The Trash Pandas Horror Game
Design Documentation

Group Members: Matthew Tobino, Nicholas Lotierzo, Komalpreet Dhinju, James Robinson

Rowan University
Senior Project
Spring 2024

Table of Contents

1 Purpose

As with most video games, the goal of the project is to provide an enjoyable and fun experience for any and all who play it. In the case of this game, the fun will come from the exhilarating experience of running from an enemy whose location is **unbeknownst** to the player, while trying to find items to allow progression to the next stage of the game. This style of gameplay consistently provides tense situations and satisfying payoffs that the player(s) will enjoy.

2 Scope

This game will not have very intense graphics or high-quality sounds, as it is not within the scope of the project. However, the graphics and sounds that will be included will fit the game's overall style better, allowing there to be no need to venture outside of the scope in that department. In terms of the overall gameplay, there will be no more than 3 stages to go through until the player completes the game.

2.1 Requirements

2.1.1 Requirements

As with any project, there are requirements that must be met. Listed second are the requirements we set for ourselves to meet and exceed. The first list is a guide to our requirements so that one can understand what each requirement is designed around.

- P.1.1 – The default player shall use the keyboard and mouse to navigate through the maps.
- P.1.2 – The default player shall interact with collectible objects such as keys to doors in the mansion, or items to reconstruct the bridge in the forest.

- P.1.3 – The default player shall be capable of holding up to 4 objects at a time.
- P.1.4 – The default player shall use “F” to toggle their flashlight.
- P.1.5 – The default player shall use “ESC” to open the menu.
- P.1.6 – The default player shall complete some task to advance in the game.
- P.1.7 – The default player shall choose a class that provides a unique ability that will assist the player during their playthrough.
-
- E.1.1 – The enemy shall move around in the environment the player(s) is/are present in.
- E.1.2 – The enemy shall target a player if it gets too close.
- E.1.3 – The enemy shall become more challenging with each stage.
- E.1.4 – The enemy shall track the player by sound.
- E.1.5 – The enemy shall attack the player if they get too close or the enemy manages to catch up to the player.
- M.1.1 – The overarching map of the game shall include two different maps.
- M.1.2 – Each stage shall have a different objective to move on.
- M.1.3 – The first map, the Forest, shall include an open area with many trees, night sky, foggy areas, rocks, and abandoned buildings.
- M.1.4 – The second map, the Mansion, shall include intricately designed and confusing hallways, corridors, and rooms, spread out across a vast mansion.
- M.1.5 – The win condition is to escape the mansion by collecting keys to unlock the door to the helipad.
- G.1.1 – The game menu shall display 3 buttons.
- G.1.2 – The game menu shall give the player an option to start the game, edit their settings, or quit the application.
- S.1.1 – The title screen shall display buttons to start the game, quit the game, edit the settings of the game, and review the controls of the game.
- S.1.2 – The victory screen shall display buttons to restart the game, go back to the start screen, and quit the game.
- S.1.3 – The defeat screen shall display buttons to restart the level, restart the game, and return to the title.
- S.1.4 – The “How to Play” screen shall display the basic controls of the game in an easy-to-understand manner.

2.1.2 Usage

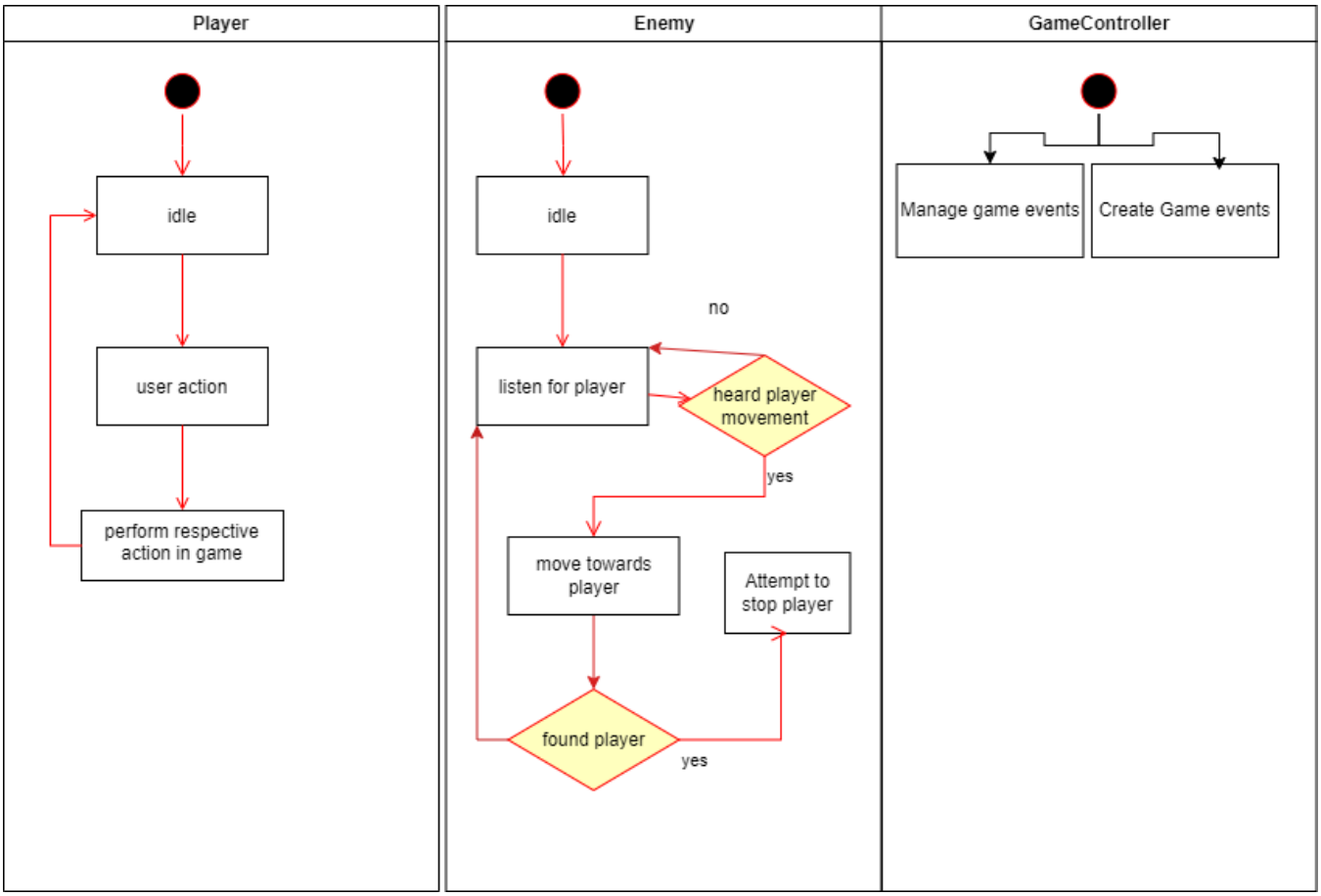
The usage of our game is solely up to the user. We may design the game to be played a certain way, but it is up to the player to determine how they want to play the game.

2.1.2.1 User Roles

The user’s role is to play as the main character of the game. This will allow the player to control how they want the game to run. They can make it harder on themselves by ignoring the objectives/goals that the game tries to achieve. They could also make it easier on themselves by playing to those objectives/goals and working towards the end of the game. The true purpose

of the user's role is to have fun while playing the game we design and that allows them to use the final product in any way they please.

2.1.2.2 *Activity Diagram*



2.2 Assumptions and Limitations

2.2.1 Minimum Required System Specifications

Operating System	Windows 10
Processor	Intel Core i7-8565U @ 1.8 GHZ
Graphics	Intel UHD Graphics 620
DirectX	Version 12
Storage	1 GB

2.2.2 Design Exclusions

Our game is currently designed for a Windows Operating System with game controller support. This is because all our systems use Windows OS, and thus we cannot accurately test our game on Mac, Linux, Console, or Mobile platforms.

3 Design

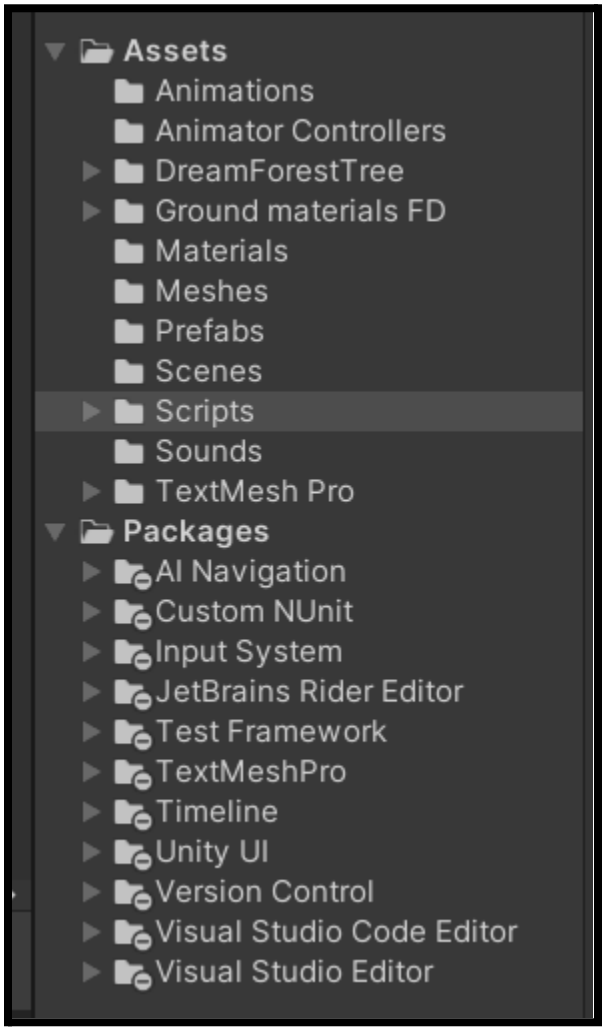
3.1 High-Level Overview

Since C# is essentially just Microsoft's version of Java, the design for this project will be using a lot of object-oriented design principles. The first of which would be the use of an abstract class to create different unique ability subclasses for the player to choose from. This project is being made inside Unity Engine 2022 and a benefit of that is that everything we are working on within Unity will be sealed up in a package and it will automatically create an executable file, making it incredibly easy for deployment.

3.1.1 Packages Needed

The user is not required to install any external packages. The only packages they will need to install are the game files which will be accessed entirely through the executable file. If you are a developer who wants to add to the game itself through development, you would just need to install a couple of packages for the assets: DreamForestTrees, GroundMaterialsFD, plus the custom-made GameObjects.

3.1.2 File and Folder Hierarchy



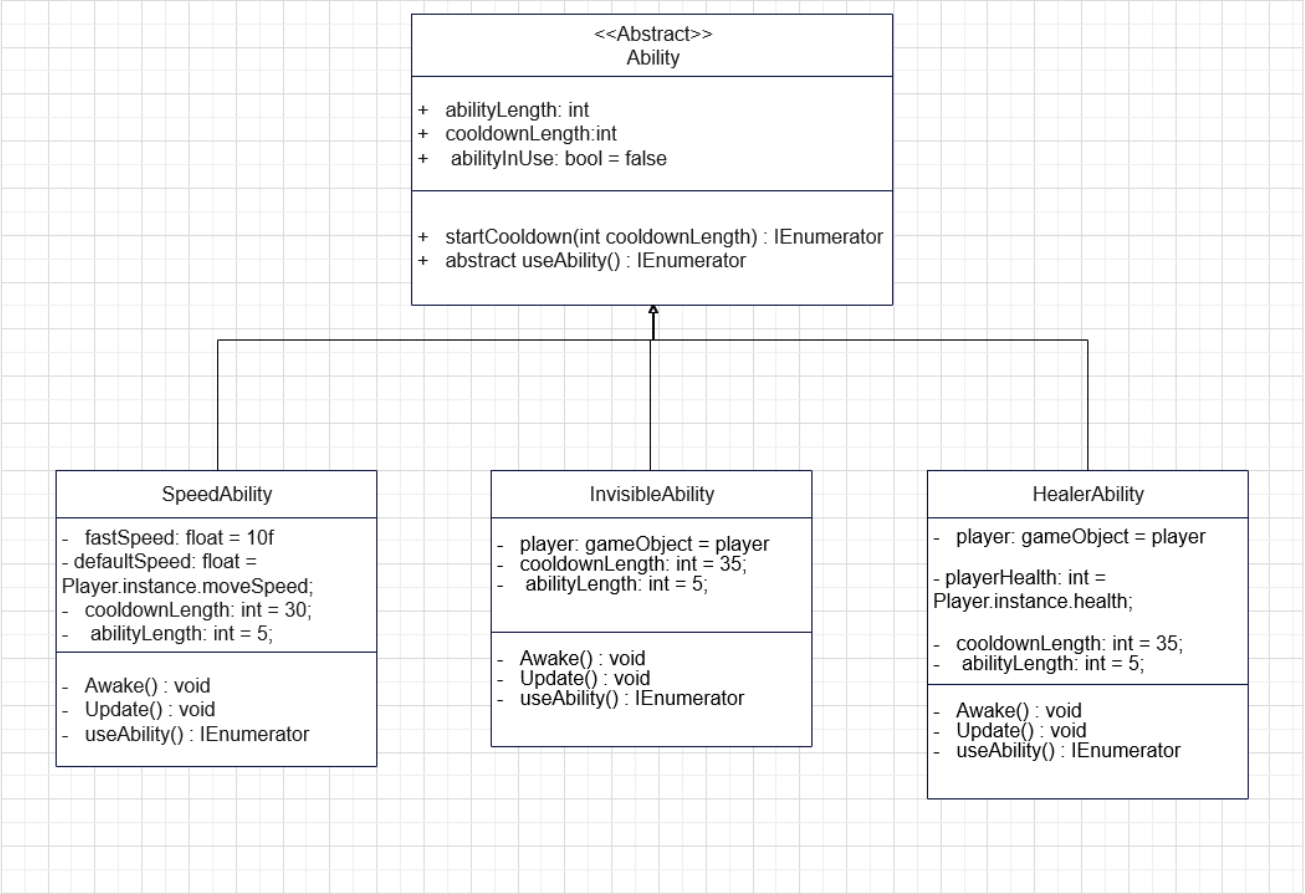
3.2 Components

3.2.1 Ability Components

3.2.1.1 Interface Design

For the ability design, I wanted to make it as easy to add new abilities down the road as possible if need be and I wanted as little redundant code as possible. So, I made Ability an abstract class where I declared the baseline variables and methods to be used. For example, each ability must have an abilityLength (duration) and a cooldownLength, but the exact numbers might be different for each class depending on how good the ability is. The startCooldown function is a pretty universal method since it wouldn't need to necessarily be specific to each ability, the only thing you would need to change is the individual value of the variable "cooldownLength" in the respective class. The abilities mainly utilize coroutines for the methods within each class which makes it a lot easier to keep track of how long an ability and its respective cooldown will last for.

3.2.1.2 Initial Object Diagrams



3.2.1.3 Speed Ability

This ability increases the player's speed temporarily which is done by grabbing an instance of the Player.cs class and modifying the moveSpeed variable for a short duration.

3.2.1.4 Invisible Ability

This ability makes the player invisible to the AI for a temporary amount of time which is done by grabbing the EnemyAI.cs class and changing the enemy's sightRange and attackRange both to 0. This makes it so that no matter how close the enemy is to you, using this ability essentially makes you invisible to the enemy and he will go back to his patrolling state.

3.2.1.5 Healer Ability

For the ability design, I wanted to make it as easy to add new abilities down the road as possible if need be and I wanted as little redundant code as possible. So, I made Ability an abstract class where I declared the baseline variables and methods to be used. Each ability

3.2.1.5 Enemy AI

3.2.1.5.1 AI Design of Stage 1

For the enemy AI, I utilized some tools that Unity already has to make this process a lot easier. The NavMesh is able to perform spatial queries like pathfinding. In order to use this though you must first bake the NavMesh component which is essentially a map of locations the AI can or can not traverse.

The first level was essentially the test to see what the limitations were of this navmesh and to find the best way to go about programming the AI. The enemy at this stage has three main states he can be in at any given time: patrolling, chasing and attacking.

While patrolling, the enemy generates a random spot anywhere within 25 units around him on the navmesh and walks to that location. After getting to this location, a new random spot is generated and this pattern continues until it gets broken one of two ways. The enemy can hear up to 65 units away and the player makes the most noise while sprinting. So, if the player is within that 65 unit radius while sprinting, the AI will drop what it's doing and run to the location it last heard the player at. The other way to break out of the patrolling state is if the player happens to walk into the enemy's site or attack range.

The enemy's chasing state will continue until he reaches the player or until the player is out of the site range.

The enemy's attacking state will continue as long as the player is within that certain distance from the enemy.

The enemy has two ways to "attack" the player, throwing rocks and bonking the player on the head. In this stage, there is an 80% chance that the enemy will bonk the player on the head for 20 damage every 3 seconds. There is a 20% chance that he will instead throw a rock at the player for 25 damage, but it is pretty inaccurate.

3.2.1.5.1 AI Design of Stage 2

As with the first stage, I also used Unity's NavMesh here but it was actually a lot easier to implement. There was no terrain with trees involved, it was just walls, doors and a floor so the navmesh was able to map out a much better area for the enemy to walk in. Plus, I utilized a different navmesh function called SamplePosition which tests the random spot first to see if it's even accessible and if it's not then it moves the spot to the closest area that the AI can walk to. The sight range and attack range are also a lot smaller in this level since it's a smaller map with close quarters rather than a large open terrain. But similar to the first level, if the enemy sees you then it will chase you until it's able to get close enough to attack you. The enemy in this stage only utilizes one attack, throwing that rock once every 3 seconds when the player is within range. The enemy is smaller and moves slower than it did on stage 1, but there are also three of them. One is located somewhere within each subsection of the mansion level, so eventually you might have three of these small enemies chasing and trying to attack you.

3.2.2 Player Control

3.2.2.1 How we handled Player Control

Unity has an InputSystem that allows users to define what buttons will do what. This replaced the old system of long-winded if checks for if a button was pressed. Building upon their new system, we utilized Unity's Event System to fire off events when buttons were pressed. This allows us to not have to continuously check in our Player class if a button was pressed but rather delegate that task to a different class (GameInput in our case). That class contains a reference to the InputSystem and fires off events when it detects an input.

3.2.3 Environment

The environment is crucial to the success of any game. As such we had to take measures to ensure ours was just right as a horror game can only be as good as the environment it takes place in.

3.2.3.1 Sound

We used royalty-free sounds from Pixabay which allowed us to not have to spend hours on sound design. From there we attached sound objects to the player and the enemy and set up state machines to play the sound only when it should be played. Additionally, we also set up a Random Sound Generator which spawns in hitboxes in random locations around the maps to play a sound that might spook the player.

3.2.3.2 Lighting

Horror games generally take place in dark settings and our game is no different. We utilized Unity's Post Processing Package and a Night Sky Skybox from the Unity Asset Store to darken the surroundings and give the game that eerie feeling. This also encourages the player to use their flashlight for better visibility and rely on their sense of hearing to watch out for potential dangers.

3.2.4 Inventory System

The inventory system plays an essential role in the progression of the player through the levels of the game. The first level requires the player to collect four wooden planks, two ropes, and two gears to gain access to the bridge which then leads to the mansion. The second level requires the player to collect a key which will open to a second area of the mansion, collect a key from the second area which will open to a third area, and collect a key from the third area which will open to the helipad and allow escape from the enemy.

3.2.4.1 Item

We created our items as scriptable objects. Considering that we would have several of the same items spread across the scene, this allowed us to manage the properties of the items centrally and ensure a consistent game experience. The data regarding each type of item - plank, rope, gear, or key - was then attached to each physical collectable object present in the scene through an Item Pickup System. This way, when the objects get picked up, all the item data would be transferred to and stored in a list in the Inventory Manager.

3.2.4.2 Management

The Inventory Manager progressed from a game wide script to an interface that would be used to write scene specific inventory manager scripts. This change occurred late into the project as we realized the different interaction requirements for the bridge items and the key items. Nonetheless, the Inventory Managers are responsible for methods to add items, remove items, use items, drop items, and select items. The user accesses these methods through different player inputs, like pressing keys on the keyboard or using the mouse to click UI elements.

3.2.4.3 Interactions & Using Items

To simulate using the items, many interactions were put into place. First, the item goes from being a physical object in the scene to being an icon on the UI.

The level one bridge interaction allows the player to approach a physical bridge, open the bridge repair menu, use any items in the inventory, and visually see that part of the bridge being filled in on the menu. Once all items have been used, an animation simulates the bridge lowering. These sequences of events are controlled by Bridge Interaction Script, and its methods that were included in the use method of the Inventory Manager.

The second level door interaction allows the player to approach a door, use any key in the inventory, visually watch a key turning animation and door unlocking animation. There are three door states that the player stumbles upon: trying to open a door without a key, trying to open the wrong door with a key, and successfully opening a door with a key. These states and sequence of events are controlled by Door Interaction Scripts, and its methods that are included in the use method of the Inventory Manager.

4 Technical Architecture

This game is developed using the Unity Engine, providing extensive support for cross platform development. The technical architecture allows for a smooth and flexible integration of game mechanics and immersive environments. In regards to deployment, the game is initially set up to run locally with builds tailored for Windows, or any other platform we would like to support and will be able to test on. Unity offers a streamlined deployment process that makes testing and distributing the game across devices effortless.

4.1 Deployment Instructions

To deploy the game locally, start by adjusting the project settings in the Unity Editor to match the requirements. Navigate to build settings, select the target platform, and proceed to Build. This process will generate an executable file and a data folder that is essential for running the game. To make access even simpler, we plan to package these into an installer, enabling users to install the game on their devices for quick launch.

5 Configuration

Leveraging Unity's capabilities, this game will be optimized for seamless play on the Windows operating system, and any platform we are able to perform testing on. Simple configurations that are being considered, but are not guaranteed, include the choice between single or multiplayer modes and using AI for different levels of difficulty. These settings aim to tailor the gaming experience without complicating the core code.

5.1 Configuration Files

To enable easy adjustments in-game difficulty, game modes, or simple settings, we will utilize JSON configuration files. These files provide a direct way to vary gameplay challenges without altering the game's fundamental design, therefore keeping the game accessible and exciting. No such configuration files have been created at this stage in the development process.

6 Roles and Responsibilities

Quick Summary

Role	Member
Scrum Master & Developer	Komalpreet Dhinju
Trello Board & Developer	Matthew Tobino
Game Models, Animations & Developer	Nicholas Lotierzo
Concept Artist & Developer	James Robinson

7 Terms and Definitions

- User: The individual who interacts with the game interface and controls the actions of the player character within the virtual environment.
- Player: The primary character controlled by the user throughout the game.
- Collectable Objects: Items within the game environment that the player can interact with, like bridge materials and keys, needed for progression.
- Enemy: The antagonistic entity within the game that poses a threat to the player.
- AI (Artificial Intelligence): The programming that governs the behavior of the Enemy in this game.
- Ability: A special skill or capability possessed by the player character that will be activated at the start of the game and utilized during gameplay to achieve advantages.
- NavMesh: Navigation Mesh, is a built-in feature of the Unity game engine that enables the creation of dynamic, pathfinding systems for the AI-controlled characters within the game environment. NavMesh works by creating a simplified representation of the game

world's geometry, allowing the AI Enemy, in our case, to navigate and move intelligently through the space.

8 Supporting References

9 Revision History

- **Version 1 – Version 1.1:** Requirements now reflect the change in levels and win conditions for each level. We took out level three due to time constraints and placed the win condition in level two. Some small grammatical changes were made as well.
- **Version 1.1 - Version 1.2:** Updated the Components Section with the latest information on our components and Requirements with the latest version of the requirements
- **Version 1.2 - Version 1.3:** We added the Terms & Definitions sections, and filled in the necessary information.