DOCUMENTACIÓN DEL PROYECTO: GENETIC KINGDOM

1. INTRODUCCIÓN

El presente documento describe el desarrollo del proyecto "Genetic Kingdom", un juego de estilo Tower Defense ambientado en la época medieval donde se implementan algoritmos genéticos y pathfinding. El juego está desarrollado en C++ utilizando la biblioteca SDL2 para la interfaz gráfica.

Este proyecto integra conceptos avanzados de programación orientada a objetos y algoritmos de inteligencia artificial, permitiendo crear un entorno donde los enemigos evolucionan con cada oleada, aumentando la dificultad progresivamente mientras el jugador construye y mejora sus defensas.

2. DESCRIPCIÓN DEL PROBLEMA

El problema principal consiste en desarrollar un juego de tipo Tower Defense que implemente:

1. **Algoritmos genéticos**: Las oleadas de enemigos deben evolucionar después de cada ronda, mejorando sus características para superar mejor las defensas del jugador.
2. **Algoritmos de pathfinding**: Los enemigos deben encontrar rutas óptimas para llegar al puente del castillo, adaptándose a los obstáculos (torres) que coloca el jugador.
3. **Diseño orientado a objetos**: La solución debe implementarse siguiendo los principios de OOP en C++, con un diseño de clases coherente y modular.

El desafío radica en integrar estos tres componentes de manera efectiva para crear un juego funcional, visualmente atractivo y con mecánicas desafiantes y equilibradas.

3. DESCRIPCIÓN DE LA SOLUCIÓN

A continuación, se detalla cómo se implementó cada uno de los requerimientos del proyecto, junto con las decisiones de diseño, limitaciones y desafíos encontrados durante el proceso de desarrollo.

3.1. Mapa de Juego (Requerimiento 001)

Implementación

El mapa se implementó como una cuadrícula de 12×16 celdas donde cada celda puede ser uno de tres tipos:

- Celda vacía (0): Lugar donde se pueden colocar torres
- Camino (1): Ruta por donde se desplazan los enemigos
- Torre (2): Posición ocupada por una torre

La clase `GameBoard` gestiona el tablero de juego, controlando la generación de caminos, verificando posiciones válidas para colocar torres y renderizando visualmente el mapa. Se implementaron múltiples caminos posibles hacia el puente para dar variedad al juego y opciones estratégicas al jugador.

```
GameBoard::GameBoard(int r, int c) : rows(r), cols(c) {
    // Inicializar el tablero con celdas vacías
    grid.resize(rows, std::vector<int>(cols, 0));

    // Definir entrada y salida
    entrance = {0, r/2};
    exit = {c-1, r/2};

    // Inicializar el mapa con múltiples caminos
    initializeMap();
}
```

El sistema verifica constantemente que al colocar una torre no se bloqueen todos los caminos posibles mediante un algoritmo de búsqueda (BFS) que comprueba la conectividad entre entrada y salida:

```
bool GameBoard::isValidTowerPosition(int r, int c) const {
    // ...
    // Simular colocación para verificar si bloquearía todos los caminos
    std::vector<std::vector<int>> tempGrid = grid;
```

```
    tempGrid[r][c] = 2; // Colocar torre temporalmente

    // Crear tablero temporal para verificar camino
    GameBoard tempBoard = *this;
    tempBoard.grid = tempGrid;

    // Verificar si hay un camino válido después de colocar la torre
    return tempBoard.hasValidPath();
}
```

Alternativas Consideradas

1. **Mapa predefinido**: Se consideró crear un mapa estático con caminos fijos, pero se descartó por limitar la rejugabilidad.
2. **Generación procedural completa**: Se evaluó generar todo el mapa aleatoriamente, pero esto dificultaba el balance del juego.

Limitaciones y Problemas Encontrados

- La representación visual del mapa es básica, utilizando colores para diferenciar los tipos de celdas.
- El tamaño fijo de la cuadrícula puede resultar limitante para ciertos niveles de dificultad.
- Inicialmente surgieron problemas para garantizar múltiples caminos válidos simultáneamente.

3.2. Atributos de Torres (Requerimiento 002)

Implementación

Las torres se implementaron mediante una jerarquía de clases donde `Tower` es la clase base abstracta que define los atributos comunes:

```
class Tower {
protected:
    int level;          // Nivel actual de la torre (1-3)
    int damage;         // Daño base
    int range;          // Alcance en píxeles
    int attackSpeed;    // Velocidad de ataque (ms entre ataques)
```

```
    int specialCooldown; // Tiempo de regeneración del poder especial
    int row, col;       // Posición en el tablero
    // ...
};
```

Cada torre gestiona sus propios temporizadores de ataque y ataque especial. El sistema detecta enemigos dentro del rango y aplica el daño correspondiente considerando las resistencias de cada tipo de enemigo.

El sistema de recompensa de oro se diseñó para ser equilibrado, donde el oro retornado está en función de:

- Tipo de enemigo (los más difíciles otorgan más oro)
- Salud base del enemigo
- Velocidad del enemigo (enemigos más rápidos suelen dar más oro)

Alternativas Consideradas

1. **Torres con recursos agotables**: Se evaluó implementar munición limitada para las torres, pero se descartó por complejidad adicional.
2. **Torres con cooldown global**: Se consideró un sistema donde todas las torres compartieran un cooldown, pero reducía las opciones estratégicas.

Limitaciones y Problemas Encontrados

- Surgieron dificultades para equilibrar los valores de daño y alcance entre los diferentes tipos de torre.
- Inicialmente las torres atacaban a través de obstáculos; se corrigió implementando verificación de línea de visión.
- El sistema de detección de colisiones requirió ajustes para mayor precisión.

3.3. Sistema de Mejoras de Torres (Requerimiento 003)

Implementación

Se implementó un sistema de mejoras con 3 niveles por torre. Cada mejora incrementa el daño en un 30%, reduce el tiempo de recarga en un 10% y aumenta ligeramente el alcance:

```
bool Tower::upgrade() {
    // Solo puede mejorarse hasta nivel 3
    if (level >= 3) {
        return false;
    }

    // Actualiza nivel y aumenta estadísticas
    level++;
    damage += damage * 0.3; // 30% más de daño
    range += 10; // Aumenta ligeramente el alcance
    attackSpeed -= attackSpeed * 0.1; // 10% más rápido

    // Mejorar ataque especial
    specialAttackProbability += 0.1f; // Mayor probabilidad
    specialCooldown -= specialCooldown * 0.1f; // Menor tiempo de recarga

    return true;
}
```

Los ataques especiales se implementaron con un sistema de probabilidad y cooldown, donde cada nivel de mejora incrementa la probabilidad de activación y reduce el tiempo de recarga:

- **Arqueros**: "Lluvia de flechas" que inflige triple daño
- **Magos**: "Explosión arcana" que causa daño de área (1.5x de daño en radio)
- **Artilleros**: "Proyectil aturdidor" con 1.2x de daño y efecto de aturdimiento

Alternativas Consideradas

1. **Árbol de habilidades**: Se evaluó un sistema más complejo donde el jugador eligiera la dirección de mejora, pero se descartó por complejidad de implementación.
2. **Mejoras aleatorias**: Se consideró incluir mejoras con efectos aleatorios, pero reducía el control estratégico del jugador.

Limitaciones y Problemas Encontrados

- El sistema visual de indicación de nivel de mejora es simple (líneas en la parte superior de la torre).

- Se enfrentaron desafíos para implementar los efectos de área del mago y el aturdimiento del artillero.
- Balancear el costo incremental de las mejoras requirió varios ajustes para mantener la economía del juego equilibrada.

3.4. Tipos de Torres (Requerimiento 004)

Implementación

Se implementaron tres tipos de torres con características distintivas:

1. **Arqueros** (`ArcherTower`):
   - Bajo daño (10)
   - Alto alcance (300 píxeles)
   - Ataque rápido (500ms)
   - Bajo costo (25 oro)
2. **Magos** (`MageTower`):
   - Daño medio (20)
   - Alcance medio (100 píxeles)
   - Velocidad de ataque media (1000ms)
   - Costo medio (50 oro)
3. **Artilleros** (`ArtilleryTower`):
   - Alto daño (40)
   - Bajo alcance (80 píxeles)
   - Ataque lento (2000ms)
   - Alto costo (75 oro)

Cada tipo de torre está definido como una clase derivada de la clase base `Tower` , implementando comportamientos específicos mediante polimorfismo.

Alternativas Consideradas

1. **Sistema basado en composición**: Se evaluó implementar torres como componentes ensamblables, pero se optó por herencia para mantener claridad en el código.

2. **Torres especializadas por enemigo**: Se consideró tener torres con bonificaciones contra tipos específicos de enemigos, pero se descartó para mantener el sistema más comprensible.

Limitaciones y Problemas Encontrados

- Balancear las características de cada torre requirió múltiples iteraciones.
- La representación visual de los diferentes tipos de torres se mejoró con sprites básicos.
- La interfaz de selección y colocación de torres requirió ajustes para mayor claridad.

3.5. Colocación de Torres (Requerimiento 005)

Implementación

Se desarrolló un sistema de interfaz donde el jugador primero selecciona el tipo de torre desde un panel y luego selecciona la posición en el mapa. La clase `TowerManager` gestiona este proceso:

```
bool TowerManager::createTower(int row, int col) {
    // Comprobamos que haya un tipo seleccionado
    if (selectedType == TowerType::NONE) {
        return false;
    }

    // Determinar el costo basado en el tipo
    int cost = 0;
    switch (selectedType) { /* ... */ }

    // Verificar si hay suficiente oro
    if (!resources->spendGold(cost)) {
        std::cout << "¡No hay suficiente oro para esta torre!" << std::endl;
        return false;
    }

    // Crear la torre del tipo seleccionado
    switch (selectedType) { /* ... */ }
```

```
    return true;
}
```

La interfaz muestra claramente:

- Tipos de torre disponibles
- Costo de cada tipo
- Posiciones válidas para colocación
- Torres ya colocadas

Alternativas Consideradas

1. **Modo de construcción avanzado**: Se evaluó un sistema con arrastre y previsualización, pero se optó por clicks para simplificar.
2. **Colocación automática sugerida**: Se consideró implementar sugerencias de colocación óptima, pero se descartó para mantener el desafío estratégico.

Limitaciones y Problemas Encontrados

- La interfaz de usuario es funcional pero básica visualmente.
- Se tuvo que mejorar el feedback visual para indicar posiciones no válidas.
- La selección de torres existentes para mejora requirió ajustes en la detección de clicks.

3.6. Sistema de Enemigos (Requerimiento 006)

Implementación

Se implementaron cuatro tipos de enemigos con atributos diferenciados:

1. **Ogros**:
    - Alta vida (150 HP)
    - Baja velocidad (20 píxeles/s)
    - Alta resistencia a flechas (0.7)
    - Baja resistencia a magia (0.2)
    - Baja resistencia a artillería (0.3)

2. **Elfos Oscuros**:
    - Baja vida (80 HP)
    - Alta velocidad (60 píxeles/s)
    - Baja resistencia a flechas (0.2)
    - Alta resistencia a magia (0.7)
    - Baja resistencia a artillería (0.3)
3. **Harpías**:
    - Muy baja vida (70 HP)
    - Velocidad media (45 píxeles/s)
    - Resistencia media a flechas (0.3)
    - Resistencia media a magia (0.3)
    - Inmunidad a artillería (1.0)
4. **Mercenarios**:
    - Vida media (100 HP)
    - Velocidad media (35 píxeles/s)
    - Alta resistencia a flechas (0.6)
    - Baja resistencia a magia (0.2)
    - Alta resistencia a artillería (0.6)

Cada enemigo implementa un sistema de resistencias que afecta al daño recibido:

```
int Enemy::takeDamage(int damage, std::string towerType) {
    float resistance = 0.0f;

    // Aplicar resistencia según el tipo de torre
    if (towerType == "Arquero") {
        resistance = arrowResistance;
    } else if (towerType == "Mago") {
        resistance = magicResistance;
    } else if (towerType == "Artillero") {
        resistance = artilleryResistance;
```

```
    }

    // Calcular daño real después de aplicar resistencia
    int actualDamage = static_cast<int>(damage * (1.0f - resistance));

    // Aplicar daño
    health -= actualDamage;
    if (health < 0) health = 0;

    return actualDamage;
}
```

Alternativas Consideradas

1. **Sistema con debilidades en lugar de resistencias**: Se evaluó implementar multiplicadores de daño positivos, pero se optó por resistencias para mayor claridad.
2. **Habilidades especiales de enemigos**: Se consideró que algunos enemigos tuvieran habilidades activas, pero se descartó por complejidad.

Limitaciones y Problemas Encontrados

- La representación visual de los enemigos comenzó con colores simples y se mejoró con sprites básicos.
- Hubo dificultades para implementar la inmunidad de las harpías a la artillería de manera consistente.
- El balance de las resistencias requirió múltiples iteraciones para lograr un desafío adecuado.

3.7. Algoritmo Genético (Requerimiento 007)

Implementación

El algoritmo genético se implementó en la clase `GeneticAlgorithm` que gestiona:

- Población de genomas (atributos de enemigos)
- Cálculo de fitness basado en rendimiento
- Selección de padres mediante torneos

- Cruce de genomas con interpolación
- Mutación de atributos
- Evolución entre oleadas

Los genomas contienen:

```
struct Genome {
    float health;
    float speed;
    float arrowResistance;
    float magicResistance;
    float artilleryResistance;
    int enemyType;  // 0=Ogro, 1=Elfo Oscuro, 2=Harpía, 3=Mercenario
    float fitness;
};
```

El fitness se calcula considerando tres factores:

1. Progreso en el camino (mayor peso)
2. Daño causado al jugador
3. Tiempo de supervivencia

```
void GeneticAlgorithm::updateFitness(Genome& genome, float progressMa
de, float damageDealt, float timeAlive) {
    float fitness = progressMade * 10.0f + (damageDealt / 100.0f) + (timeAliv
e / 1000.0f);
    genome.fitness = std::max(0.0f, fitness);
}
```

La evolución se realiza mediante:

- Selección elitista (conservar mejores individuos)
- Selección por torneo para padres
- Cruce con interpolación entre atributos
- Mutación con probabilidad configurable

```
Genome GeneticAlgorithm::crossover(const Genome& parent1, const Geno
me& parent2) {
    Genome child;

    // Distribución uniforme para el crossover
    std::uniform_real_distribution<float> dist(0.0f, 1.0f);

    // Decidir si hay cruce
    if (dist(rng) < crossoverRate) {
        // Cruce de punto único para tipo
        child.enemyType = (dist(rng) < 0.5f) ? parent1.enemyType : parent2.en
emyType;

        // Cruce de atributos numéricos: interpolación
        float alpha = dist(rng);  // Factor de mezcla

        child.health = parent1.health * alpha + parent2.health * (1.0f - alpha);
        child.speed = parent1.speed * alpha + parent2.speed * (1.0f - alpha);
        child.arrowResistance = parent1.arrowResistance * alpha + parent2.arr
owResistance * (1.0f - alpha);
        child.magicResistance = parent1.magicResistance * alpha + parent2.m
agicResistance * (1.0f - alpha);
        child.artilleryResistance = parent1.artilleryResistance * alpha + parent
2.artilleryResistance * (1.0f - alpha);
    } else {
        // Sin cruce: copiar uno de los padres directamente
        child = dist(rng) < 0.5f ? parent1 : parent2;
    }

    return child;
}
```

Alternativas Consideradas

1. **Algoritmo NEAT**: Se evaluó implementar neuroevolución para comportamientos más complejos, pero se descartó por su complejidad.

2. **Sistema de reglas**: Se consideró un sistema basado en reglas predefinidas, pero limitaba la adaptabilidad.

Limitaciones y Problemas Encontrados

- El mayor desafío fue el cálculo correcto del fitness, que inicialmente no reflejaba adecuadamente el éxito de los enemigos.
- Se enfrentaron problemas para mantener la diversidad de tipos en la población.
- El balance entre mutación y selección requirió ajustes para evitar convergencia prematura.
- Se tuvo que implementar un sistema para transferir correctamente el fitness entre enemigos y genomas.

3.8. Pathfinding A* (Requerimiento 008)

Implementación

Se implementó el algoritmo A* para que los enemigos encontraran caminos óptimos hacia el puente del castillo. La clase `AStar` encapsula esta funcionalidad:

```
class AStar {
    // ...
public:
    static std::vector<SDL_Point> findPath(
        const std::function<bool(int, int)>& isWalkable,
        SDL_Point start, SDL_Point end, int width, int height);

private:
    static int calculateHCost(SDL_Point a, SDL_Point b);
    static std::vector<SDL_Point> reconstructPath(...);
};
```

El algoritmo:

1. Utiliza una cola de prioridad para seleccionar nodos
2. Calcula costos G (distancia desde inicio) y H (heurística)
3. Explora vecinos en 4 direcciones

4. Reconstruye el camino óptimo
5. Considera obstáculos (torres)

Cuando se coloca una nueva torre, los enemigos recalculan sus rutas:

```
void Enemy::recalculatePath(GameBoard* board) {
    if (!board || path.empty()) return;

    // Convertir posición actual a coordenadas de grid
    int gridSize = 50;
    int currentGridX = static_cast<int>(x) / gridSize;
    int currentGridY = static_cast<int>(y) / gridSize;

    // Obtener punto final (destino)
    SDL_Point finalGridPoint = {
        path.back().x / gridSize,
        path.back().y / gridSize
    };

    // Función lambda para verificar si una celda es caminable
    auto isWalkable = [board](int x, int y) {
        return board->isCellWalkable(x, y);
    };

    // Recalcular camino con A*
    std::vector<SDL_Point> newGridPath = AStar::findPath(
        isWalkable, {currentGridX, currentGridY}, finalGridPoint,
        board->getCols(), board->getRows());

    // Convertir a coordenadas de píxeles y establecer nuevo camino
    // ...
}
```

Alternativas Consideradas

1. **Algoritmo Dijkstra**: Se evaluó utilizar Dijkstra, pero A* ofrece mejor rendimiento con su heurística.

2. **Navmesh**: Se consideró implementar mallas de navegación, pero excedía los requisitos del proyecto.

Limitaciones y Problemas Encontrados

- La implementación inicial solo utilizaba heurística de Manhattan; se mejoró la precisión.
- Hubo desafíos para gestionar múltiples enemigos recalculando rutas simultáneamente.
- La interacción entre la colocación de torres y la validación de caminos requirió optimización.
- Se tuvo que implementar la regeneración de caminos alternativos para mantener opciones viables.

3.9. Panel de Estadísticas (Requerimiento 00G)

Implementación

Se desarrolló un panel de estadísticas en la esquina inferior izquierda que muestra:

- Generación actual
- Oleada actual
- Estadísticas de fitness (promedio, mejor, peor)
- Nivel de torres
- Probabilidad de mutación y mutaciones ocurridas
- Número de enemigos activos

La información se muestra en tiempo real durante el juego:

```
void Game::renderUI() {
    // ...
    // Mostrar estadísticas genéticas
    if (font) {
        SDL_Rect geneticRect = {10, SCREEN_HEIGHT - 140, 400, 130};
        SDL_SetRenderDrawColor(renderer, 50, 50, 50, 200);
        SDL_RenderFillRect(renderer, &geneticRect);
```

```
        // Renderizar textos informativos para cada estadística
        // ...
    }
}
```

Alternativas Consideradas

1. **Estadísticas desplegables**: Se evaluó mostrar estadísticas solo bajo demanda, pero se prefirió visibilidad constante.
2. **Gráficos de evolución**: Se consideró incluir gráficas de tendencias, pero excedía los requisitos básicos.

Limitaciones y Problemas Encontrados

- El contador específico de enemigos muertos por oleada está implementado de forma básica sin mostrar históricos.
- La representación visual del panel es funcional pero podría mejorar estéticamente.
- La visualización de todos los datos requirió optimización para no sobrecargar la interfaz.

4. ASPECTOS ADICIONALES DE IMPLEMENTACIÓN

4.1. Sistema de Mensajes de Ataque

Se implementó un sistema para mostrar mensajes de ataque en tiempo real, que informa al jugador sobre:

- Tipo de torre que ataca
- Tipo de enemigo atacado
- Daño causado
- Vida restante del enemigo
- Activación de ataques especiales

```
void Game::addAttackMessage(const std::string& text, const SDL_Color& c
olor) {
    AttackMessage message;
    message.text = text;
```

```
    message.timeToLive = 2000;  // 2 segundos
    message.color = color;

    attackMessages.push_front(message);  // Añadir al principio

    // Limitar número de mensajes
    if (attackMessages.size() > MAX_MESSAGES) {
        attackMessages.pop_back();  // Eliminar el más antiguo
    }
}
```

Este sistema mejora significativamente el feedback visual para el jugador, permitiéndole entender mejor las interacciones entre torres y enemigos.

4.2. Sistema de Recursos

El sistema de recursos (`ResourceSystem`) gestiona la economía del juego:

- Oro inicial para el jugador
- Costo de construcción de torres
- Costo de mejoras
- Recompensas por eliminar enemigos

```
class ResourceSystem {
private:
    int gold;

public:
    ResourceSystem(int initialGold = 100);

    int getGold() const;
    void addGold(int amount);
    bool spendGold(int amount);
};
```

Este sistema es crucial para el balance del juego, permitiendo decisiones estratégicas sobre cuándo construir, qué tipo de torre priorizar, y cuándo mejorar.

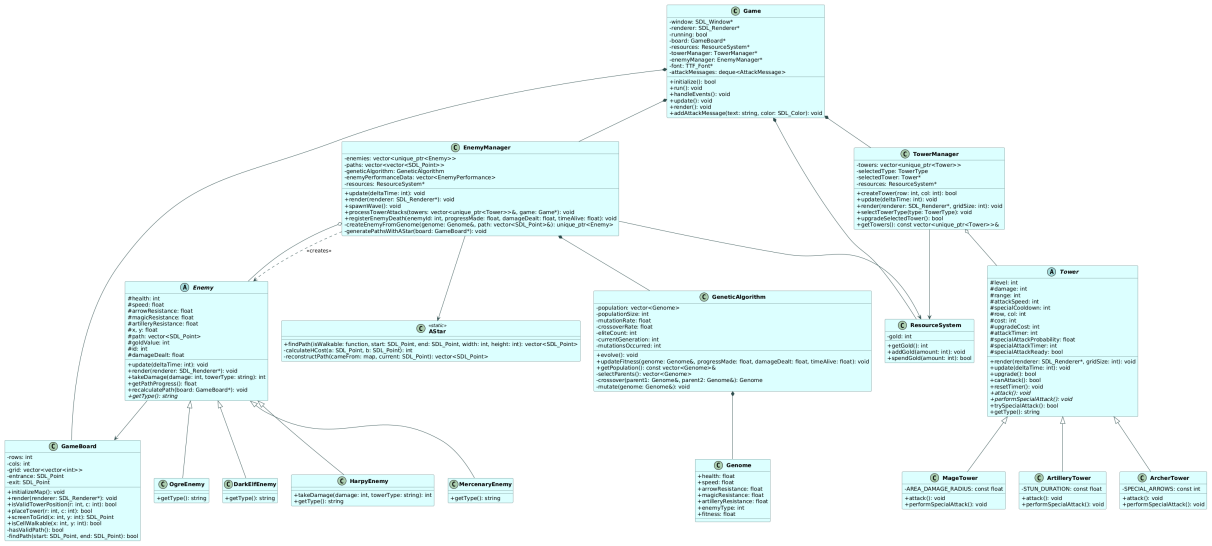
4.3. Visualización de Torres y Enemigos

Se implementó un sistema visual que incluye:

- Texturas diferenciadas para cada tipo de torre y enemigo
- Indicador de nivel de mejora para las torres
- Visualización del rango de ataque (círculo semitransparente)
- Barras de vida para los enemigos (color dinámico según porcentaje)

Estas mejoras visuales incrementan significativamente la claridad del juego para el usuario.

5.Diagrama UML



6. SETUP Y EJECUCIÓN DEL PROYECTO

6.1. Ambiente de Desarrollo

El proyecto "Genetic Kingdom" ha sido desarrollado en un entorno Linux utilizando las siguientes herramientas y bibliotecas:

- **Sistema Operativo**: Ubuntu 22.04 LTS / Debian 11
- **Compilador**: g++ 11.2.0 con soporte para C++17
- **Bibliotecas principales**:
  - SDL2 (2.0.20) - Para la gestión de ventanas y renderizado
  - SDL2_image (2.0.5) - Para cargar texturas e imágenes

- SDL2_ttf (2.0.18) - Para renderizado de texto

A pesar de utilizar un entorno Linux para el desarrollo, SDL2 es una biblioteca multiplataforma, lo que permite compilar y ejecutar el juego en otros sistemas operativos con modificaciones mínimas.

6.2. Instalación de Dependencias

Para instalar las dependencias necesarias en un sistema basado en Debian/Ubuntu, ejecute:

```
sudo apt update
sudo apt install build-essential
sudo apt install libsdl2-dev libsdl2-image-dev libsdl2-ttf-dev
```

Para otras distribuciones Linux, consulte el gestor de paquetes correspondiente.

6.3. Estructura del Proyecto

El proyecto está organizado de la siguiente manera:

```
genetic_kingdom/
├── Makefile
├── src/
│   ├── main.cpp
│   ├── Game.cpp
│   ├── Game.h
│   ├── GameBoard.cpp
│   ├── GameBoard.h
│   ├── Tower.cpp
│   ├── Tower.h
│   ├── ... (otros archivos .cpp y .h)
├── fonts/
│   └── arial.ttf
├── images/
```

```
│   ├── arquero.png
│   ├── mago.png
│   ├── artillero.png
│   ├── ogro.png
│   ├── elfo.png
│   ├── harpia.png
│   └── mercenario.png
└── README.md
```

6.4. Compilación del Proyecto

El proyecto utiliza un Makefile para facilitar la compilación. Para compilar el juego, siga estos pasos:

1. Abra una terminal en el directorio raíz del proyecto
2. Ejecute el comando `make` para compilar:

```
make
```

3. El comando generará el ejecutable `genetic_kingdom` en el directorio raíz

Si desea limpiar los archivos objeto y el ejecutable, puede utilizar:

```
make clean
```

6.5. Ejecución del Juego

Para iniciar el juego, desde el directorio raíz del proyecto, ejecute:

```
./genetic_kingdom
```

Alternativamente, puede usar el comando específico incluido en el Makefile:

```
make run
```

6.6. Controles e Interfaz

Una vez iniciado, el juego muestra el mapa principal con una cuadrícula donde:

- **Parte superior**: Panel de selección de torres y recursos
- **Centro**: Mapa de juego (cuadrícula)
- **Parte inferior izquierda**: Panel de estadísticas genéticas
- **Parte derecha**: Panel de mensajes de ataque

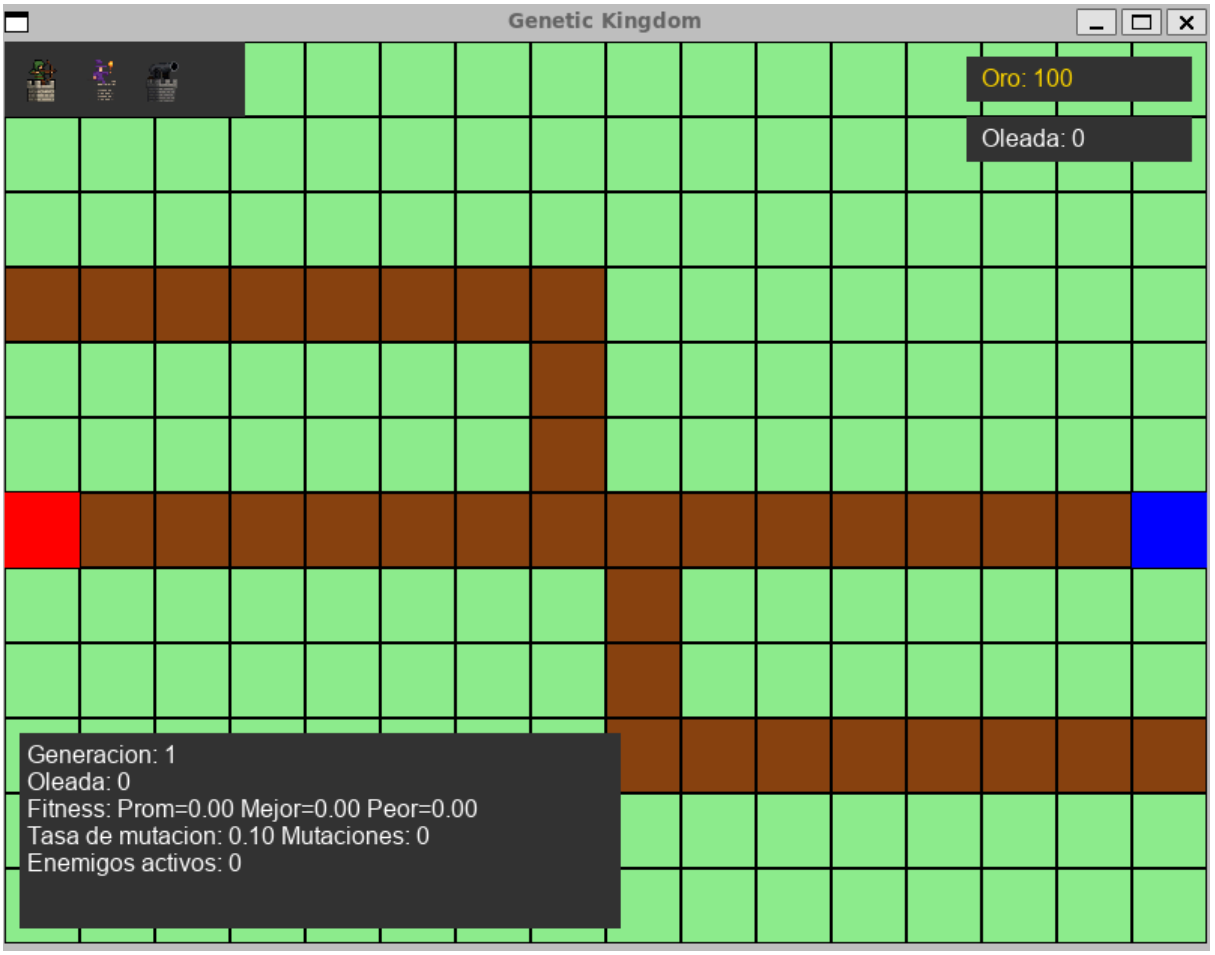
Los controles básicos son:

- **Click izquierdo en panel superior**: Seleccionar tipo de torre
- **Click izquierdo en casilla vacía**: Colocar torre seleccionada
- **Click izquierdo en torre existente**: Seleccionar para mejora
- **Click en botón de mejora**: Mejorar torre seleccionada (si hay oro suficiente)
- **Tecla Espacio**: Generar enemigos de prueba (en modo de prueba)
- **Tecla W**: Generar nueva oleada manualmente (en modo de prueba)

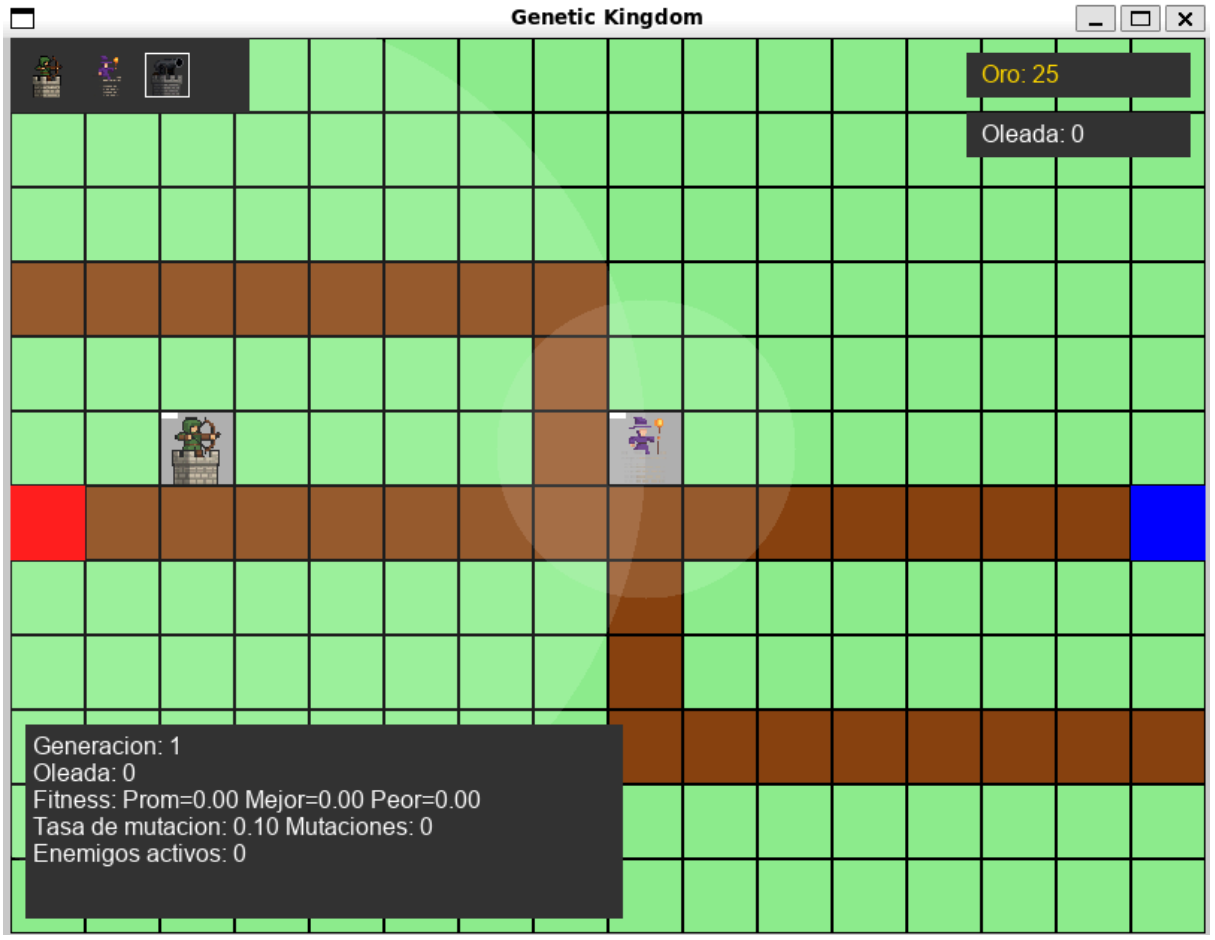
6.7. Capturas de Pantalla Recomendadas

Para documentar visualmente el juego, se recomienda incluir las siguientes capturas de pantalla:

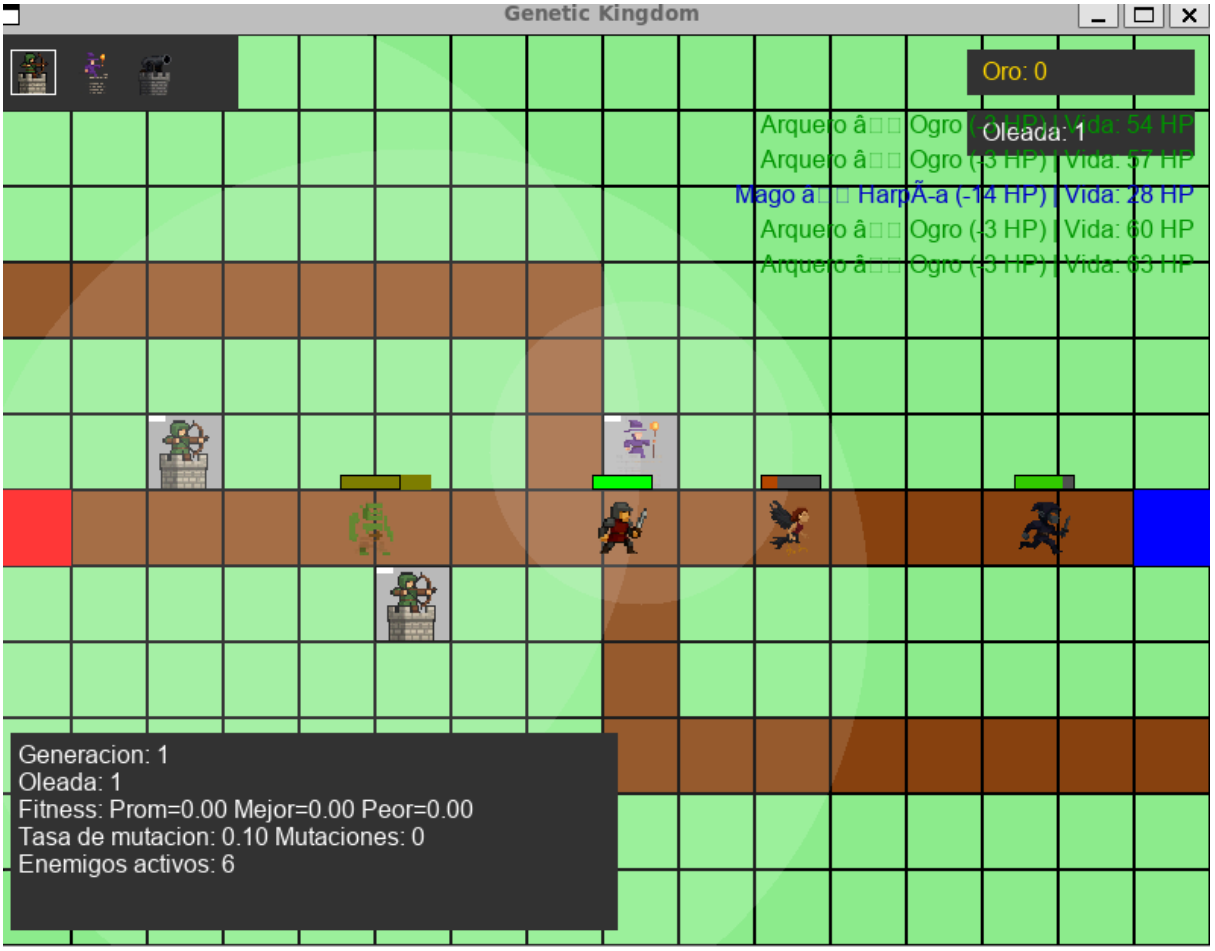
- **Interfaz principal del juego**: Captura mostrando el mapa completo, panel de selección de torres y estadísticas.



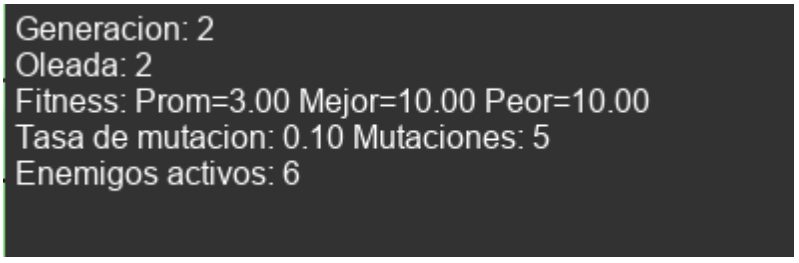
- **Colocación de torres y caminos**: Imagen que muestre varias torres colocadas y los caminos que siguen los enemigos.



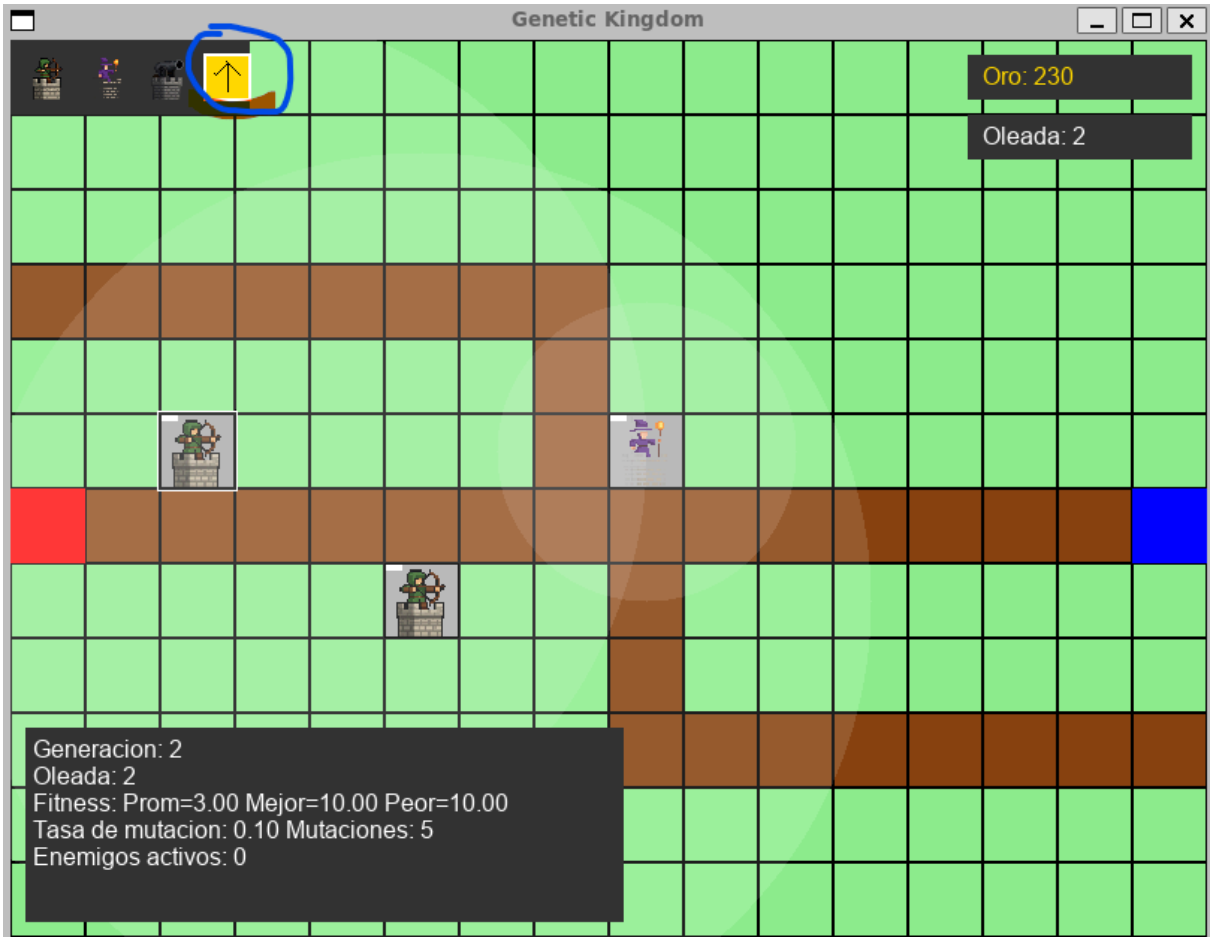
- **Combate en acción**: Captura durante un ataque, mostrando torres atacando a los enemigos con mensajes de ataque visibles.



- **Panel de estadísticas genéticas**: Acercamiento al panel que muestra la evolución de los enemigos.



- **Mejora de torres**: Imagen mostrando el proceso de selección y mejora de una torre.



7. CONCLUSIONES

El desarrollo de "Genetic Kingdom" ha sido un desafío técnico que ha permitido implementar y combinar algoritmos avanzados como A* y algoritmos genéticos en un contexto de juego interactivo. Las principales lecciones y logros incluyen:

1. **Integración efectiva de algoritmos complejos**: Se ha logrado incorporar A* para pathfinding y algoritmos genéticos para la evolución de enemigos de manera cohesiva.
2. **Diseño orientado a objetos**: La implementación utiliza conceptos avanzados de OOP como herencia, polimorfismo, y encapsulamiento para crear un código modular y mantenible.
3. **Balance de juego**: Se han ajustado cuidadosamente los parámetros de torres, enemigos y economía para crear una experiencia desafiante pero justa.
4. **Interfaces de usuario funcionales**: Se ha desarrollado un sistema de interacción que permite al jugador entender claramente el estado del juego

y tomar decisiones estratégicas.

8. ENLACE AL REPOSITORIO DE GITHUB

https://github.com/ned8120/GeneticKingdom