Last Updated: 3/11/2025

Developer Documentation

General Information

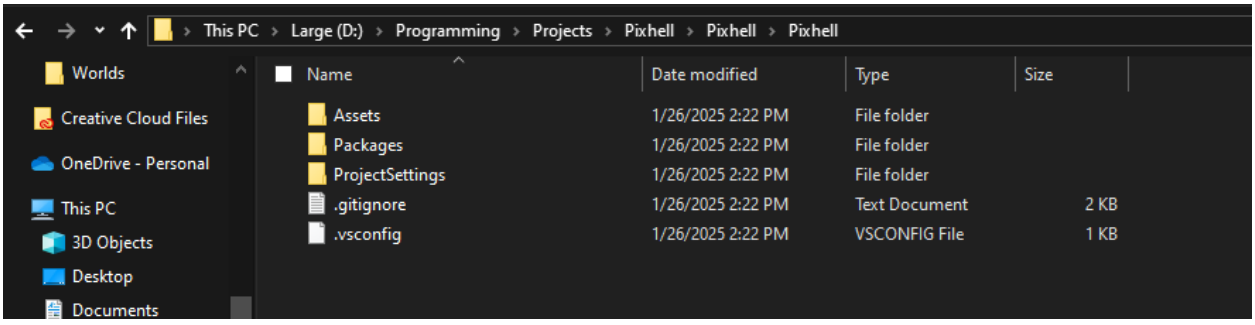
Source Code

The source code is located in the Pixhell folder located in the repository found here. Inside this folder, the folder with most of the code is the Assets folder, which contains various folders related to sprites, animations, scenes, and scripts. This folder also contains the StreamingAssets folder, which is home to information such as play run information and item data. The other folders located in the Pixhell folder contain information related to Unity settings and are generally automatically generated by Unity.

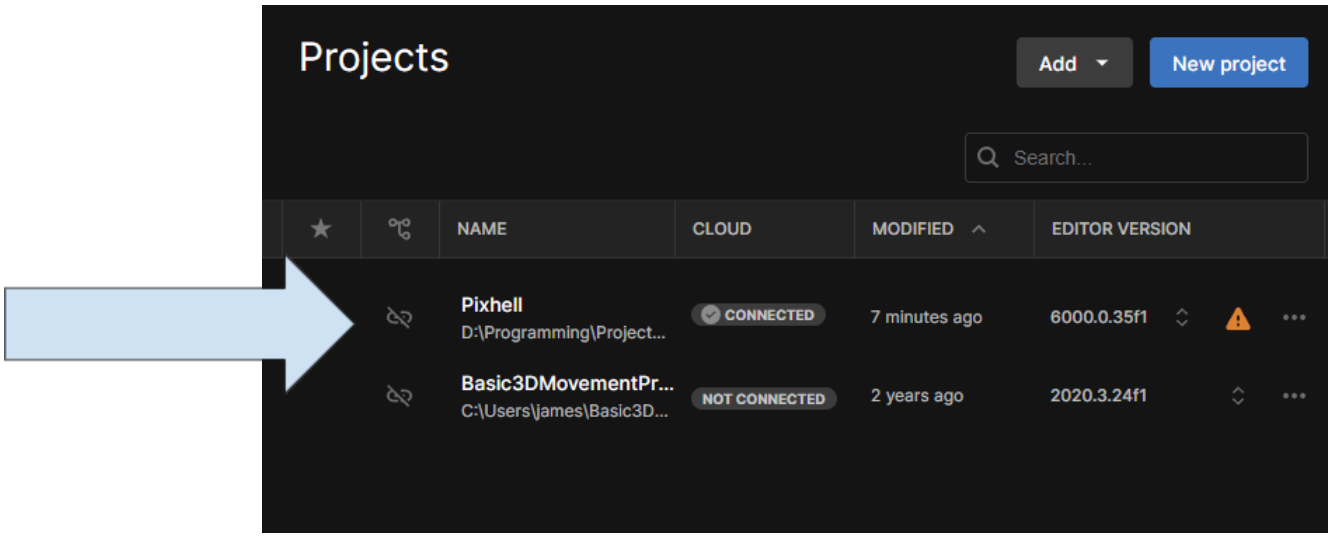
The documentation, weekly reports, and other general information are all located outside of the Pixhell folder and are in the source folder of the repository found here. The reports are found in the "/reports" folder, and the documentation is found in the "/documentation" folder.

In order to open the source code and test the project:

1. Download Github Desktop - Download GitHub Desktop | GitHub Desktop
2. Clone the repository to an easily findable folder - Use this link: https://github.com/CS362-Team12/Pixhell.git
3. Download Unity Hub for desktop - Start Your Creative Projects and Download the Unity Hub | Unity
4. In Projects - click the add button -> Add project from disk
5. Navigate to the Repository folder
6. Select the Unity folder, which is the folder that has Assets, Packages, and Project Settings



7. Download any needed updates (correct Unity version, etc)
8. Open the Pixhell project



Layout

Pixhell/

This is the root of our repository, and contains folders related to documentation, the source code, presentations, reports, and our living document

Pixhell/documentation

Contains developer and user documentation

Pixhell/presentations and /reports

Contains our project proposal, midterm, and weekly reports respectively

Pixhell/Pixhell

This folder is the root of our source code. All of the project's assets, and unity settings are contained within this folder.

Pixhell/Pixhell/Assets

This folder contains the majority of our work on this project, including sprites, animations, and scripts.

.../Assets/Animations, /Hero and Opponents, /Evil Wizard 3

These folders contain the majority of the animation and animation controllers for the game.

.../Assets/PNGs

This folder contains the majority of the UI elements, including the health and XP bar sprites and the dash ability icon sprite.

.../Assets/Prefabs and /Resources

These folders contains all of the pre-configured objects that are used throughout the game, such as the player characters, enemies, triggers, damage text, among other things. Any prefabs that are accessed through script are located in the /Resources folder.

.../Assets/Scene

This folder contains all of the scenes for levels and game environments, including our global scene, menus, lobby (Limbo) and arenas that are used by Unity.

.../Assets/Scripts

This folder contains all of our scripting and behavior control, which includes character control, inventory, enemy spawning, sound control, and many others.

.../Assets/StreamingAssets

This folder stores essential game data during runtime, including player data (such as items and money), and arena data (such as enemy wave patterns).

.../Assets/Tests/TestS

This folder contains the test scripts that can be automatically run through Unity to verify that features have not broken between updates.

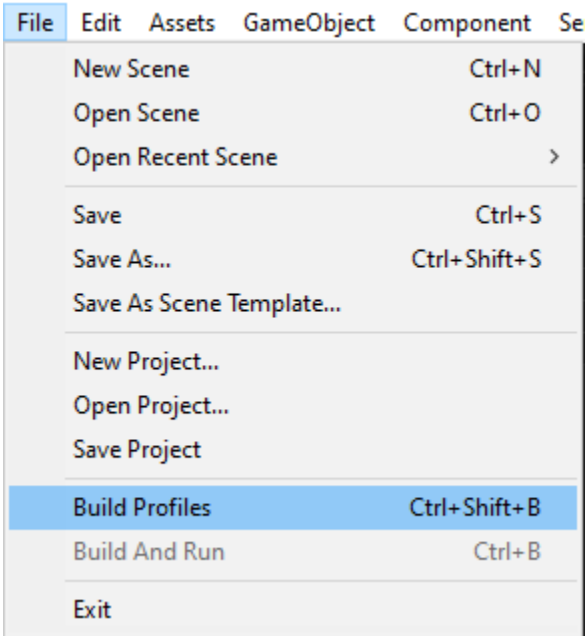
Building

The build process is mostly automated by Unity, with the custom build scripts located within the project when obtaining the source code. Here are the process steps for building a new version:

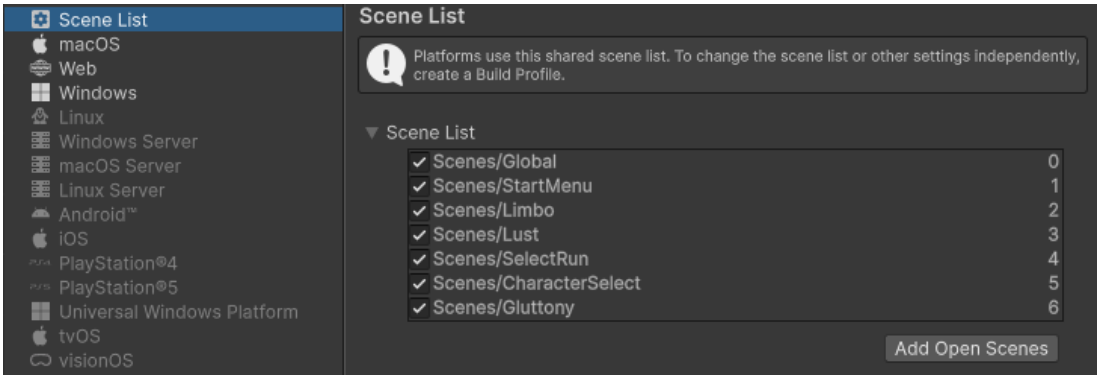
Building

The following instructions assume that Pixhell has already been launched through Unity and the Unity Editor is open. If not, the instructions for that are located above.

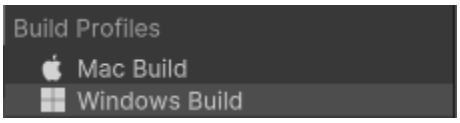
1. Open Pixhell in Unity
2. Ensure that in `scripts/GameConstants.cs` DEBUG is set to `false`. Leaving DEBUG on can cause some debug effects, such as increased drop rates and forced upgrades, to be on. Changing this variable to `false` will force these effects to be off.
3. For official builds (non beta), any save files should also be deleted from the local machine. To do this, navigate to Assets/StreamingAssets/Runs and delete all files there. Run the project in Unity to ensure that no runs are saved.
4. Go to File > Build Profiles



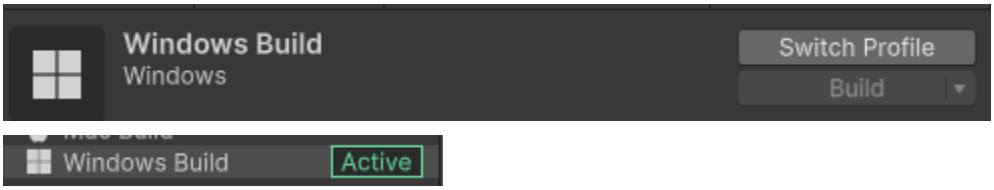
5. Check that the Scene List contains all the scenes included in the game. If necessary, click “Add Open Scenes” to include all the open scenes. See the image below for scenes from the last update.



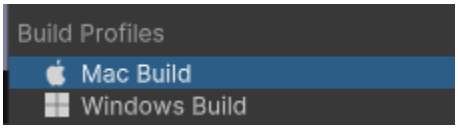
6. Windows:
    a. Find “Windows Build”



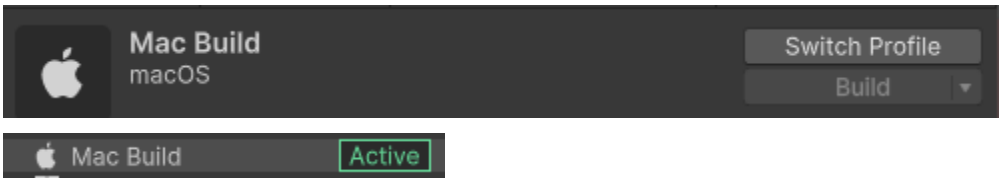
    b. Click “Switch Profile” to make the Windows Build active



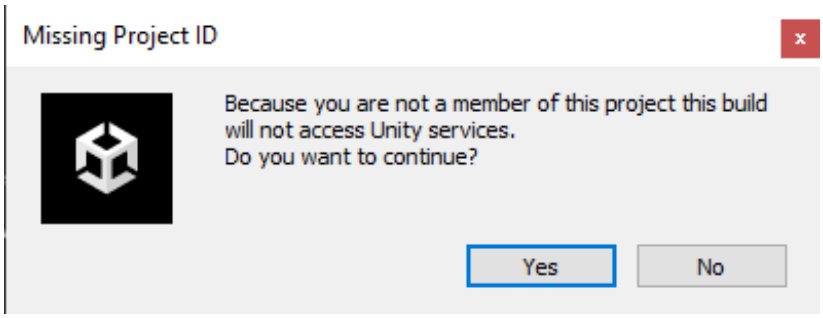
7. Mac:
    a. Find “Mac Build”



    b. Click “Switch Profile” to make the Mac Build active



8. Click Build on your chosen operating system, and select a folder that is easy to find.
    a. If you get the error pictured below, click “Yes” to continue. The project does not rely on Unity services and the build should continue without it.



9. Navigate to the folder the game got built (in some cases, the folder will automatically open). Run pixhell.exe to ensure it works.

Releasing on Github:

10. Navigate to https://github.com/CS362-Team12/Pixhell/releases
11. Utilize Github best practices for releases: https://docs.github.com/en/repositories/releasing-projects-on-github/managing-releases-in-a-repository
    a. Choose a proper tag based on version history

12. Attach a zip file of the Mac build and Windows build from the previous steps. Name them Mac-<TAG>.zip and Windows-<TAG>.zip, where you replace <TAG> with the name of the tag used.
13. Select “Set as a pre-release” for beta and draft versions.
14. Click “Publish release”.

Tests

Testing is currently done both automatically and manually in Pixhell.

Automated White Box testing is done through a script located in Pixhell/Assets/Tests/TestS. These tests automatically interact with the game and verify that certain game states have been reached, and can be used before and after updating code to ensure that the game functionality has not unexpectedly changed. The instructions for running these tests are located below.

Manual Black Box testing is currently done through debugging statements and console logs while playing. For instance, when getting hit as a player, the health will be displayed in the console each time the player is hit, and so the health can be compared to an expected output.

Adding Tests

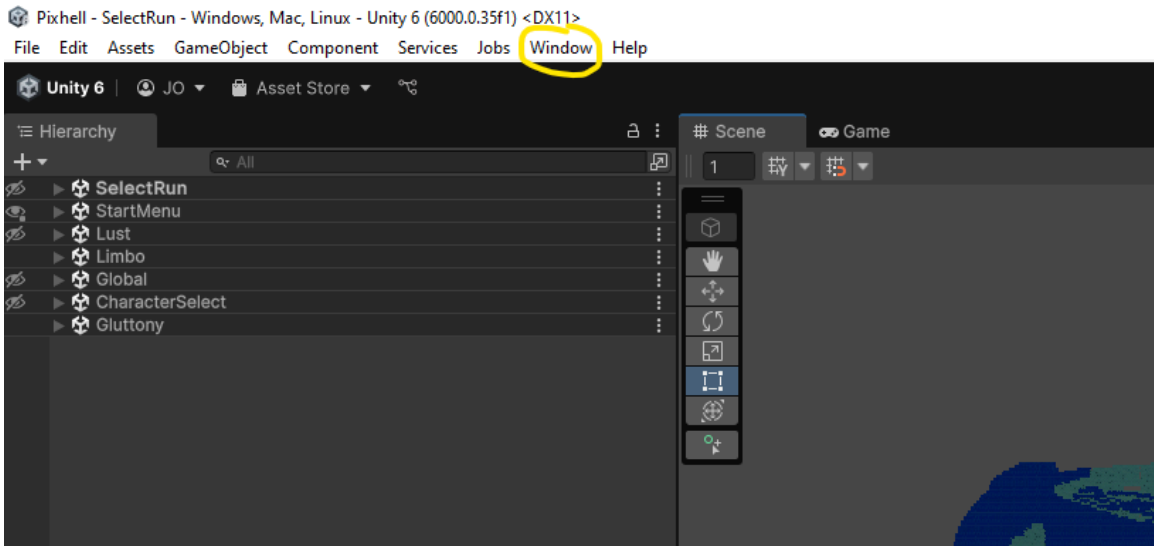
To add Automated White Box tests, tests should be IEnumerators that have descriptive names such as “TestRunSelectCreateButton,” so it can be easily identified in the test runner window. If the test requires a setup other than the one currently in the GeneralTests.cs file, then create a similar file with a different Setup function (tagged with SetUp) in the same folder.

To add Manual Black Box tests, simply add Debug.Log statements throughout the code.

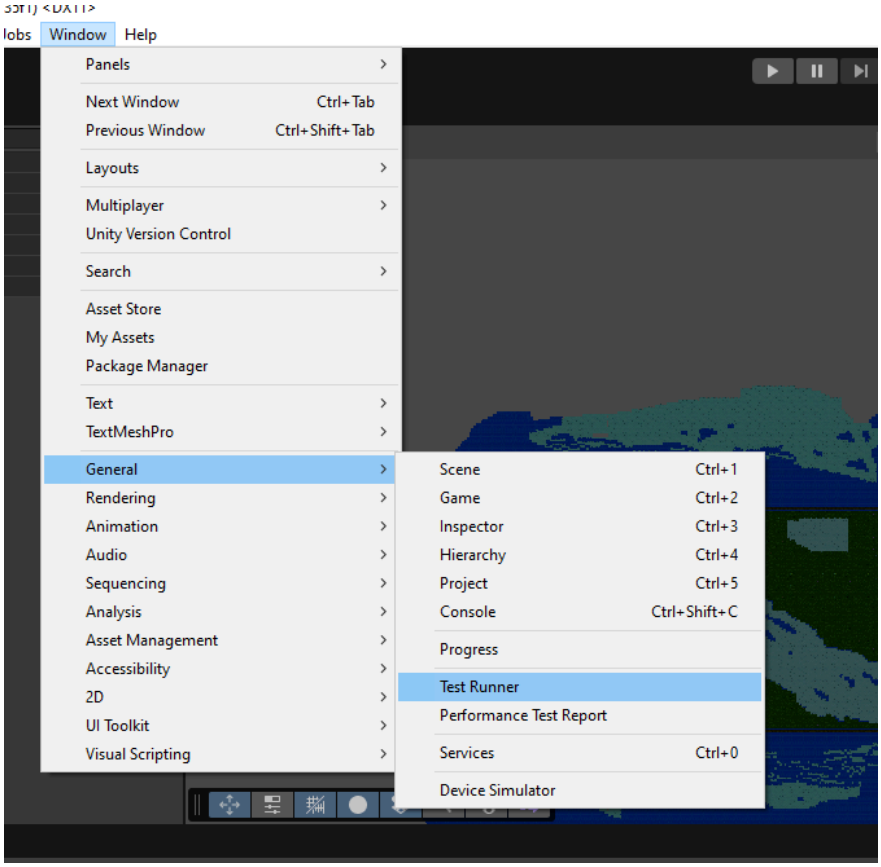
Running Automated Tests

The following instructions assume that Pixhell has already been launched through Unity and the Unity Editor is open. If not, the instructions for that are located above.

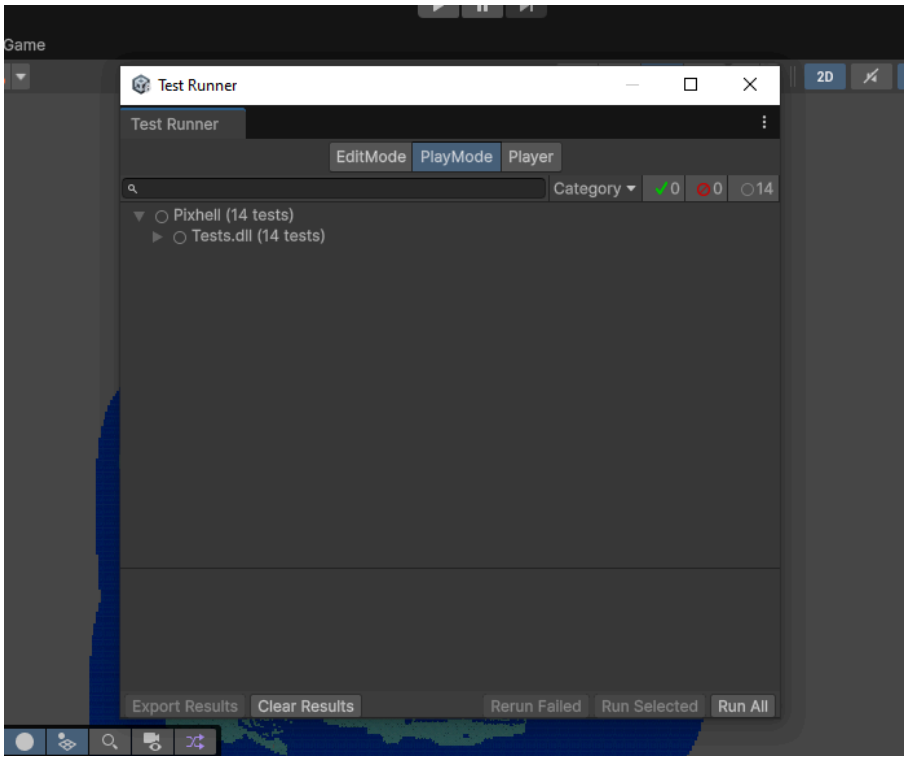
1. Click the “Window” option on the menu bar to open the dropdown.



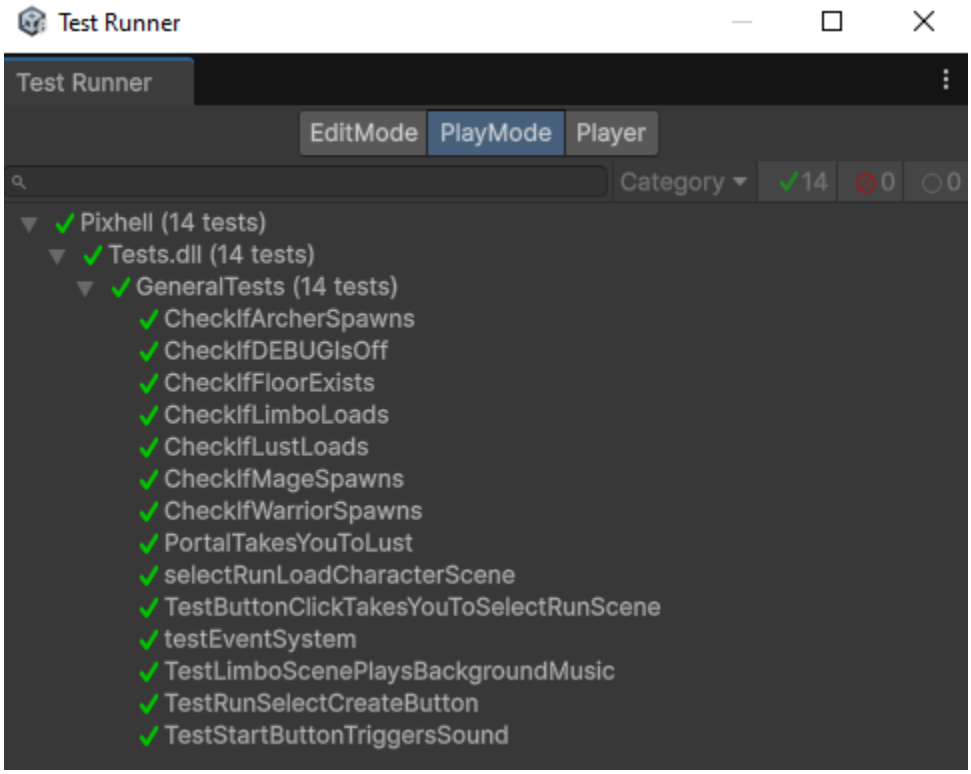
2. Navigate to General -> Test Runner. This will open the Test Runner popout window.



3. At the top of the popout, click “PlayMode”.



4. Select “Run All” to run all of the listed tests. If you wish to only run a group of tests, select them and press “Run Selected.”
5. If all tests pass, it should look like this. Failed tests will have a red X instead of a green check mark, and clicking on it will show what error was thrown.



Mechanic Specific

Characters

The characters are built using polymorphism off of the character controller class. The character controller class script has all default values set in start and checks for all movement and animation changes in the update function. It also includes functions that all characters will incorporate to their build. The start, update, and attack functions are protected overridable functions so that each individual character can override it for their own purposes. For instance each class overrides basic things like the damage that character can do and health that character has and each has their own attack abilities.

Items

Assets/Scripts/Items/ and Assets/StreamingAssets/Items/GlobalItems.csv

Adding Items:

Adding items can be done through the GlobalItems.csv file, by simply adding a new row to the file. All numbers are percentage based (so 100 = 100% increase), and additive. A sprite file location will need to be added (all of which are relative to the StreamingAssets folder) for display in the shop.

Once items are added, they should automatically show up in the shop in the lobby (Limbo) to be purchased.

Enemies

Assets/Scripts/Combat/Enemies/

Base Enemy Class:

States

Enemies have a state table which determines **what** actions they perform.
Use the following game constants to create an array of the states you want the enemy to be in.
GameConstants: MOVING, ATTACKING, IDLING, CHARGING, HOMINGATTACK
More GameConstants and states can easily be added by updating the Enemy.cs file to include:
1. A condition checking for that state in the Update() function
2. A function call to a public, virtual function. This should not affect any other enemies. Then, add the constant to GameConstants.cs in the scripts folder using the next available integer.

For example:
states = [MOVING, IDLING, ATTACKING, IDLING] will have an enemy move around, then idle, then attack, then idle again, before looping back to the start of the array.

Timers

Enemies have a timer table which determines **how long** they perform their actions. This should be a float for how many seconds you wish for that state to run.
All timers are scaled with a multiplier of 0.8 to 1.2.
This ensures that enemies spawned around the same time don't perform on the exact same schedule, and adds variability to timings. If needed, a way to surpass this randomness could be added.

Heal Panic State

When enemies drop below a given health percentage (30% by default) they enter a panic state where they look for healing. Then, one of two things happens

1. They find healing: If they find healing, they will drop their current action and start moving quickly towards the healing.
2. They don't find healing: If they can't find healing, they'll continue their normal actions.

They stay in this panic state until they go above a certain health level (50% by default). Upon returning to 50% health, they resume their normal states and timers.

Restrictions

The timer array should be the same length as the states array
Any state which is ATTACKING should be of length 0: Any timer of length zero will run for exactly one frame. If you wish for the enemy to attack multiple times, or for a length of time, either add a new state, or have the enemy be in the ATTACKING state multiple times in a row. i.e. [MOVING, ATTACKING, ATTACKING, IDLE].

For example, using
states = [MOVING, IDLING, ATTACKING, IDLING], and
timers = [3f, 0.75f, 0f, 0.75f],
Will make the enemy move for 3 seconds, idle for ¾ of a second, attack, and then idle for ¾ of a second.

Animations

For each state change, either a boolean or a trigger is sent to the enemy's animation controller. Every enemy has a centralized animation controller, with identical parameters, which are used to create conditions for animation switching.

For example,
When a character's state is set to move, is_moving is set to true and is_shooting is set to false. Or when a character dies, a 'die' trigger is sent to the controller which starts the death animation regardless of any animation state.

Upgrades

Assets/Scripts/Upgrades

Level Up

The LevelUp.cs script manages player XP. This script includes the amount of XP players are required to collect before leveling up and gaining another upgrade. When that amount is reached, it prompts the upgrade controller to display the UI. LevelUp is attached to the player.

Upgrade Controller

The UpgradeController.cs script works with the event system to display the UI and select 3 upgrades for the player from a bucket system. Most of the variables should be left untouched, but if desired, the odds array displays the odds of obtaining each rarity of upgrade, and the setUpgrades() function sets up the upgrades array to include all the available upgrades to be selected from.
Once the LevelUp script tells the upgrade controller to trigger a level up, it will pause the game and display the UI, allowing the character to choose from 3 different options. First, it selects a rarity based on the odds mentioned before, then it picks an upgrade from that bucket from the array to be chosen. Once an upgrade has been selected, the upgrade manager marks the upgrade as selected (which only applies to some individual upgrades, see below) and applies the upgrade.

Game Constants

COMMON, UNCOMMON, RARE, and LEGENDARY are all integers that correspond to the row they are in in the upgrades array.
RARITY_STRINGS gives the string when indexed at the game constant of the desired rarity.
RARITY_COLORS gives the color of the text displayed in the UI when indexed at the game constant of the desired rarity.

Individual Upgrades

Assets/Scripts/Upgrades/IndividualUpgrades
Each upgrade is based on the polymorphic class Upgrade.cs. It contains a title, description, rarity, whether it’s reusable, and an array of its valid player characters. When adding a new upgrade, follow the following naming scheme:
{UpgradeName}_U{_CharacterLetters} where {UpgradeName} is the name of the upgrade, and {_CharacterLetters} is an optional string of letters corresponding to which characters it works on. A for archer, W for warrior, and M for mage (or a combination).
If an upgrade can not be selected multiple times, set reusable to false, and if an upgrade can only be used by certain characters, override validCharacters to be the list of valid characters, “Archer”, “Warrior” and “Mage”.

Sound

Assets/Scripts/Sounds
Assets/Sounds (for storing actual .wav files)

Audio Manager

A single AudioManager GameObject controls all sounds within the game that persists with DontDestroyOnLoad() across scenes. It acts as the hub for playing background music and sound effects, ensuring everything runs smoothly. The AudioManager.cs file contains a number of functions for dealing with starting, stopping, pausing, resuming, and setting volume of various sounds, most of which are self-explanatory.

Music Triggers

The AudioManager (hooked to SceneManager.sceneLoaded) listens for scene changes and starts the right music automatically using UpdateBGMForScene(). For sound effects, each effect is linked to a specific moment, whether it be a button press, enemy hit, or ability used, and plays instantly using a shared function. Sounds are added via the Unity inspector by dragging audio files (e.g., example.wav) into slots on the right scripts or objects, allowing for convenient customization.

Background Music (BGM)

Lobby Music: In the Limbo scene that acts as the lobby of the game, a calm lobbyTrack plays at a relatively low (0.08) default volume through lobbyBGMSource.
Gameplay Music: Scenes where the player is fighting enemies (AKA: levels or stages) such as Lust and Gluttony use a more cinematic gameplayTrack at a volume of 0.1, played on a separate channel to avoid potential music overlap.
Boss Music: When a boss fight is triggered and the boss enemy spawns in (is_boss = true), the gameplay music dynamically switches to a more upbeat and dramatic boss song. StartBossMusic() swaps gameplayBgmSource.clip to bossTrack which is a modified version of the gameplayTrack. It switches back to gameplayTrack via StopBossMusic() when the boss is defeated (or to lobbyTrack if the player fails and returns to Limbo).
Volume Control: Through the options menu, the player can adjust music volume with two sliders that are handled by SetGameplayMusicVolume() and SetLobbyMusicVolume() respectively: one for the lobbyTrack and one for the gameplayTrack (that also modifies the boss music volume).
Pausing: When the options menu is opened by the player (with the *esc* key), whatever BGM track is presently playing will be paused using the PauseBackgroundMusic() function until the player either exits the options menu (triggering ResumeBackgroundMusic()) or selects one of the options from within the menu.

Sound Effects

Sound effects utilize a unique AudioSource: effectSource, which focuses on one-shot playback via the PlaySoundEffect() function. Every sound is equalized to match its relevance in the scene. The following is a non-exhaustive list of sound effects currently implemented.

Button Clicks: In StartMenu and CharacterSelect, the corresponding start button and three select character buttons all trigger a sound via the ButtonSound script. The UpgradeButtonSound script handles the same effect but for selecting one of three upgrades inside a given level.
Character Sounds: Player actions trigger sounds through character-specific prefabs, each using serialized AudioClip fields in the inspector allowing unique sounds to be assigned and played through PlaySoundEffect(). For example, the mage's ChainLightning attack is assigned in the inspector and triggers whenever the ability is used. Each character has three main audio settings: a dodge sound, damage sound, death sound, and level up sound. The warrior character has a unique attack sound since there is no projectile attached. Projectiles follow a unique ArrowSound script attached to the corresponding Arrow prefab (that all projectiles stem from) which has an attached sound that only plays when 'is Player Arrow' is true. The level up sound stems from the LevelUp script and the sound triggers when the XP threshold is reached.
EnemySounds: Enemy interactions play sounds via the Enemy script. A death sound triggers in the Die() coroutine while other potential effects rely on other events (player damage sound).
Environmental Sounds: Using a Portal plays a sound in OnTriggerEnter2D() when the player enters. The GameWin script plays a sound in OnGameWin() when a level is completed (the boss is slain) tied to the win event.

Map Creation

All maps are made of different layered textures. The "floors" that the player walks across are made of 2D tilemaps, which are grids that can be assigned repeating textures to mimic the ground. The textures used were found on the Unity asset store and mesh together based on what other textures are surrounding them, these behaviors make the textures appear to be more dynamic, this can be seen in the center of the lust map, where the logic of certain textures were purposely changed in the middle of the map to create a broken walkway across the water.

For all maps, these tilemaps were layered on top of each other to provide further depth and contrast, in Limbo, there are actually two floor grids, one of which to provide the pre-programmed meshing that makes the floor more "interesting" and a flat plane beneath it to connect the other tilemap neatly with the walls. This layering can also be used to fix certain camera issues with the tiles. In Lust, there was a bug where in certain locations, the player could see in between the tilemap's textures briefly, which looked messy. This was fixed by layering an additional tilemap beneath the displayed one, but slightly offsetting it to cover these slits.

Walls

The walls used in Pixhell were also created using tilemaps. There are four grid prefabs (presets), each with a repeatable texture and basic hitbox. Both the provided texture and hitbox would be incomplete for a final wall, so they act as placeholders for the right proportions. The wall textures were also found on the unity asset store, though as part of a different texture set, so there was some minor color changes to help the walls seem to better fit the floors they were matched with. The walls utilize a rigid body and collision box to determine where they will become interactable. This initially created physics bugs with the arrows, but was later corrected so that the walls also acted as a physical barrier.

Layering

The final step of each map is figuring out the proper perspective. Since Pixhell is a top-down game, the appearance of each overlaid tilemap needed to be adjusted accordingly. The walls in the foreground were given the highest layer value (meaning that they appeared on top of the other tilemaps) so that they seemed to cover the floor and player behind them, and the opposite was done for the upper walls. Besides this, each map's floors had to have a similar value provided so that each tilemap would provide the desired features, in Lust, the water elements of the map were created in a rectangular grid that was valued beneath the stone floor grid so it looks like the water is beneath the stone floors. This was similarly executed in both Gluttony and Limbo to achieve the desired effect.