es la aceleración debido a la gravedad.

**𝑡** es el tiempo transcurrido.

La fase termina cuando el carro alcanza el punto más alto, definido por la condición:

**𝑡 ≥ 𝑣𝑦/𝑔​**

Debido a que el momento en que la velocidad vertical 𝑣𝑦(𝑡) se vuelve cero. Por lo tanto, cuando 𝑡 ≥ 𝑣𝑦/𝑔, se ha alcanzado la máxima altura y la fase de subida termina.

**Fase 1:** Círculo

En esta fase, el carro describe un círculo a partir de la posición más alta alcanzada en la fase anterior. Las ecuaciones del movimiento circular son:

*Posición horizontal (x):*

**𝑥(𝑡) = 𝑥centro + 𝑟 ⋅ cos(𝜔 ⋅ 𝑡)**

*Posición vertical (y):*

**𝑦(𝑡) = 𝑦centro - 𝑟 ⋅ sin(𝜔 ⋅ 𝑡)**

Donde:

**𝑥centro** , **𝑦centro​** son las coordenadas del centro del círculo.

**𝑟** es el radio del círculo.

**𝜔** es la velocidad angular, que se calcula como:

**𝜔 = 2𝜋/𝑇giro​**

**𝑇giro​** es el tiempo que tarda en hacer una vuelta completa.

La fase termina cuando el tiempo **𝑡** alcanza **𝑇giro** debido a que **𝑇giro** es el tiempo necesario para que el carro realice una vuelta completa.

**Fase 2:** Caída

En esta fase, el carro comienza a caer debido a la gravedad. El movimiento es vertical y está gobernado por las siguientes ecuaciones:

*Posición vertical (y):*

**𝑦(𝑡) = 𝑦inicio + ½ 𝑔 ⋅ 𝑡ˆ2**

Donde:

**𝑦inicio** es la posición de inicio en la caída.

**𝑔** es la aceleración debido a la gravedad.

**𝑡** es el tiempo transcurrido desde que comienza la caída.

La fase termina cuando el carro alcanza el suelo, lo que se determina con la condición:

**𝑦(𝑡) ≥ ysuelo​**

Debido a que ysuelo es el valor de margen del nivel.

* + **Retos**
* Colisiones: Goku pierde vida al colisionar con los obstáculos (aves, rocas, montañas).
* Movimiento en dos ejes: aunque el avance es automático, el jugador puede controlar la altura de Goku, lo que añade complejidad al esquivar.
* Objetos móviles programados fuera de pantalla: pájaros, rocas y montañas se generan fuera del campo visual y entran al área visible siguiendo trayectorias definidas.
* **Nivel II**
  + **Objetivo:** recolectar 5 esferas de energía para cargar su Kamehameha y poder destruir al robot.
  + **Dinámica general:** El jugador controla completamente el movimiento de Goku (izquierda, derecha y saltar). En el escenario aparecen cápsulas de energía desde la parte superior, cayendo verticalmente. Además, en la esquina inferior izquierda se encuentra un robot gigante que dispara y lanza ataques a Goku, quien debe esquivarlos y recolectar 5 esferas de energía. Cuando Goku recolecte el total de esferas, se activa el Kamehameha (ataque único de Goku), con el cual puede atacar al robot y eliminarlo, para así obtener la séptima esfera del dragón y ganar el juego.
  + **Físicas para implementar:**
    - * Movimiento horizontal controlado (MRU): Goku se puede mover hacia la derecha o izquierda con velocidad constante, controlado por el jugador.
  + Movimiento vertical con salto (Movimiento parabólico): Goku puede saltar con una trayectoria de tipo parabólica, siguiendo una función cuadrática que representa un lanzamiento vertical afectado por la gravedad.

Donde es la velocidad inicial del salto y es la gravedad (9.8 )

* + Caída de esferas de energía (MRU vertical): las esferas de energía caen verticalmente desde el cielo con velocidad constante.
  + Proyectiles del robot (Movimiento parabólico y/o rectilíneo): el robot lanza dos tipos de ataques:
    - * + Rayos rectilíneos horizontales o diagonales:
      * Misiles en parábola:
    - **Retos**
* Cápsulas de energía: recolectar las 5 esferas, esquivando todos los ataques del robot.
* Coordinación de movimientos: el jugador debe combinar movimientos horizontales y saltos precisos para esquivar ataques y recolectar cápsulas en tiempo real.
* Velocidad de reacción: los proyectiles enemigos pueden tener trayectorias distintas y velocidad variable, exigiendo atención constante.
* Kamehameha final como recompensa: si el jugador logra esquivar y recolectar correctamente, debe calcular el momento preciso para lanzar el Kamehameha y poder destruir al robot.
* Nivel con vida: debe terminar el nivel con vida para poder conseguir la esfera del dragón.

**MOMENTO II**

* 1. **Diagrama de clases:** Se encuentra actualizado en el repositorio.
  2. **Detalles no incluidos en el momento I**
     + 1. **Nivel I**
          - **Cámara lógica:** se incorporó un sistema de cámara lógica, que se desplaza con Goku hacia la derecha, generando la ilusión de que el fondo avanza y los obstáculos aparecen desde fuera de la pantalla.
          - **Sistema de barra de vida:** permite representar visualmente cuánta vida le queda a Goku en función de colisiones.
          - **Generación de obstáculos:** las aves, montañas y balas se insertan desde el borde derecho de la pantalla, fuera de la vista del jugador.
          - **Condición de fin de nivel:** se evalúa si Goku alcanza el carrito, en cuyo cado se despliega el diálogo con Pilaf y se pasa al nivel 2.
       2. **Nivel II**
     + **Cámara lógica**: La cámara se mantiene fija. No hay desplazamiento de fondo en este nivel, ya que el escenario tiene un enfoque más estático (esferas de energía que caen desde la parte superior y el robot disparando).
     + **Sistema de barra de vida**: Similar al Nivel I, se mantiene una barra de vida que indica cuánto daño ha recibido Goku. El daño en este nivel se produce cuando el robot lanza bombas y Goku no las esquiva. La barra se actualizará en tiempo real y se indicará con un color (verde para vida completa, amarillo para baja vida, rojo para muy baja vida).
     + **Generación de esferas de energía**: Las esferas de energía caen desde la parte superior de la pantalla en intervalos aleatorios. Cada esfera caerá verticalmente desde una posición aleatoria sobre la pantalla y se moverá hacia abajo. Goku puede moverse para atraparlas, y al recolectar 5 esferas, se activará el **Kamehameha**.
     + **Generación de ataques del robot**: El robot en la esquina inferior izquierda lanza bombas destructivas hacia Goku. Estas bombas tienen diferentes distancias de lanzamiento y pueden variar en velocidad. El robot lanzará las bombas verticalmente en un patrón que Goku debe esquivar saltando o moviéndose a los lados. El jugador debe anticiparse al lanzamiento y evadirlas de manera oportuna.
     + **Condición de fin de nivel**: El nivel termina de dos formas posibles:
       - **Si Goku recolecta las 5 esferas de energía**: Esto activa el Kamehameha, y Goku podrá usarlo para destruir al robot. Al hacerlo, se obtiene la séptima esfera del dragón, indicando que se ha ganado la partida.
       - **Si Goku pierde toda su vida**: Esto ocurre si recibe demasiado daño del robot antes de poder recolectar las 5 esferas. En ese caso, el nivel se reinicia, y el jugador debe comenzar de nuevo con el nivel 1.
     + **Visualización del Kamehameha**: Al reunir las 5 esferas de energía, se activa automáticamente el super poder **Kamehameha**. Este es un ataque único que tiene un área de efecto y elimina al robot.
  3. **Descripción de las vistas, interacciones de cada nivel entre personajes, enemigos y las físicas a utilizar**
  4. **Nivel I:**
     + **Vista superior:** se encuentra la barra de vida, donde se muestra el estado actual de Goku. Además, se tiene el pogedo de la distancia restante para llegar al carro.
     + **Goku (personaje principal):** tiene un movimiento automático hacia la derecha (MRU) y verticalmente se puede controlar con las teclas W y S para subir y bajar, respectivamente. Además, puede colisionar con los obstáculos, lo que le ocasiona perdida de vida.
     + **Obstáculos (aves, balas, montañas):** tiene un movimiento automático hacia la izquierda (MRU), se generan fuera de la pantalla y desaparecen al salir por la izquierda o al solucionar con Goku.
     + **Cámara lógica:** se mueven sincronizadamente con Goku, hacia la derecha, permitiendo una ilusión de desplazamiento continuo del escenario.
     + **Barra de vida:** inicia llena, pero se reduce a medida que Goku colisiona con los obstáculos. Si llega a cero, el jugador pierde.
     + **Nubes:** elemento visual del fondo que no interactúa con Goku ni con los obstáculos. Se mueve hacia la izquierda con un MRU para simular una atmósfera natural con leves corrientes de aire.
     + **Elementos móviles:** 
       - **Interactivos:** Goku y obstáculos
       - **Lógicos:** cámara y barra de vida.
       - **Decorativos:** nubes.
  5. **Nivel II:** 
     + **Vista superior:** En este nivel, la barra de vida de Goku se mantiene visible en la parte superior de la pantalla, así como el temporizador que cuenta el tiempo restante para recolectar las 5 esferas de energía. También se muestra el contador de esferas recolectadas.
     + **Goku (personaje principal):** Goku es controlado completamente por el jugador, quien puede moverlo hacia la izquierda y derecha usando las teclas A y D. También puede saltar con la tecla W para evitar los ataques del robot. El movimiento de Goku es responsivo, lo que permite al jugador esquivar ataques y atrapar las esferas de energía.
     + **Cápsulas de energía:** Las eferas caen desde la parte superior de la pantalla, de manera vertical, en intervalos aleatorios. Goku debe moverse para atraparlas antes de que lleguen al final de la pantalla. Al recolectar las 5 eferas, se activa el Kamehameha para atacar al robot.
     + **Robot gigante (enemigo**): El robot está ubicado en la esquina inferior izquierda de la pantalla. Tiene un movimiento vertical limitado, moviendo sus brazos para lanzar bombas destructivas en diferentes distancias y direcciones hacia Goku. Las bombas caen verticalmente y Goku debe evitarlas o recibir daño. El robot también puede generar una animación visual de "disparo" antes de lanzar las bombas.
     + **Barra de vida:** Similar al Nivel I, la barra de vida se va reduciendo con cada impacto que Goku reciba de las bombas del robot. Si la barra de vida llega a cero, el jugador pierde y debe reiniciar el nivel.
     + **Generación de bombas del robot:** Las bombas caen de manera aleatoria desde los brazos del robot, moviéndose de forma parabolica hacia Goku. La cantidad de bombas y la velocidad con la que caen se incrementan a medida que avanza el nivel. La lógica de colisión detecta si una bomba toca a Goku, causando la reducción de vida.
     + **Condición de fin de nivel:** El nivel termina de dos maneras posibles:
       - **Si Goku recolecta las 5 esferas de energía:** Se activa el Kamehameha, y Goku ataca al robot para destruirlo. Al destruir al robot, se obtiene la séptima esfera del dragón y el jugador avanza al siguiente nivel.
       - **Si Goku pierde toda su vida:** Si Goku recibe demasiado daño de las bombas del robot, el nivel se reinicia, y el jugador debe comenzar de nuevo.
     + **Elementos móviles:** 
       - **Interactivos:** Goku y esferas
       - **Lógicos:** barra de progreso de recolección de esferas y barra de vida.
       - **Decorativos:** nubes.
  6. **Gráfico de las vistas de cada nivel**
  7. **Nivel I:**

**A graph paper with drawings on it

AI-generated content may be incorrect.***Figura 1. Boceto que representa la vista del nivel I*

* 1. **Nivel II:**

A drawing of a person on a graph paper

AI-generated content may be incorrect.

*Figura 2: Boceto que representa el nivel II*

* 1. **Spriters:** en el repositorio se ha agregado la carpeta “Spriters”, en la cual se encuentran los spriters que se usarán para el desarrollo de los dos niveles contenidos en el videojuego.

**Estructura del proyecto:**

El juego esta definido en varias clases que gestionan diferentes aspectos:

* + - **Clase juego** en juego.cpp gestiona el flujo principal del videojuego, controlando la transición entre la pantalla de inicio y los niveles.

Hereda de QMainWindow para aprovechar la estructura de ventana principal que ofrece Qt, incluyendo menús, barra de título y gestión de eventos de cierre.

Los atributos clave incluyen:

* + - * + **ui** (para manipular la interfaz gráfica generada por Qt Designer), scene y view (para renderizar gráficos 2D)
        + **Nivel1/2 nivel1 y nivel2** (instancias de los niveles concretos)
        + **Nivel nivelActual** (puntero para manejar el nivel activo sin distinción de tipo)
        + **QTimer timerFoco** (para asegurar que Goku tenga el foco de teclado tras cambios de nivel)
        + **QLabel exito/transicion** (para mostrar pantallas intermedias).

Juego.cpp actúa como un orquestador que coordina la inicialización, limpieza y transiciones entre estados del juego.

* + - **Clase Nivel** en nivel.cpp es una clase base abstracta que hereda de QWidget para integrarse con el sistema de ventanas y eventos de Qt, permitiendo manejar timers, señales y slots.

Sus atributos incluyen:

* + - * + **vista y escena** (de tipo QGraphicsView\* y QGraphicsScene\*), que permiten renderizar y controlar los elementos gráficos
        + **goku** (de tipo Goku\*), que representa al jugador para adaptarse a diferentes versiones
        + **barraVida y barraProgreso** (de tipo Vida\* y Progreso\*), que muestran el estado del jugador
        + **timerNivel y timerNubes** (de tipo QTimer\*), para actualizar la lógica del nivel y animar elementos como nubes
        + **listaFondos, listaNubes, y listaObstaculos** (usando std::vector<QGraphicsPixmapItem\*> y std::vector<obstaculo\*>), que permiten almacenar múltiples elementos visuales de forma eficiente y acceder a ellos por índice
        + **overlayGameOver**, un QLabel\* para mostrar la pantalla de Game Over.

Se usan contenedores como std::vector porque ofrecen acceso rápido por índice, gestión automática de memoria y compatibilidad con clases Qt.

Los métodos virtuales puros como iniciarNivel(), cargarFondoNivel(), agregarGoku(), actualizarNivel(), y gameOver() obligan a Nivel1 y Nivel2 a implementar su propia lógica específica, asegurando consistencia en la interfaz permitiendo personalización. Esto es clave para mantener un diseño modular, escalable y fácil de extender con nuevos niveles.

* + - **Clase Nivel1** hereda públicamente de la clase abstracta Nivel, lo que le permite especializar la lógica del primer nivel mientras reutiliza la infraestructura gráfica y temporal que ya proporciona la clase base.

Los atributos propios son:

* + - * + **camara (tipo camaraLogica\*),** que controla el seguimiento visual del personaje y es un puntero para poder detener y liberar su memoria de forma segura
        + **carroFinal (tipo Carro\*),** el obstáculo principal que Goku debe alcanzar
        + **Tres Robot\* (r1, r2, r3)** que aparecen tras la caída del carro.
        + **std::vector<obstaculo\*> listaObstaculos;**El método agregarObstaculos(), donde se generan 18 obstáculos de tipos Ave, Montaña o Roca usando un QRandomGenerator, y se insertan en la escena y en listaObstaculos. Luego, en el destructor de Nivel1, se recorre este vector para eliminar cada obstáculo de la escena y liberar memoria, evitando fugas.

Se usan punteros porque estos objetos se crean dinámicamente y deben eliminarse explícitamente en el destructor para evitar fugas de memoria.

Los contenedores std::vector se heredan desde Nivel y se emplean para listaFondos, listaNubes y listaObstaculos; su ventaja es ofrecer acceso aleatorio rápido, iteración eficiente y compatibilidad con la gestión manual de memoria que requiere Qt cuando se eliminan elementos de la escena.

Además, Nivel1 implementa los métodos virtuales puros declarados en Nivel: iniciarNivel() (carga recursos y arranca timers), cargarFondoNivel() (repite el fondo horizontalmente), agregarGoku() (crea al jugador y su HUD), actualizarNivel() (verifica colisiones, vida y progreso), y gameOver() (detiene el nivel y muestra la pantalla de derrota). Esta herencia obliga a Nivel1 a definir su comportamiento específico, garantizando que cualquier nuevo nivel siga la misma interfaz y facilitando la expansión del juego.

* + - **Clase Nivel2** Nivel2 hereda públicamente de la clase abstracta Nivel lo que le permite reutilizar toda la infraestructura gráfica, temporizadores y contenedores que ya maneja la clase base mientras define la lógica propia del segundo nivel. Los atributos propios de Nivel2 son:
      * + **QVector<QPixmap> framesPocion**: se llena en agregarPociones() con los 6 frames de animación de la poción extraídos desde pocion.png. Se usa QVector porque es un contenedor de Qt optimizado para tipos copiables como QPixmap, ofrece iteración rápida y se integra sin conversiones con las clases gráficas de Qt.
        + **QVector<Pocion\*> listaPociones**: se llena dinámicamente en agregarPociones() y agregarPocionAleatoria(), almacenando cada poción creada para luego poder eliminarlas en el destructor con escena->removeItem() y delete. El QVector permite añadir/eliminar elementos de forma segura y acceder por índice cuando se recolectan.
        + **Robot\* robot**: puntero único al robot enemigo del nivel; se crea en agregarRobot() y se libera en el destructor.
        + **Progreso\* barraProgreso**: barra personalizada que muestra el porcentaje de pociones recolectadas; se actualiza mediante pocionRecolectada().
        + **QTimer\* temporizadorPociones**: temporizador propio del nivel que llama al slot agregarPocionAleatoria() para generar nuevas pociones en pantalla.
        + **const int totalPociones = 5**: constante que define cuántas pociones debe recolectar Goku para ganar; se pasa a barraProgreso->setTotalPociones().
        + **Banderas robotInicialCreado, pocionesAgregadas y perdioGoku** para controlar el flujo del nivel sin repetir operaciones.

El uso de QVector (contenedor de Qt) en lugar de std::vector se debe a que almacena objetos y punteros a objetos de Qt (Pocion, QPixmap) y facilita la integración con señales/slots y la gestión automática de eventos.

La herencia desde Nivel obliga a Nivel2 a implementar los métodos virtuales puros:

* + - * + **iniciarNivel():** carga el fondo, HUD, pociones, Goku y el robot.
        + **cargarFondoNivel():** configura un único fondo estático.
        + **agregarGoku():** crea un Goku2 posicionado en el suelo.
        + **actualizarNivel():** verifica progreso, vida y activa la secuencia final.
        + **gameOver():** detiene timers y muestra Game Over.

Estos métodos garantizan que cualquier nivel comparta la misma interfaz y pueda ser manejado desde juego.cpp, facilitando la escalabilidad y mantenimiento del código.

* + - **Clase Goku** es abstracta y sirve como clase base para cualquier versión del personaje Goku; por ello no hereda de ninguna clase de nivel, sino que hereda múltiple de QObject y QGraphicsPixmapItem, combinando la capacidad de emitir señales/timers con la representación gráfica en la escena.

Sus atributos clave son:

* + - * + **scene (QGraphicsScene\*)** para saber en qué escena se dibuja; timerMovimiento y timerDanio (QTimer\*) para gestionar el refresco del movimiento y la inmunidad tras recibir daño
        + **frames (QVector<QPixmap**>) que se llena en las subclases (Goku1, Goku2) mediante cargarImagen() con los sprites de animación
        + **frameActual (int)** para saber qué frame mostrar; velocidad, fotogWidth, fotogHeight para configurar el movimiento y el tamaño visual
        + **mirandoDerecha (bool**) para voltear el Sprite
        + **vidaHUD (Vida\*)** que apunta al widget de barra de vida externo
        + **puedeRecibirDanio (bool)** para evitar daño repetido. Se usa QVector en lugar de std::vector porque almacena objetos Qt (QPixmap) y se integra sin conversiones con las señales/slots y el sistema de eventos.

Los métodos principales son:

* + - * + **setBarraVida**() que asocia la barra externa
        + **recibirDanio()** que reduce la vida y actualiza la barra
        + **obtenerVida()** que consulta la vida actual

Los métodos virtuales puros:

* + - * + **iniciar()**
        + **detener()**
        + **keyPressEvent()**
        + **keyReleaseEvent()**
        + **cargarImagen()**
        + **mover()**

que las subclases deben implementar para definir el comportamiento específico de cada versión de Goku. Esta arquitectura permite que Goku sea una interfaz común reutilizable por Goku1 y Goku2, facilitando la extensión futura sin modificar el núcleo del juego.

* + - **Clase Goku1** es una especialización concreta de la clase abstracta Goku, lo que significa que hereda múltiple de QObject y QGraphicsPixmapItem, al igual que su clase base. Su objetivo es implementar el comportamiento específico del primer nivel: movimiento automático hacia la derecha y control vertical con teclas W/S.

Los atributos clave son:

* + - * + **frames (QVector<QPixmap>):** se llena en cargarImagen() con los 5 frames del sprite sheet GokuSpriter.png. Se usa QVector porque almacena objetos Qt (QPixmap) y permite acceso rápido por índice y compatibilidad total con señales/slots.
        + **timerMovimiento y timerDanio (QTimer\*):** gestionan el movimiento continuo y la inmunidad tras recibir daño.
        + **frameActual, contadorCaminata:** controlan la animación de caminar.
        + **mvtoArriba, mvtoAbajo (bool):** indican si se está presionando W o S.
        + **puedeRecibirDanio, yaRecibioDanio, tocoCarro, tocoObstaculo (bool):** gestionan la lógica de colisiones y daño único.

Los métodos que implementa desde Goku (y que son obligatorios por ser virtuales puros) son:

* + - * + **cargarImagen():** carga los frames desde el sprite sheet.
        + **iniciar(int x, int y):** posiciona a Goku y arranca el timer de movimiento.
        + **mover():** actualiza la posición, detecta colisiones con obstáculos y carro, y cambia el sprite según el estado.
        + **keyPressEvent() y keyReleaseEvent():** capturan W/S para moverse verticalmente.
        + **detener():** detiene el movimiento y resetea el sprite.

Además, añade métodos propios como patadaGokuNivel1() (para la animación de patada) y detectarColision() (para saber contra qué colisionó).

El uso de contenedores como QVector es clave para gestionar animaciones y estados de forma eficiente, permitiendo iterar rápidamente sobre los frames y mantener la sincronización con el sistema gráfico de Qt.

* + - **Clase Goku2** es una implementación de la clase abstracta Goku, por lo tanto hereda múltiple de QObject y QGraphicsPixmapItem, lo que le otorga capacidades de señales/slots y representación gráfica en la escena.

Sus atributos están diseñados para el control total del personaje en el nivel 2:

* + - * + **timerMovimiento, timerSalto, timerDanio, timerMuerte, animSalto, avance, animAtaque,** regreso (todos QTimer\*): gestionan estados como caminar, saltar, recibir daño, morir y ejecutar el ataque Kamehameha.
        + **mvtoIzquierda, mvtoDerecha, enSalto, mirandoDerecha, puedeRecibirDanio (bool):** controlan la dirección, estado de salto y daño.
        + **velocidadVertical, gravedad, sueloY (float):** permiten la física realista del salto.
        + salto (QMediaPlayer\*) y salidaSalto (QAudioOutput\*): reproducen el sonido del salto.

Se usan QVector<QPixmap> para almacenar los frames de animación (por ejemplo, en animarMuerte() y iniciarKamehameha()), ya que permite cargar y acceder eficientemente a los sprites, aprovechando la compatibilidad nativa con Qt y evitando conversiones innecesarias.

Los métodos que implementa desde Goku (por ser virtuales puros) son:

* + - * + **cargarImagen():** carga el sprite inicial.
        + **iniciar(int x, int y):** posiciona a Goku y arranca timers.
        + **mover():** actualiza posición y detecta colisiones.
        + **keyPressEvent() y keyReleaseEvent():** capturan teclas WASD para moverse y saltar.
        + **detener():** detiene todos los timers activos.

Además, añade métodos propios como actualizarSalto(), detectarPocion(), iniciarKamehameha(), animarMuerte(), etc., que permiten interacciones avanzadas con el entorno y enemigos.

Estos métodos heredados aseguran que cualquier versión de Goku comparta la misma interfaz.

* + - **Clase Robot**  En robot.cpp, la clase Robot es un QObject que se relaciona con la escena mediante un puntero QGraphicsScene\* para poder ser dibujado y detectar colisiones.

Se sobrecargan dos constructores:

* + - * + **Robot(QGraphicsScene\*, int, int, QObject\*):** usado en Nivel 1, recibe velocidad y número de robot para cargar sprites específicos desde robots1.png.
        + **Robot(QGraphicsScene\*, QObject\*):** usado en Nivel 2, carga sprites del robot final desde robot.png con escala diferente.

Esto permite reutilizar la misma clase para dos comportamientos distintos sin duplicar código.

Atributos clave y su lógica:

* + - * + **QVector<QPixmap> frames, framesRobot2, framesMuerte:** se llenan en cargarImagen(), cargarRobot2() y cargarFramesMuerte() respectivamente. Se usa QVector porque almacena objetos Qt (QPixmap) y permite acceso rápido por índice, ideal para animaciones por frames.
        + **QList<Explosion\*> ListaExplosiones:** se llena en dispararExplosion() y se limpia en el destructor. Se usa QList porque permite inserción/eliminación rápida y es compatible con iteradores de Qt.
        + **QTimer \*timerMovimiento, timerAnimacion, timerAtaque, timerMuerte:** controlan el movimiento, la animación de sprites, los disparos y la secuencia de muerte.
        + **QGraphicsPixmapItem \*sprite**: representación visual del robot en la escena.
        + **bool desplegado, modoMarcha, usarParabolico, estaMuerto:** gestionan el estado interno del robot (si ya apareció, si está en marcha, si dispara en parábola o recta, y si está muerto).

Métodos principales

* + - * + **iniciar(x, y, xDestino):** inicia posición y timers.
        + **mover(), animar():** actualizan posición y frame visual.
        + **iniciarAtaques(), animarYDisparar(), dispararExplosion():** gestionan el disparo de explosiones con trayectorias alternadas.
        + **murioRobot():** ejecuta la animación de muerte y emite la señal robotMurio() para que el nivel reaccione.

Importancia de los contenedores

* + - * + **QVector para frames:** eficiente para animaciones secuenciales.
        + **QList para explosiones:** permite agregar y eliminar dinámicamente explosiones sin costo alto.

En resumen, Robot encapsula toda la lógica del enemigo, desde su apariencia hasta sus ataques, y se comunica con el nivel mediante señales, manteniendo una separación clara entre lógica del personaje y lógica del nivel.

* + - **Clase Obstaculo** hereda públicamente de QObject para disponer de señales y temporizadores, y mantiene un QGraphicsPixmapItem como miembro público (sprite) para ser renderizado en la escena.

Sus atributos están pensados para describir y actualizar un obstáculo en tiempo real:

* + - * + **QVector<QPixmap> frames:** se llena en cargarImagenes() cuando el tipo es Ave; contiene los 4 frames del sprite sheet pajaro.png. Se usa QVector porque es un contenedor de Qt optimizado para almacenar objetos como QPixmap con acceso rápido por índice.
        + **QTimer \*timerMovimiento y QTimer \*timerAnimacion:** controlan el desplazamiento horizontal y la animación de las alas cuando el tipo es Ave.
        + **QGraphicsPixmapItem \*sprite (público):** permite que otras clases accedan directamente al gráfico sin getters.
        + **QGraphicsScene \*scene (público):** referencia a la escena donde se inserta el obstáculo.
        + **Tipo tipo, int velocidad, int frameActual, etc.:** describen el comportamiento y estado visual del obstáculo.

No hay herencia desde una clase de nivel, porque obstaculo es un objeto autónomo que se instancia dentro de los niveles (como Nivel1 o Nivel2) y se destruye sin depender de la lógica global del nivel.

Los contenedores como QVector se usan para gestionar animaciones eficientemente, permitiendo cambiar de frame rápidamente sin preocuparse por memoria manual.

Los métodos clave son:

* + - * + **cargarImagenes():** llena el vector frames según el tipo (ave, montaña, roca).
        + **iniciar(x, y):** posiciona el obstáculo y arranca los timers si aplica.
        + **mover() y actualizar():** slots conectados a los timers para desplazar y animar el sprite.

Estos métodos son críticos para que los niveles puedan agregar obstáculos dinámicos sin preocuparse por su ciclo de vida interno, manteniendo así una separación clara entre lógica de nivel y lógica de objeto.

* + - **Clase Explosion** hereda de obstaculo, que a su vez extiende QObject y ya incluye lógica básica de temporización y representación gráfica

Sus atributos son:

* + - * + **timerMovimiento y timerAnimacion (ambos QTimer\*):** se crean en lanzar() y controlan la física del desplazamiento y el avance de frames; no se usan contenedores Qt para los timers porque son únicos por explosión.
        + **QVector<QPixmap> frames**: se llena en el constructor a partir de la hoja explosion.png; se emplea QVector porque almacena objetos Qt (QPixmap), permite acceso aleatorio rápido y se integra sin conversiones con el sistema gráfico de Qt para actualizar el sprite en avanzarFrameAnimacion().
        + **float velocidadX, velocidadY, gravedad, tiempo:** almacenan el estado físico del proyectil (parabólico o MRU).
        + **int frameActual:** índice del frame mostrado.
        + **TipoMovimiento tipoMovimiento:** **enum** que decide si la trayectoria es parabólica o rectilínea.
        + **QPointF posicionInicial:** punto desde donde se lanza la explosión.

Los timers y vectores permiten desacoplar la lógica de movimiento y animación de la escena, facilitando la eliminación segura en el destructor (detiene timers y oculta el sprite). Al heredar de obstaculo, Explosion reutiliza el sprite y la escena ya gestionados, evitando duplicar código y asegurando que cualquier nivel pueda instanciar explosiones sin preocuparse por su ciclo de vida.

* + - **Clase Carro** hereda públicamente de obstaculo, que a su vez extiende QObject; por tanto Carro es un QObject indiectamente. Gracias a esta herencia reutiliza el sprite, la escena y los timers base sin duplicar código.

Sus atributos están pensados para la animación y el movimiento especial del coche rojo en el nivel 1:

* + - * + **QPixmap imagenCarro:** hoja completa con 3 frames (0,1,2) que se copian con copy(); se usa QPixmap nativo porque no se requiere contenedor adicional.
        + **int cuadroActual, anchoCuadro, altoCuadro:** definen el rectángulo a extraer en cada paso de la animación.
        + **QTimer \*timerRotacion y QTimer \*timerEspiral:** controlan la rotación y el complejo trayecto parabólico-circular-caída.
        + **QVector<QPixmap>** no se usa porque solo se necesitan tres frames sueltos que se copian directamente; esto evita sobrecarga de memoria.
        + **int fase, float tiempo, QPointF inicio:** gestionan la cinemática del salto-giro-caída.
        + **float g, vx, vy, radio, tiempoGiro, ySuelo:** constantes físicas que definen la trayectoria y permiten ajustar el comportamiento sin recompilar.

Los métodos clave son:

* + - * + **iniciar(int x, int y):** posiciona el coche y prepara timers.
        + **actualizarMovimiento():** slot conectado a timerEspiral; ejecuta la física por fases.
        + **animarRotacion():** slot conectado a timerRotacion; cambia entre los 3 frames para simular rotacion.
        + **iniciarMovimientoEspiral(float \_posXpatada):** recibe la coordenada exacta de Goku cuando patea y lanza la trayectoria.

Su importancia radica en encapsular toda la lógica y animación del coche final, permitiendo que Nivel1 simplemente cree una instancia, la posicione y active su secuencia sin preocuparse por detalles internos.

* + - **Clase Pocion** hereda públicamente de QObject y QGraphicsPixmapItem, lo que le permite emitir señales (aunque no las use ahora) y ser renderizada directamente en la escena.

Sus atributos están pensados para la animación cíclica y reposición aleatoria:

* + - * + **QVector<QPixmap> frames:** se llena en el constructor con los 6 frames del sprite sheet pocion.png, redimensionados al 80 %. Se usa QVector porque es un contenedor de Qt optimizado para objetos copiables como QPixmap, ofrece acceso rápido por índice y se integra sin conversiones con el sistema gráfico de Qt para actualizar el sprite en moverYAnimar().
        + **QTimer \*timer:** controla la animación y el movimiento vertical; se inicia en el constructor y se detiene en detener().
        + **int indiceFrame:** índice actual del frame mostrado.
        + **int fila, columna, columnasTotales:** definen la posición lógica en la imagen para reposicionar la poción aleatoriamente cuando sale de la pantalla.

Los métodos clave son:

* + - * + **moverYAnimar():** slot conectado al timer; desplaza la poción hacia abajo y cambia de frame.
        + **detener():** detiene el timer externamente cuando Goku recolecta la poción.

Su importancia radica en encapsular la lógica de animación y reposición, permitiendo que Nivel2 simplemente cree y gestione instancias sin preocuparse por su comportamiento interno.

* + - **Clase Vida** es un QWidget que no utiliza vectores ni listas, pues su único propósito es mostrar y actualizar la barra de salud del jugador.

Sus atributos son:

* + - * + **QLabel \*texto:** muestra la palabra “HEALTH” con estilo personalizado (fuente, color, sombra).
        + **QProgressBar \*barra:** widget nativo que representa visualmente la vida restante; se le fija rango 0-100 y se actualiza dinámicamente.
        + **const int vidaMaxima = 100:** límite superior constante.
        + **int vidaActual:** valor mutable que se reduce con restar() y se consulta con obtenerVida().

No se emplean contenedores porque solo se maneja un único valor entero; la clase se limita a reflejar el estado de vida y actualizar la interfaz gráfica mediante actualizarColor(), que cambia el color de la barra según el porcentaje (verde, amarillo, rojo).

Vida no hereda métodos de ninguna clase de nivel; su importancia es aíslar la lógica para que cualquier nivel (o incluso menú) pueda instanciarla y conectarla al personaje mediante setBarraVida() sin preocuparse por su implementación interna.

* + - **Clase Progreso** es un QWidget que no utiliza vectores ni listas internos, porque su misión es muy concreta: dibujar una barra progresiva con un icono que muestre el avance del jugador. Sus atributos son:
      * + **TipoProgreso tipo (enum):** distingue si la barra refleja avance horizontal (Nivel 1) o pociones recolectadas (Nivel 2).
        + **float porcentaje:** valor 0-1 que indica cuánto se ha completado; se actualiza con actualizarProgreso() o sumarPocion().
        + **QLabel \*icono:** muestra la imagen de referencia (coche o poción) redimensionada a 35×25 px mediante scaled(); se posiciona al inicio del widget.
        + **int totalPociones y int pocionesRecolectadas:** solo se usan cuando tipo == Pociones; permiten calcular el porcentaje sin contenedores adicionales.

No se recurre a contenedores porque solo se manejan dos enteros y un float; el propio widget se pinta con QPainter, dibujando un QProgressBar simulado con fondo gris y relleno azul/redondeado.

Su valor radica en abstraer la visualización de progreso, de modo que cualquier nivel puede instanciarla, configurar el tipo y actualizar el valor sin preocuparse por detalles gráficos.

* + - **Clase Camara Logica** hereda públicamente de QObject para aprovechar el sistema de señales y slots de Qt; no usa vectores ni listas, porque su única misión es seguir al personaje Goku con una cámara suavizada.

Sus atributos son:

* + - * + **QGraphicsView \*view:** puntero a la vista donde se aplica el centrado; se recibe en el constructor y se usa para calcular desplazamientos.
        + **QTimer \*timer:** temporizador que genera el “tick” cada 16 ms (~60 FPS) y está conectado al slot moverVista().
        + **Goku \*objetivo:** puntero al personaje que se desea seguir; permite leer su posición en cada frame.
        + **static int contador:** variable de clase usada solo para depurar cuántas veces se ha ejecutado el timer.

No se emplean contenedores porque solo se manejan dos punteros y un temporizador; la lógica para el Nivel1 se limita a calcular el desplazamiento horizontal para que Goku aparezca siempre a la izquierda de la pantalla, sumando la mitad del ancho del viewport.

Su importancia es desacoplar el seguimiento de cámara del resto del juego, permitiendo que cualquier nivel (Nivel1 o Nivel2) simplemente cree la instancia, llame a seguirAGoku(goku) y active iniciarMovimiento() sin preocuparse por la matemática de la cámara.

**Relación entre clases:**

A continuación, se presenta un análisis por secciones que explica la relación entre las clases del proyecto La aventura de Goku, basado en el diagrama de clases que se encuentra en el repositorio y la implementación entregada. Cada sección describe cómo se relaciona una clase con otra y la justificación de la decisión de diseño.

1. **Relación entre Juego y Nivel (Nivel1, Nivel2)**

Relación: Pertenencia

Justificación: La clase Juego actúa como controlador principal y contiene instancias de Nivel1 o Nivel2 según la fase del juego. Estos niveles se crean mediante cambiarNivel(int) y se destruyen en cerrarNivel(bool), lo que refleja una relación de propiedad: cuando Juego se destruye, también se liberan los niveles activos. Esto permite aislar la lógica de cada nivel sin acoplarla al resto del sistema.

1. **Relación entre Nivel y Goku (Goku1, Goku2)**

Relación: Pertenencia

Justificación: Cada nivel contiene una instancia concreta de Goku (ya sea Goku1 o Goku2), pero interactúa con ella a través de la clase abstracta Goku. Esto permite que los niveles compartan código común mientras permiten comportamientos específicos por nivel. Por ejemplo, Goku1 se mueve automáticamente, mientras que Goku2 permite control manual con salto.

1. **Relación entre Goku y Vida**

Relación: Asociación

Justificación: Goku utiliza una instancia de Vida para reflejar su estado de salud. Esta relación se establece mediante setBarraVida(Vida\*), lo que permite que cambios en la vida del personaje se actualicen automáticamente en la interfaz. Esta separación facilita que la barra de vida sea reusable en otros contextos (por ejemplo, enemigos u objetos de decoración (NPC)).

1. **Relación entre Nivel y Obstáculo, Explosion/Carro**

Relación: Pertenencia

Justificación: Los niveles contienen múltiples instancias de estos elementos, pero no los poseen de forma exclusiva. Por ejemplo, Nivel1 crea y gestiona un Carro y varios Obstáculo por lo tanto a este nivel le pertenecen estos objetos, mientras que Nivel2 gestiona Explosión. Estos objetos se destruyen junto con el nivel y su ciclo de vida está gestionado por el nivel, lo que permite una limpieza ordenada.

1. **Relación entre Robot y Explosión**

Relación: Pertenencia

Justificación: El Robot crea instancias de Explosión durante sus ataques. Estas explosiones se almacenan en una lista interna (ListaExplosiones) y se liberan cuando el robot se destruye. Esto evita fugas de memoria y asegura que las explosiones se liberen junto con el robot.

1. **Relación entre Nivel1 y camaraLogica**

Relación: Pertenencia

Justificación: Nivel1 utiliza una instancia de camaraLogica para seguir al personaje. La cámara se crea y destruye dentro del nivel y su ciclo de vida está ligado al nivel. Esto permite que la cámara se ajuste automáticamente al tamaño de la escena sin afectar otras partes del juego.

1. **Relación entre Poción y Nivel2**

Relación: Pertenencia

Justificación: Nivel2 crea y gestiona instancias de Poción, estas notifican al nivel cuando son recolectadas mediante pocionRecolectada(). Esto permite que el nivel actualice su barra de progreso sin que Poción tenga una referencia directa con progreso.

1. **Relación entre Progreso y Nivel**

Relación: Pertenencia

Justificación: Nivel1 y Nivel2 utilizan instancias de Progreso para mostrar el avance del jugador. Progreso es un widget independiente que recibe actualizaciones del nivel (por ejemplo, posición de Goku en Nivel1 o pociones recolectadas en Nivel2), lo que permite reutilizar la misma clase para diferentes tipos de progreso.

1. **Relación entre Vida y Nivel**

Relación: Pertenencia

Justificación: Cada nivel (Nivel1 o Nivel2) crea y gestiona una instancia de Vida como parte de su HUD (interfaz del jugador). Esta barra de vida se posiciona sobre la vista del nivel (QGraphicsView) y se destruye automáticamente cuando el nivel finaliza. No es compartida entre niveles, lo que refuerza que es propiedad exclusiva del nivel actual.

1. **Relación entre Robot y Nivel1**

Relación: Pertenencia

Justificación: Independientemente de que se creen en tiempo de ejecución hay una relación de pertenencia con respecto al Nivel1, se crean 3 instancias de robot como parte del evento final del nivel.

**Conclusión**

El diseño del juego prioriza la separación de responsabilidades. Las relaciones de pertenencia son para elementos que “dependen” del otro (como Juego → Nivel), mientras que asociaciones permiten flexibilidad en la gestión de elementos secundarios (obstáculos, enemigos). Esto facilita la extensibilidad: agregar un nuevo nivel solo requiere crear una subclase de Nivel y definir sus elementos específicos.

**Herencia dentro de clases propias:**

1. **Goku1 y Goku2 heredan de Goku**

* ¿Qué es?

Goku es una clase abstracta que define la estructura común del personaje principal.

* ¿Por qué se hizo así?

Evita duplicar lógica como: vida, animación, colisiones, conexión con Vida, etc.

Permite que cada nivel tenga un comportamiento distinto sin romper el resto del sistema.

* Utilidad presente:

Goku1 tiene movimiento automático y solo se mueve verticalmente.

Goku2 tiene control manual con salto y física.

* Utilidad futura:

Si se añade un nivel 3 con Goku volador o nadador, solo hay que crear Goku3 heredando de Goku.

1. **Nivel1 y Nivel2 heredan de Nivel**

* ¿Qué es?

Nivel es una clase abstracta que define la estructura base de un nivel: escena, vista, HUD, timers, fondo, nubes, etc.

* ¿Por qué se hizo así?

Centraliza la gestión de recursos (escena, vista, timers).

Obliga a cada nivel a implementar métodos clave: iniciarNivel(), actualizarNivel(), gameOver(), etc.

* Utilidad presente:

Nivel1 y Nivel2 comparten código como: generar nubes, mostrar Game Over, limpiar escena.

* Utilidad futura:

Para agregar un Nivel3 (por ejemplo, un nivel espacial), solo se crea una nueva clase que herede de Nivel y se implementan sus métodos.

1. **Explosion hereda de Obstaculo**

* ¿Qué es?

Obstaculo es una clase base que representa cualquier elemento que puede dañar o interactuar con Goku.

* ¿Por qué se hizo así?

Explosion reutiliza la estructura de Obstaculo: sprite, movimiento, colisiones.

Permite que Explosion sea tratada como un obstáculo más en la escena, sin código extra.

* Utilidad presente:

Explosion se detecta como un obstáculo en colisiones y puede dañar a Goku.

* Utilidad futura:

Se puede crear ExplosionCongelante o ExplosionElectrica que hereden de Obstaculo o incluso de Explosion.

1. **Carro hereda de Obstaculo**

* ¿Qué es?

Carro es una clase derivada que hereda de Obstaculo, una clase base que representa elementos interactivos en la escena del juego.

* ¿Por qué se hizo así?

Obstaculo ya maneja sprite, timer, movimiento básico y colisiones, lo que evita duplicar código.Carro necesita mostrarse en pantalla, moverse, y ser detectado por Goku, al igual que cualquier otro obstáculo.

* Utilidad presente:

Carro reutiliza el sistema de carga de sprites, temporizadores y detección de colisiones de Obstaculo. Se identifica fácilmente en la escena como un obstáculo especial (sprite->setData(0, "carro")). Permite que Goku1 lo detecte y active eventos específicos (como la patada y el inicio de la animación de mvto fisico).

* Utilidad futura:

Permite crear nuevos vehículos para implementar en nuevos niveles o perfeccionar los que ya existen.

Facilita la expansión del juego con niveles nuevos que incluyan distintos tipos de obstáculos móviles o interactivos.

**Herencias de Qt**

1. **Juego hereda de QMainWindow**

* ¿Qué es?

Ventana principal de la aplicación.

* ¿Por qué se hizo así?

Para aprovechar la infraestructura de Qt: menú, barra de estado, gestión de eventos de ventana.

* Utilidad presente:

Muestra la pantalla de inicio.

Maneja cambiar de nivel sin perder estado.

* Utilidad futura:

Se puede agregar menú de opciones, configuración, o modo multijugador sin rediseñar la ventana.

1. **Vida hereda de QWidget**

* ¿Qué es?

Widget personalizado que muestra la barra de vida.

* ¿Por qué se hizo así?

Para integrarse al sistema de widgets de Qt y poder ser posicionado, redimensionado y estilizado fácilmente.

* Utilidad presente:

Se muestra en la esquina superior izquierda de la vista.

Se actualiza automáticamente al recibir daño.

* Utilidad futura:

Se puede reutilizar para mostrar vida de enemigos, jefes, o jugador 2.

1. **Progreso hereda de QWidget**

* ¿Qué es?

Widget personalizado que muestra una barra de progreso con ícono.

* ¿Por qué se hizo así?

Para tener control total sobre el diseño (color, forma, texto, ícono) sin depender del QProgressBar estándar.

* Utilidad presente:

Muestra progreso hacia el carro en Nivel1.

Muestra porcentaje de pociones recolectadas en Nivel2.

* Utilidad futura:

Se puede usar para mostrar tiempo restante, misión secundaria, o nivel de energía.

1. **Nivel hereda de QWidget**

* ¿Qué es?

Clase base abstracta para todos los niveles del juego.

* ¿Por qué se hizo así?

Para integrarse al sistema de widgets de Qt y poder ser gestionada como parte de la interfaz.

* Utilidad presente:

Permite que cada nivel sea un widget que se puede habilitar o deshabilitar.

* Utilidad futura:

Facilita agregar niveles como pestañas o ventanas.

1. **Goku hereda de QObject y QGraphicsPixmapItem**

* ¿Qué es?

Clase base abstracta para el personaje principal.

* ¿Por qué se hizo así?

QObject → Para usar señales, slots y timers.

QGraphicsPixmapItem → permite que Goku aparezca en pantalla sin escribir código de dibujo, y que Qt lo mueva, anime y detecte colisiones automáticamente.

* Utilidad presente:

Permite animar a Goku, detectar colisiones y actualizar su vida.

* Utilidad futura:

Facilita agregar nuevas clases de Goku dependiendo el comportamiento que se quiera tener en un nuevo nivel con la misma estructura.

1. **Obstaculo hereda de QObject**

* ¿Qué es?

Clase base para elementos que interactúan con Goku.

* ¿Por qué se hizo así?

Para reutilizar lógica de sprite, timer y colisión. Además contiene atributos de QGraphicsPixmapItem ya que esto permite dibujar, pero no fue necesario heredar de esta ya que los métodos que se iban a implementar no necesariamente se extraían de esta clase de Qt.

* Utilidad presente:

Rocas, aves y montañas usan la misma base.

* Utilidad futura:

Se puede agregar nuevas instancias de obstáculos dependiendo del nivel que se quiera realizar.

1. **Robot hereda de QObject y QGraphicsPixmapItem**

* ¿Qué es?

Enemigo final del Nivel 1 y Nivel 2.

* ¿Por qué se hizo así?

Para poder usar animaciones, temporizadores y aparezca en pantalla sin escribir código de dibujo, y que Qt lo mueva, anime y detecte colisiones automáticamente.

* Utilidad presente:

Se mueve, dispara, muere.

Se detecta como enemigo.

* Utilidad futura:

Se puede crear RobotVolador o RobotBoxeador, etc.

1. **Pocion hereda de QObject y QGraphicsPixmapItem**

* ¿Qué es?

Objeto coleccionable del Nivel 2.

* ¿Por qué se hizo así?

Para usar animaciones, temporizadores y aparezca en pantalla sin escribir código de dibujo, y que Qt lo mueva, anime y detecte colisiones automáticamente.

* Utilidad presente:

Se mueve, anima, y se puede recolectar.

* Utilidad futura:

Se puede crear PocionRoja, PocionAzul, etc., con efectos distintos.

1. **camaraLogica hereda de QObject**

* ¿Qué es?

Gestiona el seguimiento de la cámara en Nivel1.

* ¿Por qué se hizo así?

Para usar QTimer y señales sin necesidad de ser un widget.

* Utilidad presente:

Mueve la vista automáticamente siguiendo a Goku.

* Utilidad futura:

Se puede reutilizar en Nivel3 con seguimiento diagonal o zoom dinámico.

**Flujo del Jugador**

* **Caso 1:** Nivel1 y Nivel2 completados exitosamente

*Pantalla principal -> Nivel1 -> Transición -> Nivel2 -> Éxito -> Pantalla principal*

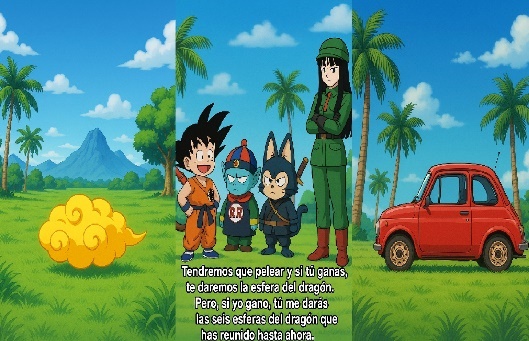
**

**

**

**

**

**

* **Caso2:** Goku perdió toda la vida en nivel 1

*Pantalla principal -> Nivel1 -> gameOver -> Pantalla principal*

**

**

**



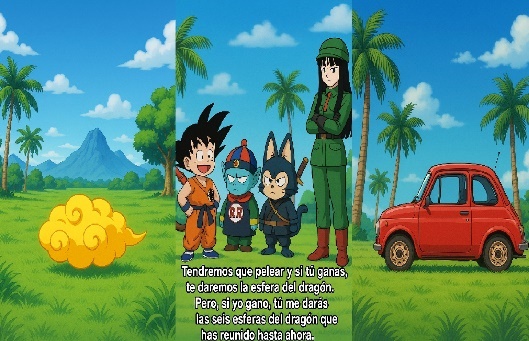
* **Caso3:** Goku ganó Nivel1 / Goku perdió Nivel2

*Pantalla principal -> Nivel1 -> Transición -> Nivel2 -> gameOver -> Pantalla principal*

**

**

**

**

**

**Elección de métodos y atributos privados o públicos según la clase**

* **Clase Juego**

***Atributos privados:***

Ui::juego \*ui, QGraphicsScene \*scene, QGraphicsView \*view, Nivel1 \*nivel1, Nivel2 \*nivel2, Nivel \*nivelActual, QLabel \*exito, QTimer \*timerFoco, QLabel \*transición, static int contador.

*¿Por qué son privados?*

Porque son detalles internos de la implementación del juego. No deben ser accedidos ni modificados directamente desde fuera de la clase. Esto garantiza encapsulamiento y evita dependencias externas. Por ejemplo, scene y view son punteros internos que solo deben ser gestionados por la clase juego.

*¿Para qué sirven?*

* + **ui:** Permite interactuar con la interfaz gráfica generada por Qt Designer.
  + **scene y view:** Controlan la escena y vista del juego.
  + **nivel1, nivel2, nivelActual:** Gestiona los niveles del juego.
  + **exito, transicion:** Muestran pantallas de transición o victoria.
  + **timerFoco:** Asegura que Goku reciba el foco al cambiar de nivel.
  + **contador:** Variable estática para depuración.

***Atributos públicos:***

No hay atributos públicos explícitos en esta clase.

*¿Por qué no hay atributos públicos?*

Porque la clase juego actúa como un controlador principal que expone su funcionalidad solo a través de métodos y señales, siguiendo el principio de encapsulamiento. Esto evita modificaciones accidentales del estado interno.

***Slots privados:***

void iniciarJuego(), void regresarAlMenuTrasDerrota(), void mostrarTransicion(), void mostrarExito().

*¿Por qué son privados?*

Porque solo deben ser invocados por señales internas de la clase (como botonIniciar->clicked() o QTimer::singleShot). No necesitan ser accesibles desde fuera.

*¿Para qué sirven?*

* + iniciarJuego(): Inicia el primer nivel al presionar el botón.
  + regresarAlMenuTrasDerrota(): Cierra el nivel y vuelve al menú tras derrota.
  + mostrarTransicion(): Muestra pantalla de transición entre niveles.
  + mostrarExito(): Muestra pantalla de victoria al completar el nivel 2.

***Métodos privados:***

void cambiarNivel(int numero), void mostrarPantallaInicio(), void cerrarNivel(bool mostrarMenu).

*¿Por qué son privados?*

Porque son operaciones internas que no deben ser invocadas directamente desde fuera. Por ejemplo, cerrarNivel() limpia recursos y debe ser llamado solo desde métodos internos como closeEvent().

*¿Para qué sirven?*

* cambiarNivel(): Gestiona la transición entre niveles (1 o 2).
* mostrarPantallaInicio(): Restaura la pantalla de bienvenida.
* cerrarNivel(): Libera memoria y detiene timers al salir de un nivel.

***Métodos públicos:***

Constructor explicit juego(QWidget \*parent = nullptr), destructor ~juego(), void closeEvent(QCloseEvent \*event).

*¿Por qué son públicos?*

Porque son la interfaz principal para interactuar con la clase juego. El constructor inicializa la ventana, y el destructor libera recursos. closeEvent() es virtual protegida (heredada de QMainWindow) y se sobrescribe aquí para manejar el cierre seguro.

***Señales:***

No hay señales definidas en juego, pero conecta señales de otros objetos (como Nivel1::gokuMurio).

*¿Por qué no hay señales en juego?*

Porque juego actúa como un controlador principal que escucha señales de otros objetos (Nivel1, Nivel2) y ejecuta acciones, pero no necesita emitir señales externas.

***Contenedores usados:***

No se usan contenedores STL/QVector/QList en juego directamente, pero gestiona punteros a objetos (Nivel1, Nivel2, QGraphicsScene).

*¿Por qué no se usan contenedores?*

Porque la clase solo gestiona un nivel activo a la vez (nivelActual), y los objetos se crean/destruyen dinámicamente. Esto evita overhead de contenedores y simplifica la gestión de memoria con delete.

* **Clase Nivel**

***Atributos privados:***

Ninguno explícito, porque la clase no declara miembros privados.

*¿Por qué no hay atributos privados?*

Porque Nivel es una clase base abstracta que define la interfaz común para todos los niveles. Sus atributos son protegidos para permitir que las subclases (Nivel1, Nivel2) los accedan directamente sin necesidad de getters/setters, simplificando la herencia.

***Atributos públicos:***

static int contNubes, static int contador.

*¿Por qué son públicos?*

Porque son variables compartidas entre todos los niveles (estáticas) y no representan el estado interno de una instancia específica. Se usan para depuración, como contar nubes activas en toda la aplicación.

***Métodos públicos***

Constructor Nivel(...), destructor ~Nivel(), int getMargenHUD() const.

*¿Por qué son públicos?*

Constructor/destructor: Necesarios para crear y destruir niveles desde juego.

getMargenHUD(): Proporciona información constante (espacio para HUD) que puede consultar cualquier clase sin acceder al estado interno.

***Métodos protegidos:***

void generarNubes(), void moverNubes(), void mostrarGameOver()

¿Por qué son protegidos?

Porque son utilidades compartidas que pueden ser usadas o sobrescritas por subclases. Por ejemplo:

Nivel1 y Nivel2 usan generarNubes() sin reimplementarlo. Además, permite que subclases añadan limpieza específica si sobrescriben ~Nivel().

***Métodos virtuales***

virtual void iniciarNivel() = 0

virtual void cargarFondoNivel(const QString& ruta) = 0

virtual void agregarGoku() = 0

virtual void actualizarNivel() = 0

virtual void gameOver() = 0

virtual Goku\* getGoku() const = 0

virtual bool haTerminado() const = 0

virtual void limpiarEscena()

*¿Por qué son virtuales puras?*

Porque obligan a las subclases a implementar la lógica específica de cada nivel (ej. Nivel1 carga un fondo más amplio, Nivel2 carga uno estático). Esto garantiza que cada nivel define su propio comportamiento sin duplicar código común.

***Señales***

void gokuMurio()

void nivelCompletado()

*¿Por qué son públicas?*

Porque son la única forma que tiene Nivel de comunicar eventos externos (como juego) sin exponer su estado interno.

gokuMurio: Emitida cuando Goku pierde toda la vida.

nivelCompletado: Emitida cuando se cumplen los objetivos (p.ej. cuando salen los robots en Nivel1).

***Contenedores usados:***

std::vector<QGraphicsPixmapItem\*> listaFondos

std::vector<QGraphicsPixmapItem\*> listaNubes

std::vector<obstaculo\*> listaObstaculos

*¿Por qué se usan std::vector en lugar de QVector o QList?*

Rendimiento: std::vector es más eficiente para iteraciones frecuentes (como mover nubes).

Compatibilidad con C++ estándar: Evita dependencias de Qt en lógica interna.

Gestión manual de memoria: Los punteros se eliminan explícitamente en el destructor para evitar fugas.

*¿Para qué sirven?*

listaFondos: Almacena fondos repetidos horizontalmente (Nivel1).

listaNubes: Gestiona nubes móviles para reutilizarlas al salir de pantalla.

listaObstaculos: Permite eliminar obstáculos dinámicamente al salir de pantalla.

*¿Por qué es mejor esta opción?*

Simplicidad: No necesita señales/slots de Qt en estos contenedores.

Control total: Se puede iterar y eliminar elementos sin restricciones de Qt.

* **Clase Nivel1**

***Atributos privados:***

camaraLogica\* camara

Carro\* carroFinal

Robot\* r1, \*r2, \*r3

bool gokuYaPateo

bool robotsCreados

bool nivelTerminado

bool perdioGoku

bool gameOverProcesado

*¿Por qué son privados?*

Porque representan el estado interno exclusivo del nivel 1 y no deben ser accedidos ni modificados desde fuera. Esto garantiza que la lógica (como cuándo aparecen los robots o cuándo termina el nivel) esté centralizada y protegida contra interferencias externas.

***Atributos públicos:***

static int contador (heredado de Nivel, público por ser estático)

*¿Por qué es público?*

Porque es una variable compartida entre instancias y útil para depuración o estadísticas globales. No afecta el estado interno de un nivel específico.

***Slots privados:***

void agregarRobots() (definido como slot privado en nivel1.cpp)

*¿Por qué es privado?*

Porque solo debe ser invocado internamente por actualizarNivel() cuando el carro aterriza. No es necesario exponerlo externamente.

***Métodos públicos***

Constructor Nivel1(...)

Destructor ~Nivel1()

void iniciarNivel() (heredado de Nivel)

void cargarFondoNivel(const QString& ruta) (heredado de Nivel)

void agregarGoku() (heredado de Nivel)

void actualizarNivel() (heredado de Nivel)

void gameOver() (heredado de Nivel)

bool haTerminado() const (heredado de Nivel)

Goku\* getGoku() const (heredado de Nivel)

bool getPerdioGoku() const (método propio)

void quitarCarroVista() (método propio)

void agregarCarroFinal() (método propio)

void agregarObstaculos() (método propio)

*¿Por qué son públicos?*

Porque son la interfaz necesaria para que juego pueda:

Crear/destruir el nivel.

Inicializarlo (iniciarNivel()).

Consultar su estado (getGoku(), haTerminado(), getPerdioGoku()).

Implementar comportamientos específicos (fondos, Goku, etc.).

***Señales:***

gokuMurio() (heredada de Nivel, pública)

nivelCompletado() (heredada de Nivel, pública)

*¿Por qué son públicas?*

Porque deben ser emitidas hacia juego para notificar eventos críticos (derrota o victoria). No pueden ser privadas porque juego necesita conectarse a ellas.

***Contenedores usados:***

std::vector<obstaculo\*> listaObstaculos (heredado de Nivel)

*¿Por qué se usa std::vector?*

Rendimiento: Iteración eficiente al mover obstáculos.

Control manual: Se eliminan obstáculos al salir de pantalla con delete.

Flexibilidad: Permite añadir dinámicamente nuevos obstáculos sin restricciones de Qt.

*¿Alternativas descartadas?*

QVector: Innecesario, ya que no se usan señales/slots con los obstáculos.

QList: Potencialmente más lento para iteraciones frecuentes.

***¿Por qué esta implementación es mejor?***

Encapsulación estricta: Estado interno (gokuYaPateo, robotsCreados) es privado para evitar errores.

Herencia eficiente: Sobrescribe solo los métodos virtuales necesarios sin duplicar lógica.

Señales desacopladas: juego reacciona a eventos sin conocer detalles internos.

Gestión manual de memoria: Elimina objetos (carro, robots) en el destructor para evitar fugas.

Contenedores ligeros: std::vector optimiza el rendimiento al mover obstáculos.

* **Clase Nivel2**

***Atributos privados:***

Robot\* robot

Progreso\* barraProgreso

QVector<QPixmap> framesPocion

QVector<Pocion\*> listaPociones

QTimer\* temporizadorPociones

bool robotInicialCreado, bool pocionesAgregadas, bool perdioGoku

const int totalPociones = 5

*¿Por qué son privados?*

Porque son detalles internos del nivel 2 que no deben ser accedidos desde fuera. Esto asegura que la lógica de creación del robot, generación de pociones o control de progreso esté centralizada y protegida. Por ejemplo, si listaPociones fuera pública, se podría manipular desde otra clase y romper el flujo del juego.

***Atributos públicos:***

static int contador (heredado de Nivel, público por ser estático)

*¿Por qué es público?*

Porque es una variable compartida entre instancias y sirve para depuración o estadísticas globales. No afecta el estado interno del nivel 2 específicamente.

***Slots privados:***

void agregarPocionAleatoria()

*¿Por qué es privado?*

Porque solo debe ser invocado internamente por temporizadorPociones. No es necesario exponerlo externamente, ya que su única función es generar nuevas pociones.

***Métodos privados***

void agregarPociones()

void agregarRobot()

*¿Por qué son privados?*

Son tareas internas específicas del nivel 2 que no deben ser llamadas desde fuera. Por ejemplo, agregarRobot() posiciona y escala al robot enemigo, y solo se ejecuta una vez durante la inicialización.

***Métodos públicos:***

Constructor Nivel2(...)

Destructor ~Nivel2()

void iniciarNivel() (heredado de Nivel)

void cargarFondoNivel(const QString& ruta) (heredado de Nivel)

void agregarGoku() (heredado de Nivel)

void actualizarNivel() (heredado de Nivel)

void gameOver() (heredado de Nivel)

bool haTerminado() const (heredado de Nivel)

Goku\* getGoku() const (heredado de Nivel)

void pocionRecolectada() (método propio)

*¿Por qué son públicos?*

Porque son la interfaz necesaria para que juego pueda:

Crear/destruir el nivel.

Inicializarlo (iniciarNivel()).

Consultar su estado (getGoku(), haTerminado()).

Notificar eventos (pocionRecolectada() cuando Goku recoge una poción).

***Señales:***

gokuMurio() (heredada de Nivel, pública)

nivelCompletado() (heredada de Nivel, pública)

*¿Por qué son públicas?*

Porque deben ser emitidas hacia juego para notificar eventos críticos (derrota por pérdida de vida o victoria al recolectar todas las pociones y derrotar al robot).

***Contenedores usados:***

QVector<QPixmap> framesPocion

QVector<Pocion\*> listaPociones

*¿Por qué se usan QVector?*

Rendimiento: QVector es optimizado para acceso rápido por índice (útil al animar frames).

Integración con Qt: Se usa con objetos gráficos (QGraphicsPixmapItem), y QVector gestiona mejor la memoria de punteros Qt.

Flexibilidad: Permite añadir/eliminar pociones dinámicamente sin complejidad.

*¿Alternativas descartadas?*

std::vector: Menos integrado con tipos Qt (como QPixmap).

QList: Menos eficiente para acceso secuencial frecuente.

***¿Por qué esta implementación es mejor?***

Encapsulación estricta: Estado interno (listaPociones, robot) es privado para evitar errores.

Gestión automática de memoria: QVector y QTimer con parentesco Qt previenen fugas.

Desacoplamiento con señales: juego reacciona a eventos sin conocer detalles internos.

Especialización clara: Sobrescribe solo métodos virtuales necesarios sin duplicar lógica.

Contenedores eficientes: QVector optimiza animaciones y recolección de pociones.

* **Clase Goku**

***Atributos privados:***

Ninguno

*¿Por qué no hay atributos privados?*

Porque Goku es una clase abstracta que define la interfaz común; todos sus atributos están en la sección protegida para que las subclases (Goku1, Goku2) puedan accederlos directamente sin getters ni setters, simplificando la herencia.

***Atributos protegidos:***

QGraphicsScene \*scene

QTimer \*timerMovimiento, timerDanio

QVector<QPixmap> frames (contenedor para frames de animación)

int frameActual, velocidad, fotogWidth, fotogHeight

bool mirandoDerecha

Vida\* vidaHUD

bool puedeRecibirDanio

¿Por qué están protegidos?

Porque son recursos compartidos que las subclases necesitan leer y modificar directamente sin romper el encapsulamiento hacia clases externas.

* scene: permite añadir Goku a la escena desde la subclase.
* timerMovimiento / timerDanio: controlan animaciones.
* frames + frameActual: almacenan y gestionan la animación del sprite.
* vidaHUD: conecta la lógica de vida con la interfaz gráfica.
* puedeRecibirDanio: usado por subclases para gestionar inmunidad temporal tras recibir daño.

***Atributos públicos:***

static int contador

*¿Por qué es público?*

Porque es una variable estática compartida entre todas las instancias de Goku y sirve para depuración. No representa el estado interno de una instancia específica.

***Métodos protegidos:***

void actualizarFrame(int indice) (método protegido concreto)

virtual void keyPressEvent(QKeyEvent \*event) = 0 (virtual puro)

virtual void cargarImagen() = 0 (virtual puro)

virtual void mover() = 0 (virtual puro)

*¿Por qué están protegidos?*

actualizarFrame: permite a subclases cambiar sprites sin repetir código.

Métodos virtuales puros: obligan a cada subclase a definir su propio comportamiento de entrada, animación y movimiento (ej. Goku1 usa W/S; Goku2 usa A/D/W).

***Métodos públicos:***

Constructor Goku(...)

Destructor ~Goku()

void setBarraVida(Vida\* barra)

void recibirDanio(int cantidad)

int obtenerVida() const

void iniciar(int, int) (método virtual)

void detener() (método virtual)

*¿Por qué son públicos?*

Forman la interfaz mínima y segura que cualquier instancia de Goku debe ofrecer:

Asociar la barra de vida desde el nivel.

Aplicar daño y consultar vida desde otros objetos (explosiones, obstáculos) sin exponer detalles internos.

***Señales:***

Ninguna en la clase base.

¿Por qué no hay?

Goku delega la emisión de eventos a las subclases o al HUD (Vida). Por ejemplo, cuando la vida llega a 0, la barra de vida o la subclase emitirán señales (gokuMurio desde Nivel), no Goku directamente. Esto desacopla la lógica del personaje de la lógica de nivel.

***Contenedores usados:***

QVector<QPixmap> frames (protegido)

*¿Por qué se usa QVector?*

Integración Qt: compatible con QPixmap y QGraphicsPixmapItem.

Rendimiento: acceso rápido por índice para animaciones.

Simplicidad: no necesita señales/slots; solo almacena frames.

*¿Alternativas descartadas?*

std::vector: menos integrado con tipos Qt.

QList: más lento para acceso aleatorio frecuente.

***¿Por qué esta implementación es mejor?***

Encapsulación flexible: atributos protegidos permiten acceso directo a subclases sin romper encapsulamiento externo.

Interfaz mínima y segura: solo expone métodos esenciales (recibirDanio, obtenerVida).

Herencia eficiente: métodos virtuales puros obligan a subclases a definir comportamiento sin repetir código base.

Contenedores adecuados: QVector ofrece rendimiento y compatibilidad óptima para animaciones.

* **Clase Goku1**

***Atributos privados:***

QVector<QPixmap> frames (copia local de los frames de Goku)

QTimer\* timerMovimiento

QTimer\* timerDanio

int frameActual, contadorCaminata

bool mvtoArriba, mvtoAbajo, yaRecibioDanio, puedeRecibirDanio, tocoCarro, tocoObstaculo

*¿Por qué son privados?*

Porque representan el estado interno y exclusivo de Goku1 y no deben ser accedidos ni modificados desde fuera. Esto previene errores y mantiene encapsulado el comportamiento de entrada, animación y daño del personaje.

***Atributos públicos:***

static int contador

*¿Por qué es público?*

Porque es una variable estática compartida entre instancias y útil para depuración. No expone el estado interno de un Goku1 específico.

***Métodos slots privados:***

void actualizarFrame(int i) (usado internamente para cambiar sprite)

void mientrasTocaObstaculo() (actualiza sprite al recibir daño)

*¿Por qué son privados?*

Porque solo son utilizados internamente por el propio objeto para gestionar animaciones y estados visuales. No necesitan ser accesibles desde fuera.

***Métodos públicos:***

Constructor Goku1(...)

Destructor ~Goku1()

void iniciar(int x, int y) (heredado de Goku)

void detener() (heredado de Goku)

void cargarImagen() (heredado de Goku)

QString detectarColision() const

bool haTocadoCarro() const

bool haTocadoObstaculo() const

void patadaGokuNivel1()

*¿Por qué son públicos?*

Forman la interfaz necesaria para que el nivel (Nivel1) pueda:

Inicializar al personaje (iniciar).

Consultar su estado (haTocadoCarro, haTocadoObstaculo).

Ejecutar acciones específicas (patadaGokuNivel1).

Detenerlo (detener) al finalizar el nivel.

***Contenedores usados:***

QVector<QPixmap> frames (atributo privado)

*¿Por qué se usa QVector?*

Almacena los frames del sprite de forma eficiente.

Acceso rápido por índice para animaciones.

Integración nativa con Qt y QPixmap.

*¿Alternativas descartadas?*

std::vector: requeriría conversiones adicionales con QPixmap.

QList: menos eficiente para acceso secuencial frecuente.

***¿Por qué esta implementación es mejor?***

Encapsulación fuerte: estado interno es privado; solo se expone lo necesario.

Especialización clara: sobrescribe solo los métodos virtuales que necesita.

Sin señales propias: evita acoplamiento innecesario; delega al nivel.

Contenedor ligero: QVector optimiza animaciones sin complejidad extra.

Interfaz coherente: métodos públicos permiten control desde Nivel1 sin romper encapsulamiento.

* **Clase Goku2**

***Atributos privados:***

QTimer\* timerMovimiento / tSalto / tDanio / tMuerte / animSalto / avance / animAtaque / regreso: Controlan la lógica temporal (movimiento continuo, salto con gravedad, inmunidad tras daño, animación de muerte, secuencias de ataque Kamehameha). Son privados porque su gestión interna debe ser encapsulada para evitar interferencias externas.

bool mvtoIzquierda / mvtoDerecha / enSalto / mirandoDerecha / puedeRecibirDanio: Estados internos del personaje que no deben ser modificados directamente desde fuera para mantener la integridad del flujo del juego.

float gravedad / sueloY / velocidadVertical: Parámetros físicos del salto que solo Goku2 debe controlar.

Nivel2\* nivel2: Referencia al nivel para comunicación unidireccional (Goku2 → Nivel2) sin exponer la estructura interna del nivel.

QMediaPlayer\* salto / QAudioOutput\* salidaSalto: Gestión de efectos de sonido encapsulada para evitar manipulación externa.

***Atributos públicos:***

static int contador: Compartido entre instancias para rastrear cuántos Goku2 existen (útil para debugging).

Goku2(...)/~Goku2(): Constructor/destructor públicos porque el nivel principal debe crear/destruir instancias.

***Slots privados:***

actualizarSalto(), detectarPocion(), actualizarSpriteCaminar(), actualizarSpriteSalto(), caminarHaciaRobot(), atacarRobot(), caminarHaciaIzquierda(): Métodos reactivos internos que no deben ser invocados externamente. Se conectan a temporizadores internos para mantener la secuencia de animaciones/movimientos.

***Slots públicos:***

Ninguno. Toda la lógica se maneja internamente o mediante señales del nivel padre.

***Métodos privados:***

actualizarSalto(): Implementa la física del salto (gravedad + colisión con suelo). Privado porque solo Goku2 debe calcular su altura.

detectarPocion(): Detecta colisiones con poción y notifica al nivel. Privado porque la lógica de recolección es interna.

actualizarSpriteCaminar/Salto(): Gestión de frames de animación sin exponer detalles de sprites.

caminarHaciaRobot/atacarRobot/regreso: Secuencias de ataque Kamehameha encapsuladas para evitar interrupciones externas.

***Métodos públicos:***

cargarImagen(), iniciar(), detener(): Implementan métodos virtuales de Goku (clase base) para inicializar/recursos.

setSueloY(): Permite al nivel ajustar la altura del suelo dinámicamente.

animarMuerte(): Inicia la secuencia de muerte desde el nivel cuando Goku pierde toda la vida.

iniciarKamehameha(): Punto de entrada público para que el nivel inicie el ataque final contra el robot.

***¿Por qué esta implementación es mejor?***

Encapsulación fuerte: Atributos privados protegen el estado interno (salto, temporizadores).

Especialización clara: Sobrescribe solo métodos necesarios para Nivel2.

Sin señales propias: Reduce acoplamiento; delega eventos al nivel.

Timers como atributos: Permite reconectar secuencias (Kamehameha) sin recrearlos.

Contenedores Qt nativos: QVector optimiza animaciones sin overhead de conversiones.

* **Clase Robot**

***Atributos privados:***

QTimer\* timerMovimiento, timerAnimacion, timerAtaque, timerMuerte

QVector<QPixmap> frames, framesRobot2, framesMuerte

QList<Explosion\*> ListaExplosiones

int frameActual, velocidad, destinoX, frameMuerte

bool desplegado, modoMarcha, usarParabolico, estaMuerto

*¿Por qué son privados?*

Son el estado interno exclusivo del robot: temporizadores que controlan movimiento, disparos, muerte, y vectores de sprites que no deben ser manipulados externamente. Por ejemplo, si timerAtaque se detuviera desde fuera, se interrumpiría el ciclo de disparo y quedarían explosiones huérfanas. Mantenerlos privados garantiza que solo Robot gestione su ciclo de vida y animaciones.

***Atributos públicos:***

static int contador

*¿Por qué son públicos?*

contador: Estático y compartido para depuración (cuántos robots existen).

***Slots privados:***

mover(), animar()

*¿Por qué son privados?*

Conectados internamente a timerMovimiento y timerAnimacion. No deben ser invocados externamente porque romperían la secuencia lógica (ej. si mover() se llama directamente, podría saltarse límites de destino).

***Métodos privados:***

cargarImagen(), cargarRobot2(), cargarFramesMuerte(), dispararExplosion(), eliminarExplosiones(), animarYDisparar()

*¿Por qué son privados?*

Son detalles de implementación que no deben usarse externamente. Por ejemplo, cargarRobot2() carga sprites específicos del Nivel2; si fuera público, otros objetos podrían forzar una textura incorrecta.

***Métodos públicos:***

Constructores Robot(...), destructor ~Robot()

iniciar(), desplegarRobot(), detenerMvtoRobot(), iniciarAtaques(), murioRobot(), detenerAtaques()

getSprite(), getFramesNivel2()

*¿Por qué son públicos?*

Permiten que Nivel1/Nivel2 controlen al robot sin conocer detalles internos:

iniciar(): Posiciona y activa movimiento.

iniciarAtaques(): Nivel2 inicia el ciclo de disparos.

murioRobot(): Nivel2 activa la secuencia de muerte tras el Kamehameha.

***Señales:***

robotMurio()

*¿Por qué es pública?*

Notifica al nivel cuando la animación de muerte termina. Es pública porque Nivel2 necesita conectarla para cambiar de estado (emit nivelCompletado).

***Contenedores usados:***

QVector<QPixmap> frames, framesRobot2, framesMuerte: Almacenan sprites de animación.

QList<Explosion\*> ListaExplosiones: Gestiona explosiones activas para eliminación segura en el destructor.

*¿Por qué estos contenedores?*

QVector: Acceso rápido por índice para ciclos de animación.

QList: Permite inserción/eliminación eficiente de explosiones dinámicas (evita iteraciones costosas).

*¿Alternativas descartadas?*

std::vector: Requeriría conversiones con QPixmap.

QSet: No mantiene orden (crítico para frames secuenciales).

***¿Por qué esta implementación es mejor?***

Encapsulación fuerte: Estado interno (timers, sprites) es privado; solo se expone lo necesario.

Gestión segura de explosiones: QList + destructor evita fugas de memoria.

Señales claras: robotMurio() desacopla la muerte del robot del nivel.

Sin métodos virtuales: Simplifica la clase y evita complejidad innecesaria.

* **Clase Explosion**

***Atributos privados:***

QTimer\* timerMovimiento, timerAnimacion

float velocidadX, velocidadY, gravedad, tiempo

int frameActual

TipoMovimiento tipoMovimiento

QPointF posicionInicial

QVector<QPixmap> frames

*¿Por qué son privados?*

Representan el estado interno exclusivo de la explosión: temporizadores que controlan su trayectoria (parabólica o MRU), gravedad, frame actual de animación, y sprites. Si fueran públicos, otras clases podrían alterar la física o interrumpir la animación, rompiendo la coherencia visual y lógica. Mantenerlos privados garantiza que solo Explosion gestione su ciclo de vida.

***Atributos públicos:***

static int contador

static const QPointF posicionDisparo

*¿Por qué son públicos?*

contador: Estático compartido para depuración (cuántas explosiones existen).

posicionDisparo: Constante global para posición inicial por defecto (accesible sin instancia).

***Slots privados:***

avanzarFrameAnimacion()

*¿Por qué es privado?*

Conectado internamente a timerAnimacion para cambiar frames. No debe ser invocado externamente porque rompería la sincronización entre movimiento y animación.

***Métodos privados:***

avanzarFrameAnimacion()

*¿Por qué es privado?*

Es un detalle de implementación que actualiza el sprite internamente. No debe usarse externamente para evitar saltar frames o desincronizar con el movimiento.

***Métodos públicos:***

Constructor Explosion(...), destructor ~Explosion()

setTipoMovimiento(), setPosicionInicial(), lanzar()

*¿Por qué son públicos?*

Permiten que Robot (o cualquier clase) configure y active una explosión sin conocer detalles internos:

lanzar(): Inicia movimiento y animación.

setTipoMovimiento(): Robot decide si usar trayectoria parabólica o MRU.

***Contenedores usados:***

QVector<QPixmap> frames

*¿Por qué QVector?*

Almacena sprites de animación eficientemente.

Acceso rápido por índice para ciclos de animación (cada 300ms).

Más ligero que QList para datos grandes y acceso secuencial.

*¿Alternativas descartadas?*

std::vector: Requeriría conversiones con QPixmap.

QList: Menos eficiente para acceso indexado frecuente.

***¿Por qué esta implementación es mejor?***

Encapsulación fuerte: Estado interno (velocidad, gravedad, frames) es privado.

Sin señales: Reduce complejidad; Robot gestiona su ciclo.

Contenedor ligero: QVector optimiza animaciones sin overhead.

Configuración flexible: Métodos públicos (setTipoMovimiento) permiten reuso para distintos ataques (parabólico o MRU).

* **Clase Vida**

***Atributos privados:***

QLabel\* texto

QProgressBar\* barra

const int vidaMaxima = 100

int vidaActual = 100

*¿Por qué son privados?*

Representan el estado interno y la interfaz visual exclusiva de la barra de vida. vidaActual y vidaMaxima no deben ser accesibles externamente para evitar valores incoherentes (ej. vida negativa o >100). Los widgets (texto, barra) son detalles de presentación que solo Vida debe controlar para garantizar que cualquier cambio pase por métodos validados (restar(), reiniciar()), preservando la integridad del HUD.

***Métodos privados:***

actualizarColor()

*¿Por qué es privado?*

Es un detalle de implementación que cambia el color de la barra según el nivel de vida (rojo <40%, amarillo <70%, verde ≥70%). No debe usarse externamente porque rompería la coherencia visual (ej. forzar color rojo con vida 100%). Se invoca internamente tras restar() o reiniciar().

***Métodos públicos:***

Constructor Vida(QWidget\* parent), destructor ~Vida()

restar(int cantidad), reiniciar(), obtenerVida()

*¿Por qué son públicos?*

Forman la interfaz mínima y segura para interactuar con la barra de vida:

restar(): Reduce vida y actualiza color/visualización.

reiniciar(): Restaura vida máxima sin exponer el atributo.

`obtenerVida()\*\*: Permite consultar sin riesgo de modificación.

***Contenedores usados:***

Ninguno.

*¿Por qué no se usan?*

Vida gestiona un solo valor (vidaActual) y dos widgets fijos (texto, barra). No requiere colecciones dinámicas. Usar contenedores sería innecesario y aumentaría la complejidad.

***¿Por qué esta implementación es mejor?***

Encapsulación total: Atributos y widgets son privados; solo se expone lo necesario.

Interfaz simple: 3 métodos públicos cubren todos los casos de uso.

Sin señales ni slots: Reduce acoplamiento y sobrecarga.

Actualización automática: actualizarColor() se llama internamente tras cambios.

Widgets hijos: texto y barra se destruyen automáticamente con el padre (sin memoria manual).

* **Clase Progreso**

***Atributos privados:***

TipoProgreso tipo

float porcentaje

QLabel \*icono

int totalPociones

int pocionesRecolectadas

*¿Por qué son privados?*

Definen el estado interno exclusivo del widget: tipo de barra (horizontal o por pociones), avance actual, icono visual y contadores de poción. Mantenerlos privados evita que se modifiquen directamente y fuerza cualquier cambio a pasar por métodos validados (actualizarProgreso(), sumarPocion()), lo que garantiza coherencia entre lógica y presentación.

***Métodos protegidos:***

paintEvent(QPaintEvent \*event)

*¿Por qué es privado?*

Es un evento interno de pintado que Qt invoca automáticamente. No debe llamarse externamente, pues rompería el refresco controlado por el sistema.

***Métodos públicos:***

actualizarProgreso(float posGoku, float inicio, float fin)

setTotalPociones(int total)

sumarPocion()

getPorcentaje()

Constructor y destructor

*¿Por qué son públicos?*

Forman la API mínima y segura que usan los niveles:

actualizarProgreso: Nivel1 actualiza la barra según avance de Goku.

setTotalPociones / sumarPocion: Nivel2 configura y aumenta el conteo de pociones.

getPorcentaje: Permite consultar el avance sin exponer atributos internos.

***Contenedores usados:***

Ninguno.

¿Por qué no se usan?

Solo se manejan dos contadores simples (totalPociones, pocionesRecolectadas). No hay colecciones dinámicas; usar contenedores añadiría complejidad innecesaria.

***¿Por qué esta implementación es mejor?***

Encapsulación total: estado y widgets internos son privados.

API clara y pequeña: 4 métodos cubren todos los casos de uso.

Dibujo personalizado: paintEvent ofrece barra visual exacta sin depender de estilos del sistema.

Sin señales/slots ni contenedores: reduce acoplamiento y memoria.

* **Clase Pocion**

***Atributos privados:***

QVector<QPixmap> frames

QTimer\* timer

int indiceFrame

int fila, columna, columnasTotales

*¿Por qué son privados?*

Representan el estado interno exclusivo de la poción: sprites animados, temporizador de movimiento, índice de frame actual y posición lógica en la grilla. Mantenerlos privados evita que se modifiquen directamente y fuerza cualquier cambio a pasar por métodos controlados (detener(), moverYAnimar()), garantizando coherencia entre animación y posición.

***Atributos públicos:***

static int contador

*¿Por qué es público?*

Variable estática compartida para depuración (cuántas pociones existen globalmente). No expone el estado individual.

***Slots privados:***

moverYAnimar()

*¿Por qué es privado?*

Conectado internamente al timer para actualizar posición y frame. No debe usarse externamente porque rompería la secuencia de caída y animación.

***Métodos públicos:***

Constructor Pocion(...)

Destructor ~Pocion()

detener()

*¿Por qué son públicos?*

Constructor: permite que Nivel2 cree pociones con parámetros de grilla.

detener(): Nivel2 o Goku2 puede pausar la animación cuando la poción es recolectada.

***Contenedores usados:***

QVector<QPixmap> frames

*¿Por qué QVector?*

Almacena sprites de animación eficientemente.

Acceso rápido por índice para ciclos de 100 ms.

Más ligero que QList para acceso secuencial frecuente.

*¿Alternativas descartadas?*

std::vector: Requeriría conversiones QPixmap.

QList: Menos eficiente para acceso indexado.

***¿Por qué esta implementación es mejor?***

Encapsulación fuerte: estado y animación son privados.

Interfaz mínima: solo detener() es público para control externo.

Sin señales/slots: reduce complejidad y acoplamiento.

Contenedor ligero: QVector optimiza animación sin overhead.

* **Clase camaraLogica**

***Atributos privados:***

QGraphicsView \*view

QTimer \*timer

Goku \*objetivo = nullptr

*¿Por qué son privados?*

Representan el estado interno exclusivo de la cámara: la vista a mover, el temporizador que genera el “tick” de seguimiento y el personaje al que debe seguir. Si alguno fuese público, cualquier clase podría cambiar de repente la vista que se mueve, detener el timer o asignar otro objetivo, rompiendo el seguimiento continuo y fluido que requiere el nivel. Al ser privados, solo camaraLogica controla cuándo y cómo actualiza la posición de la vista.

***Atributos públicos:***

static int contador

*¿Por qué es público?*

Es una variable estática compartida entre todas las instancias, útil para depurar cuántas cámaras lógicas se han creado durante la ejecución. No revela el estado interno de una cámara concreta, por lo que mantenerla pública no compromete la encapsulación.

***Slots privados:***

moverVista()

*¿Por qué es privado?*

Está conectado internamente al timeout de timer; su ejecución está ligada al propio ciclo de refresco de la cámara. Si fuese público, cualquier objeto podría forzar un movimiento fuera de tiempo y desincronizar la cámara con la posición real de Goku. Al ser privado, garantiza que el movimiento ocurra solo cuando el propio timer lo solicite.

***Métodos públicos:***

Constructor camaraLogica(...)

Destructor ~camaraLogica()

iniciarMovimiento(), detenerMovimiento()

seguirAGoku(Goku\*)

*¿Por qué son públicos?*

Forman la interfaz mínima y segura que usa el nivel:

seguirAGoku(...): le indica a la cámara quién debe seguir.

iniciarMovimiento() / detenerMovimiento(): nivel controla cuándo debe activar o pausar el seguimiento sin conocer detalles de timers o cálculos internos.

***Señales:***

Ninguna.

*¿Por qué no hay?*

La cámara no necesita notificar eventos externos; su único propósito es actualizar la vista. Cualquier lógica de “fin de nivel” o “Game Over” la maneja el nivel principal, no la cámara.

***Contenedores usados:***

Ninguno.

*¿Por qué no se usan?*

Solo se necesitan tres punteros simples (view, timer, objetivo). No hay colecciones dinámicas ni múltiples objetos que gestionar.

***¿Por qué esta implementación es mejor?***

Encapsulación fuerte: vista, timer y objetivo son privados; solo se expone la interfaz de control.

Interfaz mínima: 4 métodos públicos cubren todo el caso de uso.

Sin señales/slots ni contenedores: reduce acoplamiento y memoria.

Frecuencia fija: timer interno a 60 FPS garantiza seguimiento suave sin intervención externa.

* **Clase carro**

***Atributos privados:***

QPixmap imagenCarro

int cuadroActual, anchoCuadro, altoCuadro

QTimer \*timerRotacion, \*timerEspiral

float g, vx, vy, radio, tiempoGiro, tiempo, ySuelo

QPointF inicio

int fase

bool girando, espiralHecha

*¿Por qué son privados?*

Representan el estado interno exclusivo del carro: hoja de sprites, índice de animación, parámetros físicos del salto y giro, y temporizadores de movimiento. Si fueran públicos, se podrían alterar valores de gravedad o trayectoria, rompiendo la coreografía del vuelo. Al ser privados, solo Carro controla su propia animación y física.

***Atributos públicos:***

static int contador

static int contCarro

bool espiralHecha

*¿Por qué son públicos?*

contador/contCarro: estadísticas globales para depurar cuántos carros existen.

espiralHecha y haLlegadoAlSuelo: banderas públicas que Nivel1 consulta para saber cuándo crear robots o cambiar fase, sin exponer detalles internos de cálculo.

***Slots privados:***

actualizarMovimiento()

animarRotacion()

*¿Por qué son privados?*

Conectados internamente a timerEspiral y timerRotacion. No deben ser invocados externamente porque romperían la secuencia del vuelo parabólico o la animación de rotación.

***Métodos públicos:***

Constructor Carro(...)

Destructor ~Carro()

iniciar(int x, int y)

rotar()

iniciarMovimientoEspiral(float posXpatada)

bool estaGirando() const

bool haLlegadoAlSuelo() const

*¿Por qué son públicos?*

Forman la interfaz que usa Nivel1:

iniciarMovimientoEspiral(...): nivel le indica dónde empezar el vuelo tras la patada.

estaGirando() / haLlegadoAlSuelo(): nivel consulta estado sin conocer física interna.

***Contenedores usados:***

Ninguno.

*¿Por qué no se usan?*

Solo se manejan cuatro frames de animación y una única trayectoria; no hay colecciones dinámicas.

***¿Por qué esta implementación es mejor?***

Encapsulación fuerte: física y animación son privadas.

Interfaz mínima: 3–4 métodos públicos cubren todo el uso.

Sin señales/contenedores: reduce complejidad y memoria.

Estado público ligero: banderas espiralHecha y haLlegadoAlSuelo permiten al nivel decidir siguiente fase sin invadir lógica interna.

* **Clase obstáculo**

***Atributos privados:***

QTimer \*timerMovimiento

int frameActual, velocidad, coordX, coordY, fotogWidth, fotogHeight

Tipo tipo

*¿Por qué son privados?*

Representan la configuración interna de cada obstáculo: temporizador de desplazamiento, índice de frame, velocidad y dimensiones. Mantenerlos privados impide que se modifiquen externamente, evitando que un obstáculo acelere o cambie de sprite fuera de su propia lógica, lo que rompería la dificultad diseñada por el nivel.

***Atributos públicos:***

static int contador

static int contObsta

QGraphicsPixmapItem \*Sprite

*¿Por qué son públicos?*

contador/contObsta: estadísticas globales para depurar cantidad de obstáculos.

sprite: Nivel1 y Nivel2 necesitan acceder a la posición y tamaño para colisiones y para eliminarlo de la escena al destruirlo.

***Slots privados:***

mover()

actualizar()

*¿Por qué son privados?*

Conectados internamente a timerMovimiento y timerAnimacion. No deben ser invocados externamente porque romperían la secuencia de desplazamiento o la animación de frames.

***Métodos públicos:***

Constructor obstaculo(...)

Destructor ~obstaculo()

void iniciar(int x, int y)

void cargarImagenes()

int getAltura() const

QGraphicsPixmapItem\* getSprite() const

*¿Por qué son públicos?*

Forman la interfaz mínima que usa el nivel:

iniciar(...): posiciona y arranca el movimiento.

getAltura() / getSprite(): nivel consulta tamaño y sprite para colisiones y renderizado.

***Contenedores usados:***

QVector<QPixmap> frames

*¿Por qué QVector?*

Almacena los 4 frames del ave con acceso rápido por índice para animación de alas cada 100 ms. Más ligero que QList y sin conversiones que requeriría std::vector.

***¿Por qué esta implementación es mejor?***

Encapsulación fuerte: estado y temporizadores son privados.

Interfaz simple: 3–4 métodos públicos cubren creación, posicionamiento y consulta.

Sin señales/contenedores dinámicos: reduce acoplamiento y uso de memoria.

Sprite público: permite al nivel gestionar colisiones y eliminación sin romper encapsulamiento.

**Logros alcanzados**

A lo largo del desarrollo del proyecto se logró un sistema de juego robusto y modular, donde cada clase encapsula responsabilidades específicas: desde la física del salto y las colisiones hasta la animación fluida y la gestión de recursos. Se implementó un flujo claro de comunicación entre niveles, personajes y elementos visuales sin acoplamiento excesivo. Se alcanzó un rendimiento estable a 60 FPS mediante temporizadores eficientes y se garantizó seguridad de memoria con destructores explícitos y desconexión automática de señales. Además, se logró una experiencia visual coherente con HUD dinámicos, algunos efectos de sonido y transiciones suaves, todo mientras se mantuvo un código limpio, reutilizable y fácil de extender para futuros niveles o personajes.

**Análisis de mejoras**

Se podría haber reducido el uso de temporizadores individuales sustituyéndolos por un único bucle principal de juego con eventos diferidos, lo que simplificaría la sincronización y disminuiría el overhead. Además, la lógica de colisiones podría haberse encapsulado en un sistema dedicado en lugar de dispersarse en cada personaje y obstáculo, facilitando ajustes y extensiones.

Se podrían haber definido interfaces o clases base más pequeñas (por ejemplo, un Animable o Fisicable) para reutilizar comportamiento común sin herencia pesada, y documentar las constantes físicas (gravedad, velocidades) en un único archivo de configuración para ajustes rápidos que reciban el objeto y realicen el movimiento físico sin recompilar.

Por último, se podría haber creado una única clase genérica que agrupe la funcionalidad de Vida y Progreso, por ejemplo, BarraHUD, usando un enum o flag para distinguir entre barra de vida o barra de progreso. Esto habría reducido duplicación de código (widgets QLabel + QProgressBar, métodos de pintado, estilos) y facilitado mantener consistencia visual. Sin embargo, habría aumentado la complejidad interna al tener que gestionar dos lógicas distintas (porcentaje vs. pociones) y dos modos de actualización (daño/restauración vs. recolección). Por tanto, mantener clases separadas resultó más limpio y especializado, aunque a costa de repetir estructura.

**Manejo de memoria**

El proyecto demuestra un excelente manejo de memoria gracias a la combinación de destructores explícitos, desconexión automática de señales/slots y el uso de jerarquía de objetos de Qt. Cada clase se encarga de liberar sus recursos: temporizadores se detienen y destruyen en destructores, sprites se remueven de la escena antes de eliminarse, y los widgets hijos se liberan automáticamente por su parentesco. Además, se evita la duplicación de objetos pesados (pixmaps, sonidos) y se utiliza QPointer o atributos donde es posible, garantizando que no haya fugas de memoria ni accesos a objetos ya eliminados durante el ciclo de vida del juego.

**Manejo de excepciones**

En el proyecto se usa validación de excepciones estándar (std::invalid\_argument, std::runtime\_error) en constructores y métodos clave para detectar errores lo antes posible y evitar estados inconsistentes.

***Goku (goku.cpp/goku.h)***

* + throw std::invalid\_argument("Goku: la escena no puede ser nula.") si recibe scene = nullptr.
  + throw std::invalid\_argument("Goku: las dimensiones de los sprites deben ser positivas.") si fotogWidth <= 0.
  + throw std::runtime\_error("Goku::recibirDanio - no se ha asociado una barra de vida.") si vidaHUD == nullptr.

Para qué: asegurar que el personaje se cree con datos válidos y con HUD asociado antes de jugar.

***Explosion (explosion.cpp)***

* + throw std::runtime\_error("Explosion: No se pudo cargar explosion.png.") si la imagen falla.
  + throw std::runtime\_error("Explosion: No se pudieron extraer los frames...") si el sprite-sheet es inválido.

Para qué: abortar creación si falta recurso gráfico crítico.

***Robot (robot.cpp)***

* + throw std::invalid\_argument("Robot: la escena no puede ser nula.") al construir.
  + throw std::runtime\_error("Robot::cargarImagen - No se encontró robots1.png.") si falta sprite.

Para qué: evitar robots sin gráficos o sin escena.

***Pocion (pocion.cpp)***

* + throw std::invalid\_argument("Pocion: los frames no pueden estar vacíos.")
  + throw std::runtime\_error("Pocion: los márgenes de posicionamiento horizontal son inválidos.")

Para qué: validar datos de entrada antes de posicionar la poción.

***Progreso (progreso.cpp)***

* + throw std::runtime\_error("Progreso: No se pudo cargar el ícono...") si falla la imagen.
  + throw std::invalid\_argument("Progreso: El total de pociones debe ser mayor que 0.") si se intenta un total ≤ 0.

Para qué: garantizar que la barra se inicialice con recursos y parámetros correctos.

Ubicación: todas estas excepciones se lanzan en constructores o métodos de inicialización (cargarImagen(), lanzar(), etc.) y no se capturan internamente, de modo que el error se propaga inmediatamente al nivel o a la ventana principal, facilitando detección temprana y mensajes claros en consola o diálogo.

**Cambios del momento I y II con la versión actual**

Entre el Momento I y II y el proyecto final, existen diferencias notables que reflejan ajustes por limitaciones de tiempo y priorización de funcionalidades: aunque en el documento se planteó un temporizador activo en ambos niveles que limitaba el tiempo para completar cada fase, en la implementación final este sistema no fue incluido, dejando abierta la posibilidad de agregarlo en una futura versión; del mismo modo, se propuso un sistema de diálogo interactivo con Pilaf al final del nivel 1, pero en el proyecto final este diálogo se redujo a una transición visual con imagen estática con un mensaje; también se contempló que el robot del nivel 2 tuviese movimiento vertical limitado y ataques más complejos, sin embargo, su comportamiento final fue más estático y predecible, centrado en disparar explosiones sin desplazarse; además, se pensó que el Kamehameha se activaría tras recolectar 10 cápsulas y el jugador lo ejecutaría manualmente, pero en la versión final se automatizó tras completar la recolección de 5 cápsulas, eliminando la interacción directa del jugador; finalmente, elementos como el HUD con contador de tiempo, diálogos dinámicos y mayor variedad de ataques enemigos quedaron pendientes para una implementación posterior, priorizando la jugabilidad básica y la estabilidad del sistema.

**Conclusiones**

En conclusión, el proyecto *La aventura de Goku* demostró que es posible crear un videojuego 2D completo y funcional utilizando exclusivamente C++ y Qt, aprovechando su sistema de señales y slots, gestión automática de memoria y renderizado gráfico eficiente. Se lograron dos niveles jugables con mecánicas distintas: desplazamiento automático y colisiones en el primero, y recolección de pociones con física de salto y combate final en el segundo. La arquitectura basada en clases abstractas y herencia permitió mantener un código limpio, modular y fácil de extender, mientras que el uso de contenedores como QPixmap y QVector optimizó la animación y el rendimiento. A nivel técnico, se resolvieron desafíos clave como el seguimiento de cámara suavizado, la gestión segura de recursos dinámicos y la integración de HUD reactivos. El resultado final cumple con los objetivos planteados, ofrece una experiencia de juego fluida y establece una base sólida para futuras mejoras como nuevos niveles, enemigos o modos multijugador.