Matthew S. Boudreaux

C00269064

CMPS 490: Senior Project

5 May 2022

CMPS 490 Senior Project

Fantasy Action Game, Working Title “It's Safer In Here”

Section 1: General Description

I have always been fascinated by the process of video game development. The potential of video games as a creative and artistic medium has been discussed widely recently, as gaming has entered the mainstream through internet-mass media outlets such as Twitch and YouTube. Having been invested in video games since a very young age, designing video games as a creative outlet has always appealed to me, and for my senior project, my goal was to design and implement a video game drawing inspiration from several popular modern video game properties, as well as several older franchises that informed, tonally or mechanically, modern game design philosophies.

The technical goals of the project are as follows: Develop a dark fantasy, first-person action-adventure video game drawing inspiration from fantasy action games such as the Dark Souls franchise, as well as retro video games such as From Software's earlier King's Field series, as well as PC games from the late 90's. Create systems that allow for first-person melee combat, inventory management, and environmental interaction. Create assets using computer graphics software such as Blender and Adobe Photoshop.

Additionally, several of the project's goals were creative rather than technical: learning to use Blender for 3D-modeling and animation, creating concept art for environments and characters, and creating background for the game's story and setting could be uncovered through and enhance game play.

Ultimately, however, each of these goals was intended to create learning experiences in developing workflows for various elements of the project that would be used to continue development beyond the scope of CMPS 490. Ensuring that I had goals to focus on helped ensure that I was able to continue developing skills such as improving programming, becoming comfortable with 3D modeling software, and developing the art and writing elements of game projects.

Section 2: Proposed Solutions

For the project, I would be working in the Unity game engine, version 2020.3.17f1, using C# as my choice of programming languages: its similarities to Java, which is taught in several CMPS courses, made it easy to adapt to and use for more complex programming. For 3D modeling and animation, Blender was used due to its free and open-source accessibility and breadth of learning material available on the internet. Adobe Photoshop would be used for concept art and assets, but ultimately these were not a major part of the project's implementation.

The following section will outline the creative draft for the project: while goals such as unique art and concepts were not met, the foundation is set for them to be implemented post-semester.

-The game (working title “It's Safer In Here”)is a first-person action-adventure game set in a mysterious, winding dungeon beneath a doomed world beset by a mysterious and all-encompassing evil. With the outside world so hostile and unliveable, the unnamed player character is forced to enter through a mysterious stone door deep in a haunted forest, reasoning that however bad it must be in there, it's still safer than outside. Prevalent themes of isolation and the effects of stress on mental health inform cautious, methodical gameplay including fantasy melee combat with an emphasis on using a weapon and shield to block damage, and retaliating during brief safe moments.

-A unique health system gives the player both a “health” meter and a “stress” meter: supernatural attacks or circumstances will cause the player's Stress to increase, which is subtracted from the player's maximum health. If the player's health reaches 0, or their Stress reaches their

maximum Health, the level is lost and restarted.

-The game is laid out into distinct, sequential levels where the player explores detailed, nonlinear environments populated with enemies, traps, and secrets. Resources such as healing items are scarce, causing each new encounter to feel dangerous and important.

-Aesthetically, the game aims to emulate the graphical styles of early 3D games such as Quake, The Elder Scrolls III: Morrowind, King's Field, and other games from the late 90's and early 2000's. Heavy, dark ambience is used to create a tense, dread-filled mood that enhances the unknown dangers of the dungeon.

Section 3: CMPS Course References

The following section outlines the courses offered by the university that were helpful in implementing the project.

-Implementing algorithms for game systems was a large part of development, but CMPS 340's emphasis on algorithm performance on large data sizes was not an important part of development.

-Several data structures are used to manage player inventory and statistics: Unity's ScriptableObject framework allowed for creation of instances of a data structure as discrete game assets. Algorithms for creating, manipulating, and interacting with these data structures were a large part of the project's development time.

-General programming knowledge was used extensively for implementation, as well as general structuring of the game's software systems. In particular, CMPS 455 emphasizing writing code for distinct instances of objects aided in development for several pieces of code designed to be modular for re-use multiple times over game play: in particular, enemy behavior logic, environmental interactions, and inventory management are each designed to be replicable and apply to many instances of each.

-Fundamental computing and software design courses provided experience required to both design and implement code solutions.

-CMPS 327 and 427, Intro To and Video Game Development use Unity as a framework to teach game design ideas and implementation of algorithms. These courses also introduced and reinforced the workflow of designing separate algorithms and data structures that are able to interact, through Unity's MonoBehavior framework, as discrete instances at runtime. This framework is the main technique used to implement runtime algorithms for all game objects.

-While no longer listed as a required Video Game Development course, taking a visual arts class, in particular VIAR 235 Art and the Computer, during the development of the project offered a secondary space to experiment with creative elements such pas animation and visual design that were essential to becoming comfortable with the tools used for asset creation, in particular Blender, and Unity's built-in animation suite.

Section 4: Details of Implementation

The project was developed in several stages, each of which will be discussed in detail below.

1: Character Controller

The player character was the first element created, and underwent gradual development over the whole of the project. The Player prefab contains all components related to player control, including first-person animations, movement, camera movement with the mouse, and an instance of PlayerStats. The monobehavior class PlayerStats uses a reference to an InventoryObject, which is saved as an asset and discussed later, to modify player stats, animations, and other values based on those found in currently equipped ItemObjects. PlayerStats also contains references to UI objects, and public functions for adding or reducing stats, equipping items, and using consumable items.

During gameplay, the player prefab uses the FPSController script to handle player input and responses to environment objects and enemies. The FPSController is modeled as a pair of state machines that respond to player input to perform actions: State and ActionState. State is responsible for determining which types of input are available to the player at a given time, The State states are: FreeMovement, which allows full action and allows ActionState to read for player input. Talking,

entered from FreeMovement by interacting with an object of type DialogueObject (discussed in Section 2), which locks the player's movement in place and allows a dialogue object to display text, for reading or having a conversation with an NPC. State Menu occurs in FreeMovement upon pressing Tab, and freezes all character movement and opens the inventory menu, which displays tabs for each array in the InventoryObject.

ActionState checking occurs during the FreeMovement character state. The states for ActionState are: Null, Attack, Block, Parry, and UseItem. After any state, the player always returns to the Null state, meaning the character is taking no actions. The Attack state causes the player's current attack animation to play, and starts a coroutine preventing input of other actions until the duration of the Attack animation has passed. Parry and Block occur in sequence after the player inputs and holds the block command (right mouse button by default): a Parry animation plays, raising the character's shield, based on the duration held in the currently equipped ShieldObject. After the parry animation finishes, the player transitions to the Block state, playing an idle loop of the player's shield being held up. Either of these states can be interrupted by releasing the right mouse button.

The player contains the TakeDamage function, which uses the player prefab's collider as a hitbox and occurs when the player enters a trigger tagged “eHitbox”, representing any attacks being performed by enemies or environmental hazards. The function takes the EnemyHitbox component of the collider's GameObject to see what type of and how much damage is being received. The damage is reduced by a percentage based on the defense values found in PlayerStats, and further reduced by the currently equipped shield's BlockRating if the player is in the Block state. If the player is in the Parry state, however, the TakeDamage function is never called in OnTriggerEnter, a parry animation is played, and a message is sent to the enemy's EnemyController to play their animation for being parried.

Lastly, the FPS controller handles raycasting and interacting with environmental objects. Each frame, the LookForward function casts a ray forward from the camera and sets the current Interactable item to the collider hit by the raycast. Based on the current Interactable game object, inputting the E

key while in the FreeAction state will trigger interactions from talking to a dialogue object, to interacting with a door or pushing a button, to picking up an item.

The Player prefab also uses a PlayerAnimator class to interact with an Animation Controller, Unity's system for creating state machines that play animations based on triggers activated through scripting. The main function of the PlayerAnimator class is to house public functions that the FPSController can call to trigger animation states.

2: Environmental Interactions

The script InteractTrigger allows an object hit by the player's LookForward raycast to trigger a variety of effects from objects in the scene: it contains an array of GameObjects, and when interacted with, triggers the appropriate function on each object based on their tag: a door will open, or check the player's inventory for keys (handled in FPSController) and unlock if tagged to be a locked door, trigger traps to deal damage to the player, wake enemies up and set them on a patrol route, or cause text to pop up on the player's screen.

3: Inventory System

The inventory system was developed early in the project, as many other parts rely on the objects defined as parts of the inventory system. The inventory system uses Unity's ScriptableObjects framework to model items (class ItemObject), an item database containing all ItemObjects, and individual instances of the player inventory. Container classes Item and InventorySlot abstract the ScriptableObject itemObjects into an integer ID, which is referenced in the Item Database, which automatically sets the ID of ItemObjects to their index in the database. The monobehavior class PlayerStats, used in the Player prefab uses a reference to an InventoryObject, which is saved as an asset, to modify player stats, animations, and other values based on those found in currently equipped ItemObjects. PlayerStats also contains references to UI objects.

There are five subclasses of Item Object, each of which contains unique qualities that define how they affect the player when equipped or used, a name and description to be displayed in the

Inventory Display, and the prefab for a model to be used when the item is referenced. Weapons contain a damage value, an attack speed, an Animation Override Controller to replace the player's attack animation, and a flag for being two-handed, causing the player's shield to be unequipped when the weapon is equipped. The Shield object contains a block value and parry speed, which modifies how quickly the player's Parry animation finishes. The Armor and Accessory objects are currently unimplemented, but will contain a defense value for a percentage of damage they protect the player from and unique effects, respectively. Finally, the Consumable item class represents potions, keys, and other miscellaneous items that can be used by the player: they contain either values for healing, or are checked for by external scripts for the sake of using keys or similar interactions.

4: User Interface/Menu System

UI objects, a part of the Player prefab, handle displaying player statistics and opening/closing the inventory menu to allow for equipping and de-equipping of items.

The UI always displays a health bar, which is updated each frame by PlayerStats to represent the player's current health and stress values.

The main component of the UI, created alongside the inventory system, is the Inventory Display function and prefab. InventoryDisplay contains definitions for a template inventory button, a list of sub-menus for each inventory array, and a dictionary of all currently displayed items. It also contains the UpdateDisplay function which is called each frame, and is responsible for displaying inventory items in the InventoryObject UI object. If an object does not currently have a button, a new one will be created from the ButtonListButton prefab, and if a new inventory is selected to be displayed, all buttons of that inventory's type will be displayed instead. An item button can be clicked to display information in a middle panel, and the Equip button allows the button's inventory item to be passed to the PlayerStats script to replace the currently equipped item of that type.

The inventory menu system is currently one of the less developed parts of the game: several functions are vestigial, including those to remove inventory objects from the inventory, and UI graphics

are not completed, and currently use Unity's default sprites. Additionally, the right third of the inventory display is intended to display a paper doll showing the player's current stats and equipped items, but this is not yet implemented.

5: 3D Assets

3D assets are all created in Blender: a large portion of this project's motivation was improving my ability to create assets for a game under a time limit. A low-poly and low-resolution art style allowed assets to be modeled and textured quickly. One of the larger challenges of the project, however, was underestimating how long UV-unwrapping and texturing models can take: for a low-resolution texture to align properly on a model, the process of UV-unwrapping to map the model's faces to areas of a 2D texture must allow each face of each model to be rectangular, to avoid pixels being stretched unnaturally.

The most complex asset created for the project, a model for the first implemented enemy NPC, a skeleton, required additional work creating a rig and implementing inverse kinematics (IK) to allow for procedural calculation of limb positions for easy animation.

6: Enemy Behavior

Currently, behavior is implemented for only one type of enemy NPC, a melee enemy who chases the player and attacks them. Enemy AI uses three scripts similar to the player controller: EnemyController, EnemyStats, and EnemyAnimator, which serve as a template for a variety of enemy prefabs to be created from. EnemyStats contains current and maximum health values, damage values for up to two attacks, a detection range, walk and run speeds, and a “chase distance” at which the enemy will attempt to attack the player. Altering these values allows creation of different types of enemy NPC's, including enemies who will not approach the player at all(by setting “chase distance” arbitrarily high”). Finally, Unity's built in navigation system is used for navigation: a NavMeshAgent is attached to each enemy NPC to allow them to traverse environments to reach waypoints (typically set to the player's location through a PlayerManager script that simply passes them the player's

GameObject's position. Finally, the EnemyController uses the LookForPlayer function, which casts a ray from the enemy's position to the player's and returns true if the first object it hits is the player, to govern whether the enemy is within the player's line of sight.

The Enemy Controller script governs enemy action through a similar state machine to the player character, with states to handle behavior for a variety of conditions. Enemies can be instantiated in three default states:

-Asleep, where the enemy will not become active until attacked or triggered by a TriggerScript

-Ambush, where the enemy will remain inactive until LookForPlayer returns true

-Patrol, where the enemy will attempt to traverse between the points in an array of Position values, until LookForPlayer returns true.

When the enemy is activated, it enters the Active state, where it remains until it either enters its minimum range of the player or loses sight of them. When within range, the enemy will begin an attack using similar logic to the player's attack animation. Hit detection is performed with a collider that is activated and deactivated as part of the attack animation: this allows extra attack animations to be implemented quickly.

In the current state of the project, when an enemy loses sight of the player, it will. simply begin walking its point of origin, and reenter either the Ambush or Patrol states.

The enemy also has a TakeDamage function that interrupts all other animations and actions, called when the enemy takes damage from the player, calling the EnemyStats' Take Damage function. If the EnemyStats' health is reduced to zero, the enemy plays a death animation and is de-instantiated.

Section 5: Evaluation

Although the project is not in a finished state, much of the work done over the semester has laid an important groundwork to continue working after the semester. New items and enemies can be easily implemented, level geometry can be quickly imported, and more creative aspects from the game's framework can be integrated.

However, there are still many tasks that remain: first, there is currently no way to save and load a player inventory between scenes, which would allow work on level transitions and other essential level logic: this will be important to implement for a full release. Several item types, namely Armor and Accessory, are not implemented, and special abilities from player equipment is not yet implemented.

Additionally, there are many bugs that need fixing and assets that need polishing: the inventory screen is currently very minimalist, only using Unity's default graphics, and several inventory functions do not work: vestigial Unequip and Drop functions in the InventoryDisplay script would allow more fine-tuned inventory management, but currently cause the InventoryDisplay to display buttons for items that are not in the player's inventory. The desired “paper doll” feature displaying currently equipped items and stats needs implementing. 2D art needs to be created for UI buttons, backgrounds, text boxes, inventory objects, and the health/stress bar.

Finally, while a proof of concept test level is finished, it is not polished, and is currently the only level. More assets for level geometry, enemies, environment clutter, items, interactable level objects such as doors and switches, and additional sprites such as lighting effects will be a large part of implementing further levels. Sound is also an as-yet unfinished aspect of the game: while there is a simple AudioManager script to allow game objects to trigger audio clips to play using Unity's default audio player/listener systems, it currently does not use polished sounds, nor is there any variability in playback, looping audio, or ambient sound.

Section 6: Lessons Learned

This project provided many learning opportunities for working with new software methodologies, programs and new artistic disciplines. In terms of programming, the main new experience was with ScriptableObjects, with this being my first time working with them, and ensuring that each has an associated non-ScriptableObject class will allow me to develop systems for saving and loading in the future. Designing several complex classes to interact dynamically with room to expand was a useful and interesting challenge in software design. Open-ended systems such as I have developed will create many opportunities in the future to expand them for the purposes of this, or other future projects, and getting experience programming such systems will make it faster to develop and program them.

This project was also a helpful exercise in anticipating scope creep and managing expectations from an original draft within a time limit. Developing necessary systems for gameplay and creating assets took most of the development time, leaving less for implementing many desired features such as finished UI, interesting level design, world building, and enemy/encounter design. With the necessary systems nearing completion, however, there is now more time to plan progression through an entire game.

Perhaps the most surprising lesson was how much enjoyable asset creation and animation were to learn. While modeling and texturing clutter elements and levels was important to learn, creating animations for the player character and enemy. Although the most daunting step, having no prior animation experience, it was the most rewarding to learn, and I am excited to use this skill to create more animation assets for this and future projects.

Section 7: Summary/Conclusion

Overall, while I was unable to complete as much of the project as I expected, the main goals of the project were setting goals and working towards them, creating learning experiences for programming and art, and motivating myself to work on a project long-term. As far as these go, I feel that the project was successful, and I now have the framework to continue to work on the project in the future, as a personal passion project, or even to potentially develop a full digital release on a platform such as Steam or itch.io.

Section 8: References

"Unity User Manual 2021.3 (LTS)." *Unity*, Unity Technologies, https://docs.unity3d.com/Manual/index.html.

"Blender 3.2 Reference Manual." *Blender 3.1 Reference Manual – Blender Manual.* Blender Documentation Team. 2 May 2022. https://docs.blender.org/manual/en/latest/index.html

Abbitt, Grant. "Rigging a Low Poly Person | Blender 2.8 | Beginners - YouTube." *YouTube*, 14 Nov. 2019, https://www.youtube.com/watch?v=srpOeu9UUBU.

"UNITY3D - Scriptable Object Inventory System | Part 1 - YouTube." *YouTube*, Coding With Unity, 6 Sept. 2019, https://www.youtube.com/watch?v=_IqTeruf3-s.

"Unity UI - Scroll Menu Pt 1: Intro - Fundamentals behind a Scroll Menu / List." *YouTube*, c00pala, 15 May 2017, https://www.youtube.com/user/c00pala/videos.