LOW-LEVEL DESIGN REPORT

for

“Sanctified Retribution”

by

Metehan TÜTER

Elif Aysu KÜRŞAD

Murat BEDİZ

Erkan SANCAK

Date: 11.03.2024

Table of Contents

1.Introduction

The Low-Level Design report serves as a detailed blueprint for the development of Sanctified Retribution a 2D Roguelike single-player action game with pixel art aesthetics and an immersive storyline.

1.1 Purpose of the System

The primary objective of "Sanctified Retribution" is to deliver a captivating gaming experience that blends engaging gameplay mechanics, visually appealing pixel art aesthetics, and a compelling narrative. By leveraging tools such as Aseprite, Blender, and the Unity game engine, the system aims to create an immersive world where players can interact with diverse characters, adversaries, and dynamic terrains.

1.2 Design Goals

The design goals established for the system encompass various aspects crucial to its success:

Engaging Gameplay: Striving to create mechanics that are intuitive, responsive, and rewarding to players' actions.

Captivating Storyline: Crafting a narrative that captivates players' attention, featuring intriguing characters, plot twists, and impactful decisions.

Visual Appeal: Employing pixel art aesthetics to create visually stunning imagery that enhances the game's ambiance and immersion.

Exploration and Diverse Content: Providing players with opportunities for exploration and interaction within the game world.

Clear Communication and Player Guidance: Ensuring that players understand the game mechanics, objectives, and progression through intuitive tutorials, in-game hints, and visual/audio feedback.

Player Immersion: Enhancing immersion through sound effects, atmospheric music, and cohesive world-building elements that enrich the gaming experience

1.3 Object design trade-offs

In the object design phase of Sanctified Retribution, various trade-offs were considered to ensure the creation of a robust and efficient system. These trade-offs encompassed factors such as performance, flexibility, maintainability, and scalability. By carefully weighing these considerations, it is aimed to achieve a balanced design that meets the project's objectives while optimizing resource utilization.

Performance vs. Flexibility: One of the primary trade-offs involved balancing performance optimizations with the flexibility to accommodate future enhancements. For instance, optimizing rendering algorithms for smoother gameplay might require sacrificing some flexibility in rendering customization options. However, ensuring sufficient flexibility in the design allows for easier adaptation to new features or changes in requirements over time.

Maintainability vs. Optimization: Another crucial trade-off was between maintainability and optimization. While optimizing code for performance can lead to more efficient execution, it may also result in more complex and less maintainable code. Balance between optimizing critical performance bottlenecks and maintaining code readability and modularity to facilitate future updates and bug fixes.

Scalability vs. Resource Consumption: Scalability considerations were paramount to accommodate potential increases in the game's complexity and player base over time. Design decisions were made with scalability in mind, such as implementing modular subsystems that can be easily expanded or modified without causing significant performance overhead. However, ensuring scalability often comes at the cost of increased resource consumption, requiring careful resource management to maintain optimal performance across different hardware configurations.

User Experience vs. Technical Constraints: Balancing user experience requirements with technical constraints posed another set of trade-offs. For example, implementing complex AI behaviors or procedural level generation algorithms might enhance gameplay depth but could also introduce performance bottlenecks or development complexities. By prioritizing essential user experience elements while considering technical limitations, our team aimed to deliver a seamless and enjoyable gaming experience without compromising technical integrity.

Overall, navigating these object design trade-offs required careful consideration of project requirements, technical constraints, and development priorities. By making informed decisions and prioritizing trade-offs based on project goals, our team aimed to achieve a well-rounded and optimized design for the project.

1.4 Interface Documentation Guidelines

This preliminary interface description outlines the expected functionalities and interactions of Unity Engine as the game engine within the project. As development progresses, the interface may evolve to accommodate additional features and optimizations specific to the Unity environment.

Anticipated Functionality:

- **startGame():** Initializes the game session, setting up initial game assets, player characters, and environmental parameters.
- **updateGameState():** Updates the game state in real-time based on user input, AI behavior, and environmental interactions.

- **pauseGame():** Pauses the game session, suspending gameplay and allowing players to access menus or settings.
- **resumeGame():** Resumes the game session from the paused state, restoring gameplay and user interactions.
- **endGame():** Terminates the current game session, displaying end-game statistics and options to restart or exit.
- **loadLevel(levelID):** Loads the specified game level, transitioning players to new environments with unique challenges.
- **saveGame():** Saves the current game progress, including player achievements, inventory items, and completed objectives.
- **loadSavedGame():** Loads a previously saved game session, enabling players to continue from their last checkpoint.

Usage Anticipation:

```
GameInterface gameInterface = new GameInterface();
gameInterface.StartGame();
gameInterface.LoadLevel("Level1");
// Perform anticipated game actions...
gameInterface.SaveGame();
gameInterface.PauseGame();
// Handle anticipated user input...
gameInterface.ResumeGame();
gameInterface.EndGame();
```

1.5 Engineering Standards

In the development of Sanctified Retribution, adherence to established engineering standards ensures consistency, maintainability, and interoperability across the project. Key standards followed include those set forth by the Unified Modeling Language (UML) and the Institute of Electrical and Electronics Engineers (IEEE).

1.6 Definitions, Acronyms, and Abbreviations

This section provides definitions for key terms, acronyms, and abbreviations used throughout the Low-Level Design report for Sanctified Retribution.

Definitions:

1. **2D:** Refers to two-dimensional graphics and gameplay perspective, where objects are represented on a flat plane with width and height dimensions.
2. **Roguelike:** A genre of video games characterized by procedurally generated levels, permanent death, and a focus on exploration, strategy, and resource management.
3. **Pixel Art:** The intentional arrangement of pixels to create images, often characterized by a retro aesthetic and limited color palette.
4. **Sprites:** 2D graphical images or animations used in video games to represent characters, objects, or visual elements within the game world.

5. **NPCs (Non-Playable Characters):** Characters in the game controlled by the computer rather than the player. NPCs may provide quests, information, or trade services to enhance the player's interaction with the game world.

Acronyms and Abbreviations:

- **GUI:** Graphical User Interface
- **API:** Application Programming Interface
- **DBMS:** Database Management System
- **QA:** Quality Assurance
- **UI:** User Interface
- **UML:** Unified Modeling Language
- **IEEE:** Institute of Electrical and Electronics Engineers

2. Packages

2.1 Gameplay

2.1.1 Player

PlayerController: Manages player input, movement, and interactions with the game world.

PlayerAttributes: Defines attributes such as health, stamina, and inventory for the player character.

PlayerActions: Handles player actions such as attacking, jumping, and interacting with objects.

2.1.2 Enemies

EnemyController: Controls the behavior and movement patterns of enemy characters.

EnemyAttributes: Defines attributes such as health, attack power, and AI parameters for different types of enemies.

EnemySpawner: Spawns and manages the placement of enemy units within game levels.

2.2 Environment

2.2.1 Levels

LevelManager: Manages the loading, progression, and generation of game levels.

LevelAttributes: Defines attributes such as terrain layout, environmental hazards, and interactive elements within levels.

LevelEvents: Handles scripted events and triggers within game levels, such as cutscenes and environmental changes.

2.2.2 Objects

ObjectManager: Controls the spawning, placement, and behavior of interactive objects within game levels.

ObjectAttributes: Defines attributes such as size, weight, and functionality for different types of objects (e.g., platforms, power-ups).

ObjectInteractions: Manages interactions between the player character and environmental objects, such as collision detection and physics simulations.

2.3 User Interface (UI)

2.3.1 HUD

HUDManager: Displays relevant information to the player during gameplay, including health, inventory, and game objectives.

HUDElements: Defines individual UI elements such as health bars, weapon icons, and mini-maps.

HUDAnimations: Implements animations and transitions for UI elements to enhance visual feedback and player immersion.

2.3.2 Menus

MenuManager: Controls the navigation and functionality of in-game menus, including the main menu, pause menu, and options menu.

MenuLayouts: Designs the layout and visual presentation of menu screens, including buttons, text fields, and background images.

MenuInteractions: Handles player input and interactions within menu screens, such as button clicks and slider adjustments.

3. Class Interfaces

The interface definitions presented below are conceptual and provided as a preliminary framework for the Sanctified Retribution project. As the project is still in the early stages of development, these interfaces are subject to further refinement and modification as the development progresses. Any adjustments required to address evolving project requirements will be made during the development process.

3.1 Gameplay

3.1.1 Player

PlayerController Interface

- `void Move(Vector2 direction)`: Simulates player movement based on the provided direction.
- `void Attack()`: Simulates player attacking action.
- `void Interact(GameObject target)`: Simulates player interaction with objects in the game world.

PlayerAttributes Interface

- `int Health { get; }`: Property to get the player character's health.

3.1.2 Enemies

EnemyController Interface

- `void MoveToPlayer(Vector2 playerPosition)`: Simulates enemy movement towards the player character.
- `void AttackPlayer()`: Simulates enemy attacking action towards the player character.
- `void TakeDamage(int damageAmount)`: Simulates enemy taking damage.

EnemyAttributes Interface

- `int Health { get; }`: Property to get the enemy character's health.
- `int AttackPower { get; }`: Property to get the enemy character's attack power.

3.2 Environment

3.2.1 Levels

LevelManager Interface

- `void LoadLevel(string levelName)`: Simulates loading the specified game level.
- `void GenerateLevel()`: Simulates generating the layout and content for a new game level.

3.2.2 Objects

ObjectManager Interface

- `void SpawnObject(GameObject objectPrefab, Vector3 spawnPosition)`: Simulates spawning an interactive object at the specified position.
- `void DestroyObject(GameObject objectInstance)`: Simulates destroying the specified interactive object.
- `float Size { get; }`: Property to get the size of the interactive object.

3.3 User Interface (UI)

3.3.1 HUD

HUDManager Interface

- `void UpdateHealth(int currentHealth, int maxHealth):` Simulates updating the health display on the HUD.

3.3.2 Menus

MenuManager Interface

- `void OpenMenu(MenuType menuType):` Simulates opening the specified menu screen.
- `void CloseMenu(MenuType menuType):` Simulates closing the specified menu screen.

4.Glossary

1. 2D:
Definition: Refers to two-dimensional graphics and gameplay perspective, where objects are represented on a flat plane with width and height dimensions.

2. Roguelike:
Definition: A genre of video games characterized by procedurally generated levels, permanent death, and a focus on exploration, strategy, and resource management.

3. Pixel Art:
Definition: The intentional arrangement of pixels to create images, often characterized by a retro aesthetic and limited color palette.

4. Sprites:
Definition: 2D graphical images or animations used in video games to represent characters, objects, or visual elements within the game world.

5. NPCs (Non-Playable Characters):
Definition: Characters in the game controlled by the computer rather than the player. NPCs may provide quests, information, or trade services to enhance the player's interaction with the game world.

6. Aseprite:
Definition: A software tool used for creating pixel art and animations, commonly used in game development projects.

7. Blender:
Definition: An open-source 3D modeling and animation software used for creating visual assets and environments in game development.

8. Unity:
Definition: A popular cross-platform game engine used for developing 2D and 3D games, providing tools for game development, including rendering, physics, audio, and scripting.

9. Performance Optimization:
Definition: The process of improving the efficiency and speed of a system or software application to enhance its performance, often involving techniques such as code optimization, resource management, and algorithmic improvements.

10. Maintainability:
Definition: The ease with which a software system can be modified, updated, or extended over time while preserving its functionality and integrity, typically achieved through clear code structure, documentation, and modular design.

11. Scalability:
Definition: The ability of a system to handle increasing loads, demands, or growth without compromising performance, stability, or user experience, often achieved through distributed architectures, load balancing, and efficient resource management.

12. Technical Constraints:
Definition: Limitations or restrictions imposed by technological factors, such as hardware limitations, software dependencies, or compatibility issues, which may influence the design and development of a system.

13. Cutscenes:
Definition: Pre-rendered or scripted sequences in a video game that advance the storyline, provide exposition, or showcase key events, often featuring non-interactive cinematic presentations.

Acronyms and Abbreviations:

- **GUI:** Graphical User Interface
- **API:** Application Programming Interface
- **DBMS:** Database Management System
- **QA:** Quality Assurance
- **UI:** User Interface
- **UML:** Unified Modeling Language
- **IEEE:** Institute of Electrical and Electronics Engineers

5.References

Unity Documentation:

- Unity User Manual 2022.3 (LTS). (n.d.). Retrieved from https://docs.unity3d.com/Manual/index.html
- Unity Scripting Reference. (n.d.). Retrieved from https://docs.unity3d.com/ScriptReference/index.html

Çandıroğlu, A., Erken, A., Mandıracıoğlu, B., Azak, H. İ., & Özdal, M. M. (2019). Musync Low-Level Design Report. Supervisor: M. Mustafa Özdal. Jury Members: Özcan Öztürk, Cevdet Aykanat. Bilkent University, Department of Computer Engineering.

[Link to document: Project Low-Level Design Report]