Project final report

Curriculum: **Practices in Software Development**

Department: **Software Engineering**

Major: **Software Engineering**

Student Name: **OUALID DADAH**

Instructor: **Dr. Yi Zhang**

Dec. 22th 2024

Table of Contents

1. Introduction
2. Game Logics
3. Key Design & Difficulties
4. Conclusions

Introduction

- Tower Defense games represent a significant genre in the gaming industry, combining strategic thinking with resource management. This project implements a Tower Defense game using Python and the Pygame framework, creating an engaging experience where players must defend their territory through strategic tower placement and management.

Core Mechanics and Unique Features

- The game implements several fundamental mechanics that create engaging gameplay. The resource management system requires players to balance their economy between defensive expansion and upgrade investments. Each successful enemy elimination rewards players with currency, creating a dynamic economy that responds to player performance.
- Unique to this implementation is the sophisticated turret targeting system. Rather than using simple proximity-based targeting, turrets employ vector calculations to predict enemy movements and adjust their firing angles accordingly. This creates more realistic and challenging gameplay, as players must consider both positioning and timing in their strategic decisions.

The wave-based spawning system introduces varying combinations of enemy types, each with distinct attributes:

1. Weak enemies test basic defensive setups
2. Medium enemies require more coordinated defense
3. Strong enemies challenge resource management
4. Elite enemies demand optimal turret placement and upgrading

Development Process and Technical Challenges

- The development process encountered several significant technical challenges. One primary challenge involved implementing smooth enemy movement along predefined paths. This was solved through vector-based movement calculations, ensuring consistent enemy speed regardless of path complexity. The solution required careful manipulation of pygame.math.Vector2 objects and normalized vectors for precise movement control.
- Another significant challenge arose in creating an efficient turret targeting system. The implementation needed to balance accuracy with performance, particularly when multiple turrets were tracking numerous enemies simultaneously. This was resolved through optimized mathematical calculations and efficient use of Pygame's sprite grouping system.

- The upgrading system presented unique challenges in managing state transitions and maintaining game balance. Each turret upgrade needed to meaningfully improve capabilities while preserving strategic depth. This required careful tuning of parameters such as range increases and cooldown reductions.

Architecture and Code Organization

- The project architecture emphasizes modularity and maintainability through clear separation of concerns. The main game loop coordinates various components while maintaining loose coupling between systems. This architecture provides several advantages:
- The enemy management system operates independently, handling movement, health, and state transitions. The turret system manages targeting, upgrades, and animation states without direct dependency on other systems. The world management system coordinates game state, resource management, and level progression through well-defined interfaces.
- Code organization follows object-oriented principles, with each major component encapsulated in its own module. This approach facilitates testing, debugging, and future enhancements. The use of type hints throughout the codebase enhances code readability and helps prevent potential runtime errors.

Development Tools and Practices

- The development process utilized several key tools and practices to ensure code quality and maintainability. Version control through Git helped track changes and manage feature implementations. Type hints were employed throughout the codebase to enhance code reliability and developer experience.
- Documentation was prioritized, with comprehensive docstrings providing clear information about function parameters, return values, and expected behaviors. This documentation approach helps future developers understand the codebase and facilitates maintenance and updates.

Performance Considerations

- Early in development, performance optimization became a crucial consideration. The game needed to maintain smooth performance even with numerous enemies and turrets on screen. This led to several optimization strategies.
- Sprite group management was optimized to minimize unnecessary collision checks. Image loading was streamlined to reduce memory usage, with sprites and animations carefully managed to prevent memory leaks. The game loop was optimized to maintain consistent frame rates even during intense gameplay moments.

The project structure is organized as follows:

- **Core Game Scripts**:
  - `main.py`: Serves as the entry point of the game, managing the primary game loop, event handling, and overall integration of game elements.

    - turret.py: Defines the properties and functionalities of turrets, including placement logic, targeting systems, and firing mechanisms.
    - enemy.py: Manages enemy behavior, including movement patterns, health attributes, and interactions with turrets and projectiles.

- **Visual Assets**:
    - High-quality graphical resources for enemies, turrets, and GUI elements provide visual clarity and enhance the player’s experience.

- **Map Designs**:
    - Levels are crafted using the Tiled map editor, leveraging tile-based layouts to create complex paths and dynamic environments that challenge players’ strategies.

- **Audio Assets**:
    - Immersive sound effects, such as turret firing and enemy hits, contribute to a compelling gameplay atmosphere.

Game Logics

Enemy System Implementation

- The enemy system represents one of the most complex aspects of the game architecture, implementing multiple interrelated systems for movement, health management, and state tracking.

Movement and Rotation Mechanics

- The enemy movement system utilizes PyGame's Vector2 class for precise position calculations and smooth rotations. Each enemy maintains its current position and rotation state through a sophisticated vector-based calculation system:

```
def rotate(self):
      dist = self.target - self.pos
      self.angle = math.degrees(math.atan2(-dist[1], dist[0]))
      self.image = pg.transform.rotate(self.original_image, self.angle)
      self.rect = self.image.get_rect()
      self.rect.center = self.pos
```

- This rotation system ensures enemies always face their movement direction, creating fluid visual feedback. The rotation algorithm calculates the angle between the current position and target waypoint, applying the appropriate transformation to the enemy sprite.

Enemy Types and Attributes

- The game implements a hierarchical enemy structure through the enemy_data.py configuration:

```
ENEMY_DATA = {
      "weak": {
      "health": 10,
      "speed": 2
      },
      "medium": {
      "health": 15,
      "speed": 3
      },
      "strong": {
      "health": 20,
      "speed": 4
      },
      "elite": {
      "health": 30,
      "speed": 6
      }
}
```

- Each enemy type's attributes directly influence their behavior and challenge level:

  1. The weak enemy type serves as the basic unit, with balanced health and speed attributes making them predictable threats. Medium enemies introduce increased challenge through higher speed and health values, requiring more focused defensive attention. Strong enemies

represent a significant threat escalation, demanding efficient turret placement and upgrades. Elite enemies constitute the highest threat level, combining high health pools with rapid movement speed that can quickly overwhelm inadequate defenses.

Wave Generation System

- The wave generation system employs a sophisticated spawning mechanism defined in ENEMY_SPAWN_DATA:

```
ENEMY_SPAWN_DATA = [
      {
      "weak": 15,
      "medium": 0,
      "strong": 0,
      "elite": 0
      },
      # Progressive difficulty increase through subsequent levels
      {
      "weak": 5,
      "medium": 20,
      "strong": 0,
      "elite": 0
      }
]
```

- This spawning system creates dynamic difficulty progression through carefully balanced enemy combinations. Early waves focus on weak enemies to allow players to establish basic defenses, while later waves introduce increasingly complex enemy type combinations that test different aspects of the player's strategy.

Turret Defense Implementation

Targeting System Architecture

- The turret targeting system implements a sophisticated detection and engagement mechanism:

```
def update(self, enemy_group, world):
      if self.target:
      self.play_animation()
      else:
      if pg.time.get_ticks() - self.last_shot > (self.cooldown / world.game_speed):
            self.pick_target(enemy_group)
```

- This system maintains target tracking while implementing cooldown management for balanced gameplay. The targeting algorithm considers multiple factors:

    1. The range calculation uses Pythagorean theorem for accurate distance measurement, ensuring consistent targeting regardless of enemy approach angle. Target selection prioritizes enemies within range through efficient iteration of the enemy group. The cooldown system scales with the game's speed setting, maintaining consistent challenge during fast-forward gameplay.

Turret Upgrade Progression

- The upgrade system implements a four-tier progression structure through TURRET_DATA:

```
TURRET_DATA = [
      {
      "range": 90,
      "cooldown": 1500,
      },
      {
      "range": 110,
      "cooldown": 1200,
      },
      {
      "range": 125,
      "cooldown": 1000,
      },
      {
      "range": 150,
      "cooldown": 900,
      }
]
```

- Each upgrade level introduces significant improvements to turret capabilities:
  - Level 1 provides balanced starting capabilities
  - Level 2 increases range by 22% and reduces cooldown by 20%
  - Level 3 further extends range and reduces cooldown for enhanced performance
  - Level 4 represents maximum efficiency with optimal range and minimal cooldown

Animation System

- The turret animation system manages visual feedback through frame-based animation:

```
def play_animation(self):
      self.original_image = self.animation_list[self.frame_index]
      if pg.time.get_ticks() - self.update_time > c.ANIMATION_DELAY:
      self.update_time = pg.time.get_ticks()
      self.frame_index += 1
      if self.frame_index >= len(self.animation_list):
            self.frame_index = 0
            self.last_shot = pg.time.get_ticks()
            self.target = None
```

- The animation system coordinates with the targeting system to provide visual feedback for turret actions. Frame timing is managed through PyGame's time system, ensuring smooth animations regardless of game speed settings.

Resource and Economy System

Economic Balance

- The economy system implements multiple resource flows:

```
def check_alive(self, world):
      if self.health <= 0:
      world.killed_enemies += 1
      world.money += c.KILL_REWARD
      self.kill()
```

- Initial resource allocation provides players with 650 currency units, allowing for strategic early-game decisions. The reward system implements multiple revenue streams:

    - Base enemy elimination reward (KILL_REWARD = 1)
    - Level completion bonus (LEVEL_COMPLETE_REWARD = 100)
    - Strategic resource management through turret placement (BUY_COST = 200)
    - Upgrade investments (UPGRADE_COST = 100)

Game State Management

Level Progression System

- The level management system tracks multiple state variables:

```
def check_level_complete(self):
      return (self.killed_enemies + self.missed_enemies) == len(self.enemy_list)
```

```python
def reset_level(self):
      self.enemy_list = []
      self.spawned_enemies = 0
      self.killed_enemies = 0
      self.missed_enemies = 0
```

- This system maintains level progression while tracking player performance metrics. The completion check ensures all enemies are accounted for before advancing to the next level, while the reset function prepares the game state for the next challenge.

Key Design & Difficulties

Object-Oriented Architecture

- The foundation of this tower defense game rests upon a carefully structured object-oriented architecture. This section explores the core design decisions, implementation challenges, and solutions that I developed throughout the project.

Component Interaction System

- At the heart of the game lies a sophisticated component interaction system based on the Observer pattern. Each major component maintains its own state while communicating through well-defined interfaces. The World class serves as the central nervous system of the game, coordinating all other components through a hierarchical structure:

```
class World:
      def __init__(self, data: Dict, map_image: pg.Surface):
      self.level = 1
      self.game_speed = 1
      self.health = c.HEALTH
      self.money = c.MONEY
      self.tile_map = []
      self.waypoints: List[Tuple[float, float]] = []
```

- This centralized state management requires careful orchestration of updates, following a strict order: World state updates occur first, followed by Enemy positions and states, then Turret targeting and firing logic, and finally visual updates. This hierarchy prevents race conditions and maintains game state consistency.

Type System Implementation

- The implementation leverages Python's type hinting system extensively to ensure code reliability and maintainability. This approach manifests clearly in the Turret class implementation:

```
class Turret(pg.sprite.Sprite):
      def __init__(self, sprite_sheets: List[Surface],
            tile_x: int,
            tile_y: int,
            shot_fx: pg.mixer.Sound):
```

- The typing system provides early error detection and improves code documentation. It handles generic type parameters for collections, union types for optional values, and protocol implementations for pygame interfaces. This strong typing foundation proved invaluable during development iterations.

Performance Optimization Challenges

Sprite Management and Memory Optimization

- The game's sprite management system implements several sophisticated optimization techniques. The core of this system lies in the efficient handling of sprite sheets:

```
def load_images(self, sprite_sheet: Surface) -> List[Surface]:
      size = sprite_sheet.get_height()
      animation_list = []
      for x in range(c.ANIMATION_STEPS):
      temp_img = sprite_sheet.subsurface(x * size, 0, size, size)
      animation_list.append(temp_img)
      return animation_list
```

- This implementation minimizes texture switches during rendering and reduces memory fragmentation while maintaining smooth animations. The animation state management system handles frame timing synchronization and efficient image references:

```
def play_animation(self) -> None:
      self.original_image = self.animation_list[self.frame_index]
      if pg.time.get_ticks() - self.update_time > c.ANIMATION_DELAY:
      self.update_time = pg.time.get_ticks()
      self.frame_index += 1
      if self.frame_index >= len(self.animation_list):
            self.frame_index = 0
            self.last_shot = pg.time.get_ticks()
            self.target = None
```

Collision Detection System

- The collision detection system implements spatial partitioning for improved performance. The implementation optimizes range calculations through mathematical efficiency:

```
def pick_target(self, enemy_group: Group) -> None:
      for enemy in enemy_group:
      if enemy.health > 0:
            x_dist = enemy.pos[0] - self.x
            y_dist = enemy.pos[1] - self.y
            dist_squared = x_dist ** 2 + y_dist ** 2
            range_squared = self.range ** 2
            if dist_squared < range_squared:
            self.target = enemy
            self.angle = math.degrees(math.atan2(-y_dist, x_dist))
```

- This approach avoids unnecessary square root calculations and implements early exit conditions, significantly improving performance during heavy combat scenarios.

Enemy Pathfinding System

Vector-Based Movement Implementation

- The movement system employs sophisticated vector mathematics to ensure smooth enemy movement across the game map:

```
def move(self, world) -> None:
        if self.target_waypoint < len(self.waypoints):
        self.target = Vector2(self.waypoints[self.target_waypoint])
        self.movement = self.target - self.pos

        dist = self.movement.length()
        movement_this_frame = self.speed * world.game_speed

        if dist >= movement_this_frame:
                normalized_movement = self.movement.normalize()
                self.pos += normalized_movement * movement_this_frame
        else:
                if dist != 0:
                self.pos += self.movement.normalize() * dist
                self.target_waypoint += 1
```

- This implementation provides smooth interpolation between waypoints while handling variable game speeds and maintaining precise position tracking.

Waypoint Processing System

- The waypoint system processes level data through a sophisticated pipeline:

```
def process_waypoints(self, data: List[Dict[str, float]]) -> None:
        for point in data:
        temp_x = point.get("x")
        temp_y = point.get("y")
        if temp_x is not None and temp_y is not None:
                world_x = temp_x + self.offset_x
                world_y = temp_y + self.offset_y
                self.waypoints.append((world_x, world_y))
```

Game Balance and Progression System

Dynamic Difficulty Scaling

- The enemy spawning system implements a carefully crafted difficulty curve through structured data:

```
ENEMY_SPAWN_DATA = [
      {
      "weak": 15,
      "medium": 0,
      "strong": 0,
      "elite": 0
      },
      {
      "weak": 30,
      "medium": 0,
      "strong": 0,
      "elite": 0
      }
]

ENEMY_DATA = {
      "weak": {
      "health": 10,
      "speed": 2
      }
}
```

- This system creates progressive difficulty scaling through dynamic enemy type mixing while maintaining balanced resource distribution and adaptive challenge levels.

Economic System Implementation

- The economy system manages resource flow through a carefully balanced progression model:

```
class World:
      def __init__(self):
      self.money = c.MONEY  # Starting money: 650

      def update_economy(self):
      self.money += self.killed_enemies * c.KILL_REWARD

      if self.level_complete:
            self.money += c.LEVEL_COMPLETE_REWARD

      if self.upgrade_purchased:
            self.money -= c.UPGRADE_COST
```

Memory Management and Resource Allocation

Surface Caching System

- The surface caching system manages memory usage through an intelligent caching mechanism:

```
class SurfaceCache:
      def __init__(self, max_size: int = 1000):
      self._cache = {}
      self._access_count = {}
      self._max_size = max_size

      def get_surface(self, key: str, creator_func) -> Surface:
      if key not in self._cache:
            if len(self._cache) >= self._max_size:
            self._evict_least_used()
            self._cache[key] = creator_func()
      self._access_count[key] = self._access_count.get(key, 0) + 1
      return self._cache[key]
```

Dynamic Resource Management

- Resource management handles asset loading and unloading based on game state:

```
class ResourceManager:
      def __init__(self):
      self._current_level_resources = set()
      self._global_resources = set()
      self._loaded_resources = {}

      def prepare_level(self, level_number: int):
      required_resources = self._get_level_requirements(level_number)
      self._unload_unnecessary_resources(required_resources)
      self._load_required_resources(required_resources)
```

Technical Limitations and Future Improvements

- The current implementation presents several opportunities for enhancement:

```
class AssetManager:
      def __init__(self):
```

```
      self.cache = {}

      def load_asset(self, path: str) -> Surface:
      if path not in self.cache:
            self.cache[path] = pg.image.load(path).convert_alpha()
      return self.cache[path]
```

- The targeting system could be enhanced with predictive capabilities:

```
def predict_target_position(self, enemy: Enemy) -> Vector2:
      time_to_impact = self.calculate_time_to_impact(enemy)
      predicted_pos = enemy.pos + (enemy.movement * time_to_impact)
      return predicted_pos
```

Future Development Roadmap

- Looking forward, several technical improvements could enhance the game:

  1. Implementation of a dynamic asset loading system to improve memory management
  2. Development of an advanced pathfinding system using A* algorithm
  3. Enhancement of the turret targeting system with predictive capabilities
  4. Implementation of a more sophisticated UI framework for better scalability
  5. Integration of a component-based architecture for improved modularity

Conclusion

This project has served as a comprehensive exploration of game development, blending theoretical concepts with practical implementation to deliver a functioning game prototype. Through the course of this endeavor, significant progress was made in understanding the complexities of game mechanics, logic design, and user interaction.

The development process provided valuable insights into the intricacies of designing engaging gameplay while addressing technical challenges. From implementing the core game logic to refining the user experience, each phase required meticulous planning and problem-solving. These efforts have resulted in a product that aligns closely with the initial objectives and showcases the creative and technical skills involved.

One of the key highlights of the project was tackling unforeseen difficulties, such as optimizing performance and resolving design inconsistencies. Overcoming these obstacles not only enhanced the quality of the final output but also provided a practical learning experience in managing software development challenges.

Furthermore, the project underscored the importance of iterative testing and feedback in achieving a balance between playability and technical efficiency. Each iteration brought improvements, reinforcing the necessity of adaptability and continuous refinement in game development.

In summary, this project has been instrumental in advancing skills in software engineering and game design. It has demonstrated the value of combining creative thinking with technical acumen to produce a cohesive and enjoyable gaming experience. The knowledge and expertise gained through this process will undoubtedly contribute to future endeavors in software and game development, paving the way for even more ambitious projects.