INFORME DEL TRABAJO PRÁCTICO

"La Invasión de los Zombies Grinch"

ENCABEZADO

Universidad: [Nombre de la Universidad]
Facultad: [Nombre de la Facultad]
Carrera: [Nombre de la Carrera]
Materia: Programación con Objetos I
Docente: [Nombre del Docente]

INTEGRANTES DEL GRUPO:

Nombre y Apellido	Legajo	Email
[Integrante 1]	[Legajo 1]	[email1@dominio.com]
[Integrante 2]	[Legajo 2]	[email2@dominio.com]
[Integrante 3]	[Legajo 3]	[email3@dominio.com]

Fecha de Entrega: [Fecha]
Año: 2024

INTRODUCCIÓN

Este trabajo práctico consiste en el desarrollo de un juego de estrategia tipo "Tower Defense" inspirado en el clásico Plants vs Zombies, implementado completamente en Java utilizando la librería gráfica "entorno".

Objetivos principales:

- Aplicar los conceptos de Programación Orientada a Objetos (herencia, polimorfismo, encapsulamiento)
- Implementar un sistema de juego complejo con múltiples estados y transiciones
- Desarrollar mecánicas de combate, colisiones y progresión de niveles
- Crear una interfaz de usuario intuitiva con controles duales (mouse y teclado)
- Gestionar recursos gráficos y lógica de juego de manera eficiente

El juego presenta una temática navideña donde el jugador debe defender regalos del ataque de zombies Grinch utilizando plantas con habilidades especiales a lo largo de 4 niveles de dificultad creciente.

DESCRIPCIÓN

ARQUITECTURA GENERAL

El juego sigue una arquitectura modular con separación clara de responsabilidades:

```
Juego (Principal)
├── Sistema de Estados
│   ├── MENU_PRINCIPAL
│   ├── INSTRUCCIONES
│   ├── JUGANDO
│   ├── GANO
│   └── PERDIO
├── Sistema de Plantas
│   ├── RoseBlade (Ofensiva)
│   ├── WallNut (Defensiva)
│   └── IceFlower (Soporte)
├── Sistema de Enemigos
│   ├── ZombieGrinch (Básico)
│   ├── SuperZombieGrinch (Avanzado)
│   └── ZombieBoss (Final)
├── Sistema de Proyectiles
│   ├── BolaDeFuego (Daño)
│   └── BolaDeHielo (Control)
├── Sistema de UI
│   ├── Barra Superior
│   ├── Selector de Plantas
│   └── Indicadores de Estado
└── Sistema de Niveles
    ├── Progresión Dificultad
    ├── Generación de Enemigos
    └── Condiciones de Victoria
```

DETALLE COMPLETO DE CLASES

1. Clase `Juego` (Principal)

Variables de instancia:

```
// Configuración gráfica
private Entorno entorno;
private static final int ANCHO_VENTANA = 800;
private static final int ALTO_VENTANA = 600;

// Sistema de estados
private EstadoJuego estadoActual;
private int opcionMenuSeleccionada;
private boolean juegoPausado;

// Sistema de césped y plantas
private Planta[][] cesped;
private static final int FILAS_CESPED = 5;
private static final int COLUMNAS_CESPED = 9;
private int TAMANIO_CASILLA; // Calculado dinámicamente

// Sistema de enemigos
private Object[] zombies;
private ZombieBoss jefe;
private int zombiesEliminados;
private int zombiesRestantes;

// Sistema de proyectiles
private BolaDeFuego[] bolasFuego;
private BolaDeHielo[] bolasHielo;

// Sistema de objetivos
private Regalo[] regalos;

// Sistema de selección
private String plantaSeleccionada;
private boolean arrastrando;
private Planta plantaSeleccionadaParaMover;

// Sistema de tiempo
private long tiempoInicio;
private long tiempoTranscurrido;
private boolean tiempoDetenido;
```

```
// Sistema de niveles
private int nivelActual;
private int zombiesParaSiguienteNivel;
private boolean nivelCompletado;
private boolean jefeAparecio;

// Recursos gráficos
private Image imagenFondoMenu;

private Image avatarRoseBlade, avatarWallNut, avatarIceFlower;
```

Métodos principales del sistema:

Métodos de Inicialización:

- `Juego()`: Constructor principal que inicializa todos los sistemas
- `inicializarSistemaEstados()`: Configura máquina de estados
- `inicializarSistemaNiveles()`: Prepara progresión de niveles
- `calcularDimensionesCesped()`: Calcula posiciones dinámicamente
- `inicializarSistemaCesped()`: Crea matriz de plantas
- `inicializarSistemaObjetivos()`: Coloca regalos a defender
- `inicializarSistemaEnemigos()`: Prepara arrays de zombies
- `inicializarSistemaDisparos()`: Configura proyectiles
- `inicializarSistemaPlantas()`: Carga recursos de plantas
- `cargarImagenesMenu()`: Carga assets del menú
- `inicializarSistemaSeleccion()`: Prepara sistema de selección

Método `tick()` - Bucle Principal:

```
public void tick() {
    switch (estadoActual) {
        case MENU_PRINCIPAL:
            ejecutarMenuPrincipal();
            break;
        case INSTRUCCIONES:
            ejecutarPantallaInstrucciones();
            break;
        case JUGANDO:
            ejecutarJuegoPrincipal();
            break;
        case GANO:
            ejecutarPantallaVictoria();
            break;
        case PERDIO:
            ejecutarPantallaGameOver();
```

```
            break;
    }
}
```

Sistema de Menú Principal:

- `ejecutarMenuPrincipal()`: Coordina menú completo
- `dibujarMenuPrincipal()`: Renderiza todos los elementos del menú
- `dibujarFondoMenu()`: Fondo con imagen o color sólido
- `dibujarTituloJuego()`: Título principal con efectos
- `dibujarOpcionesMenu()`: Lista de opciones seleccionables
- `dibujarOpcionMenu()`: Opción individual con estado de selección
- `procesarInputMenu()`: Maneja toda la entrada del usuario
- `procesarNavegacionMenu()`: Flechas para cambiar opción
- `procesarSeleccionMenu()`: Enter para confirmar selección

Sistema de Juego Principal:

- `ejecutarJuegoPrincipal()`: Frame completo del juego
- `actualizarSistemas()`: Actualiza toda la lógica del juego
- `dibujarElementosJuego()`: Renderiza todos los elementos visuales
- `procesarInputJuego()`: Maneja controles durante el juego
- `verificarCondicionesJuego()`: Chequea victoria/derrota

Actualización de Sistemas (`actualizarSistemas()`):

- `actualizarTiemposRecarga()`: Decrementa contadores de plantas
- `actualizarPlantas()`: Actualiza estado de todas las plantas
- `actualizarEnemigos()`: Mueve y actualiza todos los zombies
- `actualizarProyectiles()`: Mueve bolas de fuego/hielo
- `generarNuevosZombies()`: Crea enemigos según progresión
- `manejarDisparos()`: Gera creación de proyectiles
- `manejarColisiones()`: Detecta y procesa colisiones
- `limpiarElementosMuertos()`: Elimina elementos destruidos

Sistema de Colisiones:

- `manejarColisionesFuego()`: Colisiones de bolas de fuego
- `manejarColisionesHielo()`: Colisiones de bolas de hielo
- `verificarColisionFuegoConZombies()`: Fuego vs zombies normales
- `verificarColisionFuegoConJefe()`: Fuego vs jefe
- `verificarColisionHieloConZombies()`: Hielo vs zombies
- `verificarColisionHieloConJefe()`: Hielo vs jefe

- `aplicarEfectoHielo()`: Aplica efectos según tipo de zombie

Sistema de Dibujo (`dibujarElementosJuego()`):

- `dibujarFondo()`: Fondo según nivel actual
- `dibujarCesped()`: Cuadrícula del campo de juego
- `dibujarRegalos()`: Objetivos a proteger
- `dibujarPlantas()`: Renderiza todas las plantas
- `dibujarBarraSuperior()`: Interfaz de usuario superior
- `dibujarEnemigos()`: Dibuja todos los zombies
- `dibujarProyectiles()`: Renderiza bolas de fuego/hielo
- `dibujarElementosSeleccion()`: Elementos de selección

Sistema de UI Superior:

- `dibujarBarraSuperior()`: Contenedor principal
- `dibujarSeccionNivel()`: Información del nivel actual
- `dibujarSeccionProgreso()`: Progreso de zombies eliminados
- `dibujarSeccionPlantas()`: Selector de plantas
- `dibujarSeccionTiempo()`: Tiempo y estadísticas
- `dibujarAvatarRoseBlade()`, `dibujarAvatarWallNut()`, `dibujarAvatarIceFlower()`: Iconos de plantas

Sistema de Selección y Movimiento:

- `manejarSeleccionYArrastre()`: Control con mouse
- `manejarClicBarraPlantas()`: Selección desde barra
- `manejarClicCesped()`: Selección de plantas existentes
- `manejarArrastrePlanta()`: Arrastre visual
- `manejarSueltaPlanta()`: Colocación en césped
- `manejarSeleccionConTeclado()`: Selección con teclado
- `manejarMovimientoConTeclado()`: Movimiento WASD/flechas
- `moverPlantaDireccion()`: Movimiento en dirección específica
- `moverPlantaACasilla()`: Traslado entre casillas

Sistema de Verificación:

- `verificarDerrota()`: Chequea condiciones de pérdida
- `verificarAvanceNivel()`: Verifica progresión de nivel
- `verificarVictoria()`: Condiciones de victoria final
- `zombiesAlcanzaronRegalos()`: Derrota por invasión
- `jefeAlcanzoRegalos()`: Derrota por jefe

Sistema de Reinicio:

- `reiniciarJuegoCompleto()`: Reinicio total
- `reiniciarParaNuevoNivel()`: Preparación entre niveles
- `reiniciarSistemaNiveles()`, `reiniciarSistemaCesped()`, etc.: Reinicios individuales

2. Clase `Planta` (Abstracta)

Variables:

```
protected double x, y;      // Posición en pantalla
protected int salud;        // Puntos de vida

protected String tipo;      // Identificador
```

Métodos:

- `Planta(double x, double y, String tipo)`: Constructor base
- `getX()`, `getY()`, `getSalud()`, `getTipo()`: Getters
- `setSalud(int salud)`, `setX(double x)`, `setY(double y)`: Setters
- `estaViva()`: Verifica si la planta está activa (salud > 0)
- `dibujar(Entorno entorno)`: Método abstracto para renderizado

3. Clase `RoseBlade`

Variables:

```
private int tiempoRecarga;

private static final int RECARGA_MAXIMA = 120;

private Image imagenRoseBlade;
```

Métodos:

- `RoseBlade(double x, double y)`: Constructor específico
- `dibujar(Entorno entorno)`: Renderizado con imagen o geometría
- `actualizar()`: Decrementa tiempo de recarga si está viva
- `puedeDisparar()`: Verifica si recarga == 0 y está viva
- `reiniciarRecarga()`: Restablece contador de recarga
- `hayZombiesEnFilaYDelante()`: Lógica de detección de objetivos

4. Clase `IceFlower`

Variables:

```
private int tiempoRecarga;
private static final int RECARGA_MAXIMA = 200;

private Image imagenIceFlower;
```

Métodos:

- Similar estructura a RoseBlade pero con diferentes parámetros
- Dispara bolas de hielo con efectos de ralentización

5. Clase `WallNut`

Variables:

```
private Image imagenWallNut;

// Salud inicial: 5 puntos
```

Métodos:

- `WallNut(double x, double y)`: Constructor defensivo
- `dibujar(Entorno entorno)`: Renderizado de nuez defensiva
- Sin métodos de ataque (planta puramente defensiva)

6. Clase `ZombieGrinch`

Variables:

```
private double x, y, velocidad;
private int salud, fila;
private boolean atacando;
private Planta plantaObjetivo;
private int tickAtaque;

private Image imagenZombie, imagenZombieAtacando;
```

Métodos:

- `ZombieGrinch(double x, double y, int fila)`: Constructor básico
- `mover(Planta[][])`: Movimiento y detección de plantas
- `atacar()`: Daña planta objetivo cada 60 ticks
- `detectarPlantaCercana()`: Encuentra plantas en rango
- `recibirDano()`: Reduce salud en 1 punto
- `estaMuerto()`: Verifica salud <= 0

- `dibujar(Entorno)`: Renderizado con imágenes o geometría
- `dibujarGeometrico(Entorno)`: Fallback visual

7. Clase `SuperZombieGrinch` (hereda de ZombieGrinch)

Variables:

```
private boolean inmuneACongelacion;

private Image imagenSuperZombie;
```

Métodos:

- `SuperZombieGrinch(double x, double y, int fila)`: Constructor avanzado
- `esInmuneACongelacion()`: Getter de inmunidad
- `aplicarRalentizacion(double factor)`: Ralentización limitada
- `dibujar(Entorno)`: Renderizado especial para super zombie
- `dibujarGeometrico(Entorno)`: Versión geométrica

8. Clase `ZombieBoss`

Variables:

```
private double x, y, velocidad, velocidadBase;
private int salud, tickAtaque;
private final int RECARGA_ATAQUE = 600;
private boolean ataqueActivo;
private int tickRalentizacion;

private Image imagenBoss;
```

Métodos:

- `ZombieBoss(double x, double y)`: Constructor de jefe
- `mover()`: Movimiento hacia izquierda
- `actualizar()`: Actualiza temporizador de ataque
- `ejecutarAtaque(Planta[][])`: Ataque masivo que elimina 5 plantas
- `aplicarRalentizacion(double factor)`: Solo ralentización, no congelación
- `recibirDano(int dano)`: Reduce salud del jefe
- `dibujar(Entorno)`: Renderizado con barras de información
- `dibujarGeometrico(Entorno)`: Versión geométrica de respaldo

9. Clase `BolaDeFuego`

Variables:

```
private double x, y, velocidad;

private int dano;
```

Métodos:

- `BolaDeFuego(double x, double y)`: Constructor de proyectil
- `mover()`: Desplazamiento hacia derecha
- `estaFueraDePantalla(int anchoPantalla)`: Verifica límites
- `dibujar(Entorno)`: Renderizado con efectos de fuego

10. Clase `BolaDeHielo`

Variables:

```
private double x, y, velocidad;
private int dano, hits;

private boolean activa;
```

Métodos:

- `BolaDeHielo(double x, double y)`: Constructor de hielo
- `incrementarHits()`: Aumenta contador de impactos
- `desactivar()`: Desactiva el proyectil
- `mover()`: Movimiento más lento que fuego
- `estaFueraDePantalla(int anchoPantalla)`: Verifica límites
- `dibujar(Entorno)`: Renderizado con efectos de hielo

11. Clase `Regalo`

Variables:

```
private double x, y;

private boolean destruido;

private Image imagenRegalo;
```

Métodos:

- `Regalo(double x, double y)`: Constructor de objetivo
- `getX()`, `getY()`, `isDestruido()`: Getters

- `destruir()`: Marca como destruido
- `dibujar(Entorno)`: Renderizado del regalo
- `dibujarGeometrico(Entorno)`: Versión geométrica

PROBLEMAS ENCONTRADOS Y SOLUCIONES

1. Problema: Gestión de Colisiones Complejas

Problema: Diferentes tipos de proyectiles (fuego/hielo) con efectos distintos sobre diferentes enemigos (normal/super/boss). La lógica se volvía muy compleja con múltiples condicionales anidadas.

Solución: Implementamos un sistema de doble dispatch basado en tipos:

```
private void aplicarEfectoHielo(Object zombie, BolaDeHielo bolaHielo) {
    if (zombie instanceof SuperZombieGrinch) {
        aplicarEfectoHieloSuperZombie((SuperZombieGrinch) zombie,
        bolaHielo);
    } else if (zombie instanceof ZombieGrinch) {
        aplicarEfectoHieloZombieNormal((ZombieGrinch) zombie, bolaHielo);
    }
}

private void aplicarEfectoHieloSuperZombie(SuperZombieGrinch superZombie,
BolaDeHielo bolaHielo) {
    double factorRalentizacion = 1.0 - (bolaHielo.getHits() * 0.15);
    superZombie.aplicarRalentizacion(factorRalentizacion);
    aplicarDanoZombie(superZombie, 1);
    bolaHielo.desactivar();

}
```

2. Problema: Coordenadas del Césped Dinámicas

Problema: El césped necesitaba adaptarse a diferentes resoluciones manteniendo proporciones visuales correctas. Las posiciones fijas causaban desalineaciones.

Solución: Sistema de cálculo dinámico basado en el área disponible:

```
private void calcularDimensionesCesped() {
```

```
    // Margenes para UI
    int margenSuperior = 120;
    int margenInferior = 50;
    int margenLateral = 20;

    // Área disponible para el césped
    int anchoDisponible = ANCHO_VENTANA - (2 * margenLateral);
    int altoDisponible = ALTO_VENTANA - margenSuperior - margenInferior;

    // Calcular tamaño de casilla basado en la dimensión más restrictiva
    int tamCasillaPorAncho = anchoDisponible / COLUMNAS_CESPED;
    int tamCasillaPorAlto = altoDisponible / FILAS_CESPED;

    // Usar el tamaño más pequeño para mantener proporción
    this.TAMANIO_CASILLA = Math.min(tamCasillaPorAncho, tamCasillaPorAlto);

    // Calcular posiciones iniciales para centrar
    int anchoTotalCesped = COLUMNAS_CESPED * TAMANIO_CASILLA;
    int altoTotalCesped = FILAS_CESPED * TAMANIO_CASILLA;

    this.CESPED_X_INICIO = margenLateral + (anchoDisponible -
anchoTotalCesped) / 2 + TAMANIO_CASILLA / 2;
    this.CESPED_Y_INICIO = margenSuperior + (altoDisponible -
altoTotalCesped) / 2 + TAMANIO_CASILLA / 2;

}
```

3. Problema: Estados del Juego y Transiciones

Problema: Las transiciones entre menú, juego, pausa, victoria y derrota generaban código espagueti con muchos flags booleanos.

Solución: Implementación de máquina de estados explícita:

```
private enum EstadoJuego {
    MENU_PRINCIPAL,     // Pantalla principal con opciones
    JUGANDO,            // Juego activo
    INSTRUCCIONES,      // Pantalla de cómo jugar
    GANO,               // Pantalla de victoria
    PERDIO              // Pantalla de game over
}

public void tick() {
    switch (estadoActual) {
```

```
        case MENU_PRINCIPAL:
            ejecutarMenuPrincipal();
            break;
        case INSTRUCCIONES:
            ejecutarPantallaInstrucciones();
            break;
        // ... otros estados
    }

}
```

4. Problema: Carga de Recursos Gráficos

Problema: Las imágenes podían fallar al cargar, rompiendo el juego completamente si no existían los archivos.

Solución: Sistema robusto de fallback con respaldo geométrico:

```
java
public ZombieBoss(double x, double y) {
    this.x = x;
    this.y = y;
    this.velocidadBase = 0.08;
    this.velocidad = velocidadBase;
    this.salud = 100;

    try {
        this.imagenBoss = Herramientas.cargarImagen("img/zombie_boss.png");
    } catch (Exception e) {
        System.out.println("Error al cargar imagen del Zombie Boss: " +
e.getMessage());
        this.imagenBoss = null; // Usará dibujo geométrico
    }
}

public void dibujar(Entorno entorno) {
    if (imagenBoss != null) {
        entorno.dibujarImagen(imagenBoss, x, y, 0, 0.8);
    } else {
        dibujarGeometrico(entorno); // Fallback geométrico
    }
    // ... resto del dibujo

}
```

5. Problema: Selección y Movimiento de Plantas

Problema: Dos sistemas de interacción (arrastre mouse + teclado) que podían conflictuar y causar estados inconsistentes.

Solución: Estados mutuamente excluyentes con reset adecuado:

```
private void seleccionarPlantaParaColocar(String tipoPlanta) {
    plantaSeleccionada = tipoPlanta;
    arrastrando = true;
    resetearSeleccionTeclado(); // Asegura que no hay conflicto
}

private void seleccionarPlantaParaMover(Planta planta, int fila, int
columna) {
    plantaSeleccionadaParaMover = planta;
    plantaSeleccionadaConTeclado = true;
    filaPlantaSeleccionada = fila;
    columnaPlantaSeleccionada = columna;
    plantaSeleccionada = null; // Limpia selección de arrastre
    arrastrando = false;
}

private void resetearSeleccionTeclado() {
    plantaSeleccionadaParaMover = null;
    plantaSeleccionadaConTeclado = false;
    filaPlantaSeleccionada = -1;
    columnaPlantaSeleccionada = -1;

}
```

6. Problema: Generación Balanceada de Enemigos

Problema: La generación de zombies necesitaba ser progresiva según el nivel, manteniendo desafío pero sin ser abrumadora.

Solución: Sistema de probabilidades por nivel y generación controlada por ticks:

```
private Object crearZombieSegunNivel() {
    int filaAleatoria = (int)(Math.random() * FILAS_CESPED);
    int y = calcularPosicionYCesped(filaAleatoria);

    switch (nivelActual) {
        case 1:
```

```
            return new ZombieGrinch(850, y, filaAleatoria);
        case 2:
            if (Math.random() < 0.7) {
                return new ZombieGrinch(850, y, filaAleatoria);
            } else {
                return new SuperZombieGrinch(850, y, filaAleatoria);
            }
        case 3:
            return new SuperZombieGrinch(850, y, filaAleatoria);
        case 4:
            if (Math.random() < 0.4) {
                return new ZombieGrinch(850, y, filaAleatoria);
            } else if (Math.random() < 0.8) {
                return new SuperZombieGrinch(850, y, filaAleatoria);
            }
            return null;
        default:
            return new ZombieGrinch(850, y, filaAleatoria);
    }

}
```

7. Problema: Gestión de Memoria con Arrays

Problema: Uso intensivo de arrays para zombies y proyectiles podía causar fugas de memoria o límites excedidos.

Solución: Sistema de limpieza periódica y reutilización de slots:

```
private void limpiarElementosMuertos() {
    limpiarPlantasMuertas();
    // Los proyectiles se limpian automáticamente al salir de pantalla
    // Los zombies se eliminan cuando mueren en actualizarEnemigos()
}

private void limpiarPlantasMuertas() {
    for (int fila = 0; fila < FILAS_CESPED; fila++) {
        for (int columna = 0; columna < COLUMNAS_CESPED; columna++) {
            if (cesped[fila][columna] != null &&
cesped[fila][columna].getSalud() <= 0) {
                if (plantaSeleccionadaParaMover == cesped[fila][columna]) {
                    resetearSeleccionTeclado();
                }
                cesped[fila][columna] = null; // Libera la posición
```

```
            }
        }
    }
}
```

IMPLEMENTACIÓN

CARACTERÍSTICAS TÉCNICAS DESTACADAS

1. Sistema de Colisiones Avanzado

Método `manejarColisiones()`:

- Función: Coordina todas las detecciones de colisiones del juego
- Lógica: Separa colisiones por tipo de proyectil para procesamiento especializado
- Efecto: Garantiza que cada tipo de proyectil aplique sus efectos específicos

Método `manejarColisionesFuego()`:

- Función: Procesa colisiones de todas las bolas de fuego activas
- Lógica: Primero verifica zombies normales, luego el jefe si no hubo colisión
- Optimización: Evita procesamiento duplicado mediante flag `colisiono`

Método `verificarColisionFuegoConZombies(int indiceBola)`:

- Función: Detecta colisiones entre bola de fuego y zombies
- Lógica: Usa distancia Manhattan para detección simple y eficiente
- Umbrales: 30px horizontal, 40px vertical para hitboxes oblongas
- Resultado: Aplica daño y destruye el proyectil

2. Sistema de Efectos por Tipo

Método `aplicarEfectoHielo(Object zombie, BolaDeHielo bolaHielo)`:

- Función: Dispatcher principal para efectos de hielo
- Patrón: Implementa double dispatch basado en tipos de runtime
- Ventaja: Elimina condicionales complejos y facilita extensibilidad

Método `aplicarEfectoHieloZombieNormal(ZombieGrinch zombie, BolaDeHielo bolaHielo)`:

- Función: Aplica efecto acumulativo de hielo a zombies normales
- Mecánica:
    - Ralentización progresiva (20% por hit)
    - Daño constante (1 punto)
    - Congelación total al tercer impacto
- Balance: Proyectil se desactiva después de congelar

Método `aplicarEfectoRalentizacion(ZombieGrinch zombie, int hits)`:

- Función: Calcula y aplica ralentización progresiva
- Fórmula: 20% de reducción por hit, mínimo 30% de velocidad
- Base: Velocidad normal de zombie (0.5) como referencia

3. Sistema de Interfaz de Usuario Completa

Método `dibujarBarraSuperior()`:

- Función: Coordina el renderizado de toda la UI superior
- Organización: Divide en secciones temáticas con posiciones predefinidas
- Flujo: Secuencia ordenada para superposición correcta de elementos

Método `dibujarSeccionProgreso(int x, int y)`:

- Función: Muestra progreso de zombies eliminados hacia siguiente nivel
- Componentes:
    - Icono visual del zombie
    - Texto descriptivo
    - Contador numérico
    - Barra de progreso gráfica
- UX: Información clara y visualmente atractiva

Método `dibujarBarraProgresoZombies(int x, int y)`:

- Función: Renderiza barra de progreso visual con porcentaje
- Cálculos: Conversión de valores numéricos a dimensiones visuales
- Feedback visual: Cambio de color según progreso (rojo → amarillo → verde)

4. Sistema de Progresión de Niveles

Método `verificarAvanceNivel()`:

- Función: Controla transición automática entre niveles
- Condiciones:
    - No estar en último nivel
    - Haber eliminado zombies suficientes
    - No haber completado ya el nivel actual
- Transición: Incrementa nivel y reinicia sistemas necesarios

Método `generarNuevosZombies()`:

- Función: Controla la generación progresiva de enemigos
- Mecánica:
    - Basada en ticks para control de frecuencia
    - Busca slots vacíos en el array
    - Condición especial para jefe final
- Balance: Evita sobrepoblación y mantiene desafío constante

Método `obtenerFrecuenciaGeneracion()`:

- Función: Define ritmo de generación según dificultad
- Progresión:
    - Nivel 1: 180 ticks (más lento, para aprendizaje)
    - Nivel 2: 150 ticks (incremento gradual)
    - Nivel 3: 120 ticks (alto ritmo)
    - Nivel 4: 200 ticks (más lento para enfocarse en jefe)

5. Sistema de Gestión de Recursos

Método `manejarDisparos()`:

- Función: Coordina la creación de todos los proyectiles
- Estrategia: Itera sobre todas las plantas vivas y delega por tipo
- Optimización: Solo procesa plantas existentes y vivas

Método `manejarDisparosRoseBlade(int fila, int columna)`:

- Función: Gestiona disparos específicos de RoseBlade
- Condiciones:
    - Planta debe poder disparar (recarga completa)
    - Debe haber zombies en la fila y adelante
- Posicionamiento: Proyectil sale 30px a la derecha de la planta

6. Sistema de Reinicio y Gestión de Estado

Método `reiniciarJuegoCompleto()`:

- Función: Restaura el juego a estado inicial completo
- Modularidad: Delega a subsistemas específicos para reset
- Uso: Al iniciar nuevo juego o después de game over

Método `reiniciarParaNuevoNivel()`:

- Función: Prepara el juego para un nuevo nivel manteniendo progreso
- Preserva: Plantas colocadas y configuración del césped
- Reinicia: Enemigos, proyectiles y flags de nivel

7. Sistema de Coordenadas Dinámicas

Método `calcularDimensionesCesped()`:

- Función: Calcula posiciones y tamaños adaptativos del césped
- Adaptabilidad: Funciona para cualquier resolución manteniendo proporciones
- Centrado: Calcula offsets para centrar el césped en pantalla

Método `calcularPosicionXCesped(int columna)`:

- Función: Convierte coordenadas de matriz a coordenadas de pantalla (X)
- Uso: Para posicionamiento preciso de plantas y detección de clics

Método `calcularPosicionYCesped(int fila)`:

- Función: Convierte coordenadas de matriz a coordenadas de pantalla (Y)
- Consistencia: Garantiza que todos los elementos usen el mismo sistema de coordenadas

---

CONCLUSIONES

LOGROS TÉCNICOS PRINCIPALES

1. Arquitectura Escalable: El sistema modular permite fácil expansión con nuevos tipos de plantas, enemigos y mecánicas
2. Código Mantenible: Separación clara de responsabilidades y uso apropiado de patrones OOP
3. Rendimiento Optimizado: Gestión eficiente de memoria con arrays de tamaño fijo y limpieza periódica

4. Experiencia de Usuario Pulida: Interfaz intuitiva con múltiples sistemas de control e información clara

APRENDIZAJES TÉCNICOS

1. Diseño de Arquitecturas de Juego:
    - La máquina de estados es fundamental para gestionar pantallas complejas
    - La separación entre lógica de juego y renderizado mejora el mantenimiento
    - Los sistemas de colisiones necesitan ser eficientes y modulares
2. Manejo de Recursos:
    - Los fallbacks gráficos son esenciales para robustez
    - La gestión de memoria en tiempo real requiere estrategias específicas
    - Los sistemas de carga deben ser tolerantes a fallos
3. Diseño de APIs:
    - Los métodos deben tener responsabilidades únicas y claras
    - Los sistemas de interacción necesitan manejar estados conflictivos
    - La documentación durante el desarrollo acelera el debugging

MECÁNICAS DE JUEGO IMPLEMENTADAS EXITOSAMENTE

1. Sistema de Combate por Tipos: Fuego vs Hielo con resistencias específicas
2. Progresión de Dificultad: 4 niveles con enemigos y mecánicas progresivas
3. Estrategia de Posicionamiento: Plantas con roles complementarios
4. Gestión de Recursos: Tiempos de recarga y colocación limitada
5. Objetivos Claros: Sistema de victoria/derrota bien definido

POSIBLES MEJORAS FUTURAS

1. Sistema de Sonido: Efectos de sonido y música ambiental
2. Más Contenido: Plantas y enemigos adicionales con habilidades únicas
3. Sistema de Guardado: Persistencia de progreso entre sesiones
4. Efectos Visuales: Partículas, animaciones y transiciones
5. Modo Multijugador: Cooperativo o competitivo
6. Editor de Niveles: Creación de niveles personalizados

IMPACTO DEL DISEÑO ORIENTADO A OBJETOS

El uso consistente de los principios OOP demostró su valor en:

- Encapsulamiento: Cada clase maneja su estado internamente
- Herencia: Reducción de código duplicado en jerarquías de plantas/enemigos
- Polimorfismo: Comportamientos específicos mediante sobreescritura de métodos
- Abstracción: Interfaces claras entre sistemas del juego

REFLEXIÓN FINAL

Este proyecto no solo cumplió con los objetivos académicos de demostrar competencia en programación orientada a objetos, sino que también resultó en un producto jugable y entretenido. La arquitectura diseñada permitió que características complejas como el sistema de estados, colisiones avanzadas y progresión de niveles se implementarán de manera mantenible y extensible.

La experiencia reforzó la importancia del diseño anticipado, las pruebas incrementales y la documentación continua. El resultado final es un testimonio de cómo los principios sólidos de programación pueden traducirse en experiencias de usuario pulidas y engaging.

Fin del Informe

CODIGO DEL JUEGO

```
package juego;
import java.awt.Color;
import java.awt.Image;
import entorno.Entorno;
import entorno.InterfaceJuego;
import entorno.Herramientas;
/**
* JUEGO PRINCIPAL: LA INVASIÓN DE LOS ZOMBIES GRINCH
*
*/
public class Juego extends InterfaceJuego {

  // ==================== CONSTANTES Y CONFIGURACIÓN ====================

        // Configuración de la ventana
        private static final int ANCHO_VENTANA = 800;
        private static final int ALTO_VENTANA = 600;
        // Configuración del césped - OCUPANDO TODA LA PANTALLA
        private static final int FILAS_CESPED = 5;
        private static final int COLUMNAS_CESPED = 9; // Reducido para mejor proporción
        // CÁLCULO DINÁMICO DEL TAMAÑO DE CASILLA
        private int TAMANIO_CASILLA; // Ya no es constante, se calcula
        // POSICIONES CENTRADAS
        private int CESPED_X_INICIO;
        private int CESPED_Y_INICIO;

  // Tiempos de recarga de plantas (en ticks)
  private static final int RECARGA_ROSE_BLADE = 150;
  private static final int RECARGA_WALL_NUT = 200;
  private static final int RECARGA_ICE_FLOWER = 180;

  // Configuración de niveles
  private static final int TOTAL_NIVELES = 4;
  private static final int ZOMBIES_NIVEL_1 = 50;

  // Opciones del menú principal
  private static final String[] OPCIONES_MENU = {
     "JUGAR",
     "INSTRUCCIONES",
     "SALIR"
  };
  private static final int[] POSICIONES_Y_MENU = {280, 340, 400};

  // ==================== ENUMERACIONES ====================

  /**
   * Estados posibles del juego para manejar diferentes pantallas
   */
  private enum EstadoJuego {
     MENU_PRINCIPAL,    // Pantalla principal con opciones
```

```
    JUGANDO,          // Juego activo
    INSTRUCCIONES,    // Pantalla de cómo jugar
    GANO,             // Pantalla de victoria
    PERDIO            // Pantalla de game over
}

// ==================== COMPONENTES PRINCIPALES ====================

// Motor gráfico y estado del juego
private Entorno entorno;
private EstadoJuego estadoActual;

// Sistema de menú principal
private int opcionMenuSeleccionada;

// Imágenes del juego
private Image imagenFondoMenu;
private Image avatarRoseBlade;
private Image avatarWallNut;
private Image avatarIceFlower;

// ==================== SISTEMA DE NIVELES Y PROGRESO ====================

private int nivelActual;
private int zombiesParaSiguienteNivel;
private boolean nivelCompletado;
private boolean jefeAparecio;

// ==================== CONTROL DE TIEMPO ====================

private long tiempoInicio;
private long tiempoTranscurrido;
private boolean tiempoDetenido;
private boolean juegoPausado;

// ==================== SISTEMA DE PLANTAS ====================

// Matriz que representa el césped con plantas
private Planta[][] cesped;

// Tiempos de recarga actuales
private int tiempoCargaRoseBlade;
private int tiempoCargaWallNut;
private int tiempoCargaIceFlower;

// ==================== SISTEMA DE SELECCIÓN Y ARRASTRE ====================

private String plantaSeleccionada;        // Tipo de planta seleccionada para colocar
private boolean arrastrando;              // Si se está arrastrando una planta
private int plantaArrastradaX, plantaArrastradaY; // Posición del arrastre
```

```
  // ==================== SISTEMA DE MOVIMIENTO CON TECLADO
====================

  private Planta plantaSeleccionadaParaMover; // Planta seleccionada para mover
  private boolean plantaSeleccionadaConTeclado;
  private int filaPlantaSeleccionada;
  private int columnaPlantaSeleccionada;

  // ==================== SISTEMA DE OBJETIVOS Y ENEMIGOS ====================

  private Regalo[] regalos;              // Objetivos a proteger
  private Object[] zombies;              // Array de enemigos (puede contener diferentes tipos)
  private ZombieBoss jefe;               // Jefe final (nivel 4)

  // ==================== SISTEMA DE DISPAROS Y PROYECTILES
====================

  private BolaDeFuego[] bolasFuego;
  private BolaDeHielo[] bolasHielo;

  // ==================== ESTADÍSTICAS Y PROGRESO ====================

  private int zombiesEliminados;
  private int zombiesRestantes;
  private int tickGeneracionZombie;

  // ========================================================================
  // CONSTRUCTOR PRINCIPAL
  // ========================================================================

  /**
   * Inicializa todos los sistemas del juego y configura el entorno gráfico
   */
  public Juego() {
      // Configurar entorno gráfico
      this.entorno = new Entorno(this, "La Invasión de los Zombies Grinch", ANCHO_VENTANA,
ALTO_VENTANA);

      // Inicializar sistemas en orden
      calcularDimensionesCesped();
      inicializarSistemaEstados();
      inicializarSistemaNiveles();
      inicializarSistemaTiempo();
      inicializarSistemaCesped();
      inicializarSistemaObjetivos();
      inicializarSistemaEnemigos();
      inicializarSistemaDisparos();
      inicializarSistemaPlantas();
      inicializarSistemaSeleccion();
      cargarImagenesMenu(); // Cargar imágenes del menú

      // Iniciar bucle principal del juego
```

```
    this.entorno.iniciar();
}

// ============================================================================
// MÉTODOS DE INICIALIZACIÓN
// ============================================================================

/**
 * Configura el estado inicial y sistema de menús
 */
private void inicializarSistemaEstados() {
    this.estadoActual = EstadoJuego.MENU_PRINCIPAL;
    this.opcionMenuSeleccionada = 0;
    this.juegoPausado = false;
}

/**
 * Configura el sistema de niveles y progresión
 */
private void inicializarSistemaNiveles() {
    this.nivelActual = 1;
    this.zombiesParaSiguienteNivel = ZOMBIES_NIVEL_1;
    this.nivelCompletado = false;
    this.jefeAparecio = false;
}

/**
 * Configura el sistema de control de tiempo
 */
private void inicializarSistemaTiempo() {
    this.tiempoInicio = System.currentTimeMillis();
    this.tiempoTranscurrido = 0;
    this.tiempoDetenido = false;
}

/**
 * Calcula las dimensiones del césped para ocupar toda la pantalla
 */
private void calcularDimensionesCesped() {
    // Margen superior para la barra de información
    int margenSuperior = 120;
    int margenInferior = 50;
    int margenLateral = 20;

    // Área disponible para el césped
    int anchoDisponible = ANCHO_VENTANA - (2 * margenLateral);
    int altoDisponible = ALTO_VENTANA - margenSuperior - margenInferior;

    // Calcular tamaño de casilla basado en la dimensión más restrictiva
    int tamCasillaPorAncho = anchoDisponible / COLUMNAS_CESPED;
    int tamCasillaPorAlto = altoDisponible / FILAS_CESPED;
```

```
    // Usar el tamaño más pequeño para mantener proporción
    this.TAMANIO_CASILLA = Math.min(tamCasillaPorAncho, tamCasillaPorAlto);

    // Calcular posiciones iniciales para centrar
    int anchoTotalCesped = COLUMNAS_CESPED * TAMANIO_CASILLA;
    int altoTotalCesped = FILAS_CESPED * TAMANIO_CASILLA;

    this.CESPED_X_INICIO = margenLateral + (anchoDisponible - anchoTotalCesped) / 2 +
TAMANIO_CASILLA / 2;
    this.CESPED_Y_INICIO = margenSuperior + (altoDisponible - altoTotalCesped) / 2 +
TAMANIO_CASILLA / 2;
  }

  /**
   * Inicializa la matriz del césped (tablero de juego)
   */
  private void inicializarSistemaCesped() {
    this.cesped = new Planta[FILAS_CESPED][COLUMNAS_CESPED];
    for (int fila = 0; fila < FILAS_CESPED; fila++) {
      for (int columna = 0; columna < COLUMNAS_CESPED; columna++) {
        cesped[fila][columna] = null;
      }
    }
  }

  /**
   * Configura los regalos que deben ser protegidos
   */
  private void inicializarSistemaObjetivos() {
    this.regalos = new Regalo[FILAS_CESPED];
    for (int i = 0; i < FILAS_CESPED; i++) {
      int x = CESPED_X_INICIO; // Columna 0
      int y = CESPED_Y_INICIO + i * TAMANIO_CASILLA;
      regalos[i] = new Regalo(x, y);
    }
  }

  /**
   * Inicializa arrays para enemigos y sistema de generación
   */
  private void inicializarSistemaEnemigos() {
    this.zombies = new Object[25];
    this.zombiesEliminados = 0;
    this.zombiesRestantes = zombiesParaSiguienteNivel;
    this.tickGeneracionZombie = 0;
    this.jefe = null;
  }

  /**
   * Configura arrays para proyectiles y disparos
   */
  private void inicializarSistemaDisparos() {
```

```
    this.bolasFuego = new BolaDeFuego[100];
    this.bolasHielo = new BolaDeHielo[50];
}

/**
 * Carga recursos de plantas y configura tiempos de recarga
 */
private void inicializarSistemaPlantas() {
    // Cargar imágenes de avatares
    cargarImagenesPlantas();

    // Inicializar tiempos de recarga
    this.tiempoCargaRoseBlade = 0;
    this.tiempoCargaWallNut = 0;
    this.tiempoCargaIceFlower = 0;
}

/**
 * Carga imágenes para el menú principal
 */
private void cargarImagenesMenu() {
    try {
        this.imagenFondoMenu = Herramientas.cargarImagen("img/fondo_menu.png");
        System.out.println("Imagen de fondo del menú cargada correctamente");
    } catch (Exception e) {
        System.out.println("No se pudo cargar fondo_menu.jpg: " + e.getMessage());
        System.out.println("Usando fondo de color por defecto");
        this.imagenFondoMenu = null;
    }
}

/**
 * Carga las imágenes de las plantas desde archivos (avatares)
 */
private void cargarImagenesPlantas() {
    try {
        this.avatarRoseBlade = Herramientas.cargarImagen("img/rose_blade.png");
        this.avatarWallNut = Herramientas.cargarImagen("img/nuez.png");
        this.avatarIceFlower = Herramientas.cargarImagen("img/ice_flower.png");
    } catch (Exception e) {
        System.out.println("Error al cargar imágenes de plantas: " + e.getMessage());
        // Usar placeholders geométricos si falla la carga
        this.avatarRoseBlade = null;
        this.avatarWallNut = null;
        this.avatarIceFlower = null;
    }
}

/**
 * Configura el sistema de selección y arrastre
 */
private void inicializarSistemaSeleccion() {
```

```
    this.plantaSeleccionada = null;
    this.arrastrando = false;
    this.plantaSeleccionadaParaMover = null;
    this.plantaSeleccionadaConTeclado = false;
    this.filaPlantaSeleccionada = -1;
    this.columnaPlantaSeleccionada = -1;
}

// ========================================================================
// BUCLE PRINCIPAL DEL JUEGO
// ========================================================================

/**
 * Método principal llamado en cada frame del juego
 * Delega a diferentes sistemas según el estado actual
 */
public void tick() {
    switch (estadoActual) {
        case MENU_PRINCIPAL:
            ejecutarMenuPrincipal();
            break;

        case INSTRUCCIONES:
            ejecutarPantallaInstrucciones();
            break;

        case JUGANDO:
            ejecutarJuegoPrincipal();
            break;

        case GANO:
            ejecutarPantallaVictoria();
            break;

        case PERDIO:
            ejecutarPantallaGameOver();
            break;
    }
}

// ========================================================================
// SISTEMA DE MENÚ PRINCIPAL - CON FONDO DE IMAGEN
// ========================================================================

/**
 * Ejecuta la lógica completa del menú principal
 */
private void ejecutarMenuPrincipal() {
    dibujarMenuPrincipal();
    procesarInputMenu();
}
```

```java
/**
 * Dibuja todos los elementos del menú principal
 */
private void dibujarMenuPrincipal() {
    dibujarFondoMenu();
    dibujarTituloJuego();
    dibujarOpcionesMenu();
    dibujarElementosDecorativos();
    dibujarInstruccionesNavegacion();
}

/**
 * Dibuja el fondo del menú principal con imagen
 */
private void dibujarFondoMenu() {
    if (imagenFondoMenu != null) {
        // Fondo con imagen
        entorno.dibujarImagen(imagenFondoMenu, 400, 300, 0, 1.0);
        // Capa semitransparente para mejor legibilidad
        entorno.dibujarRectangulo(400, 300, 800, 600, 0, new Color(0, 0, 0, 100));
    } else {
        // Fondo de color sólido
        entorno.dibujarRectangulo(400, 300, 800, 600, 0, new Color(20, 60, 20));
    }
}

/**
 * Dibuja el título principal del juego
 */
private void dibujarTituloJuego() {
    entorno.cambiarFont("Arial", 48, Color.GREEN);
    entorno.escribirTexto("ZOMBIE GRINCH WARS", 150, 120);

    // Línea decorativa bajo el título
    entorno.dibujarRectangulo(400, 145, 450, 3, 0, new Color(0, 200, 0, 150));
}

/**
 * Dibuja las opciones seleccionables del menú
 */
private void dibujarOpcionesMenu() {
    for (int i = 0; i < OPCIONES_MENU.length; i++) {
        boolean seleccionada = (opcionMenuSeleccionada == i);
        dibujarOpcionMenu(OPCIONES_MENU[i], POSICIONES_Y_MENU[i], seleccionada);
    }
}

/**
 * Dibuja una opción individual del menú
 */
private void dibujarOpcionMenu(String texto, int posY, boolean seleccionada) {
    // Configurar estilo según si está seleccionada
```

```
        if (seleccionada) {
            entorno.cambiarFont("Arial", 32, Color.YELLOW);
            // Efectos visuales para opción seleccionada
            entorno.dibujarRectangulo(400, posY + 15, 180, 3, 0, Color.YELLOW);
            entorno.dibujarCirculo(300, posY, 8, Color.YELLOW);
            entorno.dibujarCirculo(500, posY, 8, Color.YELLOW);
        } else {
            entorno.cambiarFont("Arial", 28, Color.WHITE);
        }

        entorno.escribirTexto(texto, 400, posY);
    }

    /**
     * Dibuja elementos decorativos en el menú
     */
    private void dibujarElementosDecorativos() {
        dibujarZombieDecorativo(100, 200);
        dibujarZombieDecorativo(700, 400);
        dibujarEfectosEstrellas();
    }

    /**
     * Dibuja instrucciones de control en la parte inferior
     */
    private void dibujarInstruccionesNavegacion() {
        entorno.cambiarFont("Arial", 16, new Color(200, 200, 200, 180));
        entorno.escribirTexto("Usa las flechas ↑↓ para navegar • ENTER para seleccionar", 400, 520);
    }

    /**
     * Procesa toda la entrada del usuario en el menú
     */
    private void procesarInputMenu() {
        procesarNavegacionMenu();
        procesarSeleccionMenu();
        procesarSalidaMenu();
    }

    /**
     * Maneja la navegación entre opciones del menú
     */
    private void procesarNavegacionMenu() {
        // Movimiento hacia abajo
        if (entorno.sePresiono(entorno.TECLA_ABAJO) || entorno.sePresiono('s')) {
            opcionMenuSeleccionada = (opcionMenuSeleccionada + 1) % OPCIONES_MENU.length;
        }

        // Movimiento hacia arriba
        if (entorno.sePresiono(entorno.TECLA_ARRIBA) || entorno.sePresiono('w')) {
            opcionMenuSeleccionada = (opcionMenuSeleccionada - 1 + OPCIONES_MENU.length) %
OPCIONES_MENU.length;
```

```
    }
}

/**
 * Maneja la selección de opciones del menú
 */
private void procesarSeleccionMenu() {
    if (entorno.sePresiono(entorno.TECLA_ENTER)) {
        switch (opcionMenuSeleccionada) {
            case 0: // JUGAR
                estadoActual = EstadoJuego.JUGANDO;
                reiniciarJuegoCompleto();
                break;

            case 1: // INSTRUCCIONES
                estadoActual = EstadoJuego.INSTRUCCIONES;
                break;

            case 2: // SALIR
                System.exit(0);
                break;
        }
    }
}

/**
 * Maneja la salida del juego desde el menú
 */
private void procesarSalidaMenu() {
    if (entorno.sePresiono(entorno.TECLA_ESCAPE)) {
        System.exit(0);
    }
}

// ========================================================================
// SISTEMA DE PANTALLA DE INSTRUCCIONES
// ========================================================================

/**
 * Ejecuta la pantalla de instrucciones
 */
private void ejecutarPantallaInstrucciones() {
    dibujarPantallaInstrucciones();
    procesarInputInstrucciones();
}

/**
 * Dibuja la pantalla de instrucciones
 */
private void dibujarPantallaInstrucciones() {
    // Fondo
    entorno.dibujarRectangulo(400, 300, 800, 600, 0, new Color(20, 40, 80));
```

```
    // Título
    entorno.cambiarFont("Arial", 36, Color.CYAN);
    entorno.escribirTexto("INSTRUCCIONES", 400, 80);

    // Lista de instrucciones
    entorno.cambiarFont("Arial", 18, Color.WHITE);
    String[] instrucciones = {
        "• Coloca plantas haciendo clic en la barra superior",
        "• Usa WASD o flechas para mover plantas seleccionadas",
        "• Protege los regalos de los zombies",
        "• RoseBlade: Dispara fuego • IceFlower: Ralentiza",
        "• Super zombies son inmunes a congelación completa",
        "• Derrota al jefe en el nivel 4 para ganar"
    };

    for (int i = 0; i < instrucciones.length; i++) {
        entorno.escribirTexto(instrucciones[i], 400, 150 + i * 30);
    }

    // Instrucción para volver
    entorno.cambiarFont("Arial", 24, Color.YELLOW);
    entorno.escribirTexto("Presiona ESC para volver al menú", 400, 450);
}

/**
 * Procesa input en pantalla de instrucciones
 */
private void procesarInputInstrucciones() {
    if (entorno.sePresiono(entorno.TECLA_ESCAPE)) {
        estadoActual = EstadoJuego.MENU_PRINCIPAL;
    }
}

// ========================================================================
// SISTEMA PRINCIPAL DEL JUEGO
// ========================================================================

/**
 * Ejecuta un frame completo del juego principal
 */
private void ejecutarJuegoPrincipal() {
    actualizarSistemas();
    dibujarElementosJuego();
    procesarInputJuego();
    verificarCondicionesJuego();
}

/**
 * Actualiza todos los sistemas del juego
 */
private void actualizarSistemas() {
```

```
    if (!tiempoDetenido) {
        tiempoTranscurrido = System.currentTimeMillis() - tiempoInicio;
    }

    actualizarTiemposRecarga();
    actualizarPlantas();
    actualizarEnemigos();
    actualizarProyectiles();
    generarNuevosZombies();
    manejarDisparos();
    manejarColisiones();
    limpiarElementosMuertos();
}

/**
 * Dibuja todos los elementos visuales del juego
 */
private void dibujarElementosJuego() {
    dibujarFondo();
    dibujarCesped();
    dibujarRegalos();
    dibujarPlantas();
    dibujarBarraSuperior();
    dibujarEnemigos();
    dibujarProyectiles();
    dibujarElementosSeleccion();
}

/**
 * Procesa la entrada del usuario durante el juego
 */
private void procesarInputJuego() {
    manejarSeleccionYArrastre();
    manejarSeleccionConTeclado();
    manejarMovimientoConTeclado();

    // Volver al menú principal
    if (entorno.sePresiono(entorno.TECLA_ESCAPE)) {
        estadoActual = EstadoJuego.MENU_PRINCIPAL;
    }
}

/**
 * Verifica condiciones de victoria/derrota
 */
private void verificarCondicionesJuego() {
    verificarDerrota();
    verificarAvanceNivel();
    verificarVictoria();
}

// ========================================================================
```

```java
// SISTEMA DE CÉSPED
// ========================================================================

/**
 * Calcula la posición X correcta para una columna del césped
 */
private int calcularPosicionXCesped(int columna) {
    return CESPED_X_INICIO + columna * TAMANIO_CASILLA;
}

/**
 * Calcula la posición Y correcta para una fila del césped
 */
private int calcularPosicionYCesped(int fila) {
    return CESPED_Y_INICIO + fila * TAMANIO_CASILLA;
}

/**
 * Dibuja la cuadrícula del césped CORREGIDA
 */
/**
 * Dibuja la cuadrícula del césped ocupando toda la pantalla
 */
private void dibujarCesped() {
    for (int fila = 0; fila < FILAS_CESPED; fila++) {
        for (int columna = 0; columna < COLUMNAS_CESPED; columna++) {
            int x = CESPED_X_INICIO + columna * TAMANIO_CASILLA;
            int y = CESPED_Y_INICIO + fila * TAMANIO_CASILLA;

            // Color de fondo
            Color colorCesped = (columna == 0) ? new Color(255, 200, 200, 150) : new Color(200, 255,
200, 150);

            // Fondo semitransparente
            entorno.dibujarRectangulo(x, y, TAMANIO_CASILLA, TAMANIO_CASILLA, 0, colorCesped);

            // Bordes con líneas
            int medio = TAMANIO_CASILLA / 2;
            entorno.dibujarRectangulo(x, y - medio, TAMANIO_CASILLA, 1, 0, Color.BLACK);
            entorno.dibujarRectangulo(x, y + medio, TAMANIO_CASILLA, 1, 0, Color.BLACK);
            entorno.dibujarRectangulo(x - medio, y, 1, TAMANIO_CASILLA, 0, Color.BLACK);
            entorno.dibujarRectangulo(x + medio, y, 1, TAMANIO_CASILLA, 0, Color.BLACK);

            // Etiquetar filas
            if (columna == 0) {
                entorno.cambiarFont("Arial", 12, Color.BLACK);
                entorno.escribirTexto("F" + fila, x - 20, y);
            }
        }
    }
}
```

```java
/**
 * Dibuja los regalos (objetivos a proteger)
 */
private void dibujarRegalos() {
    for (Regalo regalo : regalos) {
        regalo.dibujar(entorno);
    }
}

// ========================================================================
// SISTEMA DE ACTUALIZACIÓN DEL JUEGO
// ========================================================================

/**
 * Actualiza los tiempos de recarga de las plantas
 */
private void actualizarTiemposRecarga() {
    if (tiempoCargaRoseBlade > 0) tiempoCargaRoseBlade--;
    if (tiempoCargaWallNut > 0) tiempoCargaWallNut--;
    if (nivelActual >= 2 && tiempoCargaIceFlower > 0) tiempoCargaIceFlower--;
}

/**
 * Actualiza el estado de todas las plantas
 */
private void actualizarPlantas() {
    for (int fila = 0; fila < FILAS_CESPED; fila++) {
        for (int columna = 0; columna < COLUMNAS_CESPED; columna++) {
            if (cesped[fila][columna] != null) {
                if (cesped[fila][columna] instanceof RoseBlade) {
                    ((RoseBlade) cesped[fila][columna]).actualizar();
                } else if (cesped[fila][columna] instanceof IceFlower) {
                    ((IceFlower) cesped[fila][columna]).actualizar();
                }
            }
        }
    }
}

/**
 * Actualiza posición y estado de todos los enemigos
 */
private void actualizarEnemigos() {
    for (int i = 0; i < zombies.length; i++) {
        if (zombies[i] != null) {
            if (zombies[i] instanceof ZombieGrinch) {
                ZombieGrinch zombie = (ZombieGrinch) zombies[i];
                actualizarZombieNormal(zombie, i);
            } else if (zombies[i] instanceof SuperZombieGrinch) {
                SuperZombieGrinch superZombie = (SuperZombieGrinch) zombies[i];
                actualizarSuperZombie(superZombie, i);
            }
```

```
        }
    }

    actualizarJefe();
}

/**
 * Actualiza un zombie normal
 */
private void actualizarZombieNormal(ZombieGrinch zombie, int indice) {
    if (zombie.estaMuerto()) {
        zombies[indice] = null;
        zombiesEliminados++;
        zombiesRestantes--;
    } else {
        zombie.mover(cesped);
        zombie.atacar();
    }
}

/**
 * Actualiza un super zombie
 */
private void actualizarSuperZombie(SuperZombieGrinch superZombie, int indice) {
    if (superZombie.estaMuerto()) {
        zombies[indice] = null;
        zombiesEliminados++;
        zombiesRestantes--;
    } else {
        superZombie.mover(cesped);
        superZombie.atacar();
    }
}

/**
 * Actualiza el jefe final
 */
private void actualizarJefe() {
    if (jefe != null) {
        if (jefe.estaMuerto()) {
            jefe = null;
            zombiesEliminados += 10;
            zombiesRestantes = Math.max(0, zombiesRestantes - 10);
        } else {
            jefe.actualizar();
            jefe.mover();

            // Ataque especial del jefe
            if (jefe.isAtaqueActivo()) {
                jefe.ejecutarAtaque(cesped);
            }
        }
```

```
    }
}

/**
 * Actualiza posición de todos los proyectiles
 */
private void actualizarProyectiles() {
    actualizarBolasFuego();
    actualizarBolasHielo();
}

/**
 * Actualiza las bolas de fuego
 */
private void actualizarBolasFuego() {
    for (int i = 0; i < bolasFuego.length; i++) {
        if (bolasFuego[i] != null) {
            if (bolasFuego[i].estaFueraDePantalla(ANCHO_VENTANA)) {
                bolasFuego[i] = null;
            } else {
                bolasFuego[i].mover();
            }
        }
    }
}

/**
 * Actualiza las bolas de hielo
 */
private void actualizarBolasHielo() {
    for (int i = 0; i < bolasHielo.length; i++) {
        if (bolasHielo[i] != null) {
            if (bolasHielo[i].estaFueraDePantalla(ANCHO_VENTANA)) {
                bolasHielo[i] = null;
            } else {
                bolasHielo[i].mover();
            }
        }
    }
}

/**
 * Genera nuevos zombies según la progresión del nivel
 */
private void generarNuevosZombies() {
    tickGeneracionZombie++;

    int frecuencia = obtenerFrecuenciaGeneracion();

    if (tickGeneracionZombie >= frecuencia && zombiesRestantes > 0) {
        for (int i = 0; i < zombies.length; i++) {
            if (zombies[i] == null) {
```

```
                zombies[i] = crearZombieSegunNivel();
                if (zombies[i] != null) {
                    tickGeneracionZombie = 0;
                }
                break;
            }
        }
    }

    // Generar jefe en nivel 4
    if (nivelActual == 4 && !jefeAparecio && zombiesEliminados >= 25) {
        generarJefe();
    }
}

/**
 * Obtiene la frecuencia de generación según el nivel
 */
private int obtenerFrecuenciaGeneracion() {
    switch (nivelActual) {
        case 1: return 180;
        case 2: return 150;
        case 3: return 120;
        case 4: return 200;
        default: return 180;
    }
}

/**
 * Crea un zombie según las probabilidades del nivel actual
 */
private Object crearZombieSegunNivel() {
    int filaAleatoria = (int)(Math.random() * FILAS_CESPED);
    int y = calcularPosicionYCesped(filaAleatoria); // CÁLCULO DINÁMICO

    switch (nivelActual) {
        case 1:
            return new ZombieGrinch(850, y, filaAleatoria);
        case 2:
            if (Math.random() < 0.7) {
                return new ZombieGrinch(850, y, filaAleatoria);
            } else {
                return new SuperZombieGrinch(850, y, filaAleatoria);
            }
        case 3:
            return new SuperZombieGrinch(850, y, filaAleatoria);
        case 4:
            if (Math.random() < 0.4) {
                return new ZombieGrinch(850, y, filaAleatoria);
            } else if (Math.random() < 0.8) {
                return new SuperZombieGrinch(850, y, filaAleatoria);
            }
```

```
            return null;
        default:
            return new ZombieGrinch(850, y, filaAleatoria);
    }
}

/**
 * Genera el jefe final
 */
private void generarJefe() {
    if (jefe == null) {
        int y = calcularPosicionYCesped(2); // USA CÁLCULO DINÁMICO
        jefe = new ZombieBoss(850, y);
        jefeAparecio = true;
    }
}
// ========================================================================
// SISTEMA DE DISPAROS Y COLISIONES
// ========================================================================

/**
 * Maneja la creación de disparos de todas las plantas
 */
private void manejarDisparos() {
    for (int fila = 0; fila < FILAS_CESPED; fila++) {
        for (int columna = 0; columna < COLUMNAS_CESPED; columna++) {
            if (cesped[fila][columna] != null && cesped[fila][columna].estaViva()) {
                manejarDisparosRoseBlade(fila, columna);
                manejarDisparosIceFlower(fila, columna);
            }
        }
    }
}

/**
 * Maneja los disparos de RoseBlade (bolas de fuego)
 */
private void manejarDisparosRoseBlade(int fila, int columna) {
    if (cesped[fila][columna] instanceof RoseBlade) {
        RoseBlade roseBlade = (RoseBlade) cesped[fila][columna];

        if (roseBlade.puedeDisparar() &&
            roseBlade.hayZombiesEnFilaYDelante(zombies, fila, cesped[fila][columna].getX())) {

            crearBolaFuego(cesped[fila][columna].getX() + 30, cesped[fila][columna].getY());
            roseBlade.reiniciarRecarga();
        }
    }
}

/**
 * Maneja los disparos de IceFlower (bolas de hielo)
```

```
 */
private void manejarDisparosIceFlower(int fila, int columna) {
    if (cesped[fila][columna] instanceof IceFlower) {
        IceFlower iceFlower = (IceFlower) cesped[fila][columna];

        if (iceFlower.puedeDisparar() &&
            iceFlower.hayZombiesEnFilaYDelante(zombies, fila, cesped[fila][columna].getX())) {

            crearBolaHielo(cesped[fila][columna].getX() + 30, cesped[fila][columna].getY());
            iceFlower.reiniciarRecarga();
        }
    }
}

/**
 * Crea una nueva bola de fuego
 */
private void crearBolaFuego(double x, double y) {
    for (int i = 0; i < bolasFuego.length; i++) {
        if (bolasFuego[i] == null) {
            bolasFuego[i] = new BolaDeFuego(x, y);
            break;
        }
    }
}

/**
 * Crea una nueva bola de hielo
 */
private void crearBolaHielo(double x, double y) {
    for (int i = 0; i < bolasHielo.length; i++) {
        if (bolasHielo[i] == null) {
            bolasHielo[i] = new BolaDeHielo(x, y);
            break;
        }
    }
}

/**
 * Maneja todas las colisiones entre proyectiles y enemigos
 */
private void manejarColisiones() {
    manejarColisionesFuego();
    manejarColisionesHielo();
}

/**
 * Maneja colisiones de bolas de fuego
 */
private void manejarColisionesFuego() {
    for (int i = 0; i < bolasFuego.length; i++) {
        if (bolasFuego[i] != null) {
```

```
            // Colisión con zombies normales
            boolean colisiono = verificarColisionFuegoConZombies(i);

            // Colisión con jefe (si no colisionó con zombies)
            if (!colisiono && jefe != null) {
                verificarColisionFuegoConJefe(i);
            }
        }
    }
}

/**
 * Verifica colisión de bola de fuego con zombies
 */
private boolean verificarColisionFuegoConZombies(int indiceBola) {
    for (int j = 0; j < zombies.length; j++) {
        if (zombies[j] != null) {
            double distanciaX = Math.abs(bolasFuego[indiceBola].getX() - obtenerZombieX(zombies[j]));
            double distanciaY = Math.abs(bolasFuego[indiceBola].getY() - obtenerZombieY(zombies[j]));

            if (distanciaX < 30 && distanciaY < 40) {
                aplicarDanoZombie(zombies[j], 1);
                bolasFuego[indiceBola] = null;
                return true;
            }
        }
    }
    return false;
}

/**
 * Verifica colisión de bola de fuego con jefe
 */
private void verificarColisionFuegoConJefe(int indiceBola) {
    double distanciaX = Math.abs(bolasFuego[indiceBola].getX() - jefe.getX());
    double distanciaY = Math.abs(bolasFuego[indiceBola].getY() - jefe.getY());

    if (distanciaX < 50 && distanciaY < 100) {
        jefe.recibirDano(1);
        bolasFuego[indiceBola] = null;
    }
}

/**
 * Maneja colisiones de bolas de hielo
 */
private void manejarColisionesHielo() {
    for (int i = 0; i < bolasHielo.length; i++) {
        if (bolasHielo[i] != null && bolasHielo[i].isActiva()) {
            boolean colisiono = verificarColisionHieloConZombies(i);

            if (!colisiono && jefe != null) {
```

```
                verificarColisionHieloConJefe(i);
            }
        }
    }
}

/**
 * Verifica colisión de bola de hielo con zombies
 */
private boolean verificarColisionHieloConZombies(int indiceBola) {
    for (int j = 0; j < zombies.length; j++) {
        if (zombies[j] != null) {
            double distanciaX = Math.abs(bolasHielo[indiceBola].getX() - obtenerZombieX(zombies[j]));
            double distanciaY = Math.abs(bolasHielo[indiceBola].getY() - obtenerZombieY(zombies[j]));

            if (distanciaX < 30 && distanciaY < 40) {
                aplicarEfectoHielo(zombies[j], bolasHielo[indiceBola]);
                return true;
            }
        }
    }
    return false;
}

/**
 * Verifica colisión de bola de hielo con jefe
 */
private void verificarColisionHieloConJefe(int indiceBola) {
    double distanciaX = Math.abs(bolasHielo[indiceBola].getX() - jefe.getX());
    double distanciaY = Math.abs(bolasHielo[indiceBola].getY() - jefe.getY());

    if (distanciaX < 50 && distanciaY < 100) {
        double factorRalentizacion = 1.0 - (bolasHielo[indiceBola].getHits() * 0.2);
        jefe.aplicarRalentizacion(factorRalentizacion);
        jefe.recibirDano(1);
        bolasHielo[indiceBola].desactivar();
    }
}

/**
 * Aplica efecto de hielo a un zombie según su tipo
 */
private void aplicarEfectoHielo(Object zombie, BolaDeHielo bolaHielo) {
    if (zombie instanceof SuperZombieGrinch) {
        aplicarEfectoHieloSuperZombie((SuperZombieGrinch) zombie, bolaHielo);
    } else if (zombie instanceof ZombieGrinch) {
        aplicarEfectoHieloZombieNormal((ZombieGrinch) zombie, bolaHielo);
    }
}

/**
 * Aplica efecto de hielo a SuperZombie (solo ralentización)
```

```
     */
    private void aplicarEfectoHieloSuperZombie(SuperZombieGrinch superZombie, BolaDeHielo
bolaHielo) {
        double factorRalentizacion = 1.0 - (bolaHielo.getHits() * 0.15);
        superZombie.aplicarRalentizacion(factorRalentizacion);
        aplicarDanoZombie(superZombie, 1);
        bolaHielo.desactivar();
    }

    /**
     * Aplica efecto de hielo a Zombie normal (ralentización + posible congelación)
     */
    private void aplicarEfectoHieloZombieNormal(ZombieGrinch zombie, BolaDeHielo bolaHielo) {
        aplicarEfectoRalentizacion(zombie, bolaHielo.getHits());
        aplicarDanoZombie(zombie, 1);
        bolaHielo.incrementarHits();

        // Congelar completamente después de 3 hits
        if (bolaHielo.getHits() >= 3) {
            aplicarCongelacionCompleta(zombie);
            bolaHielo.desactivar();
        }
    }

    /**
     * Aplica efecto de ralentización a zombie normal
     */
    private void aplicarEfectoRalentizacion(ZombieGrinch zombie, int hits) {
        double factorRalentizacion = 1.0 - (hits * 0.2);
        factorRalentizacion = Math.max(0.3, factorRalentizacion);
        zombie.setVelocidad(0.5 * factorRalentizacion);
    }

    /**
     * Aplica daño a un zombie según su tipo
     */
    private void aplicarDanoZombie(Object zombie, int dano) {
        if (zombie instanceof ZombieGrinch) {
            ZombieGrinch z = (ZombieGrinch) zombie;
            for (int i = 0; i < dano; i++) {
                z.recibirDano();
            }
        } else if (zombie instanceof SuperZombieGrinch) {
            SuperZombieGrinch sz = (SuperZombieGrinch) zombie;
            for (int i = 0; i < dano; i++) {
                sz.recibirDano();
            }
        }
    }

    /**
     * Congela completamente un zombie normal
```

```
     */
    private void aplicarCongelacionCompleta(ZombieGrinch zombie) {
        zombie.setVelocidad(0);
    }

    /**
     * Obtiene la coordenada X de un zombie
     */
    private double obtenerZombieX(Object zombie) {
        if (zombie instanceof ZombieGrinch) {
            return ((ZombieGrinch) zombie).getX();
        } else if (zombie instanceof SuperZombieGrinch) {
            return ((SuperZombieGrinch) zombie).getX();
        }
        return -1000;
    }

    /**
     * Obtiene la coordenada Y de un zombie
     */
    private double obtenerZombieY(Object zombie) {
        if (zombie instanceof ZombieGrinch) {
            return ((ZombieGrinch) zombie).getY();
        } else if (zombie instanceof SuperZombieGrinch) {
            return ((SuperZombieGrinch) zombie).getY();
        }
        return -1000;
    }

    /**
     * Limpia elementos muertos o fuera de pantalla
     */
    private void limpiarElementosMuertos() {
        limpiarPlantasMuertas();
    }

    /**
     * Elimina plantas con salud agotada
     */
    private void limpiarPlantasMuertas() {
        for (int fila = 0; fila < FILAS_CESPED; fila++) {
            for (int columna = 0; columna < COLUMNAS_CESPED; columna++) {
                if (cesped[fila][columna] != null && cesped[fila][columna].getSalud() <= 0) {
                    // Si la planta seleccionada para mover murió, resetear selección
                    if (plantaSeleccionadaParaMover == cesped[fila][columna]) {
                        resetearSeleccionTeclado();
                    }
                    cesped[fila][columna] = null;
                }
            }
        }
    }
```

```java
// ========================================================================
// SISTEMA DE DIBUJO DEL JUEGO
// ========================================================================

/**
 * Dibuja el fondo según el nivel actual
 */
private void dibujarFondo() {
    Color colorFondo = obtenerColorFondoNivel();
    entorno.dibujarRectangulo(400, 300, 800, 600, 0, colorFondo);
}

/**
 * Obtiene el color de fondo según el nivel
 */
private Color obtenerColorFondoNivel() {
    switch (nivelActual) {
        case 1: return new Color(240, 255, 240);
        case 2: return new Color(240, 240, 255);
        case 3: return new Color(255, 240, 240);
        case 4: return new Color(255, 255, 200);
        default: return Color.WHITE;
    }
}

/**
 * Dibuja todas las plantas en el césped
 */
private void dibujarPlantas() {
    for (int fila = 0; fila < FILAS_CESPED; fila++) {
        for (int columna = 0; columna < COLUMNAS_CESPED; columna++) {
            if (cesped[fila][columna] != null) {
                // FORZAR a la planta a dibujarse en la posición correcta del césped
                int xCorrecto = CESPED_X_INICIO + columna * TAMANIO_CASILLA;
                int yCorrecto = CESPED_Y_INICIO + fila * TAMANIO_CASILLA;

                // Actualizar la posición de la planta antes de dibujar
                cesped[fila][columna].setX(xCorrecto);
                cesped[fila][columna].setY(yCorrecto);

                cesped[fila][columna].dibujar(entorno);
            }
        }
    }
}

/**
 * Dibuja todos los enemigos en pantalla
 */
private void dibujarEnemigos() {
    for (Object zombie : zombies) {
```

```
            if (zombie != null) {
                if (zombie instanceof ZombieGrinch) {
                    ((ZombieGrinch) zombie).dibujar(entorno);
                } else if (zombie instanceof SuperZombieGrinch) {
                    ((SuperZombieGrinch) zombie).dibujar(entorno);
                }
            }
        }

        if (jefe != null) {
            jefe.dibujar(entorno);
        }
    }

    /**
     * Dibuja todos los proyectiles en pantalla
     */
    private void dibujarProyectiles() {
        for (BolaDeFuego bolaFuego : bolasFuego) {
            if (bolaFuego != null) {
                bolaFuego.dibujar(entorno);
            }
        }

        for (BolaDeHielo bolaHielo : bolasHielo) {
            if (bolaHielo != null) {
                bolaHielo.dibujar(entorno);
            }
        }
    }

    /**
     * Dibuja elementos relacionados con selección y arrastre
     */
    private void dibujarElementosSeleccion() {
        dibujarPlantaArrastrada();
        dibujarPlantaSeleccionadaParaMover();
    }

    // ========================================================================
    // SISTEMA DE BARRA SUPERIOR (UI)
    // ========================================================================

    /**
     * Dibuja la barra superior con información del juego
     */
    private void dibujarBarraSuperior() {
        dibujarFondoBarraSuperior();
        dibujarSeccionNivel(100, 50);
        dibujarSeccionProgreso(250, 50);
        dibujarSeccionPlantas(450, 50);
        dibujarSeccionTiempo(650, 50);
```

```
    dibujarSeparadoresBarra();
}

/**
 * Dibuja el fondo de la barra superior
 */
private void dibujarFondoBarraSuperior() {
    entorno.dibujarRectangulo(400, 50, 800, 100, 0, new Color(40, 40, 60, 220));
    entorno.dibujarRectangulo(400, 0, 800, 5, 0, new Color(100, 150, 255));
}

/**
 * Dibuja la sección de información del nivel
 */
private void dibujarSeccionNivel(int x, int y) {
    // Icono de nivel
    entorno.dibujarRectangulo(x - 40, y, 30, 30, 0, new Color(70, 130, 180));
    entorno.cambiarFont("Arial", 16, Color.WHITE);
    entorno.escribirTexto("" + nivelActual, x - 40, y + 5);

    // Información textual
    entorno.cambiarFont("Arial", 14, Color.WHITE);
    entorno.escribirTexto("NIVEL", x, y - 15);

    entorno.cambiarFont("Arial", 16, Color.YELLOW);
    entorno.escribirTexto(obtenerNombreNivel(), x, y);

    entorno.cambiarFont("Arial", 12, Color.LIGHT_GRAY);
    entorno.escribirTexto(obtenerDescripcionNivel(), x, y + 15);
}

/**
 * Dibuja la sección de progreso de zombies
 */
private void dibujarSeccionProgreso(int x, int y) {
    // Icono de zombie
    entorno.dibujarCirculo(x - 40, y, 12, Color.GREEN);
    entorno.dibujarCirculo(x - 40, y - 3, 8, new Color(150, 250, 150));

    // Información textual
    entorno.cambiarFont("Arial", 14, Color.WHITE);
    entorno.escribirTexto("ELIMINADOS", x, y - 15);

    // Progreso numérico
    entorno.cambiarFont("Arial", 18, Color.CYAN);
    entorno.escribirTexto(zombiesEliminados + " / " + zombiesParaSiguienteNivel, x, y);

    // Barra de progreso visual
    dibujarBarraProgresoZombies(x, y);
}

/**
```

```
    * Dibuja la barra de progreso de zombies eliminados
     */
    private void dibujarBarraProgresoZombies(int x, int y) {
        double porcentaje = (double) zombiesEliminados / zombiesParaSiguienteNivel;
        int anchoBarra = 120;
        int progresoBarra = (int) (anchoBarra * Math.min(porcentaje, 1.0));

        // Fondo de la barra
        entorno.dibujarRectangulo(x, y + 15, anchoBarra, 8, 0, new Color(100, 100, 100));

        // Barra de progreso
        if (progresoBarra > 0) {
            Color colorProgreso = obtenerColorProgreso(porcentaje);
            entorno.dibujarRectangulo(x - anchoBarra/2 + progresoBarra/2, y + 15, progresoBarra, 8, 0,
colorProgreso);
        }

        // Porcentaje
        entorno.cambiarFont("Arial", 10, Color.WHITE);
        entorno.escribirTexto((int)(porcentaje * 100) + "%", x, y + 25);
    }

    /**
     * Obtiene el color de la barra de progreso según el porcentaje
     */
    private Color obtenerColorProgreso(double porcentaje) {
        if (porcentaje >= 1.0) return Color.GREEN;
        if (porcentaje >= 0.7) return Color.YELLOW;
        return Color.ORANGE;
    }

    /**
     * Dibuja la sección de plantas disponibles
     */
    private void dibujarSeccionPlantas(int x, int y) {
        entorno.cambiarFont("Arial", 14, Color.WHITE);
        entorno.escribirTexto("PLANTAS", x, y - 15);

        // Dibujar avatares de las plantas
        dibujarAvatarRoseBlade(x - 40, y);
        dibujarAvatarWallNut(x, y);
        dibujarAvatarIceFlower(x + 40, y);
    }

    /**
     * Dibuja el avatar de RoseBlade
     */
    private void dibujarAvatarRoseBlade(int x, int y) {
        boolean disponible = (tiempoCargaRoseBlade == 0);
        dibujarAvatarPlanta(x, y, avatarRoseBlade, tiempoCargaRoseBlade,
                            RECARGA_ROSE_BLADE, "Rose Blade", Color.RED, disponible);
    }
```

```java
/**
 * Dibuja el avatar de WallNut
 */
private void dibujarAvatarWallNut(int x, int y) {
    boolean disponible = (tiempoCargaWallNut == 0);
    dibujarAvatarPlanta(x, y, avatarWallNut, tiempoCargaWallNut,
                        RECARGA_WALL_NUT, "Wall Nut", new Color(160, 120, 60), disponible);
}

/**
 * Dibuja el avatar de IceFlower
 */
private void dibujarAvatarIceFlower(int x, int y) {
    if (nivelActual >= 2) {
        boolean disponible = (tiempoCargaIceFlower == 0);
        dibujarAvatarPlanta(x, y, avatarIceFlower, tiempoCargaIceFlower,
                            RECARGA_ICE_FLOWER, "Ice Flower", Color.CYAN, disponible);
    } else {
        dibujarAvatarPlantaBloqueada(x, y, "Ice Flower");
    }
}

/**
 * Dibuja un avatar de planta genérico
 */
private void dibujarAvatarPlanta(int x, int y, Image imagen, int tiempoCarga,
                                 int cargaMaxima, String nombre, Color colorBase, boolean disponible) {
    int tamañoAvatar = 35;

    // Fondo del avatar
    Color colorFondo = disponible ? new Color(255, 255, 255, 50) : new Color(0, 0, 0, 100);
    entorno.dibujarRectangulo(x, y, tamañoAvatar, tamañoAvatar, 0, colorFondo);

    // Borde según estado
    Color colorBorde = disponible ? Color.GREEN : Color.YELLOW;
    int grosorBorde = disponible ? 2 : 1;
    entorno.dibujarRectangulo(x, y, tamañoAvatar, tamañoAvatar, grosorBorde, colorBorde);

    // Imagen o placeholder
    if (imagen != null) {
        double escala = 0.22;
        entorno.dibujarImagen(imagen, x, y, 0, escala);

        // Oscurecer si está en recarga
        if (!disponible) {
            entorno.dibujarRectangulo(x, y, tamañoAvatar, tamañoAvatar, 0, new Color(0, 0, 0, 120));
        }
    } else {
        // Placeholder geométrico
        Color colorPlaceholder = disponible ? colorBase : Color.GRAY;
        entorno.dibujarRectangulo(x, y, tamañoAvatar-10, tamañoAvatar-10, 0, colorPlaceholder);
```

```
            entorno.cambiarFont("Arial", 10, Color.WHITE);
            String abreviatura = nombre.substring(0, 2);
            entorno.escribirTexto(abreviatura, x, y + 5);
        }

        // Indicador de estado (listo o en recarga)
        if (disponible) {
            entorno.cambiarFont("Arial", 14, Color.GREEN);
            entorno.escribirTexto("✓", x, y + 5);
        } else {
            dibujarBarraRecarga(x, y, tiempoCarga, cargaMaxima);
        }

        // Tooltip al pasar el mouse
        if (estaMouseSobreAvatar(x, y, tamañoAvatar) && disponible) {
            dibujarTooltipPlanta(x, y, nombre);
        }
    }

    /**
     * Dibuja la barra de recarga de una planta
     */
    private void dibujarBarraRecarga(int x, int y, int tiempoCarga, int cargaMaxima) {
        double progreso = 1.0 - (double)tiempoCarga / cargaMaxima;
        int anchoBarra = 25;

        // Barra de progreso
        entorno.dibujarRectangulo(x, y + 12, anchoBarra * progreso, 4, 0, Color.YELLOW);

        // Tiempo restante
        int segundos = (tiempoCarga / 60) + 1;
        entorno.cambiarFont("Arial", 9, Color.YELLOW);
        entorno.escribirTexto(segundos + "s", x, y + 8);
    }

    /**
     * Dibuja tooltip informativo para una planta
     */
    private void dibujarTooltipPlanta(int x, int y, String nombre) {
        entorno.dibujarRectangulo(x, y, 35, 35, 2, Color.CYAN);

        entorno.cambiarFont("Arial", 10, Color.WHITE);
        entorno.escribirTexto(nombre, x, y - 15);
        entorno.escribirTexto("LISTA", x, y - 25);
    }

    /**
     * Dibuja un avatar de planta bloqueada
     */
    private void dibujarAvatarPlantaBloqueada(int x, int y, String nombre) {
        int tamañoAvatar = 35;
```

```
    // Fondo gris para indicar bloqueado
    entorno.dibujarRectangulo(x, y, tamañoAvatar, tamañoAvatar, 0, Color.DARK_GRAY);
    entorno.dibujarRectangulo(x, y, tamañoAvatar, tamañoAvatar, 2, Color.GRAY);

    // Icono de candado
    entorno.cambiarFont("Arial", 16, Color.GRAY);
    entorno.escribirTexto("🔒", x, y + 5);

    // Texto del nivel requerido
    entorno.cambiarFont("Arial", 8, Color.LIGHT_GRAY);
    entorno.escribirTexto("Niv " + nivelActual, x, y + 15);

    // Tooltip informativo
    if (estaMouseSobreAvatar(x, y, tamañoAvatar)) {
        entorno.cambiarFont("Arial", 10, Color.WHITE);
        entorno.escribirTexto(nombre + " (Nivel " + (nivelActual + 1) + ")", x, y - 20);
    }
}

/**
 * Dibuja la sección de tiempo y estadísticas
 */
private void dibujarSeccionTiempo(int x, int y) {
    // Icono de reloj
    dibujarIconoReloj(x - 40, y);

    // Tiempo transcurrido
    entorno.cambiarFont("Arial", 14, Color.WHITE);
    entorno.escribirTexto("TIEMPO", x, y - 15);

    entorno.cambiarFont("Arial", 18, Color.WHITE);
    entorno.escribirTexto(obtenerTiempoSegundos() + "s", x, y);

    // Zombies restantes
    entorno.cambiarFont("Arial", 12, Color.LIGHT_GRAY);
    entorno.escribirTexto("Restantes: " + zombiesRestantes, x, y + 15);

    // Indicador de estado del juego
    if (tiempoDetenido) {
        entorno.cambiarFont("Arial", 10, Color.RED);
        entorno.escribirTexto("JUEGO PAUSADO", x, y + 25);
    }
}

/**
 * Dibuja el icono de reloj
 */
private void dibujarIconoReloj(int x, int y) {
    entorno.dibujarCirculo(x, y, 12, Color.ORANGE);
    entorno.dibujarCirculo(x, y, 8, Color.YELLOW);
    entorno.dibujarRectangulo(x, y - 8, 2, 6, 0, Color.BLACK);
```

```
        entorno.dibujarRectangulo(x, y, 5, 2, 0.5, Color.BLACK);
    }

    /**
     * Dibuja separadores entre secciones de la barra
     */
    private void dibujarSeparadoresBarra() {
        int[] separadoresX = {200, 350, 550};
        for (int x : separadoresX) {
            entorno.dibujarRectangulo(x, 10, 2, 80, 0, new Color(255, 255, 255, 80));
        }

        // Línea inferior decorativa
        entorno.dibujarRectangulo(400, 100, 800, 2, 0, new Color(100, 150, 255, 150));
    }

    // ========================================================================
    // SISTEMA DE SELECCIÓN Y MOVIMIENTO - CORREGIDO
    // ========================================================================

    /**
     * Maneja la selección y arrastre de plantas con el mouse - ✅ CORREGIDO
     */
    private void manejarSeleccionYArrastre() {
        int mouseX = entorno.mouseX();
        int mouseY = entorno.mouseY();

        if (!plantaSeleccionadaConTeclado && entorno.sePresionoBoton(entorno.BOTON_IZQUIERDO))
{
            manejarClicBarraPlantas(mouseX, mouseY);
            manejarClicCesped(mouseX, mouseY);
        }

        manejarArrastrePlanta(mouseX, mouseY);
        manejarSueltaPlanta(mouseX, mouseY);
    }

    /**
     * Maneja clics en la barra de plantas para seleccionar
     */
    private void manejarClicBarraPlantas(int mouseX, int mouseY) {
        if (mouseY <= 85 && mouseY >= 35) { // Área de la barra de plantas
            // Rose Blade - posición del avatar
            if (mouseX >= 410 && mouseX <= 445) {
                if (tiempoCargaRoseBlade == 0) {
                    seleccionarPlantaParaColocar("roseblade");
                }
            }
            // Wall Nut
            else if (mouseX >= 450 && mouseX <= 485) {
                if (tiempoCargaWallNut == 0) {
                    seleccionarPlantaParaColocar("wallnut");
```

```
            }
        }
        // Ice Flower (solo nivel 2+)
        else if (nivelActual >= 2 && mouseX >= 490 && mouseX <= 525) {
            if (tiempoCargaIceFlower == 0) {
                seleccionarPlantaParaColocar("iceflower");
            }
        }
    }
}

/**
 * Maneja clics en el césped para seleccionar plantas existentes
 */
private void manejarClicCesped(int mouseX, int mouseY) {
    if (mouseY > 100) {
        for (int fila = 0; fila < FILAS_CESPED; fila++) {
            for (int columna = 0; columna < COLUMNAS_CESPED; columna++) {
                if (cesped[fila][columna] != null) {
                    Planta planta = cesped[fila][columna];
                    // USA POSICIONES CALCULADAS PARA DETECCIÓN
                    int plantaX = CESPED_X_INICIO + columna * TAMANIO_CASILLA;
                    int plantaY = CESPED_Y_INICIO + fila * TAMANIO_CASILLA;

                    double distanciaX = Math.abs(mouseX - plantaX);
                    double distanciaY = Math.abs(mouseY - plantaY);

                    if (distanciaX < TAMANIO_CASILLA/2 && distanciaY < TAMANIO_CASILLA/2) {
                        seleccionarPlantaParaMover(planta, fila, columna);
                        return;
                    }
                }
            }
        }
        resetearSeleccionTeclado();
    }
}

/**
 * Selecciona una planta para colocar (arrastre)
 */
private void seleccionarPlantaParaColocar(String tipoPlanta) {
    plantaSeleccionada = tipoPlanta;
    arrastrando = true;
    resetearSeleccionTeclado();
}

/**
 * Selecciona una planta existente para mover
 */
private void seleccionarPlantaParaMover(Planta planta, int fila, int columna) {
    plantaSeleccionadaParaMover = planta;
```

```
    plantaSeleccionadaConTeclado = true;
    filaPlantaSeleccionada = fila;
    columnaPlantaSeleccionada = columna;
    plantaSeleccionada = null;
    arrastrando = false;
}

/**
 * Maneja el arrastre de una planta seleccionada
 */
private void manejarArrastrePlanta(int mouseX, int mouseY) {
    if (arrastrando && entorno.estaPresionado(entorno.BOTON_IZQUIERDO)) {
        plantaArrastradaX = mouseX;
        plantaArrastradaY = mouseY;
    }
}

/**
 * Maneja la suelta de una planta arrastrada
 */
private void manejarSueltaPlanta(int mouseX, int mouseY) {
    if (arrastrando && entorno.seLevantoBoton(entorno.BOTON_IZQUIERDO)) {
        arrastrando = false;
        colocarPlantaEnCesped(mouseX, mouseY);
        plantaSeleccionada = null;
    }
}

/**
 * Intenta colocar una planta en la posición del mouse
 */
private void colocarPlantaEnCesped(int mouseX, int mouseY) {
    int[] posicion = obtenerPosicionCesped(mouseX, mouseY);
    int fila = posicion[0];
    int columna = posicion[1];

    if (fila != -1 && columna != -1 && columna >= 1 && cesped[fila][columna] == null) {
        crearPlantaEnPosicion(fila, columna);
    }
}

/**
 * Obtiene la posición en el césped a partir de coordenadas de pantalla
 */
private int[] obtenerPosicionCesped(int mouseX, int mouseY) {
    int fila = -1;
    int columna = -1;

    // Encontrar fila
    for (int i = 0; i < FILAS_CESPED; i++) {
        int yCasilla = calcularPosicionYCesped(i);
        if (mouseY >= yCasilla - TAMANIO_CASILLA/2 &&
```

```
            mouseY <= yCasilla + TAMANIO_CASILLA/2) {
            fila = i;
            break;
        }
    }

    // Encontrar columna
    if (fila != -1) {
        for (int j = 0; j < COLUMNAS_CESPED; j++) {
            int xCasilla = calcularPosicionXCesped(j);
            if (mouseX >= xCasilla - TAMANIO_CASILLA/2 &&
                mouseX <= xCasilla + TAMANIO_CASILLA/2) {
                columna = j;
                break;
            }
        }

        // Validar que no sea la columna de regalos (columna 0)
        if (columna < 1) {
            fila = -1;
            columna = -1;
        }
    }

    return new int[]{fila, columna};
}

/**
 * Crea una nueva planta en la posición especificada
 */
private void crearPlantaEnPosicion(int fila, int columna) {
    int x = CESPED_X_INICIO + columna * TAMANIO_CASILLA;
    int y = CESPED_Y_INICIO + fila * TAMANIO_CASILLA;

    if ("roseblade".equals(plantaSeleccionada)) {
        cesped[fila][columna] = new RoseBlade(x, y);
        tiempoCargaRoseBlade = RECARGA_ROSE_BLADE;
    } else if ("wallnut".equals(plantaSeleccionada)) {
        cesped[fila][columna] = new WallNut(x, y);
        tiempoCargaWallNut = RECARGA_WALL_NUT;
    } else if ("iceflower".equals(plantaSeleccionada)) {
        cesped[fila][columna] = new IceFlower(x, y);
        tiempoCargaIceFlower = RECARGA_ICE_FLOWER;
    }
}

/**
 * Maneja la selección de plantas con teclado
 */
private void manejarSeleccionConTeclado() {
    if (entorno.sePresiono(entorno.TECLA_ESPACIO)) {
        if (!plantaSeleccionadaConTeclado) {
```

```
            seleccionarPrimeraPlantaDisponible();
        }
    }

    if (entorno.sePresiono(entorno.TECLA_ESCAPE)) {
        resetearSeleccionTeclado();
    }
}

/**
 * Selecciona la primera planta disponible en el césped
 */
private void seleccionarPrimeraPlantaDisponible() {
    for (int fila = 0; fila < FILAS_CESPED; fila++) {
        for (int columna = 0; columna < COLUMNAS_CESPED; columna++) {
            if (cesped[fila][columna] != null && cesped[fila][columna].estaViva()) {
                seleccionarPlantaParaMover(cesped[fila][columna], fila, columna);
                return;
            }
        }
    }
}

/**
 * Maneja el movimiento de plantas seleccionadas con teclado
 */
private void manejarMovimientoConTeclado() {
    if (plantaSeleccionadaConTeclado && plantaSeleccionadaParaMover != null) {
        if (entorno.sePresiono('a') || entorno.sePresiono(entorno.TECLA_IZQUIERDA)) {
            moverPlantaDireccion(-1, 0); // Izquierda
        }
        if (entorno.sePresiono('d') || entorno.sePresiono(entorno.TECLA_DERECHA)) {
            moverPlantaDireccion(1, 0); // Derecha
        }
        if (entorno.sePresiono('w') || entorno.sePresiono(entorno.TECLA_ARRIBA)) {
            moverPlantaDireccion(0, -1); // Arriba
        }
        if (entorno.sePresiono('s') || entorno.sePresiono(entorno.TECLA_ABAJO)) {
            moverPlantaDireccion(0, 1); // Abajo
        }
    }
}

/**
 * Mueve la planta seleccionada en una dirección específica
 */
private void moverPlantaDireccion(int deltaColumna, int deltaFila) {
    int nuevaFila = filaPlantaSeleccionada + deltaFila;
    int nuevaColumna = columnaPlantaSeleccionada + deltaColumna;

    if (esPosicionValida(nuevaFila, nuevaColumna)) {
        moverPlantaACasilla(nuevaFila, nuevaColumna);
```

```
    }
}

/**
 * Verifica si una posición del césped es válida para mover una planta
 */
private boolean esPosicionValida(int fila, int columna) {
    return fila >= 0 && fila < FILAS_CESPED &&
           columna >= 1 && columna < COLUMNAS_CESPED;
}

/**
 * Mueve la planta seleccionada a una nueva casilla
 */
private void moverPlantaACasilla(int nuevaFila, int nuevaColumna) {
    if (cesped[nuevaFila][nuevaColumna] == null) {
        // Calcular nuevas coordenadas CORRECTAS
        int nuevaX = calcularPosicionXCesped(nuevaColumna);
        int nuevaY = calcularPosicionYCesped(nuevaFila);

        // Actualizar posición de la planta
        plantaSeleccionadaParaMover.setX(nuevaX);
        plantaSeleccionadaParaMover.setY(nuevaY);

        // Actualizar matriz del césped
        cesped[filaPlantaSeleccionada][columnaPlantaSeleccionada] = null;
        cesped[nuevaFila][nuevaColumna] = plantaSeleccionadaParaMover;

        // Actualizar posición de selección
        filaPlantaSeleccionada = nuevaFila;
        columnaPlantaSeleccionada = nuevaColumna;
    }
}

/**
 * Resetea la selección por teclado
 */
private void resetearSeleccionTeclado() {
    plantaSeleccionadaParaMover = null;
    plantaSeleccionadaConTeclado = false;
    filaPlantaSeleccionada = -1;
    columnaPlantaSeleccionada = -1;
}

/**
 * Dibuja la planta que está siendo arrastrada
 */
private void dibujarPlantaArrastrada() {
    if (arrastrando && plantaSeleccionada != null) {
        Color color = obtenerColorPlantaArrastrada();
        entorno.dibujarRectangulo(plantaArrastradaX, plantaArrastradaY, 50, 50, 0, color);
```

```
            String nombre = obtenerNombrePlantaSeleccionada();
            entorno.escribirTexto("Arrastrando: " + nombre, plantaArrastradaX, plantaArrastradaY - 40);
        }
    }

    /**
     * Obtiene el color para la planta arrastrada
     */
    private Color obtenerColorPlantaArrastrada() {
        switch (plantaSeleccionada) {
            case "roseblade": return new Color(255, 0, 0, 128);
            case "wallnut": return new Color(139, 69, 19, 128);
            case "iceflower": return new Color(100, 200, 255, 128);
            default: return new Color(255, 255, 255, 128);
        }
    }

    /**
     * Obtiene el nombre de la planta seleccionada
     */
    private String obtenerNombrePlantaSeleccionada() {
        switch (plantaSeleccionada) {
            case "roseblade": return "Rose Blade";
            case "wallnut": return "Wall Nut";
            case "iceflower": return "Ice Flower";
            default: return "Desconocida";
        }
    }

    /**
     * Dibuja el indicador de planta seleccionada para mover
     */
    private void dibujarPlantaSeleccionadaParaMover() {
        if (plantaSeleccionadaConTeclado && plantaSeleccionadaParaMover != null) {
            // Resaltar la planta seleccionada
            entorno.dibujarRectangulo(
                plantaSeleccionadaParaMover.getX(),
                plantaSeleccionadaParaMover.getY(),
                55, 55, 0, Color.YELLOW
            );

            // Mostrar instrucciones de movimiento
            entorno.escribirTexto("Mover con WASD - Pos: F" + filaPlantaSeleccionada + " C" +
columnaPlantaSeleccionada,
                plantaSeleccionadaParaMover.getX(),
                plantaSeleccionadaParaMover.getY() - 40
            );
        }
    }

    // ========================================================================
    // SISTEMA DE VERIFICACIÓN DE CONDICIONES
```

```
    // ========================================================================

    /**
     * Verifica si el jugador perdió (zombies alcanzaron los regalos)
     */
    private void verificarDerrota() {
        if (zombiesAlcanzaronRegalos() || jefeAlcanzoRegalos()) {
            estadoActual = EstadoJuego.PERDIO;
            juegoPausado = true;
            tiempoDetenido = true;
        }
    }

    /**
     * Verifica si algún zombie alcanzó la columna de regalos
     */
    private boolean zombiesAlcanzaronRegalos() {
        for (Object zombie : zombies) {
            if (zombie != null && obtenerZombieX(zombie) < CESPED_X_INICIO + TAMANIO_CASILLA/2)
{
                return true;
            }
        }
        return false;
    }

    /**
     * Verifica si el jefe alcanzó los regalos
     */
    private boolean jefeAlcanzoRegalos() {
        return jefe != null && jefe.getX() < CESPED_X_INICIO + TAMANIO_CASILLA/2;
    }

    /**
     * Verifica si se puede avanzar al siguiente nivel
     */
    private void verificarAvanceNivel() {
        if (nivelActual < TOTAL_NIVELES &&
            zombiesEliminados >= zombiesParaSiguienteNivel &&
            !nivelCompletado) {

            nivelCompletado = true;
            nivelActual++;
            reiniciarParaNuevoNivel();
        }
    }

    /**
     * Verifica si se cumplieron las condiciones de victoria
     */
    private void verificarVictoria() {
        if (nivelActual == TOTAL_NIVELES && !nivelCompletado) {
```

```
        boolean jefeDerrotado = (jefe == null || jefe.estaMuerto());
        boolean zombiesCompletados = (zombiesEliminados >= 50);

        if (jefeDerrotado && zombiesCompletados) {
            estadoActual = EstadoJuego.GANO;
            juegoPausado = true;
            nivelCompletado = true;
            tiempoDetenido = true;
        }
    }
}

// ========================================================================
// SISTEMA DE PANTALLAS DE FINALIZACIÓN
// ========================================================================

/**
 * Ejecuta la pantalla de Game Over
 */
private void ejecutarPantallaGameOver() {
    dibujarPantallaGameOver();
    procesarInputPantallaFin();
}

/**
 * Ejecuta la pantalla de Victoria
 */
private void ejecutarPantallaVictoria() {
    dibujarPantallaVictoria();
    procesarInputPantallaFin();
}

/**
 * Dibuja la pantalla de Game Over
 */
private void dibujarPantallaGameOver() {
    dibujarFondoPantallaFin(new Color(0, 0, 0, 200));

    entorno.cambiarFont("Arial", 40, Color.RED);
    entorno.escribirTexto("GAME OVER", 400, 180);

    dibujarEstadisticasFinJuego();
    dibujarOpcionesReinicio();
}

/**
 * Dibuja la pantalla de Victoria
 */
private void dibujarPantallaVictoria() {
    dibujarFondoPantallaFin(new Color(0, 150, 0, 200));

    entorno.cambiarFont("Arial", 40, Color.GREEN);
```

```
    entorno.escribirTexto("¡VICTORIA TOTAL!", 400, 180);

    dibujarEstadisticasFinJuego();
    dibujarOpcionesReinicio();
}

/**
 * Dibuja el fondo para pantallas de finalización
 */
private void dibujarFondoPantallaFin(Color color) {
    entorno.dibujarRectangulo(400, 300, 800, 600, 0, color);
}

/**
 * Dibuja las estadísticas del juego terminado
 */
private void dibujarEstadisticasFinJuego() {
    entorno.cambiarFont("Arial", 20, Color.WHITE);

    if (estadoActual == EstadoJuego.PERDIO) {
        entorno.escribirTexto("Los zombies alcanzaron los regalos", 400, 230);
    } else {
        entorno.escribirTexto("Completaste todos los niveles y derrotaste al Zombie Boss", 400, 230);
    }

    entorno.escribirTexto("Nivel alcanzado: " + nivelActual + "/" + TOTAL_NIVELES, 400, 260);
    entorno.escribirTexto("Zombies eliminados: " + zombiesEliminados, 400, 290);
    entorno.escribirTexto("Tiempo final: " + obtenerTiempoSegundos() + " segundos", 400, 320);
}

/**
 * Dibuja las opciones de reinicio en pantallas de finalización
 */
private void dibujarOpcionesReinicio() {
    entorno.cambiarFont("Arial", 24, Color.YELLOW);
    entorno.escribirTexto("OPCIONES:", 400, 370);

    entorno.cambiarFont("Arial", 20, Color.WHITE);
    entorno.escribirTexto("R - Reiniciar Juego", 400, 410);
    entorno.escribirTexto("ESC - Volver al Menú Principal", 400, 440);
}

/**
 * Procesa el input en pantallas de finalización
 */
private void procesarInputPantallaFin() {
    if (entorno.sePresiono('r') || entorno.sePresiono('R')) {
        reiniciarJuegoCompleto();
        estadoActual = EstadoJuego.JUGANDO;
    }

    if (entorno.sePresiono(entorno.TECLA_ESCAPE)) {
```

```
            estadoActual = EstadoJuego.MENU_PRINCIPAL;
        }
    }

    // ========================================================================
    // SISTEMA DE REINICIO Y GESTIÓN DE NIVELES
    // ========================================================================

    /**
     * Reinicia completamente el juego para empezar desde el nivel 1
     */
    private void reiniciarJuegoCompleto() {
        reiniciarSistemaNiveles();
        reiniciarSistemaCesped();
        reiniciarSistemaEnemigos();
        reiniciarSistemaDisparos();
        reiniciarSistemaPlantas();
        reiniciarSistemaTiempo();
        reiniciarSistemaSeleccion();
    }

    /**
     * Reinicia el sistema de niveles
     */
    private void reiniciarSistemaNiveles() {
        this.nivelActual = 1;
        this.zombiesParaSiguienteNivel = ZOMBIES_NIVEL_1;
        this.nivelCompletado = false;
        this.jefeAparecio = false;
    }

    /**
     * Reinicia el césped (elimina todas las plantas)
     */
    private void reiniciarSistemaCesped() {
        for (int fila = 0; fila < FILAS_CESPED; fila++) {
            for (int columna = 0; columna < COLUMNAS_CESPED; columna++) {
                cesped[fila][columna] = null;
            }
        }
    }

    /**
     * Reinicia el sistema de enemigos
     */
    private void reiniciarSistemaEnemigos() {
        for (int i = 0; i < zombies.length; i++) {
            zombies[i] = null;
        }
        this.zombiesEliminados = 0;
        this.zombiesRestantes = zombiesParaSiguienteNivel;
        this.tickGeneracionZombie = 0;
```

```
    this.jefe = null;
}

/**
 * Reinicia el sistema de disparos
 */
private void reiniciarSistemaDisparos() {
    for (int i = 0; i < bolasFuego.length; i++) {
        bolasFuego[i] = null;
    }
    for (int i = 0; i < bolasHielo.length; i++) {
        bolasHielo[i] = null;
    }
}

/**
 * Reinicia el sistema de plantas (tiempos de recarga)
 */
private void reiniciarSistemaPlantas() {
    this.tiempoCargaRoseBlade = 0;
    this.tiempoCargaWallNut = 0;
    this.tiempoCargaIceFlower = 0;
}

/**
 * Reinicia el sistema de tiempo
 */
private void reiniciarSistemaTiempo() {
    this.tiempoInicio = System.currentTimeMillis();
    this.tiempoTranscurrido = 0;
    this.tiempoDetenido = false;
    this.juegoPausado = false;
}

/**
 * Reinicia el sistema de selección
 */
private void reiniciarSistemaSeleccion() {
    resetearSeleccionTeclado();
    this.plantaSeleccionada = null;
    this.arrastrando = false;
}

/**
 * Reinicia el juego para un nuevo nivel
 */
private void reiniciarParaNuevoNivel() {
    reiniciarSistemaEnemigos();
    reiniciarSistemaDisparos();
    this.nivelCompletado = false;
    this.jefeAparecio = false;
    this.jefe = null;
```

```
}

// ========================================================================
// MÉTODOS DE UTILIDAD
// ========================================================================

/**
 * Obtiene el tiempo transcurrido en segundos
 */
private int obtenerTiempoSegundos() {
    if (tiempoDetenido) {
        return (int)(tiempoTranscurrido / 1000);
    } else {
        return (int)((System.currentTimeMillis() - tiempoInicio) / 1000);
    }
}

/**
 * Obtiene el nombre del nivel actual
 */
private String obtenerNombreNivel() {
    switch (nivelActual) {
        case 1: return "INICIO";
        case 2: return "AVANZADO";
        case 3: return "ÉPICO";
        case 4: return "JEFE FINAL";
        default: return "DESCONOCIDO";
    }
}

/**
 * Obtiene la descripción del nivel actual
 */
private String obtenerDescripcionNivel() {
    switch (nivelActual) {
        case 1: return "Zombies Normales";
        case 2: return "Zombies Normales y Super Zombies";
        case 3: return "Solo Supers Zombies";
        case 4: return "¡BOSS!";
        default: return "";
    }
}

/**
 * Verifica si el mouse está sobre un área específica
 */
private boolean estaMouseSobreAvatar(int x, int y, int tamaño) {
    int mouseX = entorno.mouseX();
    int mouseY = entorno.mouseY();
    return mouseX >= x - tamaño/2 && mouseX <= x + tamaño/2 &&
           mouseY >= y - tamaño/2 && mouseY <= y + tamaño/2;
}
```

```
    /**
     * Dibuja un zombie decorativo para el menú
     */
    private void dibujarZombieDecorativo(int x, int y) {
        entorno.dibujarCirculo(x, y, 20, new Color(0, 150, 0, 150));
        entorno.dibujarRectangulo(x, y + 15, 30, 30, 0, new Color(0, 120, 0, 150));
    }

    /**
     * Dibuja efectos de estrellas en el fondo del menú
     */
    private void dibujarEfectosEstrellas() {
        for (int i = 0; i < 10; i++) {
            int x = (int)(Math.random() * ANCHO_VENTANA);
            int y = (int)(Math.random() * ALTO_VENTANA);
            int tamaño = 1 + (int)(Math.random() * 3);
            entorno.dibujarCirculo(x, y, tamaño, new Color(255, 255, 255, 100));
        }
    }

    // ========================================================================
    // MÉTODO MAIN
    // ========================================================================

    /**
     * Punto de entrada principal del juego
     */
    @SuppressWarnings("unused")
    public static void main(String[] args) {
        new Juego();
    }
}
```

CLASES (IceFlower)

```java
//CLASES IceFlower.java
package juego;
import java.awt.Color;
import java.awt.Image;
import entorno.Entorno;
import entorno.Herramientas;
/**
* Planta IceFlower - Dispara bolas de hielo
* - Ralentiza zombies progresivamente
* - Congela zombies después de 3 hits
* - Disponible desde el nivel 2
*/
public class IceFlower extends Planta {
    private int tiempoRecarga;
    private static final int RECARGA_MAXIMA = 200;
    private Image imagenIceFlower;

    public IceFlower(double x, double y) {
        super(x, y, "iceflower");
        this.salud = 2;
        this.tiempoRecarga = RECARGA_MAXIMA;

        // carga de imagen
        try {
            this.imagenIceFlower = Herramientas.cargarImagen("img/ice_flower.png");
        } catch (Exception e) {
            System.out.println("Error al cargar imagen de IceFlower: " + e.getMessage());
            this.imagenIceFlower = null;
        }
    }

    @Override
    public void dibujar(Entorno entorno) {
        // el código de dibujo geométrico con esto:
        if (imagenIceFlower != null) {
            // Dibujar la imagen
            entorno.dibujarImagen(imagenIceFlower, x, y, 0, 0.7); // Ajusta 0.4 al tamaño que quieras
        } else {
            // Código original como respaldo si no hay imagen
            Color colorFlor = new Color(100, 200, 255);
            entorno.dibujarRectangulo(x, y, 40, 40, 0, colorFlor);
            entorno.dibujarRectangulo(x, y + 25, 5, 20, 0, Color.CYAN);
        }

        // estos elementos de información (opcional)
        entorno.escribirTexto("IF " + salud, x - 10, y + 5);

        // Barra de recarga visual
        if (tiempoRecarga > 0 && salud > 0) {
            double progreso = 1.0 - (double)tiempoRecarga / RECARGA_MAXIMA;
            entorno.dibujarRectangulo(x, y - 30, 30 * progreso, 3, 0, Color.CYAN);
```

```
    }
}

/**
 * Actualiza el temporizador de recarga
 */
public void actualizar() {
    if (tiempoRecarga > 0 && salud > 0) {
        tiempoRecarga--;
    }
}

/**
 * Verifica si puede disparar (recarga completa y viva)
 */
public boolean puedeDisparar() {
    return tiempoRecarga == 0 && salud > 0;
}

/**
 * Verifica si la planta está viva
 */
public boolean estaViva() {
    return salud > 0;
}

/**
 * Reinicia el temporizador de recarga
 */
public void reiniciarRecarga() {
    tiempoRecarga = RECARGA_MAXIMA;
}

/**
 * Verifica si hay zombies en la misma fila y adelante de la planta
 */
public boolean hayZombiesEnFilaYDelante(Object[] zombies, int fila, double posicionXPlanta) {
    for (Object obj : zombies) {
        if (obj instanceof ZombieGrinch) {
            ZombieGrinch zombie = (ZombieGrinch) obj;
            if (zombie != null && zombie.getFila() == fila && !zombie.estaMuerto()) {
                if (zombie.getX() > posicionXPlanta) {
                    return true;
                }
            }
        } else if (obj instanceof SuperZombieGrinch) {
            SuperZombieGrinch zombie = (SuperZombieGrinch) obj;
            if (zombie != null && zombie.getFila() == fila && !zombie.estaMuerto()) {
                if (zombie.getX() > posicionXPlanta) {
                    return true;
                }
            }
```

```
        }
      }
      return false;
    }
}
```

CLASES (RoseBlade)

```java
//CLASES RoseBlade.java
package juego;
import java.awt.Color;
import java.awt.Image;
import entorno.Entorno;
import entorno.Herramientas;
public class RoseBlade extends Planta {
  private int tiempoRecarga;
  private static final int RECARGA_MAXIMA = 120;
  private Image imagenRoseBlade; // variable para la imagen

  public RoseBlade(double x, double y) {
    super(x, y, "roseblade");
    this.salud = 1;
    this.tiempoRecarga = RECARGA_MAXIMA;

    // carga de imagen
    try {
      this.imagenRoseBlade = Herramientas.cargarImagen("img/rose_blade.png");
    } catch (Exception e) {
      System.out.println("Error al cargar imagen de RoseBlade: " + e.getMessage());
      this.imagenRoseBlade = null;
    }
  }

  @Override
  public void dibujar(Entorno entorno) {
    // código de dibujo geométrico con esto:
    if (imagenRoseBlade != null) {
      // Dibujar la imagen
      entorno.dibujarImagen(imagenRoseBlade, x, y, 0, 0.7); // Ajusta 0.4 al tamaño que quieras
    } else {
      // Código original como respaldo si no hay imagen
      Color colorFlor = salud > 0 ? Color.RED : new Color(100, 0, 0);
      entorno.dibujarRectangulo(x, y, 40, 40, 0, colorFlor);
      entorno.dibujarRectangulo(x, y + 25, 5, 20, 0, Color.GREEN);
    }

    // elementos de información (opcional)
    entorno.escribirTexto("RB " + salud, x - 12, y + 5);

    if (tiempoRecarga > 0 && salud > 0) {
      double progreso = 1.0 - (double)tiempoRecarga / RECARGA_MAXIMA;
      entorno.dibujarRectangulo(x, y - 30, 30 * progreso, 3, 0, Color.YELLOW);
    }

    if (salud == 0) {
      entorno.escribirTexto("MUERTA", x - 15, y - 40);
    }
  }
```

```java
    // todos los demás métodos igual

    public void actualizar() {
        if (tiempoRecarga > 0 && salud > 0) {
            tiempoRecarga--;
        }
    }

    public boolean puedeDisparar() {
        return tiempoRecarga == 0 && salud > 0;
    }

    @Override
    public boolean estaViva() {
        return salud > 0;
    }

    public void reiniciarRecarga() {
        tiempoRecarga = RECARGA_MAXIMA;
    }

    public boolean hayZombiesEnFilaYDelante(Object[] zombies, int fila, double posicionXPlanta) {
        for (Object obj : zombies) {
            if (obj instanceof ZombieGrinch) {
                ZombieGrinch zombie = (ZombieGrinch) obj;
                if (zombie != null && zombie.getFila() == fila && !zombie.estaMuerto()) {
                    if (zombie.getX() > posicionXPlanta) {
                        return true;
                    }
                }
            } else if (obj instanceof SuperZombieGrinch) {
                SuperZombieGrinch zombie = (SuperZombieGrinch) obj;
                if (zombie != null && zombie.getFila() == fila && !zombie.estaMuerto()) {
                    if (zombie.getX() > posicionXPlanta) {
                        return true;
                    }
                }
            }
        }
        return false;
    }
}
```

CLASE (WallNut)

```java
//CLASE WallNut.java
package juego;
import java.awt.Color;
import java.awt.Image;
import entorno.Entorno;
import entorno.Herramientas;
public class WallNut extends Planta {
  private Image imagenWallNut;

  public WallNut(double x, double y) {
     super(x, y, "wallnut");
     this.salud = 5;

     // carga de imagen
     try {
        this.imagenWallNut = Herramientas.cargarImagen("img/nuez.png");
     } catch (Exception e) {
        System.out.println("Error al cargar imagen de WallNut: " + e.getMessage());
        this.imagenWallNut = null;
     }
  }

  @Override
  public void dibujar(Entorno entorno) {
     //  código de dibujo geométrico con esto:
     if (imagenWallNut != null) {
        // Dibujar la imagen
        entorno.dibujarImagen(imagenWallNut, x, y, 0, 0.7); // Ajusta 0.4 al tamaño que quieras
     } else {
        // Código original como respaldo si no hay imagen
        Color colorCascara = salud < 3 ? new Color(160, 80, 30) : new Color(139, 69, 19);
        Color colorInterior = salud < 3 ? new Color(230, 200, 160) : new Color(210, 180, 140);

        entorno.dibujarRectangulo(x, y, 50, 50, 0, colorCascara);
        entorno.dibujarRectangulo(x, y, 30, 30, 0, colorInterior);
     }

     // estos elementos de información (opcional)
     entorno.escribirTexto("WN " + salud, x - 12, y + 5);

     // Barra de salud (opcional - puedes quitarla si prefieres)
     double anchoBarraSalud = 40 * ((double)salud / 5);
     Color colorBarra = salud > 3 ? Color.GREEN : salud > 1 ? Color.YELLOW : Color.RED;
     entorno.dibujarRectangulo(x, y - 35, anchoBarraSalud, 4, 0, colorBarra);
  }
}
```

CLASE (ZOMBIEBOSS)

```java
//CLASE ZombieBoss.java
package juego;
import java.awt.Color;
import java.awt.Image;
import entorno.Entorno;
import entorno.Herramientas;
/**
* Zombie Boss final del juego
* - Tiene mucha vida (100 puntos)
* - Ataque lento que mata plantas aleatoriamente en todo el tablero
* - Es inmune a congelación completa, solo se ralentiza
* - Aparece en el nivel 4
*/
public class ZombieBoss {
   private double x, y;
   private double velocidad;
   private double velocidadBase;
   private int salud;
   private int tickAtaque;
   private final int RECARGA_ATAQUE = 600;
   private boolean ataqueActivo;
   private int tickRalentizacion;
   private Image imagenBoss;

   public ZombieBoss(double x, double y) {
      this.x = x;
      this.y = y;
      this.velocidadBase = 0.08;
      this.velocidad = velocidadBase;
      this.salud = 100;
      this.tickAtaque = 0;
      this.ataqueActivo = false;
      this.tickRalentizacion = 0;

      // ← AÑADIR carga de imagen
      try {
         this.imagenBoss = Herramientas.cargarImagen("img/zombie_boss.png");
      } catch (Exception e) {
         System.out.println("Error al cargar imagen del Zombie Boss: " + e.getMessage());
         this.imagenBoss = null;
      }
   }

   // Getters para acceso externo
   public double getX() { return x; }
   public double getY() { return y; }
   public int getSalud() { return salud; }
   public boolean isAtaqueActivo() { return ataqueActivo; }

   /**
    * Mueve al jefe hacia la izquierda
```

```
     */
    public void mover() {
        this.x -= velocidad;

        // Actualizar efecto de ralentización temporal
        if (tickRalentizacion > 0) {
            tickRalentizacion--;
            if (tickRalentizacion == 0) {
                // Restaurar velocidad normal cuando termina el efecto
                velocidad = velocidadBase;
            }
        }
    }

    /**
     * Actualiza el temporizador de ataque
     */
    public void actualizar() {
        tickAtaque++;
        if (tickAtaque >= RECARGA_ATAQUE) {
            ataqueActivo = true;
            tickAtaque = 0;
        }
    }

    /**
     * Ejecuta el ataque masivo que mata 5 plantas aleatoriamente
     * @param cesped Matriz de plantas del juego
     */
    public void ejecutarAtaque(Planta[][] cesped) {
        if (ataqueActivo) {
            int plantasMatadas = 0;
            int intentos = 0;
            int maxIntentos = 20;

            // Matar exactamente 5 plantas aleatoriamente
            while (plantasMatadas < 5 && intentos < maxIntentos) {
                int filaAleatoria = (int)(Math.random() * cesped.length);
                int columnaAleatoria = (int)(Math.random() * cesped[0].length);

                if (cesped[filaAleatoria][columnaAleatoria] != null &&
                    cesped[filaAleatoria][columnaAleatoria].estaViva()) {

                    cesped[filaAleatoria][columnaAleatoria].setSalud(0);
                    plantasMatadas++;
                }
                intentos++;
            }

            ataqueActivo = false;
        }
    }
```

```
    /**
     * Aplica efecto de ralentización al jefe (solo ralentización, NO congelación)
     * @param factor Factor de ralentización (0.0 a 1.0)
     */
    public void aplicarRalentizacion(double factor) {
        // El jefe solo se ralentiza, no se congela completamente
        double nuevaVelocidad = velocidadBase * Math.max(0.3, factor); // Mínimo 30% de velocidad
        if (nuevaVelocidad < velocidad) {
            velocidad = nuevaVelocidad;
            tickRalentizacion = 180; // Efecto dura 3 segundos
        }
    }

    /**
     * Reduce la salud del jefe
     */
    public void recibirDano(int dano) {
        this.salud -= dano;
    }

    /**
     * Verifica si el jefe está muerto
     */
    public boolean estaMuerto() {
        return salud <= 0;
    }

    /**
     * Dibuja al jefe con imagen y barras de información
     */
    public void dibujar(Entorno entorno) {
        // código de dibujo geométrico con imagen
        if (imagenBoss != null) {
            // Dibujar la imagen del jefe
            entorno.dibujarImagen(imagenBoss, x, y, 0, 0.8); // Escala mayor por ser el jefe
        } else {
            // Código original como respaldo si no hay imagen
            dibujarGeometrico(entorno);
        }

        // elementos UI importantes

        // Barra de salud (proporcional a los 100 puntos de vida)
        double anchoBarraSalud = 70 * ((double)salud / 100);
        Color colorBarraSalud = salud > 50 ? Color.RED : salud > 25 ? Color.ORANGE :
Color.YELLOW;
        entorno.dibujarRectangulo(x, y - 150, anchoBarraSalud, 10, 0, colorBarraSalud);

        // Barra de carga de ataque
        double progresoAtaque = (double)tickAtaque / RECARGA_ATAQUE;
        entorno.dibujarRectangulo(x, y - 165, 70 * progresoAtaque, 5, 0, Color.MAGENTA);
```

```
        // Indicador de ralentización
        if (tickRalentizacion > 0) {
            entorno.dibujarRectangulo(x, y - 175, 30, 3, 0, Color.CYAN);
            entorno.escribirTexto("RALENTIZADO", x - 25, y - 185);
        }

        // Texto informativo
        entorno.escribirTexto("BOSS " + salud + "/100", x - 25, y - 160);

        // Advertencia de ataque inminente
        if (ataqueActivo) {
            entorno.escribirTexto("¡ATAQUE INMINENTE!", x - 40, y - 200);
        }
    }

    /**
     * Método de respaldo para dibujo geométrico
     */
    private void dibujarGeometrico(Entorno entorno) {
        Color colorCuerpo = new Color(128, 0, 128);

        // Cuerpo del jefe
        entorno.dibujarRectangulo(x, y, 80, 200, 0, colorCuerpo);
        entorno.dibujarCirculo(x, y - 90, 40, colorCuerpo);

        // Ojos
        entorno.dibujarCirculo(x - 15, y - 95, 8, Color.RED);
        entorno.dibujarCirculo(x + 15, y - 95, 8, Color.RED);

        // Corona
        entorno.dibujarRectangulo(x, y - 130, 50, 12, 0, Color.YELLOW);
        entorno.dibujarRectangulo(x - 15, y - 120, 8, 15, 0.5, Color.YELLOW);
        entorno.dibujarRectangulo(x + 15, y - 120, 8, 15, -0.5, Color.YELLOW);
        entorno.dibujarRectangulo(x, y - 120, 8, 15, 0, Color.YELLOW);
    }
}
```

CLASE (ZombieGrinch)

```java
//Clase ZombieGrinch.java
package juego;
import java.awt.Color;
import java.awt.Image;
import entorno.Entorno;
import entorno.Herramientas;
/**
* Clase que representa al zombie básico del juego.
* - Se mueve hacia la izquierda atacando plantas
* - Tiene salud y velocidad base
* - Puede ser ralentizado por bolas de hielo
* - Ataca plantas cuando están cerca
*/
public class ZombieGrinch {
   private double x;        // Posición en el eje X
   private double y;        // Posición en el eje Y
   private double velocidad; // Velocidad de movimiento (puede ser modificada por hielo)
   private int salud;       // Salud del zombie (4 puntos base)
   private int fila;        // Fila en la que se encuentra (0-4)
   private boolean atacando; // Indica si está atacando una planta
   private Planta plantaObjetivo; // Planta que está siendo atacada
   private int tickAtaque;  // Contador para temporizar los ataques
   private Image imagenZombie; // variable para la imagen
   private Image imagenZombieAtacando; // imagen para estado de ataque (opcional)

   /**
    * Constructor del zombie básico
    * @param x Posición inicial en X
    * @param y Posición inicial en Y
    * @param fila Fila asignada (0-4)
    */
   public ZombieGrinch(double x, double y, int fila) {
      this.x = x;
      this.y = y;
      this.fila = fila;
      this.velocidad = 0.5;  // Velocidad base
      this.salud = 4;        // Salud base
      this.atacando = false;
      this.plantaObjetivo = null;
      this.tickAtaque = 0;

      // carga de imágenes
      try {
         this.imagenZombie = Herramientas.cargarImagen("img/zombie_grinch.png");
         this.imagenZombieAtacando = Herramientas.cargarImagen("img/zombie_grinch.png");
      } catch (Exception e) {
         System.out.println("Error al cargar imágenes del zombie: " + e.getMessage());
         this.imagenZombie = null;
         this.imagenZombieAtacando = null;
      }
   }
```

```
// ==================== GETTERS Y SETTERS ====================
// todos los getters y setters existentes SIN CAMBIOS
public double getX() {
        return x;
}

public double getY() {
        return y;
}

public int getSalud() {
        return salud;
}

public int getFila() {
        return fila;
}

public boolean estaAtacando() {
        return atacando;
}

public void setSalud(int salud) {
        this.salud = salud;
}

public void setVelocidad(double velocidad) {
        this.velocidad = velocidad;
}

public double getVelocidad() {
        return velocidad;
}

// ==================== MÉTODOS DE COMPORTAMIENTO ====================
// todos los métodos de comportamiento SIN CAMBIOS
public void mover(Planta[][] cesped) {
    if (atacando) {
        return;
    }

    Planta plantaCercana = detectarPlantaCercana(cesped);
    if (plantaCercana != null) {
        atacando = true;
        plantaObjetivo = plantaCercana;
    } else {
        this.x -= velocidad;
    }
}

public void atacar() {
```

```
    if (atacando && plantaObjetivo != null) {
        tickAtaque++;

        if (tickAtaque >= 60) {
            plantaObjetivo.setSalud(plantaObjetivo.getSalud() - 1);
            tickAtaque = 0;

            if (plantaObjetivo.getSalud() <= 0) {
                atacando = false;
                plantaObjetivo = null;
            }
        }
    }
}

private Planta detectarPlantaCercana(Planta[][] cesped) {
    for (int col = 0; col < cesped[0].length; col++) {
        if (cesped[fila][col] != null) {
            Planta planta = cesped[fila][col];
            double distancia = Math.abs(this.x - planta.getX());

            if (distancia < 50) {
                return planta;
            }
        }
    }
    return null;
}

public void recibirDano() {
    this.salud--;
}

public boolean estaMuerto() {
    return salud <= 0;
}

// ==================== MÉTODOS GRÁFICOS ====================

/**
 * Dibuja el zombie en el entorno gráfico
 * @param entorno Entorno donde se dibujará
 */
public void dibujar(Entorno entorno) {
    // código de dibujo geométrico con esto:
    if (imagenZombie != null) {
        // Seleccionar imagen según estado
        Image imagenActual = atacando && imagenZombieAtacando != null ?
                             imagenZombieAtacando : imagenZombie;

        // Dibujar la imagen del zombie
        entorno.dibujarImagen(imagenActual, x, y, 0, 0.8); // Ajusta 0.5 al tamaño que quieras
```

```
        } else {
            // Código original como respaldo si no hay imágenes
            dibujarGeometrico(entorno);
        }

        // elementos de información y UI
        // Barra de salud (proporcional a los 4 puntos base)
        double anchoBarraSalud = 40 * ((double)salud / 4);
        entorno.dibujarRectangulo(x, y - 35, anchoBarraSalud, 5, 0, Color.RED);

        // Indicador visual de ataque
        if (atacando) {
            entorno.dibujarRectangulo(x, y - 45, 30, 3, 0, Color.ORANGE);
            entorno.escribirTexto("ATACANDO", x - 25, y - 50);
        }

        // Información de debug: fila y salud
        entorno.escribirTexto("F" + fila + " S:" + salud, x, y + 45);
    }

    /**
     * Método de respaldo para dibujo geométrico si no se cargan las imágenes
     */
    private void dibujarGeometrico(Entorno entorno) {
        // Color del cuerpo (verde normal o verde oscuro si está atacando)
        Color colorCuerpo = atacando ? new Color(0, 150, 0) : Color.GREEN;

        // Cuerpo principal del zombie
        entorno.dibujarRectangulo(x, y, 40, 60, 0, colorCuerpo);

        // Cabeza del zombie
        entorno.dibujarCirculo(x, y - 15, 25, colorCuerpo);

        // Ojos del zombie
        entorno.dibujarCirculo(x - 8, y - 18, 5, Color.BLACK);
        entorno.dibujarCirculo(x + 8, y - 18, 5, Color.BLACK);

        // Boca del zombie (roja si está atacando)
        Color colorBoca = atacando ? Color.RED : new Color(200, 0, 0);
        entorno.dibujarRectangulo(x, y - 5, 15, 3, 0, colorBoca);
    }
}
```

CLASE (SuperZombieGrinch)

```java
//Clase SuperZombieGrinch.java
package juego;
import java.awt.Color;
import java.awt.Image;
import entorno.Entorno;
import entorno.Herramientas;
public class SuperZombieGrinch extends ZombieGrinch {
  private boolean inmuneACongelacion;
  private Image imagenSuperZombie;

  public SuperZombieGrinch(double x, double y, int fila) {
     super(x, y, fila);
     setSalud(12);
     setVelocidad(0.6);
     this.inmuneACongelacion = true;

     // carga de imagen
     try {
        this.imagenSuperZombie = Herramientas.cargarImagen("img/super_zombie_grinch.png");
     } catch (Exception e) {
        System.out.println("Error al cargar imagen del super zombie: " + e.getMessage());
        this.imagenSuperZombie = null;
     }
  }

  /**
   * Getter para verificar inmunidad
   */
  public boolean esInmuneACongelacion() {
     return inmuneACongelacion;
  }

  /**
   * Aplica ralentización pero NO congelación completa
   */
  public void aplicarRalentizacion(double factor) {
     // El super zombie se ralentiza pero no se congela completamente
     double nuevaVelocidad = 0.3 * Math.max(0.5, factor); // Mínimo 50% de velocidad
     setVelocidad(nuevaVelocidad);
  }

  @Override
  public void dibujar(Entorno entorno) {
     // ← REEMPLAZAR código de dibujo geométrico con imagen
     if (imagenSuperZombie != null) {
        // Dibujar la imagen del super zombie
        entorno.dibujarImagen(imagenSuperZombie, getX(), getY(), 0, 0.8); // Tamaño un poco mayor
     } else {
        // Código original como respaldo si no hay imagen
        dibujarGeometrico(entorno);
     }
```

```
    // elementos UI importantes

    // Barra de salud naranja (proporcional a 8 puntos de vida)
    double anchoBarraSalud = 40 * ((double)getSalud() / 8);
    entorno.dibujarRectangulo(getX(), getY() - 45, anchoBarraSalud, 6, 0, Color.ORANGE);

    // Indicador de inmunidad al hielo
    if (inmuneACongelacion) {
        entorno.dibujarRectangulo(getX(), getY() - 55, 20, 3, 0, Color.CYAN);
        entorno.escribirTexto("INMUNE HIELO", getX() - 25, getY() - 60);
    }

    // Texto identificador
    entorno.escribirTexto("SUPER", getX() - 15, getY() + 50);
}

/**
 * Método de respaldo para dibujo geométrico
 */
private void dibujarGeometrico(Entorno entorno) {
    Color colorCuerpo = estaAtacando() ? new Color(0, 0, 150) : Color.BLUE;
    entorno.dibujarRectangulo(getX(), getY(), 45, 65, 0, colorCuerpo);

    entorno.dibujarCirculo(getX(), getY() - 15, 28, colorCuerpo);
    entorno.dibujarCirculo(getX() - 8, getY() - 18, 6, Color.WHITE);
    entorno.dibujarCirculo(getX() + 8, getY() - 18, 6, Color.WHITE);

    // Corona amarilla
    entorno.dibujarRectangulo(getX(), getY() - 40, 25, 8, 0, Color.YELLOW);
    entorno.dibujarRectangulo(getX() - 8, getY() - 35, 5, 10, 0.5, Color.YELLOW);
    entorno.dibujarRectangulo(getX() + 8, getY() - 35, 5, 10, -0.5, Color.YELLOW);
}
}
```

CLASE (Regalo)

```java
// Clase Regalo.java
package juego;
import java.awt.Color;
import java.awt.Image;
import entorno.Entorno;
import entorno.Herramientas;
public class Regalo {
  private double x;
  private double y;
  private boolean destruido;
  private Image imagenRegalo;

  public Regalo(double x, double y) {
     this.x = x;
     this.y = y;
     this.destruido = false;

     // Cargar la imagen del regalo
     try {
        this.imagenRegalo = Herramientas.cargarImagen("img/regalo.png");
     } catch (Exception e) {
        System.out.println("Error al cargar imagen del regalo: " + e.getMessage());
        this.imagenRegalo = null;
     }
  }

  public double getX() { return x; }
  public double getY() { return y; }
  public boolean isDestruido() { return destruido; }
  public void destruir() { this.destruido = true; }

  public void dibujar(Entorno entorno) {
     if (destruido) return;

     if (imagenRegalo != null) {
        // SOLO LA IMAGEN - sin animaciones
        entorno.dibujarImagen(imagenRegalo, x, y, 0, 0.8); // Ajusta el 0.4 al tamaño que prefieras
     } else {
        // Código de respaldo si no se carga la imagen
        dibujarGeometrico(entorno);
     }
  }

  // Método de respaldo por si falla la carga de la imagen
  private void dibujarGeometrico(Entorno entorno) {
     // Caja del regalo
     entorno.dibujarRectangulo(x, y, 50, 50, 0, Color.RED);

     // Moño
     entorno.dibujarRectangulo(x, y - 10, 40, 8, 0, Color.GREEN);
     entorno.dibujarRectangulo(x, y, 8, 40, 0, Color.GREEN);
```

```
    // Detalles decorativos
    entorno.dibujarRectangulo(x, y, 50, 5, 0, Color.WHITE);
    entorno.dibujarRectangulo(x, y, 5, 50, 0, Color.WHITE);
  }
}
```

CLASE (Planta)

```java
// Clases Planta.java
package juego;
import java.awt.Color;
import entorno.Entorno;
/**
* Clase abstracta que representa una planta en el juego.
* Sirve como base para todos los tipos de plantas implementadas.
*
* Características principales:
* - Posición en el campo de juego (x, y)
* - Salud o puntos de vida
* - Tipo específico de planta
* - Estado de vida (viva/muerta)
*
* Implementa el patrón Template Method para forzar a las subclases
* a implementar su propio método de dibujo.
*/
public abstract class Planta {

  // ==================== VARIABLES DE INSTANCIA ====================

  /**
   * Coordenada X de la planta en el campo de juego
   * Representa la posición horizontal en píxeles
   */
  protected double x;

  /**
   * Coordenada Y de la planta en el campo de juego
   * Representa la posición vertical en píxeles
   */
  protected double y;

  /**
   * Puntos de vida de la planta
   * Cuando llega a 0, la planta muere y es removida del juego
   */
  protected int salud;

  /**
   * Identificador del tipo de planta
   * Ejemplos: "roseblade", "wallnut", "iceflower"
   * Usado para lógica específica por tipo
   */
  protected String tipo;

  // ==================== CONSTRUCTOR ====================

  /**
   * Constructor base para todas las plantas.
   * Inicializa posición y tipo, pero deja la salud para las subclases específicas.
```

```
     *
     * @param x Posición horizontal inicial en píxeles
     * @param y Posición vertical inicial en píxeles
     * @param tipo Identificador único del tipo de planta
     */
    public Planta(double x, double y, String tipo) {
        this.x = x;
        this.y = y;
        this.tipo = tipo;
    }

    // ==================== MÉTODOS ACCESORES (GETTERS) ====================

    /**
     * @return La coordenada X actual de la planta
     */
    public double getX() {
        return x;
    }

    /**
     * @return La coordenada Y actual de la planta
     */
    public double getY() {
        return y;
    }

    /**
     * @return Los puntos de vida actuales de la planta
     */
    public int getSalud() {
        return salud;
    }

    /**
     * @return El identificador del tipo de planta
     */
    public String getTipo() {
        return tipo;
    }

    // ==================== MÉTODOS MUTADORES (SETTERS) ====================

    /**
     * Establece una nueva cantidad de puntos de vida para la planta.
     * Usado cuando la planta recibe daño o se cura.
     *
     * @param salud Nuevo valor de salud (debe ser >= 0)
     */
    public void setSalud(int salud) {
        this.salud = salud;
    }
```

```java
    /**
     * Cambia la posición horizontal de la planta.
     * Útil para el sistema de movimiento con teclado.
     *
     * @param x Nueva coordenada X
     */
    public void setX(double x) {
        this.x = x;
    }

    /**
     * Cambia la posición vertical de la planta.
     * Útil para el sistema de movimiento con teclado.
     *
     * @param y Nueva coordenada Y
     */
    public void setY(double y) {
        this.y = y;
    }

    // ==================== MÉTODOS DE ESTADO ====================

    /**
     * Verifica si la planta está viva.
     * Una planta se considera viva si tiene más de 0 puntos de salud.
     *
     * @return true si la planta está viva, false si está muerta
     */
    public boolean estaViva() {
        return salud > 0;
    }

    // ==================== MÉTODO ABSTRACTO ====================

    /**
     * Método abstracto que debe ser implementado por todas las subclases.
     * Define cómo se dibuja cada tipo específico de planta en pantalla.
     *
     * Patrón: Template Method - fuerza a las subclases a proporcionar
     * su propia implementación de renderizado.
     *
     * @param entorno El objeto Entorno usado para dibujar en pantalla
     */
    public abstract void dibujar(Entorno entorno);
}
```

CLASE (BolaDeFuego)

```java
// Clase BolaDeFuego.java
package juego;
import java.awt.Color;
import entorno.Entorno;
public class BolaDeFuego {
    private double x;
    private double y;
    private double velocidad;
    private int dano;

    public BolaDeFuego(double x, double y) {
        this.x = x;
        this.y = y;
        this.velocidad = 2.0; // Velocidad moderada
        this.dano = 1;
    }

    public double getX() { return x; }
    public double getY() { return y; }
    public int getDano() { return dano; }

    public void mover() {
        this.x += velocidad; // Se mueve hacia la derecha
    }

    public boolean estaFueraDePantalla(int anchoPantalla) {
        return x > anchoPantalla + 50; // +50 para margen
    }

    public void dibujar(Entorno entorno) {
        // Bola de fuego principal
        entorno.dibujarCirculo(x, y, 15, Color.ORANGE);
        entorno.dibujarCirculo(x, y, 10, Color.YELLOW);

        // Efecto de llamas
        entorno.dibujarRectangulo(x + 8, y, 5, 12, 0.3, Color.RED);
        entorno.dibujarRectangulo(x - 8, y, 5, 12, -0.3, Color.RED);
    }
}
```

CLASE (BolaDeHielo)

```java
// Clase BolaDeHielo.java
package juego;
import java.awt.Color;
import entorno.Entorno;
/**
* Representa una bola de hielo disparada por IceFlower
* - Ralentiza zombies progresivamente
* - Congela completamente después de 3 hits
* - Se mueve más lento que las bolas de fuego
*/
public class BolaDeHielo {
   private double x;
   private double y;
   private double velocidad;
   private int dano;
   private int hits;
   private boolean activa; // Controla si la bola sigue activa después de golpear

   public BolaDeHielo(double x, double y) {
      this.x = x;
      this.y = y;
      this.velocidad = 1.5; // Más lenta que la bola de fuego
      this.dano = 1;
      this.hits = 0;
      this.activa = true; // La bola comienza activa
   }

   // Getters para acceso externo
   public double getX() { return x; }
   public double getY() { return y; }
   public int getDano() { return dano; }
   public int getHits() { return hits; }
   public boolean isActiva() { return activa; }

   public void incrementarHits() {
      this.hits++;
      // Si llega a 3 hits, la bola se desactiva (se usa para congelar)
      if (this.hits >= 3) {
         this.activa = false;
      }
   }

   public void desactivar() {
      this.activa = false;
   }

   /**
    * Mueve la bola de hielo hacia la derecha
    */
   public void mover() {
      if (activa) {
```

```
            this.x += velocidad;
        }
    }

    /**
     * Verifica si la bola salió de la pantalla
     */
    public boolean estaFueraDePantalla(int anchoPantalla) {
        return x > anchoPantalla + 50 || !activa;
    }

    /**
     * Dibuja la bola de hielo con efectos visuales
     */
    public void dibujar(Entorno entorno) {
        if (!activa) return;

        // Bola de hielo principal
        entorno.dibujarCirculo(x, y, 12, Color.BLUE);
        entorno.dibujarCirculo(x, y, 8, Color.BLUE);

        // Efecto de cristales de hielo
        entorno.dibujarRectangulo(x + 6, y, 3, 8, 0.2, new Color(200, 230, 255));
        entorno.dibujarRectangulo(x - 6, y, 3, 8, -0.2, new Color(200, 230, 255));

        // Indicador de hits acumulados
        if (hits > 0) {
            entorno.escribirTexto("" + hits, x, y - 15);
        }
    }
}
```