

Name of Project: “CG-Defense”

Project Developers:

1. Ahmed Reza Tausif (Roll: FH-012)
2. Mayesha Binte Liton (Roll: SK-062)
3. Tukabbir Hossain Sadi (Roll: FH-048)
4. Mishkatul Ferdousi (Roll: SN-045)

“A Report on Modules & Tools of the Project”

Overview

This **Tower Defense game** is designed to be a **real-time strategy game** where players use towers to defend against waves of enemies. The goal is to prevent enemies from reaching a designated target area by strategically placing towers that attack the enemies as they follow a path.

The game features multiple levels (doesn't represent difficulty), various tower types, projectile handling(2D), and an intuitive game editor for level creation. The core mechanics revolve around resource management (e.g., money and health), strategic tower placement, enemy pathfinding, and tower upgrades. The game also integrates features like UI, timers, animations, and audio for a smooth player experience.

Core Game Components

1. Game Flow and Management (`game.cpp`, `game.h`)

- **Game Class:** This is the heart of the game. It runs the main game loop, switching between different game states (`MainMenu`, `LevelSelect`, `Playing`, `GameOver`), handling user input, and updating various game components such as enemies, towers, and UI elements.
- **Game State Management:**

- **MainMenu:** The player can start a new game, load an existing level, or adjust game settings.
 - **LevelSelect:** Allows the player to choose a level to play.
 - **Playing:** This state manages the gameplay. It handles enemy spawning, wave progression, and player actions like tower placement and upgrades.
 - **GameOver:** Displays when the player's health reaches zero or when enemies breach the target area. The player can restart or quit the game.
- **Handling User Input:** The game listens for mouse clicks to place towers or interact with UI elements. The `processEvents` function checks for these interactions and updates the game state accordingly.

2. Level Management (`level.cpp`, `level.h`)

- **Level Class:** Defines the layout of the game grid and the flow for enemies. It uses a **tile-based** system where each tile can either be empty, a wall, or a spawner.
 - **Tile Types:** The tiles could represent different objects, such as:
 - **Walls:** Block the movement of enemies.
 - **Spawners:** Spawn enemies at regular intervals.
 - **Target:** The target area that the enemies are trying to reach.
 - **Flow Field:** The `Level` class includes a **flow field** algorithm, which helps to determine the optimal path for enemies. It guides enemies from their spawn point to the target tile.
 - **Level Reset and Progression:** When the player completes a level, they move to the next one. If the player loses all their health or the enemies reach the target, the level is reset.

3. Enemy and Unit Management (`unit.cpp`, `unit.h`)

- **Unit Class:** Manages enemy characteristics and behaviors. Each enemy has attributes such as:
 - **Health:** How much damage the enemy can take before it is destroyed.
 - **Speed:** How fast the enemy moves.
 - **Type:** The enemy's behavior (e.g., fast, tanky).
- **Movement and AI:** Enemies follow paths determined by the flow field and may have unique behaviors based on their type. For example, fast enemies move quickly but have less health, while tank enemies are slower but have more health.
- **Enemy Interaction:** When enemies collide with projectiles, they take damage. If they reach the target area, the player loses health. When defeated, enemies drop money, which can be used to place or upgrade towers.

4. Tower Mechanics (`tower.cpp`, `tower.h`)

- **Tower Class:** This class represents the various types of towers in the game. Each tower has properties such as:
 - **Damage:** How much damage a tower deals per attack.
 - **Range:** The distance within which a tower can target enemies.
 - **Fire Rate:** How quickly the tower shoots projectiles.
- **Tower Types:**
 - **Basic Towers:** Standard towers that shoot projectiles at enemies.
 - **Sniper Towers:** Long-range towers with higher damage but slower fire rates. It has a special feature called **Dynamic Threshold**. If the tower level is greater than 1, it becomes more likely to target

higher-health enemies, making the Sniper Tower progressively more powerful at targeting tougher enemies. This dynamic threshold increases by upgrading sniper towers. If two enemies are within the Sniper Tower's range and have the **same health** (even considering the dynamic threshold), the current behavior in the `findEnemy ()` function will prioritize the **enemy that is closest to the tower**.

- **Cannon Towers:** Area damage towers with explosive projectiles.

- **Upgrade System:** Towers can be upgraded to increase their damage, range, and attack speed. Players spend money earned from defeating enemies to upgrade their towers.
- **Projectile Firing:** Towers aim at the nearest enemy within their range and fire projectiles toward the target.

5. Projectile Mechanics (`projectile.cpp`, `projectile.h`)

- **Projectile Class:** Manages the projectiles fired by the towers. Each projectile has its own behavior:
 - **Basic Projectiles:** Straight-moving projectiles with single-target damage.
 - **Cannon Projectiles:** These deal area damage, affecting multiple enemies within a certain radius.
- **Projectile Movement:** Projectiles travel toward their target and check for collisions. If they hit an enemy or reach their maximum range, they either deal damage or trigger an explosion (for cannon projectiles).
- **Explosion Handling:** When a projectile hits an enemy, it may trigger an explosion, damaging nearby enemies.

6. Texture Management (`textureloader.cpp`, `textureloader.h`)

- **TextureLoader Class:** Handles the loading and management of textures. The textures used for towers, projectiles, enemies, and backgrounds are loaded into memory efficiently using this class.
- **Texture Caching:** The textures are loaded once and stored in a static map, ensuring that they are reused rather than reloaded multiple times, reducing memory usage and improving performance.
- **Texture Optimization:** The game ensures textures are properly managed to prevent memory leaks, unloading textures that are no longer needed.

7. Timers (`timer.cpp`, `timer.h`)

- **Timer Class:** Used for managing timed events in the game, such as:
 - **Tower Cooldowns:** Each tower has a cooldown between shots, and the timer ensures that the tower doesn't shoot continuously.
 - **Projectile Timers:** Timers also control the flight time of projectiles, ensuring they follow their paths and explode at the right time.
 - **Animation Timers:** Some animations (like explosions) are time-based and are handled by timers.

8. JSON Data Handling (`json.hpp`, `leveldataio.h`)

- **Saving and Loading Levels:** Custom levels created in the level editor can be saved and loaded using **JSON** format. This allows players to create and share levels with ease.
- **Level Data:** Each level's layout (tile positions, spawners, target positions) is serialized into a JSON file, making it easy to load and modify levels.

Game Flow

1. Main Menu:

- The player is presented with the main menu, where they can start a new game, load a saved level, or access settings.

2. Level Selection:

- The player can select a level to play from a list of available levels. Each level has a unique difficulty and layout.

3. Gameplay:

- The player starts placing towers on the grid. Each tower has a set range and damage. The game starts by spawning enemies from the spawn points, and the player's goal is to prevent these enemies from reaching the target area.
- The player can upgrade towers using money earned from killing enemies.
- As the player progresses through waves, enemies get stronger, requiring the player to place more advanced towers.

4. Tower and Enemy Interaction:

- **Tower Actions:** Towers automatically target enemies within their range and shoot at them. The type of tower determines how effective it is against different enemies.
- **Projectile Actions:** Projectiles fired from towers collide with enemies, reducing their health. Some projectiles, like the cannon, cause explosions that damage multiple enemies at once.

5. End of Level/Restart:

- The game ends when all waves are completed or if the player's health reaches zero. The player is then presented with the option to restart or go to the main menu.

Why I Used Smart Pointers in the Tower Defense Game

I used smart pointers, like `std::shared_ptr`, in this Tower Defense game to **automate memory management**. They prevent **memory leaks** by ensuring objects are

automatically deallocated when no longer referenced. Smart pointers also allow **shared ownership**, making it easier for multiple components (e.g., towers and enemies) to safely interact with the same object. This reduces the risk of **dangling pointers** and simplifies the code by eliminating manual memory management (**new/delete**). In a dynamic game environment, smart pointers improve **stability**, **maintainability**, and **safety** in handling frequently created and destroyed game objects.

Additional Features and Possible Improvements

1. **AI Pathfinding:** Implementing dynamic pathfinding algorithms like **A*** would make enemy movement more intelligent, allowing them to avoid obstacles or adapt to the player's actions.
2. **Multiplayer Mode:** Adding multiplayer support, either cooperative or competitive, would greatly enhance the replayability of the game. We would love to execute it in the future.
3. **Advanced Tower Types:** Adding more towers, such as freezing or poison towers, would add variety to the strategies players can employ.
4. **Performance Optimization:** While textures are managed efficiently, using **texture atlases** could further improve performance by reducing the number of draw calls.
5. **Increased Level Complexity:** Introducing more complex levels with multiple paths, environmental hazards, or special events would challenge the player's strategic thinking.

Conclusion:

In this Tower Defense game project, I successfully implemented a robust system that allows for dynamic gameplay, intelligent enemy movement, and strategic tower placement. By utilizing smart pointers, I ensured efficient memory management and minimized risks of memory leaks and dangling pointers, allowing the game to run smoothly and safely. The use of dynamic thresholds for tower damage added depth to the gameplay, offering a more adaptive and challenging experience. The project's modular structure, with distinct components for towers, projectiles, units, and levels,

allows for easy expansion and customization, paving the way for future enhancements and optimizations.