ITCR

Ingeniería en Computadores

Algoritmos y Estructuras de Datos 2

Documentación Proyecto 2

Estudiantes:
Esteban Campos Abarca - 2022207705
Jafet Díaz Morales - 2023053249
Steven Pérez Aguilar - 2024118003

I Semestre 2025

Tabla de Contenidos

Introducción

En este documento se presenta la documentación de un juego estilo "Tower defense" llamado Genetic Kingdom. El objetivo de diseñar e implementar este juego es poder aplicar algoritmos genéticos y de pathfinding en el lenguaje de programación C++ además de diseñar la solución del problema mediante Programación Orientada a Objetos (OOP).

Breve Descripción del Problema

El proyecto consiste en implementar un juego estilo "Tower Defense" ambientado en la edad media en C++ para desktop. El juego genera oleadas de enemigos de distintas clases y categorías. El jugador se encarga de colocar torres en lugares predeterminados para evitar que los enemigos crucen el puente del castillo. Después de cada oleada, los enemigos evolucionan haciendo más difícil evitar que los enemigos crucen el puente.

El problema a resolver de la implementación radica en principalmente en tres aspectos:

1) Generación de enemigos: Se utiliza un algoritmo genético para generar constantemente oleadas de enemigos que sean de diferente tipo y puedan mutar.
2) Movimiento de los enemigos: Se utiliza un algoritmo de Pathfinding A* para que los enemigos se muevan de forma óptima por el camino.
3) Torres: Se deben generar torres de diferente tipo y atributos para que eliminen a los enemigos.

Descripción de la solución

Se procederá a explicar la solución en referente a cada requerimiento.

Se tienen los siguientes requerimientos:

Requerimiento 001:

Descripción: Generar un mapa cuadriculado de tamaño fijo donde se puedan colocar torres en cualquier cuadro que no bloquee el camino al puente. El mapa tiene un único punto de ingreso de los enemigos y está en el lado contrario al puente del castillo.

Solución: Se utiliza la clase TileMap para manejar el mapa cuadriculado. Consiste en una clase TileMap.hpp y su posterior implementación en TileMap.cpp (véase el diagrama UML para todos los atributos y métodos de la clase).

En main.cpp se utiliza el constructor/instanciador de TileMap para generar por única vez el mapa (TileMap map) cuadriculado el cual define sus obstáculos a través de una matriz (int laberinto, que contiene una lista con listas) donde 1 es el obstáculo y 0 el camino donde los enemigos pueden pasar (al ser llamado, TileMap dibuja toda la cuadrícula y el método .SetObstacle (línea 108) asigna los colores de acuerdo a si es obstáculo o no). El método .draw (línea 206) TileMap también realiza otras cosas además de dibujar los tiles, pues se encarga de dibujar los enemigos en el mapa de acuerdo a sus características, así como de dibujar proyectiles y torres.

Agregar fotos de set obstacle, draw y main

Requerimiento 002:

Descripción: Cada torre tiene los siguientes atributos: daño, velocidad, alcance, tiempo de regeneración del poder especial, tiempo de recarga de ataque. Las torres atacan a los enemigos cuando están en su alcance. Con cada muerte de enemigo, se considera su categoría y tipo para devolver cierta cantidad de oro al jugador. La cantidad de oro retornado debe ser calculado de forma justa y consistente.

Solución: Se utiliza la clase Tower para manejar a las torres. Consiste en una clase Tower.hpp y su posterior implementación en Tower.cpp (véase el diagrama UML para todos los atributos y métodos de la clase).

El instanciador/constructor de Tower tiene todos los atributos mencionados en el requerimiento (daño, velocidad, etc.). El instanciador crea la torre en un valor “base” (línea 52) y luego dependiendo del valor de el tower_level (atributo del instanciador) se realiza una cierta llamada al método .upgrade( ) para definir los atributos actualizados.

Sobre el ataque de las torres a los enemigos, se explica en el siguiente requerimiento.

```cpp
Tower::Tower(sf::Vector2f startPosition, int tower_type, int tower_level, float tile_size)
: position(startPosition), tower_type(tower_type){
// Visuals:
    sprite.setRadius(10.f);

    sf::Color tower_colors[3] = {
        sf::Color(190, 190, 255),
        sf::Color(255, 170, 210),
        sf::Color(255, 190, 150)
    };
    sprite.setFillColor(tower_colors[tower_type]);
    sprite.setOrigin(10.f, 10.f);

    sprite.setPosition(position);
    // Stats. Depend on level and type
    // Period.
    int towers_period_by_type[3] = {60, 120, 240};
    period = towers_period_by_type[tower_type];
    // Damage
    float towers_damage_by_type[3] = {7.5f, 20.f, 25.f};
    damage = towers_damage_by_type[tower_type];
    // Fire range.
    float towers_range_by_type[3] = {10.f, 7.1f, 5.0f};
    this->fire_range = towers_range_by_type[tower_type] * tile_size;
    // Projectile speed.
    float projectile_speed_by_type[3] = {300.f, 150.f, 200.f};
    projectile_speed = projectile_speed_by_type[tower_type];
    // Upgrades many times in a row.
    for (size_t i = 0; i < tower_level; i++)
    {
        upgrade();
    }

}
```

Requerimiento 003:

Descripción: Todas las torres tienen 3 upgrades. Cada upgrade aumenta el daño que pueden causar y cada torre tiene ataques especiales que ocurren cada cierto tiempo con una probabilidad definida por los estudiantes. Los upgrades tienen un costo en oro. Cada upgrade es más caro que la anterior.

Solución: La clase Tower tiene su respectivo constructor/instanciador el cual tiene atributos esenciales que representan la lógica de la implementación. En Tower.cpp se define que al llamar a Tower se requieren de varios parámetros. En este requerimiento nos interesa explicar los siguientes:

int tower_type y int tower_level: pide un número, dependiendo del número que reciba, ese tipo y nivel de torre se va a designar, así se manejan los upgrades a través de una simple comparación numérica.

Por otro lado, Tower tiene el método .update (línea 96) el cual se encarga de manejar los tiempos para que se realicen los disparos. Es una función booleana que compara periodos de tiempo. En TileMap,cpp se recurre al método update (línea 345) a través de un bucle for. Se realiza en TileMap debido a que es la función que está dibujando todo. Este bucle actualiza (en tiempo deltaTime) todos los atributos de tower de acuerdo al tipo de torre que se tiene y posteriormente se encarga de, cada que es necesario, mediante el método .ShootNearest() (en base a la posición y otros atributos), de disparar al enemigo más cercano.

foto de .update, constructor y bucle for en tilemap???

Requerimiento 004:

Descripción: Hay 3 tipos de torres:
• Arqueros: bajo costo, alto alcance, poco daño, tiempo de recarga de ataque bajo
• Magos: costo medio, alcance medio, daño medio, tiempo de recarga de ataque medio
• Artilleros: costo alto, alcance bajo, daño alto, tiempo de recarga de ataque alto

Solución: Los tipos de torre y como se implementaron ya se explicó en los requerimientos 2 y 3. En resumen, se crea la torre en valor base y dependiendo del tipo, se llama a .upgrade( ) para actualizar sus valores.

Requerimiento 005:

Descripción: El jugador puede ir colocando las torres en cada lugar disponible. Al seleccionar un lugar disponible, puede escoger el tipo de torre que desea crear. Cada torre tiene un costo en oro.

Solución: La implementación de este requerimiento se hace en TileMaps pues es la clase en la que se está manejando lo dibujado del grid. Cuando se llama al inicio al instanciador de TileMaps se crean 3 rectángulos. Estos rectángulos pueden recibir un input manejado a través de una variable button. Dependiendo del button que se active (verificación mediante if) se crea la torre respectiva, mandando el input a las funciones ya descritas en requerimientos previos. En general, el instanciador crea los rectángulos y el método clickEvents( ) de TileMaps redirige los inputs recibidos a las funciones que se encargan de dibujar. En main , hay un bucle de while que espera los inputs de clickEvent en todo momento.

```
166         sf::Event event;
167         while (window.pollEvent(event)) {
168             if (event.type == sf::Event::Closed) {
169                 window.close();
170             }
171             else if (event.type == sf::Event::MouseButtonPressed) {
172                 if (event.mouseButton.button == sf::Mouse::Left) {
173 
174                     sf::Vector2i target_pos_pixels(event.mouseButton.x, event.mouseButton.y);
175                     map.clickEvents(target_pos_pixels);
176 
177                 }
178             }
179         }
180 
181         map.update(deltaTime);
182 
183         window.clear(sf::Color::White);
184         map.draw(window);
185 
186         // Dibujar objetivo
187         if (hasTarget) {
188             sf::CircleShape target(10.f);
189             target.setPosition(targetPos.x * 30 + 5, targetPos.y * 30 + 5);
190             target.setFillColor(sf::Color::Red);
191             window.draw(target);
192         }
193 
194         // character->draw(window); // Moved to TileMap.draw
195         window.display();
196     }
197 
198     return 0;
199 }
```

Requerimiento 006:

Descripción: Los enemigos aparecen por oleadas. Cada oleada es una generación que evoluciona. Los enemigos pueden ser los siguientes:

• Ogros: son el enemigo más básico. Son resistentes a los arqueros y débiles contra la magia y la artillería. Son lentos.

• Elfos Oscuros: son resistentes a la magia, pero débiles a los arqueros y a la artillería. Son muy rápidos

• Harpías: solo pueden ser atacadas por magia y arqueros. Tienen una velocidad intermedia.

• Mercenarios: son débiles a la magia, pero resistentes a arqueros y artillería.

Cada enemigo tiene atributos como:
• Vida
• Velocidad
• Resistencia a flechas

• Resistencia a la magia
• Resistencia a la artillería

Solución: Cada enemigo se crea en TileMaps mediante el método .draw( ). El método recibe una lista enemy_species el cual es una lista generada desde genetics.cpp. En base al valor de enemy_species[i] se genera el enemigo respectivo con sus respectivas características. genetics.cpp trabaja con matrices que contienen datos numéricos que reflejan las características de cada individuo. Utiliza randoms para generar cada valor nuevo. genetics.cpp tiene otra lista best_individuals que contiene a los mejores enemigos de cada ciclo. Estos atributos de esta lista son los utilizados para generar los enemigos de cada oleada. En cada oleada, los best_individual se pasan a través de la variable enemy_species a TileMaps para que los dibuje.

Requerimiento 007:

Descripción: Cada generación selecciona los individuos con el mejor fitness, los cruza e ingresa los nuevos individuos a la población. Pueden ocurrir mutaciones con cierto grado de probabilidad. El estudiante debe definir cada elemento del algoritmo genético y explicarlo adecuadamente en la documentación. Las oleadas son de tamaño variable y se generan con un intervalo parametrizable.

Solución: Se tiene una clase Individual para cada especie de enemigo (por ejemplo,un ogro es un Individual). La clase Individuals tiene un método CalculateFitness( ) para calcular los atributos de cada individuo por sí solo y luego compararlo con otros. Así Genetics :: Genetics( ) contiene como atributos (Individuals) a todos los “tipos de especies”, ogros, elfos, etc. En cada ciclo se llama al método SimulateWave( ) el cual a través de un for crea nuevos enemigos en base a un porcentaje de aparición y un n definido. Posteriormente, se llama a la función MutateStats( ), la cual utilizando un valor mutation_relative_change (atributo en Individuals) y utilizando RandomBool aleatoriza los valores en base al valor de mutación definido y la operación matemática específica definida. Posteriormente se vuelve a calcular el fitness.

```
96  void Individual::MutateStats()
97  {
98      this->max_health    *= 1 + this->mutation_relative_change * (1 - 2*RandomBool(0.5));
99      this->speed_multiplier        *= 1 + this->mutation_relative_change * (1 - 2*RandomBool(0.5));
100     this->pierce_armor  *= 1 + this->mutation_relative_change * (1 - 2*RandomBool(0.5));
101     this->magic_armor   *= 1 + this->mutation_relative_change * (1 - 2*RandomBool(0.5));
102     this->siege_armor   *= 1 + this->mutation_relative_change * (1 - 2*RandomBool(0.5));
103     CalculateFitness();
104 }
```

Aquí se muestra el cálculo del fitness mediante una operación matemática.

```
84  void Individual::CalculateFitness(float completed_path, float remaining_health)
85  {
86      // Fitness range: [0,2]
87      // Actual fitness function.
88      float calc_fitness = completed_path/(1 - remaining_health/2);  //remaining_health = health/max_health. Same for the completed path.
89      this->fitness = calc_fitness;
90  };
```

Una vez hecho el proceso de mutar y calcular fitness, se le llama al método CalculateBest5FitnessIDs( ) (linea 207) para escoger los 5 mejores best_individuals (recuerde que previamente se mencionó que se tenía una lista best_individuals, la cual se le van a actualizar sus valores). (Nota: todo esto se que se menciona se encuentra en genetics.cpp). Así se actualiza la lista con los individuos y también se calcula el mejor enemigo creado:

```
283  void Genetics::CalculateNewBestIndividual()
284  {
285      for (size_t i = 0; i < species.size(); i++)
286      {
287          // Each of the 5 genes in an Individual.
288          // gene order float max_health, float speed_multiplier, float pierce_armor, float magic_armor, float siege_armor.
289          best_individuals[i] = Individual(
290              best_individuals_matrix[i][0 % best_individuals_matrix[i].size()].max_health,
291              best_individuals_matrix[i][1 % best_individuals_matrix[i].size()].speed_multiplier,
292              best_individuals_matrix[i][2 % best_individuals_matrix[i].size()].pierce_armor,
293              best_individuals_matrix[i][3 % best_individuals_matrix[i].size()].magic_armor,
294              best_individuals_matrix[i][4 % best_individuals_matrix[i].size()].siege_armor);
295      }
296  }
```

Con esto termina el proceso genético de SimulateWave y se pasa la lista a TileMaps para que dibuje los enemigos. genetics.cpp tiene funciones Show…() para mostrar los enemigos creados, sus especies y el mejor individuo en cada oleada. Es importante notar que genetics.cpp también cuenta con un método CleanWave que se encarga de limpiar cada oleada cuando va a iniciar el SimulateWave( ).

Requerimiento 008:

Descripción: Los enemigos utilizan Pathfinding A* para encontrar el camino hacia el puente del castillo.

Solución: Una vez creados los enemigos,

Requerimiento 009:

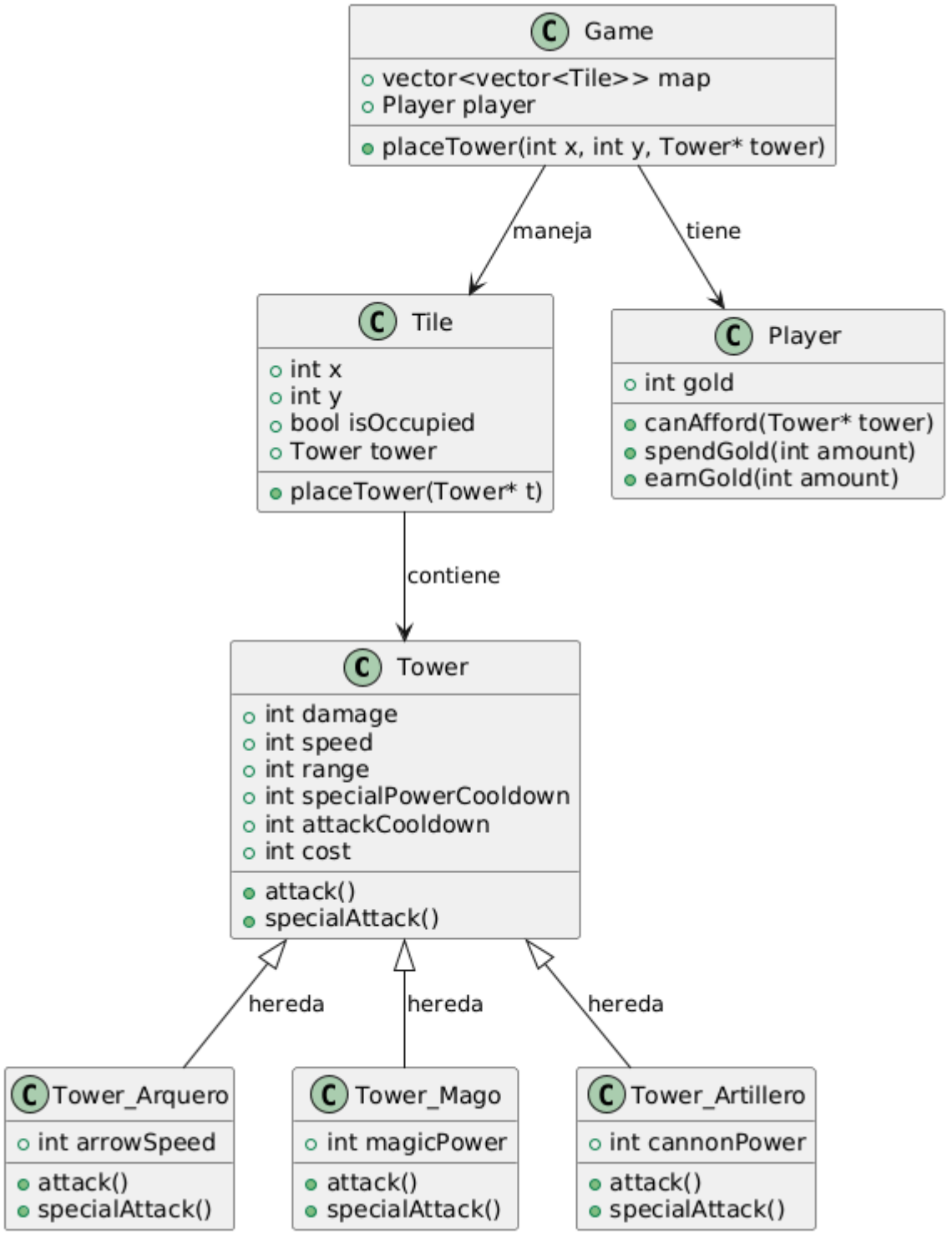
Descripción: El juego muestra un panel con estadísticas como:
• Generaciones transcurridas
• Enemigos muertos en cada oleada
• Fitness de cada individuo de la oleada
• Nivel de cada torre
• Probabilidad de mutaciones y cantidad de mutaciones ocurridas

Solución: Se utilizan las funciones Show … ( ) de genetics.cpp para printear y mostrar los enemigos creados, valores de fitness, mutaciones, entre otras cosas referentes a los enemigos.

En TileMaps (linea 322) se elimina a los enemigos sí Event detecta que no tienen vida. Esto printea y actualiza el número de enemigos muertos. En TileMaps (linea 230) se dibuja a las Torres, allí se realiza un print y se actualiza el nivel de las torres presentes.

Diagramas UML

A continuación se muestran los diagramas UML con las clases más importantes del proyecto.



Enlace a github:
https://github.com/EstebanACamposA/campos-diaz-perez-genetic-kingdom/tree/master