Design Document & Roadmap - Project Ambedo

Prepared by: Team Nodus Tollens

Thomas Dykstra - 400029662

Corey Szeto - 400025728

Zhiwen Yang - 400023048

Tim Zhang - 400006216

Instructor: Dr. Jacques Carette

Course: Software Engineering & Game Design (4GP6)

Date: 09-04-2020

Document Overview

This document outlines the major design decisions of our games. They have been split up with respect to various components of our game. The components that we have decided to split them up into are found below:

- Player Game Interactions
    - Main View
    - Player Controls and Movement
    - Game Settings/Options
    - Combat
    - Objectives
    - UI and Menus
    - Game Time
- Game Systems
    - Game Data Save
    - Enemies/Bosses
    - Exp Bar (Evilness)
    - Final Cutscene

Player Game Interactions

Player game interactions refer to all aspects of the game that the player is in control of.

Main View

The view in the game will be a side-scrolling side view angle that will follow the player up, down, left and right wherever the player goes. As Ambedo is a 2D platformer, this felt the most ideal for the camera model and shows just enough to the player. There is an “invisible” bound around the player and if the player comes into contact with it, it will push the bound in the direction the player is going. This allows the camera to follow the player but it will not always move if the player doesn’t move a lot. The player does not have direct control of the camera and can only move it if they move the player enough in any direction.

Player Controls and Movement

The player takes control of a single avatar, the ghost character. Nearly all interaction with the game world occurs through this avatar. In controlling this avatar, players will exclusively use the keyboard and mouse. There is no support for any third party controllers or alternate input methods.

Horizontal movement of the avatar is achieved by pressing and holding the ‘A’ or ‘S’ key on the keyboard (for moving left and right movement, respectively). Horizontal movement can also be achieved by the left and right arrow keys (again, for left and right movement). When the key is pressed and held, the player will accelerate in the specified direction until it reaches a set speed. The player will always accelerate in a single horizontal direction as if from rest, so long as the respective key is held by the player. This means that abrupt direction changes (e.g. holding down the right arrow key, then simultaneously releasing it and pressing the left arrow key) will not preserve the player’s original momentum, and instead will reset it to 0 before accelerating in the new direction. This was done to sharpen horizontal movement precision and to give players more direct control over the avatar’s speed and direction at any given time.

The player avatar’s vertical movement is achieved partially by the Unity physics engine, and partially by player input. Gravity affects the player avatar at all times, applying an always-present downwards acceleration. This gravity is achieved by the Unity physics engine acting on the Rigidbody2D component added on to the player avatar. Players are allowed to make the avatar jump by pressing the spacebar. This applies a momentary force directly upwards on the player avatar, allowing it to overtake the acceleration of gravity. Once airborne, the player avatar will accelerate downwards until it interacts with a platform collider, enemy collider, death zone, or other such object that would interrupt free fall. The player has control on the jump height by holding the space bar for shorter/longer periods of time. For instance, a quick tap will make the player perform a small hop, which has little height and takes little time before the player hits the ground again (assuming the player is stationary). Holding the spacebar will cause the player to jump very high, so the player spends significantly longer in the air. Players have full horizontal movement control while airborne, regardless of if they had previously jumped or simply fell off of a ledge.

Ladder objects are an alternative method of changing the player avatar's vertical position. While the player avatar is in direct contact with a ladder, they are not affected by the Unity physics engine in the vertical direction. This disallows the player to jump while touching a ladder. Pressing the 'W' or 'S' keys will cause vertical movement up or down, respectively. Alternatively, the up and down arrow keys achieve the same effect. The player still has full horizontal control on the avatar while touching a ladder.

Occasionally, control of the player avatar will be disallowed by the game system. This situation occurs either when the player is in a cutscene (signalled using various methods) or when the main/pause menu is active (see **In-Game UI and Menus**). Player control is re-allowed upon exiting these states.

Further interaction is achieved by allowing the in-game avatar to somewhat control the movements of certain in-game items. This can be achieved either through the physics engine (e.g. pushing a rock or pushing a crate) or by specific hardcoded mechanics (e.g. the "peaceful" method of defeating the second boss). Avatar movement is slowed for the former, as dictated by the Unity physics engine.

Combat is another form of player interaction through the avatar. For more information on this, see **Combat**.

Game Settings/Options

The player is presented with the option to make changes from either the main menu or the pause menu via the “Settings” option. However, although all of the buttons for changing game settings do not have any effect on the game itself. The justification for their existence centres around an unfinished implementation of these features, for which no documentation exists that we know of, and we are unable to get any information from the party originally responsible for these features. This lack of effect is not conveyed to the player because, again, we simply do not understand the implementation and have no reliable means of doing so. As a result, while changing these features will not crash the game, they will also not do anything to the game in any way.

Combat

Combat in Ambedo includes players fighting against enemies and bosses they find throughout levels. We want players to feel that each enemy is unique and threatening and that bosses not only reflect the environment, but prove to be a much harder task than other enemies. The player can attack by pressing the left mouse button which will begin the attacking animation. The player cannot attack again until the animation is finished which will make the player have to time their attacks. The hitbox spawns in front of where the player is facing when the button is pressed. If the hitbox touches an enemy or a boss, that enemy/boss will take damage according to the player’s current attack stat. Combat is not mandatory, especially not in Ambedo. The player can choose to fight or ignore/run from enemies depending on whichever playstyle they choose to pursue.

Objectives

The goal of the player is to survive at all costs. They are thrown into the world of Ambedo after the first cutscene and must find a way back to their family. Progress is clear in this game as they can only move to the right to move forward through the levels. In terms of whether or not the player should kill enemies/bosses, that decision is up to the player. The consequences of killing or not killing will be shown in the results in either an increase in strength and hp or a more demented character look respectively. At intermediary points in the game and the final cutscene, depending on how many enemies/bosses were killed, the outcomes of those events will be different.

In-Game UI and Menus

The In-Game UI in Ambedo includes two parts, health bar, and exp bar, which represents the evilness of the player. First, the health bar indicates the health status of the character. It is laid on the top-left corner. At the initial state, the character will have three slots of health. When the character gets hitted by the enemies or bosses, one slot of health will be deducted. The character will gain an invincible buff a few seconds, after another few seconds, the loss of health will be restored, which is represented by the health bar. This restoring design reduces the difficulty of the game, and increases the rate of fault tolerance for the player. Second, the exp bar indicates the evilness of the character. This is the invisible status in Ambedo. After the player kills a boss, the shape of the character will be changed, the player will notice the evilness of the character. And the player will gain an extra slot of health.

There are two main components of the in-game UI that (not including the menus, which are discussed further in this section). The first main component is a series of health icons, located in the top left corner of the screen. These icons indicate how much health the player currently has, as well as the maximum amount of health the player is allowed to have at a given time. Current health is denoted by filled in symbols, and the current amount of health lost is denoted by the empty symbols. The second main component is an experience bar located directly underneath the aforementioned health icons. This is a grey bar that fills red, from left to right, as you kill enemies in game. Once the bar is filled, the player avatar will undergo the transformation process. The bar will then be emptied back to grey, and any excess experience will carry over and be used to partially fill the bar again. These icons clearly demonstrate the amount of health they have at any given time, as well as a way of measuring how many enemies the player has killed throughout the game.

Initially, the player will have 2 health. Upon taking damage (denoted by a sound effect), they will lose one health and go into an invincibility state, (shown by the player avatar turning red for the duration of the invincibility state). After several seconds without taking damage after invincibility has ended, the player will start restoring health. This functionality reduces the overall difficulty of the game and increases the fault tolerance for the player, especially if/when their maximum health increases.

After a transformation process, the player's avatar will change to something significantly more demented-looking. The player will also gain an extra max health slot and do slightly more damage. The number of times the player has transformed is saved as the player's "level" (the fact that the game is saving this information is not explicitly told to the player), which changes the ending cutscene.

There are two menus present in the game: the main menu and the pause menu.

The main menu is present upon initially loading the game, or when the "Quit to Main Menu" button is pressed from the pause menu. The main menu has the following four options, listed in order from top to bottom: "New Game", "Continue", "Settings", and "Quit". The functionality of these options is described further in this section.

The pause menu is made available to the player by pressing the 'ESC' on the keyboard while in the game view. From here, there are the following four options, listed in order from top to bottom: "Continue", "Settings", "Quit to Main Menu", and "Quit".

In order to better explain these menu options, it is first imperative to understand the autosave functionality. See the **Game Data** section under the **Game Systems** header for more information on this feature.

From the main menu, we have the following options.:

- “New Game” - Allows the player to start a new game. The save file will be reset to a default state (beginning of the first level, no enemies killed, etc.) and will load/play the opening cutscene
- “Continue” - Allows the player to continue from where they previously left off. The save file is read and the game state is reloaded using this information.
- “Settings” - Described further below
- “Quit” - Stops the game from running. The save file is stored persistently.

From the pause menu, we have the following options::

- “Continue” - unpauses the game. The same effect can be achieved by pressing the ‘ESC’ key on the keyboard.
- “Settings” - described further below
- “Quit to Main Menu” - Leaves the game view and brings the player back to the main menu, with all options from that menu available
- “Quit” - Stops the game from running. The save file is stored persistently.

Game Time

Each of the opening and closing cutscenes are about a minute and a half long. There are also two short cutscenes in level 1, which take about 20 seconds each. This gives a total of around 3 minutes and 40 seconds of cutscenes.

In-game, each level should take anywhere from 5-10 minutes on first playthrough, depending on skill level. This should decrease to about 3-6 minutes for subsequent playthroughs due to increased experience.

In total, the first playthrough of the game by some player should take about 15-30 minutes (not including cutscene time), and subsequent playthroughs should take about 9-18 minutes.

Game Systems

This section is written with regards to everything the player cannot directly interact with or influence.

Game Data

As the player progresses through the game, they will reach checkpoints at pre-designated fixed locations. Reaching one of these checkpoints will save the current game state to a save file locally stored on the player's device. This game state includes, but is not limited to, the player's position, the current level, enemies killed, etc. This save is done automatically upon reaching a checkpoint, and there is no indication to the player that they have reached a checkpoint. There is no way of manually saving the game. Every enemy in each level is kept track of, and if an enemy is killed, that enemy will not respawn upon reloading at previous checkpoints. This helps convey the message that the player's choices matter, and also prevents players from "farming" experience by abusing the autosave system.

Enemies/Bosses

Enemies through the level generally have two major ways to damage the player. One is the collision contact with the player. The other is the hitbox contact with the player from the enemy attacking action. Bosses generally have three major ways to damage the player. The first is the collision contact with the player. The second is the spell fired by the boss like a fireball. The third will be the magic to change the boss room layout like creating the trap or falling stone to cause damage to the player. The boss will dash to player's position periodically. The boss room consists a lots of platforms and at the end of the boss room it is the exit of the level.

Boss 1 has no rigidbody attached to it. Since it is a ghost and it intend to fly around and pass through the physical object. Boss 2 and Boss 3 use the rigidbody. Because of this difference, the AI movement scripts between them are implemented differently.

Evilness/Exp Bar

The player can gain experience (EXP) every time they kill an enemy or boss. The enemies and bosses are all assigned their own arbitrary number of “orbs” that get spawned upon death. These orbs will track the player and move towards them and upon coming in contact with the player, the player will “absorb” the orb and gain experience. Each level takes 100 experience and each orb is worth 10 experience, thereby needing 10 orbs total to level up.

Upon leveling up, the player will gain an additional heart (which is equivalent to 100 Health Points) and this can be easily seen in the HUD at the top left corner. In Ambedo, the higher level you are, the more “evil” the character becomes and will impact which cutscene gets played at the end of the game. Any extra exp that was gained will carry over to the next level. Bosses are worth more and spawn more orbs upon death while the intermediary enemies are worth less and spawn less orbs.

There are a total of 6 transformations (including the starting character model) that the player can undergo; which means a range of levels 1 to 6.

Final Cutscene

There are 2 final cutscenes featured at the end of the game and the current level of the player (see Evilness/EXP bar section above) will decide which cutscene gets played. If the player’s current level is 3 or higher when the player reaches the end of the game, the Bad cutscene will be played. If the player’s current level is 2 or lower when the player reaches the end of the game, the good cutscene will be played. The two endings are drastically different than one another in order to show the severity of the resulting player’s actions. This heavily contributes to our intended player experience as it shows clearly to the player the consequences of their decisions throughout the entirety of the game.

Roadmap

Our collection of scripts are grouped based on their purpose and we decided to isolate scripts based on their commonalities and whether or not their behaviour is needed elsewhere. If the scripts are indented, they belong and or are related to the parent script.

Scripts

Scripts for Boss

Boss1AI: Boss 1 movement and attack.

- **ParabolaController**: controls the boss parabola movement.
    - **MathParabola**: helps to calculate parabola movement.

Boss2AI: Boss 2 movement and attack.

- **PlayerOnHead**: Attach to the head collider to detect if the player is on the boss head.
- **DestroyPlat**: Attach to the pop up platform.

Boss3AI: Boss 3 movement and attack

BossHealth: Attach to all the bosses to provide health attributes.

BossProjectile: Control the projectectile speed, and animation.

Scripts for Boss Room

BossRoomZone: For Boss1 entry,will change the BGM and camera area.

BossRoomCameraPan: This script manages the camera pan cutscene in the first level. This is needed to clearly show the player their available options which ties heavily into the desired player experience.

Level2Exit: Attach on the level 2 exit door

Level3to4Transition: This script handles the short cutscene transition from the end of level 3 to level 4. The behaviour needed for this cutscene was more complicated than the transitions in the other level, therefore needing its own script.

Level3Lock: Use for the cross puzzle in the Boss 3 room

BossRoomLock: Use to open door for Boss 3 room

Level3PuzzleCode: Use to open escape door for Boss 3 room

Scripts for Puzzles

RockController: This script controls the behaviour for the rock puzzle in the first level. Because this was a very niche and specific behaviour, we decided to isolate the behaviour into this script.

LightController: Gives functionality to the light objects that exist in the first level. Also controls the flickering process the lights go through.

DestroyDoorLock: This script is used for the level 2 puzzle door.

Trigger_switch: This script controls the trigger in level 2.

Level3Lock: This script is used for the cross puzzle in the beginning of level 3.

Level3Puzzle2: This script is used to detect player hitbox in puzzle 2 of level 3.
Level3Puzzle2SecondPart:This script is used to trigger the EnemySummoner for puzzle 2 of level 3.

Level3PuzzleCode: This script is used to open the escape door for puzzle 2 of level 3.

Scripts for Enemies

EnemyAggroZone: This script determines whether or not the player is within the aggro zone of an enemy. If the player is within the aggro zone, the enemy's behaviour will change depending on the enemy.

EnemyHealth: This script manages all health related aspects of each enemy. This includes health, damage from player upon contact, damage to player upon contact, spawning of exp orbs, death, animations and saving to the save file.

EnemyController: Controls the movement and behaviour of regular enemies. A single generic script that can be attached to each enemy, and movement can be chosen from a drop-down menu of different behaviours. For consistency purposes, there are no "variations" of a given enemy; each enemy with the same sprite will have the same behaviour.

ProjectileSpawnerController: Manages the periodic spawning of projectile objects for enemies that shoot projectiles, including fire rate.

ProjectileMovementController: Manages the movement of the projectile objects themselves. Designates their speed and the amount of time they exist before despawning.

Scripts for Player

PlayerHealthController: This script manages all health related aspects of the player. This includes health, damage from enemy upon contact, damage to enemies/bosses upon contact, death, animations and saving to the save file, current level, exp, max hp, current hp, health HUD and the player transformations.

PlayerAttackScript: Controls the player's attack. This includes the animation of the sword swinging and spawning the hitbox.

PickUpByPlayer: This script is used during the sword cutscene, and designates the sword as a child object of the player

PlayerMovement: Manages all direct player avatar movement through the game world. This includes horizontal movement, jumping, falling, and climbing ladders.

Script for Other Features

OrbController: This script controls the movement of the orbs when they are spawned upon enemy death. This script needed to be isolated in order for the orbs to track the position of the player every frame and move towards the player. Upon coming into contact with the player, the orb game objects will get destroyed, the player will gain exp and the game will be saved.

CheckPointController: This script invokes the save function when the player comes into contact with the checkpoint. If the player dies and respawns, or loads the game back up, they will spawn at the position designated in the save file.

CameraMovement: This script ensures that the camera is centered around the player and follows the player around as the player moves throughout the levels. This script also allows us to manipulate it during cutscenes in order to show the player key aspects in the level.

DeathZoneController: This script controls the behaviour for the death zone that is present in all levels. If the player falls off the map and touches the out of bounds zone, this script will handle killing the player and forcing the player to respawn.

EnemySummoner: This script summons the enemies in level 3 and is the only level with enemy summoning behaviour, therefore needing its own script.

UIManager: UI control system.

ApplicationManager: This script manages the transition between scenes when using the main menu/pause menu.

CanvasScaler: This script manages the pause menu and the main menu animations/behaviour.

Save: This class has all the fields needed for the user's save file and also has a save function that can be used if any other scripts need it (i.e. player level up, enemy death etc.). The save function will take all required fields and will overwrite the existing file.

LoadScene: This script handles all loading scene behaviour throughout the game.

Scripts for Cutscenes

SwordCutscene: This script manages the cutscene where the player first gets the sword they will use throughout the entirety of the game. Because this only occurs once, we gave this cutscene its own script. After the player finishes the sword cutscene, the player's status is saved to the save file and will always have a sword from that point on.

EndOfLevelCutScene: This script manages the transition from one level to the next. It also sets the correct starting position for the next level so that the player spawns in the correct position. This script is applied to a game object in each level.

FinalCutsceneController: This script manages which cutscene gets played at the end of the game (see Final Cutscene). If the player's level is 3 or higher, the bad ending cutscene will be played. If the player's level is either 1 or 2, the good ending cutscene will be played. This controller determines this based on the current level of the player that is saved in the save file.

Design choices - System Hierarchy

Character uses different scripts to control the following camera, movement, and health controller and there are scripts for controlling the play of cutscenes.

The game has save files and functions of respawn, through the map, there are respawn points, death zone detection. Players could also save their game files on PC for next time. These are controlled by the same scripts in the structure. Any script that needs to save will invoke the save function that is found in the save class but will have their own logic in terms of which save fields they need to change/modify. This allows for us to specify different behaviours across different scripts without having too much duplicate code.

There are multiple types of puzzles in the game: First, a sudden event puzzle, such as the dropping rock, is controlled by an event script to handle this kind of event: rock dropping; second, a light puzzle, such as light in the level 1, is controlled by light controller script to damage the player properly; third, a hidden rock puzzle, is controlled by a specific controller to control the destruction action on the hidden wall; fourth, a platform puzzle, such as first puzzle in the level 2, is controlled only by the design of the front-end, this puzzle provide two different ways for the player to choose: one is jumping through the platforms, the other is walking through the bottom line but fighting with enemies; fifth, a lock puzzle, such as second puzzle in the level 2, the player could choose to open two switches in the puzzle or break the second lock due to evilness in this puzzle, these actions are controlled by the separate scripts to control the door lock event; sixth, a rock puzzle, such as the third level puzzle in the level 2, is controlled by front-end and a specific rock event script to control the choices in the door lock event; seventh, sequence puzzle, such as the first puzzle in level 3, is controlled by a specific script to get the input from the player to go through the puzzle or fight against the enemies.

Each kind of enemy has control scripts to manage their AI, but controlled by the same health script. This allows us to more easily design different kinds of enemies in the game.

Every boss uses the same boss health script to control health. Each boss is designed to follow the theme of choices. Each boss room provides a specific way to escape from the fight with the boss in order to give player’s the choice in how they want to approach the different fights. These events are all controlled by separate scripts in the structure. Each boss uses a different helper function to the Boss AI to support the specific design mechanics for each boss.