Algoritmos y Estructuras de Datos II

Proyecto II

Genetic-Kingdom

Profesor:

Leonardo Araya M.

Estudiantes:

Fiorela Sofía González Rubí | 2024211034

Paula Melina Porras Mora | 2023082886

Guillermo Sánchez Cedeño | 2024132555

I semestre, 2025

Tabla de contenidos

Introducción

Este documento presenta la documentación del proyecto Genetic-Kingdom, desarrollado como parte del curso Algoritmos y Estructuras de Datos II. El objetivo principal del proyecto es implementar un juego estilo *Tower Defense*, aplicando los conocimientos adquiridos en el curso, especialmente en el diseño de algoritmos.

En esta documentación se detallan los aspectos clave del desarrollo, incluyendo una descripción del problema y la solución que se ha propuesto. Se explican las decisiones de implementación tomadas para cada requerimiento, las alternativas consideradas durante el proceso de desarrollo, así como las limitaciones y los problemas encontrados.

También se incluye un diagrama de clases UML que representa el diseño general del sistema y finalmente, se proporciona un enlace al repositorio de GitHub donde se encuentra el código fuente del proyecto.

Descripción del problema

El proyecto Genetic-Kingdom desarrolla un videojuego de estrategia estilo *Tower Defense*. En el juego, el jugador debe evitar que las oleadas de enemigos logren llegar al castillo ubicado del otro lado del mapa, utilizando torres de defensa que debe ir colocando en el mapa haciendo uso de oro. Cada torre tiene atributos de daño, alcance, velocidad de ataque y habilidades especiales, y el jugador adquiere oro durante la partida que además de ser utilizado para colocar torres, se puede utilizar también para mejorar sus torres.

El principal reto del proyecto es la evolución y mutación de cada oleada de enemigos de los 4 tipos (ogros, elfos, arpías y mercenarios), que se vuelven más fuertes con el tiempo mediante la implementación de un algoritmo genético. Este algoritmo selecciona a los enemigos con mayor fitness de una generación, cruza sus características y genera nuevos enemigos con probabilidades de mutación.

Además, los enemigos deben ser capaces de encontrar mejor camino posible hacia el castillo utilizando el algoritmo de búsqueda A*. El jugador debe ir colocando sus torres estratégicamente para maximizar el daño y minimizar las posibilidades de vencer de los enemigos y a la vez asegurarse de no bloquear por completo los caminos posibles.

Finalmente, en el juego se deben mostrar estadísticas en tiempo real, incluyendo datos sobre las oleadas, el desempeño de las torres y el proceso de evolución de los enemigos. Todo esto implica un desarrollo que combina algoritmos genéticos y de búsqueda, estructuras de datos, programación orientada a objetos y la interfaz gráfica, el proyecto es implementado en el lenguaje de programación C++.

Descripción de la solución

001: Generación del mapa y posibilidad de colocar torres sin bloquear completamente el camino hacia el castillo.

El mapa que decidimos generar en nuestra solución está conformado de una estructura y una clase, la estructura denominada Tile y la clase TileMap. Para realizarlo usamos TileMap como un controlador que en primera instancia, generará un vector bidimensional que, adentro contiene estructuras Tile, en su constructor definiremos puntos claves que afectan de forma positiva después en la implementación de los distintos requisitos, como los atributos goal y start, que definen un punto “x” y “y” del vector en donde uno es el punto final a donde llegarán los enemigos y otro es de donde saldrán.

Esto se podría haber implementado de forma estática, es decir programar a mano y definiendo diferentes caminos sobre la matriz, lo que reduciría en gran medida la complejidad del resultado final, pero afecta en gran parte a la escalabilidad, puesto a que es algo muy rígido. Usando valores binarios en cada posición de la matriz se le puede indicar al enemigo donde se puede mover, que es útil en los próximos requisitos.

002: Torres con atributos definidos que otorgan oro al jugador al eliminar enemigos.

Las torres están hechas a partir del polimorfismo y la herencia, donde hay una clase central llamada “tower” que controlará gran parte del comportamiento de éstas y cuenta con algunos métodos que pueden ser sobreescritos, lo que es de gran ayuda cuando decidimos crear varios tipos de torres con distintas categorías de ataques especiales.

Usando los conceptos adquiridos en diversos cursos anteriores se decidió que, para instanciar cada objeto diferente de “tower” lo más conveniente es usar la Arquitectura Factory, que nos construye fácilmente las torres que necesitemos en el momento que el usuario decida desplegarlas en el campo.

El oro se entrega al usuario mediante el cociente de su vida máxima entre dos y en cada frame se revisa la lista de enemigos que se encuentran en el grid del juego, para así determinar si es que están muertos o aún poseen vida restante.

003: Implementación de upgrades en las torres.

Las torres se actualizan todas al mismo tipo, entonces tener un atributo separado para cada torre se vuelve absurdo, es por ello que se pensó como una implementación inteligente el uso de una clase que maneje toda la información necesaria que uno de estos objetos necesita, como su daño. La torre solo almacena un puntero al objeto

“towerUpgrades” en donde ella va estar consultando su daño para realizar el próximo ataque.

004: Diferenciación entre tipos de torres: arqueros, magos y artilleros, cada uno con atributos únicos.

Como se habló en un párrafo anterior, las torres aplican los conceptos de polimorfismo y herencia que las clases nos brindan, en donde se implementan distintos métodos ya sean “virtuales” o propios de la clase. El único método virtual que podríamos señalar es el referente al ataque especial, que varía de torre a torre, las otras características englobadas en este requisito pueden llegar a ser implementadas de forma genérica para las demás clases que hereden de la principal.

Entonces, se usó la clase “towerFactory”, una arquitectura muy conveniente para nuestro objetivo pues permite crear las distintas torres que el usuario pueda solicitar al momento de jugar y es muy positivo pues en términos de escalabilidad, potencia el proyecto, haciendo que el implementar una torre sea relativamente fácil.

005: Posibilidad de que el jugador coloque torres del tipo que desee en las posiciones disponibles del mapa.

Antes de colocar cualquier tipo de entidad que obstaculice el camino de los enemigos se consulta al pathFinder si aún existe una ruta la cual seguir, en caso de que no haya ninguna, se le niega al usuario la colocación de la torre.

006: Aparición de enemigos por oleadas, de distintos tipos (ogros, elfos oscuros, harpías, mercenarios) y atributos específicos.

La generación de los enemigos se realiza por medio de la clase “EnemyManager”, en la cual se obtiene el control de todos los enemigos, en cada oleada se instancia una lista de enemigos que toma de “EnemyGenome” atributos genéticos como vida, velocidad y resistencia a las distintas torres.

Estos enemigos están definidos mediante “enum class”, cada uno tiene una clase propia que hereda de “Enemy” siendo la lógica base para estas subclases con sprites propios. El recorrido de cada enemigo se calcula mediante el algoritmo pathfinding A* y su ruta se actualiza dinámicamente (como se menciona en el punto 008).

Una limitación inicial fue mantener la consistencia entre el tipo de enemigo y su representación visual y lógica; para resolver esto se usó un switch sobre “genome.type” para crear la instancia correcta

007: Algoritmo genético para evolucionar enemigos entre oleadas.

Para la implementación de la evolución de los enemigos entre oleadas se diseñó una clase “GeneticManager” que básicamente se encarga de aplicar un algoritmo genético basado en la selección, cruce y mutación, cuando se finaliza cada oleada los genomas se recuperan y se evalúan su desempeño.

Se consideraron las alternativas como; regenerar los enemigos desde 0 en cada oleada, sin herencia genética o guardar los genomas de manera fija. Una mejora futura sería visualizar gráficamente el fitness y los mutados.

008: Algoritmo A* para que los enemigos encuentren el mejor camino hacia el castillo.

El algoritmo se implementó en la cuadrícula haciendo que cada casilla del mapa guarde los valores:

G: costo desde la casilla hasta el castillo.

H: parte heurística.

F: suma de G + H

parent: casilla anterior en el camino.

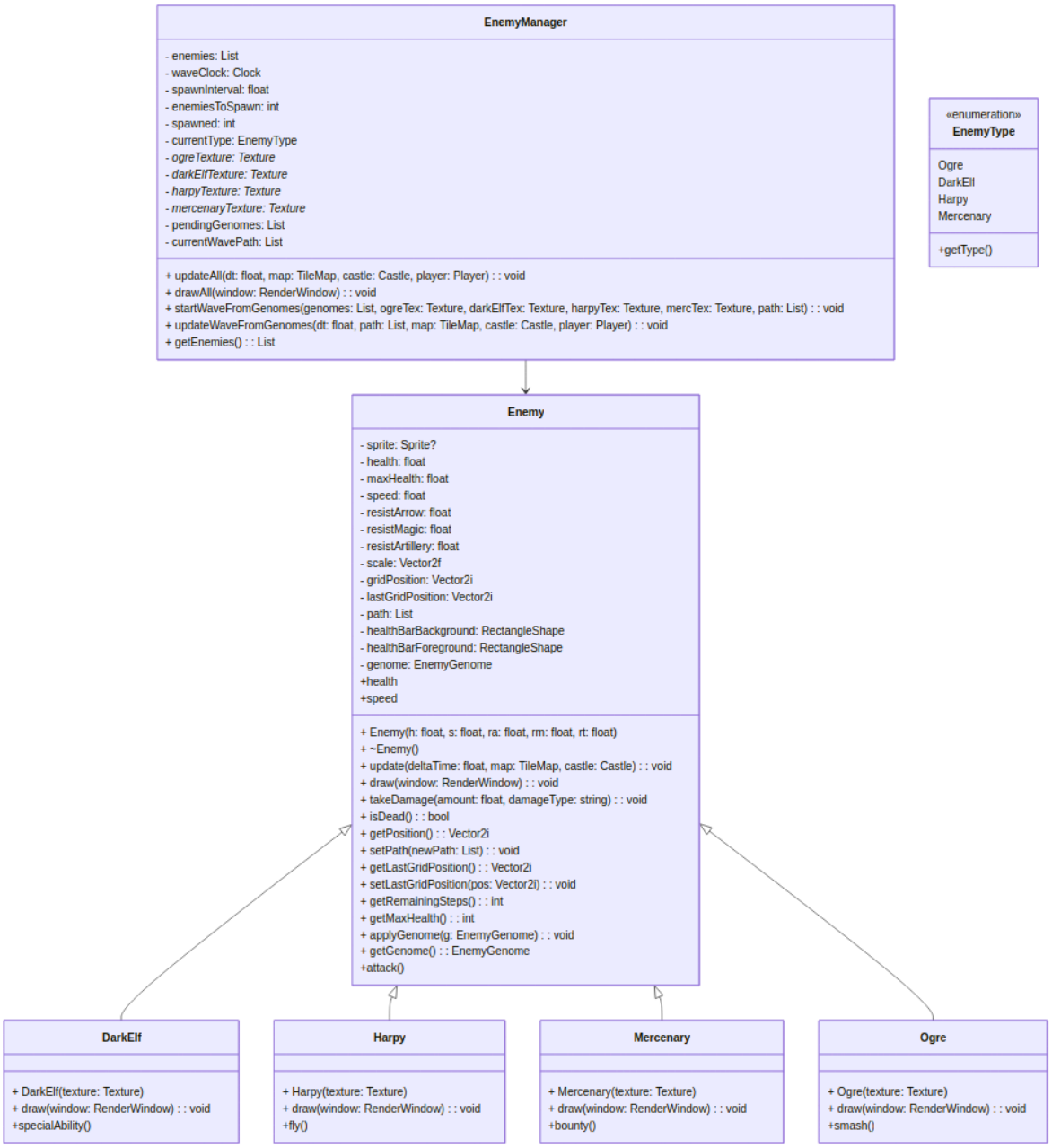
Se tomaron en cuenta las 8 direcciones, verticales, horizontales y diagonales asignando un costo de 10 para los movimientos verticales y horizontal y de 14 para los movimientos en diagonal. Con una cola de prioridad se manejó la open list organizados según su valor de F y se marcan las casillas ya evaluadas para ir a closed list. Al llegar al castillo se reconstruye el path siguiendo el "parent" hasta llegar al del origen.

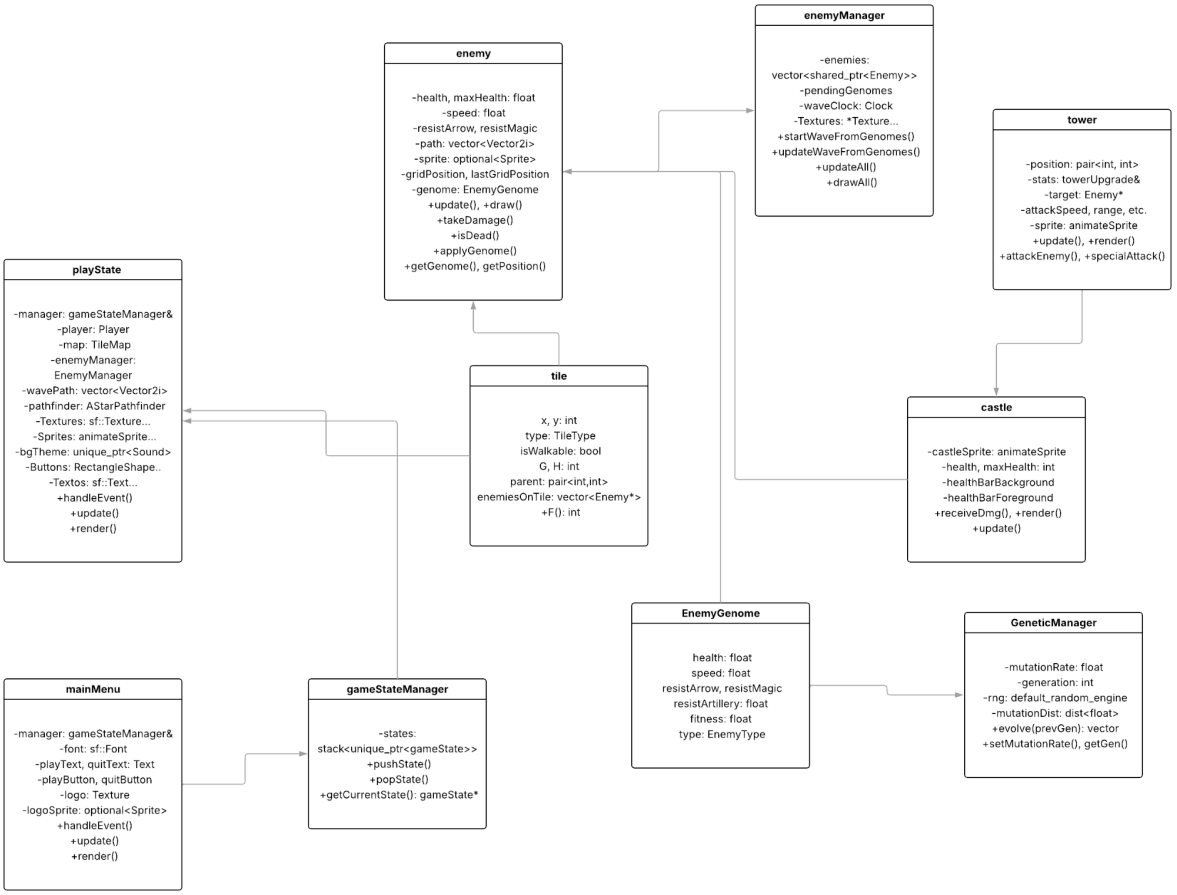
Aunque el algoritmo es eficiente el rendimiento puede disminuir si crece el mapa o conforme se agregan más torres y hay gran uso de memoria ya que debe mantener los valores de `G`, `H`, `F` y `parent` para cada casilla.

Se encontró el problema de que las rutas debían volverse a calcular cada vez que se colocaba una nueva torre. Se solucionó ejecutando el algoritmo nuevamente cada que se colocaban nuevas torres.

Diseño general

Diagrama uml para package enemies





Enlace al repositorio de Github

https://github.com/Mandamientos/Genetic-Kingdom.git