Aetherbound

Game Development Documentation

Project Proposal

Initial Objectives

The primary objective of this project was to develop a multi-game system that would allow seamless selection and transition between multiple games through a unified menu interface.

Original Plan vs. Actual Implementation

Initial Proposal (Submission 1):

- Planned to develop a Street Fighter replica as one of the games in the multi-game system
- Focused on fighting game mechanics and character-based combat

Revised Approach:

- Switched to developing a Legend of Zelda-inspired action RPG (Aetherbound)
- The decision was influenced by feedback on the complexity of the previous project and my personal interest in RPG mechanics.

Multi-Game System Vision

The unified menu system is designed to:

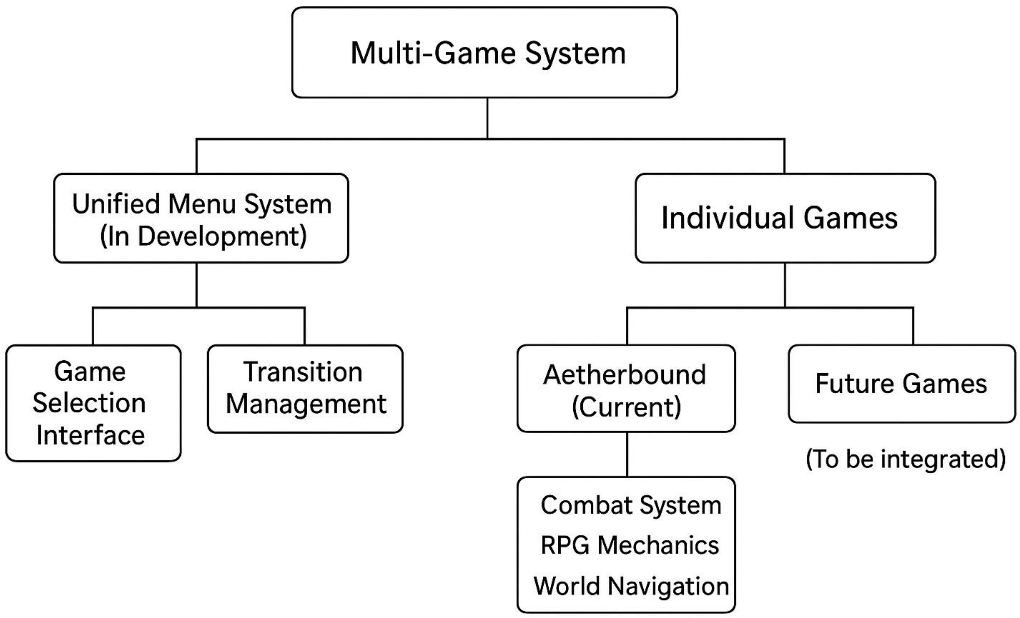
- Provide a central hub for game selection
- Enable seamless transitions between different game genres
- Maintain consistent UI/UX across all games
- Allow independent game development while ensuring integration compatibility

Current Status

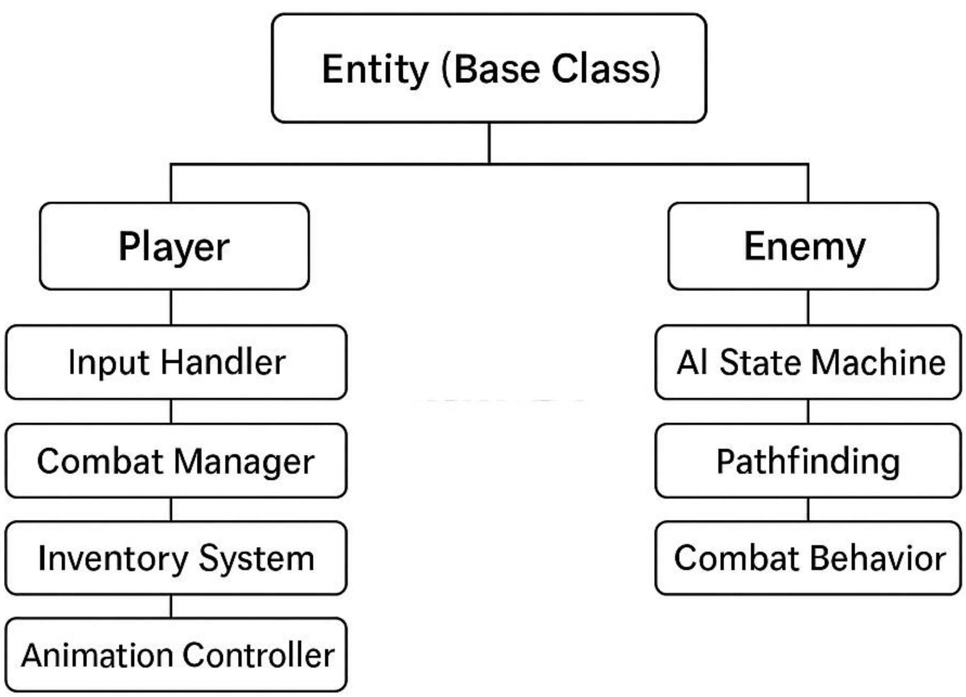
- **Primary Game**: Aetherbound (Legend of Zelda-style RPG) - Currently implemented
- **Menu System**: Start screen implemented as foundation for future multi-game menu
- **Future Integration**: Additional games to be added through the unified menu system

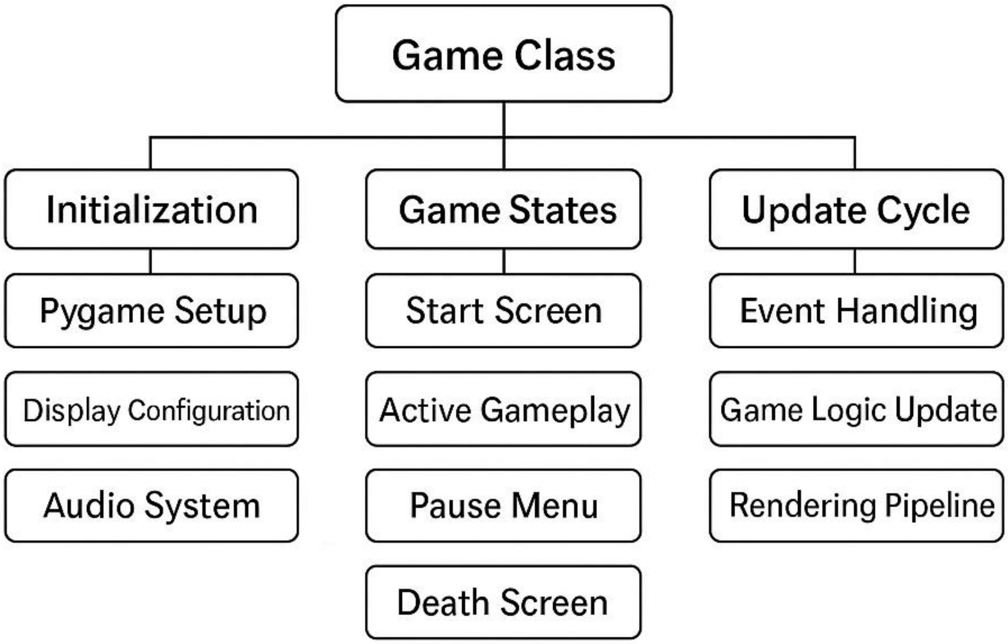
Design Documentation

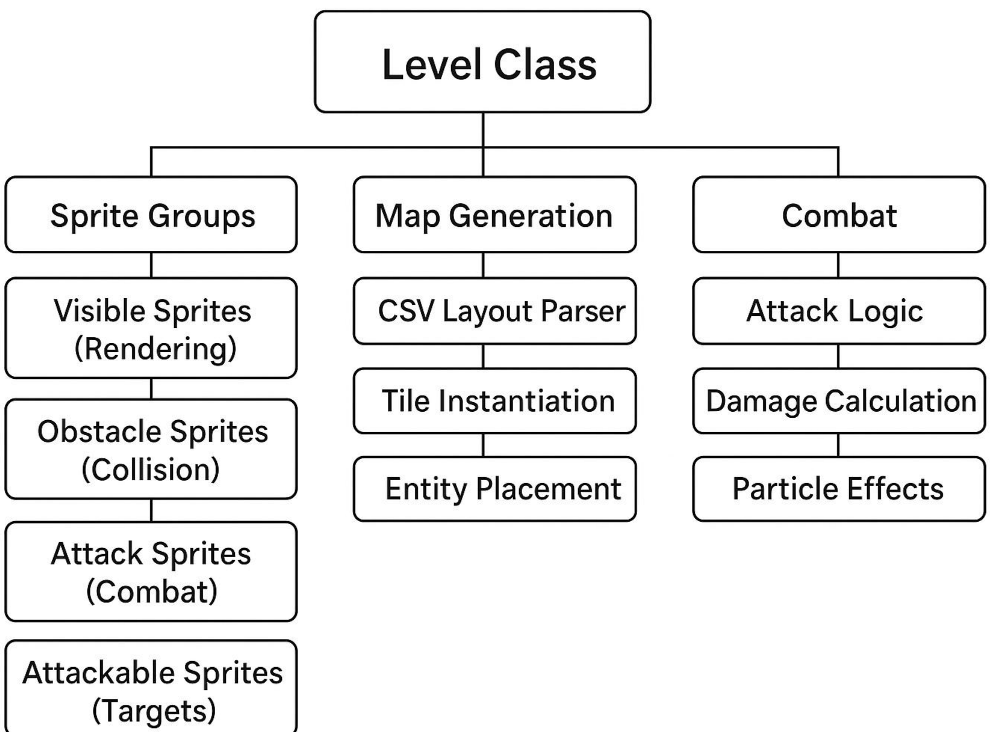
Overall System Architecture



Aetherbound Game Structure




Game Class
Initialization
Game States
Update Cycle
Pygame Setup
Start Screen
Event Handling
Display Configuration
Active Gameplay
Game Logic Update
Audio System
Pause Menu
Rendering Pipeline
Death Screen


Level Class
Sprite Groups
Map Generation
Combat
Visible Sprites (Rendering)
CSV Layout Parser
Attack Logic
Obstacle Sprites (Collision)
Tile Instantiation
Damage Calculation
Attack Sprites (Combat)
Entity Placement
Particle Effects
Attackable Sprites (Targets)

Weekly Progress Reports

Week 1. Foundation and System Architecture

Status: Initialization & Design Consolidation

Objectives:

- Establish environment and toolchain.
- Finalize design scope and technical architecture for the Zelda-style RPG.

Tasks:

1. Configure Python and Pygame environment.
2. Define high-level system architecture (Entity Component System, map design, AI framework).

Week 2. Core System Implementation

Status: Implementation Phase

Objectives:

- Implement foundational gameplay systems enabling player movement and basic world interaction.

Tasks:

1. Develop Entity and Player base classes with vector-based physics.
2. Implement:
    - Movement and input system with diagonal normalization.
    - Collision detection (axis-separated).
    - Animation manager.
3. Establish placeholder map tiles for navigation testing.
4. Begin integration testing of player physics and animation.

Week 3. Combat, AI, and Progression Systems

Status: Mid-Development Integration

Objectives:

- Develop combat logic, enemy AI, and player progression to achieve a complete gameplay loop.

Tasks:

1. Implement:
    - Weapon and magic systems (flame/heal).
    - Enemy AI finite state machine (idle, move, attack).
    - Distance-based pathfinding and attack triggers.
2. Integrate combat collision detection and damage calculation.
3. Introduce progression subsystems:
    - Experience and stat upgrade system (five attributes).
    - Exponential cost scaling and stat capping.
4. Add UI elements:
    - Health/energy bars.
    - Weapon/magic selection.

Technical Focus:

- Attack cooldown management and invulnerability frames.
- Input cooldown logic.

Week 4. System Integration, Testing, and Refinement

Status: Optimization Phase

Objectives:

- Integrate all subsystems into a stable version with good performance and visual feedback.

Tasks:

1. Conduct focused testing:
    - Combat balance, AI consistency, and UI functionality.
    - Collision accuracy and frame-rate stability.
2. Optimize:

- Sprite rendering.
- Physics and collision loops.

Timeline

Week	Phase	Primary Focus	Deliverables
1	Foundation	Environment setup, architecture, and design documentation	System design package
2	Core Systems	Entity, movement, and animation systems	Playable movement prototype
3	Gameplay Systems	Combat, AI, and progression integration	Functional gameplay loop
4	Integration & Testing	System unification, performance tuning, testing	Stable prototype & progress report

Algorithm Choices & Implementation

Finite State Machine (FSM)

```
def get_status(self, player):
  distance = self.get_player_distance_direction(player)[0]

  if distance <= self.attack_radius and self.can_attack:
    if self.status != 'attack':
      self.frame_index = 0  # Reset animation
    self.status = 'attack'
  elif distance <= self.notice_radius:
    self.status = 'move'
  else:
    self.status = 'idle'
```

Why This Approach:

1. **Simplicity**: Three states (idle, move, attack) cover all enemy behaviors
2. **Extensibility**: Easy to add new states (patrol, flee, etc.)
3. **Clarity**: Clear separation of behaviors for each enemy type

State Transition Logic:

- **Idle → Move**: Player enters notice_radius
- **Move → Attack**: Player enters attack_radius AND cooldown expired
- **Attack → Idle**: Player exits notice_radius
- **Attack → Move**: Attack cooldown active, player in notice_radius

Enemy Specific Parameters

```
monster_data = {
  'squid': {'attack_radius': 80, 'notice_radius': 360},
  'raccoon': {'attack_radius': 120, 'notice_radius': 400},
  'spirit': {'attack_radius': 60, 'notice_radius': 350},
  'bamboo': {'attack_radius': 50, 'notice_radius': 300}
}
```

Axis-Aligned Bounding Box (AABB) with Axis Separation - Collision System

```
def move(self, speed):
  if self.direction.magnitude() != 0:
    self.direction = self.direction.normalize()

  # Separate X and Y collision detection
  self.hitbox.x += self.direction.x * speed
  self.collision('horizontal')
```

```
    self.hitbox.y += self.direction.y * speed
    self.collision('vertical')

    self.rect.center = self.hitbox.center

def collision(self, direction):
    if direction == 'horizontal':
        for sprite in self.obstacle_sprites:
            if sprite.hitbox.colliderect(self.hitbox):
                if self.direction.x > 0:  # Moving right
                    self.hitbox.right = sprite.hitbox.left
                if self.direction.x < 0:  # Moving left
                    self.hitbox.left = sprite.hitbox.right

    if direction == 'vertical':
        for sprite in self.obstacle_sprites:
            if sprite.hitbox.colliderect(self.hitbox):
                if self.direction.y < 0:  # Moving up
                    self.hitbox.top = sprite.hitbox.bottom
                if self.direction.y > 0:  # Moving down
                    self.hitbox.bottom = sprite.hitbox.top
```

Why This Approach:

Flexibility: Configurable hitbox offsets per sprite type

```
HITBOX_OFFSET = {
    'player': -26,
    'object': -40,
```

```
    'grass': -10,
    'invisible': 0
}
```

Technical Details:

- Hitboxes are smaller than visual sprites for better feel
- Collision resolves by moving entity to edge of obstacle
- Direction checked to determine which edge to align to
- Separate rect and hitbox allows visual sprite != collision bounds

Vector Mathematics for Pathfinding

```
def get_player_distance_direction(self, player):
    enemy_vec = pygame.math.Vector2(self.rect.center)
    player_vec = pygame.math.Vector2(player.rect.center)

    # Calculate vector from enemy to player
    distance = (player_vec - enemy_vec).magnitude()

    if distance > 0:
        # Normalize to unit vector for consistent speed
        direction = (player_vec - enemy_vec).normalize()
    else:
        direction = pygame.math.Vector2()  # Zero vector

    return (distance, direction)
```

Why This Approach:

Real-Time Updates: Recalculates every frame for responsive feeling

Limitations & Future Enhancements:

- No obstacle avoidance (walks through walls until collision)
- Future: Implement A* for smarter pathfinding around obstacles
- Current approach sufficient for open-area combat