Andrew Spate, Nicholas Spudich, Kevin Andor, Nathaniel DeHart

[Email address]

Vulcan’s Spire

Design Document

# **Instructor Comments/Evaluation**

***Table of Contents***

[**Instructor Comments/Evaluation** 1](#_Toc89542126)

[**Abstract** 4](#_Toc89542127)

[**Description of the Document** 4](#_Toc89542128)

[Purpose and Use (of the document) 4](#_Toc89542129)

[Ties to the Specification document 5](#_Toc89542130)

[Intended Audience 5](#_Toc89542131)

[**Project Block Diagram with description** 6](#_Toc89542132)

[Description of Figure 1 6](#_Toc89542133)

[**Design Details** 7](#_Toc89542134)

[System Modules and responsibilities 7](#_Toc89542135)

[*Architectural Diagram* 7](#_Toc89542136)

[*Module Cohesion (Modules seen in Figure 2)* 8](#_Toc89542137)

[*Module Coupling (Modules seen in Figure 2)* 10](#_Toc89542138)

[Design Analysis 12](#_Toc89542139)

[*Data Flow Diagram* 12](#_Toc89542140)

[*Description* 13](#_Toc89542141)

[Design Organization (Object Oriented Design) 14](#_Toc89542142)

[*Detailed tabular description of Classes/Objects* 14](#_Toc89542143)

[*Functional description* 16](#_Toc89542144)

[Files accessed 25](#_Toc89542145)

[Real-Time Requirements 25](#_Toc89542146)

[Messages 26](#_Toc89542147)

[Narrative / PDL 26](#_Toc89542148)

[Decision: Programming language/Reuse/Portability 27](#_Toc89542149)

[**Implementation Timeline** 28](#_Toc89542150)

[**Design Testing** 28](#_Toc89542151)

[**Appendix: Team Details** 29](#_Toc89542152)

[**Appendix: Writing Center report** 30](#_Toc89542153)

[**Appendix: Workflow Authentication** 31](#_Toc89542154)

[**References** 32](#_Toc89542155)

# 

# **Abstract**

Vulcan’s Spire is a non-traditional game that will provide users a fun introduction to the mythological figure of Vulcan while inside the most iconic locations of the California University of PA campus. The driving focus of this game is to create an interactive way to attract potential students into the Computer Science Program while developing a captivating narrative about the University’s beloved mascot Vulcan as he completes challenges and wards off antagonists through his quest to find himself and save the beloved clocks at old main. The game will be targeted towards prospective and current students; however, it will be available to anyone who wants to play it. This document will outline the development side designs and requirements of Vulcan’s Spire. The client and developers will utilize this document to gain insight on agreed practices for the execution of the project. This document outlines the modules and interactions between those modules including programming details that will be vital to the development of Vulcan’s Spire.

# **Description of the Document**

## Purpose and Use (of the document)

The purpose of this document is to give the software development team guidance and structure to the architecture of Vulcan’s Spire game project. The document also aims to detail the programming language and functions needed to make the game work and prosper. It will be used by the development team as a guideline for keeping the project on track without going into scope creep or losing focus on the structural pillars of Vulcan’s Spire. This document will also outline the anticipated methods that will be used in the implementation into code.

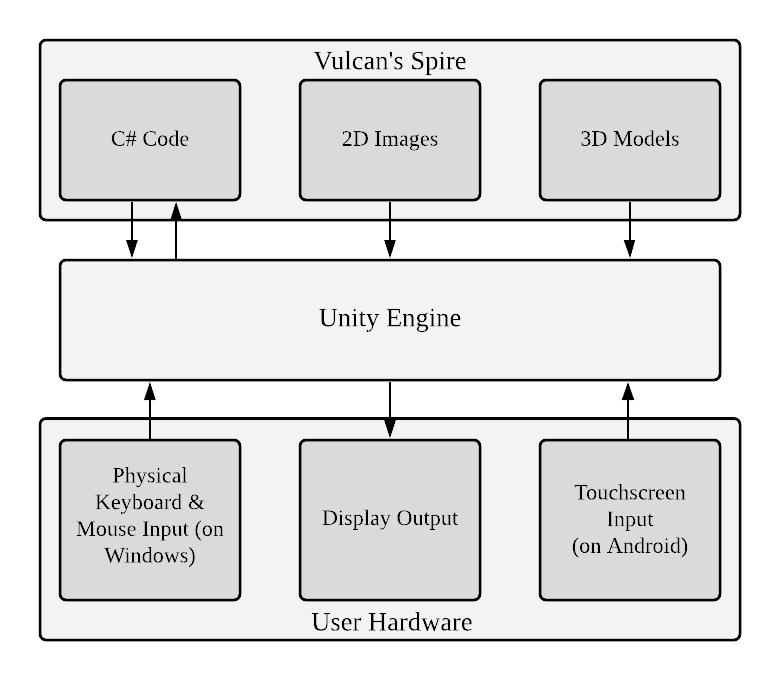
## Ties to the Specification document

This design document extends the agreements and outline presented by the specification document and will add components relevant to the development side. This document will explain the uses of our specific programming language used and explain the architecture that will be implemented into the project. In addition to its ties to the specification document, this document will build off of the Requirements Documents as CASE examples are implemented and elaborated upon.

## Intended Audience

This document’s intended audience is the software development team for Vulcan’s Spire. This document will contain diction and verbiage which is technical in nature that aims itself to a high-level document. This document may be understood by the client, if they are obtaining an auditory presence over the project by a 3rd party consulting group, however much of the document may not make sense or have great impact on the client’s understanding of the project. Vulcan’s Spire’s software development team should use this document to better understand the organization and architecture of the game. Its contents will include the game’s module, data flow, variables, functionality, architecture, and system requirements.

# **Project Block Diagram with description**



*Figure 1*

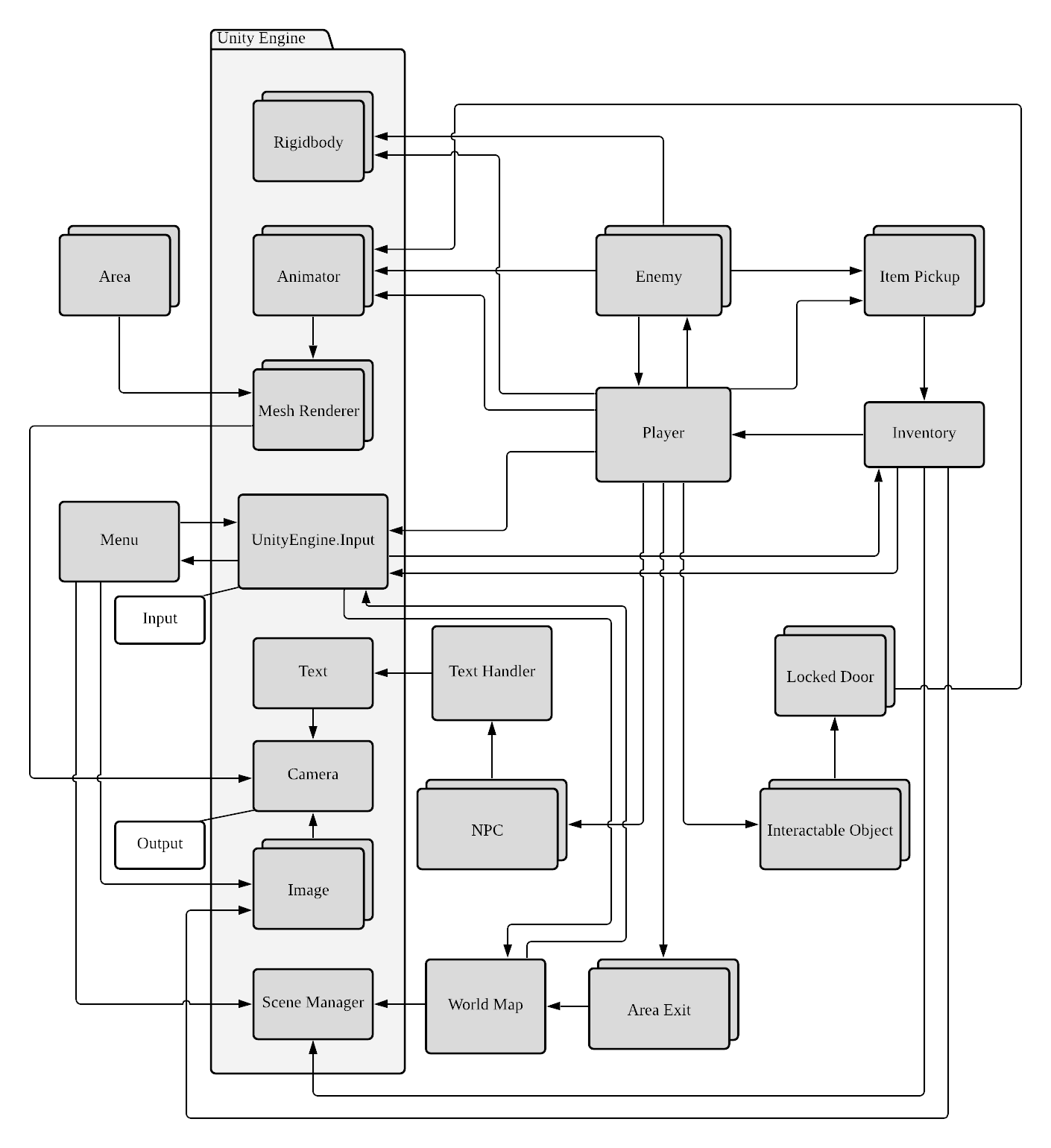
## Description of Figure 1:

This project can be broken into the following three layers: Vulcan’s Spire, Unity Engine, and User Hardware (Figure 1). The Vulcan’s Spire layer will use the Unity Engine layer as an intermediary layer for checking various user inputs, that way the C# code will not have to interface with device drivers in any direct way (Game Programming Patterns in Unity with C#, 2021). The user inputs can take the form of physical keyboard and mouse inputs on devices running Windows or touchscreen inputs on devices running Android. In terms of output, the C# code will make use of the many visual-based components of the Unity Engine layer to render 3D models and 2D images to the user’s display (Games, n.d.).

# **Design Details**

## System Modules and responsibilities

### *Architectural Diagram*



*Figure 2*

### *Module Cohesion (Modules seen in Figure 2)*

* Player:

Contains all the methods needed for controlling a virtual character in the game world. This includes, but is not limited to, methods for movement, attacking enemies, setting equipped items, receiving character damage, and “dying.”

* Enemy:

Contains all the methods needed for controlling a virtual enemy character in the game world. This includes, but is not limited to, methods for movement, attacking, receiving damage, dying, and dropping (instantiating) “items.” Artificial Intelligence will be a part of this module and will control all methods involving movement and attacking.

* Inventory:

Contains methods for handling all things related to the inventory menu, like a way to manage the user’s picked-up in-game items and a way to save & quit the game.

* Menu:

Contains all methods involved in starting the game. Methods will include loading a save game, starting the game over, and closing the program. The Menu module will also contain the graphics needed for the user to interface.

* Item Pickup:

Contains all necessary item data like the item’s name and item statistics. This information will then be passed to the playable character’s inventory.

* Area:

Contains all pieces of visual world data, namely in the form of 3D models.

* NPC:

Contains data from the *Enemy* module allowing for movement and attacking as well as text data which will be used for in-game dialogue.

* Text Handler:

Contains methods for passing text data character by character so that it may printed from one source.

* Interactable Object:

This abstract module will contain a state for discerning whether it is actuated or de-actuated. This will act as a container for in-game physical puzzles that may be interacted with by the user.

* Locked Door:

Contains data pertaining to visually opening/closing a virtual 3D door and controlling a virtual collider to allow for other objects to move through this virtual door.

* Area Exit:

Contains all needed scene transitions within buildings in the game world. These graphics will be loaded via the Area Methods.

* World Map:

Contains everything that encompasses the game’s 2D representation of the enterable buildings of the Cal U campus.

### *Module Coupling (Modules seen in Figure 2)*

* Player:

Interfaces with Unity’s *Rigidbody* module for handling character movement, any instance of the *Enemy* module for attacking, Unity’s *Animator* module for animating the user’s character, Unity’s *UnityEngine.Input* module for polling for user inputs, any instance of the *NPC* module for interacting with NPCs, any instance of the *Area Exit* module for triggering the loading of different areas, any instance of the *Interactable Object* for moving objects with that component around the virtual world, and any instance of the *Item* *Pickup* module to call a “pickup” function(Games, n.d.)(Unity Game Development Tutorials to Hone your Skills, 2020).

* Enemy:

Interfaces with Unity’s *Rigidbody* module for handling enemy character movement, the *Player* module for detecting the player character’s position, and attacking. Then Enemy module also interfaces with Unity’s *Animator* module for animating the enemy’s character and the *Item Pickup* module for dropping (instantiating) items.

* Inventory:

Interfaces with the *Player* module to “equip” different items on the player character therefore changing their stats (such as defense and attack power), Unity’s *Input* module for assigning menu UI-related inputs, instances of Unity’s *Image* module for setting UI images for menu items, and Unity’s *Scene Manager* module for “quitting” (unloading the currently loaded scene and loading the main menu’s scene).

* Menu:

Interfaces with Unity’s *Input* module for assigning menu UI-related inputs, instances of Unity’s *Image* module for setting UI images for menu items, and Unity’s *Scene Manager* module for quitting.

* Item Pickup:

Interfaces with the *Inventory* module for adding new inventory items representing themselves to the menu.

* Area:

Interfaces with instances of Unity’s *Mesh Renderer* module. Each *Mesh Renderer* contained will represent the virtual 3D world.

* NPC:

Interfaces with the *Text Handler* module to display text in a more formatted manner.

* Text Handler:

Interfaces with an instance of Unity’s *Text* module to display text on the user’s display (Games, n.d.).

* Interactable Object:

Interfaces with instances of the *Locked Door* module in order to unlock doors when a “solved” state is reached.

* Locked Door:

Interfaces with an instance of Unity’s *Animator* module in order to play visual door open/close animations.

* Area Exit:

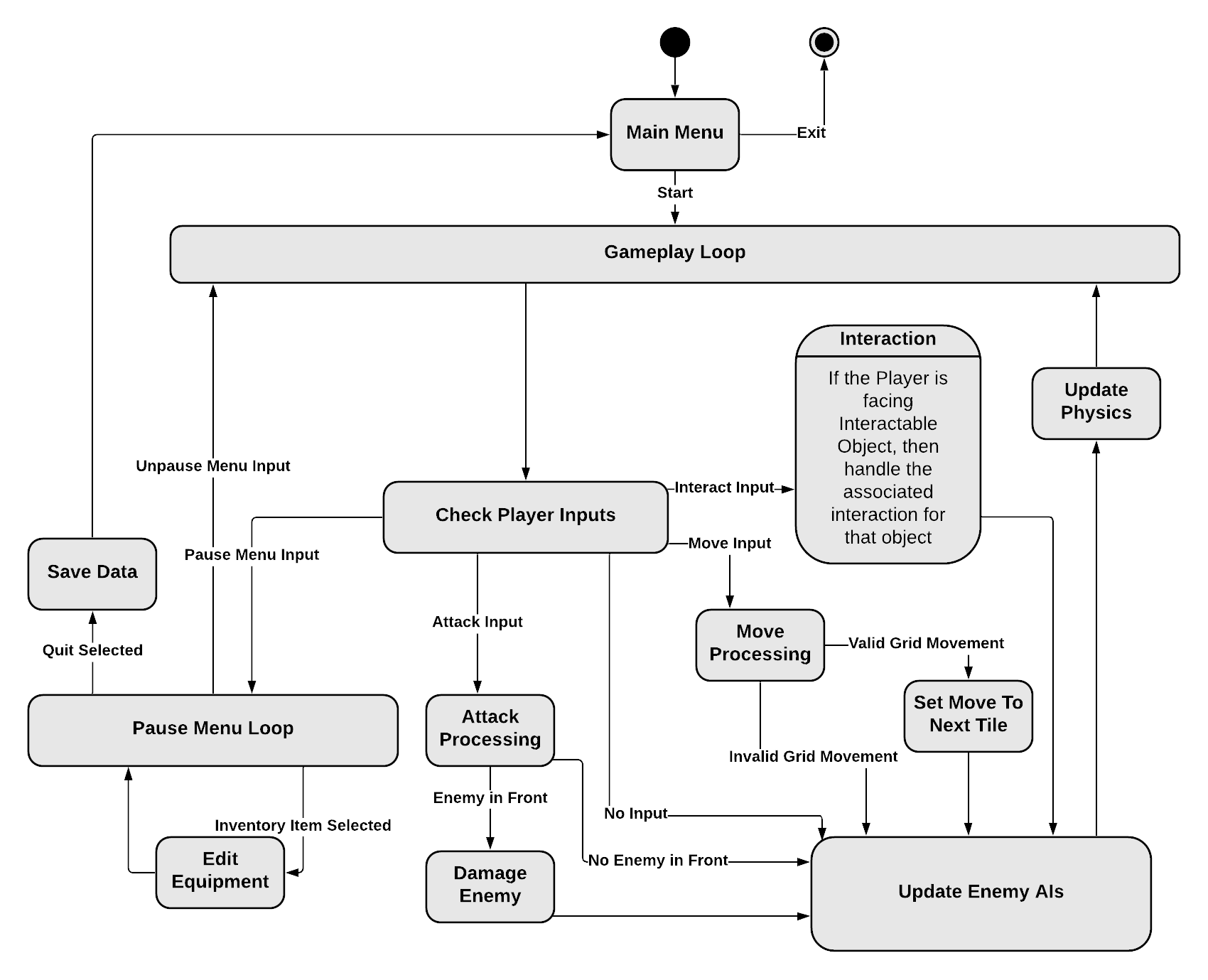
Interfaces with the *World Map* module to trigger the loading of the game’s Cal U campus map.

* World Map:

Interfaces with Unity’s *Scene Manager* module to unload the current scene and load up the Unity scene that contains the Cal U campus map and Unity’s *UnityEngine.Input* module for setting up UI related inputs (Dolan, 2019).

## Design Analysis

### *Data Flow Diagram*



*Figure 3*

### *Description:*

The data flow of this project will not have a determined end. The user must provide an input that will cause the program to exit (as seen in Figure 3). At the starting process, *Main Menu*, the user will have the ability to either exit the program or to “start” the game. Starting the game will bring the data flow to the *Gameplay Loop*; from here, all main logical processes involved in the project will be updated at a regular time interval. This central process can be broken up into a few other processes. One of the first processes in the *Gameplay Loop* that will be run is *Check for Player Inputs* this will check for user inputs and try to perform the related actions. The processes that can be entered depending on player inputs are the following: *Pause Menu Loop*, *Attack Processing*, and *Move Processing*. If the user enters an input that leads to the *Attack Processing* process, a determination will be made between whether the attack hit something or not. If the user enters an input that leads to the *Move Processing* process, then a check will be performed involving whether there is an obstruction to the character’s path or not. If there is no obstruction, the user’s character will move in the direction they are facing by way of the *Set Move to Next Tile* process. If the user does not provide any input or provides any input other than a “pause input”, then the processing will continue next at the *Update Enemy AIs* process; *Update Enemy AIs* will involve letting every active instance of an enemy character make decisions that involve moving about the virtual game world on their own. After *Update Enemy AIs* comes the *Update Physics* process. The *Update Physics* process will handle the virtual position changes of all virtual characters that are currently “moving”. After the *Update Physics* process finishes, the *Gameplay Loop* process will be re-entered.

While in the *Check Player Inputs* process a “pause” input may have been detected from the user; this would cause the program control to move from the *Gameplay Loop* process to the *Pause Menu Loop*. The process *Pause Menu Loop* can be used to continue to check for user inputs related to the player’s inventory items. If an item is selected to be equipped/unequipped, then a separate process will be entered for handling the respective chosen option before going back to the *Pause Menu Loop*; this loop will only be exited, if the user chooses to “un-pause” (return control to the *Gameplay Loop* process) or chooses to quit the game. Quitting the game will enter a *Save Data* process for saving various player character-related data before going back to the *Main* *Menu Loop*.

## Design Organization (Object Oriented Design)

### *Detailed tabular description of Classes/Objects*

**Class name:** Player : Monobehavior (base class)

**Class description:** Controls/handles all aspects of the player game object.

**Class data members:** Status (strength, health, defense), xPos, yPos, zPos for determining player positioning, money or in-game currency tracker.

**Class member functions:** start(), update()

**Class name:** Camera

**Class description:** Controls the camera that will display the output that the player will see when they are playing the game.

**Class data members:** xPos, yPos, zPos... (x, y, z) x-coordinate, y-coordinate, and z-coordinate to determine where the camera is positioned.

**Class member functions:** moveCam(x,y,z), clamp(object, offset x, y, z),

**Class name:** NPC

**Class description:** Controls/handles all aspects of the enemy game object.

**Class data members:**  NPC name, prefab

**Class member functions:** start(), update()

**Class name:** Enemy

**Class description:** Handles the enemies that the player character fights throughout the game.

**Class data members:** Enemy name, prefab

**Class member functions:** start(), update()

**Class name:** Item

**Class description:** Contains list of items found throughout the game.

**Class data members:** Text itemName, prefab

**Class member functions:** findItem(), removeItem(), addItem()

**Class name:** Menu

**Class description:** Handles menu configuration and accessibility.

**Class data members:** Text newGame, Text loadGame, Text options, Text exitGame

**Class member functions:** loadGame(), createGame(), loadScene(), changeScene(), exit()

**Class name:** UI/HUD

**Class description:** Handles the user interface or heads-up display that the player sees as they are playing the game.

**Class data members:** Text display, Input key

**Class member functions:** GetButtonDown(), GetRawAxis(), GetButton(), GetTouch()

### *Functional description*

**Player Class : Monobehavior (base class)**

Start()

**Input**:

The start() function has no input.

**Output**:

The function initializes all variables that the player character will have upon the start of a new game or when the game is loaded.

**Return Parameters**:

No values are returned from this function.

**Types**:

Called once before the update() function.

Update()

**Input**:

The start() function has no input.

**Output**:

The function initializes all variables that the player character will have upon the start of a new game or when the game is loaded.

**Return Parameters**:

No values are returned from this function.

**Types**:

Called once before the update() function.

**Camera Class**

moveCam(float x, float y, float z)

**Input**:

Input of x-coordinate, y-coordinate, and z-coordinate of the current map that the player is on.

**Output**:

Will move the camera to the location specified by the above. Mostly used for eventing.

**Return Parameters**:

Does not return anything.

**Types**:

Float real values of the map coordinates using the x, y, and z axis.

clamp(Gameobject name, offset x, offset y, offset z)

**Input**:

Input of Gameobject and value of offseted x-coordinate, y-coordinate, and z-coordinate to follow that gameobject.

**Output**:

Will loosely or closely follow an object determined by the offset specified. Mostly for eventing purposes.

**Return Parameters**:

Returns coordinates of the game object and stores them within the camera class. Performed during the Update () function.

**Types**:

Float real values of the map coordinates using the x, y, and z axis. Gameobject that is being followed.

**NPC Class**

Start()

**Input**:

The start() function has no input.

**Output**:

The function initializes all variables that the player character will have upon the start of a new game or when the game is loaded.

**Return Parameters**:

No values are returned from this function.

**Types**:

Called once before the update() function.

Update()

**Input**:

The update() function has no input.

**Output**:

Makes changes or “updates” to the non-playable character.

**Return Parameters**:

No values are returned from this function.

**Types**:

Called once every frame of the game.

**Enemy Class**

Start()

**Input**:

The start() function has no input.

**Output**:

The function initializes all variables that the enemy will have upon the start of a new game or when the game is loaded.

**Return Parameters**:

No values are returned from this function.

**Types**:

Called once before the update() function.

Update()

**Input**:

The update() function has no input.

**Output**:

Performs cycled updates from enemy for each in-game frame

**Return Parameters**:

No values are returned from this function.

**Types**:

Called once every frame of the game.

**Item Class**

findItem()

**Input**: takes in an integer input and looks for it in the dynamic array list

**Output**: for finding items that are added to the player inventory

**Return Parameters**:

Returns item if found in inventory.

**Types**: int ItemNum

removeItem()

**Input**: input item integer value to be removed from item list

**Output**: removes item in player inventory

**Return Parameters**: no return

**Types**: int ItemNum

addItem()

**Input**: input item integer to be added to the item list

**Output**: adds item to player inventory

**Return Parameters**: no return

**Types**: int ItemNum

**Menu Class**

loadGame()

**Input**: no input

**Output**: loads the scene, where game saved slots are accessed.

**Return Parameters**: no return

**Types**: Int SceneValue

createGame()

**Input**: no input

**Output**: set-ups up the game upon start-up/new-game

**Return Parameters**: no return

**Types**: Int SceneValue

exit()

**Input**: no input

**Output**: For exiting the game after start-up

**Return Parameters**: no return

**Types**: Int SceneValue

**UI/HUD Class**

GetButtonDown()

**Input**: get integer button value from keyboard

**Output**: performs in-game action

**Return Parameters**: return hotkey action

**Types**: int keyValue

GetRawAxis()

**Input**: get integer button value from joystick/mouse

**Output**: performs in-game action (movement)

**Return Parameters**: return movement direction

**Types**: int keyValue

GetButton()

**Input**: get integer button value from controller

**Output**: performs in-game action

**Return Parameters**: return hotkey action

**Types**: int keyValue

GetTouch()

**Input**: get integer button value from touchpad

**Output**: performs in-game action

**Return Parameters**: return hotkey action

**Types**: int keyValue

## Files accessed

The game will store files as prefabs. Prefabs are a special type of component that allows fully configurable GameObjects to be saved in the project for reuse (Unity, 2021). Other files such as artwork will be used and source code will pull these files as required. Unity makes use of built-in or the ability to make scriptable objects where art can be directly linked. This ability will be used extensively as we believe it will create the smoothest outcomes within Vulcan’s Spire game.

## Real-Time Requirements

Vulcan’s Spire will need to be responsive and run at a reasonable framerate that the player can enjoy. A game that feels clunky or otherwise slow will leave the player frustrated, and with 3D graphics, this will have an adverse effect on customer satisfaction. Quick response times will be executed through mediums, such as: a controller, touchpad, or mouse and keyboard. Most of the devices have some small sources of delay so minimizing the response to the game itself will be important as external components may each respond differently (e.g. a controller or keyboard designed for gamming with specialty buttons will function with less delay than a general office keyboard). Another consideration that might have an impact on this requirement would be that of display. Display can be directly related to framerate and may cause other errors if it is not re-sizeable. That is, making sure the game follows a general 1920 x 1080 resolution as to avoid this issue entirely. This will not only make the game more marketable, but also allow it to effectively meet the real-time requirements.

## Messages:

|  |  |  |
| --- | --- | --- |
| **Message Type** | **Source/Destination** | **Data** |
| GetButton()/GetButtonDown() | Player <-> Object/NPC | Interaction/state change request between Player/Object/NPC |
| Input.Axis()/ GetRawAxis() | Player-> New Location | Sends new position to check for collision |
| GetTouch() | Player -->Action | Sends command from touchpad to game |
| Update() (I.e. swing()) | Player <--> Enemy | Player attacks or interacts with enemy or vice-versa. |

## Narrative / PDL:

#### Main Menu:

#### 1. New Game

#### 2. Load Game

*3. Options*

*4. Exit Game*

#### New Game:

#### 1. New Game

#### Choose character name & move on to step 2

#### 2. Load into game

#### Load correctly?

*yes? Move to step 3*

*no? Exit and reload game and/or send error report.*

*3. Play game*

## Decision: Programming language/Reuse/Portability

The project code will be written in C# programming language within the Unity game engine, as development environment has been designed extensively for gamers with numerous tools and extensive documentation. Most of the functions and classes that will be written for this game will be compatible with other game platforms as Unity is a cross-platform game engine (Unity, 2021).

# **Implementation Timeline**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Dec** | **Jan** | **Feb** | **Mar** | **Apr** | **May** |
| Learning the Platform | Development | Development | Development | Internal Play Testing and Refinement | External Play Testing, Refinement, and Presenting |

\*Please note this time table is subject to minor modifications as programming commences, and a few in our group are new to the Unity game designer, and a learning curve may occur. We are attempting to account for this as December will be utilized for group members to explore the platform.

# **Design Testing**

At regular intervals we will be rapidly prototyping throughout the game’s development. In order to test the game early on, we will be using placeholder assets until the final assets are complete. The team knows that testing during the creation phase will provide a more beneficial error checking than at the end, and time spent testing will be better spent earlier rather than later in the process. Therefore, each game mechanic and feature will be thoroughly tested as they’re implemented and fixed as needed either early on or later in development. When the game has most of the core features and mechanics implemented in some form, we will begin testing the game as a whole, fixing bugs and issues relating to game design our team is able to find. If the game is ahead of schedule, we may implement and test new features. These features will be assessed similarly to the expansion of an agile workflow project, by assessing a time vs impact qualitative assessment. As the game nears a complete state, we may seek out third party participants to playtest the game for bugs and game design issues and provide feedback on their experience.

# **Appendix: Team Details**

The workflow leader is Kevin Andor. The document was divided into sections for each team member to complete. Any member who had questions or concerns voiced them using Discord. Each member checked-in upon completion to give a summary of their contributions. This ensured a deeper and clearer understanding of the work to be performed. Upon completion of this document, each member read through and gave their “okay” for the document to be pushed forward to the writing center and submitted.

# **Appendix: Writing Center report**

Graphical user interface, text, application, email

Description automatically generated

# **Appendix: Workflow Authentication**

I, Nicholas Spudich, hereby confirm that I have contributed to the information outlined in this document.

Signature:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

I, Andrew Spate, hereby confirm that I have contributed to the information outlined in this document.

Signature:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

I, Kevin Andor, hereby confirm that I have contributed to the information outlined in this document.

Signature:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

I, Nathaniel DeHart, hereby confirm that I have contributed to the information outlined in this document.

Signature:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# **References**

Dolan, H. (2019, March 21). *The Ultimate Beginners Guide to Game* . Retrieved from freeCodeCamp: https://www.freecodecamp.org/news/the-ultimate-beginners-guide-to-game-development-in-unity-f9bfe972c2b5/

*Game Programming Patterns in Unity with C#*. (2021). Retrieved from Habrador: https://www.habrador.com/tutorials/programming-patterns/

Games, U. (n.d.). *Documentation*. Retrieved from Unity Game Development: https://docs.unity.com/

*Unity Game Development Tutorials to Hone your Skills*. (2020). Retrieved from Game Designing: https://www.gamedesigning.org/learn/unity/