Software Requirements and Design Document

For

Group <6>

Version 1.0

Authors:
Aiden Lassiter
Benjamin Reich
Kevin O'Malley
Reese Bouleware
Nate Longberry

- **Overview (5 points)**

We will develop a 2D puzzle platformer on Unity via C#. The theme of our platformer is Time-Travel. Over the course of this semester, we will develop the assets, code, levels, and art of this game. By the end of the project, it will be fully playable from start to finish, with clear end conditions and satisfactory gameplay.

Give a general overview of the system in 1-2 paragraphs (similar to the one in the project proposal).

- **Functional Requirements (10 points)**
  1. Character movement: The system shall allow the player to control the character's movement using standard directional inputs (WASD + Space). The character must be able to move left, right, and jump -- High Priority
  2. Puzzle Interaction: The system shall present time-based puzzles that require the player to manipulate objects' time states to progress. These puzzles shall be solvable only by using the time-travel mechanics in combination with character movement and object interaction. -- High Priority
  3. Transitions Between Eras: Whether in a level or advancing to the next, the system shall smoothly transition the environment when shifting between different eras. Objects, enemies, and terrain shall change dynamically to reflect the era's state. -- Medium Priority
  4. Combat Mechanics: The system shall include combat mechanics where the player can attack, defend, or avoid enemies. The combat mechanics shall be simple, with the ability to defeat enemies using basic attacks or time manipulation -- Medium Priority
  5. Health and Status Measurement: The system shall include a health and/or power-up system for the player character. Health shall decrease when the character is damaged by enemies or environmental hazards, and the player shall be able to restore health or power ups through specific in-game items. -- High Priority
  6. Save-and-load functionality: The palyer should be able to save their progress and load it back in -- Medium Priority
  7. Level-Progression System: The system shall track the player's progression through levels, unlocking new levels (new eras) as the player completes each one. Each level shall introduce new puzzles, enemies, and environments. -- High Priority

*List the **functional requirements** in sentences identified by numbers and for each requirement state if it is of high, medium, or low priority. Each functional requirement is something that the system shall do. Include all the details required such that there can be no misinterpretations of the requirements when read. Be very specific about what the system needs to do (not how, just what). You may provide a brief design rationale for any requirement which you feel requires explanation for how and/or why the requirement was derived.*

- **Non-functional Requirements (10 points)**
  1. Usability: The system shall have an intuitive user interface (UI) and clear tutorials to ensure that users of varying experience levels can easily learn the mechanics of time manipulation and game navigation. The user experience (UX) should be designed for minimal confusion, with clear visual and audio feedback for time-based actions. -- High Priority
  2. Load Times: The system shall ensure that level loading times do not exceed 5 seconds on modern hardware. Era-shifting transitions shall occur within 1 second, ensuring smooth gameplay -- High Priority
  3. Modularity and Maintainability: The system's codebase shall be modular, allowing individual components (e.g., time manipulation mechanics, level design) to be easily modified or extended without affecting the rest of the game. The code should be well-documented and follow industry-standard programming practices

to allow for future updates and bug fixes -- High Priority

4. Version Control and Updates: The system shall include a version control mechanism to ensure smooth updates and bug fixes. Players should be able to update the game without losing progress, and developers should be able to deploy patches without overwriting key system configurations -- Medium Priorityw

*List the **non-functional requirements** of the system (any requirement referring to a property of the system, such as security, safety, software quality, performance, reliability, etc.) You may provide a brief rationale for any requirement which you feel requires explanation as to how and/or why the requirement was derived.*

- **Use Case Diagram (10 points)**

```
Player
|
├── Move Character
|
├── Time Manipulation
|   ├──> Shift Era
|   └──> Solve Puzzle
|
├── Combat Enemy
|
├── Collect Item
|   └──> Solve Puzzle
|
├── Play Tutorial
|   ├──> Move Character
|   ├──> Time Manipulation
|   └──> Combat Enemy
|
└── Solve Puzzle
    ├──> Time Manipulation
    └──> Collect Item
```

*This section presents the **use case diagram** and the **textual descriptions** of the use cases for the system under development. The use case diagram should contain all the use cases and relationships between them needed to describe the functionality to be developed. If you discover new use cases between two increments, update the diagram for your future increments.*
***Textual descriptions of use cases**: For the first increment, the textual descriptions for the use cases are not required. However, the textual descriptions for all use cases discovered for your system are required for the second and third iterations.*

- **Class Diagram and/or Sequence Diagrams (15 points)**

1. GameManager
    - Attributes: currentLevel, gameState, checkpoints
    - Methods:

    - startGame()
    - loadLevel(level)
    - saveProgress()
- Relationships:
    - Aggregates multiple Level objects
    - Manages Player class

2. Player
- Attributes: health, position, inventory, timeEnergy
- Methods:
    - move(direction)
    - useTimeManipulation(object)
    - combat(enemy)
    - collectItem(item)
- Relationships:
    - Has a composition relationship with Inventory (one inventory per player)
    - Can interact with Level, Enemy, and Puzzle

3. Level
- Attributes: era, layout, environmentObjects
- Methods:
    - loadEra(era)
    - updateEnvironment()
    - resetLevel()
- Relationships:
    - Aggregates Puzzle objects (1..*)
    - Aggregates Enemy objects (0..*)
    - Contains Player (1)

4. Puzzle
- Attributes: puzzleState, requiredItems, solutionSteps
- Methods:
    - solve()
    - checkSolution()
    - resetPuzzle()
- Relationships:
    - Aggregates Item objects (0..*)
    - Interacts with Player class
    - Extends TimeManipulation class

5. Enemy
- Attributes: health, type, attackPattern
- Methods:
  - attack(player)
  - takeDamage(amount)
  - interactWithTimeManipulation()
- Relationships:
  - Can interact with Player and Level
  - Can be affected by TimeManipulation

6. TimeManipulation
- Attributes: timeEffectType, cooldown
- Methods:
  - ageObject(object)
  - revertObject(object)
- Relationships:
  - Can be used on Level objects, Enemy, and Puzzle
  - Accessed by Player

7. Inventory
- Attributes: items[], maxCapacity
- Methods:
  - addItem(item)
  - removeItem(item)
  - useItem(item)
- Relationships:
  - Aggregates Item objects (0..maxCapacity)

8. Item
- Attributes: name, type, effect
- Methods:
  - applyEffect()
- Relationships:
  - Can be used in Puzzle
  - Part of Inventory

*This section presents a high-level overview of the anticipated system architecture using a **class diagram** and/or **sequence diagrams**.*

*If the main **paradigm** used in your project is **Object Oriented** (i.e., you have classes or something that acts similar to classes in your system), then draw the **Class Diagram of the entire system and Sequence Diagrams for the three (3) most important use cases in your system.***

*If the main **paradigm** in your system is **not** **Object Oriented** (i.e., you **do not** have classes or anything similar to classes in your system) then only draw **Sequence Diagrams**, **but for all the use cases of your system.** In this case, we will use a modified version of Sequence Diagrams, where instead of objects, the lifelines will represent the functions in the system involved in the action sequence.*

***Class Diagrams** show the **fundamental objects/classes** that must be modeled with the system to satisfy its requirements and **the relationships** between them. Each class rectangle on the diagram **must also include the attributes and the methods of the class** (they can be refined between increments). All the **relationships between classes and their multiplicity** must be shown on the class diagram.*

*A **Sequence Diagram** simply depicts **interaction between objects** (or **functions -** in our case - for non-OOP systems) in a sequential order, i.e. the order in which these interactions take place. Sequence diagrams describe how and in what order the objects in a system function.*

• Operating Environment (5 points)

The time-travel side-scroller game software will primarily operate on a PC platform. The game will be developed using Unity 2020 LTS or later, utilizing C# for scripting and leveraging Unity's multi-platform support, extensive asset libraries, and physics engine to handle complex time manipulation mechanics and seamless visual transitions. Developers will use Visual Studio (2019 or later) for its Unity-specific support, along with Git or GitHub for version control, and Unity Hub to manage projects and maintain consistency across the team. For audio, integration with FMOD or Wwise is recommended to support dynamic, adaptive soundscapes corresponding to different time periods.
Describe the environment in which the software will operate, including the hardware platform, operating system and versions, and any other software components or applications with which it must peacefully coexist.

• Assumptions and Dependencies (5 points)

1. Unity Game Engine Compatibility: The game assumes continued compatibility with Unity (2020 LTS or later) for C# scripting and cross-platform development. Any updates or deprecations in Unity's features, libraries, or platform support could require reworking code or redesigning parts of the game.
List any assumed factors (as opposed to known facts) that could affect the requirements stated in this document. These could include third-party or commercial components that you plan to use, issues around the development or operating environment, or constraints. The project could be affected if these assumptions are incorrect, are not shared, or change. Also identify any dependencies the project has on external factors, such as software components that you intend to reuse from another project.