Tower Defence: Alien Enigma

Detailed Project Design Document

Version 1.2

Keerthish Suresh

Contents

1. Overview

In this game of Tower Defence: Alien Enigma, the player must place their Watchtowers strategically to kill the enemies from reaching the target. By killing the enemies, player can earn resources that can be used to build and upgrade their towers. Each tower has its own unique range, strength, and attack style. Enemies have varying speed, health, and the reward capabilities. As the game goes on, the level gets more challenging with more waves and tougher enemies introduced to challenge the player.

Justification:

Tower Defence: Alien Enigma is chosen to challenge players with complicated improvised Watchtowers that require strategic thinking and problem-solving skills for better placements. The game integrates watchtower - defence mechanics, which adds complexity that makes it more than just a defence from the nemesis. My game selection will engage a broad audience.

This game will apply the fundamental computer science concepts such as pathfinding algorithms for nemesis moved and tower behaviour, game state management and user interaction design.

Additionally, the game will be developed using Object-Oriented Programming concepts such as modular coding, inheritance, polymorphism and encapsulation.

2. Requirements

Objectives

- Develop a challenging and problem solving Tower Defence game with increasing difficulty and boosting the interest of the player.

- Implement OOP principles to ensure flexibility and efficiency.

- Create a visually engaging technophile with distinct nemeses and watchtower designs.

- Provide multiple difficulty levels, ranging from beginner to expert.

- Ensuring the game is up scalable and open to future enhancements.

Problem statement

Most of the Tower defence games mainly focus on killing the rivals, they lack in keeping the players engaged. However my game aims to solve these problems by introducing the below features,

1. Creating dynamic tower structures that can be paced at a different location and different direction.

2. Tactical watchtower placement, requiring players to think ahead.

3. Adaptive nemesis behaviour, ensuring progressing challenges.

4. An evolving system with increasing difficulty levels.

By combining these elements, the game provides a progressing challenge that keeps players involved.

Target Audience

- Casual, professional and committed gamers looking for immersive problem solving experiences.

- Technophiles who enjoy futuristic game aesthetics.

- Suitable for all age groups above 12, medium complexity will suite a broader audience.

- Entry level game developers can analyse and modify the game’s OOP structure.

3. Game outline and Features

Basic Features

1. Tower Generation: Procedurally generated paths ensuring fresh challenges.

2. Watchtower: Players can deploy different types of watchtower to slow, damage, or slow nemeses.

3. Nemeses AI: Increasingly intelligent opponents adapt to player strategies.

4. Power-Ups: Items that enhance player movement or watchtower efficiency.

5. Heart points: Performance tracking for competition and progression.

6. Credit scores: Allow players to buy watchtowers to destroy the Nemesis.

Game Progression

- Level-based advancement: Players must clear multiple levels, each harder than the previous one.

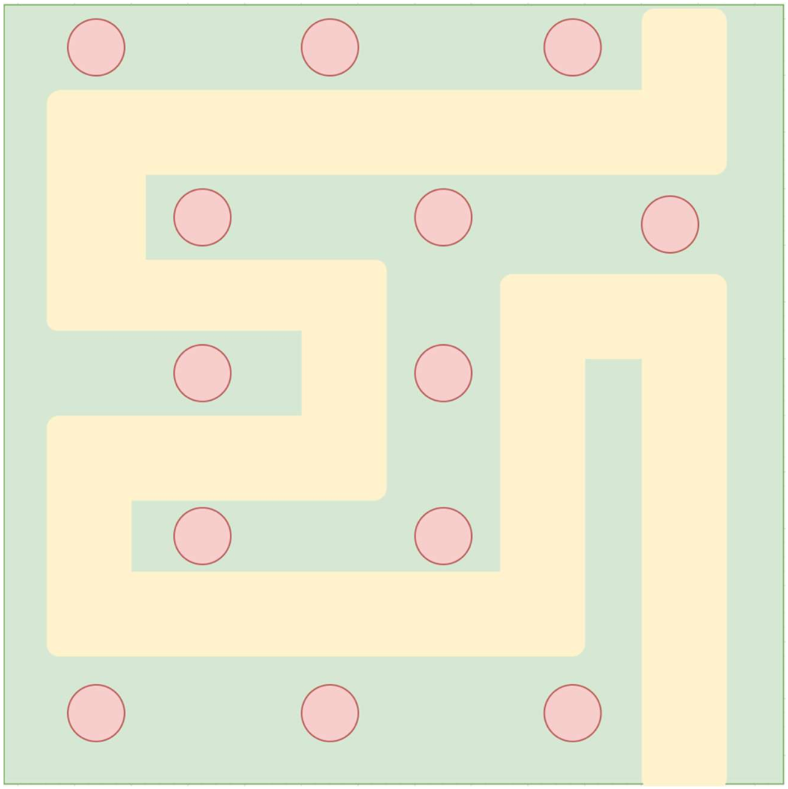
- Upgrades system: Players can improve watchtowers and abilities over time by gaining more heart points.

- Storyline elements: Background knowledge connecting the towers to a larger skiffy narrative.

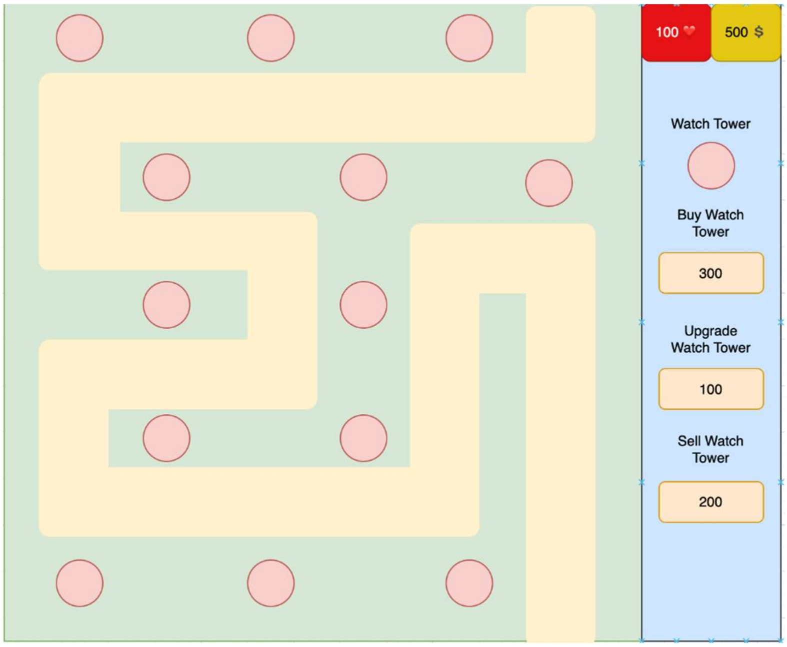
4. Initial Design Sketches

Design Sketches

a. A blank play map

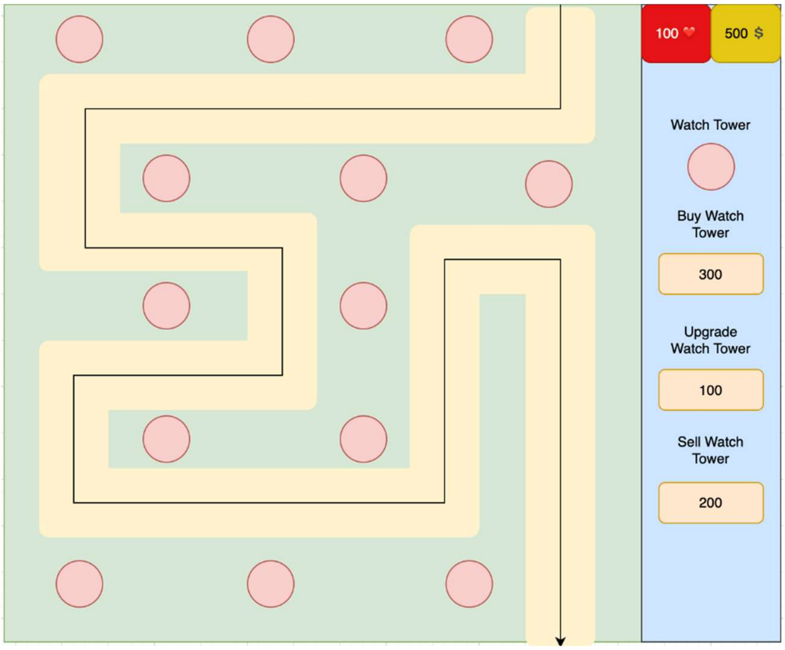


b. Initial screen with points and player life



`

c. Route map for the enemies



5. Modules of the Game

1. GameManager Module

Purpose: This module manages game variables and controls the game flow, level information, nemesis list, and money

OOP Concepts Implemented:

Concept	Implementation
Encapsulation	Nemesis, and WatchTower objects manage their own attributes, keeping their internal state hidden.
Inheritance	SniperWatchTower inherit from WatchTower, reusing base class logic.
Polymorphism	The game loop calls update() on all objects, regardless of type of the object.
Abstraction	The GameManager abstracts game logic, and update() abstracts individual object behaviour.

2. Nemesis Module

Purpose: This module generates new nemesis, manages their health, position, and movement towards the goal.

OOP Concepts Implemented:

Concept	Implementation
Encapsulation	Nemesis attributes like speed, health, and position are hidden and accessed only from methods.
Polymorphism	WatchTowers and game logic can interact with all nemesis uniformly, regardless of their specific type.
Abstraction	The GameManager class handles nemesis behaviour, hiding implementation details from the main game logic.

3. Watchtower Module

Purpose: This module allows the player to place the watchtowers in the play area for shooting the nemesis in the range, handles watchtower upgrade, and removing of watchtowers.

OOP Concepts Implemented:

Concept	Implementation
Encapsulation	Tower attributes (e.g., damage, range, attack speed) are hidden inside tower objects and accessed from methods.

Inheritance	Different types of towers (e.g., SniperTower, CannonTower) inherit from a base Tower class, reusing common functionality.

6. Pseudocode for the core game modules

6.1. Game Loop

```
// This module controls the complete game flow

CLASS GameLoop

   INITIALIZE:
       Create game window (screen)
        Set clock for time tracking
       Create empty lists for nemesis and watchtowers
       Load background image, sound effects, and other resources
       Initialize GameManager (level, map, and game data)

    FUNCTION start_game()
       WHILE player_health > 0 AND nemesis_remaining()
           CALL process_user_input()
           CALL update_game_objects()
           CALL render_graphics()
           WAIT (frame_time)
       END WHILE

       IF player_health <= 0
           DISPLAY "Game Over!"
       ELSE
           DISPLAY "You Win!"

    FUNCTION process_user_input()
       IF user_clicks_on_watchtower
           CALL place_watchtower_at(mouse_position)

       IF user_clicks_upgrade
           CALL upgrade_watchtower(selected_watchtower)

        IF player_clicks_quit
           DISPLAY "Game Over!"

    FUNCTION update_game_objects()
       FOR each watchtower IN game_watchtowers
           CALL watchtower.attack()

       FOR each nemesis IN nemesis_list
```

```
        CALL nemesis.move()
        CALL check_collisions()

    FUNCTION check_collisions()
      FOR each watchtower:
        Check if it is targeting any nemesis
              if yes, update damage to the nemesis and play shooting
           sound

   FUNCTION render_graphics()
      DRAW game_grid
      DRAW watchtowers
      DRAW nemesis
      DISPLAY UI elements
```

6.2. WatchTower Management Module

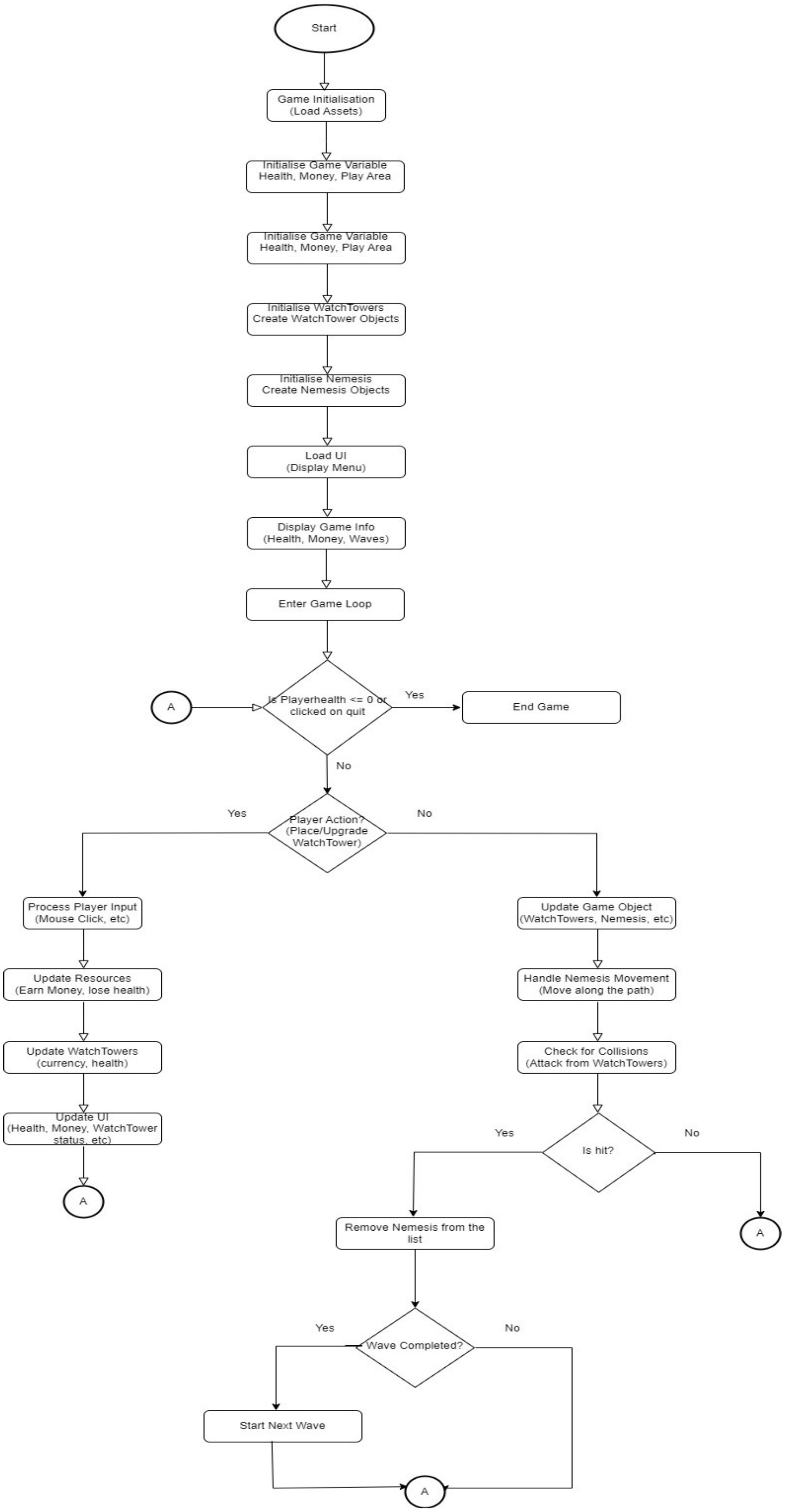
```
// This module manages the behaviour of the WatchTower

CLASS WatchTower

  INITIALIZE:
      set upgrade_level, range, cooldown, last_shot, target, and position
(tile_x, tile_y)
      set initial watchtower and animation for the watchtower
      create range circle for visual indication

   FUNCTION Attack(nemesis)
      IF nemesis WITHIN range:
        nemesis.TakeDamage(damage)
   END FUNCTION
END CLASS

FUNCTION PlaceWatchTower(type, position)
   IF player has enough resources:
      CREATE new watchtower of type at position
      DEDUCT watchtower cost from player resources
   ELSE:
      DISPLAY "Not enough resources"
   END IF
END FUNCTION

FUNCTION UpgradeWatchTower(watchtower)
   IF player has enough resources:
      watchtower.IncreaseDamage()
      watchtower.IncreaseRange()
      DEDUCT upgrade cost
```

```
    ELSE:
        DISPLAY "Not enough resources"
    END IF
END FUNCTION
```

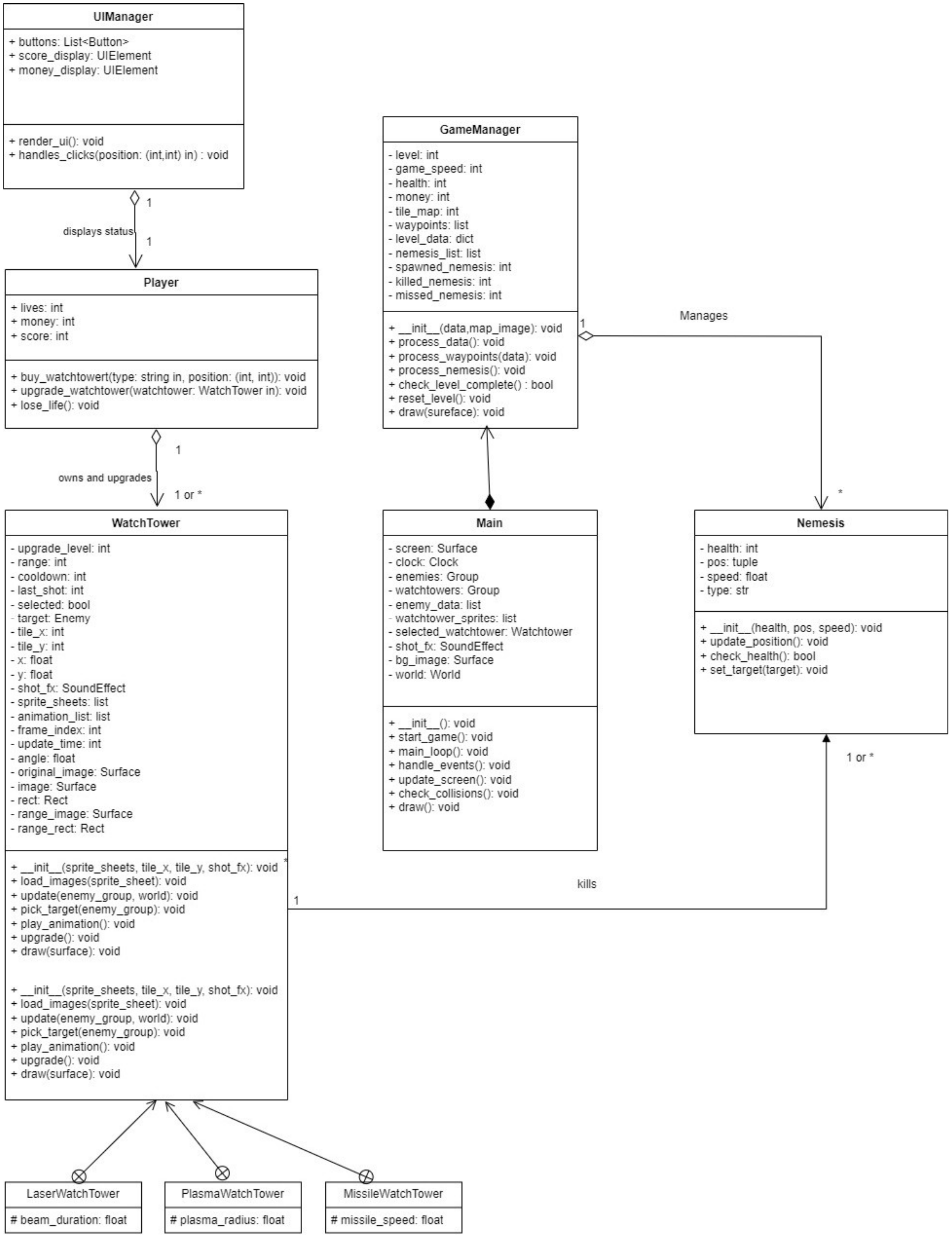
6.3. Nemesis Management Module

```
// This module manages the behaviour of the Nemesis

CLASS Nemesis

    INITIALIZE:
        Set health, position, speed based on input

    FUNCTION Move()
        UPDATE position towards goal
    END FUNCTION

    FUNCTION TakeDamage(amount)
        health = health - amount
        IF health <= 0:
            CALL OnDeath()
    END FUNCTION

    FUNCTION OnDeath()
        INCREASE player resources by reward_value
        REMOVE nemesis from game
    END FUNCTION
END CLASS

FUNCTION SpawnNemesis()
    IF spawn timer reached:
        CREATE new Nemesis at starting point
        ADD to nemesis list
    RESET spawn timer
END FUNCTION

FUNCTION UpdateNemesis()
    FOR EACH nemesis in nemesis list:
        nemesis.Move()
        IF nemesis REACHED goal:
            DECREASE player health
            REMOVE nemesis
END FUNCTION
```

7. Flow Chart

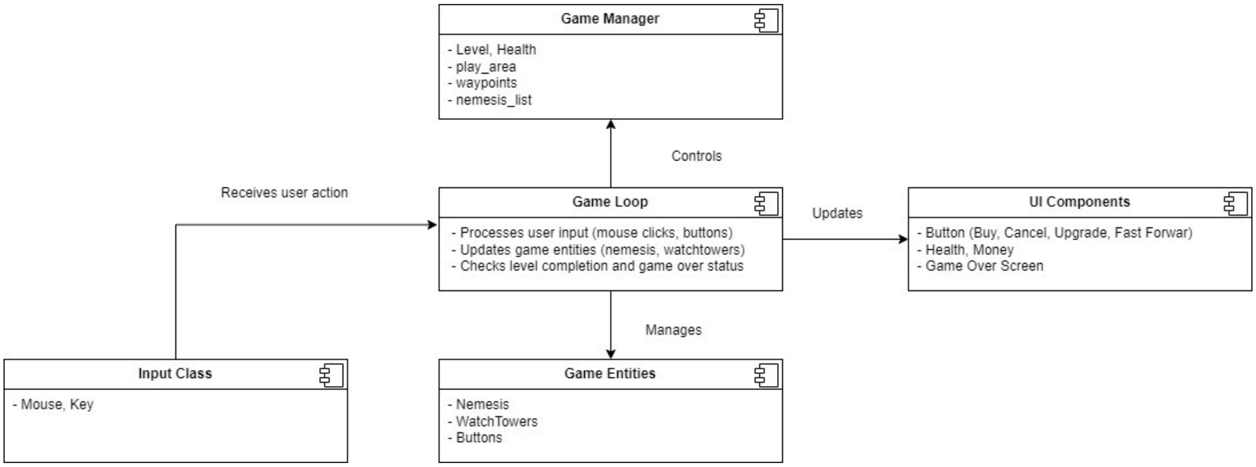

Start
Game Initialisation (Load Assets)
Initialise Game Variable Health, Money, Play Area
Initialise Game Variable Health, Money, Play Area
Initialise WatchTowers Create WatchTower Objects
Initialise Nemesis Create Nemesis Objects
Load UI (Display Menu)
Display Game Info (Health, Money, Waves)
Enter Game Loop
A
Is Playerhealth <= 0 or clicked on quit
Yes
End Game
No
Yes
Player Action? (Place/Upgrade WatchTower)
No
Process Player Input (Mouse Click, etc)
Update Resources (Earn Money, lose health)
Update WatchTowers (currency, health)
Update UI (Health, Money, WatchTower status, etc)
A
Update Game Object (WatchTowers, Nemesis, etc)
Handle Nemesis Movement (Move along the path)
Check for Collisions (Attack from WatchTowers)
Yes
Is hit?
No
Remove Nemesis from the list
A
Yes
Wave Completed?
No
Start Next Wave
A

8. Class Diagram



9. System Architecture

Below is the System Architecture connecting all modules and their interactions



10. Key Algorithms used

Alien Enigma is implemented using various algorithm. Below table highlights the key algorithms used in the game:

Where Used	Algorithm Used	Purpose
Nemesis Pathfinding	Pathfinding Navigation	Moves nemesis from the initial point towards goal
WatchTower Targeting	Nearest Enemy Targeting	Used for finding and attacking the closest nemesis in the range
Nemesis Spawning	Wave-Based Spawning	Generates nemesis dynamically based on level data
Level Progression	Enemy Kill Count Check	Determines when the level is complete

11. Design Decisions on error handling

Some of the error handling strategies that will be implemented are as follows:

- Designing the game with clear and informative error messages for the player
- Use of structured error handling (e.g., try-catch blocks) to prevent game crashes
- Validating inputs and game state before processing actions to prevent invalid operations.

1. **Input Validation**

   This is to ensure the player input (such as watchtower placement, selecting enemies, and menu interactions) is validated before it is processed. Invalid inputs are ignored without causing the game to crash.

   For example, if a player tries to place a tower in an invalid position (e.g., outside the grid or overlapping with another tower), the game will not accept the input prompting the player to try again and if a player enters an invalid key press or unresponsive input, the game will ignore the input.

2. **Boundary and Edge Case Handling**

   This is to carefully handle edge cases where the game's state might be in a rare or unexpected situation (e.g., resources running out, health dropping to zero, or having no more waves).

   In these scenarios the game never crashes and the handled gracefully by displaying a game-over screen when health reaches zero and an alert when the player runs out of resources or encounters an invalid action.

3. **Game State and Resource Integrity**

   This is to ensure that game resources, health, and other critical game variables (such as nemesis spawn counters) are never in an invalid state (e.g., negative health, impossible wave number).

   In order to prevent this, these games variables are always set to a known to start with.

12. Testing strategy

Below testing strategies is used in the game development. Where possible the testing is automated, if not they are manual done.

- Unit testing individual game components to ensure correctness.
- Integration tests to verify that different game modules interact correctly.
- Perform UI testing to ensure proper visual feedback and interaction.
- Conducting end to end testing to ensure the overall gameplay experience is enjoyable and bug-free.

a. Unit Testing

This will be an automated test where the module is broken down into smaller units (e.g., Nemesis, WatchTower, GameState) and are verified for their correctness in isolation.

This testing will be done using mock data and compared with the expected behaviours.

An example of the Unit Test:

```
  def test_watchtower_initialization(self):
      """Test if the watchtower initializes correctly."""
      self.assertEqual(self.watchtower.x, self.watchtower_x)
      self.assertEqual(self.watchtower.y, self.watchtower_y)
      self.assertEqual(self.watchtower.range, self.watchtower_range)
      self.assertEqual(self.watchtower.damage, self.watchtower_damage)
      self.assertIsNone(self.watchtower.target)


 def test_attack_nemesis(self):
      """Test if the watchtower deals damage to a nemesis."""
      initial_health = self.nemesis_in_range.health
      self.watchtower.attack(self.nemesis_in_range)
      self.assertEqual(self.nemesis_in_range.health, initial_health - self.watchtower_damage)
```

b. **Integration Testing**

This automated testing is to validate how different game modules work together. For example, test if a nemesis moves correctly while watchtowers are firing and if the resources are correctly deducted when a watchtower is upgraded.

Example Integration Test:

```
def setUp(self):
"""Set up a watchtower and a nemesis for integration testing."""
# Watchtower at (100,100) with range 75 and damage 10
self.watchtower = Watchtower(100, 100, 75, 10)
# Nemesis with 10 health
self.nemesis = Nemesis(120, 120, c.NEMESIS_SPEED, 10)
```

```
def test_nemesis_detection_and_attack(self):

"""Test if a watchtower detects and attacks a nemesis correctly.

""" # Step 1: Check if the watchtower detects the nemesis
self.assertTrue(self.watchtower.is_nemesis_in_range(self.nemesis))

# Step 2: Attack the nemesis attack = 10 damage

self.watchtower.attack(self.nemesis)

# Step 3: Check if the nemesis's health reached zero
self.assertEqual(self.nemesis.health, 0)

# Step 4: Verify that the nemesis is eliminated from the game
self.assertTrue(self.nemesis.is_dead())
```

c. UI Testing

This will be a manual testing to ensure the game’s interface is user-friendly and displays accurate information, such as health, resources, and current wave.

d. End-to-End Testing

This testing is to simulate a complete game session from start to finish to ensure that all game features and logic work as expected. This testing is conducted manually.

13. Ethical and Legal issues

Below are some of the ethical issues considered:

- Alien Engima does not use any excessive violence, or inappropriate themes. The nemesis are fictional, non-human, or robotic to avoid real-world implications.
- This game always has an end and will not promote unhealthy gaming habits.
- Game difficulty is balanced and does not unfairly punish players.

Below are some of the legal issues considered:

- All art work, sounds, and code copied from other sources will be follow the license agreement and will be appropriately acknowledged.

14. Technical Requirements and Implementation Plan

Software Requirements

- Programming Language:

  - Python v3.12.0, pygame v2.5.2, unittest v3.13.2
  - Visual Studio Code

- Version Control:

  - GitHub for source control
  - Git Bash and Source tree for committing the code and pulling from the source control.

- Graphics & Assets:

  - Custom-designed images for gameplay.

- Documentation and designing/sketching:

  - Microsoft Word for documentation
  - Draw.io for UML Designs

Hardware Requirements

- System Requirements: A PC with any operating system.

- Storage: At least 500MB for assets and game files.

Implementation Plan

1. Stage 1 (Week 1-2): Research & Initial Game Design Sketches.

Week 3 - Deliverable 1 – Project proposal and equipment specification.

2. Stage 2 (Week 3-4): Develop the Game Engine and Player Movement.

3. Stage 3 (Week 5-6): Implement WatchTower and Nemesis AI Mechanics.

Week 5 - Deliverable 2 – Detailed project design document.

4. Stage 4 (Week 7-8): Refine Game Levels and Introduce Power-Ups.

Week 7 - Deliverable 3 – Initial Prototype / Development milestone.

5. Stage 5 (Week 9-10): Testing, Debugging, and Performance Optimization.

Week 9 - Deliverable 4 – Intermediate project update.

Week 11 – Final Delivery – Final Project Submission.

15. Challenges and Future Scope

Potential Challenges

- Balancing difficulty levels, to ensure impartiality and engagement.
- Optimizing AI behaviour for smooth performance and pragmatism.
- Using high quality 3D images without affecting game performance.

Scope for enhancement:

- Enhancement to include multiplayer features.
- Additional game modes, like time-based challenges and persistence modes.

16. GitHub Repository and Conclusion

For ongoing development, access the source code and latest updates here:

GitHub Repository: Alien Enigma
https://github.com/skeerthish/AlienEnigma

Conclusion

Tower Defence: The Alien Enigma is designed to push players' strategic thinking while providing an immersive experience. With structured OOP design and sensibly created game mechanics, this game will provide both technical and entertainment value.

This document serves as a broad project proposal and requirements specification, guiding the structured development of the game. Additional enhancements will be made based on game development, iterative testing and feedback.