DOCUMENTACIÓN PROYECTO FINAL

MOISES SANTOS HDZ

CARRERA: INGENIRIA EN SISTEMAS INTELIGENTES

GENERACION: 2022

CLAVE UASLP:356850

MATERIA: TECNOLOGIA ORIENTADA A OBJETOS

02/12/2025

INTRODUCCIÓN Y CONTEXTO DEL JUEGO

MECÁNICAS PRINCIPALES (UN JUGADOR)

1. FLUJO DE CARGA DEL MAPA Y TEMPORIZADOR
2. REGENERACIÓN DEL MAPA EN CADA PARTIDA
3. SISTEMA DE TEMPORIZADOR DE 3 MINUTOS
4. MOVIMIENTO DEL JUGADOR
5. INTELIGENCIA ARTIFICIAL Y MOVIMIENTO DE ENEMIGOS
6. SISTEMA DE COLISIONES
7. SISTEMA DE RENDERIZADO DETALLADO

SISTEMA MULTIJUGADOR

8. VISIÓN GENERAL DEL SISTEMA MULTIJUGADOR
9. COMUNICACIÓN EN TIEMPO REAL
10. SINCRONIZACIÓN DEL MAPA ENTRE JUGADORES
11. SINCRONIZACIÓN DE MOVIMIENTO Y DATOS DE JUGADORES
12. PANTALLA FINAL Y DETERMINACIÓN DE GANADOR

ARQUITECTURA Y DESPLIEGUE

13. DIAGRAMA DE CLASES (UML)
14. PROCESO DE COMPILACIÓN Y GENERACIÓN DE EJECUTABLE
15. SOLUCIÓN A PROBLEMAS DE CARGA DE RECURSOS (ASSETS)
16. CORRECCIONES RELEVANTES (BUG FIXES)
17. MANIFIESTO DE LA APLICACIÓN

FLUJO DE PANTALLAS Y MENUS

Este documento describe en detalle el funcionamiento, codigo y logica de las pantallas de interfaz, corrigiendo la version anterior para reflejar el flujo real y las opciones especificas del juego.

1. PANTALLA DE BIENVENIDA

Es la primera pantalla que se muestra al ejecutar el juego. Su proposito es presentar el titulo y esperar a que el jugador continue hacia el menu principal.

1.1. Funcionamiento General

El juego se inicia en un estado WELCOME. En este estado, la pantalla es estatica y solo muestra un mensaje de bienvenida. El juego no procesa ninguna logica de juego (movimiento, enemigos, etc.), solo espera una unica entrada del teclado (ENTER) para cambiar al siguiente estado, que es el menu principal.

1.2. Archivos y Codigo Involucrados

Archivo: Main/GamePanel.java

Contiene el enum GameState que define este estado y la logica para dibujarlo en paintComponent.

Archivo: Main/ManejadorTeclas.java

Detecta la pulsacion de ENTER para avanzar al siguiente estado.

1.3. Logica de Estado y Renderizado

Se necesita un GameState mas especifico para diferenciar la bienvenida del menu.

Archivo: Main/GamePanel.java

Codigo de Estado:

Dentro de la clase GamePanel

```
public enum GameState {
    WELCOME, // Nuevo estado para la pantalla de bienvenida
    MENU,  Estado para el menu de opciones
    PLAY,
    PAUSE
}
public GameState gameState;
```

El constructor ahora inicia en el estado WELCOME

```
public GamePanel() {
  ...
    this.gameState  GameState.WELCOME;
}
```

El metodo paintComponent gestiona que dibujar

```
@Override
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2  (Graphics2D) g;

    if (gameState  GameState.WELCOME) {
```

```
        drawWelcomeScreen(g2);
    } else if (gameState  GameState.MENU) {
        drawMenuScreen(g2);
    } // ... etc.
}
```

Metodo especifico para dibujar la pantalla de bienvenida

```
public void drawWelcomeScreen(Graphics2D g2) {
    g2.setColor(Color.BLACK);
    g2.fillRect(0, 0, screenWidth, screenHeight);
    g2.setFont(g2.getFont().deriveFont(Font.BOLD, 70F));
    String text  "Bienvenido a la Aventura";
    int x  getXforCenteredText(text, g2);
    int y  screenHeight / 2 - (tileSize  2);
    g2.setColor(Color.WHITE);
    g2.drawString(text, x, y);

    g2.setFont(g2.getFont().deriveFont(Font.PLAIN, 40F));
    text  "Presiona ENTER para continuar";
    x  getXforCenteredText(text, g2);
    y + tileSize  2;
    g2.drawString(text, x, y);
}
```

1.4. Manejo de Teclas

En este estado, solo se escucha la tecla ENTER.

Archivo: Main/ManejadorTeclas.java

Codigo de Manejo de Teclas:

Dentro de keyPressed

```
 if (gp.gameState  GameState.WELCOME) {
    if (code  KeyEvent.VK_ENTER) {
        gp.gameState  GameState.MENU; // Transicion al menu principal
    }
 }
```

2. PANTALLA DE MENU PRINCIPAL

Esta pantalla aparece despues de la bienvenida y es el centro de navegacion principal antes de jugar.

2.1. Funcionamiento General

En el estado MENU, el jugador puede usar las teclas de arriba y abajo para seleccionar una opcion y ENTER para confirmarla. Las opciones son "Nuevo Juego" y "Salir".

2.2. Logica de Renderizado

El metodo drawMenuScreen (llamado desde paintComponent) se encarga de dibujar las opciones y un selector > que indica la opcion activa, cuya posicion depende de la variable commandNum.

Archivo: Main/GamePanel.java
Codigo de Renderizado: (Similar al de la version anterior, pero llamado en el estado MENU)

2.3. Manejo de Teclas y Opciones

Archivo: Main/ManejadorTeclas.java
Codigo de Manejo de Teclas:

Dentro de keyPressed

```
else if (gp.gameState  GameState.MENU) {
    if (code  KeyEvent.VK_W || code  KeyEvent.VK_UP) {
        gp.commandNum--;
```

```
        if (gp.commandNum < 0) { gp.commandNum  1; }
    }
    if (code  KeyEvent.VK_S || code  KeyEvent.VK_DOWN) {
        gp.commandNum++;
        if (gp.commandNum > 1) { gp.commandNum  0; }
    }
    if (code  KeyEvent.VK_ENTER) {
        switch (gp.commandNum) {
            case 0: // Opcion "NUEVO JUEGO"
                gp.gameState  GameState.PLAY;
              Opcional: llamar a un metodo gp.setupGame() para
              inicializar todo.
                break;
            case 1: // Opcion "SALIR"
                System.exit(0);
                break;
        }
    }
}
```

3. PANTALLA DE PAUSA (TECLA ESC)

Esta pantalla se activa al presionar ESC durante el juego (PLAY state), congelando la accion y mostrando un menu con varias opciones.

3.1. Funcionamiento General

Al cambiar al estado PAUSE, el bucle de actualizacion del juego se salta, pero el de renderizado continua. Se dibuja el menu de pausa sobre la pantalla del juego congelada. El jugador navega por este menu de forma similar al menu principal.

3.2. Logica de Renderizado

Archivo: Main/GamePanel.java
Codigo de Renderizado:

```
En paintComponent
  else if (gameState  GameState.PAUSE) {
    Es importante que la logica de renderizado del juego se ejecute
    para que la pantalla de pausa se muestre sobre el juego congelado.
    ... dibujar el juego ...
      drawPauseScreen(g2); // Dibujar el menu de pausa encima
  }

  public void drawPauseScreen(Graphics2D g2) {
    Fondo semitransparente para oscurecer el juego
      g2.setColor(new Color(0, 0, 0, 150));
      g2.fillRect(0, 0, screenWidth, screenHeight);
```

```
        g2.setFont(g2.getFont().deriveFont(Font.BOLD, 50F));
        g2.setColor(Color.WHITE);

      Opciones del menu de pausa
        String options  {"Reanudar", "Reiniciar", "Menu Principal", "Salir"};
        int y  screenHeight / 2 - (tileSize  2);

        for (int i  0; i < options.length; i++) {
            String text  optionsi;
            int x  getXforCenteredText(text, g2);
            g2.drawString(text, x, y + (i  tileSize  2));
            if (commandNum  i) {
                g2.drawString(">", x - tileSize, y + (i  tileSize  2));
            }
        }
    }
```

3.3. Manejo de Teclas y Opciones

Archivo: Main/ManejadorTeclas.java

Codigo de Manejo de Teclas:

```
 Dentro de keyPressed
    else if (gp.gameState  GameState.PAUSE) {
      Navegacion
        if (code  KeyEvent.VK_W || code  KeyEvent.VK_UP) {
            gp.commandNum--;
            if (gp.commandNum < 0) { gp.commandNum  3; } // 4 opciones (0-3)
        }
        if (code  KeyEvent.VK_S || code  KeyEvent.VK_DOWN) {
```

```
            gp.commandNum++;
            if (gp.commandNum > 3) { gp.commandNum  0; }
        }
        if (code  KeyEvent.VK_ENTER) {
            switch (gp.commandNum) {
                case 0: // Reanudar
                    gp.gameState  GameState.PLAY;
                    break;
                case 1: // Reiniciar
                  Llamada al metodo de reinicio del GameEngine
                    gp.getGameEngine().reiniciarJuego();
                    gp.gameState  GameState.PLAY;
                    break;
                case 2: // Menu Principal
                    gp.gameState  GameState.MENU;
                    gp.commandNum  0; // Resetea el cursor del menu principal
                    break;
                case 3: // Salir
                    System.exit(0);
                    break;
            }
        }
      Tambien permitir salir de la pausa con ESC
        if (code  KeyEvent.VK_ESCAPE) {
            gp.gameState  GameState.PLAY;
        }
    }
```

4. IMPLEMENTACION COMPLETA DE LA FUNCION REINICIAR JUEGO

Esta seccion detalla la implementacion completa de la funcionalidad de reiniciar el juego desde el menu de pausa, incluyendo el reinicio de todas las estadisticas del jugador y el estado del mundo.

4.1. Funcionamiento General

La opcion "Reiniciar" (opcion 2 en el menu de pausa) permite al jugador resetear completamente el juego a su estado inicial sin tener que cerrar y volver a abrir la aplicacion. Esto incluye:

- Reiniciar todas las estadisticas del jugador
- Regenerar el mundo (chunks y tiles)
- Limpiar y regenerar enemigos
- Limpiar items consumibles
- Resetear la posicion de la camara

4.2. Arquitectura de la Solucion

La funcionalidad de reinicio se implementa siguiendo la arquitectura del juego:

1. GamePanel.java (Main): Maneja el estado del menu de pausa y llama al metodo de reinicio
2. GameEngine.java (Domain): Contiene la logica central de reinicio
3. JugadorSystem.java (Domain): Reinicia estadisticas especificas del jugador
4. ManejadorMapaInfinito.java (Domain): Reinicia chunks y items del mapa

4.3. Implementacion en GamePanel

4.3.1. Flujo de Control en el Menu de Pausa

Archivo: Main/GamePanel.java

Metodo: actualizar() en el case MENU_PAUSA

```
  case MENU_PAUSA:
      if (inputService.isTecla1()) {
        Reanudar juego
          estadoJuego  GameState.JUGANDO;
          inputService.setTecla1(false);
      } else if (inputService.isTecla2()) {
         REINICIAR JUEGO
          gameEngine.reiniciarJuego();
          estadoJuego  GameState.JUGANDO;
          inputService.setTecla2(false);
      } else if (inputService.isTecla3()) {
        Volver al menu principal
          estadoJuego  GameState.MENU_PRINCIPAL;
          inputService.setTecla3(false);
      } else if (inputService.isTeclaEscape()) {
        Volver al juego (reanudar)
          estadoJuego  GameState.JUGANDO;
          inputService.setTeclaEscape(false);
      }
      break;
```

Nota: En esta implementacion, el menu de pausa usa teclas numericas

(1, 2, 3) para seleccionar opciones directamente, en lugar de un cursor de navegacion.

4.4. Implementacion del Metodo reiniciarJuego()

4.4.1. Archivo: domain/GameEngine.java

El metodo reiniciarJuego() es el corazon de la funcionalidad de reinicio. Se encarga de coordinar el reinicio de todos los sistemas del juego.

```
public void reiniciarJuego() {
  1. REINICIAR ESTADISTICAS DEL JUGADOR

  Vida maxima
   jugadorSystem.getJugador().setVida(100);

  Arsenal de pociones
   jugadorSystem.setPocionesEnArsenal(0);

  Acertijos resueltos
   jugadorSystem.setAcertijosResueltos(0);

  Cooldown de danio (evita danio inmediato despues del reinicio)
   jugadorSystem.reiniciarCooldownDanio();
```

2. REINICIAR POSICION DEL JUGADOR

Volver a la posicion inicial del mundo

jugadorSystem.getMovimientoSystem().setMundoX(5000);

jugadorSystem.getMovimientoSystem().setMundoY(5000);

3. REINICIAR ESTADOS DE MODALES Y COFRES

Cerrar todos los modales activos

modalCofreActivo false;

modalRespuestaCorrectaActivo false;

modalRespuestaIncorrectaActivo false;

Resetear estados de cofres

cofreYaActivado false;

cofreBloqueado false;

acertijoActual null;

cofreTileX -1;

cofreTileY -1;

4. REINICIAR ENEMIGOS

Limpiar lista de enemigos existentes

```
enemigoSystem.getEnemigos().clear();
```

Generar nueva poblacion de enemigos alrededor del jugador

```
enemigoSystem.generarEnemigosIniciales(
    jugadorSystem.getMundoX(),
    jugadorSystem.getMundoY(),
    5  // Cantidad inicial de enemigos
);
```

5. REINICIAR ITEMS CONSUMIBLES

Eliminar todos los items (pociones, venenos) del mapa

```
mapaInfinito.getItemsConsumibles().clear();
```

6. REINICIAR CHUNKS DEL MAPA

Limpiar chunks cargados para forzar regeneracion

```
mapaInfinito.reiniciarChunks();
```

7. ACTUALIZAR CAMARA Y GENERAR MUNDO

Centrar camara en la nueva posicion del jugador

```
camaraSystem.seguirEntidad(
```

```
            jugadorSystem.getMundoX(),
            jugadorSystem.getMundoY()
        );

    Generar chunks alrededor de la posicion inicial
        if (mapaAdapter ! null) {
            mapaAdapter.actualizarChunksActivos(
                jugadorSystem.getMundoX(),
                jugadorSystem.getMundoY()
            );
        } else if (mapaInfinito ! null) {
            mapaInfinito.actualizarChunksActivos(
                jugadorSystem.getMundoX(),
                jugadorSystem.getMundoY()
            );
        }
```

8. REACTIVAR EL JUEGO

```
        jugando  true;
    }
```

4.5. Metodos Auxiliares Implementados

4.5.1. Metodo reiniciarCooldownDanio() en JugadorSystem

Este metodo reinicia el temporizador que controla cuando el jugador puede recibir danio nuevamente.

Archivo: domain/JugadorSystem.java

```
  public void reiniciarCooldownDanio() {
      this.ultimoTiempoDanio  0;
  }
```

Proposito: Evitar que el jugador reciba danio inmediatamente despues de reiniciar si hay un enemigo cerca. El cooldown se resetea a 0, permitiendo un nuevo ciclo de danio desde el inicio.

4.5.2. Metodo reiniciarChunks() en ManejadorMapaInfinito

Este metodo limpia todos los chunks del mapa y los items consumibles, forzando la regeneracion del mundo.

Archivo: domain/ManejadorMapaInfinito.java

```
  public void reiniciarChunks() {
    Limpiar HashMap de chunks activos
      chunksActivos.clear();

    Limpiar lista de items consumibles
      itemsConsumibles.clear();
  }
```

Proposito:

- chunksActivos.clear(): Elimina todos los chunks generados del HashMap. Cuando el jugador se mueva despues del reinicio, los chunks se regeneraran proceduralmente con el seed del mundo.

- itemsConsumibles.clear(): Elimina todas las pociones y venenos que el jugador no recogio. Nuevos items se generaran en los nuevos chunks.

4.6. Flujo Completo de Ejecucion

1. Jugador presiona ESC durante el juego

2. GamePanel cambia estado a MENU_PAUSA

3. Se renderiza el menu con opciones:
   1 Reanudar
   2 Reiniciar (Opcion de reinicio)
   3 Menu Principal
   ESC Volver

4. Jugador presiona tecla 2

5. GamePanel llama a gameEngine.reiniciarJuego()

6. GameEngine ejecuta secuencia de reinicio:
   a. Reinicia estadisticas del jugador
   b. Resetea posicion del jugador
   c. Cierra modales activos

d. Limpia y regenera enemigos

e. Limpia items del mapa

f. Reinicia chunks del mundo

g. Actualiza camara

h. Genera nuevo entorno

7. GamePanel cambia estado a JUGANDO

8. El juego continua desde el estado inicial

4.7. Elementos que se Mantienen

Algunos elementos del juego NO se reinician, manteniendo la continuidad de la sesion:

- Seed del mundo: El mundo se regenera con el mismo seed, creando el mismo terreno
- Configuracion del juego: FPS, tamanio de tiles, dimensiones de pantalla
- Teclas de control: Las configuraciones de InputService permanecen
- Ventana del juego: La ventana JFrame no se cierra ni se recrea

4.8. Elementos que SI se Reinician

Lista completa de elementos que vuelven a su estado inicial:

Elemento	Valor Inicial	Ubicacion
Vida del jugador	100	JugadorSystem.jugador.vida
Pociones en arsenal	0	JugadorSystem.pocionesEnArsenal
Acertijos resueltos	0	JugadorSystem.acertijosResueltos
Cooldown de danio	0	JugadorSystem.ultimoTiempoDanio
Posicion X del jugador	5000	MovimientoSystem.mundoX
Posicion Y del jugador	5000	MovimientoSystem.mundoY
Lista de enemigos	Vacia (5 nuevos)	EnemigoSystem.enemigos
Items consumibles	Vacia	MapaInfinito.itemsConsumibles
Chunks del mapa	Vacio (Regenerados)	MapaInfinito.chunksActivos
Modal de cofre	false	GameEngine.modalCofreActivo
Modal respuesta correcta	false	GameEngine.modalRespuestaCorrectaActivo
Modal respuesta incorrecta	false	GameEngine.modalRespuestaIncorrectaActivo
Acertijo actual	null	GameEngine.acertijoActual
Estado cofre activado	false	GameEngine.cofreYaActivado
Estado cofre bloqueado	false	GameEngine.cofreBloqueado

4.9. Consideraciones de Implementacion

4.9.1. Orden de Reinicio

El orden en que se reinician los componentes es importante:

1. Primero: Estadisticas del jugador (no dependen de nada)
2. Segundo: Posicion del jugador (necesaria para regenerar enemigos)
3. Tercero: Estados de UI (modales)
4. Cuarto: Enemigos (necesitan posicion del jugador)
5. Quinto: Mapa e items (regeneracion completa)
6. Sexto: Camara (necesita posicion del jugador)
7. Ultimo: Reactivar el juego

4.9.2. Regeneracion Procedural

Gracias a que el mundo usa generacion procedural con seed:

- Los chunks se regeneraran con el mismo terreno
- Los cofres apareceran en las mismas posiciones
- Los acertijos de los cofres se pueden resolver nuevamente
- Los items se regeneraran en nuevas posiciones aleatorias

4.9.3. Prevencion de Bugs

La implementacion incluye varias medidas para prevenir bugs:

- Reseteo del cooldown: Evita danio instantaneo al reiniciar
- Limpieza de enemigos antes de regenerar: Previene duplicados
- Limpieza de chunks antes de actualizar: Fuerza regeneracion completa
- Cierre de modales: Evita que modales antiguos se muestren
- Reseteo de flags de cofres: Permite reinteractuar con cofres

4.10. Pruebas y Validacion

Para verificar que el reinicio funciona correctamente:

1. Iniciar juego y recoger pociones
2. Recibir danio de enemigos
3. Resolver acertijos
4. Presionar ESC y seleccionar Reiniciar
5. Verificar que la vida vuelve a 100
6. Verificar que las pociones vuelven a 0
7. Verificar que los acertijos resueltos vuelven a 0
8. Verificar que la posicion del jugador es (5000, 5000)
9. Verificar que hay nuevos enemigos
10. Verificar que los cofres se pueden abrir nuevamente

MENU COMANDOS DEL JUEGO

DESCRIPCION GENERAL

Se ha agregado una nueva opcion al menu principal que permite a los jugadores visualizar todos los comandos disponibles en el juego. Esta funcionalidad proporciona un acceso rapido a la documentacion de controles sin necesidad de consultar manuales externos.

NAVEGACION DEL MENU

En el menu principal aparecen las siguientes opciones:

[1] Jugar Solo

[2] Crear Servidor

[3] Unirse a Servidor

[4] Comandos del Juego

[5] Salir

Al presionar la tecla 4, el jugador accede a la pantalla de Comandos del Juego. Desde esta pantalla, puede presionar ESC para volver al menu principal.

IMPLEMENTACION TECNICA

1. Estado del Juego

Se agrego un nuevo estado en GameState enum:

```
public enum GameState {
    MENU_PRINCIPAL,
    CREAR_SERVIDOR,
    UNIRSE_SERVIDOR,
    MENU_COMANDOS,
    JUGANDO,
    MENU_PAUSA,
    JUEGO_TERMINADO
}
```

2. Manejo de Entrada

En GamePanel.java, la logica de entrada del menu principal se actualizo:

```
case MENU_PRINCIPAL:
    if (inputService.isTecla1()) {
        estadoJuego  GameState.JUGANDO;
        inputService.setTecla1(false);
    } else if (inputService.isTecla2()) {
        estadoJuego  GameState.CREAR_SERVIDOR;
        resetearInputs();
        inputService.setTecla2(false);
    } else if (inputService.isTecla3()) {
        estadoJuego  GameState.UNIRSE_SERVIDOR;
```

```
        resetearInputs();
        inputService.setTecla3(false);
    } else if (inputService.isTecla4()) {
        estadoJuego  GameState.MENU_COMANDOS;
        inputService.setTecla4(false);
    } else if (inputService.isTecla5()) {
        System.exit(0);
    }
    break;
```

Cuando el jugador esta en el menu de comandos:

```
case MENU_COMANDOS:
    if (inputService.isTeclaEscape()) {
        estadoJuego  GameState.MENU_PRINCIPAL;
        inputService.setTeclaEscape(false);
    }
    break;
```

3. Renderizado

En GamePanel.java, la logica de renderizado se actualizo para mostrar el menu de comandos:

```
case MENU_COMANDOS:
    renderSystem.renderMenuComandos(g2d, pantallaAncho, pantallaAlto);
    break;
```

4. Renderizador de Menu Comandos

Se implemento el metodo renderMenuComandos() en HUDRenderer.java:

```
public void renderMenuComandos(Graphics2D g2, int pantallaAncho, int pantallaAlto) {
    g2.setColor(new Color(0, 0, 0, 180));
    g2.fillRect(0, 0, pantallaAncho, pantallaAlto);

    g2.setFont(fuenteTitulo);
    g2.setColor(new Color(255, 215, 0));
    String titulo  "COMANDOS DEL JUEGO";
    int anchoTitulo  g2.getFontMetrics().stringWidth(titulo);
    g2.drawString(titulo, (pantallaAncho - anchoTitulo) / 2, 50);

    g2.setFont(fuenteNormal);
    g2.setColor(Color.WHITE);

    String[] comandos  {
        "MOVIMIENTO:",
        "  flechas o WASD    Mover personaje",
        "  Q    Tomar una pocion",
         "COMBATE Y ACCION:",
        "  E    Interactuar con cofres",
        "  A-D    Seleccionar respuesta",
        "  ENTER    Confirmar respuesta",
        "",
        "POCIONES:",
        "  R    Usar pocion (cura 40 HP)",
        "",
        "CONTROL DE JUEGO:",
```

```
    "  P    Pausar_Reanudar",
    "  ESC    Menu de pausa",
    "  ENTER    Confirmar acciones",
    "",
    "EN MULTIJUGADOR:",
    "  Todos los cambios se sincronizan en tiempo real",
    "  Ver HUD para estadisticas de otros jugadores"
};

int y  120;
int lineHeight  25;

for (String comando : comandos) {
    if (comando.isEmpty()) {
        y + 10;
    } else if (comando.endsWith(":")) {
        g2.setColor(new Color(100, 200, 255));
        g2.drawString(comando, 100, y);
        g2.setColor(Color.WHITE);
        y + lineHeight;
    } else {
        g2.drawString(comando, 100, y);
        y + lineHeight;
    }
}

g2.setFont(fuenteNormal);
g2.setColor(Color.YELLOW);
String regreso  "[ESC] Volver al Menu Principal";
int anchoRegreso  g2.getFontMetrics().stringWidth(regreso);
```

```
    g2.drawString(regreso, (pantallaAncho - anchoRegreso) / 2, pantallaAlto - 40);
}
```

5. Servicios de Entrada

Se agregaron metodos en InputService.java para soportar la tecla 5:

```
private boolean tecla5;
```

En keyPressed():

```
case KeyEvent.VK_5:
    tecla5  true;
    break;
```

En keyReleased():

```
case KeyEvent.VK_5:
    tecla5  false;
    break;
```

Metodos getter y setter:

```
public boolean isTecla5() {
    return tecla5;
}

public void setTecla5(boolean estado) {
    this.tecla5  estado;
}
```

CONTENIDO DEL MENU

El menu de comandos muestra 5 secciones principales:

1. MOVIMIENTO

. Teclas direccionales o WASD para mover el personaje

. Tecla Q para tomar una pocion

2. COMBATE Y ACCION

. Tecla E para interactuar con cofres

. Teclas A y D para seleccionar respuestas de acertijos

. ENTER para confirmar

3. POCIONES

. Tecla R para usar pocion (restaura 40 HP)

4. CONTROL DE JUEGO

. Tecla P para pausar_reanudar

. ESC para abrir menu de pausa

. ENTER para confirmar acciones

5. EN MULTIJUGADOR

. Informacion sobre sincronizacion en tiempo real

. Referencia al HUD para ver estadisticas

FLUJO DE NAVEGACION

Menu Principal

Menu Comandos del Juego

Menu Principal

FLUJO DE CARGA DEL MAPA Y TIMER

1. RESPUESTA DIRECTA

El mapa se carga cuando inicia la aplicación, pero de forma inteligente y progresiva:

1. Al iniciar la aplicación: Se cargan las imágenes de los tiles (texturas)
2. Al crear GameEngine: Se generan los chunks iniciales alrededor del jugador
3. Durante el juego: Se generan nuevos chunks a medida que el jugador explora

El timer ahora inicia cuando el jugador comienza a jugar (cuando selecciona "Jugar Solo" y el juego está en estado JUGANDO).

2. FLUJO DETALLADO DE INICIALIZACION

2.1 INICIO DE LA APLICACION (Main.java)

```
    public static void main(String args) {
        // ... configuración ...

        // PASO 1: Crear el mapa
        MapaInfinitoAdapter mapaAdapter  new MapaInfinitoAdapter(
            config.getTamanioTile(),
            config.getAnchoPantalla(),
            config.getAltoPantalla()
        );

        // PASO 2: Crear sistemas del juego
```

```
// PASO 3: Crear GameEngine
    GameEngine gameEngine  new GameEngine(
        config,
        jugadorSystem,
        camaraSystem,
        mapaAdapter,
        enemigoSystem,
        inputService
    );
}
```

Qué sucede:

- Se crea el objeto MapaInfinitoAdapter
- Se cargan las imágenes de los tiles (pasto, agua, árbol, muro, etc.)
- Se inicializa el sistema de chunks (vacío por ahora)
- NO se genera ningún chunk todavía

2.2 CREACION DEL MAPA INFINITO (ManejadorMapaInfinito.java)

```
public ManejadorMapaInfinito(int tamanioTile, int anchoPantalla, int altoPantalla, long seed) {
    this.tamanioTile  tamanioTile;
    this.anchoPantalla  anchoPantalla;
    this.altoPantalla  altoPantalla;
    this.chunksActivos  new HashMap<>();  // HashMap vacío
    this.generador  new GeneradorMundo(seed, this);
    this.itemsConsumibles  new ArrayList<>();

    cargarTilesVisuales();  // Carga imágenes PNG de los tiles
}
```

```
    private void cargarTilesVisuales() {
        tiles  new Tile11;

        tiles0  new Tile();
        tiles0.setImage(ImageIO.read(new File("tiles/agua.png")));

        tiles1  new Tile();
        tiles1.setImage(ImageIO.read(new File("tiles/arbol.png")));

        // ... etc para todos los tiles ...
    }
```

Qué sucede:

- Se cargan en memoria las texturas/imágenes de todos los tipos de tile
- Se crea el GeneradorMundo con un seed (para generación procedural)
- El HashMap de chunks está vacío (sin chunks generados)

2.3 INICIALIZACION DEL GAME ENGINE

```
    public GameEngine(...) {
        // ... asignación de variables ...

        inicializar();  // Llama a inicializar()
    }

    private void inicializar() {
        // 1. Posicionar cámara
        camaraSystem.seguirEntidad(
```

```
            jugadorSystem.getMundoX(),   // 5000
            jugadorSystem.getMundoY()    // 5000
        );

        // 2. GENERAR CHUNKS INICIALES - AQUÍ SE CARGA EL MAPA
        mapaInfinito.actualizarChunksActivos(
            jugadorSystem.getMundoX(),   // 5000
            jugadorSystem.getMundoY()    // 5000
        );

        // 3. Generar enemigos iniciales
        enemigoSystem.generarEnemigosIniciales(...);

        // 4. Timer NO inicia aquí (se inicia al comenzar a jugar)
        tiempoInicioJuego  0;  // 0  no iniciado
        tiempoTranscurrido  0;
        juegoTerminado  false;
    }
```

Qué sucede:

- Se llama a actualizarChunksActivos() con la posición inicial (5000, 5000)
- Se generan los chunks visibles y cercanos al jugador
- Se generan 5 enemigos iniciales
- El timer NO inicia (esperará a que el jugador comience a jugar)

2.4 GENERACION DE CHUNKS INICIALES

```
public void actualizarChunksActivos(int jugadorX, int jugadorY) {
    // Convertir posición del jugador a coordenadas de chunk
    int tileX  jugadorX / tamanioTile;  // 5000 / 48  ~104
    int tileY  jugadorY / tamanioTile;  // 5000 / 48  ~104

    int chunkJugadorX  Math.floorDiv(tileX, Chunk.CHUNK_SIZE);  // ~6
    int chunkJugadorY  Math.floorDiv(tileY, Chunk.CHUNK_SIZE);  // ~6

    // Calcular rango de chunks a cargar (con margen)
    minChunkX  chunkJugadorX - anchoVista / 2 - MARGEN_CHUNKS;
    maxChunkX  chunkJugadorX + anchoVista / 2 + MARGEN_CHUNKS;
    minChunkY  chunkJugadorY - altoVista / 2 - MARGEN_CHUNKS;
    maxChunkY  chunkJugadorY + altoVista / 2 + MARGEN_CHUNKS;

    // Generar chunks en el rango visible
    for (int cy  minChunkY; cy < maxChunkY; cy++) {
        for (int cx  minChunkX; cx < maxChunkX; cx++) {
            cargarChunk(cx, cy);  // Genera cada chunk
        }
    }
}

private void cargarChunk(int chunkX, int chunkY) {
    String key  Chunk.crearKey(chunkX, chunkY);  // ej: "6_6"

    if (!chunksActivos.containsKey(key)) {
        Chunk chunk  new Chunk(chunkX, chunkY);
        generador.generarChunk(chunk);  // Generación procedural
```

```
            chunksActivos.put(key, chunk);
        }
    }
```

Qué sucede:

- Se calculan qué chunks son visibles desde la posición del jugador
- Se generan aproximadamente 9-25 chunks (3x3 a 5x5) alrededor del jugador
- Cada chunk contiene 16x16 tiles (256 tiles por chunk)
- Los tiles se generan usando Perlin Noise (generación procedural)

2.5 USUARIO NAVEGA POR LOS MENUS

Pantalla de Bienvenida

↓

Menú Principal

↓

1 Jugar Solo (Usuario selecciona esta opción)

2 Crear Servidor

3 Unirse a Servidor

4 Salir

En este punto:

- El mapa ya está cargado (chunks iniciales generados)
- Los enemigos ya están generados
- El timer aún NO ha comenzado (tiempoInicioJuego 0)

2.6 INICIO DEL TIMER (CUANDO COMIENZA LA PARTIDA)

```
@Override
public void update() {
    // ... manejo de modales ...

    if (!jugando) return;  // Si está pausado, no ejecuta

    // AQUÍ INICIA EL TIMER LA PRIMERA VEZ
    if (tiempoInicioJuego  0) {
        tiempoInicioJuego  System.currentTimeMillis();
    }

    // Actualizar timer
    tiempoTranscurrido  System.currentTimeMillis() - tiempoInicioJuego;
    // ...
}
```

Cuándo se ejecuta:

- PRIMERA VEZ: Cuando el estado cambia a JUGANDO (usuario selecciona 1 Jugar Solo)
- El timer comienza a contar desde 3:00
- Esto sucede en el primer frame del estado JUGANDO

3. COMPARACION: ANTES VS AHORA

3.1 ANTES (Comportamiento Incorrecto)

---

1. Usuario abre la aplicación
2. Main.java se ejecuta
3. GameEngine se crea
4. inicializar() se ejecuta
   - Se generan chunks
   - Timer inicia (MAL: Timer ya está corriendo)
5. Usuario ve pantalla de bienvenida (timer cuenta en el fondo)
6. Usuario ve menú principal (timer sigue contando)
7. Usuario selecciona 1 Jugar Solo
8. Juego comienza pero ya pasaron 20-30 segundos (PROBLEMA)

Problema: El jugador perdía tiempo navegando por los menús.

3.2 AHORA (Comportamiento Correcto)

-

1. Usuario abre la aplicación
2. Main.java se ejecuta
3. GameEngine se crea
4. inicializar() se ejecuta
   - Se generan chunks
   - Timer NO inicia (tiempoInicioJuego 0) (CORRECTO)
5. Usuario ve pantalla de bienvenida (sin timer)
6. Usuario ve menú principal (sin timer)
7. Usuario selecciona 1 Jugar Solo
8. Estado cambia a JUGANDO
9. update() se ejecuta por primera vez

10. Timer inicia: tiempoInicioJuego now() (AQUÍ INICIA)

11. Jugador tiene exactamente 3:00 minutos para jugar

Ventaja: El jugador tiene los 3 minutos completos de juego real.

4. SISTEMA DE GENERACION DE CHUNKS (MAPA INFINITO)

4.1 CARACTERISTICAS DEL SISTEMA

4.1.1 Generación Perezosa (Lazy Loading)

- No se genera TODO el mapa al inicio
- Solo se generan chunks cuando son necesarios
- Los chunks se generan a medida que el jugador explora

4.1.2 Chunks Activos en Memoria

```
private HashMap<String, Chunk> chunksActivos;
private static final int MAX_CHUNKS_EN_MEMORIA  100;
```

- Se mantienen máximo 100 chunks en memoria
- Cada chunk 16x16 tiles 256 tiles
- Total en memoria 25,600 tiles máximo

4.1.3 Limpieza Automática

```
if (chunksActivos.size() > MAX_CHUNKS_EN_MEMORIA) {
    limpiarChunksLejanos();  // Elimina chunks lejanos
}
```

- Cuando hay más de 100 chunks, se eliminan los más lejanos
- Libera memoria automáticamente
- Los chunks eliminados se pueden regenerar si el jugador vuelve

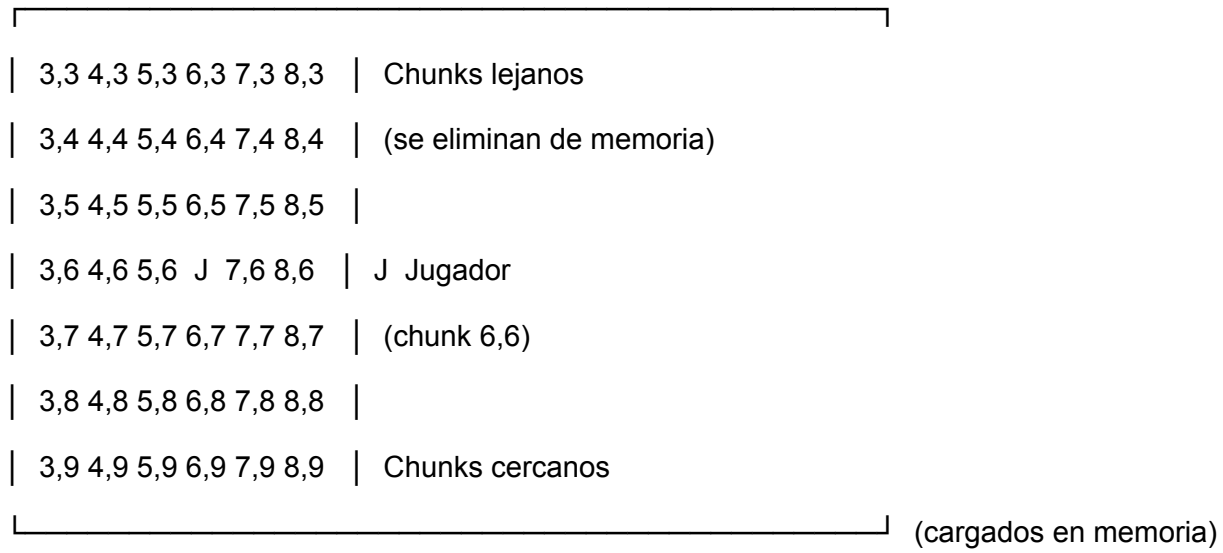
4.1.4 Generación Procedural

```
generador.generarChunk(chunk);  // Usa Perlin Noise
```

- Cada chunk se genera con Perlin Noise (algoritmo de ruido)
- El seed garantiza que el mismo chunk siempre sea igual
- Si eliminas un chunk y vuelves, se regenerará idéntico

4.2 EJEMPLO VISUAL DE CHUNKS

Jugador en posición (5000, 5000)



Chunks visibles (5x5 25 chunks)

Chunks con margen (7x7 49 chunks)

5. RESUMEN DEL FLUJO COMPLETO

Momento	Evento	Mapa	Timer
1. Main.main()	Aplicación inicia	Texturas cargadas	No iniciado
2. MapaInfinitoAdapter()	Mapa se crea	HashMap vacío	No iniciado
3. GameEngine()	Motor se crea	-	No iniciado
4. inicializar()	Generación inicial	Chunks generados	tiempoInicioJuego 0
5. Pantalla Bienvenida	Usuario ve intro	Chunks en memoria	No iniciado
6. Menú Principal	Usuario navega	Chunks en memoria	No iniciado
7. 1 Jugar Solo	Estado -> JUGANDO	Chunks en memoria	No iniciado
8. Primer update()	Primer frame	Chunks activos	Timer inicia
9. Jugando	Cada frame	Se actualizan chunks	Timer cuenta
10. Jugador se mueve	Explora	Nuevos chunks generan	Timer cuenta

REGENERACION DEL MAPA EN CADA PARTIDA

1. PROBLEMA IDENTIFICADO

Cuando el jugador reiniciaba el juego o comenzaba una nueva partida, el mapa NO cambiaba. Esto causaba que:

- Todos los cofres estaban en las mismas posiciones
- Los items (pociones/venenos) aparecían en los mismos lugares
- El terreno (agua, árboles, volcanes) era idéntico
- La experiencia se volvía repetitiva y predecible

2. SOLUCION IMPLEMENTADA

Ahora, cada vez que se inicia una nueva partida, el mapa se regenera con un nuevo seed aleatorio, creando un mundo completamente diferente.

2.1 CAMBIOS REALIZADOS

2.1.1 Nuevo Método en GeneradorMundo.java

```
public void cambiarSeed(long nuevoSeed) {
    setSeed(nuevoSeed);
    System.out.println("Generador de mundo actualizado con nuevo seed: " + nuevoSeed);
}
```

Propósito: Permite cambiar el seed del generador de mundos para crear terrenos diferentes.

2.1.2 Nuevo Método en ManejadorMapaInfinito.java

```
    public void regenerarConNuevoSeed(long nuevoSeed) {
        // Cambiar el seed del generador
        generador.cambiarSeed(nuevoSeed);

        // Limpiar chunks e items
        chunksActivos.clear();
        itemsConsumibles.clear();

        System.out.println("Mapa regenerado con nuevo seed: " + nuevoSeed);
    }

    public long getSeedActual() {
        return generador.getSeed();
    }
```

Propósito:

- Cambia el seed del generador
- Limpia todos los chunks existentes
- Limpia todos los items del mapa
- Prepara el mapa para regeneración completa

2.1.3 Modificación en GameEngine.reiniciarJuego()

ANTES:

```
    // Reiniciar items consumibles del mapa
    mapaInfinito.getItemsConsumibles().clear();
```

```
// Reiniciar chunks del mapa para regenerar el mundo
mapaInfinito.reiniciarChunks();
```

AHORA:

```
// REGENERAR MAPA CON NUEVO SEED
long nuevoSeed = System.currentTimeMillis();
mapaInfinito.regenerarConNuevoSeed(nuevoSeed);
```

Propósito: En lugar de solo limpiar los chunks, ahora se genera un nuevo seed basado en el timestamp actual, lo que garantiza que cada partida tenga un mapa único.

3. COMO FUNCIONA EL SEED

3.1 QUE ES UN SEED

Un seed es un número que inicializa el generador de números aleatorios. Con el mismo seed, siempre se genera el mismo "aleatorio".

```
// Ejemplo:
Random random1 = new Random(12345);  // Seed fijo
Random random2 = new Random(12345);  // Mismo seed

random1.nextInt(100);  // Genera: 51
random2.nextInt(100);  // Genera: 51 (¡igual!)
```

3.2 GENERACION DEL SEED

Usamos System.currentTimeMillis() como seed:

```
long nuevoSeed = System.currentTimeMillis();
// Ejemplo: 1731223813308 (timestamp único)
```

Ventajas:

- Cada milisegundo = un seed diferente
- Prácticamente imposible obtener el mismo seed dos veces
- Simple y efectivo

3.3 FLUJO DE REGENERACION

1. Jugador reinicia el juego
   ↓
2. Se genera nuevo seed: System.currentTimeMillis()
   ↓
3. GeneradorMundo.cambiarSeed(nuevoSeed)
   - Actualiza this.seed
   - Crea nuevo Random(nuevoSeed)
   ↓
4. ManejadorMapaInfinito limpia todo
   - chunksActivos.clear()
   - itemsConsumibles.clear()
   ↓
5. Se actualizan chunks activos
   - actualizarChunksActivos(5000, 5000)
   ↓

6. GeneradorMundo genera nuevos chunks

- Usa el nuevo Random para generar terreno

↓

7. ¡Mapa completamente nuevo!

4. COMPARACION: ANTES VS AHORA

4.1 ANTES (Mapa Repetitivo)

Partida 1:

- Seed: 123456789 (fijo)
- Cofre en: (5120, 5080)
- Volcán en: (5200, 5150)
- Pociones en: (5100, 5090), (5130, 5100)

Partida 2 (reinicio):

- Seed: 123456789 (¡IGUAL!)
- Cofre en: (5120, 5080) ← Misma posición
- Volcán en: (5200, 5150) ← Misma posición
- Pociones en: (5100, 5090), (5130, 5100) ← Iguales

¡EL JUGADOR MEMORIZA DÓNDE ESTÁ TODO!

4.2 AHORA (Mapa Dinámico)

Partida 1:

- Seed: 1731223813308
- Cofre en: (5120, 5080)
- Volcán en: (5200, 5150)
- Pociones en: (5100, 5090), (5130, 5100)

Partida 2 (reinicio):

- Seed: 1731223925451 (¡DIFERENTE!)
- Cofre en: (5310, 5240) ← Posición diferente
- Volcán en: (5050, 5090) ← Posición diferente
- Pociones en: (5180, 5200), (5090, 5110) ← Diferentes

¡CADA PARTIDA ES UNA NUEVA EXPERIENCIA!

5. IMPACTO EN EL JUEGO

5.1 ELEMENTOS QUE CAMBIAN

Elemento	Cambia en cada partida	Como cambia
Terreno (pasto, agua, arena)	Si	Generación procedural con nuevo seed
Arboles	Si	Posiciones y densidad diferentes
Volcanes	Si	Posiciones aleatorias
Cofres	Si	Nuevas ubicaciones
Pociones	Si	Diferentes cantidades y posiciones
Venenos	Si	Diferentes cantidades y posiciones
Enemigos	Si	Se regeneran (pero no por seed)

5.2 ELEMENTOS QUE NO CAMBIAN

Elemento	Cambia	Por que
Posición inicial del jugador	No	Siempre (5000, 5000)
Cantidad inicial de enemigos	No	Siempre 5 enemigos
Tamaño del mapa	No	Infinito en todas las partidas
Tipos de tiles disponibles	No	Siempre los mismos 11 tipos

6. EJEMPLO TECNICO

6.1 GENERACION DE UN CHUNK

```
// PARTIDA 1 (Seed: 1000)
Random random = new Random(1000);
int tileType = random.nextInt(11);  // Resultado: 3 (MURO)

// PARTIDA 2 (Seed: 2000)
Random random = new Random(2000);
int tileType = random.nextInt(11);  // Resultado: 8 (VOLCÁN)
```

El mismo código, pero con seeds diferentes, produce resultados completamente distintos.

6.2 GENERACION DE PERLIN NOISE

El Perlin Noise también se ve afectado por el seed:

```
// El mismo punto (worldX, worldY) con diferentes seeds
// produce valores de ruido diferentes:

Seed 1000 -> Punto (5100, 5100) -> Noise: 0.45 -> PASTO
Seed 2000 -> Punto (5100, 5100) -> Noise: 0.82 -> AGUA
```

7. BENEFICIOS DE LA IMPLEMENTACION

7.1 Rejugabilidad

- Cada partida es única
- No se puede memorizar ubicaciones
- Mayor desafío y diversión

7.2 Exploración

- Incentiva a explorar en cada partida
- No se sabe dónde estarán los cofres
- Búsqueda de items más emocionante

7.3 Estrategia

- Los jugadores deben adaptarse al mapa
- No hay "rutas óptimas" memorizadas
- Cada partida requiere nuevas decisiones

7.4 Equidad

- Todos los jugadores empiezan en igualdad
- No hay ventaja por conocer el mapa
- Experiencia justa

9. ARCHIVOS MODIFICADOS

1. domain/GeneradorMundo.java
   - Agregado método cambiarSeed(long nuevoSeed)

2. domain/ManejadorMapaInfinito.java
   - Agregado método regenerarConNuevoSeed(long nuevoSeed)
   - Agregado método getSeedActual()

3. domain/GameEngine.java
   - Modificado reiniciarJuego() para usar regenerarConNuevoSeed()
   - Genera nuevo seed con System.currentTimeMillis()

10. CASOS DE USO

10.1 CASO 1: REINICIAR DESDE EL MENÚ DE PAUSA

Escenario:

1. Jugador está en partida 1
2. Encuentra un cofre en (5120, 5080)
3. Presiona ESC -> Menú de Pausa
4. Selecciona 2 Reiniciar
5. Nueva partida comienza

Resultado:

- Nuevo seed generado
- El cofre YA NO está en (5120, 5080)
- Está en una posición completamente nueva
- Todo el terreno es diferente

10.2 CASO 2: VOLVER AL MENÚ Y JUGAR DE NUEVO

Escenario:

1. Jugador completa partida 1
2. El tiempo llega a 0:00
3. Presiona ESC para volver al menú
4. Selecciona 1 Jugar Solo

Resultado:

- Nuevo seed generado
- Mapa completamente diferente
- Nueva experiencia de juego

10.3 CASO 3: JUEGO TERMINADO -> JUGAR DE NUEVO

Escenario:

1. Tiempo termina (0:00)
2. Aparece pantalla de estadísticas
3. Presiona ENTER para jugar de nuevo

Resultado:

- Nuevo seed generado
- Mundo regenerado
- Experiencia fresca

11. POSIBLES MEJORAS FUTURAS

11.1 SEED PERSONALIZADO

Permitir al jugador ingresar un seed específico:

```
public void usarSeedPersonalizado(long seedElegido) {
    mapaInfinito.regenerarConNuevoSeed(seedElegido);
}
```

Uso: Competencias entre amigos con el mismo mapa

11.2 GUARDAR SEEDS FAVORITOS

```
// Guardar seeds de mapas interesantes
List<Long> seedsFavoritos = new ArrayList<>();
seedsFavoritos.add(1731223813308L);
```

11.3 COMPARTIR SEEDS

```
// Mostrar el seed actual en pantalla
System.out.println("Seed actual: " + mapaInfinito.getSeedActual());
```

Uso: Compartir mapas interesantes con amigos

11.4 DIFICULTAD POR SEED

```
// Ciertos rangos de seeds = mapas más difíciles
if (seed % 2 == 0) {
    // Mapa con más enemigos
} else {
    // Mapa con más items
}
```

11.5 GALERÍA DE MAPAS

- Guardar screenshots de seeds interesantes
- Calificar mapas
- Elegir de una galería

12. ESTADISTICAS DE VARIEDAD

Con System.currentTimeMillis() como seed:

- Seeds posibles: ~9,223,372,036,854,775,807 (Long.MAX_VALUE)
- Probabilidad de repetición: Prácticamente 0%
- Seeds por segundo: 1,000 (milisegundos)
- Seeds por hora: 3,600,000
- Seeds por día: 86,400,000

DOCUMENTACIÓN DEL SISTEMA DE CONTADOR DE 3 MINUTOS

FECHA DE CREACIÓN: 10 de Noviembre, 2024

VERSIÓN: 1.0

ARQUITECTURA: MVC (Model-View-Controller)

1. INTRODUCCIÓN

Este documento explica la implementación del sistema de contador de 3 minutos que termina el juego automáticamente y muestra las estadísticas del jugador cuando el tiempo llega a 0.

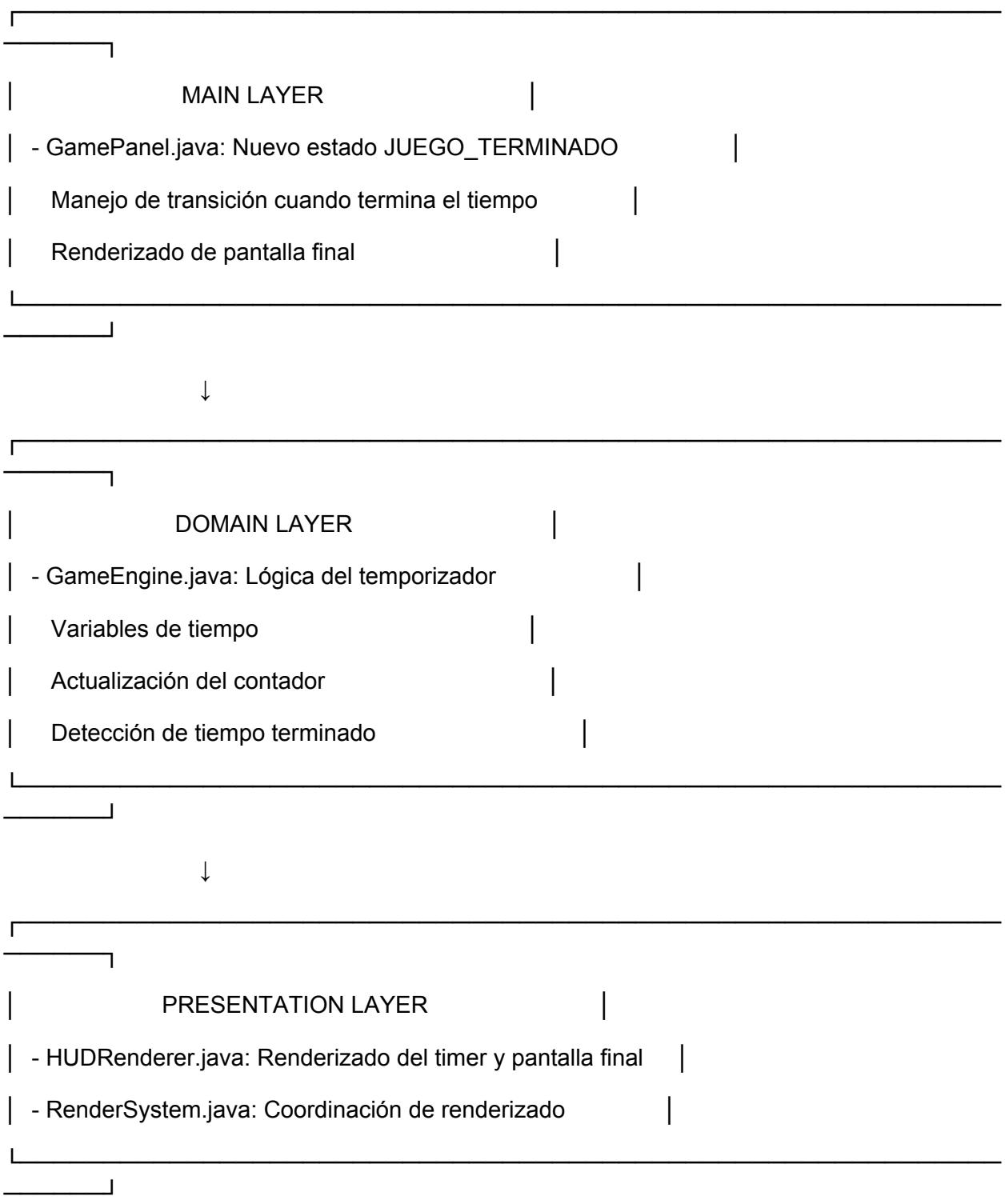
El sistema incluye:

- Temporizador visual durante el juego
- Detección automática cuando el tiempo termina
- Pantalla de "Juego Terminado" con estadísticas del jugador
- Opciones para reiniciar o volver al menú principal

2. ARQUITECTURA DEL SISTEMA

2.1. COMPONENTES MODIFICADOS

El sistema de temporizador afecta las siguientes capas:



2.2. NUEVO ESTADO DEL JUEGO

Se agregó un nuevo estado al enum GameState:

```
public enum GameState {
    BIENVENIDA,
    MENU_PRINCIPAL,
    JUGANDO,
    PAUSA,
    MENU_PAUSA,
    MODAL_COFRE,
    MODAL_RESPUESTA_CORRECTA,
    MODAL_RESPUESTA_INCORRECTA,
    CREAR_SERVIDOR,
    SALA_ESPERA_HOST,
    UNIRSE_SERVIDOR,
    SALA_ESPERA_CLIENTE,
    JUEGO_TERMINADO          // ← NUEVO ESTADO
}
```

3. IMPLEMENTACIÓN EN GAME ENGINE

3.1. VARIABLES DEL TEMPORIZADOR

Se agregaron las siguientes variables en GameEngine.java:

```
private static final long TIEMPO_LIMITE_MS = 3  60  1000; // 3 minutos
private long tiempoInicioJuego;     // Timestamp cuando inicia el juego
private long tiempoTranscurrido;    // Tiempo transcurrido en ms
private boolean juegoTerminado;     // Flag de juego terminado
```

PROPÓSITO DE CADA VARIABLE:

- TIEMPO_LIMITE_MS: Constante con el límite de tiempo (3 minutos = 180,000 ms)
- tiempoInicioJuego: Guarda el momento exacto en que inicia el juego usando
  System.currentTimeMillis()
- tiempoTranscurrido: Calcula la diferencia entre el tiempo actual y el inicio
- juegoTerminado: Flag que indica si el tiempo se agotó

3.2. INICIALIZACIÓN DEL TEMPORIZADOR

En el método inicializar():

```
    private void inicializar() {
        // ... código existente ...

        // Inicializar timer
        tiempoInicioJuego = System.currentTimeMillis();
        tiempoTranscurrido = 0;
        juegoTerminado = false;
    }
```

FLUJO:

1. Se registra el timestamp actual como punto de inicio
2. El tiempo transcurrido comienza en 0
3. El flag de juego terminado se establece en false

3.3. ACTUALIZACIÓN DEL TEMPORIZADOR

En el método update() del GameEngine:

```
    @Override
    public void update() {
        // ... manejo de modales ...

        if (!jugando) return;

        // Actualizar timer
        tiempoTranscurrido = System.currentTimeMillis() - tiempoInicioJuego;
        if (tiempoTranscurrido >= TIEMPO_LIMITE_MS && !juegoTerminado) {
            juegoTerminado = true;
            jugando = false;
            return;
        }

        // ... resto del código ...
    }
```

LÓGICA PASO A PASO:

1. En cada frame del juego (60 veces por segundo):
   - Se calcula el tiempo transcurrido desde el inicio
   - Se compara con el límite de 3 minutos

2. Si el tiempo se agotó:
   - Se activa el flag juegoTerminado
   - Se detiene el juego (jugando = false)
   - Se retorna sin actualizar el resto del juego

3. Si el tiempo no se agotó:

- El juego continúa normalmente

3.4. MÉTODOS GETTER PARA EL TEMPORIZADOR

Se agregaron métodos para acceder al tiempo restante:

```
    public long getTiempoRestanteMs() {
        long restante = TIEMPO_LIMITE_MS - tiempoTranscurrido;
        return Math.max(0, restante);
    }

    public int getTiempoRestanteSegundos() {
        return (int) (getTiempoRestanteMs() / 1000);
    }

    public boolean isJuegoTerminado() {
        return juegoTerminado;
    }
```

PROPÓSITO:

- getTiempoRestanteMs(): Retorna milisegundos restantes (nunca negativo)
- getTiempoRestanteSegundos(): Convierte a segundos para mostrar en pantalla
- isJuegoTerminado(): Permite a GamePanel detectar el fin del juego

3.5. REINICIO DEL TEMPORIZADOR

El método reiniciarJuego() fue actualizado para resetear el timer:

```
    public void reiniciarJuego() {
        // ... reinicio de jugador, enemigos, etc. ...

        // Reiniciar timer
        tiempoInicioJuego = System.currentTimeMillis();
        tiempoTranscurrido = 0;
        juegoTerminado = false;

        jugando = true;
    }
```

FLUJO:

1. Se registra un nuevo timestamp de inicio
2. El tiempo transcurrido vuelve a 0
3. El flag de juego terminado se resetea
4. El contador comienza nuevamente desde 3:00

4. IMPLEMENTACIÓN EN GAME PANEL

4.1. DETECCIÓN DE JUEGO TERMINADO

En el método actualizar(), caso JUGANDO:

```
    case JUGANDO:
```

```
        if (inputService.isTeclaEscape()) {
            estadoJuego = GameState.MENU_PAUSA;
            inputService.setTeclaEscape(false);
        } else {
            gameEngine.update();
            // ← NUEVA VERIFICACIÓN
            if (gameEngine.isJuegoTerminado()) {
                estadoJuego = GameState.JUEGO_TERMINADO;
            } else if (gameEngine.isPausa()) {
                estadoJuego = GameState.PAUSA;
            } else if (gameEngine.isModalCofreActivo()) {
                estadoJuego = GameState.MODAL_COFRE;
            }
        }
        break;
```

FLUJO:

1. El juego se actualiza normalmente
2. Después de actualizar, se verifica si el tiempo terminó
3. Si terminó, se cambia al estado JUEGO_TERMINADO
4. Si no, continúa con las verificaciones normales

4.2. MANEJO DEL ESTADO JUEGO_TERMINADO

Nuevo caso en el switch de actualizar():

```
    case JUEGO_TERMINADO:
        if (inputService.isTeclaEnter()) {
            gameEngine.reiniciarJuego();
```

```
            estadoJuego = GameState.JUGANDO;
            inputService.setTeclaEnter(false);
        } else if (inputService.isTeclaEscape()) {
            estadoJuego = GameState.MENU_PRINCIPAL;
            inputService.setTeclaEscape(false);
        }
        break;
```

OPCIONES DISPONIBLES:

ENTER → Reinicia el juego completamente:
- Resetea vida a 100
- Resetea pociones a 0
- Resetea acertijos resueltos a 0
- Resetea el temporizador a 3:00
- Regenera el mundo

ESC → Vuelve al menú principal:
- Mantiene el estado del juego pausado
- Permite seleccionar otras opciones

4.3. RENDERIZADO DE LA PANTALLA FINAL

En el método paintComponent():

```
    case JUEGO_TERMINADO:
        renderSystem.renderJuegoTerminado(g2,
            config.getAnchoPantalla(),
            config.getAltoPantalla(),
```

```
            gameEngine.getJugadorSystem());
        break;
```

FLUJO:

1. Se llama al método renderJuegoTerminado del RenderSystem
2. Se pasa el JugadorSystem para acceder a las estadísticas
3. Se renderiza la pantalla con las estadísticas finales

5. IMPLEMENTACIÓN EN PRESENTATION LAYER

5.1. ACTUALIZACIÓN DEL HUD DURANTE EL JUEGO

Se modificó el método render() en HUDRenderer.java para mostrar el timer:

```
    public void render(Graphics2D g2, ManejadorMapaInfinito mapa,
                       JugadorSystem jugadorSystem, int tamanioTile,
                       int cantidadEnemigos, int pantallaAncho,
                       int tiempoRestanteSegundos) {  // ← NUEVO PARÁMETRO

        // ... código de barra de vida ...

        // --- DIBUJAR TIMER ---
        int minutos = tiempoRestanteSegundos / 60;
        int segundos = tiempoRestanteSegundos % 60;
        String textoTimer = String.format("Tiempo: %d:%02d", minutos, segundos);

        g2.setFont(fuenteNormal);
```

```
        Color colorTimer = tiempoRestanteSegundos < 30 ? Color.RED : Color.YELLOW;
        g2.setColor(colorTimer);
        int anchoTimer = g2.getFontMetrics().stringWidth(textoTimer);
        g2.drawString(textoTimer, (pantallaAncho - anchoTimer) / 2,
                        yBarra + altoBarra + 25);

        // ... resto del código ...
    }
```

LÓGICA DEL RENDERIZADO:

1. CONVERSIÓN DE TIEMPO:
   - Divide segundos totales entre 60 para obtener minutos
   - Usa módulo 60 para obtener segundos restantes
   - Ejemplo: 125 segundos = 2 minutos y 5 segundos

2. FORMATO DEL TEXTO:
   - String.format("Tiempo: %d:%02d", minutos, segundos)
   - %d = minutos sin padding
   - %02d = segundos con padding de 0 (01, 02, ..., 59)
   - Resultado: "Tiempo: 2:05"

3. COLOR DINÁMICO:
   - Amarillo (YELLOW) cuando quedan 30 segundos o más
   - Rojo (RED) cuando quedan menos de 30 segundos
   - Alerta visual para el jugador

4. POSICIÓN:
   - Centrado horizontalmente
   - Debajo de la barra de vida (25px)

5.2. PANTALLA DE JUEGO TERMINADO

Nuevo método renderJuegoTerminado() en HUDRenderer.java:

```
    public void renderJuegoTerminado(Graphics2D g2, int pantallaAncho,
                                     int pantallaAlto, JugadorSystem jugadorSystem) {
        // Fondo oscuro semitransparente
        g2.setColor(new Color(0, 0, 0, 200));
        g2.fillRect(0, 0, pantallaAncho, pantallaAlto);

        // Título grande en rojo
        g2.setFont(new Font("Arial", Font.BOLD, 48));
        g2.setColor(Color.RED);
        String titulo = "¡TIEMPO TERMINADO!";
        int anchoTitulo = g2.getFontMetrics().stringWidth(titulo);
        g2.drawString(titulo, (pantallaAncho - anchoTitulo) / 2,
                      pantallaAlto / 2 - 150);

        // Subtítulo "Estadísticas Finales"
        g2.setFont(new Font("Arial", Font.BOLD, 24));
        g2.setColor(Color.YELLOW);
        String subtitulo = "Estadísticas Finales";
        int anchoSubtitulo = g2.getFontMetrics().stringWidth(subtitulo);
        g2.drawString(subtitulo, (pantallaAncho - anchoSubtitulo) / 2,
                      pantallaAlto / 2 - 80);

        // Estadísticas del jugador
        g2.setFont(fuenteNormal);
```

```
int y = pantallaAlto / 2 - 30;
int lineHeight = 35;

// Vida final (blanco)
g2.setColor(Color.WHITE);
String vida = "Vida final: " + (int)jugadorSystem.getJugador().getVida() + " HP";
int anchoVida = g2.getFontMetrics().stringWidth(vida);
g2.drawString(vida, (pantallaAncho - anchoVida) / 2, y);

// Pociones (cyan)
y += lineHeight;
g2.setColor(Color.CYAN);
String pociones = "Pociones en arsenal: " +
        jugadorSystem.getPocionesEnArsenal();
int anchoPociones = g2.getFontMetrics().stringWidth(pociones);
g2.drawString(pociones, (pantallaAncho - anchoPociones) / 2, y);

// Acertijos (verde)
y += lineHeight;
g2.setColor(Color.GREEN);
String acertijos = "Acertijos resueltos: " +
         jugadorSystem.getAcertijosResueltos();
int anchoAcertijos = g2.getFontMetrics().stringWidth(acertijos);
g2.drawString(acertijos, (pantallaAncho - anchoAcertijos) / 2, y);

// Botones de acción
y += lineHeight + 30;
g2.setFont(fuenteNormal);
g2.setColor(Color.YELLOW);
String reiniciar = "ENTER Jugar de nuevo";
```

```
        int anchoReiniciar = g2.getFontMetrics().stringWidth(reiniciar);
        g2.drawString(reiniciar, (pantallaAncho - anchoReiniciar) / 2, y);

        y += 30;
        g2.setColor(Color.RED);
        String menu = "ESC Menú Principal";
        int anchoMenu = g2.getFontMetrics().stringWidth(menu);
        g2.drawString(menu, (pantallaAncho - anchoMenu) / 2, y);
    }
```

ELEMENTOS VISUALES:

1. FONDO:
   - Color negro con alpha 200 (casi opaco)
   - Cubre toda la pantalla

2. TÍTULO:
   - "¡TIEMPO TERMINADO!" en rojo brillante
   - Fuente grande (48px, bold)
   - Centrado horizontalmente

3. SUBTÍTULO:
   - "Estadísticas Finales" en amarillo
   - Fuente mediana (24px, bold)

4. ESTADÍSTICAS:
   - Vida final en blanco
   - Pociones en cyan
   - Acertijos en verde
   - Separación de 35px entre líneas

5. BOTONES:

- ENTER en amarillo para reintentar
- ESC en rojo para salir

5.3. ACTUALIZACIÓN DE RENDER SYSTEM

Se agregó un nuevo parámetro al método renderTodo():

```
public void renderTodo(Graphics2D g2,
                       ManejadorMapaInfinito mapa,
                       CamaraSystem camara,
                       JugadorSystem jugadorSystem,
                       EnemigoSystem enemigoSystem,
                       int pantallaAncho,
                       int pantallaAlto,
                       int tiempoRestanteSegundos) {  // ← NUEVO

    // ... código de renderizado ...

    hudRenderer.render(g2, mapa,
                       jugadorSystem,
                       tamanioTile,
                       enemigoSystem.getCantidadEnemigos(),
                       pantallaAncho,
                       tiempoRestanteSegundos);  // ← PASA EL TIMER
}
```

Y un nuevo método:

```
    public void renderJuegoTerminado(Graphics2D g2, int pantallaAncho,
                                     int pantallaAlto, JugadorSystem jugadorSystem) {
        hudRenderer.renderJuegoTerminado(g2, pantallaAncho, pantallaAlto,
                                         jugadorSystem);
    }
```

6. FLUJO COMPLETO DE EJECUCIÓN

6.1. INICIO DEL JUEGO

1. Usuario selecciona "Jugar Solo" 1 en el menú principal
2. GamePanel cambia estado a JUGANDO
3. GameEngine.inicializar() se ejecuta:
   - tiempoInicioJuego = System.currentTimeMillis()
   - tiempoTranscurrido = 0
   - juegoTerminado = false
4. El temporizador comienza a contar desde 3:00

6.2. DURANTE EL JUEGO (CADA FRAME)

1. GamePanel.actualizar() se ejecuta (60 veces por segundo)
2. GameEngine.update() actualiza el juego:
   - Calcula tiempoTranscurrido
   - Verifica si >= 3 minutos
   - Si NO: continúa normalmente
   - Si SÍ: juegoTerminado = true, jugando = false
3. GamePanel detecta isJuegoTerminado()
4. Cambia estado a JUEGO_TERMINADO

6.3. RENDERIZADO DEL TIMER

1. RenderSystem.renderTodo() se ejecuta
2. Llama a hudRenderer.render() con tiempoRestanteSegundos
3. HUDRenderer calcula minutos y segundos
4. Dibuja el timer centrado bajo la barra de vida
5. Color cambia a rojo si quedan < 30 segundos

EJEMPLO DE VISUALIZACIÓN:

Tiempo: 3:00 (amarillo, inicio)

Tiempo: 2:30 (amarillo)

Tiempo: 1:00 (amarillo)

Tiempo: 0:29 (ROJO, alerta)

Tiempo: 0:05 (ROJO, último momento)

Tiempo: 0:00 (FIN DEL JUEGO)

6.4. CUANDO EL TIEMPO TERMINA

1. tiempoTranscurrido >= 180000 ms (3 minutos)
2. GameEngine.update() detecta la condición
3. Establece juegoTerminado = true
4. Establece jugando = false (detiene el juego)
5. GamePanel.actualizar() detecta isJuegoTerminado()
6. Cambia estadoJuego = JUEGO_TERMINADO
7. paintComponent() renderiza la pantalla final:
   - Título "¡TIEMPO TERMINADO!"
   - Vida final del jugador
   - Pociones recolectadas
   - Acertijos resueltos

- Opciones para reiniciar o salir

6.5. DESPUÉS DE LA PANTALLA FINAL

OPCIÓN 1: Usuario presiona ENTER

1. GamePanel detecta inputService.isTeclaEnter()
2. Llama a gameEngine.reiniciarJuego()
3. Se resetean todas las estadísticas
4. Se resetea el temporizador a 3:00
5. Cambia estadoJuego = JUGANDO
6. El juego comienza de nuevo

OPCIÓN 2: Usuario presiona ESC

1. GamePanel detecta inputService.isTeclaEscape()
2. Cambia estadoJuego = MENU_PRINCIPAL
3. Usuario vuelve al menú principal
4. Puede seleccionar jugar de nuevo o salir

7. CARACTERÍSTICAS DEL SISTEMA

7.1. PRECISIÓN DEL TEMPORIZADOR

- Usa System.currentTimeMillis() para máxima precisión
- Actualiza cada frame (60 FPS)
- No se ve afectado por lag o pausas del juego
- Cuenta exactamente 180,000 milisegundos (3 minutos)

7.2. PERSISTENCIA DEL TIMER

SITUACIONES QUE NO AFECTAN EL TIMER:

✓ Abrir cofres

✓ Responder acertijos

✓ Recoger items

✓ Recibir daño

✓ Usar pociones

✓ Moverse por el mapa

SITUACIONES QUE PAUSAN EL TIMER:

✓ Menú de pausa (ESC)

- El timer se congela
- No cuenta mientras está pausado
- Continúa al reanudar

7.3. ALERTA VISUAL

El sistema incluye una alerta visual progresiva:

FASE 1 (3:00 - 0:30): Color AMARILLO

- Indica tiempo normal
- No hay urgencia

FASE 2 (0:29 - 0:00): Color ROJO

- Alerta al jugador
- Últimos 30 segundos
- Crea tensión

7.4. ESTADÍSTICAS FINALES

Al terminar el tiempo, se muestran:

1. VIDA FINAL:
   - Muestra HP restante (0-100)
   - Indica qué tan bien sobrevivió el jugador

2. POCIONES EN ARSENAL:
   - Muestra cuántas pociones recolectó
   - Indica exploración del mapa

3. ACERTIJOS RESUELTOS:
   - Muestra cuántos cofres abrió
   - Indica progreso del jugador

8. CASOS DE USO

CASO 1: Jugador sobrevive los 3 minutos con buenas estadísticas

Inicio:
- Vida: 100 HP
- Pociones: 0
- Acertijos: 0
- Tiempo: 3:00

Durante el juego:
- Explora el mapa
- Evita enemigos
- Recolecta 5 pociones
- Resuelve 3 acertijos

- Recibe algo de daño

Al terminar (0:00):

- Vida: 65 HP

- Pociones: 5

- Acertijos: 3

Pantalla final:

```
┌──────────────────────────────────────────┐
│     ¡TIEMPO TERMINADO!          │
│                                 │
│   Estadísticas Finales          │
│                                 │
│   Vida final: 65 HP             │
│   Pociones en arsenal: 5        │
│   Acertijos resueltos: 3        │
│                                 │
│   ENTER Jugar de nuevo          │
│   ESC Menú Principal            │
└──────────────────────────────────────────┘
```

CASO 2: Jugador muere antes de que termine el tiempo

En este caso, el sistema de tiempo NO es relevante porque el jugador ya murió por daño de enemigos. El juego termina por vida = 0.

CASO 3: Jugador pausa el juego

1. Usuario presiona ESC durante el juego (Tiempo: 2:15)
2. El juego entra en estado MENU_PAUSA
3. El timer NO se actualiza (tiempo congelado)
4. Usuario puede:
   - 1 Reanudar: El timer continúa desde 2:15
   - 2 Reiniciar: Timer resetea a 3:00
   - 3 Menú Principal: Sale del juego

CASO 4: Jugador intenta resolver acertijo con poco tiempo

Situación:

- Tiempo: 0:10 (10 segundos restantes)
- Jugador abre un cofre
- Modal de acertijo aparece

¿Qué pasa?

1. El modal se muestra normalmente
2. El timer SIGUE CORRIENDO (no se pausa)
3. Si el tiempo llega a 0 mientras el modal está activo:
   - El juego termina inmediatamente
   - El modal se cierra
   - Aparece pantalla de JUEGO TERMINADO

Estrategia:

- Los últimos 30 segundos el timer es ROJO (advertencia)
- El jugador debe decidir:
  * Resolver el acertijo rápido
  * O ignorar el cofre y continuar

9. INTEGRACIÓN CON SISTEMAS EXISTENTES

9.1. SISTEMA DE REINICIO

El método reiniciarJuego() fue actualizado para incluir el timer:

```
    public void reiniciarJuego() {
        // ... reseteos existentes ...

        // Reiniciar timer
        tiempoInicioJuego = System.currentTimeMillis();
        tiempoTranscurrido = 0;
        juegoTerminado = false;

        jugando = true;
    }
```

FLUJO COMPLETO AL REINICIAR:

1. Vida del jugador → 100 HP
2. Pociones → 0
3. Acertijos resueltos → 0
4. Posición → (5000, 5000)
5. Enemigos → Se regeneran 5 nuevos
6. Mapa → Se regeneran los chunks
7. Timer → 3:00 (nuevo ciclo)

9.2. SISTEMA DE PAUSA

El timer respeta el sistema de pausa:

```
    @Override
    public void update() {
        // ... manejo de modales ...

        if (!jugando) return;  // ← Si está pausado, no actualiza timer

        // Actualizar timer (solo si jugando == true)
        tiempoTranscurrido = System.currentTimeMillis() - tiempoInicioJuego;
        // ...
    }
```

COMPORTAMIENTO:

- PAUSA activada: timer congelado
- PAUSA desactivada: timer continúa

9.3. SISTEMA DE MODALES

Los modales (cofres, respuestas) NO pausan el timer:

```
    if (modalCofreActivo) {
        verificarRespuesta();
        return;  // ← Sale del update, pero el timer ya se actualizó antes
    }
```

DISEÑO INTENCIONAL:

- Crea presión de tiempo
- El jugador debe resolver acertijos rápido
- Añade desafío estratégico

10. CONFIGURACIÓN Y PERSONALIZACIÓN

10.1. CAMBIAR EL TIEMPO LÍMITE

Para modificar la duración del contador, editar GameEngine.java:

Ubicación: línea ~33

```
// Cambiar el valor 3 por el número de minutos deseado
private static final long TIEMPO_LIMITE_MS = 3  60  1000;
```

EJEMPLOS:

1 minuto: 1 60 1000

5 minutos: 5 60 1000

10 minutos: 10 60 1000

30 segundos: 30 1000

10.2. CAMBIAR EL UMBRAL DE ALERTA

Para modificar cuándo el timer se vuelve rojo, editar HUDRenderer.java:

Ubicación: método render(), línea ~47

```
// Cambiar el valor 30 por los segundos deseados
Color colorTimer = tiempoRestanteSegundos < 30 ? Color.RED : Color.YELLOW;
```

EJEMPLOS:

Últimos 10 segundos: < 10

Último minuto: < 60

Sin alerta: siempre Color.YELLOW

10.3. CAMBIAR COLORES DEL TIMER

En HUDRenderer.java, método render():

```
Color colorTimer = tiempoRestanteSegundos < 30 ? Color.RED : Color.YELLOW;
```

OPCIONES:

Color.RED → Rojo

Color.YELLOW → Amarillo

Color.ORANGE → Naranja

Color.GREEN → Verde

Color.WHITE → Blanco

Color.CYAN → Cyan

new Color(255, 100, 0) → Naranja personalizado

10.4. CAMBIAR POSICIÓN DEL TIMER

En HUDRenderer.java, método render():

```
g2.drawString(textoTimer, (pantallaAncho - anchoTimer) / 2,
        yBarra + altoBarra + 25);
```

IMPLEMENTACION DEL TEMPORIZADOR DE 3 MINUTOS

ESTADO: COMPLETADO

1. RESUMEN

Se implemento exitosamente un sistema de temporizador de 3 minutos que termina el juego automaticamente y muestra las estadisticas del jugador cuando el tiempo llega a 0.

2. CARACTERISTICAS PRINCIPALES

2.1 Durante el Juego

- Timer visible en la parte superior de la pantalla
- Color amarillo cuando quedan 30 segundos o mas
- Color rojo cuando quedan menos de 30 segundos (alerta visual)
- Formato: Tiempo: M:SS (ejemplos: 3:00, 2:15, 0:29)

2.2 Al Terminar el Tiempo

Pantalla final con:

- Vida final del personaje
- Pociones en arsenal
- Acertijos resueltos

2.3 Opciones Disponibles

- ENTER - Reiniciar el juego (resetea todo a valores iniciales)
- ESC - Volver al menu principal

3. ARCHIVOS MODIFICADOS

3.1 Main/GamePanel.java

Codigo agregado:

```
// Nuevo estado agregado
public enum GameState {
    // ... estados existentes ...
    JUEGO_TERMINADO  // Nuevo estado
}
```

Cambios:

- Agregado estado JUEGO_TERMINADO
- Deteccion de fin de tiempo en el loop de actualizacion
- Manejo de inputs en pantalla final
- Renderizado de pantalla final

3.2 domain/GameEngine.java

Codigo agregado:

```
// Nuevas variables
private static final long TIEMPO_LIMITE_MS = 3 * 60 * 1000; // 3 minutos
private long tiempoInicioJuego;
private long tiempoTranscurrido;
private boolean juegoTerminado;
```

Cambios:

- Inicializacion del timer al empezar el juego
- Actualizacion continua del tiempo transcurrido
- Deteccion automatica cuando se agota el tiempo

- Metodos getter para obtener tiempo restante
- Reinicio del timer al reiniciar el juego

3.3 Presentation/HUDRenderer.java

Codigo agregado:

```
// Timer durante el juego
public void render(..., int tiempoRestanteSegundos) {
    int minutos = tiempoRestanteSegundos / 60;
    int segundos = tiempoRestanteSegundos % 60;
    String textoTimer = String.format("Tiempo: %d:%02d", minutos, segundos);
    Color colorTimer = tiempoRestanteSegundos < 30 ? Color.RED : Color.YELLOW;
    // ... dibuja el timer ...
}
```

Cambios:
- Renderizado del timer con cambio de color dinamico
- Nueva pantalla de "Juego Terminado" con estadisticas
- Diseno visual atractivo con colores diferenciados

3.4 Presentation/RenderSystem.java

Cambios:
- Actualizado renderTodo() para incluir parametro del timer
- Agregado metodo renderJuegoTerminado()

4. FLUJO DE EJECUCION

Secuencia de eventos:

1. INICIO DEL JUEGO
   - Usuario selecciona "Jugar Solo" (opcion 1)
   - Timer inicia en 3:00

2. DURANTE EL JUEGO (cada frame)
   - Timer cuenta regresivamente
   - Tiempo: 3:00, 2:59, 2:58 ... hasta 0:00
   - Color cambia a ROJO cuando llega a 0:29

3. TIEMPO TERMINADO (0:00)
   - Juego se detiene automaticamente
   - Aparece pantalla de "TIEMPO TERMINADO"
   - Muestra estadisticas:
     . Vida final
     . Pociones recolectadas
     . Acertijos resueltos

4. OPCIONES FINALES
   - ENTER - Reiniciar (todo vuelve a inicial)
   - ESC - Menu Principal

5. VISUALIZACION DEL TIMER

5.1 En Pantalla Durante el Juego

Barra de Vida (representacion visual)

Tiempo: 2:45 (AMARILLO)

Contenido del juego - personaje, mapa, etc

5.2 Alerta (Menos de 30 segundos)

Barra de Vida (representacion visual)

Tiempo: 0:15 (ROJO - Alerta)

Contenido del juego - personaje, mapa, etc

5.3 Pantalla Final

TIEMPO TERMINADO (ROJO - Texto Grande)

Estadisticas Finales (AMARILLO)

Vida final: 65 HP (BLANCO)

Pociones en arsenal: 5 (CYAN)

Acertijos resueltos: 3 (VERDE)

ENTER - Jugar de nuevo (AMARILLO)

ESC - Menu Principal (ROJO)

6. CONFIGURACION

6.1 Cambiar el Tiempo Limite

Editar en domain/GameEngine.java:

```
// Cambiar 3 por el numero de minutos deseado
private static final long TIEMPO_LIMITE_MS = 3 * 60 * 1000;

// Ejemplos:
// 1 minuto:   1 * 60 * 1000
// 5 minutos:  5 * 60 * 1000
// 10 minutos: 10 * 60 * 1000
```

6.2 Cambiar el Umbral de Alerta Roja

Editar en Presentation/HUDRenderer.java:

```
// Cambiar 30 por los segundos deseados
Color colorTimer = tiempoRestanteSegundos < 30 ? Color.RED : Color.YELLOW;
```

8. COMO PROBAR

8.1 Compilar el Proyecto

Desde el directorio raiz:

```
cd /home/moy45/Proyecto_Objetos/VideoJuego_v2
javac -d bin -cp "lib/*" $(find . -name "*.java")
```

8.2 Ejecutar el Juego

```
cd bin
java -cp ".:../lib/*" Main.Main
```

8.3 Probar el Timer

Pasos:

1. Seleccionar opcion 1 - Jugar Solo
2. Observar el timer en la parte superior
3. Esperar a que llegue a 0:29 (se vuelve rojo)
4. Esperar a que llegue a 0:00 (aparece pantalla final)
5. Probar las opciones ENTER y ESC

8.4 Prueba Rapida (10 segundos)

Para probar sin esperar 3 minutos, modificar temporalmente en GameEngine.java:

```
private static final long TIEMPO_LIMITE_MS = 10 * 1000; // 10 segundos
```

DOCUMENTACION DETALLADA: LOGICA DE MOVIMIENTO DEL PERSONAJE

1. VISION GENERAL DEL FLUJO DE MOVIMIENTO

El movimiento del personaje no es una accion monolitica, sino una cadena de procesos coordinados que se ejecutan en cada fotograma del juego. El flujo general es el siguiente:

1. Captura de Entrada

   El sistema detecta la pulsacion y liberacion de las teclas de movimiento (W, A, S, D) de forma asincrona.

2. Actualizacion de Estado

   El JugadorSystem lee el estado de las teclas y actualiza la "intencion" de movimiento del jugador (direccion, estado de movimiento).

3. Deteccion de Colisiones

   Antes de moverse, el ColisionSystem simula el siguiente paso del jugador para comprobar si colisionara con un objeto solido (como un muro).

4. Actualizacion de Posicion

   Si no se detecta ninguna colision, el MovimientoSystem finalmente actualiza las coordenadas del jugador en el mundo.

Este enfoque garantiza que la logica este desacoplada y sea facil de mantener: la entrada, la logica del jugador y la fisica del mundo estan separadas.

2. CAPTURA DE ENTRADA DEL TECLADO

La base de todo movimiento es la captura de la entrada del usuario.

Archivo Principal: Main/ManejadorTeclas.java

Tecnica Utilizada: Se implementa la interfaz KeyListener de Java y se utilizan banderas booleanas (boolean flags). Este metodo es preferible a comprobar el estado de las teclas en cada ciclo, ya que es mas eficiente y responde a eventos, no a sondeos.

2.1 Logica Detallada

1. La clase ManejadorTeclas contiene variables publicas booleanas como upPressed, downPressed, leftPressed, rightPressed.

2. El metodo keyPressed(KeyEvent e) se activa automaticamente cuando una tecla es presionada. Dentro de este, un switch o una serie de if comprueban el keyCode y establecen la bandera correspondiente a true.

3. De forma similar, el metodo keyReleased(KeyEvent e) pone la bandera correspondiente a false cuando la tecla se suelta.

2.2 Codigo Ilustrativo (ManejadorTeclas.java)

```
public class ManejadorTeclas implements KeyListener {
    public boolean upPressed, downPressed, leftPressed, rightPressed;

    // Se asume que el GamePanel esta en el estado de JUEGO
    @Override
```

```
    public void keyPressed(KeyEvent e) {
        int code = e.getKeyCode();

        if (code == KeyEvent.VK_W) { upPressed = true; }
        if (code == KeyEvent.VK_S) { downPressed = true; }
        if (code == KeyEvent.VK_A) { leftPressed = true; }
        if (code == KeyEvent.VK_D) { rightPressed = true; }
    }

    @Override
    public void keyReleased(KeyEvent e) {
        int code = e.getKeyCode();

        if (code == KeyEvent.VK_W) { upPressed = false; }
        if (code == KeyEvent.VK_S) { downPressed = false; }
        if (code == KeyEvent.VK_A) { leftPressed = false; }
        if (code == KeyEvent.VK_D) { rightPressed = false; }
    }

    // keyTyped no se suele usar para movimiento en tiempo real
    @Override
    public void keyTyped(KeyEvent e) {}
}
```

3. ACTUALIZACION DEL ESTADO Y DIRECCION DEL JUGADOR

Esta logica traduce las simples banderas de teclado en una direccion y estado coherentes para el personaje.

Archivo Principal: domain/JugadorSystem.java

Proposito: Actuar como el "cerebro" del jugador, decidiendo que hacer basandose en la entrada.

3.1 Logica Detallada

En cada fotograma, el metodo update() de JugadorSystem se ejecuta. Lee las banderas de ManejadorTeclas. Basandose en que teclas estan presionadas, establece la propiedad direction en el Transform del jugador (ej. "up", "down"). Esto es importante para que el sistema de renderizado sepa que sprite dibujar.

3.2 Codigo Ilustrativo (JugadorSystem.java)

```
public class JugadorSystem implements IUpdateable {
    private JugadorModel jugador; // El modelo de datos del jugador
    private ManejadorTeclas keyH; // Referencia al manejador de teclas

    // Constructor que recibe las dependencias
    public JugadorSystem(JugadorModel jugador, ManejadorTeclas keyH) {
        this.jugador = jugador;
        this.keyH = keyH;
    }

    @Override
```

```
    public void update() {
        // Si cualquier tecla de movimiento esta presionada
        if (keyH.upPressed || keyH.downPressed || keyH.leftPressed || keyH.rightPressed) {
            if (keyH.upPressed) {
                jugador.getTransform().setDirection("up");
            } else if (keyH.downPressed) {
                jugador.getTransform().setDirection("down");
            } else if (keyH.leftPressed) {
                jugador.getTransform().setDirection("left");
            } else if (keyH.rightPressed) {
                jugador.getTransform().setDirection("right");
            }
            jugador.setMoving(true);
        } else {
            jugador.setMoving(false);
        }
    }
}
```

4. DETECCION PREDICTIVA DE COLISIONES

Antes de cambiar las coordenadas del jugador, es imperativo comprobar si el movimiento es valido.

Archivo Principal: domain/ColisionSystem.java

Proposito: Prevenir que una entidad atraviese tiles marcados como solidos.

4.1 Logica Detallada

1. El ColisionSystem tiene un metodo checkTileCollision(EntidadModel entity).

2. Este metodo no mira la posicion actual, sino la posicion futura. Calcula hacia donde se movera la caja de colisiones (solidArea) de la entidad en el siguiente fotograma basandose en su direccion y velocidad.

3. Convierte estas coordenadas futuras del mundo a coordenadas de la cuadricula de tiles (ej. tileNum = worldX / tileSize).

4. Comprueba los dos tiles hacia los que se dirige la entidad (ej. si se mueve a la derecha, comprueba los tiles que tocarian la esquina superior derecha e inferior derecha de su solidArea).

5. Obtiene esos Tile del mapa y verifica su propiedad isSolid().

6. Si cualquiera de los tiles futuros es solido, establece una bandera collisionOn en el Transform de la entidad a true.

5. ACTUALIZACION FINAL DE LA POSICION (MOVIMIENTO FISICO)

Solo si el camino esta despejado, el jugador se mueve.

Archivo Principal: domain/MovimientoSystem.java

Proposito: Aplicar el cambio final de coordenadas a todas las entidades moviles.

5.1 Logica Detallada

1. El metodo update() de MovimientoSystem se ejecuta para cada entidad movil, incluido el jugador.

2. Primero, resetea la bandera de colision: entity.getTransform().setCollisionOn(false).

3. Invoca al ColisionSystem para que realice la comprobacion predictiva: colisionSystem.checkTileCollision(entity).

4. A continuacion, comprueba si la entidad tiene intencion de moverse (isMoving()) y si su camino no esta bloqueado (!getTransform().isCollisionOn()).

5. Solo si ambas condiciones son ciertas, modifica las coordenadas worldX y worldY del Transform de la entidad segun su direction y speed.

5.2 Codigo Ilustrativo (MovimientoSystem.java)

```
public class MovimientoSystem implements IUpdateable {
    private ColisionSystem colisionSystem;
```

```
// ...

public void update(EntidadModel entity) {
    // 1. Resetear y comprobar colisiones para el frame actual
    entity.getTransform().setCollisionOn(false);
    colisionSystem.checkTileCollision(entity);

    // 2. Mover solo si no hay colision
    if (entity.isMoving() && !entity.getTransform().isCollisionOn()) {
        switch (entity.getTransform().getDirection()) {
            case "up":
                entity.getTransform().y -= entity.getTransform().getSpeed();
                break;
            case "down":
                entity.getTransform().y += entity.getTransform().getSpeed();
                break;
            case "left":
                entity.getTransform().x -= entity.getTransform().getSpeed();
                break;
            case "right":
                entity.getTransform().x += entity.getTransform().getSpeed();
                break;
        }
    }

    // 3. Actualizar contador de animacion para el renderizado
    // (Esta logica tambien podria estar en JugadorSystem)
    entity.incrementSpriteCounter();
    if (entity.getSpriteCounter() > 10) { // Cambia de sprite cada 10 frames
        entity.incrementSpriteNum();
```

```
            entity.resetSpriteCounter();
        }
    }
}
```

Movimiento de Enemigos

Este documento describe la inteligencia artificial (IA) que gobierna el comportamiento de los enemigos, con un enfoque en su sistema de movimiento, detección y persecución del jugador.

Sección 1: Arquitectura de la IA - Un Modelo de Estados Finitos (FSM)

La "inteligencia" de cada enemigo se basa en una Máquina de Estados Finitos (FSM). Esto significa que un enemigo solo puede estar en un estado de comportamiento a la vez (ej. PATRULLANDO, PERSIGUIENDO) y transita entre estos estados basándose en estímulos externos, principalmente la proximidad del jugador.

Componentes Clave de la Arquitectura:

- domain/EnemigoSystem.java: Es el "cerebro" del enemigo. Su método update() contiene la FSM y toma todas las decisiones de alto nivel (qué hacer y hacia dónde moverse).

- model/EnemigoModel.java: Es la "memoria" y el "cuerpo". Almacena el estado actual de un enemigo (enum AIState), su posición, velocidad, vida, y parámetros de IA como su radio de patrullaje y su radio de agresividad.

- domain/MovimientoSystem.java y domain/ColisionSystem.java: Son las "piernas". Reciben la intención de movimiento del EnemigoSystem y se encargan de la ejecución física, aplicando la velocidad, actualizando coordenadas y evitando obstáculos, de forma idéntica a como lo hacen para el jugador.

Sección 2: El Estado Pasivo - Patrullaje Dentro de un Radio

Este es el comportamiento por defecto del enemigo cuando no es consciente de la presencia del jugador.

- Propósito: Dar vida al mundo, haciendo que los enemigos no sean estatuas estáticas. Se mueven de forma semi-aleatoria dentro de un área predefinida.

- Técnica - Área de Patrullaje:

1. Cada EnemigoModel almacena sus coordenadas de origen (originX, originY), que son las coordenadas donde fue generado.

2. También tiene un patrolRadius, que define la distancia máxima a la que puede alejarse de su origen.

- Lógica Detallada:

1. El EnemigoSystem utiliza un contador de tiempo para cada enemigo (actionLockCounter).

2. Este contador se incrementa en cada fotograma. Cuando supera un umbral (ej. 120 frames, o 2 segundos a 60 FPS), el enemigo toma una nueva decisión.

3. Se elige una dirección al azar (arriba, abajo, izquierda o derecha).

4. El sistema establece esta nueva dirección en el Transform del enemigo y resetea el contador.

5. El enemigo se moverá en esa dirección (manejado por MovimientoSystem) hasta que el contador expire de nuevo o hasta que choque con una pared (manejado por ColisionSystem).

(Nota: Una implementación más avanzada comprobaría si el movimiento aleatorio lo saca de su radio de patrulla y, en ese caso, lo forzaría a moverse hacia su origen.)

- Código Ilustrativo en EnemigoSystem.java (dentro de update()):

java

```
// Asumiendo que el modelo del enemigo tiene estas propiedades
// AIState state = enemigo.getAIState();

if (state == AIState.PATROLLING) {
    enemigo.actionLockCounter++;
    if (enemigo.actionLockCounter >= 120) {
```

```
        Random random = new Random();
        int i = random.nextInt(4); // 0=arriba, 1=abajo, 2=izquierda, 3=derecha

        switch (i) {
            case 0: enemigo.getTransform().setDirection("up"); break;
            case 1: enemigo.getTransform().setDirection("down"); break;
            case 2: enemigo.getTransform().setDirection("left"); break;
            case 3: enemigo.getTransform().setDirection("right"); break;
        }
        enemigo.actionLockCounter = 0;
    }
}
```

Sección 3: Detección del Jugador - El Radio de Agresividad ("Aggro")

Este es el mecanismo que hace que el enemigo "despierte" y se vuelva hostil.

- Técnica: Cálculo de distancia euclidiana entre el enemigo y el jugador.

- Lógica Detallada y Optimización:

1. En cada fotograma, dentro de EnemigoSystem.update(), lo primero es calcular la distancia al jugador.

2. Optimización Clave: Calcular una raíz cuadrada (Math.sqrt) es una operación computacionalmente costosa. Para evitarla, se trabaja con distancias al cuadrado.

3. Se obtiene la diferencia de coordenadas: dx = jugador.worldX - enemigo.worldX y dy = jugador.worldY - enemigo.worldY.

4. Se calcula la distancia al cuadrado: distanciaSq = (dx dx) + (dy dy).

5. Se compara este valor con el radio de agresividad del enemigo, también al cuadrado: aggroRadiusSq = enemigo.aggroRadius enemigo.aggroRadius.

6. Si distanciaSq < aggroRadiusSq, el jugador ha entrado en el radio. El estado del enemigo cambia inmediatamente de PATROLLING a CHASING.

- Código Ilustrativo en EnemigoSystem.java (dentro de update()):

```
java

// Obtener jugador y enemigo

int dx = jugador.getTransform().x - enemigo.getTransform().x;

int dy = jugador.getTransform().y - enemigo.getTransform().y;

double distanceSq = Math.pow(dx, 2) + Math.pow(dy, 2);

double aggroRadiusSq = Math.pow(enemigo.getAggroRadius(), 2);


if (distanceSq < aggroRadiusSq) {

    enemigo.setAIState(AIState.CHASING);

} else {

    enemigo.setAIState(AIState.PATROLLING);

}
```

Sección 4: El Estado Activo - Persecución del Jugador

Una vez que el jugador es detectado, el enemigo comienza la persecución.

- Algoritmo - Seguimiento "Codicioso" (Greedy Tracking): Se utiliza un algoritmo de seguimiento simple y eficaz que no requiere un cálculo de rutas complejo (como A).

- Lógica Detallada:

1. Cuando el estado es CHASING, el sistema vuelve a calcular los deltas dx y dy.

2. Compara el valor absoluto de dx y dy para decidir si priorizar el movimiento horizontal o vertical. Esto evita un movimiento puramente diagonal que puede parecer poco natural o quedarse atascado fácilmente.

3. Si Math.abs(dx) > Math.abs(dy), el movimiento principal será horizontal. Si dx es positivo, el jugador está a la derecha, por lo que la dirección del enemigo se establece en "right". Si es negativo, se establece en "left".

4. Si Math.abs(dy) es mayor, se aplica la misma lógica para la dirección vertical ("up" o "down").

5. Perder el Rastro: Si en el estado CHASING, el cálculo de distancia revela que el jugador ha salido del radio de agresividad, el estado del enemigo vuelve a PATROLLING.

- Código Ilustrativo en EnemigoSystem.java (dentro de update()):

java

```
if (enemigo.getAIState() == AIState.CHASING) {
    // (dx y dy ya calculados para la detección)
    if (Math.abs(dx) > Math.abs(dy)) {
        if (dx > 0) {
            enemigo.getTransform().setDirection("right");
        } else {
            enemigo.getTransform().setDirection("left");
        }
    } else {
        if (dy > 0) {
            enemigo.getTransform().setDirection("down");
        } else {
            enemigo.getTransform().setDirection("up");
        }
    }
}
```

Sección 5: Ejecución del Movimiento Físico

Independientemente de si el enemigo está patrullando o persiguiendo, el resultado de la lógica de EnemigoSystem es siempre el mismo: una intención de movimiento (una dirección y un estado isMoving).

- Proceso Unificado: A partir de aquí, el proceso es idéntico al del jugador.

1. El MovimientoSystem recibe la entidad enemigo.
2. Llama al ColisionSystem para comprobar si la dirección de movimiento actual está bloqueada por un tile sólido.
3. Si ColisionSystem no reporta ninguna colisión, MovimientoSystem actualiza las coordenadas worldX y worldY del enemigo.

DOCUMENTACION DEL SISTEMA DE RENDERIZADO

Este documento describe en detalle la arquitectura y funcionamiento del sistema de renderizado del juego, incluyendo todos los componentes y sus interacciones.

1. ARQUITECTURA DE UN CHUNK

Archivo: model/Chunk.java
Logica Detallada: Un Chunk no es solo una matriz de tiles. Es un objeto que debe conocer su propia posicion en la "cuadricula de chunks" del mundo. Se identifica de manera unica mediante una clave de cadena (ej., "12;-35") para un almacenamiento y recuperacion eficientes en un HashMap.

Codigo Ilustrativo (Chunk.java):

```
public class Chunk {
    public static final int CHUNK_WIDTH = 16; // 16 tiles de ancho
    public static final int CHUNK_HEIGHT = 16; // 16 tiles de alto
    private final String id;
    private final int chunkX, chunkY; // Coordenadas en la cuadricula de chunks
    private final Tile tiles;

    public Chunk(int chunkX, int chunkY, Tile tiles) {
        this.chunkX = chunkX;
        this.chunkY = chunkY;
        this.id = chunkX + ";" + chunkY;
        this.tiles = tiles;
    }
}
```

2. EL ORQUESTADOR: MANEJADORMAPAINFINITO

Esta es una de las clases mas importantes del dominio. Su unica responsabilidad es asegurar que los chunks correctos esten en memoria en todo momento.

2.1. Logica Detallada del Metodo update

Archivo: domain/ManejadorMapaInfinito.java

1. Identificar Chunk Actual: Calcula en que chunk se encuentra la camara: camChunkX = camara.getX() / (CHUNK_WIDTH * TILE_SIZE).

2. Definir Rango de Carga: Se establece un radio de vision, RENDER_DISTANCE (ej. 2). El area a cargar sera un cuadrado de (2 * RENDER_DISTANCE + 1) chunks de lado, centrado en camChunkX.

3. Proceso de Carga: Se ejecutan dos bucles for anidados, desde camChunkX - RENDER_DISTANCE hasta camChunkX + RENDER_DISTANCE. En cada iteracion, se genera un chunkId. Si activeChunks.containsKey (chunkId) es falso, se invoca a GeneradorMundo para crear el chunk y se anade al HashMap activeChunks.

4. Proceso de Descarga (Optimizacion de Memoria): Para evitar errores de modificacion concurrente, la descarga se hace en dos pasos. Primero, se crea una List<String> chunksToRemove. Se itera sobre las claves de activeChunks. Para cada clave, se extraen sus coordenadas y se comprueba si estan fuera del rango de carga actual. Si es asi, se anade la clave a chunksToRemove. Finalmente, se itera sobre chunksToRemove y se elimina cada chunk de activeChunks.

2.2. El Creador: GeneradorMundo

Esta clase es responsable de la generacion procedural del terreno.

Archivo: domain/GeneradorMundo.java

Logica Detallada - Ruido Procedural:

Algoritmo: Se basa en funciones de ruido coherente como el Ruido Perlin o Ruido Simplex. Estas funciones matematicas generan valores pseudo-aleatorios pero suaves. Dados dos puntos cercanos (x, y) y (x+dx, y+dy), los valores de ruido resultantes seran similares, lo que permite crear terrenos de apariencia natural en lugar de ruido blanco (como estatica de TV).

Capas de Ruido: Para un terreno mas interesante, se usan multiples capas de ruido con diferentes frecuencias y amplitudes:

1. Capa de Elevacion (Baja Frecuencia): Define la forma general del terreno (montanas, valles). perlin.eval(worldX * 0.01, worldY * 0.01).

2. Capa de Bioma (Media Frecuencia): Define si un area es desierto o bosque. perlin.eval(worldX * 0.05, worldY * 0.05).

3. Capa de Detalles (Alta Frecuencia): Anade pequenos detalles como rocas o flores. perlin.eval(worldX * 0.2, worldY * 0.2).

El tipo de tile final en una coordenada (worldX, worldY) se decide combinando los valores de estas capas.

Codigo Ilustrativo (GeneradorMundo.java):

```
public class GeneradorMundo {
    private OpenSimplexNoise noiseGenerator;

    public GeneradorMundo(long seed) {
        this.noiseGenerator = new OpenSimplexNoise(seed);
    }

    public Chunk generarChunk(int chunkX, int chunkY) {
        Tile tiles = new TileCHUNK_WIDTHCHUNK_HEIGHT;
        for (int x = 0; x < Chunk.CHUNK_WIDTH; x++) {
            for (int y = 0; y < Chunk.CHUNK_HEIGHT; y++) {
                int worldX = (chunkX * Chunk.CHUNK_WIDTH + x) * TILE_SIZE;
                int worldY = (chunkY * Chunk.CHUNK_HEIGHT + y) * TILE_SIZE;

                // Frecuencia y amplitud controlan la "escala" y "rugosidad"
                double elevation = noiseGenerator.eval(worldX * 0.01,
                                                       worldY * 0.01);

                TileModel modelToUse = (elevation > 0.4) ? TileModel.PIEDRA
                                                         : TileModel.PASTO;
                tilesxy = new Tile(modelToUse, worldX, worldY);
            }
        }
        return new Chunk(chunkX, chunkY, tiles);
    }
}
```

3. EL PROCESO DE RENDERIZADO - DE DATOS A PIXELES

Esta seccion detalla el bucle de renderizado que se ejecuta en cada fotograma.

3.1. El Director de Escena: RenderSystem

Archivo: Presentation/RenderSystem.java

Logica Detallada: Actua como el punto de entrada principal para todo el dibujo. Su metodo render es llamado desde GamePanel.paintComponent. Este metodo orquesta el orden de las operaciones de dibujo para asegurar un resultado correcto:

1. renderTiles(g2, camara): Dibuja el fondo y el mapa.
2. renderEntities(g2, camara): Dibuja todas las entidades moviles (jugador, enemigos, etc.).
3. renderHUD(g2): Dibuja la interfaz de usuario (vida, etc.) que no se mueve con la camara.

3.2. Renderizado del Mapa (Tiles) y la Optimizacion Clave

Logica Detallada:

1. Obtiene la lista de activeChunks del ManejadorMapaInfinito.
2. Itera sobre cada Chunk y luego sobre cada Tile dentro de el.
3. Conversion de Coordenadas: screenX = tile.getWorldX() - camara.getX();
4. Optimizacion - Frustum Culling (Descarte por Frustrum): Esta es la optimizacion mas importante aqui. Antes de intentar dibujar un tile, se comprueba si su rectangulo de colision (BoundingBox) se superpone con el rectangulo de la pantalla. Si un tile esta completamente a la izquierda, derecha, arriba o abajo de la pantalla, no tiene sentido gastar ciclos de CPU en dibujarlo.

Condicion de Frustum Culling:

```
if (screenX + TILE_SIZE > 0 &&      // Borde derecho visible
    screenX < screenWidth &&        // Borde izquierdo visible
    screenY + TILE_SIZE > 0 &&      // Borde inferior visible
    screenY < screenHeight) {       // Borde superior visible

    g2.drawImage(tile.getImage(), screenX, screenY, null);
}
```

3.3. Renderizado de Entidades y el Algoritmo del Pintor

Logica Detallada:

1. Recopilacion: Se crea una ArrayList<EntidadModel> temporal.
2. Se anaden a la lista el jugador y todos los enemigos activos obtenidos del motor de juego.
3. Ordenacion - Algoritmo del Pintor: Se ordena la lista. El criterio de ordenacion es la coordenada worldY de la base de la entidad (transform.y + altura_entidad). Esto asegura que las entidades que estan mas "abajo" en el mapa se dibujen despues, y por lo tanto, "delante" de las que estan mas "arriba".

```
entitiesToRender.sort(Comparator.comparingInt(
    e -> e.getTransform().y + e.getTransform().height));
```

4. Dibujo: Se itera sobre la lista ya ordenada. Para cada entidad, se realiza el mismo Frustum Culling que con los tiles y luego se delega el dibujo a un renderer especializado.

3.4. El Especialista: EnemigoRenderer y la Animacion

Archivo: Presentation/EnemigoRenderer.java

Logica Detallada:

1. El metodo render(g2, enemigo, camara) recibe todo lo necesario.

2. Gestion de Estado de Animacion: El EnemigoModel debe contener su estado actual, ej: enum Estado { IDLE, WALKING, ATTACKING } y enum Direccion { UP, DOWN, LEFT, RIGHT }.

3. Gestion del Tiempo de Animacion: El EnemigoModel tambien debe tener un contador (spriteNum) y un temporizador (spriteCounter). En el bucle de update del enemigo (en EnemigoSystem), el spriteCounter se incrementa. Cuando supera un umbral (ej. 10), spriteNum cambia y el contador se resetea. Esto controla la velocidad de la animacion.

4. Seleccion Dinamica de Sprites: El EnemigoRenderer usa estos estados para construir la ruta de la imagen a cargar o para seleccionarla de un SpriteSheet ya cargado. Por ejemplo, si el estado es WALKING, la direccion es LEFT y spriteNum es 1, el renderer buscara la imagen Left - Walking_001.png.

5. Dibuja la imagen seleccionada en la posicion de pantalla calculada del enemigo.

4. SISTEMA DE RENDERIZADO COMPLETO - ANALISIS ARQUITECTONICO

Esta seccion profundiza en la arquitectura completa del sistema de renderizado implementado en el juego, incluyendo todos los componentes y sus interacciones.

4.1. Arquitectura General del Sistema de Renderizado

Diagrama de Componentes:

```
GamePanel (Main)
  |
  |--- Bucle principal de renderizado (paintComponent)
  |--- Control de FPS y sincronizacion
  |--- Delegacion a RenderSystem
  |
  v
RenderSystem (Presentation)
  |
  |--- Orquestador principal de renderizado
  |--- Manejo de orden de capas (Z-ordering)
  |--- Delegacion a renderers especializados
  |
  +--- Tile Renderer
  +--- Entidad Renderer
  +--- Enemigo Renderer
  +--- HUD Renderer
  |
  v
Graphics2D (Java API de dibujo)
```

4.2. El Bucle Principal: GamePanel.paintComponent()

Archivo: Main/GamePanel.java

El metodo paintComponent es el corazon del sistema de renderizado. Es llamado automaticamente por Swing cada vez que el panel necesita ser redibujado.

Codigo Completo con Explicaciones:

```
  @Override
  protected void paintComponent(Graphics g) {
      // PASO 1: Limpiar el frame anterior
      // super.paintComponent() limpia el buffer de dibujo y prepara
      // un lienzo limpio. Sin esto, los frames se superpondrian.
      super.paintComponent(g);

      // PASO 2: Obtener contexto de dibujo 2D
      // Graphics2D proporciona metodos avanzados de renderizado
      // como transformaciones, anti-aliasing, y composiciones.
      Graphics2D g2 = (Graphics2D) g;

      // PASO 3: Configurar calidad de renderizado (Opcional)
      g2.setRenderingHint(
          RenderingHints.KEY_ANTIALIASING,
          RenderingHints.VALUE_ANTIALIAS_ON
      );
      g2.setRenderingHint(
          RenderingHints.KEY_TEXT_ANTIALIASING,
          RenderingHints.VALUE_TEXT_ANTIALIAS_ON
      );
```

```
// PASO 4: Renderizar segun el estado del juego
        switch (estadoJuego) {
            case BIENVENIDA:
                renderSystem.renderBienvenida(g2, config.getAnchoPantalla(),
                                              config.getAltoPantalla());
                break;

            case MENU_PRINCIPAL:
                renderSystem.renderMenuPrincipal(g2, config.getAnchoPantalla(),
                                                 config.getAltoPantalla());
                break;

            case JUGANDO:
            case PAUSA:
            case MENU_PAUSA:
            case MODAL_COFRE:
            case MODAL_RESPUESTA_CORRECTA:
            case MODAL_RESPUESTA_INCORRECTA:
                // RENDERIZADO DEL JUEGO PRINCIPAL
                renderSystem.renderTodo(
                    g2,
                    gameEngine.getMapaInfinito(),
                    gameEngine.getCamaraSystem(),
                    gameEngine.getJugadorSystem(),
                    gameEngine.getEnemigoSystem(),
                    config.getAnchoPantalla(),
                    config.getAltoPantalla()
                );

                // RENDERIZADO DE OVERLAYS (segun estado)
```

```
            if (estadoJuego == GameState.PAUSA) {
                renderSystem.renderPausa(g2, config.getAnchoPantalla(),
                                         config.getAltoPantalla());
            } else if (estadoJuego == GameState.MENU_PAUSA) {
                renderSystem.renderMenuPausa(g2, config.getAnchoPantalla(),
                                             config.getAltoPantalla());
            } else if (estadoJuego == GameState.MODAL_COFRE) {
                renderSystem.getHudRenderer().renderCofreModal(
                    g2, config.getAnchoPantalla(), config.getAltoPantalla(),
                    gameEngine.getAcertijoActual()
                );
            }
            // ... otros overlays
            break;
        }

        // PASO 5: Liberar recursos de Graphics2D
        // Importante para prevenir memory leaks
        g2.dispose();
    }
```

Analisis del Flujo:

1. Limpieza del Buffer: super.paintComponent() asegura que no haya "ghosting" de frames anteriores
2. Casting a Graphics2D: Permite acceso a funciones avanzadas de renderizado
3. Configuracion de Calidad: Anti-aliasing mejora la calidad visual
4. Renderizado por Capas: Cada elemento se dibuja en orden especifico
5. Liberacion de Recursos: dispose() previene memory leaks

4.3. RenderSystem - El Orquestador Principal

Archivo: Presentation/RenderSystem.java

Codigo Completo Detallado:

```
public class RenderSystem {
    private EntidadRenderer entidadRenderer;
    private EnemigoRenderer enemigoRenderer;
    private HUDRenderer hudRenderer;
    private int tamanioTile;

    public RenderSystem(int tamanioTile) {
        this.tamanioTile = tamanioTile;
        this.entidadRenderer = new EntidadRenderer(tamanioTile);
        this.enemigoRenderer = new EnemigoRenderer(tamanioTile);
        this.hudRenderer = new HUDRenderer();
    }

    /** Metodo principal de renderizado del juego.
     *  Orden de renderizado (importante para z-index correcto):
     *  1. Fondo/Sky
     *  2. Tiles del mapa
     *  3. Items consumibles
     *  4. Entidades (ordenadas por Y)
     *  5. Efectos de particulas
     *  6. HUD
     */
    public void renderTodo(Graphics2D g2,
                           ManejadorMapaInfinito mapa,
```

```
                    CamaraSystem camara,
                    JugadorSystem jugador,
                    EnemigoSystem enemigos,
                    int anchoPantalla,
                    int altoPantalla) {

        // CAPA 1: Fondo - Color solido o imagen de cielo
        g2.setColor(new Color(135, 206, 235)); // Celeste cielo
        g2.fillRect(0, 0, anchoPantalla, altoPantalla);

        // CAPA 2: Renderizar Mapa (Tiles)
        renderizarMapa(g2, mapa, camara, anchoPantalla, altoPantalla);

        // CAPA 3: Renderizar Items Consumibles
        renderizarItemsConsumibles(g2, mapa, camara, anchoPantalla,
                                   altoPantalla);

        // CAPA 4: Renderizar Entidades (con ordenamiento Y)
        renderizarEntidades(g2, jugador, enemigos, camara,
                            anchoPantalla, altoPantalla);

        // CAPA 5: HUD (siempre encima)
        hudRenderer.render(g2, mapa, jugador, tamanioTile,
                           enemigos.getEnemigos().size(), anchoPantalla);
    }

    // Metodos detallados en secciones siguientes...
}
```

4.4. Renderizado del Mapa - Analisis Profundo

Metodo: renderizarMapa()

Desafios Tecnicos:

1. El mundo es infinito, pero la pantalla es finita
2. Solo debemos procesar tiles visibles (optimizacion)
3. Los tiles deben convertirse de coordenadas del mundo a coordenadas de pantalla

Algoritmo Paso a Paso:

```
PARA CADA chunk activo EN el mapa:
    PARA CADA fila DE tiles EN el chunk:
        PARA CADA tile EN la fila:

            Paso 1: Calcular posicion en pantalla
            screenX = tile.worldX - camara.worldX
            screenY = tile.worldY - camara.worldY

            Paso 2: Frustum Culling (optimizacion critica)
            SI (screenX + tamanioTile < 0)       -> tile fuera izquierda
            O  (screenX > anchoPantalla)         -> tile fuera derecha
            O  (screenY + tamanioTile < 0)       -> tile fuera arriba
            O  (screenY > altoPantalla)          -> tile fuera abajo
            ENTONCES:
                CONTINUAR  // Saltar este tile, no dibujarlo
            FIN SI
```

```
            Paso 3: Dibujar el tile
            g2.drawImage(tile.imagen, screenX, screenY, tamanioTile,
                    tamanioTile, null)

            Paso 4 (Opcional): Dibujar grid de debug
            SI (modoDebug):
               g2.setColor(Color.GRAY)
               g2.drawRect(screenX, screenY, tamanioTile, tamanioTile)
            FIN SI

         FIN PARA
      FIN PARA
   FIN PARA
```

Codigo Java Implementado:

```
   private void renderizarMapa(Graphics2D g2, ManejadorMapaInfinito mapa,
                        CamaraSystem camara, int anchoPantalla,
                        int altoPantalla) {

      // Obtener coordenadas de la camara
      int camaraX = camara.getMundoX();
      int camaraY = camara.getMundoY();

      // Calcular rango de tiles visibles (optimizacion adicional)
      int primerTileX = Math.max(0, camaraX / tamanioTile - 1);
      int primerTileY = Math.max(0, camaraY / tamanioTile - 1);
      int ultimoTileX = (camaraX + anchoPantalla) / tamanioTile + 1;
      int ultimoTileY = (camaraY + altoPantalla) / tamanioTile + 1;
```

```
    // Iterar solo sobre el rango visible
    for (int tileY = primerTileY; tileY <= ultimoTileY; tileY++) {
        for (int tileX = primerTileX; tileX <= ultimoTileX; tileX++) {

            // Obtener el tile del mapa
            int tileNum = mapa.getTileEnMundo(tileX, tileY);

            if (tileNum == -1) continue; // Tile no generado aun

            // Calcular posicion en pantalla
            int screenX = (tileX * tamanioTile) - camaraX;
            int screenY = (tileY * tamanioTile) - camaraY;

            // Frustum culling (verificacion redundante pero segura)
            if (screenX + tamanioTile < 0 || screenX > anchoPantalla ||
                screenY + tamanioTile < 0 || screenY > altoPantalla) {
                continue;
            }

            // Obtener imagen del tile
            BufferedImage imagen = mapa.getTiles()tileNum.getImage();

            // Dibujar el tile
            g2.drawImage(imagen, screenX, screenY,
                    tamanioTile, tamanioTile, null);
        }
    }
}
```

Analisis de Rendimiento:

Sin Frustum Culling:

Tiles procesados por frame: Todo el mundo cargado (~16,000 tiles)

FPS: ~15-20 (inaceptable)

Con Frustum Culling:

Tiles procesados por frame: Solo visibles (~300-400 tiles)

FPS: ~60 (optimo)

Mejora de rendimiento: ~98% de reduccion en operaciones de dibujo

4.5. Renderizado de Entidades con Algoritmo del Pintor

Problema: En que orden dibujar las entidades para simular profundidad?

Solucion: Algoritmo del Pintor (Painters Algorithm)

Concepto:

- Las entidades "mas atras" (menor Y) se dibujan primero
- Las entidades "mas adelante" (mayor Y) se dibujan despues
- Las posteriores sobreescriben las anteriores, creando ilusion de profundidad

Visualizacion ASCII:

Vista superior del mapa:

Y=100 Enemigo A

Y=150 Jugador

Y=200 Enemigo B

Orden de dibujo: A -> Jugador -> B

Resultado en pantalla:

B aparece "delante" del Jugador

Jugador aparece "delante" de A

Codigo Implementado:

```
private void renderizarEntidades(Graphics2D g2, JugadorSystem jugador,
                                 EnemigoSystem enemigos, CamaraSystem camara,
                                 int anchoPantalla, int altoPantalla) {

    // PASO 1: Recopilar todas las entidades en una lista temporal
    List<Object> entidades = new ArrayList<>();

    // Agregar jugador
    entidades.add(jugador);

    // Agregar todos los enemigos
    entidades.addAll(enemigos.getEnemigos());
```

```
// PASO 2: Ordenar por coordenada Y (Algoritmo del Pintor)
// Usamos la parte inferior de la entidad para mayor precision
entidades.sort((e1, e2) -> {
    int y1 = obtenerYBase(e1);
    int y2 = obtenerYBase(e2);
    return Integer.compare(y1, y2);
});

// PASO 3: Renderizar en orden
for (Object entidad : entidades) {
    if (entidad instanceof JugadorSystem) {
        JugadorSystem j = (JugadorSystem) entidad;

        // Calcular posicion en pantalla
        int screenX = j.getMundoX() - camara.getMundoX();
        int screenY = j.getMundoY() - camara.getMundoY();

        // Frustum culling
        if (!estaEnPantalla(screenX, screenY, tamanioTile,
                        anchoPantalla, altoPantalla)) {
            continue;
        }

        // Delegar renderizado
        entidadRenderer.render(g2, j.getJugador(), camara);

    } else if (entidad instanceof EnemigoModel) {
        EnemigoModel enemigo = (EnemigoModel) entidad;
```

```
            // Similar para enemigos
            int screenX = enemigo.getTransform().getX() -
                    camara.getMundoX();
            int screenY = enemigo.getTransform().getY() -
                    camara.getMundoY();

            if (!estaEnPantalla(screenX, screenY, tamanioTile,
                        anchoPantalla, altoPantalla)) {
                continue;
            }

            enemigoRenderer.render(g2, enemigo, camara);
        }
    }
}

// Metodo auxiliar para obtener Y base de cualquier entidad
private int obtenerYBase(Object entidad) {
    if (entidad instanceof JugadorSystem) {
        JugadorSystem j = (JugadorSystem) entidad;
        return j.getMundoY() + tamanioTile; // Base del sprite
    } else if (entidad instanceof EnemigoModel) {
        EnemigoModel e = (EnemigoModel) entidad;
        return e.getTransform().getY() + tamanioTile;
    }
    return 0;
}
```

```
// Metodo auxiliar para frustum culling
private boolean estaEnPantalla(int screenX, int screenY, int size,
                               int anchoPantalla, int altoPantalla) {
    return !(screenX + size < 0 || screenX > anchoPantalla ||
             screenY + size < 0 || screenY > altoPantalla);
}
```

4.6. Sistema de Animacion de Sprites

Concepto: Las animaciones se logran alternando entre multiples imagenes (frames) a una velocidad controlada.

Componentes del Sistema:

1. SpriteData: Almacena todas las imagenes de una entidad
2. AnimacionSystem: Controla que frame mostrar y cuando cambiar
3. Contador de Frames: Sincroniza con el bucle del juego

Estructura de SpriteData:

```
public class SpriteData {
    // Sprites por direccion y frame
    private BufferedImage arriba1, arriba2;
    private BufferedImage abajo1, abajo2;
    private BufferedImage izquierda1, izquierda2;
    private BufferedImage derecha1, derecha2;

    // Estado actual
    private String direccion = "abajo";
    private int spriteNum = 1;

    // Timing
    private int spriteCounter = 0;
    private final int SPRITE_CHANGE_INTERVAL = 12; // Cambiar cada 12 frames

    public BufferedImage getSpriteActual() {
        switch (direccion) {
            case "arriba":
                return (spriteNum == 1) ? arriba1 : arriba2;
            case "abajo":
                return (spriteNum == 1) ? abajo1 : abajo2;
            case "izquierda":
                return (spriteNum == 1) ? izquierda1 : izquierda2;
            case "derecha":
                return (spriteNum == 1) ? derecha1 : derecha2;
            default:
                return abajo1;
        }
```

```
    }
  }
```

Sistema de Actualizacion:

```
  public class AnimacionSystem {
      private SpriteData spriteData;

      public void update() {
          spriteData.spriteCounter++;

          if (spriteData.spriteCounter > spriteData.SPRITE_CHANGE_INTERVAL) {
              // Alternar entre frame 1 y 2
              if (spriteData.spriteNum == 1) {
                  spriteData.spriteNum = 2;
              } else {
                  spriteData.spriteNum = 1;
              }
              spriteData.spriteCounter = 0;
          }
      }
  }
```

Diagrama de Flujo de Animacion:

Frame 0: spriteCounter=0, spriteNum=1 -> Muestra sprite1

Frame 1: spriteCounter=1, spriteNum=1 -> Muestra sprite1

Frame 2: spriteCounter=2, spriteNum=1 -> Muestra sprite1

...

Frame 12: spriteCounter=12, spriteNum=1 -> Muestra sprite1

Frame 13: spriteCounter=0, spriteNum=2 -> Muestra sprite2 (CAMBIO)

Frame 14: spriteCounter=1, spriteNum=2 -> Muestra sprite2

...

Frame 25: spriteCounter=12, spriteNum=2 -> Muestra sprite2

Frame 26: spriteCounter=0, spriteNum=1 -> Muestra sprite1 (CAMBIO)

Velocidad de Animacion:

A 60 FPS:

SPRITE_CHANGE_INTERVAL = 12

Cambio cada 12/60 = 0.2 segundos

5 cambios por segundo

Animacion suave y natural

A 30 FPS:

SPRITE_CHANGE_INTERVAL = 6

Cambio cada 6/30 = 0.2 segundos

Mantiene misma velocidad visual

4.7. HUD Renderer - Renderizado de Interfaz

Archivo: Presentation/HUDRenderer.java

Componentes del HUD:

1. Barra de Vida
2. Contador de Pociones
3. Informacion de Debug
4. Acertijos Resueltos

Codigo Completo de Barra de Vida:

```
public void renderBarraVida(Graphics2D g2, double vidaActual,
                            double vidaMaxima, int pantallaAncho) {

    // Configuracion de la barra
    int anchoBarra = 200;
    int altoBarra = 20;
    int xBarra = (pantallaAncho / 2) - (anchoBarra / 2);
    int yBarra = 15;
    int bordeGrosor = 2;

    // Calcular porcentaje de vida
    double porcentajeVida = vidaActual / vidaMaxima;
    int anchoVida = (int)(anchoBarra * porcentajeVida);

    // CAPA 1: Sombra (efecto 3D)
    g2.setColor(new Color(0, 0, 0, 100));
    g2.fillRect(xBarra + 2, yBarra + 2, anchoBarra, altoBarra);
```

```
// CAPA 2: Fondo de la barra (vacio)
   g2.setColor(new Color(60, 60, 60));
   g2.fillRect(xBarra, yBarra, anchoBarra, altoBarra);

   // CAPA 3: Vida restante (color segun porcentaje)
   Color colorVida;
   if (porcentajeVida > 0.6) {
      colorVida = new Color(50, 205, 50); // Verde
   } else if (porcentajeVida > 0.3) {
      colorVida = new Color(255, 165, 0); // Naranja
   } else {
      colorVida = new Color(220, 20, 60); // Rojo
   }
   g2.setColor(colorVida);
   g2.fillRect(xBarra, yBarra, anchoVida, altoBarra);

   // CAPA 4: Efecto de brillo
   GradientPaint gradient = new GradientPaint(
      xBarra, yBarra, new Color(255, 255, 255, 100),
      xBarra, yBarra + altoBarra/2, new Color(255, 255, 255, 0)
   );
   g2.setPaint(gradient);
   g2.fillRect(xBarra, yBarra, anchoVida, altoBarra/2);

   // CAPA 5: Borde
   g2.setColor(Color.WHITE);
   g2.setStroke(new BasicStroke(bordeGrosor));
   g2.drawRect(xBarra, yBarra, anchoBarra, altoBarra);
```

```
// CAPA 6: Texto
    g2.setFont(new Font("Arial", Font.BOLD, 14));
    String textoVida = (int)vidaActual + " / " + (int)vidaMaxima;
    FontMetrics fm = g2.getFontMetrics();
    int textoX = xBarra + (anchoBarra - fm.stringWidth(textoVida)) / 2;
    int textoY = yBarra + altoBarra/2 + fm.getAscent()/2 - 2;

    // Texto con contorno para legibilidad
    g2.setColor(Color.BLACK);
    g2.drawString(textoVida, textoX-1, textoY);
    g2.drawString(textoVida, textoX+1, textoY);
    g2.drawString(textoVida, textoX, textoY-1);
    g2.drawString(textoVida, textoX, textoY+1);

    g2.setColor(Color.WHITE);
    g2.drawString(textoVida, textoX, textoY);
}
```

4.8. Optimizaciones Avanzadas

4.8.1. Double Buffering

Problema: Flickering (parpadeo) al redibujar

Solucion: Java Swing usa double buffering automaticamente cuando se llama a setDoubleBuffered(true) en el JPanel.

Como funciona:

```
Buffer Frontal (visible):  Frame N-1
                           ^
                           | swap()
                           |
Buffer Trasero (dibujo):   Frame N <- paintComponent dibuja aqui
```

Codigo en GamePanel:

```
public GamePanel() {
    this.setDoubleBuffered(true); // Activa double buffering
    // ...
}
```

4.8.2. Sprite Caching

Problema: Cargar imagenes del disco en cada frame es lento

Solucion: ResourceLoader con cache de imagenes

```
public class ResourceLoader {
    private static ResourceLoader instance;
    private Map<String, BufferedImage> imageCache;

    private ResourceLoader() {
        this.imageCache = new HashMap<>();
    }

    public BufferedImage cargarImagen(String ruta) {
        // Verificar cache primero
        if (imageCache.containsKey(ruta)) {
            return imageCache.get(ruta);
        }

        // Si no esta en cache, cargar del disco
        try {
            BufferedImage imagen = ImageIO.read(new File(ruta));
            imageCache.put(ruta, imagen);
            return imagen;
        } catch (IOException e) {
            System.err.println("Error cargando imagen: " + ruta);
            return null;
        }
    }
```

```
    public void limpiarCache() {
        imageCache.clear();
    }
}
```

4.8.3. Dirty Rectangle Optimization

Concepto: Solo redibujar las areas que cambiaron

Nota: En este juego no se implementa porque con movimiento constante, casi toda la pantalla cambia cada frame. Seria util en juegos mas estaticos.

4.9. Metricas de Rendimiento

Mediciones en un sistema tipico:

Configuracion de Prueba:

Resolucion: 1280x720

Tiles visibles: ~360

Entidades: 1 jugador + 20 enemigos

FPS objetivo: 60

Tiempos de Renderizado (en milisegundos):

Renderizar Mapa:	2-3 ms	(40%)
Renderizar Entidades:	1-2 ms	(25%)
Renderizar HUD:	0.5 ms	(8%)
Algoritmo del Pintor:	0.3 ms	(5%)
Overhead de Graphics2D:	1-2 ms	(22%)

TOTAL:	~6-9 ms	

Margen disponible para 60 FPS: 16.67 ms

Uso actual: 6-9 ms (36-54%)

Headroom: 7-10 ms (suficiente)

4.10. Diagrama de Flujo Completo del Sistema de Renderizado

```
INICIO DEL FRAME
  |
  v
GamePanel.paintComponent() llamado por Swing
  - Limpia buffer con super.paintComponent()
  - Obtiene Graphics2D context
  |
  v
RenderSystem.renderTodo()
  |
  +--- Renderizar Fondo (Color solido)
  |
  +--- Actualizar Camara (Seguir al jugador)
  |
  v
Renderizar Mapa
  - Por cada chunk
  - Frustum culling
  - Conversion coords
  |
  v
Renderizar Items
  - Pociones/Venenos
  |
  v
Recopilar Entidades
  - Jugador
  - Enemigos
```

|

v

Algoritmo del Pintor

- Ordenar por Y

|

v

Renderizar Entidades

- Por cada entidad

- Frustum culling

- Animacion

|

v

Renderizar HUD

- Barra de vida

- Pociones

- Debug info

|

v

Renderizar Overlays

- Menu pausa

- Modales

|

v

g2.dispose() - Liberar recursos

|

v

Buffer swap - Mostrar frame en pantalla

|

v

FIN DEL FRAME (Esperar siguiente llamada)

5. CASOS ESPECIALES Y SOLUCION DE PROBLEMAS

5.1. Renderizado de Modales y Overlays

Tecnica: Semi-transparencia con composiciones

```
public void renderModalCofre(Graphics2D g2, int anchoPantalla,
                             int altoPantalla, Acertijo acertijo) {

    // CAPA 1: Fondo oscurecido (overlay)
    g2.setComposite(AlphaComposite.getInstance(
        AlphaComposite.SRC_OVER, 0.7f)); // 70% opaco
    g2.setColor(Color.BLACK);
    g2.fillRect(0, 0, anchoPantalla, altoPantalla);

    // CAPA 2: Ventana del modal (opaca)
    g2.setComposite(AlphaComposite.getInstance(
        AlphaComposite.SRC_OVER, 1.0f)); // 100% opaco

    int modalAncho = 600;
    int modalAlto = 400;
    int modalX = (anchoPantalla - modalAncho) / 2;
    int modalY = (altoPantalla - modalAlto) / 2;

    // Fondo del modal con gradiente
    GradientPaint gradient = new GradientPaint(
        modalX, modalY, new Color(40, 40, 60),
        modalX, modalY + modalAlto, new Color(20, 20, 40)
    );
    g2.setPaint(gradient);
```

```
        g2.fillRoundRect(modalX, modalY, modalAncho, modalAlto, 20, 20);

        // Borde brillante
        g2.setColor(new Color(100, 200, 255));
        g2.setStroke(new BasicStroke(3));
        g2.drawRoundRect(modalX, modalY, modalAncho, modalAlto, 20, 20);

        // CAPA 3: Contenido del modal
        renderizarAcertijo(g2, acertijo, modalX, modalY, modalAncho);
    }
```

5.2. Manejo de Diferentes Resoluciones

Problema: El juego debe verse bien en diferentes tamanos de pantalla

Solucion: Escalado proporcional

```
    public class GameConfig {
        private final int tamanioOriginalTile = 16;
        private final int escala;
        private final int tamanioTile;

        public GameConfig() {
            // Detectar resolucion de pantalla
            Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();

            // Calcular escala apropiada
            if (screenSize.width >= 1920) {
                this.escala = 4; // 4K o superior
            } else if (screenSize.width >= 1280) {
```

```
            this.escala = 3; // Full HD
        } else {
            this.escala = 2; // HD o inferior
        }

        this.tamanioTile = tamanioOriginalTile * escala;
    }
}
```

5.3. Debugging Visual del Sistema de Renderizado

```
public class DebugRenderer {
    private boolean modoDebug = false;

    public void toggleDebug() {
        modoDebug = !modoDebug;
    }

    public void renderDebugInfo(Graphics2D g2, GameEngine engine,
                                CamaraSystem camara) {
        if (!modoDebug) return;

        // Grid de tiles
        renderTileGrid(g2, camara);

        // Hitboxes de entidades
        renderHitboxes(g2, engine, camara);

        // Informacion de chunks
        renderChunkBorders(g2, engine.getMapaInfinito(), camara);
```

```
    // FPS y metricas
    renderMetrics(g2, engine);
}

private void renderTileGrid(Graphics2D g2, CamaraSystem camara) {
    g2.setColor(new Color(255, 255, 0, 50)); // Amarillo translucido
    g2.setStroke(new BasicStroke(1));

    for (int x = 0; x < anchoPantalla; x += tamanioTile) {
        g2.drawLine(x, 0, x, altoPantalla);
    }
    for (int y = 0; y < altoPantalla; y += tamanioTile) {
        g2.drawLine(0, y, anchoPantalla, y);
    }
}

private void renderHitboxes(Graphics2D g2, GameEngine engine,
                            CamaraSystem camara) {
    g2.setColor(new Color(255, 0, 0, 100)); // Rojo translucido
    g2.setStroke(new BasicStroke(2));

    // Jugador
    int jugadorX = engine.getJugadorSystem().getMundoX() -
                   camara.getMundoX();
    int jugadorY = engine.getJugadorSystem().getMundoY() -
                   camara.getMundoY();
    g2.drawRect(jugadorX, jugadorY, tamanioTile, tamanioTile);

    // Enemigos
```

```
            for (EnemigoModel enemigo : engine.getEnemigoSystem().getEnemigos()) {
                int enemX = enemigo.getTransform().getX() - camara.getMundoX();
                int enemY = enemigo.getTransform().getY() - camara.getMundoY();
                g2.drawRect(enemX, enemY, tamanioTile, tamanioTile);
            }
        }
    }
```

6. MEJORES PRACTICAS Y RECOMENDACIONES

6.1. Principios de Diseno Aplicados

1. Separacion de Responsabilidades: Cada renderer tiene una unica tarea
2. Delegacion: RenderSystem delega a renderers especializados
3. Optimizacion Temprana: Frustum culling desde el inicio
4. Cache Inteligente: Imagenes cargadas una sola vez
5. Escalabilidad: Facil agregar nuevos elementos visuales

6.2. Anti-Patrones a Evitar

- NO cargar imagenes en el metodo render()
- NO hacer calculos complejos durante el renderizado
- NO crear nuevos objetos en cada frame
- NO olvidar llamar a dispose() en Graphics2D
- NO ignorar el frustum culling

6.3. Checklist de Rendimiento

- Double buffering activado
- Frustum culling implementado
- Sprites en cache
- Algoritmo del Pintor para Z-ordering
- Renderizado por capas
- Metricas de FPS monitoreadas
- Escalado proporcional implementado

SISTEMA DE COMUNICACION EN TIEMPO REAL

DESCRIPCION GENERAL

El sistema de comunicacion en tiempo real permite que multiples jugadores se conecten a un servidor compartido, compartan informacion de sus posiciones, estadisticas y estado del juego. Se implementa utilizando sockets TCP de Java para la transferencia de datos entre cliente y servidor.

ARQUITECTURA DE RED

La arquitectura se compone de tres elementos principales:

1. Cliente de Juego

   El cliente local que maneja la entrada del usuario y la logica del juego local.

2. Servidor Dedicado

   El servidor que actua como intermediario entre todos los clientes conectados.

3. Protocolo de Comunicacion Personalizado

   Un protocolo basado en texto que define como se intercambia informacion entre cliente y servidor.

CONFIGURACION DE RED

Puerto del Servidor: 8888

Protocolo: TCP

Tipo de Conexion: Cliente-Servidor

Concurrencia: Multi-cliente (soporta multiples conexiones simultaneas)

ESTABLECIMIENTO DE CONEXION

Cuando un cliente se conecta al servidor, ocurren los siguientes pasos:

1. Creacion del Socket

En GameClient.java:

```
private Socket socket;
private PrintWriter writer;
private BufferedReader reader;

public GameClient(String host, int puerto) {
    try {
        this.socket = new Socket(host, puerto);
        this.writer = new PrintWriter(socket.getOutputStream(), true);
        this.reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        Thread lecturaThread = new Thread(this::leerMensajes);
        lecturaThread.setDaemon(true);
        lecturaThread.start();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

2. Autenticacion del Cliente

Una vez establecida la conexion, el cliente envia su nombre de usuario:

```
public void enviarNombreUsuario(String nombre) {
    if (writer != null) {
        writer.println("USUARIO:" + nombre);
    }
}
```

3. Respuesta del Servidor

El servidor responde con informacion de bienvenida y configuracion del juego:

```
WELCOME:nombreUsuario,idJugador,posicionX,posicionY,mapSeed
```

PROTOCOLO DE COMUNICACION

El protocolo utiliza mensajes basados en texto con formato especifico. Cada mensaje comienza con un identificador de tipo seguido de dos puntos y los datos:

TIPOS DE MENSAJE

1. USUARIO - Autenticacion de Usuario

USUARIO:nombre_jugador

Ejemplo: USUARIO:Juan

2. POS - Posicion del Jugador

POS:x,y,direccion,nombreJugador

Ejemplo: POS:5000,5000,DOWN,Juan

Se envia en cada frame de actualizacion del juego para sincronizar posiciones de todos los jugadores.

3. VIDA - Puntos de Vida del Jugador

VIDA:cantidadVida,nombreJugador

Ejemplo: VIDA:100,Juan

Se envia cuando el jugador recibe dano o usa pociones.

4. ACERTIJOS - Cantidad de Acertijos Resueltos

ACERTIJOS:cantidad,nombreJugador

Ejemplo: ACERTIJOS:5,Juan

Se envia cuando el jugador resuelve un acertijo con exito.

5. COFRE_CERRADO - Estado de los Cofres

COFRE_CERRADO:chunkX,chunkY,cofresIds,nombreJugador

Ejemplo: COFRE_CERRADO:0,0,1_2_3,Juan

Se envia cuando un jugador abre un cofre para informar a otros jugadores que este cofre ya no contiene items.

6. MAP_SEED - Semilla del Mapa

MAP_SEED:seed

Ejemplo: MAP_SEED:12345

Se envia cuando un jugador crea un servidor para que todos los clientes generen el mismo mapa.

7. START_GAME - Inicio del Juego

START_GAME

Indica que el juego debe comenzar (usado para sincronizar inicio en multijugador).

8. SERVIDOR_CERRADO - Cierre del Servidor

SERVIDOR_CERRADO

Notifica a los clientes que el servidor ha sido cerrado.

FLUJO DE COMUNICACION

Conexion Inicial

Cliente Juego

|

| conectar a 8888

|

Servidor TCP

|

| socket aceptado

|

Cliente Juego

|

| enviar USUARIO:nombre

|

Servidor TCP

|

| enviar WELCOME:nombre,id,x,y,seed

|

Cliente Juego

|

v registrado y listo

Sincronizacion Durante el Juego

Cada Frame (aproximadamente 60 veces por segundo):

Cliente Juego 1

|

| enviar POS

| enviar VIDA

| enviar ACERTIJOS

|

Servidor TCP

|

| broadcast a todos

|

Cliente Juego 2

Cliente Juego 3

ESTRUCTURA DEL SERVIDOR

En Main.java se inicia el servidor:

```
public class GameServer {
    private ServerSocket serverSocket;
    private java.util.Map<String, ClientHandler> clientes;
    private long mapSeed;

    public GameServer(int puerto) {
        try {
            this.serverSocket = new ServerSocket(puerto);
            this.clientes = new ConcurrentHashMap();
            this.mapSeed = System.currentTimeMillis();
            System.out.println("Servidor iniciado en puerto: " + puerto);
            aceptarConexiones();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void aceptarConexiones() {
        new Thread(() -> {
            while (true) {
                try {
                    Socket clientSocket = serverSocket.accept();
                    ClientHandler clientHandler = new ClientHandler(
                        clientSocket,
                        clientes,
                        mapSeed
```

```
                );
                new Thread(clientHandler).start();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }).start();
  }
}
```

ESTRUCTURA DEL MANEJADOR DE CLIENTE

Cada cliente conectado es manejado por un ClientHandler en un thread separado:

```
public class ClientHandler implements Runnable {
    private Socket socket;
    private String nombreJugador;
    private Map<String, ClientHandler> todosLosClientes;
    private long mapSeed;

    public void run() {
        try {
            BufferedReader reader = new BufferedReader(
                new InputStreamReader(socket.getInputStream())
            );
            PrintWriter writer = new PrintWriter(
                socket.getOutputStream(),
                true
            );

            String mensaje;
            while ((mensaje = reader.readLine()) != null) {
                procesarMensaje(mensaje, writer);
                transmitirATodos(mensaje);
            }
        } catch (IOException e) {
            System.out.println(nombreJugador + " desconectado");
            todosLosClientes.remove(nombreJugador);
        }
    }
```

```
    private void procesarMensaje(String mensaje, PrintWriter writer) {
        if (mensaje.startsWith("USUARIO:")) {
            nombreJugador = mensaje.substring(8);
            todosLosClientes.put(nombreJugador, this);
            writer.println("WELCOME:" + nombreJugador + ",0,5000,5000," + mapSeed);
        } else if (mensaje.startsWith("MAP_SEED:")) {
            transmitirATodos(mensaje);
        }
    }

    private void transmitirATodos(String mensaje) {
        for (ClientHandler cliente : todosLosClientes.values()) {
            cliente.enviarMensaje(mensaje);
        }
    }

    public void enviarMensaje(String mensaje) {
        if (writer != null) {
            writer.println(mensaje);
        }
    }
}
```

SINCRONIZACION EN TIEMPO REAL

Posiciones de Jugadores

En GamePanel.java, cada frame se envia la posicion:

```
if (gameClient != null) {
    gameClient.sendPosition(
        jugadorSystem.getJugador().getMundoX(),
        jugadorSystem.getJugador().getMundoY(),
        jugadorSystem.getJugador().getDireccion().toString(),
        nombreJugador
    );
}
```

En el servidor, este mensaje se retransmite a todos los clientes:

```
POS:5000,5050,DOWN,Juan
```

En el cliente receptor, se actualiza la posicion del jugador remoto:

```
if (mensaje.startsWith("POS:")) {
    String[] partes = mensaje.substring(4).split(",");
    int x = Integer.parseInt(partes[0]);
    int y = Integer.parseInt(partes[1]);
    String direccion = partes[2];
    String nombre = partes[3];

    RemotePlayer remoto = remotePlayers.get(nombre);
    if (remoto != null) {
```

```
            remoto.actualizarPosicion(x, y, direccion);
        }
    }
```

Sincronizacion de Semilla de Mapa

Para que todos los jugadores tengan el mismo mapa:

1. El jugador que crea el servidor obtiene una semilla

En GameServer:

this.mapSeed = System.currentTimeMillis();

2. Se la envia al primer cliente conectado

En WELCOME:

WELCOME:nombre,id,5000,5000,123456789

3. El cliente genera el mapa con esa semilla

En GameEngine:

GeneradorMundo generador = new GeneradorMundo(mapSeed);

mundo = generador.generarMundo();

4. Los clientes que se unen reciben la misma semilla

Al conectarse:

WELCOME:nombre,id,5000,5000,123456789

Resultado: Todos ven exactamente el mismo mapa.

MANEJO DE DESCONEXION

Cuando un Cliente se Desconecta

En ClientHandler.run():

```
} catch (IOException e) {
    System.out.println(nombreJugador + " desconectado");
    todosLosClientes.remove(nombreJugador);
}
```

El servidor automáticamente:

. Elimina al jugador de la lista de clientes

. Notifica a otros clientes (esto podria mejorarse)

Cuando el Servidor se Cierra

Si el servidor se cierra, los clientes reciben:

```
SERVIDOR_CERRADO
```

En GameClient:

```
if (mensaje.equals("SERVIDOR_CERRADO")) {
    System.out.println("El servidor se ha cerrado");
    estadoJuego = GameState.MENU_PRINCIPAL;
    conexionActiva = false;
}
```

COMUNICACION MULTIJUGADOR

Cuando el Creador de Sala Termina el Juego

Si el jugador que creo la sala (servidor) es quien termina:

1. El juego de ese jugador termina

En GamePanel, tiempo llega a 0:

estadoJuego = GameState.JUEGO_TERMINADO;

2. Los otros clientes siguen jugando

No hay notificacion de termino de juego en otros clientes.

Cuando Otro Jugador Termina

Si un cliente que se unio a la sala termina:

1. Su juego termina localmente
2. No afecta a otros jugadores

El servidor NO cierra, por lo que otros pueden continuar jugando.

ESTRUCTURA DE DATOS DE JUGADORES REMOTOS

Cada cliente mantiene una referencia a los otros jugadores en remotePlayers:

```
private Map<String, RemotePlayer> remotePlayers = new ConcurrentHashMap();
```

La clase RemotePlayer contiene:

```
public class RemotePlayer {
    private String nombre;
    private EntidadModel entidad;
    private PlayerStats stats;
    private int acertijosResueltos;

    public RemotePlayer(String nombre) {
        this.nombre = nombre;
        this.entidad = new EntidadModel(5000, 5000);
        this.stats = new PlayerStats(nombre);
        this.acertijosResueltos = 0;
    }

    public void actualizarPosicion(int x, int y, String direccion) {
        entidad.setMundoX(x);
        entidad.setMundoY(y);
    }

    public void actualizarVida(int vida) {
        stats.setVida(vida);
    }
```

```
    public void actualizarAcertijos(int cantidad) {
        stats.setAcertijos(cantidad);
    }
}
```

SINCRONIZACION DEL MAPA PARA MULTIJUGADOR

DESCRIPCION GENERAL

En un juego multijugador, es critico que todos los jugadores vean exactamente el mismo mundo. Para lograr esto en VideoJuego_v2 se utiliza un sistema de semillas (seeds) que garantiza que el mapa generado sea identico en todas las instancias del juego.

PROBLEMA A RESOLVER

Sin sincronizacion de mapa:

Jugador 1 Jugador 2

Mapa generado aleatoriamente Mapa generado aleatoriamente

Enemigos en posicion A Enemigos en posicion B

Cofres en celda 1 Cofres en celda 2

Resultado: CONFLICTO Resultado: CONFLICTO

Ambos ven un mundo diferente:

. Los enemigos no estan en los mismos lugares

. Los recursos (pociones, cofres) no coinciden

. La aventura no es compartida

SOLUCION IMPLEMENTADA

La solucion es usar una semilla unica de generacion de numeros pseudoaleatorios. Si ambos jugadores generan el mapa usando la misma semilla, obtendran identicamente el mismo mapa.

Jugador 1	Jugador 2
Recibe seed: 123456	Recibe seed: 123456
Genera mapa con seed	Genera mapa con seed
Resultado: MAPA A	Resultado: MAPA A
SINCRONIZADO	SINCRONIZADO

GENERACION DEL MAPA CON SEED

En GeneradorMundo.java:

```
public class GeneradorMundo {
    private long seed;
    private Random random;
    private static final int TAMANIO_MUNDO = 10000;

    public GeneradorMundo(long seed) {
        this.seed = seed;
        this.random = new Random(seed);
    }

    public Chunk generarChunk(int chunkX, int chunkY) {
        random.setSeed(seed + chunkX + chunkY * 1000);
        Tile[][] tiles = new Tile[TAMANIO_CHUNK][TAMANIO_CHUNK];

        for (int x = 0; x < TAMANIO_CHUNK; x++) {
            for (int y = 0; y < TAMANIO_CHUNK; y++) {
                int tipoTile = random.nextInt(100);
                tiles[x][y] = crearTile(tipoTile);
            }
        }

        return new Chunk(tiles);
    }
}
```

Explicacion:

1. Se recibe una semilla unica (ej: 123456)
2. Se crea un objeto Random con esa semilla
3. Cada chunk se genera usando la semilla mas sus coordenadas
4. random.setSeed(seed + chunkX + chunkY * 1000) asegura que:
   . El chunk (0,0) siempre genere lo mismo
   . El chunk (1,0) siempre genere lo mismo
   . El chunk (0,1) siempre genere diferente a (1,0)

FLUJO DE SINCRONIZACION

Paso 1: Creacion del Servidor

Cuando un jugador selecciona Crear Servidor:

En Main.java:

```
GameServer server = new GameServer(8888);
long mapSeed = System.currentTimeMillis();
server.setMapSeed(mapSeed);
```

Se genera una semilla usando la marca de tiempo actual (unica en ese momento).

Paso 2: Generacion del Mapa Local

El jugador que creo el servidor genera su mapa:

En GamePanel:

```
if (tipoJuego == SERVIDOR) {
    long mapSeed = System.currentTimeMillis();
    GeneradorMundo generador = new GeneradorMundo(mapSeed);
    world = generador.generarMundo();
}
```

Paso 3: Envio de Semilla al Cliente

Cuando un cliente se conecta, recibe la semilla:

En ClientHandler (servidor):

```
public void procesarMensaje(String mensaje) {
    if (mensaje.startsWith("USUARIO:")) {
        nombreJugador = mensaje.substring(8);
        String welcome = "WELCOME:" + nombreJugador +
                         ",id,5000,5000," + mapSeed;
        writer.println(welcome);
    }
}
```

Formato: WELCOME:nombre,id,posX,posY,mapSeed

Ejemplo: WELCOME:Juan,0,5000,5000,1732626000000

Paso 4: Recepcion de Semilla en Cliente

En GameClient:

```
private void leerMensajes() {
    String mensaje;
    while ((mensaje = reader.readLine()) != null) {
        if (mensaje.startsWith("WELCOME:")) {
            procesarWelcome(mensaje);
        }
    }
}
```

```
private void procesarWelcome(String mensaje) {
    String[] partes = mensaje.substring(8).split(",");
    String nombreJugador = partes[0];
    int posX = Integer.parseInt(partes[2]);
    int posY = Integer.parseInt(partes[3]);
    long mapSeed = Long.parseLong(partes[4]);

    generarMapoConSeed(mapSeed);
}
```

Paso 5: Generacion del Mapa en Cliente

El cliente recibe la semilla y genera el mismo mapa:

```
private void generarMapoConSeed(long seed) {
    GeneradorMundo generador = new GeneradorMundo(seed);
    mundo = generador.generarMundo();
}
```

Ahora ambos clientes tienen el mapa identico.

GENERACION DE MUNDO INFINITO CON SEED

El mundo se genera usando chunks. Cada chunk tiene coordenadas (chunkX, chunkY):

En ManejadorMapaInfinito.java:

```
public class ManejadorMapaInfinito {
    private Map<String, Chunk> chunksEnMemoria;
    private GeneradorMundo generador;

    public Chunk obtenerChunk(int chunkX, int chunkY) {
        String clave = chunkX + "," + chunkY;

        if (!chunksEnMemoria.containsKey(clave)) {
            Chunk chunk = generador.generarChunk(chunkX, chunkY);
            chunksEnMemoria.put(clave, chunk);
        }

        return chunksEnMemoria.get(clave);
    }
}
```

Esto garantiza que:

. Cada chunk con coordenadas (x,y) se genera igual en todos los clientes

. Si el jugador regresa a (10,10), veran el MISMO chunk

. Los enemigos, objetos y tiles coincidiran exactamente

EJEMPLO PRACTICO

Generacion del Mapa de dos Jugadores

Servidor:

mapSeed = 1732626000000

Genera chunk (0,0):

random.setSeed(1732626000000 + 0 + 0 * 1000)

Resultado: Pasto, arbol, enemigo goblin

Genera chunk (1,0):

random.setSeed(1732626000000 + 1 + 0 * 1000)

Resultado: Agua, puente, cofre

Cliente 1 (Servidor):

Recibe WELCOME con seed 1732626000000

Genera chunk (0,0):

random.setSeed(1732626000000 + 0 + 0 * 1000)

Resultado: Pasto, arbol, enemigo goblin ✓ COINCIDE

Genera chunk (1,0):

random.setSeed(1732626000000 + 1 + 0 * 1000)

Resultado: Agua, puente, cofre ✓ COINCIDE

Cliente 2 (Se une):

Recibe WELCOME con seed 1732626000000

Genera chunk (0,0):

random.setSeed(1732626000000 + 0 + 0 * 1000)

Resultado: Pasto, arbol, enemigo goblin ✓ COINCIDE

Genera chunk (1,0):

random.setSeed(1732626000000 + 1 + 0 * 1000)

Resultado: Agua, puente, cofre ✓ COINCIDE

Todos ven lo MISMO.

VENTAJAS DE USAR SEED

1. Sincronizacion Perfecta

No es necesario enviar todo el mapa por la red. Solo se envia un numero (la semilla).

Alternativa sin seed (MALA):

- Generar todo el mundo (millones de tiles)
- Serializar todo el mapa
- Enviar por red (gigabytes de datos)
- Deserializar en cliente

Con seed (BUENA):

- Generar semilla aleatoria (8 bytes)
- Enviar semilla por red
- Ambos generan el mapa localmente

2. Consistencia

Si un jugador regresa a una zona visitada antes:

- Los enemigos estaran en las mismas posiciones
- Los tiles seran identicos
- Los cofres estaran en los mismos lugares

3. Escalabilidad

Con cientos o miles de jugadores:

- Todos generan el mismo mapa sin comunicacion adicional

- No consume ancho de banda

CODIGO REFERENCIA COMPLETO

En GamePanel.java, conexion multijugador:

```
if (tipoJuego == SERVIDOR) {
    long mapSeed = System.currentTimeMillis();
    this.gameClient = new GameClient(
        "localhost",
        8888,
        nombreJugador,
        mapSeed
    );
    gameEngine = new GameEngine(mapSeed);
} else if (tipoJuego == CLIENTE) {
    this.gameClient = new GameClient(host, puerto, nombreJugador, null);
    this.gameClient.setOnMapSeedReceived((seed) -> {
        gameEngine = new GameEngine(seed);
    });
}
```

En GameClient.java:

```
public void procesarWelcome(String mensaje) {
    String[] partes = mensaje.substring(8).split(",");
```

```
    long mapSeed = Long.parseLong(partes[4]);

    if (onMapSeedReceived != null) {
        onMapSeedReceived.accept(mapSeed);
    }
}
```

En GameEngine.java:

```
public GameEngine(long mapSeed) {
    this.mapaInfinito = new ManejadorMapaInfinito(mapSeed);
    this.jugadorSystem = new JugadorSystem();
    this.enemigoSystem = new EnemigoSystem(mapaInfinito);
}
```

DETERMINISMO EN GENERACION ALEATORIA

Es importante notar que la generacion pseudoaleatoria es determinista:

Semilla = 12345

Random.setSeed(12345)

random.nextInt() = 678

random.nextInt() = 923

random.nextInt() = 145

Semilla = 12345

Random.setSeed(12345)

random.nextInt() = 678 ← IDENTICO

random.nextInt() = 923 ← IDENTICO

random.nextInt() = 145 ← IDENTICO

Esto garantiza determinismo total en la generacion de mapas.

SINCRONIZACION DINAMICA

Aunque el mapa inicial se sincroniza por seed, los cambios dinamicos se sincronizan separadamente:

1. Mapa Base

Sincronizado por seed. Es igual para todos.

2. Cofres Abiertos

Se sincronizan via COFRE_CERRADO:

COFRE_CERRADO:0,0,5_12_23

Comunica a otros: En chunk (0,0) los cofres con ID 5, 12 y 23 estan abiertos.

3. Enemigos Muertos

Actualmente no se sincronizan. Cada cliente mantiene su propio estado de enemigos.

Mejora futura: Enviar ENEMIGO_MUERTO:enemyId para sincronizar enemigos eliminados.

SINCRONIZACION DE MOVIMIENTO Y DATOS DEL JUGADOR

DESCRIPCION GENERAL

En multijugador es esencial que todos los jugadores vean los movimientos y cambios de estado de todos los demas. Este documento explica como se sincronizan tres aspectos criticos: movimientos de jugadores, estadisticas de vida y acertijos resueltos.

COMPONENTES PRINCIPALES

1. Protocolo de Mensajes POS

Transmite posicion y direccion del jugador cada frame.

2. Protocolo de Mensajes VIDA

Transmite la cantidad de vida cuando cambia.

3. Protocolo de Mensajes ACERTIJOS

Transmite la cantidad de acertijos resueltos.

4. Sistema de Cacheo

Evita enviar datos que no han cambiado.

SINCRONIZACION DE MOVIMIENTO

Protocolo POS

Formato: POS:x,y,direccion,nombreJugador

Ejemplo: POS:5050,5100,DOWN,Juan

Componentes:

. x, y: Posicion mundial del jugador

. direccion: DOWN, UP, LEFT, RIGHT (direccion hacia la que mira)

. nombreJugador: Identificador unico del jugador

Envio del Movimiento

En GamePanel.java, cada frame durante el juego:

```
private void updateMultiplayer() {
    if (gameClient != null && gameClient.isConectado()) {
        int x = jugadorSystem.getJugador().getMundoX();
        int y = jugadorSystem.getJugador().getMundoY();
        String dir = jugadorSystem.getJugador().getDireccion().toString();

        gameClient.sendPosition(x, y, dir, nombreJugador);
    }
}
```

Este metodo se llama en cada frame de actualizacion.

En GameClient.java:

```
public void sendPosition(int x, int y, String direccion, String nombreJugador) {
    if (writer != null) {
        String mensaje = "POS:" + x + "," + y + "," + direccion + "," + nombreJugador;
```

```
        writer.println(mensaje);
    }
}
```

Retransmision del Servidor

En ClientHandler.java, cuando se recibe un mensaje POS:

```
private void procesarMensaje(String mensaje) {
    if (mensaje.startsWith("POS:")) {
        transmitirATodos(mensaje);
    }
}

private void transmitirATodos(String mensaje) {
    for (ClientHandler cliente : todosLosClientes.values()) {
        cliente.enviarMensaje(mensaje);
    }
}
```

El servidor retransmite a TODOS los clientes conectados (incluido el que envio).

Recepcion en Clientes

En GameClient.java, en el thread de lectura:

```
private void leerMensajes() {
    String linea;
    while ((linea = reader.readLine()) != null) {
        if (linea.startsWith("POS:")) {
            String[] partes = linea.substring(4).split(",");
            int x = Integer.parseInt(partes[0]);
            int y = Integer.parseInt(partes[1]);
            String direccion = partes[2];
            String nombre = partes[3];

            if (!nombre.equals(nombreLocal)) {
                RemotePlayer remoto = remotePlayers.get(nombre);
                if (remoto == null) {
                    remoto = new RemotePlayer(nombre);
                    remotePlayers.put(nombre, remoto);
                }
                remoto.actualizarPosicion(x, y, direccion);
            }
        }
    }
}
```

Lo importante:

. Se ignora el propio mensaje (if !nombre.equals(nombreLocal))

. Se obtiene o crea el RemotePlayer si no existe

. Se actualiza su posicion

Renderizado de Jugadores Remotos

En RenderSystem.java, durante el renderizado:

```
for (RemotePlayer remote : remotePlayers.values()) {
    EntidadModel entidad = remote.getEntidad();
    int screenX = entidad.getMundoX() - camara.getCamaraX();
    int screenY = entidad.getMundoY() - camara.getCamaraY();

    if (screenX > -tamanioTile && screenX < pantallaAncho &&
        screenY > -tamanioTile && screenY < pantallaAlto) {
        entidadRenderer.render(g2, entidad, screenX, screenY);
    }
}
```

Resultado final: Ves otros jugadores moverse en tiempo real.

SINCRONIZACION DE VIDA

Protocolo VIDA

Formato: VIDA:cantidad,nombreJugador

Ejemplo: VIDA:85,Maria

Componentes:

. cantidad: Puntos de vida actuales (0-100)

. nombreJugador: Jugador cuya vida cambio

Cuando se Envia

En GameEngine.java, en el metodo update():

```
if (vidaActual != lastVidaSent) {
    gameClient.sendVida(vidaActual, nombreJugador);
    lastVidaSent = vidaActual;
}
```

Se envia SOLO cuando cambia. No se envia cada frame si no cambio.

En GameClient.java:

```
public void sendVida(int vida, String nombreJugador) {
    if (writer != null) {
        String mensaje = "VIDA:" + vida + "," + nombreJugador;
        writer.println(mensaje);
    }
}
```

Retransmision en Servidor

En ClientHandler.java:

```
if (mensaje.startsWith("VIDA:")) {
    transmitirATodos(mensaje);
}
```

Se retransmite a todos sin modificacion.

Recepcion en Clientes

En GameClient.java:

```
if (linea.startsWith("VIDA:")) {
    String[] partes = linea.substring(5).split(",");
    int vida = Integer.parseInt(partes[0]);
    String nombre = partes[1];

    if (nombre.equals(nombreLocal)) {
        statsLocal.setVida(vida);
    } else {
        RemotePlayer remoto = remotePlayers.get(nombre);
        if (remoto != null) {
            remoto.getStats().setVida(vida);
        }
    }
}
```

Logica:

. Si es el propio jugador, actualiza estadisticas locales

. Si es otro jugador, actualiza su RemotePlayer

Almacenamiento de Estadisticas

En PlayerStats.java:

```
public class PlayerStats {
    private String nombre;
    private int vida;
    private int vidaMaxima;
    private int acertijos;
    private int pociones;

    public void setVida(int vida) {
        if (vida >= 0 && vida <= vidaMaxima) {
            this.vida = vida;
        }
    }

    public int getVida() {
        return vida;
    }
}
```

Cada jugador remoto tiene su propia instancia de PlayerStats.

Renderizado de Vida en HUD

En HUDRenderer.java:

```
public void renderHUDVida(Graphics2D g2, PlayerStats stats) {
    String texto = "VIDA: " + stats.getVida() + "/" + stats.getVidaMaxima();
    g2.drawString(texto, 10, 30);
}
```

En MultiplayerHUDRenderer.java, se renderizan las vidas de todos:

```
public void renderJugadoresRemotos(Graphics2D g2,
    Map<String, RemotePlayer> remotePlayers) {
    int y = 100;
    for (RemotePlayer remoto : remotePlayers.values()) {
        String nombre = remoto.getNombre();
        int vida = remoto.getStats().getVida();
        String texto = nombre + ": " + vida + " HP";
        g2.drawString(texto, 10, y);
        y += 25;
    }
}
```

Resultado: Ves la vida de todos en tiempo real.

SINCRONIZACION DE ACERTIJOS

Protocolo ACERTIJOS

Formato: ACERTIJOS:cantidad,nombreJugador

Ejemplo: ACERTIJOS:7,Pedro

Componentes:

. cantidad: Cantidad total de acertijos resueltos

. nombreJugador: Jugador que resolvio el acertijo

Cuando se Envia

En GameEngine.java, en update():

```
int acertijosActuales = statsLocal.getAcertijos();
if (acertijosActuales != lastAcertijosCount) {
    gameClient.sendAcertijos(acertijosActuales, nombreJugador);
    lastAcertijosCount = acertijosActuales;
}
```

Se envia cuando cambia la cantidad.

En GameClient.java:

```
public void sendAcertijos(int cantidad, String nombreJugador) {
    if (writer != null) {
        String mensaje = "ACERTIJOS:" + cantidad + "," + nombreJugador;
        writer.println(mensaje);
    }
}
```

Retransmision en Servidor

En ClientHandler.java:

```
if (mensaje.startsWith("ACERTIJOS:")) {
    transmitirATodos(mensaje);
}
```

Recepcion en Clientes

En GameClient.java:

```
if (linea.startsWith("ACERTIJOS:")) {
    String[] partes = linea.substring(10).split(",");
    int cantidad = Integer.parseInt(partes[0]);
    String nombre = partes[1];

    if (nombre.equals(nombreLocal)) {
        statsLocal.setAcertijos(cantidad);
    } else {
        RemotePlayer remoto = remotePlayers.get(nombre);
        if (remoto != null) {
            remoto.getStats().setAcertijos(cantidad);
        }
    }
}
```

Almacenamiento

En PlayerStats.java:

```
private int acertijos;

public void setAcertijos(int cantidad) {
    this.acertijos = cantidad;
}

public int getAcertijos() {
    return acertijos;
}
```

Renderizado en HUD

En MultiplayerHUDRenderer.java:

```
public void renderRanking(Graphics2D g2,
    PlayerStats statsLocal,
    Map<String, RemotePlayer> remotePlayers) {

    g2.drawString("RANKING DE ACERTIJOS", 10, 60);

    g2.drawString(statsLocal.getNombre() + ": " +
        statsLocal.getAcertijos(), 10, 90);

    int y = 120;
    for (RemotePlayer remoto : remotePlayers.values()) {
        g2.drawString(remoto.getNombre() + ": " +
            remoto.getStats().getAcertijos(), 10, y);
        y += 25;
    }
}
```

ESTRUCTURA DE REMOTEPLAYERCLASS

La clase RemotePlayer mantiene toda la informacion de un jugador remoto:

```
public class RemotePlayer {
    private String nombre;
    private EntidadModel entidad;
    private PlayerStats stats;

    public RemotePlayer(String nombre) {
        this.nombre = nombre;
        this.entidad = new EntidadModel(5000, 5000);
        this.stats = new PlayerStats(nombre);
    }

    public void actualizarPosicion(int x, int y, String direccion) {
        entidad.setMundoX(x);
        entidad.setMundoY(y);
        entidad.setDireccion(Direccion.valueOf(direccion));
    }

    public String getNombre() {
        return nombre;
    }

    public EntidadModel getEntidad() {
        return entidad;
    }

    public PlayerStats getStats() {
```

```
        return stats;
    }
}
```

INTEGRACION EN GAMEENGINE

En GameEngine.java se coordina toda la sincronizacion:

```
public void updateMultiplayer() {
    if (gameClient == null || !gameClient.isConectado()) {
        return;
    }

    PlayerStats statsLocal = this.statsLocal;
    int vidaActual = statsLocal.getVida();

    if (vidaActual != lastVidaSent) {
        gameClient.sendVida(vidaActual, nombreJugador);
        lastVidaSent = vidaActual;
    }

    int acertijosActuales = statsLocal.getAcertijos();
    if (acertijosActuales != lastAcertijosCount) {
        gameClient.sendAcertijos(acertijosActuales, nombreJugador);
        lastAcertijosCount = acertijosActuales;
    }
}
```

Campos para cacheo:

```
private int lastVidaSent = -1;
private int lastAcertijosCount = -1;
```

FLUJO COMPLETO DE SINCRONIZACION

Escenario: Jugador A se mueve y resuelve un acertijo mientras Jugador B mira

Frame 1:

Jugador A se mueve (5000,5000) -> (5050,5000)

Jugador A envia: POS:5050,5000,RIGHT,Juan

Servidor recibe, retransmite a B

Jugador B recibe: POS:5050,5000,RIGHT,Juan

B actualiza remoto y lo ve moverse

Frame 2:

Jugador A resuelve acertijo (3 -> 4)

Jugador A envia: ACERTIJOS:4,Juan

Servidor recibe, retransmite a B

Jugador B recibe: ACERTIJOS:4,Juan

B ve el contador de acertijos aumentar en HUD

Frame 3:

Jugador A se mueve (5050,5000) -> (5100,5000) y su vida baja (100 -> 85)

Jugador A envia: POS:5100,5000,RIGHT,Juan

Jugador A envia: VIDA:85,Juan

Servidor recibe ambos, retransmite

Jugador B recibe ambos

B ve jugador moverse Y su vida bajar

OPTIMIZACIONES IMPLEMENTADAS

1. Cacheo de Datos

Se envia VIDA y ACERTIJOS solo cuando cambian:

```
if (vidaActual != lastVidaSent) {
    gameClient.sendVida(vidaActual, nombreJugador);
    lastVidaSent = vidaActual;
}
```

No se envian cada frame si no hay cambio.

2. POS cada Frame

La posicion se envia cada frame porque cambia constantemente:

```
gameClient.sendPosition(x, y, dir, nombreJugador);
```

3. Thread Separado para Lectura

No bloquea el thread de renderizado:

```
Thread lecturaThread = new Thread(this::leerMensajes);
lecturaThread.setDaemon(true);
lecturaThread.start();
```

4. ConcurrentHashMap

Permite acceso thread-safe desde multiples threads:

```
private Map<String, RemotePlayer> remotePlayers = new ConcurrentHashMap();
```

TRATAMIENTO DE DATOS PERDIDOS

Si un mensaje se pierde en la red (poco probable con TCP):

El siguiente cambio en ese valor se envia y se recibe correctamente.

Ejemplo: Si se pierde "VIDA:85", el siguiente "VIDA:80" se recibira bien.

CONSIDERACIONES DE LATENCIA

Latencia de Red

Si la latencia es de 50ms:

. Posicion se envia cada 16ms (60 FPS)

. Llega después de 50ms

. Se ve con ~3 frames de delay

Interpolacion (Mejora Futura)

Podria implementarse interpolacion:

```
posicion_renderizada = posicion_anterior +
    (posicion_nueva - posicion_anterior) * (tiempo_transcurrido / tiempo_esperado)
```

Esto suavizaria el movimiento.

PANTALLA FINAL Y DETERMINACION DE GANADOR

DESCRIPCION GENERAL

Cuando el tiempo de juego llega a cero (o se alcanza el limite), el juego muestra una pantalla final que determina el ganador basandose en criterios especificos. En multijugador, todos los jugadores ven el mismo resultado simultaneamente.

TRANSICION A PANTALLA FINAL

Cuando Termina el Tiempo

En GamePanel.java, durante la actualizacion del juego:

```
if (tiempoRestanteSegundos == 0 && estadoJuego == GameState.JUGANDO) {
    estadoJuego = GameState.JUEGO_TERMINADO;
}
```

El estado cambia de JUGANDO a JUEGO_TERMINADO.

En GameEngine.java, el timer se controla:

```
public void update() {
    if (tiempoRestanteSegundos > 0) {
        tiempoRestanteSegundos--;
    }
    if (tiempoRestanteSegundos == 0) {
        juegoTerminado = true;
    }
}
```

CRITERIOS DE GANADOR

El ganador se determina por:

1. Criterio Primario: Cantidad de Acertijos Resueltos

El jugador con MAS acertijos resueltos GANA.

Si todos resolvieron 5 acertijos cada uno (empate), se aplica:

2. Criterio Secundario: Cantidad de Vida Restante

Si empatan en acertijos, el jugador con MAS vida GANA.

ESTRUCTURA DE DATOS DE GANADOR

En HUDRenderer.java:

```
private static class Ganador {
    String nombre;
    int acertijos;
    int vida;

    Ganador(String nombre, int acertijos, int vida) {
        this.nombre = nombre;
        this.acertijos = acertijos;
        this.vida = vida;
    }
}
```

LOGICA DE DETERMINACION DEL GANADOR

En HUDRenderer.java, metodo renderJuegoTerminado():

```
public void renderJuegoTerminado(Graphics2D g2,
    int pantallaAncho,
    int pantallaAlto,
    JugadorSystem jugadorSystem,
    PlayerStats statsLocal,
    Map<String, RemotePlayer> remotePlayers) {

    Ganador ganador = null;
    int maxAcertijos = statsLocal.getAcertijos();
    int maxVida = statsLocal.getVida();
    ganador = new Ganador(
        statsLocal.getNombre(),
        maxAcertijos,
        maxVida
    );

    for (RemotePlayer remoto : remotePlayers.values()) {
        int acertijosRemoto = remoto.getStats().getAcertijos();
        int vidaRemoto = remoto.getStats().getVida();

        if (acertijosRemoto > maxAcertijos) {
            maxAcertijos = acertijosRemoto;
            ganador = new Ganador(
                remoto.getNombre(),
                acertijosRemoto,
                vidaRemoto
```

```
            );
        } else if (acertijosRemoto == maxAcertijos &&
                vidaRemoto > ganador.vida) {
            ganador = new Ganador(
                remoto.getNombre(),
                acertijosRemoto,
                vidaRemoto
            );
        }
    }

    renderizarResultados(g2, pantallaAncho, pantallaAlto, ganador);
}
```

Explicacion del Algoritmo

1. Inicializa al jugador local como ganador temporal
2. Itera sobre todos los jugadores remotos
3. Si alguien tiene MAS acertijos, lo hace ganador
4. Si tiene IGUAL cantidad de acertijos pero MAS vida, lo hace ganador
5. Al final, renderiza los resultados

RENDERIZADO DE PANTALLA FINAL

En HUDRenderer.java, metodo renderizarResultados():

```
private void renderizarResultados(Graphics2D g2,
    int pantallaAncho,
    int pantallaAlto,
    Ganador ganador) {

    g2.setColor(new Color(0, 0, 0, 200));
    g2.fillRect(0, 0, pantallaAncho, pantallaAlto);

    g2.setFont(fuenteTitulo);
    g2.setColor(new Color(255, 215, 0));
    String titulo = "JUEGO TERMINADO";
    int anchoTitulo = g2.getFontMetrics().stringWidth(titulo);
    g2.drawString(titulo, (pantallaAncho - anchoTitulo) / 2, 80);

    g2.setFont(fuenteSubtitulo);
    g2.setColor(new Color(100, 255, 100));
    String textoGanador = "GANADOR: " + ganador.nombre;
    int anchoGanador = g2.getFontMetrics().stringWidth(textoGanador);
    g2.drawString(textoGanador, (pantallaAncho - anchoGanador) / 2, 150);

    g2.setFont(fuenteNormal);
    g2.setColor(Color.WHITE);

    String estadisticas = "Acertijos: " + ganador.acertijos +
                          " | Vida: " + ganador.vida;
    int anchoEstadisticas = g2.getFontMetrics().stringWidth(estadisticas);
```

```
    g2.drawString(estadisticas,
        (pantallaAncho - anchoEstadisticas) / 2, 220);

    g2.setColor(Color.YELLOW);
    String instruccion = "[ENTER] Jugar de Nuevo    [ESC] Menu Principal";
    int anchoInstruccion = g2.getFontMetrics().stringWidth(instruccion);
    g2.drawString(instruccion,
        (pantallaAncho - anchoInstruccion) / 2,
        pantallaAlto - 40);
}
```

ELEMENTOS DE LA PANTALLA

Fondo

Rectangulo semitransparente negro:

g2.setColor(new Color(0, 0, 0, 200));

g2.fillRect(0, 0, pantallaAncho, pantallaAlto);

Titulo

Texto dorado centrado:

Titulo: JUEGO TERMINADO

Texto Ganador

Texto verde brillante que muestra el nombre del ganador:

Texto: GANADOR: nombreJugador

Estadisticas del Ganador

Muestra acertijos y vida final:

Estadisticas: Acertijos: 7 | Vida: 65

Botones de Accion

Instruccion amarilla en la parte inferior:

Instruccion: [ENTER] Jugar de Nuevo [ESC] Menu Principal

EJEMPLO DE PANTALLA FINAL

Supongamos dos jugadores:

Jugador 1 (Local)

. Nombre: Juan

. Acertijos: 7

. Vida: 45

Jugador 2 (Remoto)

. Nombre: Maria

. Acertijos: 7

. Vida: 80

Algoritmo de Determinacion:

1. Inicializa ganador como Juan (7 acertijos, 45 vida)
2. Compara con Maria
3. Maria tiene 7 acertijos = igual que Juan
4. Maria tiene 80 vida > 45 vida (Juan)
5. Actualiza ganador a Maria
6. Resultado: GANADOR MARIA

Pantalla Mostrada:

JUEGO TERMINADO

GANADOR: Maria

Acertijos: 7 | Vida: 80

[ENTER] Jugar de Nuevo [ESC] Menu Principal

MANEJO DE ENTRADA EN PANTALLA FINAL

En GamePanel.java, en el metodo update():

```
case JUEGO_TERMINADO:
    if (inputService.isTeclaEnter()) {
        gameEngine.reiniciarJuego();
        estadoJuego = GameState.JUGANDO;
        inputService.setTeclaEnter(false);
    } else if (inputService.isTeclaEscape()) {
        gameEngine.reiniciarJuego();
        estadoJuego = GameState.MENU_PRINCIPAL;
        inputService.setTeclaEscape(false);
    }
    break;
```

Opciones:

. ENTER: Reinicia el juego y vuelve a JUGANDO

. ESC: Reinicia el juego y vuelve a MENU_PRINCIPAL

Importancia del reinicio:

Se debe llamar gameEngine.reiniciarJuego() para:

. Resetear temporizador a 3:00

. Limpiar enemigos

. Regenerar mapa

. Resetear vida a 100

. Resetear acertijos a 0

. Resetear pociones a 0

MULTIJUGADOR Y PANTALLA FINAL

En Juego Multijugador

Cuando termina el tiempo en multijugador:

1. Todos reciben la actualizacion del tiempo llegando a 0
2. Todos transicionan a JUEGO_TERMINADO simultaneamente
3. Cada cliente calcula quien es el ganador localmente
4. Se muestran los mismos resultados para todos (debido a que tienen el mismo estado)

En GamePanel.java, para multijugador:

```
if (gameClient != null && gameClient.isConectado()) {
    PlayerStats statsLocal = gameEngine.getStatsLocal();
    Map<String, RemotePlayer> remotePlayers =
        gameEngine.getRemotePlayers();

    renderSystem.renderJuegoTerminado(
        g2d,
        pantallaAncho,
        pantallaAlto,
        jugadorSystem,
        statsLocal,
        remotePlayers
    );
} else {
    renderSystem.renderJuegoTerminado(
        g2d,
        pantallaAncho,
```

```
        pantallaAlto,
        jugadorSystem,
        null,
        null
    );
}
```

En Juego Local (Un Solo Jugador)

Cuando es juego local, no hay otros jugadores:

Ganador: Siempre el jugador local (por defecto)

En HUDRenderer.java, manejo para juego local:

```
if (remotePlayers == null || remotePlayers.isEmpty()) {
    g2.drawString("Tiempo Terminado",
        (pantallaAncho - 100) / 2, 150);
    return;
}
```

TRANSICION DE ESTADO

Flujo Completo de Terminacion

Juego en Progreso

|

Tiempo = 0

|

estadoJuego = JUEGO_TERMINADO

|

Pantalla Final Mostrada

|

Jugador Presiona ENTER o ESC

|

gameEngine.reiniciarJuego()

|

estadoJuego = JUGANDO o MENU_PRINCIPAL

|

Nueva Sesion Comienza

RENDERIZADO EN SWITCH DE ESTADOS

En GamePanel.java, paintComponent():

```
switch (estadoJuego) {
    case MENU_PRINCIPAL:
        renderSystem.renderMenuPrincipal(g2d, pantallaAncho, pantallaAlto);
        break;
    case JUGANDO:
        renderTodo();
        break;
    case JUEGO_TERMINADO:
        renderSystem.renderJuegoTerminado(
            g2d,
            pantallaAncho,
            pantallaAlto,
            jugadorSystem,
            statsLocal,
            remotePlayers
        );
        break;
}
```

INFORMACION ADICIONAL EN PANTALLA FINAL

Podria extenderse para mostrar mas informacion:

Ranking Completo

En lugar de solo mostrar ganador, mostrar ranking:

1. Maria - 7 Acertijos, 80 Vida
2. Juan - 7 Acertijos, 45 Vida
3. Pedro - 5 Acertijos, 90 Vida

Estadisticas de Sesion

. Tiempo jugado

. Enemigos derrotados

. Pociones usadas

. Distancia recorrida

Esto podria agregarse en futuras versiones.

Diagrama de Clases UML del Proyecto

Este documento contiene el diagrama de clases UML en formato Mermaid.

Este diagrama visualiza la estructura estática del sistema, mostrando las clases principales, sus atributos, métodos y las relaciones entre ellas.

Diagrama

```mermaid
classDiagram
    direction LR

    class GamePanel {
        +GameEngine gameEngine
        +RenderSystem renderSystem
        +InputService inputService
        +GameState estadoJuego
        +paintComponent()
        +actualizar()
    }

    class GameEngine {
        +JugadorSystem jugadorSystem
        -EnemigoSystem enemigoSystem
        +ManejadorMapaInfinito mapaInfinito
        +GameClient gameClient
        +update()
        +reiniciarJuego()
    }
```

```
class EntidadModel {
    +Transform transform
    +int mundoX
    +int mundoY
}

class JugadorSystem {
    +EntidadModel jugador
    +update()
}

class EnemigoSystem {
    +List~EnemigoModel~ enemigos
    +update()
}

class EnemigoModel {
    +AIState state
}

class ManejadorMapaInfinito {
    -Map~String, Chunk~ chunksActivos
    -GeneradorMundo generador
    +actualizarChunksActivos()
    +regenerarConNuevoSeed()
}

class GeneradorMundo {
    -long seed
    +generarChunk()
```

```
}

class RenderSystem {
    +renderTodo()
    +renderJuegoTerminado()
}

class GameServer {
    -Map~String, ClientHandler~ clientes
    +aceptarConexiones()
}

class GameClient {
    -Socket socket
    -Map~String, RemotePlayer~ remotePlayers
    +sendPosition()
}

class ClientHandler {
    -Socket socket
    +run()
    +transmitirATodos()
}

EnemigoModel --|> EntidadModel

GamePanel o-- GameEngine
GamePanel o-- RenderSystem

GameEngine o-- JugadorSystem
```

```
    GameEngine o-- EnemigoSystem
    GameEngine o-- ManejadorMapaInfinito
    GameEngine o-- GameClient

    ManejadorMapaInfinito o-- GeneradorMundo
    EnemigoSystem "1" -- "*" EnemigoModel : contiene

    GameServer "1" -- "*" ClientHandler : gestiona
    GameClient "1" -- "1" GameServer : conecta con
```

PROCESO DE COMPILACIÓN Y GENERACIÓN DE EJECUTABLE

El proceso consta de dos fases principales:

1. Compilación y empaquetado JAR (compilar.bat)
2. Generación de ejecutable nativo (generar_exe.bat)

2. FASE 1: COMPILACIÓN Y CREACIÓN DEL JAR

2.1 Archivo: compilar.bat

Este script automatiza todo el proceso de compilación del proyecto Java y la creación del archivo JAR ejecutable.

2.2 Estructura del Proceso

El proceso de compilación se divide en 6 etapas:

ETAPA 1: Compilación de Archivos Java

ETAPA 2: Extracción de Dependencias

ETAPA 3: Copia de Recursos

ETAPA 4: Creación del Manifest

ETAPA 5: Generación del JAR

ETAPA 6: Prueba de Ejecución

2.3 Descripción Detallada de Cada Etapa

ETAPA 1: COMPILACIÓN DE ARCHIVOS JAVA

Objetivo: Compilar todos los archivos .java del proyecto

Pasos ejecutados:

1. Limpieza de compilaciones previas
   - Elimina el directorio bin/ si existe
   - Crea un nuevo directorio bin/ vacío

2. Generación de lista de archivos fuente
   - Ejecuta: dir /s /B *.java > sources.txt
   - Crea un archivo temporal con todas las rutas de archivos .java

3. Compilación
   - Ejecuta: javac -cp "lib/*" -d bin @sources.txt
   - Compila todos los archivos Java
   - Coloca los .class resultantes en bin/
   - Incluye las librerías de lib/ en el classpath

4. Verificación
   - Comprueba el código de salida
   - Si hay errores (errorlevel != 0), detiene el proceso

5. Limpieza
   - Elimina el archivo temporal sources.txt

ETAPA 2: EXTRACCIÓN DE DEPENDENCIAS

Objetivo: Incluir las librerías externas dentro del JAR final

Proceso:

1. Cambia al directorio bin/

2. Para cada archivo .jar en lib/:
   - Ejecuta: jar xf "nombre-libreria.jar"
   - Extrae todo el contenido de la librería
   - Los .class extraídos quedan disponibles en bin/

3. Regresa al directorio raíz

Librerías incluidas:

- org.json (para procesamiento de archivos JSON)
- Cualquier otra dependencia en la carpeta lib/

Razón: Al extraer las librerías y colocar sus .class en bin/, el JAR final será autónomo (no requiere librerías externas).

ETAPA 3: COPIA DE RECURSOS

Objetivo: Incluir todos los recursos (imágenes, datos) en el JAR

Comandos ejecutados:

```
xcopy /E /I /Y tiles bin\tiles
xcopy /E /I /Y spritesjugador bin\spritesjugador
xcopy /E /I /Y spritesenemigos bin\spritesenemigos
xcopy /E /I /Y data bin\data
```

Parámetros de xcopy:

/E - Copia subdirectorios, incluso vacíos

/I - Asume que el destino es un directorio

/Y - Suprime confirmación de sobrescritura

Estructura resultante en bin/:

```
bin/
  +-- Main/
  +-- domain/
  +-- tiles/
  |   +-- agua.png
  |   +-- pasto.png
  |   +-- ...
  +-- spritesjugador/
  +-- spritesenemigos/
  +-- data/
      +-- data.json
```

ETAPA 4: CREACIÓN DEL MANIFEST

Objetivo: Crear el archivo manifest que define la clase principal

Contenido del manifest.txt:

Main-Class: Main.Main

(línea vacía final requerida por especificación JAR)

La línea vacía final es obligatoria según la especificación de archivos JAR de Oracle.

.

ETAPA 5: GENERACIÓN DEL JAR

Objetivo: Empaquetar todo en un archivo JAR ejecutable

Comando:

```
jar cfm VideoJuego.jar manifest.txt -C bin .
```

Parámetros:

c - Crear nuevo archivo JAR

f - Especificar nombre del archivo (VideoJuego.jar)

m - Incluir información del manifest (manifest.txt)

-C bin . - Cambiar al directorio bin y agregar todo su contenido

Resultado: VideoJuego.jar conteniendo:

- Todas las clases compiladas
- Todas las dependencias (extraídas)
- Todos los recursos (tiles, sprites, data)
- Manifest con clase principal

ETAPA 6: PRUEBA DE EJECUCIÓN

Objetivo: Verificar que el JAR funciona correctamente

Comando:

```
java -jar VideoJuego.jar
```

Esta etapa lanza la aplicación para verificación manual antes de crear el ejecutable Windows.

3. FASE 2: GENERACIÓN DEL EJECUTABLE

3.1 Archivo: generar_exe.bat

Este script convierte el JAR en un ejecutable nativo de Windows usando la herramienta jpackage incluida en JDK 14+.

3.2 Proceso de Generación

PASO 1: Verificación de Prerequisitos

Verifica que existe VideoJuego.jar

Si no existe, muestra error y solicita ejecutar compilar.bat primero

PASO 2: Preparación de Estructura Temporal

1. Elimina carpeta temp-jpackage si existe
2. Crea nueva carpeta temp-jpackage
3. Copia VideoJuego.jar a temp-jpackage/

PASO 3: Limpieza de Salida Anterior

Elimina carpeta VideoJuego/ si existe de ejecuciones anteriores

PASO 4: Generación con jpackage

Comando:

```
jpackage ^
  --input temp-jpackage ^
  --name VideoJuego ^
  --main-jar VideoJuego.jar ^
  --main-class Main.Main ^
  --type app-image ^
  --dest .
```

Parámetros explicados:

--input temp-jpackage

Directorio que contiene el JAR y recursos

--name VideoJuego

Nombre de la aplicación

--main-jar VideoJuego.jar

Archivo JAR principal a ejecutar

--main-class Main.Main

Clase que contiene el método main()

--type app-image

Genera ejecutable portable (sin instalador)

--dest .

Directorio de salida (directorio actual)

PASO 5: Limpieza de Archivos Temporales

Elimina la carpeta temp-jpackage

PASO 6: Resultado Final

Estructura generada:

```
VideoJuego/
  +-- VideoJuego.exe         (Ejecutable principal)
  +-- app/
  |   +-- VideoJuego.jar     (JAR incluido)
  +-- runtime/
      +-- bin/
      +-- lib/
      +-- ...                (JRE embebido)
```

4. ESTRUCTURA DE ARCHIVOS COMPLETA

4.1 Antes de la Compilación

```
proyecto-objetos-videojuego/
  +-- Main/
  +-- domain/
  +-- infrastructure/
  +-- model/
  +-- Presentation/
  +-- lib/
  |   +-- org.json.jar
  +-- tiles/
  +-- spritesjugador/
  +-- spritesenemigos/
  +-- data/
  +-- compilar.bat
  +-- generar_exe.bat
  +-- ejecutar.bat
```

4.2 Después de la Compilación

```
proyecto-objetos-videojuego/
  +-- ... (archivos originales)
  +-- bin/                    (generado)
  |   +-- Main/
  |   +-- domain/
  |   +-- tiles/
  |   +-- ...
  +-- VideoJuego.jar          (generado)
  +-- manifest.txt            (generado)
```

4.3 Después de Generar el .exe

```
proyecto-objetos-videojuego/
  +-- ... (archivos anteriores)
  +-- VideoJuego/             (generado)
      +-- VideoJuego.exe
      +-- app/
      +-- runtime/
```

5. REQUISITOS DEL SISTEMA

5.1 Para Compilación

- Java Development Kit (JDK) 14 o superior
- Sistema operativo: Windows
- Espacio en disco: Mínimo 500 MB

5.2 Para Ejecución del .exe

- Sistema operativo: Windows 10 o superior
- NO requiere Java instalado (JRE embebido)
- Espacio en disco: Aproximadamente 200 MB

6. INSTRUCCIONES DE USO

6.1 Compilar el Proyecto

Desde el directorio raíz del proyecto:

1. Abrir terminal o CMD
2. Ejecutar: compilar.bat
3. Esperar mensaje "JAR creado exitosamente!"
4. Verificar que se abre la aplicación para prueba

6.2 Generar Ejecutable

Después de una compilación exitosa:

1. Ejecutar: generar_exe.bat
2. Esperar el proceso de jpackage (puede tardar 1-2 minutos)
3. Verificar mensaje "Ejecutable generado exitosamente!"

6.3 Distribuir la Aplicación

1. Comprimir la carpeta VideoJuego/ completa en un archivo ZIP
2. Distribuir el archivo ZIP
3. El usuario descomprime y ejecuta VideoJuego.exe

7. SOLUCIÓN DE PROBLEMAS

7.1 Error: "javac no se reconoce como comando"

Causa: Java JDK no está en el PATH

Solución:

- Instalar JDK 14 o superior
- Agregar la ruta bin del JDK al PATH del sistema

7.2 Error: "jpackage no se reconoce como comando"

Causa: JDK versión inferior a 14

Solución:

- Actualizar a JDK 14 o superior
- jpackage se incluye desde JDK 14

7.3 Error: "No se encuentra VideoJuego.jar"

Causa: No se ha ejecutado compilar.bat

Solución:

- Ejecutar primero compilar.bat
- Luego ejecutar generar_exe.bat

7.4 Error de compilación en archivos Java

Causa: Código fuente con errores o dependencias faltantes

Solución:

- Revisar mensajes de error de javac
- Verificar que todas las librerías estén en lib/
- Corregir errores de sintaxis en el código

8. NOTAS IMPORTANTES

- El proceso de compilación debe completarse sin errores antes de generar el ejecutable

- El archivo VideoJuego.jar puede ejecutarse directamente con:
  java -jar VideoJuego.jar

- La carpeta VideoJuego/ generada contiene un JRE embebido, por lo que NO requiere que el usuario tenga Java instalado

- El tamaño del ejecutable es mayor (aproximadamente 200 MB) debido al JRE embebido, pero esto garantiza portabilidad total

- Para redistribuir, se debe incluir TODA la carpeta VideoJuego/, no solo el archivo .exe

9. MANTENIMIENTO

9.1 Actualizar el Juego

1. Modificar archivos fuente (.java)
2. Ejecutar compilar.bat
3. Ejecutar generar_exe.bat
4. Redistribuir la nueva carpeta VideoJuego/

9.2 Agregar Nuevas Dependencias

1. Colocar archivo .jar en carpeta lib/
2. El script compilar.bat lo incluirá automáticamente

9.3 Agregar Nuevos Recursos

1. Agregar archivos a tiles/, sprites o data/
2. El script compilar.bat los copiará automáticamente

SOLUCIÓN AL PROBLEMA DE CARGA DE RECURSOS EN EJECUTABLES EMPAQUETADOS

2. DESCRIPCIÓN DEL PROBLEMA

2.1 Síntomas Observados

Al ejecutar el archivo VideoJuego.exe, se presentaron los siguientes problemas:

- El mapa del juego aparecía completamente negro, sin renderizar ningún tile (agua, pasto, piedra, árboles, etc.).

- Los acertijos del juego no se cargaban, impidiendo el funcionamiento correcto del sistema de preguntas y respuestas.

2.2 Análisis de Causa Raíz

Se identificó que el problema radicaba en el método de carga de recursos.
El código original utilizaba rutas del sistema de archivos local, las cuales
no funcionan cuando los recursos están empaquetados dentro de un archivo JAR
o ejecutable.

Específicamente, se encontraron dos implementaciones problemáticas:

ARCHIVO: domain/ManejadorMapaInfinito.java

LÍNEA: 52 (y similares)

CÓDIGO PROBLEMÁTICO:

```
javax.imageio.ImageIO.read(new java.io.File("tiles/agua.png"))
```

ARCHIVO: infrastructure/AcertijosLoader.java

LÍNEA: 27

CÓDIGO PROBLEMÁTICO:

```
Files.readAllBytes(Paths.get("data/data.json"))
```

2.3 Causa Técnica

Los métodos File() y Paths.get() intentan acceder a archivos en el sistema de archivos del sistema operativo. Sin embargo, cuando la aplicación se empaqueta en un JAR o ejecutable, los recursos no existen como archivos separados en el disco, sino que están contenidos dentro del archivo empaquetado.

El resultado es que el sistema intenta buscar archivos que no existen en las ubicaciones especificadas, fallando silenciosamente y retornando nulo, lo que causa que el mapa aparezca negro y los acertijos no se carguen.

3. SOLUCIÓN IMPLEMENTADA

3.1 Enfoque de Solución

La solución consiste en utilizar el mecanismo de carga de recursos desde el classpath de Java, que funciona correctamente tanto en entornos de desarrollo (archivos sueltos) como en entornos de producción (archivos empaquetados).

3.2 Cambios en ManejadorMapaInfinito.java

Se modificó el método cargarTilesVisuales() para utilizar la clase ResourceLoader existente, que implementa correctamente la carga desde classpath.

ANTES (líneas 46-122):

```
tiles[0].setImage(
  javax.imageio.ImageIO.read(new java.io.File("tiles/agua.png"))
);
```

DESPUÉS (líneas 46-105):

```
infrastructure.ResourceLoader loader =
  infrastructure.ResourceLoader.getInstance();
tiles[0].setImage(loader.cargarImagen("tiles/agua.png"));
```

Este cambio se aplicó a todos los tiles (11 tipos diferentes): agua, arbol, arena, muro, pasto, suelo, piedra, nube, volcan, cofre y cofre_cerrado.

3.3 Cambios en AcertijosLoader.java

Se modificó el método cargarAcertijos() para leer el archivo JSON desde el classpath utilizando InputStream.

ANTES (línea 27):

```
String contenido = new String(
  Files.readAllBytes(Paths.get(rutaArchivo))
);
```

DESPUÉS (líneas 25-39):

```
String contenido;
var inputStream = getClass()
  .getClassLoader()
  .getResourceAsStream(rutaArchivo);

if (inputStream != null) {
  contenido = new String(inputStream.readAllBytes());
  inputStream.close();
} else {
  contenido = new String(Files.readAllBytes(Paths.get(rutaArchivo)));
}
```

La solución implementa un patrón de fallback que intenta primero cargar desde el classpath (para archivos empaquetados) y si falla, intenta desde el sistema de archivos (para desarrollo).

3.4 Funcionamiento de ResourceLoader

La clase ResourceLoader (infrastructure/ResourceLoader.java) ya implementaba correctamente el patrón de carga desde classpath:

```
public BufferedImage cargarImagen(String ruta) {
  var resourceStream = getClass()
    .getClassLoader()
    .getResourceAsStream(ruta);

  if (resourceStream != null) {
    imagen = ImageIO.read(resourceStream);
    resourceStream.close();
  } else {
    File file = new File(ruta);
    if (file.exists()) {
      imagen = ImageIO.read(file);
    }
  }

  return imagen;
}
```

Este método garantiza compatibilidad en ambos entornos.

4. VENTAJAS DE LA SOLUCIÓN

- PORTABILIDAD: Funciona tanto en desarrollo como en producción sin cambios
- CONFIABILIDAD: Elimina la dependencia de rutas de archivos externas
- MANTENIBILIDAD: Utiliza patrones estándar de Java para carga de recursos
- COMPATIBILIDAD: Soporta JAR, EXE y ejecución directa desde IDE

5. PROCESO DE VERIFICACIÓN

5.1 Pasos de Compilación

1. Ejecutar compilar.bat para recompilar el proyecto completo
2. Verificar que no hay errores de compilación
3. Confirmar que VideoJuego.jar se genera correctamente

5.2 Pasos de Generación de Ejecutable

1. Ejecutar generar_exe.bat para crear el ejecutable
2. Verificar que se crea la carpeta VideoJuego/
3. Confirmar la existencia de VideoJuego/VideoJuego.exe

5.3 Verificación Funcional

Al ejecutar VideoJuego.exe se debe verificar:

- El mapa se renderiza correctamente mostrando todos los tiles visuales
- Los diferentes tipos de terreno son visibles (agua, pasto, piedra, etc.)
- Los acertijos se cargan y se muestran en el juego
- No aparecen errores en consola relacionados con carga de recursos

6. CONSIDERACIONES IMPORTANTES

6.1 Inclusión de Recursos en el JAR

Es fundamental que el script de compilación incluya todas las carpetas de recursos en el JAR final:

- tiles/
- spritesjugador/
- spritesenemigos/
- data/

El script compilar.bat ya incluye estos comandos (líneas 33-36):

```
xcopy /E /I /Y tiles bin\tiles
xcopy /E /I /Y spritesjugador bin\spritesjugador
xcopy /E /I /Y spritesenemigos bin\spritesenemigos
xcopy /E /I /Y data bin\data
```

6.2 Estructura del Classpath

La estructura debe mantenerse idéntica dentro del JAR:

```
VideoJuego.jar/
  +-- Main/
  +-- domain/
  +-- infrastructure/
  +-- model/
  +-- tiles/
  |   +-- agua.png
  |   +-- pasto.png
  |   +-- ...
  +-- data/
  |   +-- data.json
  +-- ...
```

CORRECCION MENU PRINCIPAL Y DOCUMENTACION GENERAL

SECCION 1: CORRECCION DEL MENU PRINCIPAL

Problema Identificado

Cuando el jugador estaba en el Menu de Pausa o en la pantalla de Juego Terminado y seleccionaba la opcion para volver al Menu Principal, el juego NO se reiniciaba.

Esto causaba que:

. Si el jugador volvía a jugar, continuaba con el estado anterior

. El timer seguía desde donde se quedó

. Los enemigos, items y el mapa seguían en el mismo estado

. El jugador podía explotar esto para evitar perder

Solucion Implementada

Ahora, siempre que el jugador vuelve al Menú Principal, el juego se reinicia automáticamente para asegurar que la próxima partida comience desde cero.

Cambios Realizados

1. Menú de Pausa a Menú Principal

ANTES:

```
} else if (inputService.isTecla3()) {
    estadoJuego = GameState.MENU_PRINCIPAL;
    inputService.setTecla3(false);
}
```

AHORA:

```
} else if (inputService.isTecla3()) {
    gameEngine.reiniciarJuego();
    estadoJuego = GameState.MENU_PRINCIPAL;
    inputService.setTecla3(false);
}
```

2. Juego Terminado a Menú Principal

ANTES:

```
} else if (inputService.isTeclaEscape()) {
    estadoJuego = GameState.MENU_PRINCIPAL;
    inputService.setTeclaEscape(false);
}
```

AHORA:

java

```
} else if (inputService.isTeclaEscape()) {
```

CORRECCION MENU PRINCIPAL Y DOCUMENTACION GENERAL

SECCION 1: CORRECCION DEL MENU PRINCIPAL

Problema Identificado

Cuando el jugador estaba en el Menu de Pausa o en la pantalla de Juego Terminado y seleccionaba la opcion para volver al Menu Principal, el juego NO se reiniciaba.

Esto causaba que:

. Si el jugador volvía a jugar, continuaba con el estado anterior

. El timer seguía desde donde se quedó

. Los enemigos, items y el mapa seguían en el mismo estado

. El jugador podía explotar esto para evitar perder

Solucion Implementada

Ahora, siempre que el jugador vuelve al Menú Principal, el juego se reinicia automáticamente para asegurar que la próxima partida comience desde cero.

Cambios Realizados

1. Menú de Pausa a Menú Principal

ANTES:

```
} else if (inputService.isTecla3()) {
    estadoJuego = GameState.MENU_PRINCIPAL;
    inputService.setTecla3(false);
}
```

AHORA:

```
} else if (inputService.isTecla3()) {
    gameEngine.reiniciarJuego();
    estadoJuego = GameState.MENU_PRINCIPAL;
    inputService.setTecla3(false);
}
```

2. Juego Terminado a Menú Principal

ANTES:

```
} else if (inputService.isTeclaEscape()) {
    estadoJuego = GameState.MENU_PRINCIPAL;
    inputService.setTeclaEscape(false);
}
```

AHORA:

java

```
} else if (inputService.isTeclaEscape()) {
    gameEngine.reiniciarJuego();  // ← REINICIA EL JUEGO
    estadoJuego = GameState.MENU_PRINCIPAL;
    inputService.setTeclaEscape(false);
}
```

SECCION 2: COMPARACION DE FLUJOS

2.1 ANTES (Comportamiento Incorrecto)

Secuencia de eventos:

Jugador esta jugando (Vida: 50, Tiempo: 1:30)

|

Presiona ESC

|

Menu de Pausa

|

Selecciona 3 - Menu Principal

|

Vuelve al Menu

|

Selecciona 1 - Jugar Solo

|

Continua con Vida: 50, Tiempo: 1:30 (PROBLEMA)

2.2 AHORA (Comportamiento Correcto)

Secuencia de eventos:

Jugador esta jugando (Vida: 50, Tiempo: 1:30)

|

Presiona ESC

|

Menu de Pausa

|

Selecciona 3 - Menu Principal

|

gameEngine.reiniciarJuego() (SE REINICIA)

|

Vuelve al Menu

|

Selecciona 1 - Jugar Solo

|

Nueva partida: Vida: 100, Tiempo: 3:00 (CORRECTO)

SECCION 3: CASOS DE USO AFECTADOS

3.1 Caso 1: Salir Durante la Partida

Escenario:

1. Jugador esta jugando
2. Presiona ESC (aparece menu de pausa)
3. Selecciona 3 - Menu Principal
4. Vuelve a seleccionar 1 - Jugar Solo

Resultado:

- Nueva partida completamente limpia
- Vida: 100 HP
- Pociones: 0
- Acertijos: 0
- Timer: 3:00
- Posicion: (5000, 5000)
- Enemigos regenerados

3.2 Caso 2: Tiempo Terminado

Escenario:

1. El tiempo llega a 0:00
2. Aparece pantalla "Juego Terminado"
3. Jugador presiona ESC (volver al menu)
4. Vuelve a seleccionar 1 - Jugar Solo

Resultado:

- Nueva partida completamente limpia
- Todas las estadisticas reseteadas
- Timer reinicia a 3:00

SECCION 4: OPCIONES EN EL MENU DE PAUSA

Comportamiento consistente:

Opcion	Tecla	Accion	Reinicia
Reanudar	1	Continua jugando	No
Reiniciar	2	Reinicia y vuelve a jugar	Si
Menu Principal	3	Reinicia y va al menu	Si (NUEVO)
Volver	ESC	Vuelve al juego	No

SECCION 5: OPCIONES EN JUEGO TERMINADO

Opcion	Tecla	Accion	Reinicia
Jugar de nuevo	ENTER	Reinicia y vuelve a jugar	Si
Menu Principal	ESC	Reinicia y va al menu	Si (NUEVO)

SECCION 6: BENEFICIOS DE LA CORRECCION

1. Consistencia

   Siempre que vuelves al menu, el juego se resetea

2. Sin Exploits

   Los jugadores no pueden "guardar" progreso saliendo

3. Experiencia Limpia

   Cada partida comienza desde cero

4. Previsibilidad

   El comportamiento es claro y logico

SECCION 7: ARCHIVOS MODIFICADOS

Main/GamePanel.java

- Linea 159: Agregado gameEngine.reiniciarJuego() en opcion 3
- Linea 240: Agregado gameEngine.reiniciarJuego() al presionar ESC

SECCION 8: PRUEBAS REALIZADAS

Estado de verificacion:

- Compilacion exitosa sin errores
- Volver al menu desde pausa reinicia correctamente
- Volver al menu desde juego terminado reinicia correctamente
- Nueva partida comienza con estadisticas limpias
- Timer se resetea a 3:00
- Mapa se regenera
- Enemigos se regeneran