System Design Document

Project: Plight of Pip: Wrath of the Riper

Team: Rascally Saints
Purpose of this Document

This document is intended to help the development team design the project Plight of Pip: Wrath of the Riper. It lays out the decomposition of the system into subsystems, the architecture of the system, and the core design goals the team would like to achieve.
Table of Contents

Purpose of this Document 2

Table of Contents 3

1 Introduction 4
   1.1 Purpose of the System 4
   1.2 Design Goals 4
       Performance 4
       Dependability 5
       Cost 5
       Maintenance 6
       End User Criteria 6
   1.3 Definitions, Acronyms, and Abbreviations 7
   1.4 References 7

2 Proposed Software Architecture 8
   2.1 Overview 8
   2.2 Subsystem Decomposition 8
       Player Subsystem 9
       NPC 9
       Interactables 9
       Inventory 10
       Game State 10
       Menu Systems 10
       System Flow Controls 11
   2.3 Subsystem Services 11
   2.4 Hardware Mapping 11
   2.5 Software Mapping 12
   2.6 Persistent Data Management 13
   2.7 Access Control and Security 14
   2.8 Boundary Conditions 15

Rascally Saints 3
1 Introduction

1.1 Purpose of the System

The purpose of this system is simply put: Generate a unique environment for players to explore and enjoy. This is a purpose that is hard to define further, but research into the medium of video games have proven insightful in getting a better idea of how to achieve this goal. This experience will be generated by challenging the user in a variety of ways, both in terms of narrative storytelling and physical challenges posed within the system. Mastery of skills developed to achieve narrative goals set out within the system will provide the user with a sense of accomplishment, feelings of joy, and entertainment.

1.2 Design Goals

The design goals for this project will be split into the following categories: Performance, Dependability, Cost, Maintenance, and End User Criteria. These groups have been selected as they adequately break up the design goals of this project in a manageable and legible way. We will be using Godot as our environment for developing our game. The main language we will be using to program or code is GDScript.
1.2.1 Performance

The project needs to be able to perform under general user conditions. In this sense, it needs to be able to run at an average of sixty (60) frames per second. This is an industry standard for performance shared among a vast number of games on the market\(^1\). The game is also going to need to feel responsive to the player. This means that actions taken within the game need a near instant visual or auditory response. This can be accomplished with animations, sound effects, or loading screens taking place during more strenuous times for the system.

1.2.2 Dependability

The system is going to need to be able to handle a player sending input in a sporadic and unpredictable way, due to the fact that our system is heavily event-driven. Mid-jump the player may feel inclined to activate an item, or while attacking they may attempt to attack in the opposite direction of their movement. This input is going to happen simultaneously and is going to have to be handled concurrently in a logical way. A player attacking to the left as they move right, needs to be facing left when they attack for the attack to make sense to the player. Failure to handle input of this nature at best can lead to undefined behavior. At worst, it could cause the system to become unresponsive or cause exploitable/game-breaking bugs to occur. Additionally, it is important to ensure that the game, in general, doesn’t have any game-breaking bugs, exploits, or crashes.

\(^1\) See Works Cited for source for this data.
1.2.3 Cost

In designing the system, there is a great emphasis on the development being virtually cost free. Since this development studio is a start-up with no external/third party financial support in any way, team members would have to pull non-existent money from empty pockets to fund expensive off-the-shelf components like specialty hardware and software. Any aspect of the program that is not being developed “in house” needs to be near free for this reason.

The cost of deployment on our side needs to also be lightweight. When releasing online to a PC gaming marketplace, cost of uploading and maintaining our storefront needs to be low in cost (at least until a return of investment is made on the time spent in the project).

1.2.4 Maintenance

The system needs to be able to handle the addition of various new entity classes to the system. This is due to the fact that the number of entities are going to be vast and after the conclusion of the project even more content will be added as the game is expanded from its initial demo phase. The player class and items are going to also need to be similarly expandable. As the new islands are added into the game’s core content, the player’s suite of abilities will expand with new items and functionality available in different levels. Because of this, new functions need to be able to be added on the fly. As new levels are introduced, new companion items will be added to the player’s repertoire, so any new functionality within a level needs to be able to be added without rewriting the whole program.
1.2.5 End User Criteria

The players of our system need to be able to achieve utility in our game like any other software program. In the case of this project, the players need to be able to have fun within the game. Many scholars have connected ideas of skill development and mastery to having fun in games. This philosophy will act as our guide towards trying to achieve this goal. In level designing, the gameplay of levels will focus on setting up challenges of increasing difficulty that require the player to master new skills and build upon old ones. In using skills developed to solve in-game challenges and develop new skills, the players will feel a sense of accomplishment which will lead to feelings of fun and joy.

1.3 Definitions, Acronyms, and Abbreviations


Team: Rascally Saints Productions.

RAD: Requirements Analysis Document.

PID: Project Inception Document.

Dungeon: a labyrinthian environment with various enemies, obstacles, puzzles and treasures.

RPG: role-playing game.

Side-Scroller: a style of game where the camera shows the world in profile.

Top-down: a style of game where the camera shows the world from a bird’s eye view.

\[2\] See Works Cited for sources used here.
Island: A level in the game.

NPC: non-player character.

Scene: an editable environment within Godot.

Pip: Main character of the game controlled by the user.

Player: The user controlled character within the game.

1.4 References

This document will reference Functional and Non-Functional requirements as well as use cases from the RAD v1.1 and may make references to ideas or requirements from the PID v1.

2 Proposed Software Architecture

2.1 Overview

Plight of Pip: Wrath of the Riper is a low budget game designed for computers with Windows based operating systems. The player will control Pip as he traverses across the interactive world. They will travel from island to island solving puzzles and defeating foes to collect pieces of the ultimate weapon; this ultimate weapon will be used to fight the antagonist the Riper. This task cannot be taken lightly as there is one, other foe -- the TIMER. It will track the time in the game and if the Riper is not defeated quickly, the timer will reach zero and all hope will be lost.
The simple graphics and easy to learn controls give the player a small learning curve but very engaging play experience. While each island can be played independently of each other in any order, they are also tied together as they feature aspects of other islands that can expand gameplay.

The game will be developed in a Client-Server architecture, where the player will communicate with the game, through key presses, to call functions of the components provided by the game.

2.2 Subsystem Decomposition

The following are some of the subsystems of our system. The project will also be using the components provided by the open source game engine Godot.

2.2.1 Player Subsystem

This subsystem will allow the user to move the player around the environment freely interacting with different objects in a variety of ways. It will also keep track of player data such as location and health. This subsystem contains the PlayerSidescrolling and PlayerTopdown classes. This subsystem will require interaction with many other subsystems to function fully. These include the NPC, Interactables, Inventory, Game State, and Menu Systems.

2.2.2 NPC

This subsystem is all about entities in the game that are not controlled by the user. This subsystem will allow the NPCs to move and interact with the environment as directed by their
game AI. It will keep track of the NPCs’ data such as locations, health, player interaction data, and environment interaction data. It will hold any scripted event data that the NPC uses. This subsystem contains the NPCsidescrolling, NPCTopdown, EnemySidescrolling, EnemyTopdown, Monkey, and Boss classes. This subsystem will require interaction with many other subsystems to function fully. These include the Player, Interactables, and Game State.

2.2.3 Interactables

This subsystem is all about the objects in the game that the player can interact with. These include the collectable objects, the pushable objects, the destructible objects, and the event object. It keeps track of the scripted events associated with these objects as well as what is required for the player to interact with them. This subsystem contains the SpecialEvent, Push, Event, and Collectable classes. This subsystem will require interaction with many other subsystems to function fully. These include the Player, System Flow Control, and Game State.

2.2.4 Inventory

This subsystem is all about the usable items in the game. This subsystem will keep track of whether they were collected, which items are active, and the function of each item in the game environment. This subsystem contains the Buddy class. This subsystem will require interaction with many other subsystems to function fully. These include the Player, Game State, Menu System, and NPC.
2.2.5 Game State

This subsystem contains all of the data for the game. It controls saving data from the system to the file, loading the data from a file into the system, loading and unloading scene data when transitioning scenes, and data contained that is accessible during the entire lifetime of the program. This subsystem contains the Global and GameData classes. This subsystem will require interaction with many other subsystems to function fully. These include the Player, NPC, Interactables, Inventory, Menu Systems, and System Flow Control.

2.2.6 Menu Systems

This subsystem contains user accessible functions that do not pertain to the continuing progress of the game. This will allow the interaction of the user with the system. This subsystem contains the Options, OptionsMenu, PauseMenu, Inventory, and Title classes. This subsystem will require interaction with many other subsystems to function to the fullest extent. These include the Player, Game State, and System Flow Controls.

2.2.7 System Flow Controls

This subsystem contains any hidden backend systems which will aid in the flow of the game, but the user has no ability interact with. These do not need interactions with other systems to function but are relied upon by other systems to maintain their functionality. This subsystem contains the SceneTransition, Fade, GUI, SelectionAnimation, LoadScreen, and SubselectionAnimation classes.
2.3 Subsystem Services

2.4 Hardware/Software Mapping

Hardware/software mapping for this game is relatively straight-forward. Our system is entirely contained on a single machine as a standalone system. It will interface with the Godot Engine, but this will be located on the same machine. The user will communicate with the system using a keyboard or keyboard-equivalent control device. Keys will be mapped to actions within the game, depending on the game state, and the user will use these key-action pairs to complete tasks within the system. Besides this communication, the system only requires mapping to the user’s file system in an incredibly insignificant way. Within the directory the game has been installed, the system will create game save files for the user. The memory required for this is assumed to be small, as only a small portion of the game is required to be persistent from one session to the next.
In Game Controls:

<table>
<thead>
<tr>
<th>Keys</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>W/A/S/D</td>
<td>Move</td>
</tr>
<tr>
<td>W(Side-Scrolling)</td>
<td>Jump</td>
</tr>
<tr>
<td>Q</td>
<td>Use Item 1</td>
</tr>
<tr>
<td>E</td>
<td>Use Item 2</td>
</tr>
<tr>
<td>F</td>
<td>Interact with entity/object</td>
</tr>
<tr>
<td>I</td>
<td>Open Inventory</td>
</tr>
<tr>
<td>Esc</td>
<td>Pause</td>
</tr>
<tr>
<td>Space</td>
<td>Scarf grapple</td>
</tr>
</tbody>
</table>

Menu Controls:

<table>
<thead>
<tr>
<th>Keys</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>W/A/S/D</td>
<td>Move between Menu items</td>
</tr>
<tr>
<td>Enter</td>
<td>Select Menu item</td>
</tr>
</tbody>
</table>

* Controls are assumed to be based on a regular QWERTY keyboard layout.

** Gamepad support is a result of the user mapping their controllers to the keys used by this system.
2.6 Persistent Data Management

When the user is planning to eventually leave our system, they may wish to save information pertaining to the current game state so that they can pick up where they left off. Within the system, there needs to be a viable method of storing this game state data within the computer host so that the game can retrieve it at a later time.

Fortunately, the implementation of this functionality is detailed in the Godot documentation. Detailed below is a summary of the documentation that explains how games will be saved within the system, based on Godot documentation standards.

The system will have the current player node, story nodes that store progress in plots, the current scene nodes, and any entity nodes within the current scene tagged as persistent. The nodes will be called upon based on this tag and serialized with an in-house function. After serializing the data, we will procedurally write to a new JSON file, that contains the serialized game state data, to a hidden directory on the user’s system. Once the file is written and closed, the game will be considered saved.

Loading the game will simply be a matter of reading the file and using Godot provided functions to retrieve the serialized objects contained within it. With the objects retrieved, the system will display the current scene based upon the data and will ensure progress nodes have been set to those retrieved from the save.
2.7 Access Control and Security

Since each download of the game will be a single-user based system, required security and access control efforts are fairly low and trivial. In fact, the proposed system only contains access control inherited by the host system. This access control is inherited as game data is stored within the user’s home directory tree, effectively hiding it from others as long as the user’s login information is not compromised. This is fairly standard in the industry and no data in the game will be made publicly available to other users as long as the user treats their machine securely. Should the system become compromised, there is no threat of the user’s sensitive data becoming available as no sensitive data is even held within the system!

One portion of security that should be mentioned is game save security and protection. The Godot Engine provides functions and various documentation to aid in the development of security pertaining to user save data. For the purposes of security, it is assumed that the host machine of the system is fully owned by the user and they have (or are able to obtain) full administrative privileges concerning our system’s various files. A likely attack in this scenario is that the user attempts to cheat within the system by making modifications to save data pertaining to their player’s health, items acquired, and progress in quests. If done incorrectly, this could damage the integrity of the save files and cause the player to lose most (if not all) progress in the game! Even if done correctly, the experience of the system is intended to be done without the user becoming invincible or skipping content entirely. For these reasons, it is not considered ideal for the user to be able to easily access this data. To prevent the user from tampering with save files easily and damaging the experience, save files will be encrypted upon saving with

Rascally Saints 15
tools provided by the Godot Engine. This will protect game save files from manipulation as long as the user is not supplied with unlimited computing power or a monumental amount of time.

2.8 Boundary Conditions

The final output of the application is an executable that can be directly accessed from the user’s computer. System initialization will begin when the user selects the program and runs it. To shutdown the program the user will select the exit button on the window of the program while it is running.
Works Cited


