Trabajo práctico final
Programación 1

Verón Duarte Santiago Nicolás, DNI 44997638, mail: verondsanti@gmail.com
Alex Sebastián Ramos Trinidad, DNI 94174389, mail: alexramos2086@gmail.com
Garvich Pedro Ignacio, DNI: 40475253, mail: pigarvich@campus.ungs.edu.ar

Desarrollo de videojuego “La Invasión de los Zombies Grinch”

INTRODUCCIÓN

Este trabajo práctico consiste en el desarrollo de un videojuego de defensa basado en la temática navideña de "El Grinch", implementado en Java utilizando el entorno de desarrollo 2D proporcionado por la cátedra. El juego enfrenta al jugador contra una horda de Zombies Grinch que intentan robar los regalos de Navidad del pueblo de Whoville.

El objetivo principal del juego es defender cinco regalos navideños ubicados en el lado izquierdo del patio contra el avance constante de los zombies. Para ello, el jugador dispone de tres tipos de plantas con habilidades especiales: Rose Blade (rosas que lanzan bolas de fuego), Wall-Nut (nueces defensivas que frenan a los zombies) y Chile (plantas explosivas que eliminan múltiples enemigos en un área).

El desarrollo del juego implementó todos los requerimientos obligatorios establecidos en la consigna, incluyendo el sistema de plantación y movimiento de plantas, la generación y comportamiento de diferentes tipos de zombies, las condiciones de victoria y derrota, y la gestión de recursos mediante un sistema de abono. Además, se incorporaron varios elementos opcionales como el Zombie Grinch Colosal, plantas explosivas y un sistema de combate complejo que incluye diferentes tipos de ataques y defensas.

El proyecto fue desarrollado aplicando los conceptos de programación orientada a objetos vistos en la materia, utilizando exclusivamente arrays para la gestión de colecciones de objetos y asegurando un diseño modular donde cada clase tiene responsabilidades bien definidas.

DESCRIPCIÓN DE CLASES

CLASE JUEGO

Descripción General:

La clase Juego es la clase principal y actúa como el núcleo central del programa. Esta clase coordina todas las demás componentes del juego y maneja el bucle principal de ejecución a través del método tick().

Variables de Instancia:

entorno: Controla la ventana gráfica y la gestión del tiempo
estado: Maneja los diferentes estados del juego (menú, jugando, pausa, victoria, derrota)
reloj: Administra el tiempo transcurrido en el juego
cripta: Gestiona la generación y comportamiento de los zombies
menu: Controla la interfaz de usuario y menús
jardin: Administra las plantas, su plantación y movimiento
combate: Maneja las interacciones de combate entre plantas y zombies

Métodos Principales:

Juego(): Constructor que inicializa todos los componentes del juego
tick(): Método ejecutado en cada frame que actualiza el estado del juego y coordina todas las acciones
main(String[] args): Punto de entrada del programa

Funcionalidad:

La clase Juego actúa como el director de orquesta del programa, coordinando en cada frame (tick) las actualizaciones de estado, el dibujado de elementos, la gestión del tiempo, y las interacciones entre todos los subsistemas del juego.

CLASE ESTADO

Descripción General:

La clase Estado es una clase de utilidad que maneja y controla los diferentes estados posibles del juego. Implementa un sistema de máquina de estados simple que permite transicionar entre las distintas fases del juego de manera organizada.

Variables de Instancia:

estadoActual: Almacena el estado actual del juego

estadoAnterior: Guarda el estado previo para posibles transiciones o restauraciones

Constantes estáticas que definen los posibles estados: INICIO, JUGANDO, PAUSA, DERROTA, VICTORIA

Métodos Principales:

Estado(): Constructor que inicializa el juego en estado INICIO

getEstado(): Devuelve el estado actual del juego

setEstado(int nuevoEstado): Cambia el estado del juego al especificado

Métodos de verificación: esInicio(), esJuego(), esPausa(), esDerrota(), esVictoria() - cada uno retorna true si el juego se encuentra en ese estado específico

Funcionalidad:

Esta clase proporciona una forma centralizada y segura de manejar los estados del juego. Los métodos booleanos de verificación permiten un código más legible al evitar comparaciones directas con constantes numéricas en otras partes del programa. El seguimiento del estado anterior facilita funcionalidades como reanudar el juego después de una pausa.

CLASE RELOJ

Descripción General:

La clase Reloj es un componente fundamental que gestiona el tiempo del juego de manera precisa. Proporciona funcionalidades para controlar animaciones, medir el tiempo transcurrido y manejar diferentes estados temporales como pausas y reanudaciones.

Variables de Instancia:

entorno: Referencia al entorno gráfico para acceder al tiempo del sistema

segundos, minutos, milisegundos: Almacenan el tiempo desglosado

fueraDeJuego: Tiempo acumulado fuera del juego activo

enJuego: Tiempo transcurrido durante la partida activa

enPausa: Tiempo acumulado en estado de pausa

cero100: Variable auxiliar para contadores

Métodos Principales:

ciclos(int indice, int divisor): Método fundamental para gestionar animaciones mediante división modular del tiempo

tiempoObjeto(boolean iniciar, boolean parar, boolean continuar, boolean correr): Controla los diferentes estados temporales del juego

mostrarTiempo(int tamanio, int posX, int posY): Muestra el tiempo formateado en pantalla

getTiempo(): Retorna el tiempo actual de juego

Funcionalidad:

El método ciclos() es la columna vertebral del sistema de animaciones del juego, permitiendo crear efectos visuales periódicos como el parpadeo de plantas y zombies. El sistema de gestión de tiempo permite pausar y reanudar el juego manteniendo la precisión temporal. La clase proporciona

un reloj confiable que podría extenderse para crear temporizadores específicos para diferentes objetos del juego.

CLASE MENU

Descripción General:

La clase Menu se encarga de gestionar toda la interfaz de usuario del juego, incluyendo las pantallas de inicio, pausa, victoria y derrota, así como la barra superior durante el juego. Controla la visualización de botones, fondos y elementos de interfaz.

Variables de Instancia:

entorno, estado: Referencias al entorno gráfico y estado del juego

cuantosTicks, ticksFuera, contar, cortar: Variables para control de temporizadores en transiciones

aRose, aNuez, aChile: Indicadores de selección de plantas

rosasPosibles, nuecesPosibles, chilesPosibles: Contadores de plantas disponibles para comprar

Métodos Principales:

sobre(int masX, int menosX, int masY, int menosY): Detecta si el mouse está sobre un área rectangular específica

dibujarMenu(): Método principal que dibuja todas las interfaces según el estado del juego

contarTicks(boolean iniciar): Solución implementada para prevenir colocación accidental de plantas al reanudar el juego

Funcionalidad:

La clase gestiona múltiples estados de interfaz:

Pantalla de inicio: Con efecto hover en el botón de comenzar

Juego activo: Muestra botones de plantas con estados seleccionados/no seleccionados y números indicando disponibilidad

Sistema de pausa: Implementa un temporizador crítico que requiere 20 ticks de espera antes de reanudar, evitando que al hacer click en "Continuar" se coloque accidentalmente una planta en el jardín

Pantallas de victoria/derrota: Con fondos temáticos y elementos visuales diferenciados

Problema Resuelto:

El método contarTicks() fue una solución necesaria frente al problema de las transiciones entre estados, particularmente al retornar de pausa al juego. Sin este sistema de temporizador, al presionar el botón "Continuar" muchas veces se colocaba accidentalmente una planta en el jardín debido a que el click se propagaba inmediatamente al estado de juego activo.

El método sobre() proporciona detección de colisiones mouse-rectángulo para todos los elementos interactivos. La clase utiliza un sistema de capas para dibujar correctamente los elementos (fondos, botones, números, botón de pausa) en el orden apropiado.

CLASES DE ZOMBIES

Descripción General:

Las clases ZBase, ZAlter y ZColosal representan los tres tipos de zombies Grinch atacantes. Cada una implementa un enemigo con características únicas de velocidad y resistencia, siguiendo un patrón de diseño similar pero con variaciones específicas.

CLASE ZBASE

Características:

Tipo: Zombie básico, equilibrio entre velocidad y resistencia con capacidad de disparo.

Velocidad: 1.0 píxeles por tick

Vida: 80 puntos
Ataque: Dispara bolas de nieve cada 3000 ms (3 segundos)
Comportamiento: Movimiento lineal hacia la izquierda con capacidad ofensiva

Variables de Instancia:
vivo: Estado de vida del zombie
detener: Controla si está siendo frenado por plantas
posX, posY: Posición en el mapa
vida: Puntos de salud
zombieVictorioso: Indica si este zombie causó la derrota
ultimoDisparo: Control de tiempo entre disparos de bolas de nieve

Métodos Principales:
desplazar(): Movimiento condicional según estado del juego
victoriaZombie(): Verifica si alcanzó los regalos
puedeDisparar(Reloj reloj): Verifica si puede realizar un disparo (solo si no está detenido)

CLASE ZALTER
Características:
Tipo: Zombie ágil, rápido pero frágil
Velocidad: 1.5 píxeles por tick (50% más rápido que ZBase)
Vida: 40 puntos (50% menos resistencia que ZBase)
Ataque: Dispara bolas de nieve cada 3000 ms (3 segundos)
Comportamiento: Movimiento rápido hacia la izquierda con capacidad ofensiva

Variables de Instancia:
Mismas variables que ZBase con diferentes valores de vida
ultimoDisparo: Control de tiempo entre disparos de bolas de nieve

Métodos Principales:
desplazar(): Movimiento más rápido que ZBase
victoriaZombie(): Misma lógica de victoria
puedeDisparar(Reloj reloj): Verifica si puede realizar un disparo (solo si no está detenido)

CLASE ZCOLOSAL
Características:
Tipo: Zombie final, extremadamente resistente pero lento
Velocidad: 0.2 píxeles por tick (muy lento)
Vida: 1000 puntos (enorme resistencia)
Comportamiento: Movimiento lento pero implacable

Variables de Instancia:
derecha: Control adicional para movimiento bidireccional
vida: 1000 puntos (12.5 veces más que ZBase)
posX como double para movimiento más preciso

Métodos Principales:
desplazar(): Movimiento muy lento con posibilidad de cambiar dirección
victoriaZombie(): Condición de victoria idéntica

Patrón de Diseño Común:
Todas las clases comparten:
Referencia al Estado del juego
Sistema de vida y vivo para gestión de salud
Métodos desplazar() y victoriaZombie() con implementaciones específicas
Comportamiento diferenciado en estados de juego y derrota
Detección de colisión con los regalos (posX <= 100)

Diferencias Clave:
Velocidad: ZAlter > ZBase > ZColosal
Resistencia: ZColosal >> ZBase > ZAlter
ZColosal no se detiene al colisionar con las plantas.
ZAlter y ZBase comparten capacidad de ataque a distancia.

CLASES DE PLANTAS
Descripción General:
Las clases Rosa, Nuez y Chile representan los tres tipos de plantas defensivas disponibles para proteger los regalos de los zombies Grinch. Cada planta tiene funciones especializadas: ataque a distancia, defensa y ataque en área respectivamente.

CLASE ROSA
Características:
Tipo: Planta ofensiva de ataque a distancia
Función: Lanza bolas de fuego hacia la derecha
Costo: 60 unidades de abono
Vida: 100 puntos
Cadencia: Dispara cada 2000 milisegundos (2 segundos)

Variables de Instancia:
entorno, estado, reloj: Referencias para funcionalidad del juego
vivo: Estado de la planta
posX, posY: Posición en el jardín
vida: Puntos de salud
abonoN: Costo en abono para plantar
ultimoDisparo: Control de tiempo entre disparos

Métodos Principales:
disparar(): Crea y retorna una nueva BolaDeFuego
puedeDisparar(): Verifica si ha pasado el tiempo suficiente desde el último disparo

Funcionalidad:
La Rosa es la principal planta ofensiva del juego. Implementa un sistema de cooldown para regular la cadencia de disparo y crea proyectiles que se desplazan horizontalmente para eliminar zombies a distancia.

CLASE NUEZ
Características:
Tipo: Planta defensiva pura
Función: Frenar y detener el avance de zombies
Costo: 40 unidades de abono
Vida: 1200 puntos (extremadamente resistente)
Ataque: No tiene capacidad ofensiva

Variables de Instancia:
vivo: Estado de la planta
posX, posY: Posición en el jardín
vida: Puntos de salud (muy alto)
abonoN: Costo en abono para plantar

Funcionalidad:

La Nuez actúa como barrera defensiva, diseñada específicamente para absorber daño y detener el avance de los zombies. Su enorme cantidad de vida la convierte en una excelente opción para crear líneas defensivas y proteger a las plantas ofensivas.

CLASE CHILE

Características:

Tipo: Planta explosiva de área
Función: Explota y daña múltiples zombies en un radio
Costo: 100 unidades de abono
Vida: 100 puntos
Daño: 300 puntos por explosión
Duración: 60 ticks de animación de explosión

Variables de Instancia:

posX, posY: Posición en el jardín
vida, damage: Salud y poder de ataque
abonoN: Costo en abono para plantar
explotando, contadorExplosion: Control del estado de explosión
minXExplosion, maxXExplosion, minYExplosion, maxYExplosion: Define el área de efecto

Métodos Principales:

calcularRangoExplosion(): Método clave que determina el área de efecto basado en la posición de la planta

Funcionalidad:

El Chile es una planta táctica de alto costo pero gran poder. Al ser atacada, explota causando daño masivo a todos los zombies dentro de su área de efecto calculada dinámicamente. El rango de explosión varía según la posición en el tablero, creando un sistema estratégico donde la ubicación es crucial para maximizar su efectividad.

CLASES DE PROYECTILES

Descripción General:

Las clases BolaDeFuego y BolaDeNieve representan los proyectiles utilizados en el combate del juego. Cada una implementa un tipo de ataque con características únicas de dirección, velocidad y daño, siguiendo un patrón de diseño similar pero con comportamientos opuestos.

CLASE BOLADEFUEGO

Características:

Origen: Disparada por las plantas Rose Blade
Dirección: Hacia la derecha (ataque ofensivo)
Velocidad: 7 píxeles por tick (rápida)
Daño: 20 puntos por impacto
Comportamiento: Movimiento lineal hacia la derecha

Variables de Instancia:

entorno, estado, reloj: Referencias para funcionalidad del juego

rosa: Referencia a la planta que realizó el disparo
posX, posY: Posición actual del proyectil
damage: Puntos de daño que inflige

Métodos Principales:
BolaDeFuego(): Constructor que inicializa la posición basándose en la rosa que dispara
desplazar(): Movimiento hacia la derecha a velocidad constante

Funcionalidad:
La BolaDeFuego es el proyectil ofensivo principal del jugador. Se crea en la posición exacta de la Rosa que la dispara y se desplaza rápidamente hacia la derecha para impactar contra los zombies. Su daño moderado la hace efectiva contra la mayoría de enemigos.

CLASE BOLADENIEVE
Características:
Origen: Disparada por los zombies ZBase y ZAlter
Dirección: Hacia la izquierda (ataque defensivo enemigo)
Velocidad: 5 píxeles por tick (moderada)
Daño: 10 puntos por impacto
Comportamiento: Movimiento lineal hacia la izquierda

Variables de Instancia:
entorno, estado, reloj: Referencias para funcionalidad del juego
posX, posY: Posición actual del proyectil
damage: Puntos de daño que inflige

Métodos Principales:
BolaDeNieve(): Constructor que recibe posición inicial explícita
desplazar(): Movimiento hacia la izquierda a velocidad constante

Funcionalidad:
La BolaDeNieve es el proyectil ofensivo de los zombies, diseñado para debilitar las defensas del jugador. Se desplaza hacia la izquierda a una velocidad ligeramente menor que las bolas de fuego, permitiendo cierta capacidad de reacción del jugador.

Patrón de Diseño Común:
Ambas clases comparten:
Sistema de movimiento lineal simple y eficiente
Gestión de daño diferenciado
Uso de referencias al entorno para posible expansión futura
Diseño minimalista centrado en su función principal

Diferencias Clave:
Dirección: BolaDeFuego (→ derecha) vs BolaDeNieve (← izquierda)
Velocidad: BolaDeFuego (7) > BolaDeNieve (5)
Daño: BolaDeFuego (20) > BolaDeNieve (10)
Constructor: BolaDeFuego usa referencia a Rosa, BolaDeNieve usa coordenadas directas

Estrategia de Implementación:
Los proyectiles implementan un diseño simple pero efectivo, optimizado para el rendimiento dado la gran cantidad que puede existir simultáneamente. Su comportamiento predecible facilita la detección de colisiones y crea un juego balanceado donde ambos bandos tienen capacidad ofensiva a distancia.

CLASE CRIPTA

Descripción General:

La clase Cripta es una de las más complejas del sistema, encargada de gestionar toda la lógica relacionada con los zombies: su generación, movimiento, dibujado, gestión de aspectos vinculados al combate y a la eliminación de los zombies. Actúa como el "cuartel general" de las fuerzas enemigas.

Variables de Instancia:

zBase[], zAlter[], zColosal: Arrays y objeto para los tres tipos de zombies
lapidas[]: Array para gestionar las lápidas que dejan los zombies al morir
estado, reloj, entorno: Referencias esenciales del juego

conteoBase, conteoAlter: Contadores para el spawn de zombies
zombiesMuertos, zombiesVivos: Estadísticas del juego
regalo1 a regalo5: Estado de los regalos (si fueron robados)
lapidaADisparar: Índice de la lápida objetivo actual

Métodos Principales:

Gestión de Estado:
rendirZombies(): Verifica condición de victoria (50 zombies eliminados)
verZColosal(): Controla el estado del Zombie Colosal
zombiesVictoriosos(): Maneja la animación y lógica de zombies en estado de derrota

Sistema Visual:
dibujarZombies(): Método central que dibuja todos los zombies con animaciones según su estado
dibujarLapidas(): Dibuja las lápidas en el campo de juego
mostrarZombiesVivos(), mostrarZombiesEliminados(): Muestra estadísticas en pantalla

Generación y Spawn:
spawnZombies(): Sistema complejo que genera zombies aleatoriamente con:
Control de tasa de spawn (200 ticks entre generaciones), límite máximo de 15 zombies vivos simultáneamente, probabilidades diferenciadas (ZAlter: 20%, ZBase: 80%), generación del ZColosal después de 25 zombies eliminados

Sistema de Combate:
herirZombieEnPosicion(): Aplica daño a zombies en posición específica
herirZombiesEnRango(): Maneja explosiones de chile en área
herirLapida(): Daña lápidas cuando son disparadas
hayZombieEnPosicion(), hayZombieEnFila(): Detección de colisiones

Mecánicas Avanzadas:

Sistema de Lápidas: Los ZBase tienen 50% de probabilidad de dejar lápidas al morir
Animaciones Diferenciadas: Cada zombie tiene animaciones de caminata, ataque y victoria
Control de Estado: Comportamiento diferente según estado del juego (juego activo vs derrota).
Gestión de Memoria: Eliminación explícita de objetos (no solo ocultarlos)

Problemas Resueltos:

Gestión de Arrays: Uso eficiente de arrays con verificación de null en todos los bucles
Colisiones Precisas: Sistema de detección por posición y área
Balance de Dificultad: Control de spawn rates y límites de zombies simultáneos
Transiciones de Estado: Comportamiento coherente en todos los estados del juego

Funcionalidad:

La Cripta actúa como un sistema autónomo que gestiona toda la inteligencia artificial enemiga, desde la generación estratégica hasta la eliminación controlada, manteniendo un equilibrio entre desafío y jugabilidad.

CLASE JARDÍN

Descripción General:

La clase Jardin es una de las más complejas del sistema, encargada de gestionar todas las plantas defensivas: su plantación, movimiento, dibujado, recursos y lógica de colocación. Actúa como el "centro de operaciones" de las defensas del jugador.

Variables de Instancia:

rosa[], nuez[], chile[]: Arrays para los tres tipos de plantas
bFuego[], bNieve[]: Arrays para proyectiles (gestión en Combate)
menu, cripta: Referencias para coordinación
xs[], ys[]: Arrays que definen la grilla del jardín (10x5 casillas)
abono: Sistema de recursos del jugador (inicia en 100)
aRosa, aNuez, aChile: Estados de selección de plantas
Variables de control para movimiento: plantaSeleccionada, tipoPlantaSeleccionada, moviendoPlanta
Contadores: conteoRosa, conteoNuez, etc.

Métodos Principales:

Gestión Visual:
dibujarRegalos(): Sistema complejo que anima los 5 regalos con diferentes ciclos
dibujarPlantas(): Método masivo que dibuja todas las plantas con:
Animaciones diferenciadas por estado del juego, efectos visuales de selección (marcos amarillos), animaciones de explosión para chiles y estados de victoria/derrota con sprites especiales.

Sistema de Plantación:
spawnPlanta(): Lógica compleja que maneja: Detección de clicks en botones de plantas, conversión de coordenadas mouse a posiciones de grilla, verificación de casillas ocupadas, gestión de costos de abono y prevención de colocación múltiple.

Sistema de Movimiento:
moverPlanta(): Sistema avanzado que permite, selección de plantas con click, movimiento por teclado (WASD/flechas) con sistema de grid, prevención de movimiento a casillas ocupadas, confirmación con click o tecla Q y detección robusta de plantas existentes.

Gestión de Recursos:
crearAbono(): Sistema de generación automática cada 100 ticks
posiblesPlantas(): Calcula plantas disponibles basado en abono actual

Métodos de Utilidad:
hayPlantaEnPosicion(), hayPlantaEnFila(): Detección de colisiones

plantaExiste(): Verificación robusta de plantas seleccionadas
Múltiples métodos auxiliares para conversión coordenadas-mouse

Mecánicas Avanzadas Implementadas:
Sistema de Grid Precisa: 10 columnas × 5 filas con posiciones fijas
Prevención de Errores: Verificación exhaustiva de null en todos los arrays
Feedback Visual: Marcos de selección, animaciones de estado
Gestión de Estados: Comportamiento diferenciado en juego/victoria/derrota
Sistema de Recursos Balanceado: Generación progresiva con límite de 300

Problemas Resueltos:
Colisiones Precisas: Sistema de detección por posición exacta en grid.
Movimiento por Teclado: Implementación fluida con prevención de movimiento repetido.
Gestión de Memoria: Uso eficiente de arrays con verificación de null.
Interfaz de Usuario: Feedback claro de selección y disponibilidad.
Transiciones de Estado: Comportamiento coherente en todos los estados del juego.

Funcionalidad:
El Jardín actúa como un sistema autónomo que gestiona toda la estrategia defensiva del jugador, desde la colocación inicial hasta el reposicionamiento táctico, manteniendo un equilibrio entre simplicidad de uso y profundidad estratégica.

CLASE LAPIDA
Descripción General:
La clase Lapida representa los obstáculos que aparecen en el campo de juego cuando los zombies ZBase son eliminados. Actúa como un elemento de terreno que bloquea la colocación de plantas y puede ser destruido mediante ataques.

Variables de Instancia:
entorno, estado, reloj: Referencias para funcionalidad del juego y posibles expansiones futuras
ejeX, ejeY: Posición en el campo de juego (coordenadas de la grilla)
vida: 100 puntos de resistencia (requiere múltiples ataques para ser destruida)
tipoDeLapida: Entero que permite diferentes tipos de lápidas (actualmente solo tipo 1 implementado)

Métodos Principales:
Lapida(): Constructor que inicializa todos los atributos de la lápida

Funcionalidad:
Las lápidas se generan con un 50% de probabilidad cuando un zombie ZBase es eliminado. Una vez colocadas en el campo:
Bloquean la plantación: Impiden que se coloquen nuevas plantas en esa casilla
Obstaculizan el movimiento: Las plantas no pueden moverse a casillas con lápidas
Pueden ser destruidas: Reciben daño de los proyectiles y explosiones
Ocupan espacio estratégico: Alteran la disposición del campo de batalla

Características de Implementación:
Sistema de Salud: 100 puntos de vida, requiriendo varios impactos para ser destruida
Posicionamiento Preciso: Se ubican en las coordenadas exactas de la grilla del jardín
Tipificación: Diseñada para soportar diferentes tipos de lápidas (aunque actualmente solo se usa tipo 1)
Integración con Combate: Puede ser objetivo de disparos de plantas

Impacto en el Juego:

Las lápidas añaden una capa adicional de estrategia al: forzar al jugador a adaptar sus defensas alrededor de obstáculos, crear cuellos de botella naturales en el campo, requerir gestión de recursos para limpiar áreas estratégicas, añadir elementos de aleatoriedad al campo de batalla.

Problemas Resueltos:

Colocación Inteligente: Verificación de que no se superpongan con plantas existentes
Gestión de Espacio: Integración con el sistema de grilla del jardín
Balance Gameplay: Probabilidad del 50% evita que el campo se sature excesivamente

La Lápida actúa como un elemento de terreno dinámico que evoluciona el campo de batalla a lo largo de la partida, añadiendo profundidad táctica al ataque del enemigo

CLASE COMBATE

Descripción General:

La clase Combate es el núcleo del sistema de batalla del juego, encargada de gestionar todas las interacciones ofensivas y defensivas entre plantas y zombies. Coordina disparos, colisiones, daños y efectos especiales en tiempo real.

Variables de Instancia

jardin, cripta: Referencias a los sistemas de plantas y zombies
estado, entorno, reloj: Componentes esenciales del juego
No necesita arrays propios - trabaja directamente con los arrays de Jardin y Cripta

Métodos Principales:

Sistema de Debilitamiento (debilitamiento()): maneja colisiones cuerpo a cuerpo entre plantas y zombies. Reinicia estados de detención de zombies cada frame, verifica colisiones por proximidad en ejes X e Y, aplica daño diferenciado según tipo de zombie, maneja explosiones de chiles por contacto.

Gestión de Disparos (gestionarDisparos()): Rosas: Crea bolas de fuego cuando hay zombies en su línea de fuego. Zombies: Crea bolas de nieve cuando hay plantas en su línea de ataque.
Control de Cadencia: Usa puedeDisparar() para regular frecuencia.
Gestión de Arrays: Expande dinámicamente arrays de proyectiles al 90% de capacidad

Sistema Visual (dibujarProyectiles()):
Movimiento: Actualiza posición de todos los proyectiles activos
Dibujado: Renderiza bolas de fuego y nieve con sprites diferenciados
Detección de impacto: bolas de fuego → zombies y lápidas, bolas de nieve → plantas
Limpieza: Elimina proyectiles que salen de pantalla

Métodos de Soporte:
aumentarLengthBFuego(), aumentarLengthBNieve(): Expansión dinámica de arrays
herirPlantaADisparar(): Aplica daño a plantas específicas basado en detección previa

Mecánicas de Combate Implementadas:

Sistema de Daño Diferenciado:
ZColosal: 2 puntos de daño (doble que zombies normales)
ZBase/ZAlter: 1 punto de daño por contacto
BolaDeFuego: 20 puntos de daño
BolaDeNieve: 10 puntos de daño

Comportamientos Especiales:
Detención: Zombies se detienen al colisionar con plantas
Explosiones: Chiles detonan al contacto, dañando área completa
Line of Sight: Plantas solo disparan si hay zombies adelante
Priorización: Sistema elige planta específica para daño por proyectil

Gestión de Recursos Avanzada:
Arrays Dinámicos: Expansión automática al 90% de capacidad
Conteo Eficiente: conteoBFuego y conteoBNieve para gestión rápida
Limpieza Activa: Eliminación explícita de proyectiles usados

Problemas Resueltos:
Performance: Verificación de null antes de cada operación
Colisiones Precisas: Umbrales específicos por tipo de interacción
Balance: Daños proporcionales a velocidad y resistencia
Memoria: Expansión controlada de arrays sin desperdicio

Estrategias de Implementación:
Separación de Responsabilidades: Combate solo maneja interacciones, no estado
Detección por Fila: Optimiza búsqueda de objetivos
Sistema de Indexación: indexPlantaADisparar para daño preciso
Gestión de Estado: Comportamiento diferenciado por estado del juego

La clase Combate actúa como el "árbitro" del campo de batalla, asegurando que todas las interacciones ocurran de manera balanceada, eficiente y visualmente coherente, creando un sistema de combate dinámico y estratégico.

IMPLEMENTACIÓN

RELOJ PRINCIPAL

En la clase Juego utilizamos dentro del método tick el método tiempoObjeto de la clase Reloj. Esto nos permite resolver el problema de la coordinación entre estados del juego y gestión temporal. Sin este sistema, las transiciones entre menú, juego y pausa causarían inconsistencias en el tiempo de juego. Mostramos a continuación el método y luego su implementación dentro de tick():

```
    /**
     * Sistema de relojes independientes para diferentes contextos
     * Permite pausar, reanudar y mantener múltiples tiempos simultáneamente
     */
    public void tiempoObjeto(boolean iniciar, boolean parar, boolean continuar, boolean correr) {
        if(iniciar)
            fueraDeJuego = entorno.tiempo();  // Marcar inicio desde cero
        if(parar)
            enPausa = entorno.tiempo();       // Guardar momento de pausa
        if(continuar) {
            enPausa = entorno.tiempo() - enPausa;          // Calcular tiempo en pausa
            fueraDeJuego = fueraDeJuego + enPausa;        // Ajustar tiempo total
        }
        if(correr)
            enJuego = entorno.tiempo() - fueraDeJuego;    // Tiempo neto de juego
    }
```

```
	public void tick(){
		// Procesamiento de un instante de tiempo
		menu.dibujarMenu();

//////////////////GESTION DEL RELOJ/////////////////////////////////////////
		if(estado.esJuego() && estado.estadoAnterior == 1)
			reloj.tiempoObjeto(true, false, false, false);   // Iniciar desde menú
		if(estado.esPausa() && estado.estadoAnterior == 2)
			reloj.tiempoObjeto(false, true, false, false);   // Pausar desde juego
		if(estado.esJuego() && estado.estadoAnterior == 3)
			reloj.tiempoObjeto(false, false, true, false);   // Reanudar desde pausa
		if(estado.esJuego() || estado.esDerrota() || estado.esVictoria())
			reloj.tiempoObjeto(false, false, false, true);   // Correr tiempo normal

		estado.estadoAnterior = estado.getEstado();   // Guardar estado para transiciones

////////////////////////////////////////////////////////////////////////////
//Resto de la lógica del juego
```

GESTIÓN DE ANIMACIONES

El método ciclos() es el corazón de todas las animaciones del juego. Utilizando aritmética modular sobre el tiempo, creamos efectos visuales periódicos sin necesidad de variables de control complejas. Este sistema se utiliza para animar regalos, plantas, zombies y efectos especiales. Mostramos el método y ejemplificamos con un fragmento del método de la clase Jardin dibujarRegalos() donde vamos variando la posición de la imagen de acuerdo al tiempo;

```
	public boolean ciclos(int indice, int divisor) { //Dividimos el tiempo por el divisor
		int resto = getTiempo() % divisor;
		return (resto < indice);
	}
```

```
		if(estado.esJuego() || estado.esVictoria()) {
		if (reloj.ciclos(200, 1200)) {			//1er regalo

entorno.dibujarImagen(Herramientas.cargarImagen("personajes/regalo1.png"), 55, 226, 0);
		}
		else if (reloj.ciclos(400,  1200)) {

entorno.dibujarImagen(Herramientas.cargarImagen("personajes/regalo1.png"), 55, 220, 0);
		}
		else if (reloj.ciclos(600,  1200)) {

entorno.dibujarImagen(Herramientas.cargarImagen("personajes/regalo1.png"), 55, 216, 0);
		}
		else if (reloj.ciclos(800,  1200)) {

entorno.dibujarImagen(Herramientas.cargarImagen("personajes/regalo1.png"), 55, 210, 0);
		}
		else if (reloj.ciclos(1000,  1200)) {
```

```
entorno.dibujarImagen(Herramientas.cargarImagen("personajes/regalo1.png"),
55, 204, 0);
            }
            else {

entorno.dibujarImagen(Herramientas.cargarImagen("personajes/regalo1.png"),
55, 198, 0);
            }
```

Ventajas de este diseño:
Eficiencia: Una sola función maneja todas las animaciones
Sincronización: Todas las animaciones usan el mismo tiempo base
Flexibilidad: Fácil ajuste de velocidad y duración de animaciones
Mantenimiento: Código claro y predecible

GESTIÓN DEL COMBATE

La clase Combate gestiona todas las interacciones ofensivas y defensivas del juego. Implementa un sistema complejo que maneja colisiones cuerpo a cuerpo, disparos a distancia, y efectos especiales.

debilitamiento()

```
        // Reiniciar todos los estados de detención primero
        for(int j = 0; j < cripta.zBase.length; j++) {
            if(cripta.zBase[j] != null && cripta.zBase[j].vivo) {
                cripta.zBase[j].detener = false;
            }
        }
```

```
            for(int j = 0; j < cripta.zBase.length; j++) {
                if(cripta.zBase[j] != null && cripta.zBase[j].vivo) {
                    if(jardin.nuez[i] == null) continue;
                     if(Math.abs(cripta.zBase[j].posY - jardin.nuez[i].posY) <
30 &&
                       cripta.zBase[j].posX - jardin.nuez[i].posX <= 80 &&
                       cripta.zBase[j].posX - jardin.nuez[i].posX > 0) {
                        cripta.zBase[j].detener = true;
                        jardin.nuez[i].vida -= 1;
                        if(jardin.nuez[i].vida <= 0) {
                            jardin.nuez[i] = null;
                        }
                    }
                }
            }
```

```
        for(int i = 0; i < jardin.rosa.length; i++) {
            if (jardin.rosa[i] == null) continue;

            if(cripta.zColosal != null && cripta.zColosal.vivo) {
                if(Math.abs(cripta.zColosal.posX - jardin.rosa[i].posX) <= 150
&&
                   Math.abs(cripta.zColosal.posY - jardin.rosa[i].posY) < 250)
{
                    jardin.rosa[i].vida -= 2; // Más daño del ZColosal
                    if(jardin.rosa[i].vida <= 0) {
                        jardin.rosa[i] = null;
                    }
                }
            }
```

Reinicia los estados de detención de todos los zombies en cada tick. Los estados de detención marcan que el zombie debe frenar y atacar a las plantas dada la proximidad para el combate cuerpo a cuerpo. No sucede la detención para con el zombie colosal, pero este aún así hace daño por proximidad aumentado en relación a los demás:

“Expansión automática de Arrays”

```
// Método auxiliar para aumentar el array de bolas de fuego
public void aumentarLengthBFuego() {
    int length = (int) (jardin.bFuego.length * 1.5);
    BolaDeFuego[] nuevo = new BolaDeFuego[length];
    for (int i = 0; i < jardin.bFuego.length; i++) {
        nuevo[i] = jardin.bFuego[i];
    }
    jardin.bFuego = nuevo;
}
```

Las bolas de fuego son objetos muy variables en su cantidad y muy volátiles. Este método permite ajustar el tamaño de los arrays que las contienen según necesidad.

Gestión de daño por proyectiles a distancia

```
    public void herirPlantaADisparar(int damage) {
          int i = jardin.indexPlantaADisparar;
          String tipo = jardin.tipoPlantaADisparar;
          if(i == -1) return;
          else if(tipo == "rosa") {
                jardin.rosa[i].vida -= damage;
                if (jardin.rosa[i].vida <= 0) jardin.rosa[i].vivo = false;
          }
          else if(tipo == "nuez") {
                jardin.nuez[i].vida -= damage;
                if (jardin.nuez[i].vida <= 0) jardin.nuez[i].vivo = false;
          }
          else if(tipo == "chile") {
                jardin.chile[i].vida -= damage;
                if (jardin.chile[i].vida <= 0) jardin.chile[i] = null;
          }
    }
```

Problemas resueltos por este diseño:

Gestión de memoria: Arrays dinámicos evitan límites fijos arbitrarios. Además los objetos que pierden su función por debilitamiento se convierten en null, liberando posiciones de memoria.

Detección de colisiones: Sistema de umbrales diferenciados por tipo de interacción

Balance de daño: Valores proporcionales a la resistencia de cada entidad

Performance: Verificación de null en todos los bucles previene excepciones

Innovaciones implementadas:

Sistema de detención: Las nueces frenan zombies sin eliminarlos inmediatamente

Linea de visión Plantas y zombies solo disparan si hay objetivos en su línea

Daño diferenciado: ZColosal hace doble daño en área grande

Gestión eficiente: Conteo activo de elementos para optimizar bucles

Este sistema de combate crea un gameplay estratégico donde la colocación de plantas y la gestión de recursos son esenciales para la victoria.

CONCLUSIONES

El desarrollo de "La Invasión de los Zombies Grinch" nos ha permitido aplicar en la práctica los conceptos fundamentales de la programación orientada a objetos, enfrentándonos a desafíos reales de diseño e implementación.

1. Diseño Modular y Separación de Responsabilidades

A lo largo del desarrollo, comprendimos la importancia crítica de diferenciar las responsabilidades de cada clase y definir cuidadosamente sus parámetros en función de su invariante de representación. Inicialmente tentados a crear clases "omnipotentes", aprendimos que un diseño modular no solo facilita el mantenimiento sino que también permite la extensibilidad del código. Cada clase (Jardin, Cripta, Combate, etc.) adquirió un propósito bien definido, aunque reconocemos que el código aún es mejorable en este aspecto.

2. Patrones de Diseño Emergentes

Aunque no utilizamos herencia por ser un contenido más avanzado, identificamos patrones naturales en nuestra implementación. Las clases de zombies (ZBase, ZAlter, ZColosal) y plantas (Rosa, Nuez, Chile) demostraron cómo múltiples entidades con características similares pueden manejarse mediante interfaces conceptuales comunes, innovando solo donde las particularidades de cada una lo requerían. Este enfoque nos permitió mantener consistencia en el código mientras accommodábamos comportamientos específicos.

3. Gestión de Complejidad mediante Arrays

El requisito de utilizar exclusivamente arrays nos forzó a desarrollar soluciones creativas para problemas típicamente resueltos con colecciones más avanzadas. Implementamos sistemas de expansión dinámica, conteo eficiente y gestión de null que, aunque demandaron más esfuerzo inicial, resultaron en un entendimiento más profundo de la gestión de memoria y el rendimiento.

4. Evolución del Diseño frente a Problemas Reales

El proyecto evolucionó significativamente desde su concepción inicial. Problemas como las transiciones entre estados, la detección de colisiones y la gestión temporal nos obligaron a refactorizar continuamente, demostrando que el diseño orientado a objetos es un proceso iterativo donde el contacto directo con los problemas guía las mejoras de la arquitectura.

Resultados Obtenidos vs. Potencial

Logros Implementados:

Sistema de combate completo con múltiples tipos de interacciones

Interfaz de usuario funcional con estados diferenciados

Gestión eficiente de recursos mediante el sistema de abono

Comportamientos enemigos variados y estratégicos

Áreas de Mejora Identificadas:

Si bien contamos con las herramientas para continuar mejorando el proyecto, limitaciones de tiempo nos impidieron implementar características avanzadas como sistemas de sonido,

ataques diferenciados de jefe final, o mecánicas de progresión más elaboradas. Estas quedan como oportunidades de crecimiento para proyectos futuros.

Reflexión sobre el Proceso

El desarrollo de este videojuego nos enseñó que la programación orientada a objetos es tanto sobre prevención de problemas como sobre resolución de los mismos. La inversión inicial en un buen diseño colaboró mucho durante la implementación de características complejas, mientras que las decisiones apresuradas nos costaron mucho tiempo de reelaboración.