Project Assumptions Document

ProjectOOP: Subway Surfer Game

December 4, 2025

1 Overview

This document outlines the key assumptions and design decisions made during the development of ProjectOOP, an endless runner game built with C++ and SFML.

Contents

2 General Project Assumptions

2.1 Development Environment

- **Language**: C++ (C++17 or later) due to the use of `std::filesystem`.
- **Graphics Library**: SFML (Simple and Fast Multimedia Library).
- **Audio Library**: SFML Audio module for sound effects and music.
- **Build System**: Visual Studio project file (.vcxproj) and Makefile for cross-platform support.
- **Platform**: Windows as primary platform (evidenced by USER environment variable and PowerShell commands in documentation).

2.2 Game Genre and Mechanics

- **Genre**: Endless runner (similar to Subway Surfers or Temple Run).
- **Perspective**: 2D side-scrolling or pseudo-3D.
- **Core Mechanics**: Running, jumping, sliding, lane switching.
- **Objective**: Survive as long as possible while collecting coins and avoiding obstacles.

3 Architecture and Design Assumptions

3.1 Design Patterns

3.1.1 Singleton Pattern

Assumption: `ResourceManager` should be a singleton to ensure a single, global point of access for all game resources.

Rationale:

- Prevents multiple instances from loading duplicate textures and sounds.
- Provides centralized resource management.
- Reduces memory consumption and improves performance.

Implementation: Static instance with deleted copy constructor and assignment operator.

3.1.2 Template Pattern (Container)

Assumption: The project should avoid using STL containers like `std::vector` for game objects.

Rationale:

- Academic requirement to demonstrate custom data structure implementation.
- Better understanding of memory management and pointer semantics.
- Requirements constraint: "No vector".

Implementation: Custom `GameList<T>` template class for managing game objects.

3.1.3 Inheritance Hierarchy

Assumption: Game objects should follow a clear inheritance hierarchy.

Design:

```
GameObject (abstract base)
 Obstacle (abstract)
    TrainObstacle
    BarrierObstacle
    ConeObstacle
    FenceObstacle
 PowerUp (abstract)
    MagnetPowerUp
    JetpackPowerUp
    ShieldPowerUp
    DoubleCoinPowerUp
 Coin (concrete)
```

Rationale:

- Promotes code reuse through polymorphism.
- Enables generic handling through base class pointers.
- Supports adding new game objects without modifying existing code (Open/Closed Principle).

3.2 State Management

3.2.1 Player States

Assumption: The player can be in one of three mutually exclusive states at any time.

- `RUNNING`: Default state.
- `JUMPING`: Temporary state with vertical movement.
- `SLIDING`: Temporary state with reduced hitbox.

Rationale: Simplifies collision detection and animation logic.

3.2.2 Game States

Assumption: The game has distinct states that affect rendering and input handling.

- Menu, Registration, Playing, Paused, Game Over, Highscore View.

3.3 Memory Management

Assumption: Use smart pointers for ownership and raw pointers for non-owning references.

Implementation:

- `std::unique_ptr` for owned objects (e.g., `mPlayer`, `mTrackManager`).
- `std::unique_ptr` passed to `GameList::add()` and released to raw pointer for internal storage.
- Manual `delete` in `GameList` destructor.

4 File Handling Assumptions

4.1 High Score Persistence

Assumption: The game stores only the highest score ever achieved, not a full leaderboard.

File Location: `data/highscore.txt`

Format:

```
<player_name>
<score_value>
```

4.2 Score History (Extended Feature)

Assumption: A separate file stores all game sessions for historical tracking. **File Location**: `data/scores.txt`

4.3 Auto-Save Strategy

Assumption: Save immediately when a new high score is achieved, rather than only on game exit.

5 Gameplay Assumptions

5.1 Lane System

Assumption: The game uses a 3-lane system (left, center, right). **Lane Indices**: 0 = left, 1 = center, 2 = right.

5.2 Collision Detection

Assumption: Use AABB (Axis-Aligned Bounding Box) collision detection via `sf::FloatRect::intersects()`.

5.3 Difficulty Progression

Assumption: Game difficulty increases over time by increasing speed. Spawn intervals decrease as speed increases.

5.4 Scoring System

Assumption: Multiple scoring mechanisms provide varied gameplay.

- Distance-based scoring (passive).
- Coin collection (active, 50 points per coin).
- Score multipliers from power-ups.

6 UI/UX Assumptions

6.1 Player Identification

Assumption: Attempt to auto-detect player name from system environment (`USER` or `USERNAME`), defaulting to "Player".

6.2 Name Entry Constraints

Assumption: Player names are limited to a maximum of 12 characters to ensure UI stability.

6.3 Visual Feedback

Assumption: Players need clear visual feedback for game states and power-ups (HUD, Pause overlay, Game Over sprite).

7 Resource Management Assumptions

7.1 Texture Loading

Assumption: All textures are PNG files stored in a single directory. The `loadTexturesFromDirectory()` function scans and loads all `.png` files.

7.2 Resource Lifetime

Assumption: All resources remain in memory for the entire game session to prevent loading stutters.

8 Performance Assumptions

8.1 Object Pooling

Assumption: No object pooling is used; objects are created and destroyed as needed to prioritize simpler implementation over optimization.

8.2 Frame Rate

Assumption: Game runs at a variable frame rate with delta time compensation (`sf::Time deltaTime`) to ensure consistent speed.

9 Error Handling Assumptions

9.1 File Operations

Assumption: File operations may fail but should not crash the game. Default values are used if files cannot be read.

9.2 Resource Loading

Assumption: Missing resources return empty or default textures to prevent crashes.

10 Audio Assumptions

- **Background Music**: A single track loops continuously during gameplay.
- **Sound Effects**: Triggered by specific events (Game Over, High Score, Coin collection).

11 Day-Night Cycle Assumptions

Assumption: The game features a dynamic day-night cycle controlled by `mDayNightTimer` that affects aesthetics but not gameplay.

12 Testing and Debugging Assumptions

12.1 Debug Mode

Assumption: A debug mode flag (`mIsDebugMode`) exists for development purposes (logging, hitbox visualization).

12.2 Manual Testing

Assumption: Testing is primarily manual, evidenced by the presence of a `TESTING_GUIDE.md`.

13 Code Organization Assumptions

13.1 Header Guards

Assumption: Use `#pragma once` for header guards for simplicity and compiler support.

13.2 Separation of Concerns

Assumption: Each class has a single responsibility (e.g., `ResourceManager` for assets, `TrackManager` for spawning).

14 Platform-Specific Assumptions

- **File Paths**: Use `std::filesystem::path` for cross-platform compatibility.
- **Build System**: Support both Visual Studio (Windows) and Makefile (Unix/Linux).

15 Academic Context Assumptions

Assumption: The project demonstrates understanding of OOP concepts (Polymorphism, Encapsulation, Design Patterns) while adhering to constraints like manual data structure implementation.

16 Future Extensibility

Assumption: The codebase is designed to easily accommodate new features (new obstacles, power-ups) via inheritance, though the game is not designed for multiplayer scalability.

17 Conclusion

These assumptions form the foundation of the project's architecture and implementation. They are based on explicit requirements, common game development practices, SFML conventions, and academic constraints.