Sanctum of the Chalice

Project Design Document
Cs 307 Team 28

Presented by:
Shubham Jain
Phoebus Yang
Taehoon Kim
Alec Hartline
John Pesce

Table of Contents

§ 1 Purpose

Of the many gaming genres that have emerged over the past 3 decades, the dungeon crawler genre has remained relevant and often acquires a dedicated and niche player base. This is particularly true for roguelite and roguelike dungeon crawlers, which are characterized by stat checks and high challenge levels. *Crypt of the Necrodancer* is a prime example of this phenomenon, exhibiting both turn-based combat and permanent death. However, it strays for the genre due to a lack of stat checks. In contrast, many other popular games of the roguelite genre tend to rely too heavily on stat based combat, item drops, and luck, resulting in a lack of strategy for clearing the game.

Sanctum of the Chalice aims to follow the classic roguelite/roguelike experience while also adding a degree of strategy to gameplay. To accomplish this, *Sanctum of the Chalice* will give the player both adequate time to plan out actions and play around enemies, thus minimizing the impact of stat checks and randomization on success. Furthermore, just as several other popular dungeon crawler games possess a specific gimmick that sets them apart from others of the genre, *Sanctum of the Chalice* will implement a time reversion mechanic that will allow the player to rollback to previous positions. This mechanic plays in well with the concept of strategy based gameplay since it can be utilized for solving particular puzzles or dodging around enemy attacks. By injecting the need for planned thought into the classic dungeon crawler experience, *Sanctum of the Chalice* seeks to be an entertaining and fresh take on this genre.

§ 1.1 Functional Requirements

In Game Display

- As a player, I would like to see a heads-up-display that gives me all relevant information in a clear format
- As a player, I would like to see the amount of time before the next tick executes in a clear manner

Randomized Level Generation

- As a developer, I would like to have a procedural level generation system that can accept multiple parameters and generate unique, challenging levels
- As a player, I would like to be able to experience procedurally generated levels
- As a player, I would like to be able to encounter unique puzzles every level and be rewarded for solving them

Gameplay and Diversification

- As a player, I would like to have a tutorial that explains and utilizes core game mechanics
- As a player, I would like to be able to save my game state and load it at a later time
- As a player, I would like to be able to use the time revert system to aid in combating enemies and overcoming traps
- As a player, I would like to be able to combat threats and be rewarded for succeeding in overcoming them
- As a player, I would like to be able to invest experience gained into stats to scale in power as the game gets harder

- As a player, I would like to choose from multiple character classes and specialize in certain attributes to improve my character through the game
- As a player, I would like to be able to collect and equip items or equipment and use them for combat and other purposes
- As a player, I would like to be able to learn and use unique abilities that diversify combat and allow for interesting interactions
- As a developer, I would like to provide players a challenge fighting enemies through a robust AI system
- As a developer, I would like to be able to assign ability rotations to enemies to increase combat difficulty
- As a player, I would like to be able to encounter and fight bosses that present opportunities to progress in the game
- As a player, I would like to be able to gather experience by completing tasks like killing enemies and solving puzzles
- As a player, I would like to be able to view my overall score and achievements upon the completion of levels

Customization

- As a player, I would like to be able to navigate a main menu so that I may start a new game or change game settings
- As a player, I would like to be able to tweak game settings like difficulty and sound volumes
- As a player, I would like to be able to personalize keyboard controls in the game settings
- As a modder, I would like to be able to modify or add items to the game with ease
- As a modder, I would like to be able to specify sprites or sprite-sheets for custom game objects

Other Game Elements

- As a player, I would like to be able to meet friendly non-player characters, and have unique interactions with them
- As a player, I would like to experience random events that provide unique experiences each time
- As a player, I would like to keep track of the bits of lore I have uncovered at any point so that I may put together pieces and understand the backstory (if time allows)
- As a player, I would like to see particle effects that add to the ambience of the game and can be used for special indicators (if time allows)
- As a player, I would like to be able to take in-game screenshots of high quality (if time allows)

§ 1.2 Nonfunctional Requirements

Game System

- As a developer, I would like for a clear and hierarchical class system that can represent attributes and relationships of all game objects in a logical manner to exist
- As a developer, I would like for a JSON attribute system to exist, so that I may organize and sort game data
- As a developer, I would like to be able to edit the attributes of existing items and add additional items with ease
- As a developer, I would like to have a robust serialization format to facilitate saving and loading game states
- As a developer, I would like to have a physics system that allows for player movement and collision detection, as well as projectile motion
- As a developer, I would like to have a system that can keep track of a player's movements and be able to revert the player to a set limit of previous movements

Rendering

- As a developer, I would like to use the Java Swing and AWT libraries to create a rendering engine optimized for this game
- As a developer, I would like for the rendering engine to process and render sprites or sprite-sheets as PNGs to the screen
- As a developer, I would like to secure at least 30 frames per second

Game Flow

- As a player, I would like to see input actions reflected smoothly in the game view
- As a player, I would like to see visual indication of buffered input as well as proper indication of when the action will be performed
- As a developer, I would like to only update game objects and entities that influence the player on a given tick to minimize CPU load

Sound System

- As a developer, I would like to have a sound engine that can play both music in the background as well as sounds based upon actions being performed in game
- As a player, I would like to hear distinct and clear audio cues for certain actions performed by myself or by the environment around me
- As a player, I would like the music and sound effects to increase immersion and improve the ambience of the game
- As a developer, I would like to minimize resource consumption by using Ogg Vorbis files for music and sound clips
- As a developer, I would like to be able to read in sound files in .ogg format and associate them to either music or sound cues
- As a developer, I would like to use the Java OpenAL library to play .ogg files

§ 2. Product Overview

This project will be a top down, pixel art, roguelite dungeon-crawler, with procedurally generated levels and a unique time reversion based mechanic that can be exploited to overcome challenging enemies and puzzles. The game will run on our own custom engine written in Java to manage all game processes. This project will try to use the Java standard library whenever possible in order to maximize compatibility between machines. For this reason, the rendering and window management will be handled by Swing and the Abstract Window Toolkit (AWT).

The core functions of the game will be divided amongst the following major components

1. Rendering Engine
    a. The rendering engine will be the front end of this game. It will handle both the presentation of images and animation to the player, but also receiving input and relaying it to the game engine for processing.
2. Game Engine
    a. The game engine will manage the game's update loop and interpreting user input. This includes updating the game state through both the fast tick and slow tick systems, which will be discussed in greater depth later on.
3. Object Engine
    a. The object engine is responsible for reading the JSON files and creating templates for any game objects the game engine will need. It will respond to requests from the game engine and level generator with copies of these templates.
4. Level Generator
    a. The level generator is the component that is called on the initialization of a level. Its primary role is to procedurally generate levels based on parameters passed to it from the game engine and templates from the Object engine.
5. Sound Engine
    a. The sound engine is responsible for reading in sound files and then playing them when requested by the game engine. It needs to be able to manipulate portions of files, loop music, and play multiple sounds concurrently as needed.

§ 3 Design Issues

§ 3.1 Functional Issues

Issue 3.1.1) Under what time system should the game progress?

Option 1) Turn-based

Option 2) Real time

Option 3) Slow tick: Dungeon crawlers can employ various ways to move the game world forward, each with a distinct feel and play style. Sanctum of the Chalice aims to encourage strategic play while still pressuring the player to think and react quickly. A turn-based system allows the player ample time to plan out actions and use strategy. However, the playstyle encouraged by this system moves too slowly for there to be any sense of urgency. Real time satisfies the need for fast thinking and reaction. However, real time interactions can be difficult to predict and plan around. A slow tick system, one that updates game's state at noticeable intervals, will require the player to think quickly while making a system

with clear order of actions and interaction outcomes. Moreover, A slow tick system works well with the core time mechanic by providing a clear increment for reversions.

Issue 3.1.2) How should the player control their character?

Option 1) A preset of controls that are shown to the player during a tutorial

Option 2) A settings option to allow the player to change control mappings: This approach allows us the players to set the controls to their liking. With the first approach, players used to a different set of controls than the ones we set for them might be pushed away, along with any players that might have defective keys that are mapped to some essential in game control.

Issue 3.1.3) What form of storytelling should be used?

Option 1) Through cutscenes and exposition techniques

Option 2) Through a latent, minimalistic form: The primary reason for this choice was reluctance in moving away from the traditional roguelike genre. Roguelike games generally lack any form of fleshed out lore or story. We want to provide the roguelike experience, but still give the player some form of immersion into the dungeon that they are fighting through. Most bits of lore and story will be hidden across levels and will require players to exert some amount of effort to figure out what is going on. This method gives players another goal to work towards while further immersing them in the environment.

Issue 3.1.3) How can we make the slow tick system feel responsive?

Option 1) Make a window where players can activate the next tick early

Option 2) Show a visual input buffer: This approach tells players what their next input is, indicating that they are indeed not being ignored. It also can be expanded to visually enhance other features. One example is if the player enqueues an ability, they will get immediate feedback of which one they pressed, and if they have mistakenly pressed the wrong button. If the player decides they would rather not do any action, there will be a keybind implemented that allows them to cancel the queued action.

§ 3.2 Nonfunctional Issues

Issue 3.2.1) How will we render our game view to the screen?

Option 1) Libraries supporting OpenGL (like libgdx and LWJGL)
Option 2) Libraries supporting DirectX for Windoge
Option 3) Using Swing and AWT to create our own rendering engine: While using OpenGL libraries like LWJGL is a fair choice (Minecraft), they can increases the complexity of our engine. Since our game is going to be very simple in terms of the UI, we don’t need the amount of power these libraries provide us. Making our own engine in Swing and AWT allows us to run our game on any platform that can run java (unlike a DirectX library, which would only run on a windows platform). Developing our own

rendering engine also gives us more control, allowing us to tailor the system specifically to our fairly minimalistic needs.

Issue 3.2.2) How will we allow players to mod our game?

Option 1) Create a class hierarchy to define various game object attributes, and open source the game

Option 2) Write our class system, and script all game attributes and entities using Lua.

Option 3) Create a class attribute system and populate the game with objects read in from JSON files: This approach makes it very easy for us as developers to add content to the game, while letting players to modify and add their own custom items and entities. Using Option 1 as an approach would give modders more control but would require anyone developing a feature to hardcode it in, thus requiring recompilation between changes. The second approach, while very flexible, makes introducing changes a bit more complex, and gates the entry to modding the game. With the third approach, flexibility is still prevalent in the system, and it becomes much easier for a person from a non-technical background to create mods.

Issue 3.2.3) What file format should be used for sound files?

Option 1) WAV

Option 2) MP3

Option 3) Ogg Vorbis: Wav files, while having the highest quality sound, take up large amounts of memory and are difficult to manipulate. Mp3 files are of a much more manageable size while still maintaining quality but cannot be played in a smooth loop. Ogg files were chosen since they have a similar level of quality and compression to mp3 files, but can be played in smooth loops, which will be important for implementing background music. Moreover, ogg files are open source and tools for manipulating them are readily available.

Issue 3.2.4) How should in game events be handled?

Option 1) Direct function calls to handler methods

Option 2) Observer Pattern: Unlike direct function calls, the observer pattern allows multiple systems to be notified about certain changes and events with very little load. This allows us to section off code written for say, the object engine from anything in the rendering engine, while still allowing some form of minimal communication between them.

Issue 3.2.5) How should art assets be managed for rendering?

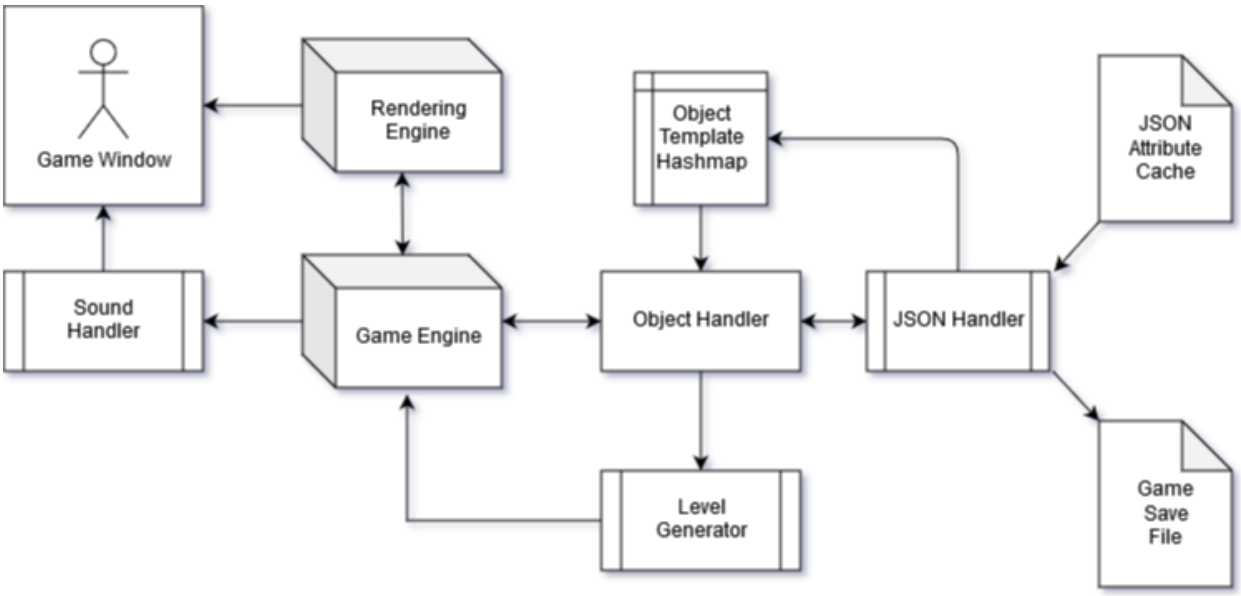
Option 1) Use multiple image assets to describe various game objects and their animation

Option 2) Use single PNG images that store game object sprites and their animation frames: While the second approach does require extra work when making and bringing art assets together, it saves a lot of space on the player's machine and allows our system to read different types of objects with different animation frames states according to a set of standardized rules. Achieving this with loose image files for each frame of an animation would be much harder since we would have to consider the naming of each image asset as well as potential differences in file-types.

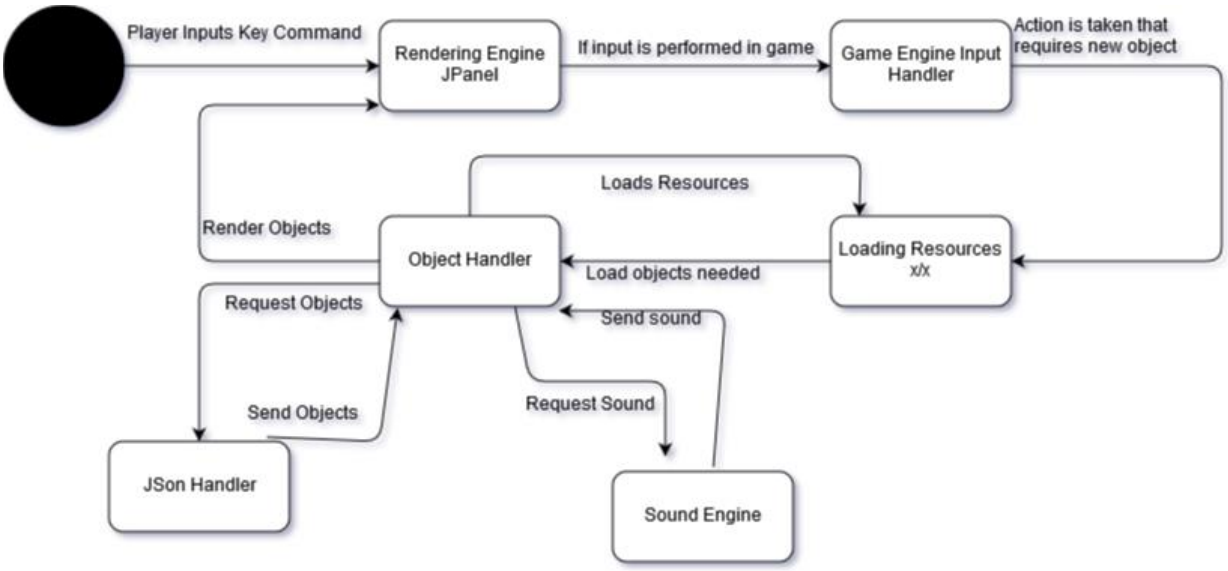
§ 4 Design Interactions

- Game engine and rendering engine:
    - Synchronization between the game engine and the rendering engine is essential because a slow on one of them would otherwise affect the other. A lack of synchronization can also cause race condition bugs.
    - The game engine and rendering engine will thus always communicate through a middle-man class that is thread safe and can ensure proper synchronization.
    - The game engine maintains a list of all objects that can currently be updated (i.e. within a certain radius of the player), whereas the rendering engines has access to a list of everything on the level that can currently be rendered and renders that.
    - Input will use a KeyboardListener, which detects state changes, and when such a change is detected it will relay this information to the middle-man. This will deliver the user input to the game engine for processing and updating.
    - The rendering engine can overlay new views when the game engine receives input for scenarios such as pausing the game.
- Game engine and sound engine:
    - The game engine will request the sound engine to play an audio file. This request will be added to a synchronized queue for processing. This will happen concurrently, as playing sound on the game engine thread would halt the system.
- Game engine and object engine:
    - The object engine will provide definitions for all the objects that the game engine will use.
    - The object engine reads in attributes and item/entity definitions from a set of JSON files, and then generates templates objects based upon those attributes. Anytime an instance of any of those objects need to be created as a game object in the game engine, the template is copied from the object engine.
    - The object engine also defines the hierarchy of all game objects.
- Game engine and level generator
    - The game engine calls the generate function with appropriate parameters, and the level generator returns a level with the provided constraints.
    - Parameters may include, but are not limited to, level number, number of enemies, and maximum room count.
- Game engine and JSON handler
    - The game engine will send information to the JSON handler when saving the game state.
- Object engine and level generator:
    - The object engine provides the level generator with all the assets and objects that the level generator can use.

The following diagram demonstrates the above interactions in terms of data flow.



The following diagram demonstrates the interaction flow upon detecting input from the user.



§ 5 Design Details

§ 5.1 Rendering Engine

1. Window.java
    a. This class will serve as the basic container for the game window. It will initialize the swing event thread and prepare a JFrame with the specifications provided upon the class's invocation.
    b. This is the head of the rendering engine and is the only allowed direct connection between the game engine and rendering engine.
    c. It will also maintain debugging variables such as tracking frames per second and directing the rendering engine what to do based of changes of game execution mode.
2. GameView.java
    a. The game view inherits from javax.swing.JPanel. It contains code to iterate through a list of currently renderable objects and call their draw methods. For every frame, the game view will calculate which sections of the window are "Dirty", referring to sections that have experienced change, and repaint only those sections.
3. SpriteLoader.java
    a. The sprite loader will be given a folder in the source directory containing PNG files. It will then load said PNGs and divide each into groups of 32 pixels by 32 pixels. These individual sprites will be placed into an array containing every image the game will use.
4. SpriteAnimator.java
    a. This class will be invoked as the game view's painting method iterates through drawn objects. It manages individual game object animations with their own animation speeds and sprite set. This is designed to free game object classes from each having their own draw method, a design that would be messy due to redundant animation code.
5. RenderGameProxy.java
    a. Atomically manages lists of game objects for the purposes of avoiding concurrency errors upon changing the lists. This unit will also bridge the gap of bringing user input from the Swing thread to the game engine thread.

§ 5.2 Game Engine

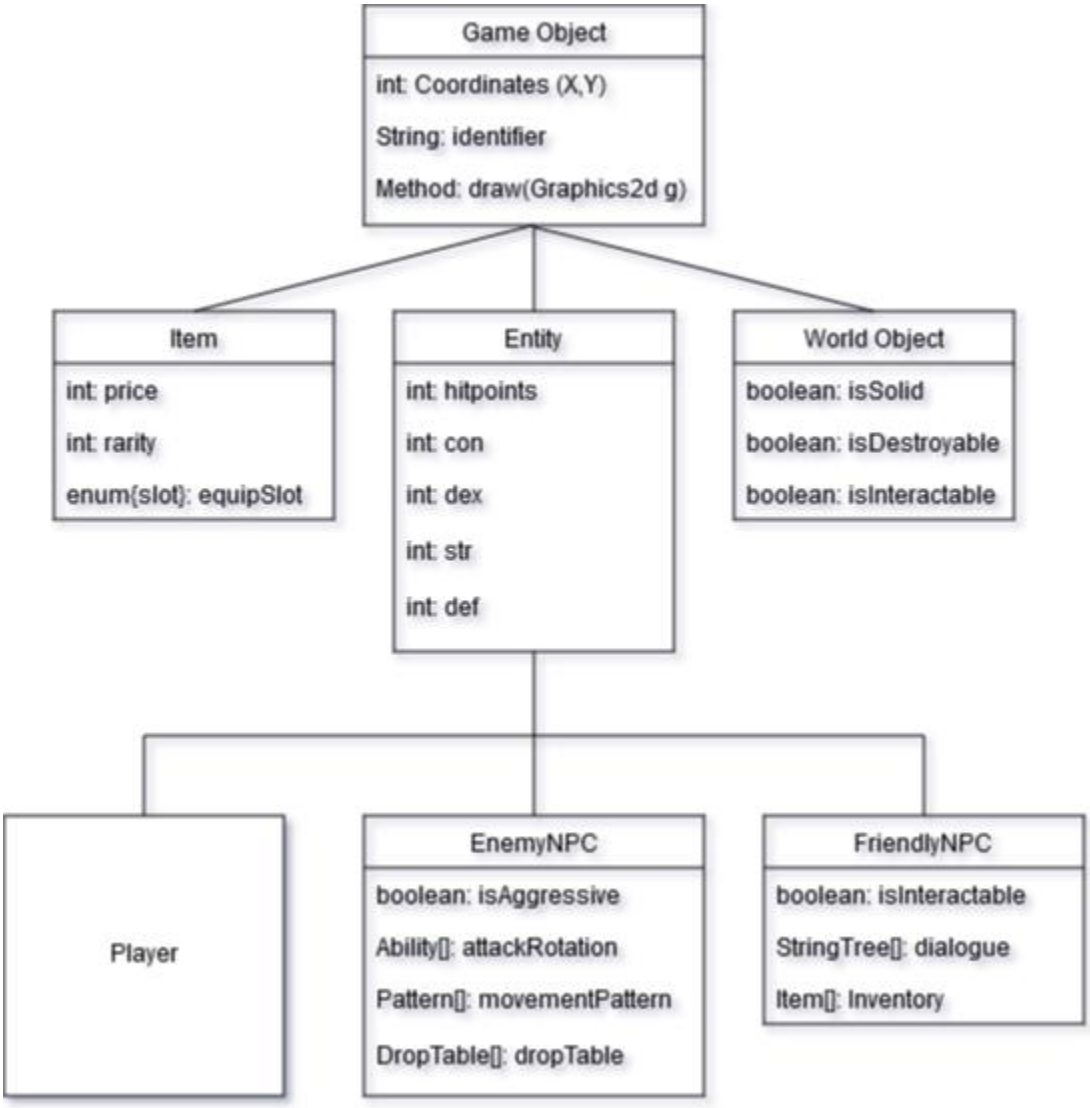
1. GameStart.java
    a. This class will handle the initialization of a given level. As such, it will send a request along with the appropriate parameters to the level generator to create the map and entity placements for the level. The class will also get templates from the object engine to initialize all game objects that will be needed for the level.
2. GameLoop.java
    a. This class will be responsible for managing the game update loop. It will contain both a slow update method and a fast update method
        i. The slow update function handles the processing and execution of all activities that occur during the slow tick such as player actions, enemy actions, and other map updates. It will be subdivided into smaller functions in order to modularize the many actions that the game loop must perform each call. This organization also makes the sequence in which the game processes entity actions clear.
            1. PlayerMove(): This function will take the most recently buffered input from the player and attempt to move the character accordingly. If the move is an illegal one, such as attempting to move into a wall, then nothing will happen. However, if the action is interpreted as an attack (moving into the same space as an enemy), then it will return data regarding the attempted action that indicates it must be handled later on. Activation of time reversion is also handled during this step.
            2. EntityMove(): This method updates the state of all relevant non-player entities including enemies, npcs, and traps. As such, it is during this phase that non-player ai is called and actions taken.
            3. PlayerOther(): If the player movement function classifies the attempted action as one that needs to be delayed, then its return value is passed into this method to be processed accordingly.
            4. The slow tick ends by registering the player's current position and adding it to the front of the history linked list. If the linked list is at a certain capacity, the position at the back of the list will be removed.
        ii. The fast update method handles processes that must occur outside of the slow tick system to keep the game running smoothly. This includes interpolating positions to represent actions performed in the last slow tick, sending requests to the rendering engine, sending requests to the sound engine, and buffering input for the next slow tick. This method will also have to handle multiple game states depending on a flag that can be set by player actions or input.
            1. Regular game loop: This is state represents normal gameplay and will call the slow tick after set intervals
            2. Pause/Inventory: During this state, all gameplay related activity will not get called or incremented. This includes the timer and interpolation towards the next slow tick. Instead, the fast update method will handle input and actions to do with managing inventory.
            3. Time reversion: During this state, update actions not related to the player, including calls to EntityMove(), will not be performed. As a

result, the game world outside of the player becomes "frozen." Instead, the only action that the player will be able to buffer is to either exit time reversion or continue. For each slow tick in time reversion, the position at the front of the history linked list is removed and the player is moved to that location. Each slow tick will also decrement a counter of how many reversions the player can make. If either the history linked list is empty or the counter is zero, then the player will be forced to exit time reversion. After exiting time reversion, the regular game loop will resume, and interactions will be processed based on the player's location upon exiting time reversion.

3. GameEnd.java
    a. This class handles activities for ending a level. There are two main ways in which this class will be called
        i. Level Complete: If the end of the level is reached, then data concerning items acquired, level, and other changes that carry over subsequent levels will be sent to the JSON handler for saving.
        ii. Level Failure: If the player dies or some other failure condition is met, then then items and levels acquired during the level will be discarded and the player will be given a chance to restart the level.

§ 5.3 Object Engine

1. JsonLoader.java
    a. Retrieves JSON files from the game's source directory and prepares them for parsing. This class will also validate the file inputs and attempt corrections if they are found to be invalid. For example, if there is a missing file, it will attempt to generate a basic version as a form of recovery.
2. JsonParser.java
    a. Accepts the contents of a JSON file from the JSON Loader. The contents will be parsed using the org.JSON library. Following the parsing process, this class will then identify which fields are present and generate a game object characteristic of the fields shown in the following diagram.
    b. This unit will also perform additional validation by throwing error upon finding malformed JSON constructions.
3. ObjectHandler.java
    a. The Object Handler is responsible for the management of all tasks revolving around game objects within the game world. This class will maintain a HashTable of object templates generated by the JSON Parser. In any circumstance where the game engine requests an object to be created, it must query the Object Handler. The Object Handler will then copy one of the templates based off the identifier passed to it, instantiate it with the requested information, and add it to the appropriate object list.
    b. When the Game Engine requests a new level, the Object Handler will retrieve the relevant level information from the level HashTable and pass that to the level generator unit.



§ 5.4 Level Generator

1. LevelGenerator.java
    a. The level generator contains all the logic for procedurally generating levels. More specifically, this class will contain methods that take is parameters like the level number (influences difficulty), a theme value (indicates the style and aesthetic of the art to be used), and size (dictates how long the generation will run before stopping; can be SMALL, MED, or LARGE). This class will also contain functions that populate levels with interesting artifacts such as loot chests, special lore bit holders, and loot items as well as another method to populate the level with threats.
2. RandomEngine.java
    a. This class will contain methods that use java.util.Random's various methods for returning randomized numbers. The idea is that this class will have different

methods that return a random number within certain bounds, a random coordinate, or the result of a weighted random roll. It is meant to be used by LevelGenerator for getting the various random values.

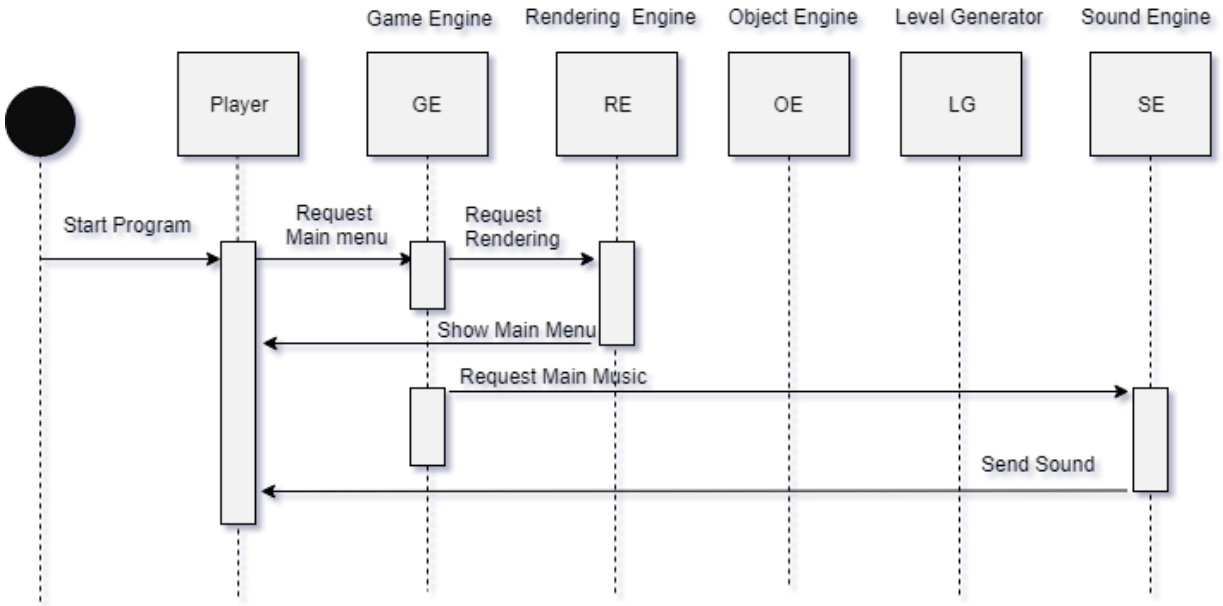
3. TemplateLoader.java
    a. This class communicates with the JSON parser to obtain any room templates stored in rooms.json file. The Level Generator communicates with this class to obtain a list of templates that can go in the level it is trying to generate and adds a selected template to that level.
4. Overview of the generation algorithm
    a. The level generation algorithm begins by initializing all blocks with solid tiles. It will then choose a location and empty out surrounding tiles to form the spawn room. From here, the level generator performs the following steps:
        i. Pick a random spot on any wall of the current room
        ii. Roll a weighted dice that adheres to conditions for rooms to spawn (conditions like minimum distance from spawn required, difficulty of the level, etc.) and selects a room from the list of templates OR just a corridor
            1. If a corridor is selected, then a template room must be generated at either the end of the corridor, or on one of the corridor walls
            2. If nothing can be generated, go to step 1
        iii. Check if the level has enough room to generate the selected structure at the selected spot
            1. If not, go to step 2
        iv. Generate the selected structure (i.e. room template or corridor)
        v. Repeat from 1
        vi. Once the condition for being done with generation is met,
            1. Populate the level with enemies
            2. Populate the level with loot and interesting features.

§ 5.5 Sound Engine

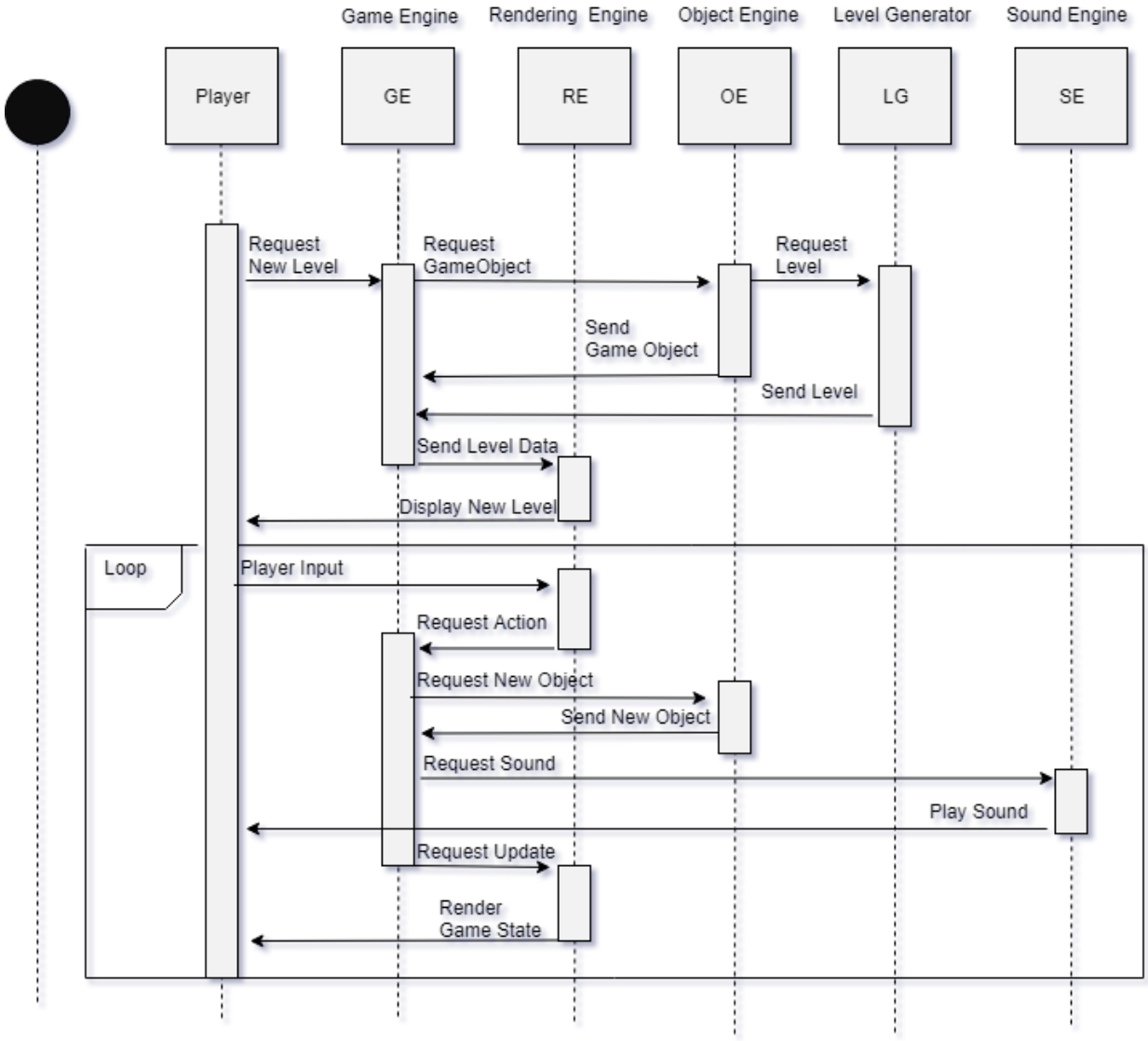
1. SoundEngine.java
    a. The sound engine will import Java OpenAL(Open Audio Library) to play selected sound files. This class will receive requests from the Game Engine for events such as opening chests, hitting enemies, boss encounters, and background music and play the appropriate sounds. The sound engine will utilize functions from the Java OpenAL library, which allows for easy reading and manipulation of Ogg files. The sound source files that we will use are referred to using string identifiers. Furthermore, our game will be two dimensional, which means that we don't need to worry about the position (AL_POSITION variable) of the sound in three-dimensional space. We are going to use the buffer queuing facility, which supports streaming sounds with buffer queuing from OpenAL.

§ 6 Sequence Diagrams

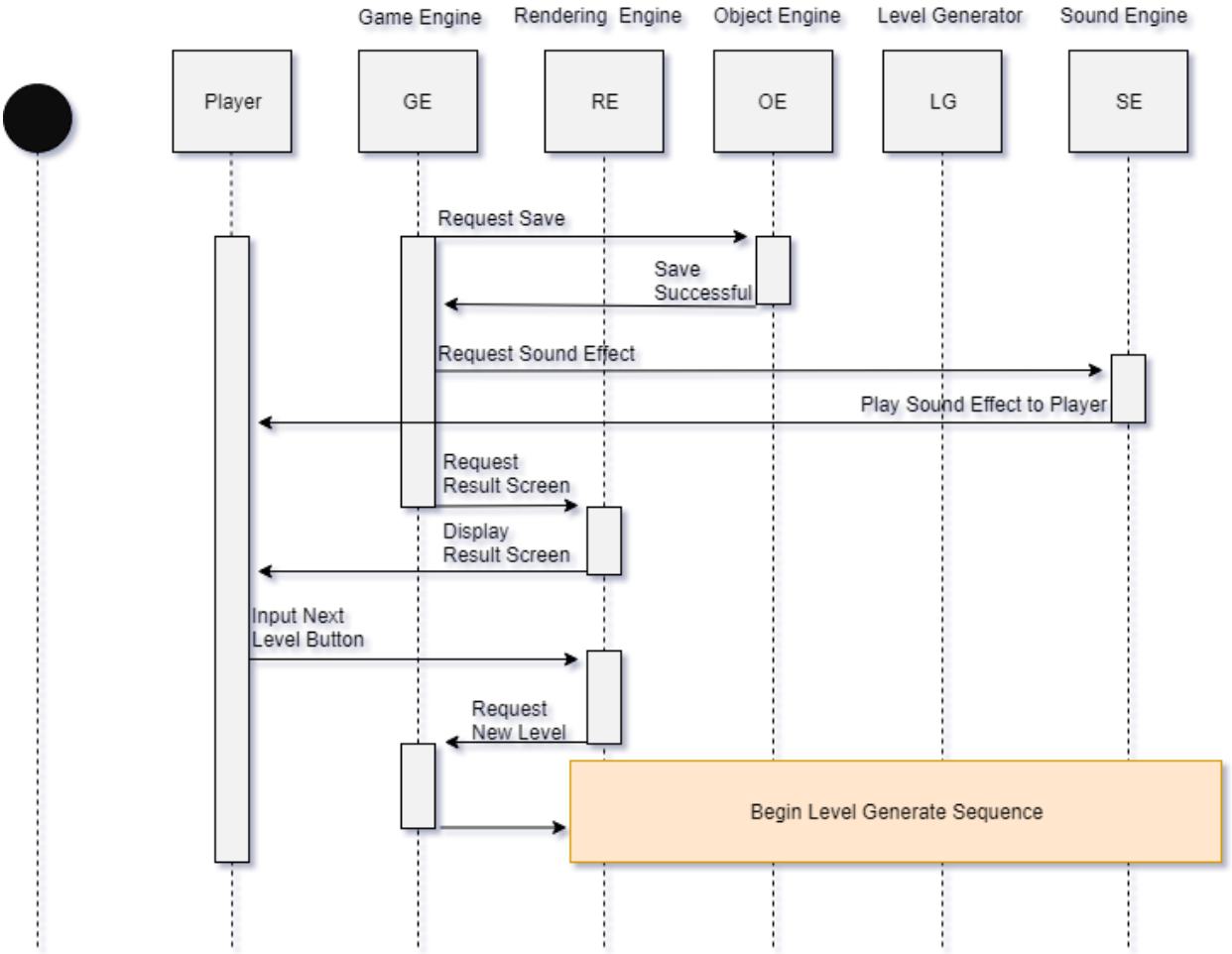
§ 6.1 Sequence of Events When Opening the Game



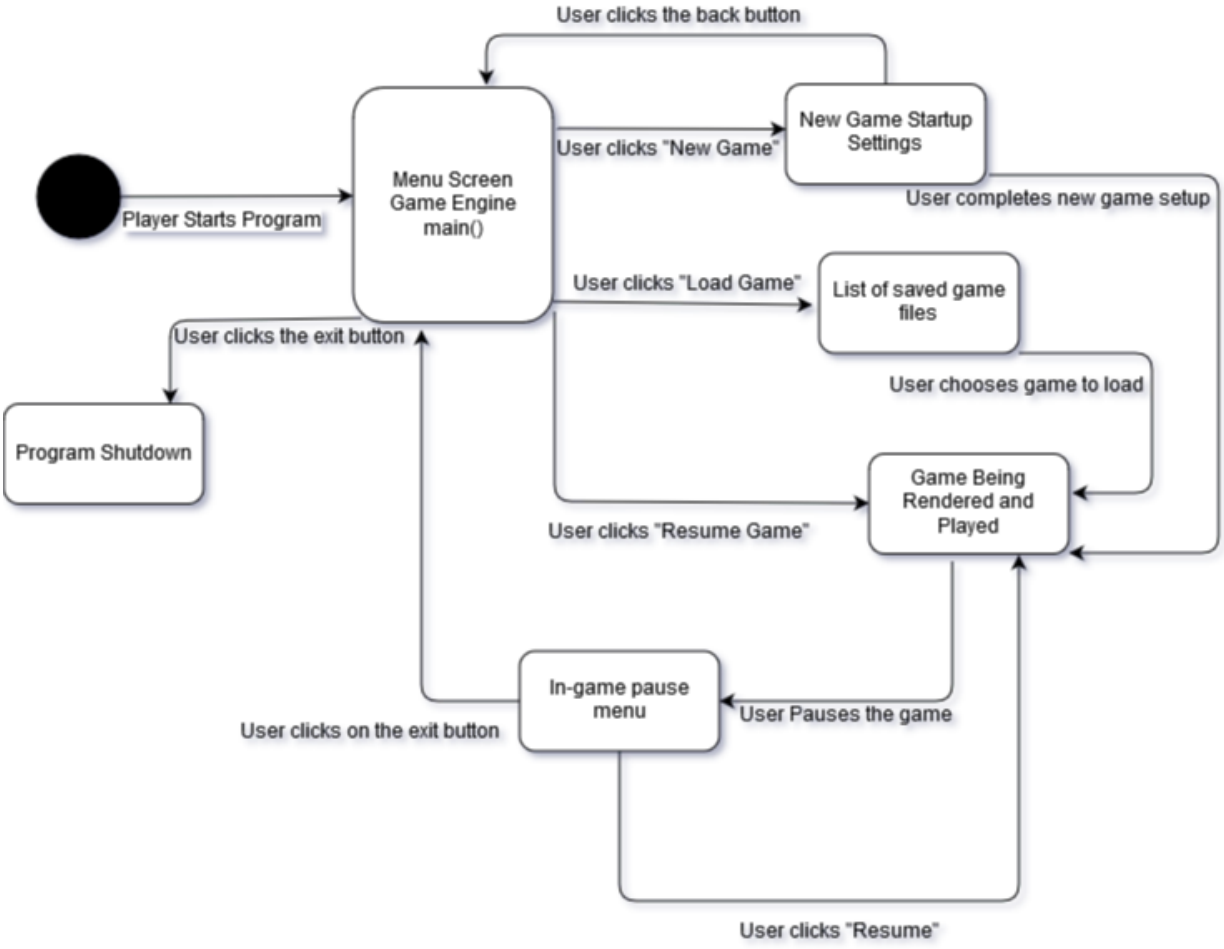
§ 6.2 Sequence of Events When Entering a Level



§ 6.3 Sequence of Events When Completing a Level

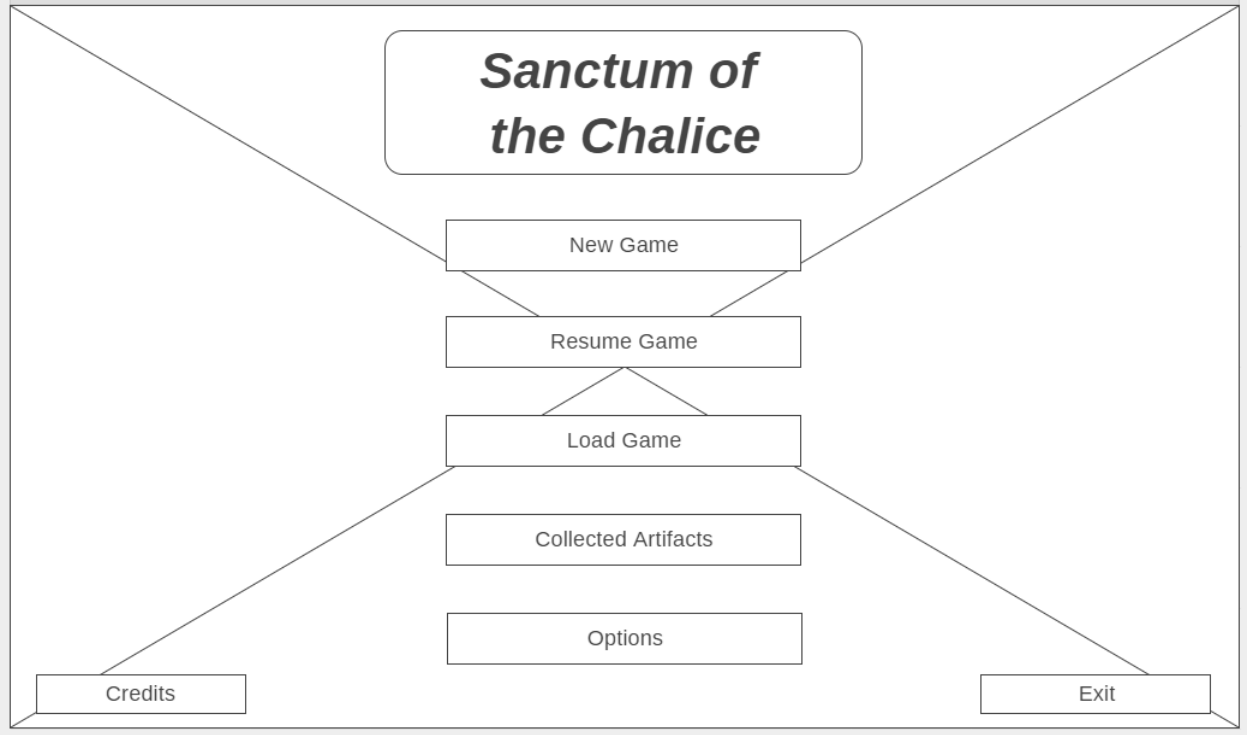


§ 6.4 Map for Game Screen Navigation

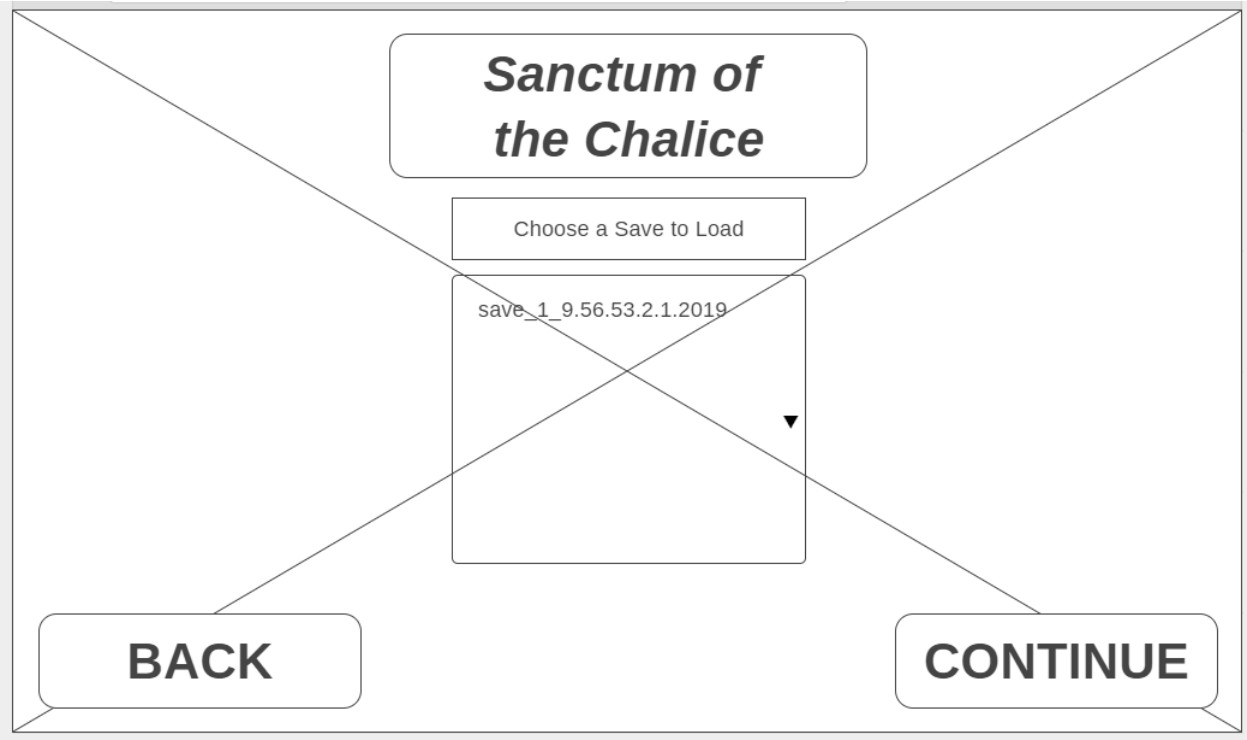


§ 7 User Interface Mockup

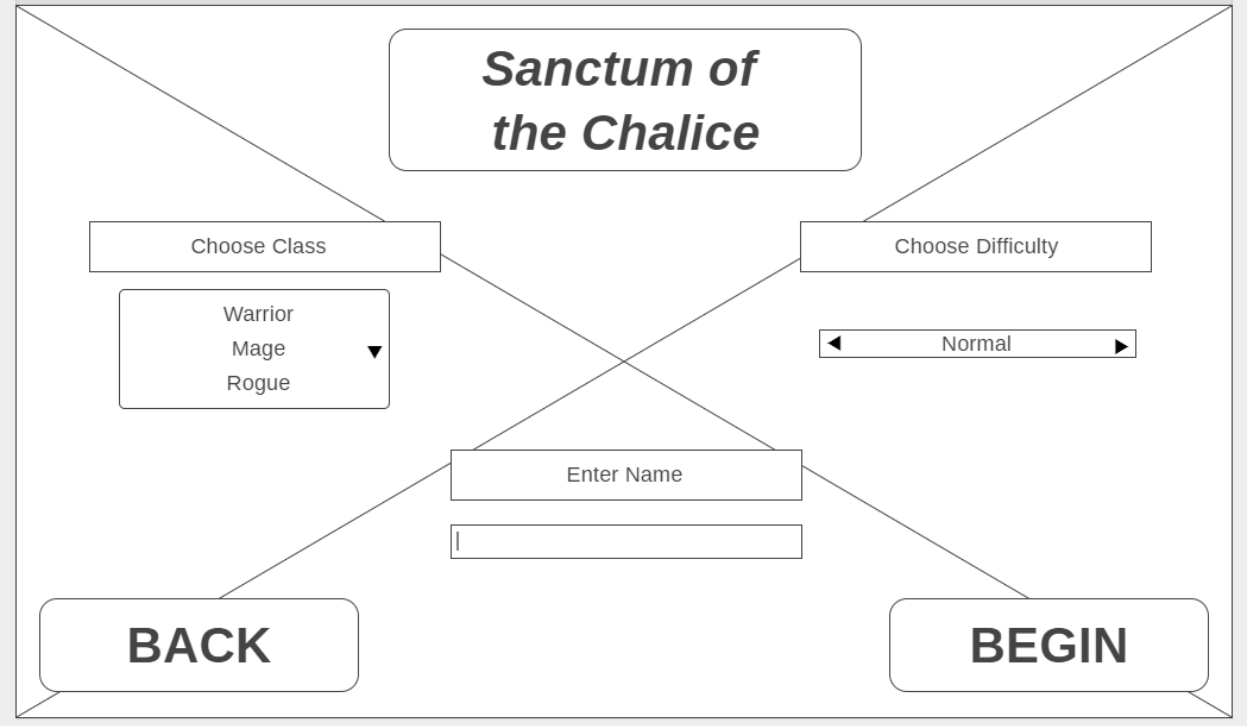
§ 7.1 Main Menu



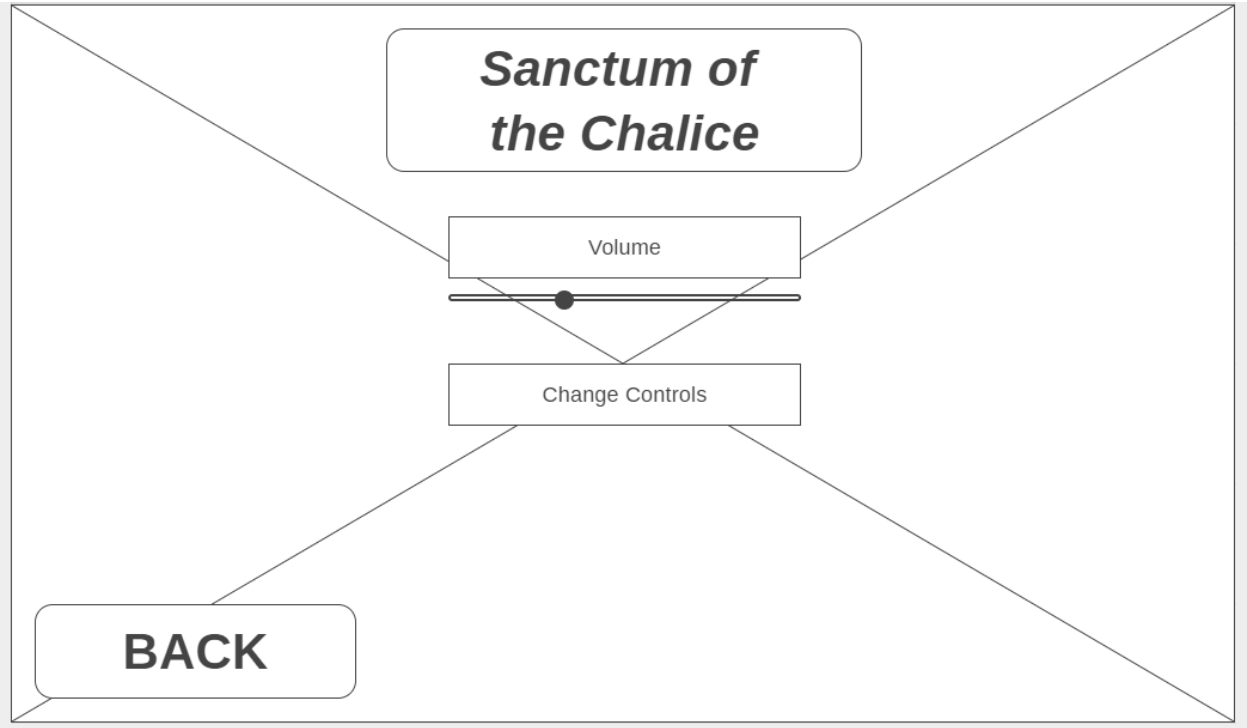
§ 7.2 Load Game Menu



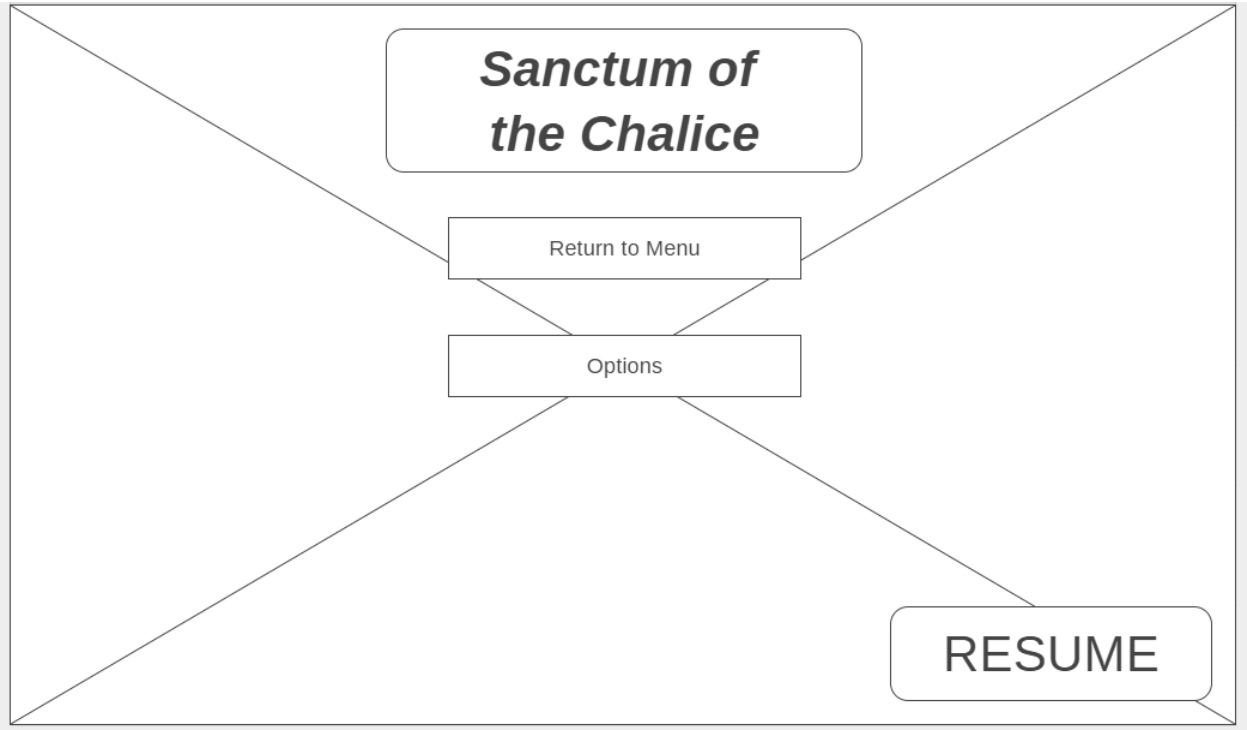
§ 7.3 New Game Menu



§ 7.4 Options Menu



§ 7.5 Pause Menu



§ 7.6 Heads-Up Display