Project Plan: Tower Defense Game (Clash of Armies)

Group Members: Aleksi Kalliomäki, Imran Mamin, Leevi Salonen, Theodora Valerie

I. Scope of Work

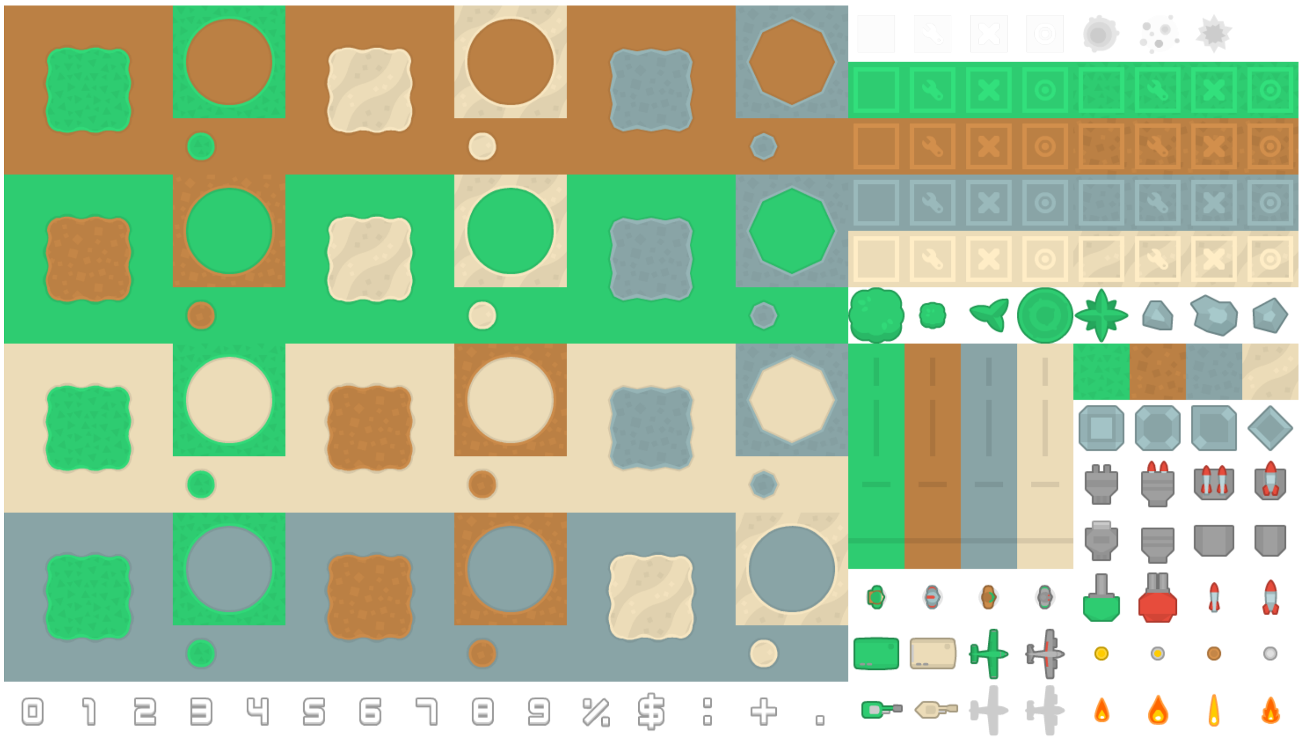
Features and Functionalities

- 2D tower defense game environment where we will design the game's play area as a grid-based map, and we plan to use tiles for this map. The tiles, which can depict the path, the ground, and the water, will allow easier map assembly at every level. The enemy will walk through the path and the player cannot place anything in the path at least in the earlier levels. The ground is grass where the player can place ground towers and the water is for possible new towers such as military ships.
- Atleast 5 maps increasing in difficulty.
- Implement a player economy system where players earn and spend money. The enemy also has a certain set amount of money that it will use for attacking. Both enemy and the player have a passive income varying on the game difficulty. In addition the player earns money for destroying enemies.
- Allow different game difficulty settings such as easy and hard which is determined by the ratio of the enemy’s money and the player’s money.
- At least three types of enemy troops with different difficulty levels, which are the tanks, soldiers, and airplanes.
- There are at least 2 types of projectiles, which are bullets and missiles from three types of towers which are cannons, missile launchers, and fighter planes. Bullets and missiles have different mechanics. Missiles follow enemies while bullets just have a set path.
- Allow tower upgrades (range, attack speed, and damage). Also towers can be repaired and sold.

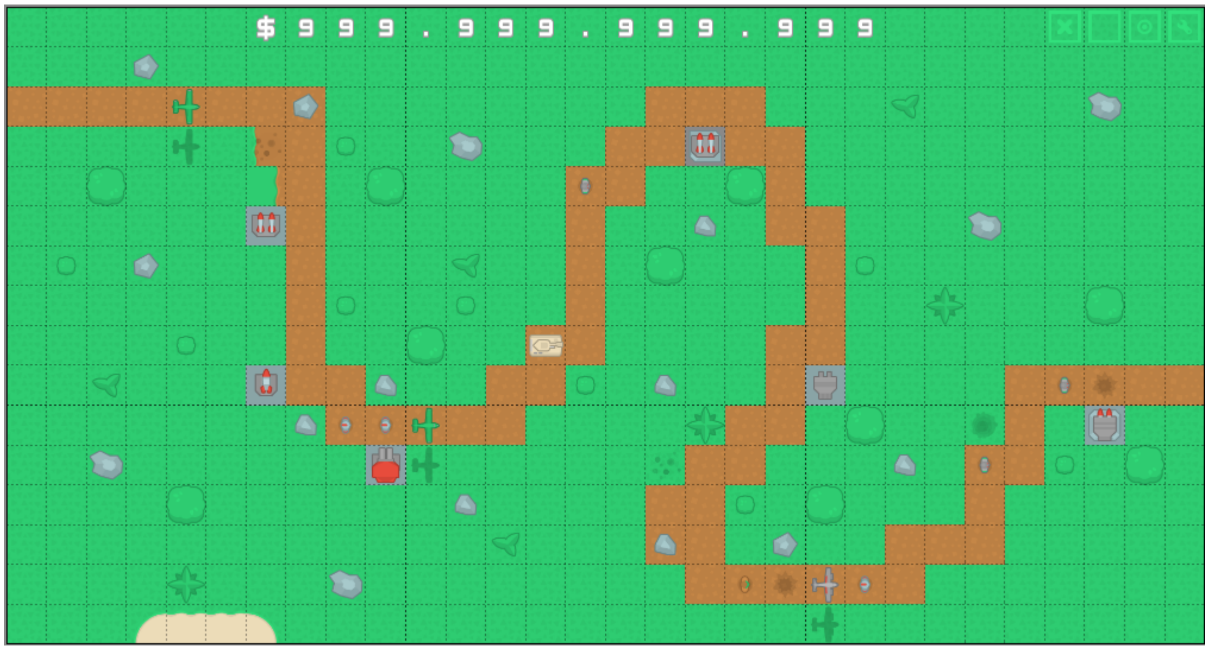
How the Program Works

The game is about military base defense where players assume the role of a strategic commander responsible for safeguarding their military base from enemy incursions. The game begins with players selecting a mission (level), each with its own unique challenges and objectives. As the game commences, waves of enemy forces approach along predefined paths on a grid-based map. The player's primary task is to strategically position and upgrade various types of military towers, ranging from cannons and missile launchers to fighter planes, to fend off the advancing threats. Money is earned by successfully repelling enemy attacks and is used to purchase and upgrade towers. The opponent which is the computer generates the enemies based on the amount of money that it has.

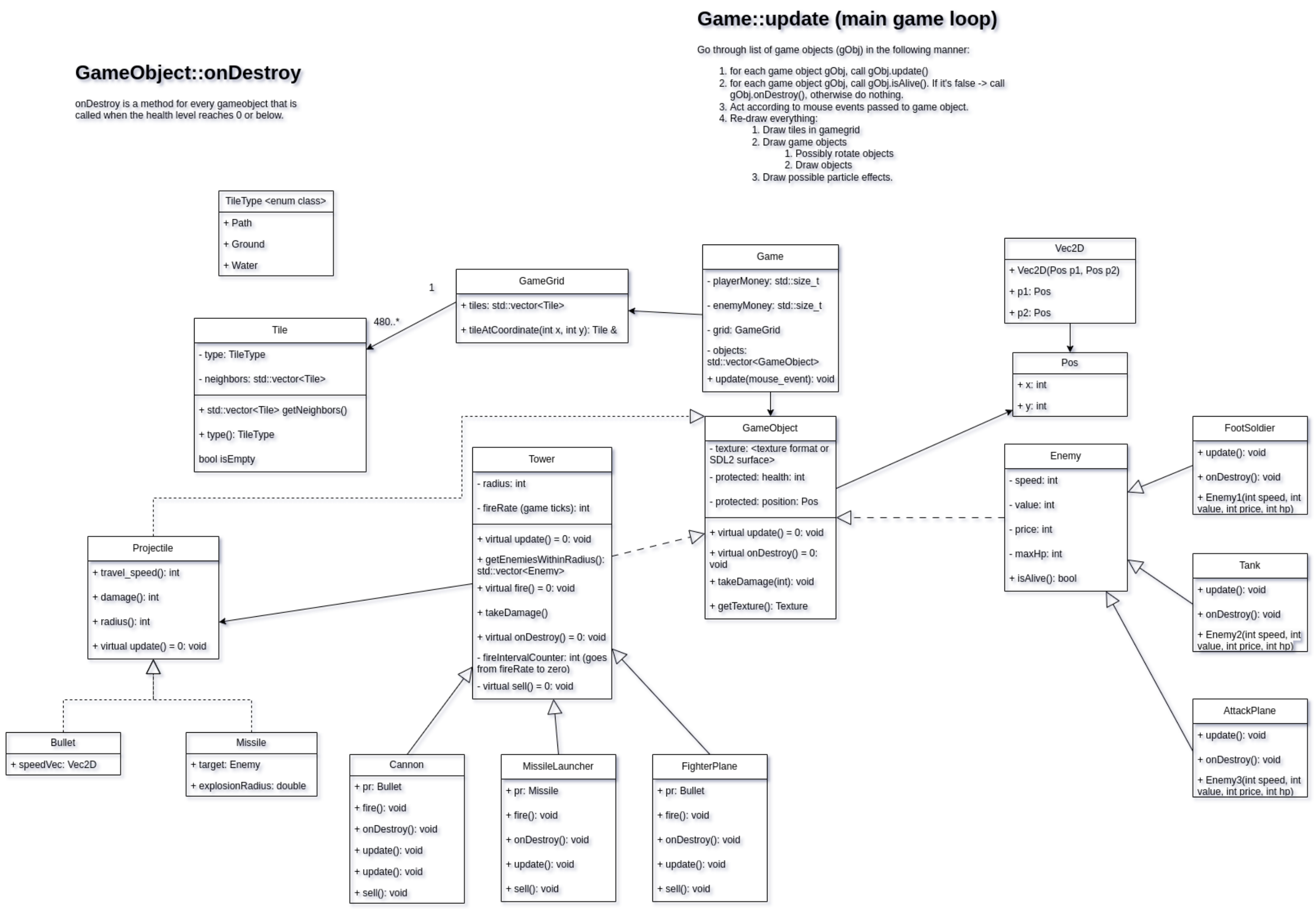
Here is a picture of game assets that we will use for the game:



Crude mockup version of an example map:



II. High-Level Structure of the Software



Game::update (main game loop)

Go through list of game objects (gObj) in the following manner:

1. for each game object gObj, call gObj.update()
2. for each game object gObj, call gObj.isAlive(). If it's false -> call gObj.onDestroy(), otherwise do nothing.
3. Act according to mouse events passed to game object.
4. Re-draw everything:
    1. Draw tiles in gamegrid
    2. Draw game objects
        1. Possibly rotate objects
        2. Draw objects
    3. Draw possible particle effects.

GameObject::onDestroy

onDestroy is a method for every gameobject that is called when the health level reaches 0 or below.

Here are the list of key classes within the game:

- Game Class:
  This class represents the main game engine and controls the game's overall flow. Has two loops: First one updates the game objects and reduces their health for example and the second one checks if they're alive and removes them accordingly.
- GameGrid Class:
  This class defines the game map. It encapsulates information about the grid-based layout, paths for enemy movement, and tiles that make up the map.
- Tile Class:
  These tiles have attributes indicating their type (e.g., path, ground, water), and they can also hold information about game objects placed on them,

such as towers or enemies. Additionally, this class includes a getNeighbors() method which is used to return a list or array of neighboring tiles in the main directions for a given tile.

- GameObject Class:
  This class provides a foundational structure for the game entities which contain core attributes such as health and position and methods such as update(), onDestroy(), and takeDamage(). The update() method ensures that the object's behavior can evolve throughout gameplay. The onDestroy() method handles the object's removal or any specific actions that occur when its health reaches 0. Lastly, the takeDamage() method enables the object to respond to attacks or adverse effects by decrementing its health, allowing for dynamic interactions with the game environment and other entities
- Tower Class:
  Each tower has attributes for its type, radius, fire rate, attack speed, and damage. This is an abstract class with three subclasses that extend from it: Cannon, MissileLauncher, and FighterPlane. These subclasses are mostly similar but differ in the choice of projectile they use. Additionally, the virtual methods of the Tower class (fire, onDestroy, update) are implemented differently in each of the subclasses.
- Enemy Class:
  The "Enemy" class represents the different types of enemy units. It includes attributes such as enemy type, health, speed, and rewards for defeating them. There are three types of enemies, and thus this class is an abstract class with virtual methods, update and onDestroy, which subclasses will implement. Enemy types are FootSoldier, Tank and AttackPlane.
- Projectile Class:
  As the name already tells, this class is an abstract class that represents a single projectile. It contains some properties of the projectile, such as travel speed, damage, and damage radius that will be used for calculations in the update function. It has two subclasses that extend from it: Bullet and Missile.
- Bullet Class:
  This class represents a single bullet that Cannon and FighterPlane are using for killing the enemies. It has one property, which is the vector from the cannon or plane to the enemy.
- Missile Class:
  This class is similar to a Bullet class, but instead of hitting just one enemy, it hits all the enemies within the defined radius. Additionally, compared to

the bullet instance, a missile uses the target (enemy) instead of a vector to the enemy. This means that the trajectory of the missile is not necessarily a straight line to the enemy, but can, for example, follow a curved path.

III. Planned Use of External Libraries

Utilize the SDL2 library for drawing graphics, playing audio and getting user input. Additionally, tmxlite will be used for loading tmx maps.

IV. Division of Work and Responsibilities

Work will be divided among team members using a Kanban system. Weekly meetings will be held to review progress and assign tasks. Points will be tracked for every feature done. Biweekly standup meetings (max 5 minutes) for quick progress updates.

We will appoint one person to maintain our build files, allowing everyone to focus on program development instead of troubleshooting errors caused by mistakes in those files. Each group member is responsible for maintaining the cleanliness of the main branch in the remote version control system. This is achieved by building and running the program without errors and warnings and adhering to code review principles. Additionally, the program should be tested using Valgrind to detect memory leaks early in development, which aids in the early detection of errors. Furthermore, once unit tests are written, each group member must run and pass them before committing changes to the main branch.

During development, developers should add asserts to the codebase when they are certain that certain variables or parameters are within the expected range. This practice will save time during the debugging process.

V. Planned Schedule and Milestones

Week 1: Finish technical planning, ideation, and define game mechanics.

Week 2: Divide tasks into Kanban tickets, assign task owners, set up the project, create a window and load a map into the window from a tmx mapfile, and begin searching for game assets (sounds, etc). Install SDL2 library. Start implementing map creator.

Week 3: Implement game mechanics, tower behavior and enemy troop behavior.

Week 4: Prepare a Minimum Viable Product (MVP) of the game with basic functionality.

Week 5: Add additional features, create new levels, and explore the possibility of network gameplay.

Week 6: Focus on bug fixes and make any necessary improvements based on feedback. Perform any last-minute quick fixes before project completion.