TEC | Tecnológico de Costa Rica

Proyecto II - GENETIC KINGDOM

Institución: Instituto Tecnológico de Costa Rica

Escuela: Ingeniería en Computadores

Curso: Algoritmos y Estructuras de Datos II (CE-2103)

Semestre: I Semestre 2025

Estudiantes: Eduardo José Canessa Quesada & Luis Felipe Chaves Mena

Fecha: 12 de mayo de 2025

---

1 Introduccion

El desarrollo de videojuegos implica enfrentar diversos desafíos relacionados con la programación, algoritmos y estructuras de datos. Este proyecto se centra en la creación de un juego de tipo **Tower Defense**, donde el jugador debe defender un objetivo, generalmente un castillo, de oleadas de enemigos mediante la colocación estratégica de torres. A lo largo del juego, el jugador debe analizar el mapa, elegir posiciones efectivas para sus torres y gestionar recursos, mientras enfrenta una creciente dificultad conforme avanza el juego.

Para impulsar la jugabilidad y aumentar la complejidad, se utilizarán dos algoritmos clave. El primero es el algoritmo **A***, que permite a los enemigos encontrar el camino más eficiente hacia su objetivo, evitando obstáculos generados por las torres del jugador. El segundo es el **algoritmo genético**, que se usará para la evolución de los enemigos, mejorando sus características a medida que avanzan las oleadas, lo que incrementa la dificultad del juego. Además de estos algoritmos, se implementarán lógicas para la generación de mapas, enemigos, movimientos y balas, asegurando una experiencia dinámica e interactiva.

2 Breve Descripción del Problema

El problema de forma general se presenta en 3 aspectos separables, en primer lugar el manejo de torres y las balas que disparan, luego se presenta la lógica de los enemigos, y por último la implementación gráfica y eficiente.

3 Descripción de la Solución

Se presenta solución en 3 aspectos, generación de enemigos y su aplicación gráfica, implementación de torres y balas junto con su implementación gráfica, y por último la implementación de un mapa eficiente y manejo de memoria adecuado.

3.1 Solución General

La solución general conlleva a realizar una separación de todo lo lógico con lo gráfico, o sea nada que sea gráfico deberá tener aspectos lógicos y visceversa, esto es una buena práctica ya que si hay fallos es más sencillo hacer una revisión. Además de esto se presenta una forma organizada de todos los aspectos del juego donde viene separados en carpetas cuyos nombres dan a entender su contenido de forma natural.

De forma general existe una clase que se encarga del manejor de dinero, esta es una clase que maneja un entero que es dinero actual, tiene métodos de obtención y aumento de dinero, se aclara en esta sección ya que es un aspecto sencillo y no lo suficientemente largo para profundizar en ello.

Una vez clara la forma de solución general, es necesario describir cada una de las soluciones para cada sub-problema descrito.

3.2 Solución de Torres y Disparos

Para el desarrollo de la solución se presentan las torres como un problema de herencia, donde la clase **Tower** tendrá las principales funciones de las torres, estas incluyen los *setters* y *getters* de: daño, velocidad, alcance y tiempo de recarga de ataque. Además para el poder especial se tiene una probabilidad de 25 % inicial de realizar un ataque que duplique el daño, tanto como las funciones de *Upgrade* 1, 2 y 3 de forma virtual para que cada mejora de torre sea especifica en cada clase hija, que estas son **Archer**, **Wizard** y **Artillery**. Para el manejo del asset especifico se tiene un entero que maneja la posición de forma horaria, arriba es 1, derecha 2, abajo 3 e izquierda 4. Existen atributos que ayudan a diferentes implementaciones como por ejemplo el contador de balas que indica si ya hay 2 balas en el mapa, la torre no puede tirar más, esto para control la cantidad de balas, o el nivel de la torre que indica el nivel actual, esto para la producción de balas con daño especifico y para la pantalla de información. Las torres también tienen una función que busca, en el vector de enemigos, si uno esta en alcance para disparar, esto genera la bala y la agrega a vector de balas, esto funciona gracias a la referenciasión propia de los punteros de C++. Ahora en relación con los *upgrades* y cada clase, estas son especificas, cada *upgrade* genera cambios distintos, en general cada *upgrade* tiene un costo genérico para todas las torres pero sus cambios se ven reflejados en daño y probabilidad de crítico.

Como se explica en la sección gráfica, hay ciertos lugares del mapa donde se pueden colocar torres, la colocación es por medio de una pantalla que indica el tipo de torre, se rebaja del monedeor existente y se coloca gráficamente la torre específica, esto cambia en el *grid* el numero de esa casilla a 5 para identificar torre colocada, además cuando se aprieta una casilla de numero 5, esto indica que se intenta mejorar de nivel, o sea hacer un *upgrade*, el cual indica el costo por aplicar, se puede cancelar cualquiera de estas transacciones.

En relación con las *bullets*, existe una estructura que se llama **Bullets** que maneja daño, posición, dirección y referencia de enemigo, esto con el objetivo de que cada *frame* la bala aumente su posición, se dibuje y revise condición de colisión con enemigo, en caso de llegar a una pared de mapa es destruida, en caso de llegar a enemigo fijado también es destruida. Las balas normales se ven de color blanco y las balas con doble de daño tienen color negro.

3.3 Solución de Enemigos

El sistema de enemigos se implementó mediante programación orientada a objetos, diseñado para ofrecer flexibilidad y escalabilidad. La arquitectura se basa en una clase principal `Enemy` que centraliza la funcionalidad común, permitiendo la creación de distintos tipos de enemigos con comportamientos específicos mientras se mantiene un código base coherente y relativamente sencillo. Este enfoque facilita la reutilización de código, la extensibilidad para futuros tipos de enemigos y el aprovechamiento del polimorfismo para personalizar comportamientos.

La clase `Enemy` integra atributos fundamentales para todos los enemigos: salud (`health`), velocidad (`speed`), posición en el mapa (`gridPosition`), dimensiones (`size`), tipo de enemigo `type`, entre otros los cuales permiten una implementación adecuada de esta clase. Con el propósito de mantener un código limpio y fácil de gestionar la implementación características y funcionalidades como `A* pathfinding`, el cálculo del daño recibo y el renderizado fueron trabajados de forma externa a la clase.

Para definir las variantes de enemigos, se implementó un sistema de tipos mediante *enum class*:

```
enum class EnemyType {
    Orc,
    DarkMage,
    Undead,
    Assassin
};
```

Cada tipo se inicializa con parámetros propios en el constructor de la clase, y es modifico a través de `algoritmo genético` de manera externa mediante los `getters` y `setters`. Un ejemplo de una inicializan básica se puede ver en el siguiente ejemplo:

```
Enemy::Enemy(EnemyType type_) {
    switch(type_) {
        case EnemyType::Orc:
            health = x;
            speed = y;
            break;
        case EnemyType::DarkMage:
            health = a;
            speed = b;
            break;
        // ... otros casos con distintos atributos
    }
}
```

A* Pathfinding en el movimiento de los Enemigos

Como se mencionó anteriormente, las funciones de `pathfinding` no se implementan directamente en la clase `Enemy`, sino que se utiliza un atributo `pathway` (lista de puntos que describe el camino en la matriz del mapa) calculado mediante `A*`. El código se encuentra en `Sources/Game/Pathfinding/` y es una implementación estándar que recibe: una matriz con diversos valores que representan ya sea puntos viables u obstáculos, punto inicial y final, devolviendo la ruta óptima.

3.4 Implementación Gráfica

La representación visual del mapa se gestiona mediante un sistema modular que combina la carga de archivos de mapa, el procesamiento de texturas y el renderizado optimizado.

3.4.1. Carga y Procesamiento del Mapa

Como primer procedimiento para la implementación gráfica, el mapa es cargado a partir de un archivo (`.txt`) en el cual la estructura del terreno se define mediante caracteres ASCII, y cada símbolo corresponde a un tipo de celda específico. Dicho archivo es procesado y guardado en un atributo de la estructura `Map` que maneja el mapa como una cuadrícula de enteros.

```
1 struct Map {
2     std::vector<std::vector<int>> grid;
3     std::vector<Vector2> start = {};
4     Vector2 goal;
5     int width;
6     int height;
7     RenderTexture2D renderTexture;
8     bool textureInitialized = false;
9 };
```

Esta estructura además, maneja datos lógicos de la cuadricula y los recursos gráficos de esta.

3.4.2. Renderizado Optimizado con Texturas

Para garantizar un rendimiento fluido, el mapa se renderiza una vez en una textura de dos dimensiones (`RenderTexture2D`) -clase proveída por la libería `RayLib`- que luego se dibuja en cada fotograma.

Colocación de Detalles

Algunos elementos gráficos, los cuales buscan aportar escenografía, son colocados de forma aleatoria en celdas predeterminadas.

3.4.3. Dibujado del Mapa

La función `DrawMap` utiliza la textura pre-renderizada para evitar cálculos redundantes en cada fotograma. Esta estrategia ayuda a reducir la carga de la GPU al reutilizar una textura estática.

Manejo de archivos

Los procesos que son exclusivos de la creación del mapa son trabajados en la carpata de nombre `Level`, y aquellos otros que implican la colocación de enemigos o torres u otras operaciones no exclusivas del mapa se traban en `Game`.

3.5 Implementación del Algoritmo Genético

Para la implementacion del algoritmo genético se hizo por medio de una clase llamada **GeneticManager**.

```
class GeneticManager
    // General data
    int mutation = 0;
    int probMut = 65;

    // Saves incoming stats from enemies
    int newStats[4][3] = {
        {0, 30, 10},
        {1, 20, 15},
        {2, 40, 15},
        {3, 15, 25}
    };

    // Initial condition of each enemy type
    int orc[2] = {30, 10}; // Type0, health, speed
    int darkmage[2] = {20, 15}; // Type1, health, speed
    int undead[2]= {40, 15}; // Type2, health, speed
    int assasin[2] = {15, 25}; // Type3, health, speed
```

En general, **GeneticManager** se encarga de tomar los arrays de enemigos y definir el mejor atributo para la vida y la velocidad. Al ser creado un enemigo, en ronda 1, este toma los valores bases de los arrays de cada enemigo, estos tienen una probabilidad de mutación del **65 %**. Si estos mutan, se aumenta la cantidad de mutaciones, ademas, existe una función que revisa en caso de mutación de velocidad o vida, si es mayor que la base actual, en ese caso se agrega a **newStats**. Este array tiene como objetivo ir guardando los nuevos mejores stats sin que cambien las bases, cuando se termina una ronda, los nuevos mejores stats se agregan a cada uno de los arrays especificos de cada enemigo.

4 Recursos y Herramientas

El desarrollo del proyecto se apoyó en los siguientes recursos tecnológicos:

- **Biblioteca RayLib**: Se utilizó de la biblioteca para la creación de juegos RayLib: RayLib GitHub Repository
- **Repositorio GitHub**: Todo el código fuente y documentación del proyecto se encuentra disponible públicamente en el repositorio: Genetic Kingdom GitHub Repository

5 Diseño General

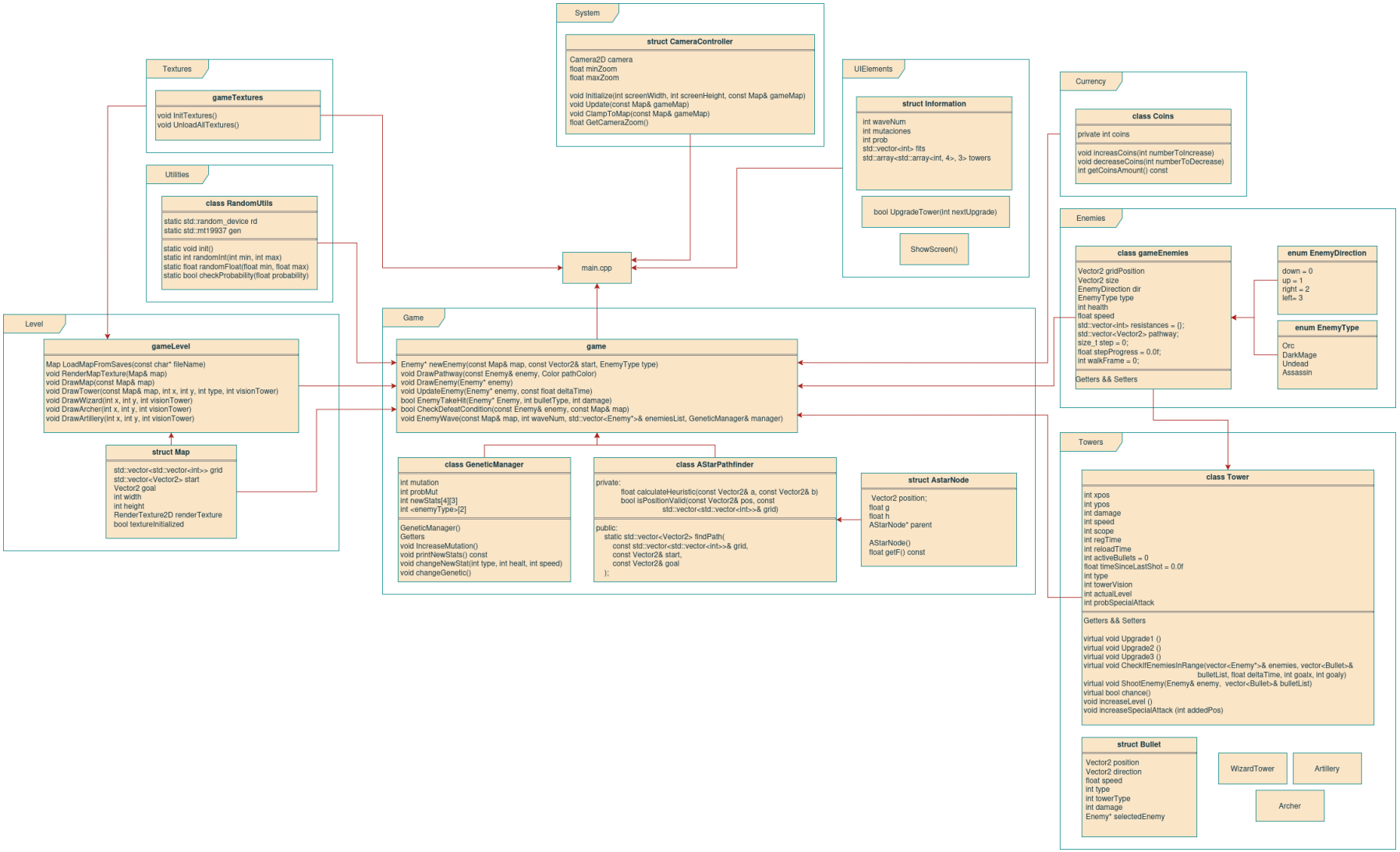

Figura 1: Diagrama UML