INFORME: PROYECTO FINAL

1. INTRODUCCIÓN Y OBJETIVOS:

1.1 Contexto del Proyecto y Motivación

Temática del proyecto: cada nivel esta inspirado en el acontecimiento de la caída de Constantinopla.

Motivación: Desarrollo de un videojuego sencillo que ponga en práctica los conocimientos adquiridos en el curso Informática II los cuales son programación Orientada a Objetos, gestión de memoria dinámica e interfaces gráficas con Qt. El evento histórico elegido para el desarrollo de los tres niveles de este proyecto es la Caída de Constantinopla. El enfoque se sitúa desde el punto de vista de los defensores, y se refleja en el objetivo general que une a todos los niveles: "Sobrevive". Lo que varía entre un nivel y otro es la forma en la que este objetivo debe alcanzarse.

1.2 Objetivos de Desarrollo

- Implementar al menos tres modelos físicos diferentes (movimiento no rectilíneo)
- Utilizar programación orientada a objetos y memoria dinámica
- Implementar al menos una herencia propia (no de Qt)
- Incluir un agente inteligente con comportamiento autónomo
- Implementar sistema de sonido (fondo + eventos)
- Generar ejecutable funcional
- Crear un videojuego con tres niveles con mecánicas claramente diferentes
- Implementar interfaz gráfica con Qt
- Desarrollar arquitectura modular
- Asegurar dificultad progresiva
- Implementar sistema de colisiones

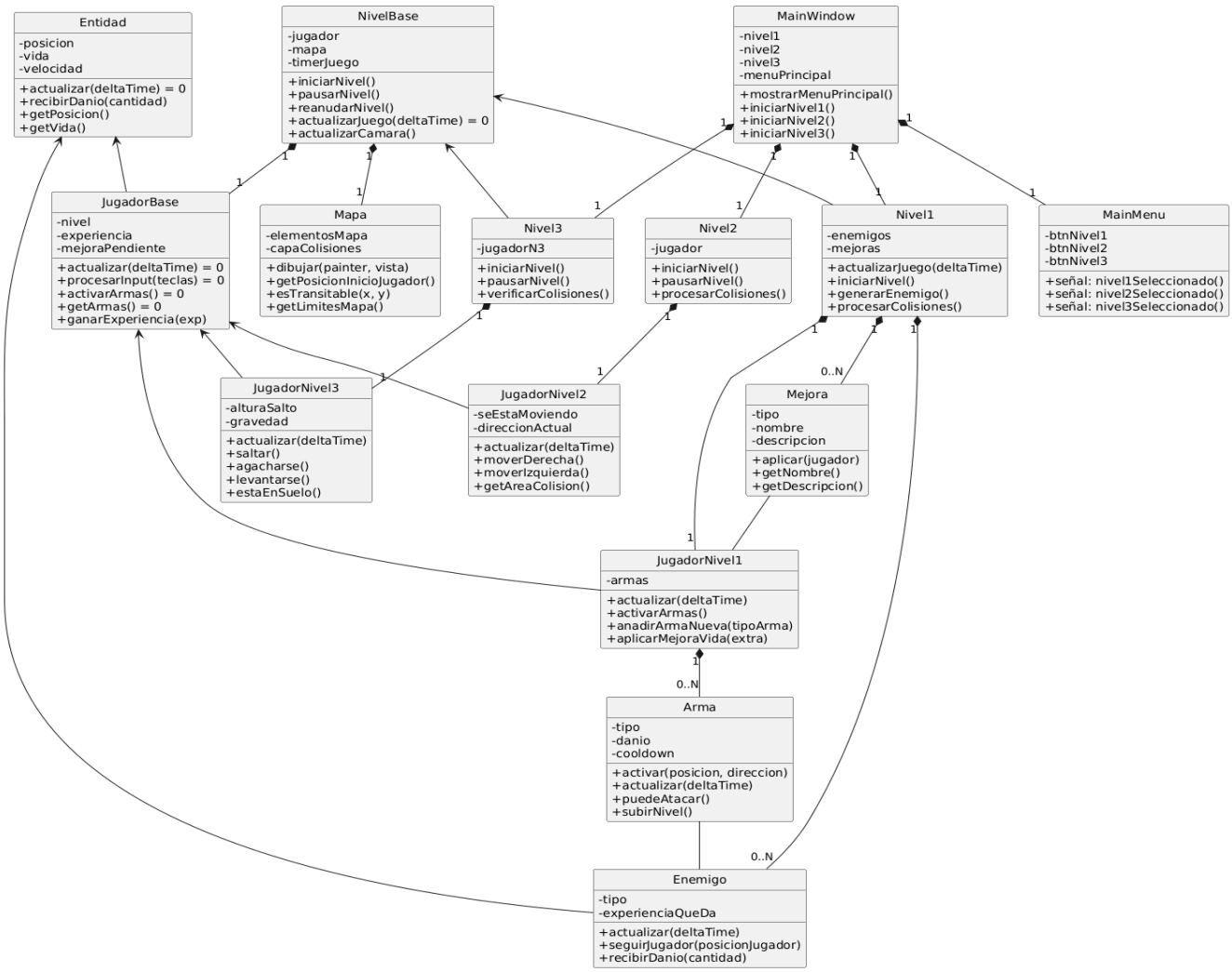
1.3 Requisitos del proyecto

Requisito	Implementación
Tres modelos físicos	Gravedad, proyectiles, movimiento oscilatorio
POO y memoria dinámica	Clases, new/delete gestionados

Diagrama de clases	UML Simplificado
Herencia propia	Entidad → JugadorBase → JugadorEspecifico
Dificultad Progresiva	Sistema de oleadas y velocidad incremental
Contenedores STL/Qt	Qlist, Qvector, std::vector
Repositorio y commits	https://github.com/joserendon1/Proyecto-Final-Informatica-II
Excepciones	Manejo de errores a lo largo del programa
Interfaz Qt	Widgets, signals/slots, Qpainter
Agente inteligente	Enemigos con percepción, decisión y acción
Sistema de sonido	Audiomanager con fondo y efectos
Ejecutable	Aplicación compilada

2. ARQUITECTURA DEL SISTEMA

2.1 Diagrama de Clases



El sistema que se representa en el diagrama de clases, muestra una jerarquía bien definida centrada en la clase abstracta `Entidad`. Esta clase base proporciona propiedades comunes como posición, vida y velocidad, junto con métodos virtuales que deben implementar todas las entidades del juego. De “Entidad” se derivan dos ramas principales: “JugadorBase” para los personajes controlables y “Enemigo” para los adversarios autónomos.

Para la jerarquía de jugadores. “JugadorBase” define una interfaz común pero abstracta, dejando que las clases concretas “JugadorNivel1”, “JugadorNivel2” y “JugadorNivel3” implementen comportamientos específicos para cada nivel. Esta decisión de diseño permite que cada nivel tenga mecánicas de juego únicas mientras comparten una base común de funcionalidad.

Los niveles siguen un patrón similar. “NivelBase” proporciona la estructura fundamental para la gestión del juego, incluyendo el sistema de cámara, manejo de input y temporización. Los niveles concretos (“Nivel1”, “Nivel2”, “Nivel3”) extienden esta base con lógica específica.

2.2 Diseño

En “NivelBase”, donde se define un algoritmo para el ciclo del juego, dejando que las subclases implementen los pasos específicos. Esto se logra mediante métodos virtuales puros como “actualizarJuego(float deltaTime)” que cada nivel debe implementar a su manera.

Cada tipo de jugador encapsula un conjunto de algoritmos para movimiento, ataque y procesamiento de input, permitiendo intercambiar comportamientos según el nivel. Por ejemplo, “JugadorNivel1” tiene un sistema completo de armas y mejoras, mientras que “JugadorNivel3” se centra en mecánicas de salto y gravedad.

Para la gestión de recursos compartidos como sprites y sonidos, se utiliza “SpriteManager” y “AudioManager”. Estos managers centralizan el acceso a recursos, evitando duplicación en memoria y proporcionando una interfaz para toda la aplicación.

Los niveles emiten señales cuando ocurren eventos importantes (como pausa del juego o finalización de nivel), y la ventana principal reacciona a estos eventos actualizando la interfaz o cambiando de pantalla. Esto permite que los componentes se comuniquen sin conocer detalles de su implementación entre sí.

2.3 Flujo

El flujo de ejecución del juego sigue un ciclo, coordinado principalmente por la clase “NivelBase”. Un temporizador Qt (“QTimer”) dispara a intervalos regulares (aproximadamente 60 veces por segundo),

activando el slot “onTimerTimeout()”. Este calcula el tiempo transcurrido desde el último frame (deltaTime) y llama al método “actualizarJuego()”, que cada nivel implementa de forma específica.

Dentro de la actualización del juego, primero se procesa el input del usuario. Las teclas presionadas se almacenan en un vector de booleanos que representa el estado actual del teclado. Cada jugador procesa este input según sus propias reglas: “JugadorNivel1” permite movimiento en cuatro direcciones, “JugadorNivel2” se limita a movimiento horizontal, y “JugadorNivel3” responde a comandos de salto y agacharse.

Posteriormente, se actualizan todas las entidades del juego. Los enemigos, que implementan un agente inteligente básico, calculan su comportamiento autónomo, generalmente persiguiendo al jugador cuando están dentro de un rango de detección. El sistema de colisiones verifica interacciones entre entidades y con el entorno, aplicando daño o eliminando objetos según corresponda.

Finalmente, se invoca el método “paintEvent()” para renderizar la escena.

2.4 Gestión de memoria

La gestión de memoria en el proyecto se enfoca según la necesidad específica de cada componente. Para objetos con ciclo de vida bien definido y propiedad clara, se utilizan punteros tradicionales con liberación manual en los destructores. Por ejemplo, en “Nivel1” se eliminan explícitamente los enemigos al finalizar el nivel:

```
Nivel1::~Nivel1()
{
    if (timerOleadas) {
        timerOleadas->stop();
        delete timerOleadas;
        timerOleadas = nullptr;
    }

    qDeleteAll(enemigos);
    enemigos.clear();

    AudioManager::getInstance().stopBackgroundMusic();

    qDebug() << "Nivel 1 destruido - recursos limpiados";
}
```

Para colecciones de objetos, se aprovechan los contenedores de Qt como QList y QVector, que gestionan automáticamente la memoria de los contenedores mismos, aunque no de los elementos apuntados. Esto por conveniencia.

Los recursos gráficos y de audio se cargan al inicio del juego mediante los managers correspondientes. **SpriteManager::preloadGameSprites()** carga todas las imágenes necesarias desde archivos de recursos embebidos en el ejecutable, asegurando que estén disponibles instantáneamente durante el juego. De manera similar, **AudioManager::loadSounds()** prepara los efectos de sonido y música de fondo.

```
void SpriteManager::preloadGameSprites()
{
    qDebug() << "=== CARGANDO SPRITES ===";

    bool loaded1 = loadSprite("player_idle", ":/sprites/player/Warrior_Idle.png");
    bool loaded2 = loadSprite("player_move", ":/sprites/player/Warrior_Run.png");
    bool loaded3 = loadSprite("enemy_weak", ":/sprites/enemies/Lancer_Run.png");
    bool loaded4 = loadSprite("enemy_strong", ":/sprites/enemies/Warrior_Run.png");
    bool loaded6 = loadSprite("projectile_arrow", ":/sprites/attacks/arrow.png");
    bool loaded7 = loadSpriteSheet("oil_effect", ":/sprites/attacks/oil.png", QSize(96, 96));
```

El manejo de errores se realiza principalmente cuando ocurren problemas no críticos, como la falta de un archivo de sprite específico. En esos casos, el sistema utiliza gráficos de respaldo (fallbacks) para garantizar que el juego continúe funcionando.

2.5 Sistema

La clase MainWindow actúa como coordinador central del juego. Gestiona la transición entre el menú principal y los diferentes niveles, manteniendo solo un nivel activo a la vez para optimizar el uso de recursos. Dependiendo de la señal recibida (nivel completado, game over, etc.), decide qué pantalla mostrar a continuación.

La comunicación entre la ventana principal y los niveles se realiza exclusivamente a través de señales Qt. Esta arquitectura modular proporciona una base sólida para extensiones.

3. COMPONENTES PRINCIPALES

3.1 Entidades

En el núcleo del juego se encuentra la clase “Entidad”, una clase abstracta que define las propiedades fundamentales de cualquier objeto interactivo del juego. Todas las entidades comparten una posición en el espacio (representada como “QPointF”), un valor de vida y una velocidad de movimiento. La abstracción se logra mediante el método puro “actualizar(float deltaTime)”, que fuerza a todas las clases derivadas a implementar su lógica de actualización específica.

La elección de “QPointF” para la posición; permite coordenadas con precisión de punto flotante, esencial para movimientos y cálculos de física. El sistema de vida es simple: cuando una entidad recibe daño, su vida disminuye, y al llegar a cero, la entidad deja de estar activa. Este sistema simplifica considerablemente la gestión de colisiones y el cálculo de daños en todo el juego.

De “Entidad” derivan dos categorías principales: los jugadores y los enemigos. Esta separación refleja la diferencia fundamental entre entidades controladas por el usuario y aquellas con comportamiento autónomo. Ambas comparten la misma base, pero sus implementaciones dependen según su rol en el juego.

3.2 Jugadores

La clase “JugadorBase” extiende “Entidad” con funcionalidades específicas para personajes jugables. Sin embargo, sigue siendo abstracta, definiendo una interfaz que todos los jugadores deben implementar pero dejando los detalles a las clases concretas. Esta decisión de diseño permite que cada nivel tenga mecánicas de juego diferentes mientras mantiene una base común de código.

JugadorNivel1

El primer tipo de jugador, utilizado en el nivel de supervivencia, es el más complejo de los tres. “JugadorNivel1” implementa un sistema completo de armas donde el jugador puede equipar y mejorar múltiples armas simultáneamente. El movimiento es libre en cuatro direcciones (WASD)

El sistema de experiencia y niveles añade profundidad al gameplay. Al derrotar enemigos, el jugador gana puntos de experiencia. Al alcanzar umbrales específicos, sube de nivel, recibiendo mejoras automáticas a sus estadísticas y la oportunidad de seleccionar una mejora especial entre varias opciones.

JugadorNivel2

En contraste, ”JugadorNivel2” simplifica las mecánicas. El movimiento se reduce a dos direcciones (izquierda y derecha), con límites claros que mantienen al jugador dentro de un corredor definido. No hay sistema de armas ni de experiencia; el objetivo es puramente de supervivencia y esquiva.

El jugador tiene una animación diferenciada entre estado idle y movimiento, y el cálculo de colisiones está para las rocas que caen desde arriba. La vida se maneja de manera más directa, recibiendo daño inmediato al colisionar con obstáculos.

JugadorNivel3

El tercer tipo de jugador introduce un entorno de scroll lateral. “JugadorNivel3” implementa un sistema de física completo con gravedad, salto y estados de agachado. La implementación del salto: utiliza fórmulas físicas para calcular la velocidad vertical inicial basada en la altura deseada del salto.

Un aspecto importante es el sistema de "cancelación de salto". Cuando el jugador presiona la tecla S durante un salto, activa una gravedad aumentada que lo hace caer más rápido. Esto añade control para que los jugadores puedan ajustar sus saltos y evitar obstáculos.

3.3 Enemigos

La clase “Enemigo” representa el componente investigativo requerido por el proyecto: un agente autónomo con comportamiento inteligente básico. Cada enemigo tiene un tipo (débil o fuerte) que determina sus estadísticas y la experiencia que otorga al ser derrotado.

La inteligencia del enemigo se manifiesta en su capacidad para "seguir" al jugador. Implementando el componente de percepción del agente, el enemigo detecta la posición del jugador cuando está dentro de un rango de detección (2000 píxeles en el código). Una vez detectado, calcula la dirección hacia el jugador y se mueve en esa dirección.

El componente de razonamiento es simple pero efectivo: si el jugador está cerca, perseguirlo; de lo contrario, moverse aleatoriamente o permanecer en su posición.

3.4 Combate

El sistema de combate está encapsulado en la clase “Arma”, que utiliza un enumerado para definir tipos específicos: ballesta, aceite, arco. Cada arma tiene propiedades únicas como daño, tiempo de recarga (cooldown) y efectos visuales. El sistema está diseñado para ser extensible; añadir un nuevo tipo de arma requiere principalmente ampliar el enumerado y proporcionar implementaciones para sus métodos de activación.

Las armas implementan un patrón de actualización basado en tiempo. El cooldown se reduce gradualmente con cada frame, y solo cuando llega a cero el arma puede dispararse nuevamente. Esto evita que los jugadores "spameen" ataques y añade un elemento estratégico de gestión de recursos.

Las mejoras, representadas por la clase “Mejora”, complementan el sistema de armas. Cuando el jugador sube de nivel en el Nivel 1, se le presentan varias opciones de mejora. Estas pueden ser nuevas armas o mejoras a armas existentes. El sistema genera opciones inteligentemente, priorizando armas que el jugador no posee todavía.

3.5 Niveles

NivelBase

“NivelBase” proporciona la infraestructura común a todos los niveles. Como clase abstracta, define la estructura del ciclo del juego pero delega la implementación específica a las clases hijas. Sus responsabilidades incluyen la gestión del temporizador principal, el manejo básico de input del teclado, y el sistema de cámara que sigue al jugador por el mundo del juego.

El sistema de cámara merece mención especial. Calcula qué porción del mundo debe mostrarse en pantalla basándose en la posición del jugador, con márgenes que mantienen al jugador cómodamente centrado. Además, proporciona métodos de utilidad como “estaEnVista()” que permiten a los niveles optimizar el renderizado, dibujando solo las entidades visibles.

Nivel1

El primer nivel implementa un modo supervivencia con generación procedural de enemigos. El mapa es cinco veces más grande que la pantalla visible, creando un sentido de exploración y espacio. Las oleadas de enemigos aumentan en dificultad con el tiempo, tanto en número como en variedad de tipos.

El sistema de generación de enemigos: intenta spawnear enemigos cerca del jugador pero fuera de la vista inmediata, y verifica que la posición de spawn sea transitable (no dentro de un obstáculo). Si después de varios intentos no encuentra una posición válida, utiliza posiciones de emergencia para garantizar que el juego continúe fluido.

El menú de mejoras es una característica destacada. Cuando aparece, pausa el juego y muestra opciones en un formato visual atractivo, con ribbons de colores y texto claro. El jugador navega las opciones con A/D y selecciona con espacio, en una interfaz intuitiva que no interrumpe la inmersión.

Nivel2

El segundo nivel cambia completamente la dinámica. En lugar de un mundo abierto, presenta un corredor fijo donde rocas caen desde arriba. La innovación aquí está en la variedad de movimientos físicos implementados: caída lineal, movimiento oscilatorio (senoidal), movimiento parabólico.

Cada tipo de movimiento no solo se ve diferente, sino que también afecta el gameplay. Los barriles parabólicos son más difíciles de esquivar porque se mueven diagonalmente, mientras que los oscilatorios requieren que el jugador anticipe su trayectoria curva. Esta variedad mantiene el nivel interesante a pesar de su mecánica aparentemente simple.

Nivel3

El tercer nivel combina mecánicas de plataforma con scroll automático continuo. El mundo se desplaza constantemente hacia la izquierda, forzando al jugador a mantener el ritmo mientras esquiva obstáculos. El sistema de generación de obstáculos es procedural, con patrones predefinidos que se activan según la distancia recorrida.

Los obstáculos vienen en dos variedades: altos y bajos. Esta simple dicotomía crea patrones interesantes cuando se combinan en secuencias.

3.6 Sprites

La clase “SpriteManager”y su método “preloadGameSprites()” carga al inicio todas las imágenes necesarias desde el archivo de recursos Qt (.qrc), asegurando que estén disponibles instantáneamente durante el gameplay.

El manager soporta dos tipos de recursos: sprites individuales y sprite sheets. Las sprite sheets son particularmente eficientes para animaciones, ya que contienen múltiples frames en una sola imagen. El manager puede extraer frames individuales de estas sheets calculando las coordenadas basadas en el índice del frame y el tamaño conocido de cada frame.

El sistema incluye mecanismos de fallback. Si un sprite no se carga correctamente, el juego no se detiene; en su lugar, utiliza formas geométricas simples con colores distintivos. Esto garantiza que el juego sea jugable incluso si hay problemas con los recursos gráficos.

Las animaciones se implementan mediante un sistema basado en tiempo. Cada entidad animada mantiene un contador de tiempo y un índice de frame actual. Cuando el contador supera un umbral específico (por ejemplo, 0.125 segundos para animación idle), avanza al siguiente frame.

El Nivel2 muestra una implementación particularmente elaborada, con animaciones separadas para estado idle y movimiento, cada una con su propio timing y número de frames.

3.7 HUD

Similar a “SpriteManager”, “UIManager” centraliza todos los recursos de interfaz. Carga fuentes personalizadas desde archivos TTF, sprites para elementos de UI como botones y ribbons, y proporciona métodos para dibujar texto formateado consistentemente en toda la aplicación.

El sistema de fuentes. Intenta cargar la fuente desde múltiples ubicaciones: primero desde recursos , luego desde directorios. Si todas las opciones fallan, recurre a fuentes del sistema, garantizando que el juego siempre tenga algún texto visible.

El menú principal es simple pero efectivo: cuatro botones claramente etiquetados para cada nivel y para salir del juego. Implementa el patrón de señal/slot de Qt, emitiendo señales específicas cuando el usuario selecciona una opción. “MainWindow” escucha estas señales y responde cambiando a la pantalla correspondiente.

Cada nivel tiene su propio HUD personalizado que muestra información relevante para esa modalidad de juego. El Nivel1 muestra vida, nivel, experiencia, tiempo y contador de enemigos. El Nivel2 añade una barra de vida visual y contadores de puntuación. El Nivel3 incluye distancia recorrida y un temporizador.

Todos los HUDs comparten un estilo visual consistente gracias a “UIManager”, pero se adaptan en contenido y layout a las necesidades específicas de cada nivel. Esta consistencia visual ayuda al jugador a sentirse en el mismo juego a pesar de las diferencias mecánicas entre niveles.

3.8 Audio

El sistema de audio, implementado en “AudioManager”, cumple con el requisito investigativo de sonido de fondo y efectos. Utiliza las capacidades multimedia de Qt para cargar y reproducir archivos de audio desde recursos.

El manager categoriza los sonidos en música de fondo y efectos de eventos. La música se reproduce en loop durante el gameplay, creando una atmósfera consistente. Los efectos (disparos, golpes, movimientos) se superponen según sea necesario, con limitación de frecuencia para evitar saturación.

Un detalle de implementación interesante es el cooldown en efectos de movimiento. En el Nivel1, el sonido de pasos se reproduce con un pequeño retraso entre repeticiones, imitando el ritmo natural de caminar en lugar de un sonido continuo. Estas pequeñas atenciones al detalle contribuyen significativamente a la inmersión del jugador.

4. MECÁNICAS DEL JUEGO

4.1 Contexto histórico

El juego se inspira en uno de los eventos más significativos de la historia: la Caída de Constantinopla en 1453. Desde la perspectiva de los defensores bizantinos, cada nivel representa una fase diferente del asedio que duró 53 días. Los jugadores encarnan a un defensor anónimo de la ciudad, experimentando la evolución de la batalla desde las primeras escaramuzas hasta la caída final de las murallas.

Constantinopla, la capital del Imperio Bizantino, fue asediada por el ejército otomano del Sultán Mehmed II. A pesar de estar enormemente superados en número, los defensores, liderados por el Emperador Constantino XI, resistieron heroicamente utilizando ingeniosas tácticas defensivas, tecnología militar avanzada y un profundo conocimiento del terreno.

4.2 Nivel 1

Contexto histórico: Los primeros días del asedio, donde los defensores repelían oleadas de ataques otomanos mientras intentaban reparar brechas en las murallas y mantener la moral alta.

Objetivo del nivel: Sobrevivir 60 segundos (representando las primeras semanas del asedio) mientras se defienden las murallas de ataques constantes.

4.3 Nivel 2

Contexto histórico: La fase de bombardeo intensivo donde los otomanos utilizaron enormes cañones (como el "Basílica" de Orbán) para derribar secciones de las murallas. Los defensores debían esquivar escombros y proyectiles mientras intentaban reparar los daños.

Objetivo del nivel: Esquivar barriles (que representan proyectiles y escombros) durante 60 segundos mientras se mantiene una posición defensiva.

4.4 Nivel 3

Contexto histórico: Los momentos finales del asedio, cuando los otomanos rompieron las murallas por la puerta de San Romano. Los defensores supervivientes intentaban retirarse ordenadamente mientras las calles se llenaban de obstáculos y los otomanos penetraban en la ciudad.

Objetivo del nivel: Avanzar lo máximo posible (medido en distancia) mientras se esquivan obstáculos en un ambiente de scroll automático.

5. IMPLEMENTACIÓN TÉCNICA

5.1 Sistema de Ciclo y Temporización

El corazón técnico del juego reside en su sistema de ciclo de actualización, implementado mediante la clase “NivelBase”. Cada nivel utiliza un “Qtimer” configurado para disparar aproximadamente 60 veces por segundo (intervalo de 16ms), lo que establece una tasa de fotogramas objetivo de 60 FPS. Este temporizador está conectado al slot “onTimerTimeout()”, que calcula el tiempo transcurrido desde el último frame (deltaTime) y lo pasa al método abstracto “actualizarJuego(float deltaTime)”.

La implementación del cálculo de deltaTime es crucial para garantizar movimientos suaves independientemente de las fluctuaciones en el rendimiento del sistema. En lugar de depender de intervalos fijos, se mide el tiempo real transcurrido:

```
void NivelBase::onTimerTimeout()
{
    qint64 tiempoActual = QDateTime::currentMSecsSinceEpoch();
    float deltaTime = tiempoActual - tiempoUltimoFrame;
    tiempoUltimoFrame = tiempoActual;
    actualizarJuego(deltaTime);
    update();
}
```

Este enfoque basado en tiempo real permite que el juego mantenga una física consistente incluso si la tasa de fotogramas varía. Los movimientos se calculan multiplicando las velocidades por deltaTime, asegurando que un objeto se mueva la misma distancia física en el mismo tiempo real, independientemente del rendimiento del hardware.

5.2 Sistema de colisiones

El sistema de colisiones implementa un enfoque jerárquico que equilibra precisión y rendimiento.

Antes de procesar colisiones, el sistema verifica si una entidad está dentro del área visible de la cámara utilizando el método estaEnVista(). Esto reduce el número de chequeos necesarios, especialmente en el Nivel 1 con su mapa grande.

Cada entidad implementa el método getAreaColision() que retorna un QRectF representando su área de colisión. Para colisiones entre entidades, se utiliza el método intersects() de Qt

5.3 Sistema de Físicas

1. Física de Proyectiles (Nivel 1):

Las armas como la ballesta y el arco implementan trayectorias de proyectiles básicas. Aunque simplificadas para el gameplay, utilizan principios físicos reales:

```
void Arma::actualizarProyectiles(float deltaTime)
{
    float velocidadBase = 0.12f;
    float velocidad = velocidadBase * (1.0f + (nivel - 1) * 0.08f);

    for(int i = 0; i < proyectiles.size(); i++) {
        proyectiles[i] += direccionesProyectiles[i] * velocidad * deltaTime;
        tiemposVidaProyectiles[i] -= deltaTime;
    }

    for(int i = proyectiles.size() - 1; i >= 0; i--) {
        if(tiemposVidaProyectiles[i] <= 0) {
            proyectiles.removeAt(i);
            direccionesProyectiles.removeAt(i);
            tiemposVidaProyectiles.removeAt(i);
        }
    }
}
```

2. Física de Movimiento Oscilatorio (Nivel 2):

Los barriles de tipo 2 implementan movimiento senoidal, utilizando las funciones trigonométricas de Qt:

```
case 2:
    posicion.setY(posicion.y() + velocidad);
    posicion.setX(posicion.x() + amplitud * qSin(frecuencia * tiempoVida + fase) * 0.1f);
    break;
```

Los parámetros amplitud, frecuencia y fase se generan aleatoriamente para cada barril, creando patrones de movimiento únicos que el jugador debe anticipar.

3. Física de Gravedad y Salto (Nivel 3):

El sistema más complejo se implementa en JugadorNivel3, que utiliza ecuaciones físicas reales para el movimiento vertical:

```cpp
void JugadorNivel3::actualizar(float deltaTime)
{
    float deltaSec = deltaTime / 1000.0f;

    static int debugCounter = 0;
    if (debugCounter++ % 60 == 0) {
        qDebug() << "Jugador actualizando - Saltando:" << estaSaltando
                 << "GravedadAumentada:" << gravedadAumentada
                 << "VelVertical:" << velocidadVertical
                 << "PosY:" << posicion.y();
    }

    if (!estaEnSuelo()) {
        velocidadVertical += gravedad * deltaSec;
    }

    posicion.setY(posicion.y() + velocidadVertical * deltaSec);

    if (posicion.y() >= 540.0f) {
        posicion.setY(540.0f);
        velocidadVertical = 0.0f;
        estaSaltando = false;
        tiempoSalto = 0.0f;
        gravedadAumentada = false;
        gravedad = gravedadNormal;
```

El cálculo del salto utiliza física : velocidadVertical = -sqrt(2.0f * gravedadNormal * alturaSalto). Esta implementación permite un control del salto y facilita la implementación de mecánicas como la cancelación de salto.

5.4 Sistema de Animación

Qt proporciona QPainter en un QWidget, eliminando el parpadeo durante el renderizado. En lugar de cargar frames individuales, se utilizan spritesheets que contienen todas las frames de una animación en una sola imagen. El SpriteManager extrae frames individuales mediante cálculos de coordenadas:

```cpp
QPixmap SpriteManager::getSpriteFrame(const QString& sheetName, int frameIndex) const
{
    if(!spriteSheets.contains(sheetName)) {
        return QPixmap();
    }

    QPixmap sheet = spriteSheets[sheetName];
    QSize frameSize = spriteSheetFrameSizes[sheetName];

    int framesPerRow = sheet.width() / frameSize.width();

    int row = frameIndex / framesPerRow;
    int col = frameIndex % framesPerRow;

    QRect frameRect(col * frameSize.width(),
                    row * frameSize.height(),
                    frameSize.width(),
                    frameSize.height());

    return sheet.copy(frameRect);
}
```

5.5 Sistema de Audio

El AudioManager implementa un sistema de audio robusto que cumple con el requisito investigativo de sonido de fondo y efectos.

Los sonidos se cargan al inicio del juego mediante loadSounds(). El sistema diferencia entre música de fondo (que se reproduce en loop) y efectos de sonido (que se reproducen una vez cuando ocurre un evento). Para evitar saturación auditiva, efectos como los pasos del jugador implementan un cooldown:

```
static int moveSoundCooldown = 0;
bool isMoving = (teclas[0] || teclas[1] || teclas[2] || teclas[3]);

if (isMoving && moveSoundCooldown <= 0) {
    AudioManager::getInstance().playPlayerMove();
    moveSoundCooldown = 30;
}
```

5.6 Sistema de Input

El manejo de input implementa un sistema basado en estado que permite respuestas fluidas:

Almacenamiento de estado de teclas: En lugar de procesar eventos de teclado directamente en la lógica del juego, se mantiene un vector de booleanos que representa el estado actual de las teclas relevantes:

```
void NivelBase::keyPressEvent(QKeyEvent *event)
{
    switch(event->key()) {
    case Qt::Key_W: teclas[0] = true; break;
    case Qt::Key_A: teclas[1] = true; break;
    case Qt::Key_S: teclas[2] = true; break;
    case Qt::Key_D: teclas[3] = true; break;
    case Qt::Key_P: pausarNivel(); break;
    case Qt::Key_R: reanudarNivel(); break;
    }
}

void NivelBase::keyReleaseEvent(QKeyEvent *event)
{
    switch(event->key()) {
    case Qt::Key_W: teclas[0] = false; break;
    case Qt::Key_A: teclas[1] = false; break;
    case Qt::Key_S: teclas[2] = false; break;
    case Qt::Key_D: teclas[3] = false; break;
    }
}
```

6. PRUEBAS Y CAMBIOS

6.1 Pruebas

El proceso de pruebas siguió una estrategia que combinó pruebas manuales sistemáticas, verificación de requisitos y pruebas de usabilidad. Dada la naturaleza del proyecto como un videojuego, se priorizó la experiencia del jugador final mientras se aseguraba el cumplimiento técnico de todos los requisitos del curso.

Enfoque de pruebas por niveles: Cada nivel se probó independientemente antes de integrarse en el juego completo, permitiendo identificar y corregir problemas específicos sin afectar otros componentes.

Pruebas: El desarrollo siguió un ciclo de implementación-prueba-ajuste, con especial atención a la retroalimentación sobre la jugabilidad y dificultad.

6.2 Cambios

El desarrollo del juego estuvo marcado por constantes ajustes. En retrospectiva, comparando el resultado final con los planteamientos iniciales de los momentos I y II, lo único que se mantuvo fue la temática general y el diseño del primer nivel. Los niveles 2 y 3 sufrieron modificaciones debido al tiempo que demandó el desarrollo del nivel 1 y a la necesidad de gestionar adecuadamente los plazos del proyecto. Por ello, se optó por simplificar su complejidad respecto a lo planeado originalmente.