Parchment

Design Document

Team 12

Jacob Brown
Raymond Xie
Xiaoyan “Larry” Li
Jack Locascio
Drew Hatfield
Josh Gerber

Table of Contents

Purpose

The video game industry is a rapidly growing industry with large appeal to millions of people. As technology has improved over the past few decades, the overhead required for developing video games has been slowly diminishing, allowing for the creation of an entire genre of “indie” games, or games that have been developed by people outside of a traditional game studio. Indie game success stories are not uncommon; for example, the game Stardew Valley has sold over 10 million copies and was entirely developed by a single person. Whatever the personal reason may be, the idea of developing your own game is attractive to many people, some of which have little to no programming experience. Getting started with game development is often seen as something you do after learning and becoming familiar with a common programming language like C++ or Java, but this level of experience should not be necessary.

The purpose of the Parchment Engine is to provide the users with an easy, lightweight platform to begin creating their own games, with or without prior programming experience. Entire games can be created using the built-in GUI, packaged and distributed in a single file, and run by anyone with the Parchment Game Virtual Machine. The engine can also be utilized and expanded upon by experienced developers using custom scripts that can modify and interact with the engine in ways that is not possible using the GUI, allowing for more advanced capabilities to those who want it. The engine aims to be as simple as possible, allowing anyone to easily begin creating their own game, while maintaining functionality for creative freedom for users who want to implement original ideas.

Functional Requirements

1. General Usage

As a user,

a. I would like to open and close Parchment natively on a Windows OS so that I can use the game engine.
b. I would like to change Parchment settings so that I can configure it to my preference.
c. I would like to view a Parchment splash screen so that I can see startup progress.
d. I would like to have my game assets loaded and rendered by Parchment so that I can focus on game development.
e. I would like to be able to view all assets associated with a game project so that I can manage them.
f. I would like to access a sprite editor view so that I can edit sprites.
g. I would like to map controls to in-game sprites so that I can control their movement.
h. I would like to access an item editor view so that I can edit items.
i. I would like to be able to define player interactions with objects so that I can implement interactivity.

j. I would like to access a map editor view so that I can edit maps.
k. I would like to add event listeners to keys so that the player can trigger game events.
l. I would like to maintain state related to players so that I can implement inventory or score systems.
m. I would like to enable source control so that I can manage different versions of my game. (time permitting)
n. I would like to view in-application tutorials or documentation so that I can learn the features of the engine

2. Game Management

As a user,

a. I would like to name/create a new game project so that I can start developing.
b. I would like to view my existing list of projects so that I can continue developing.
c. I would like to save a game project so that my work progress is kept.
d. I would like to delete an existing game project and associated files so that I can manage projects.
e. I would like to change the name of an existing game project so that I have flexibility with naming
f. I would like to initialize a game instance using Parchment so that I can test the game.
g. I would like to change the game objective so that I can make different games.
h. I would like to end the game if an objective is reached so that the game can be beat.
i. I would like to export my game project so that I can release the game.

3. Engine Appearance

As a user,

a. I would like to resize the Parchment window responsively so that I can use it on any size of monitor.
b. I would like to rearrange and resize docked views so that Parchment can better suit my workflow
c. I would like to change the UI appearance (including themes and fonts) so that I can meet my accessibility needs.

4. Sprite Editor

As a sprite editor,

a. I would like to scroll through or search all sprites I have made so that I can easily manage them.
b. I would like to edit existing sprites so that I can iterate on sprite design.
c. I would like to create simple animations so that my sprites have motion.
d. I would like to place and delete sprites so that I can create game scenes.
e. I would like to be able to undo and redo actions so that I can fix mistakes.

f. I would like to use basic pixel manipulation tools (pencil, fill, eraser, etc) so that I can edit sprites.
g. I would like to change the pixel color and use a color picker so that I can create colorful sprites.
h. I would like to use selection tools so that I can move, copy, and paste sprites in the editor.
i. I would like to import an image for a sprite so that I can bring in outside assets.
j. I would like to use multiple layers so that I can create more complex sprites. (time permitting)

5. Controls

As a user,

a. I would like to change settings for player controls so that I can have custom controls.
b. I would like to bind different mouse controls so that I can make mouse controlled games.
c. I would like to have control over the in-game camera so that I can script things like cutscenes.
d. I would like to use UI hotkeys so that I can more efficiently navigate the interface
e. I would like the option to define controller/gamepad bindings so that the game can be played with a gamepad. (time permitting)

6. Map Editor

As a map editor,

a. I would like to import an existing image or map so that I can bring in outside assets.
b. I would like to delete an existing map so that I can manage maps.
c. I would like to edit an existing map so that I can iterate on map design.
d. I would like to edit map properties such as size so that I can have different map types.

7. Item Editor

As an item editor,

a. I would like to manage item properties so that I can have different item behaviors.
b. I would like to add events to items so that I can have reactive item behavior.
c. I would like to place and delete items on the map so that I can create game scenes.
d. I would like to move an item in the editor.
e. I would like to edit existing items (on the map or otherwise) so that I can iterate on item design.

8. Audio

As a user,

a. I would like to import audio for items/sprites so that I can have specific audio tied to game objects.
b. I would like to import audio for music so that my game can have a soundtrack.
c. I would like to view and delete audio so that I can manage audio assets.

9. Code Support

As an advanced user,

a. I would like a system to write and execute my own C++ code so that I can extend Parchment's basic capabilities.

10. Networking

As a user,

a. I would like the option to implement local co-op so that I can create more types of games. (time permitting)

Non-Functional Requirements

1. Usability
    a. As a beginner user, I would like to have documentation readily available to assist and streamline development
    b. As a beginner user, I would like to be able to create a game entirely from the graphical user interface, with no programming needed.

2. Expandability
    a. As an experienced user, I would like to control aspects of the engine using custom-written C++ code

3. Performance
    a. As a user, I would like to develop games that can run within the 30-60 FPS range at minimum.
    b. As a user, I would like the engine to run with optimized resource usage

Design Outline

High Level Overview

The Parchment Engine is a GUI (referred to as the Subsystem Editor below) that allows users to edit their games. All specific game settings, assets, and other information are stored in the game data file. The game data file is then loaded by the game virtual machine, which produces an executable file allowing the user's game to be run.

Engine Components

1. Game Virtual Machine
    a. The executable file used to run any game made in the Parchment engine.
    b. Parses, loads, and executes game data and instructions from the game data file.

2. Game Data File
    a. The file that contains all the game information, assets, audio, and settings.
    b. Created when an editor subsystem session is saved.

3. Editor Subsystem
    a. The game editor GUI where users make and edit their games.
    b. Makes and saves changes to the game data file.

Game Framework Components

1. Prototype
    a. A blueprint for an entity that can be instanced multiple times.
    b. Can be a parent to another prototype, similar to classes.

2. Entity
    a. Anything that can move on the map.
    b. Can be an instance of a prototype.

3. Tile
    a. A unit of the map, a non-moving square.
4. Map

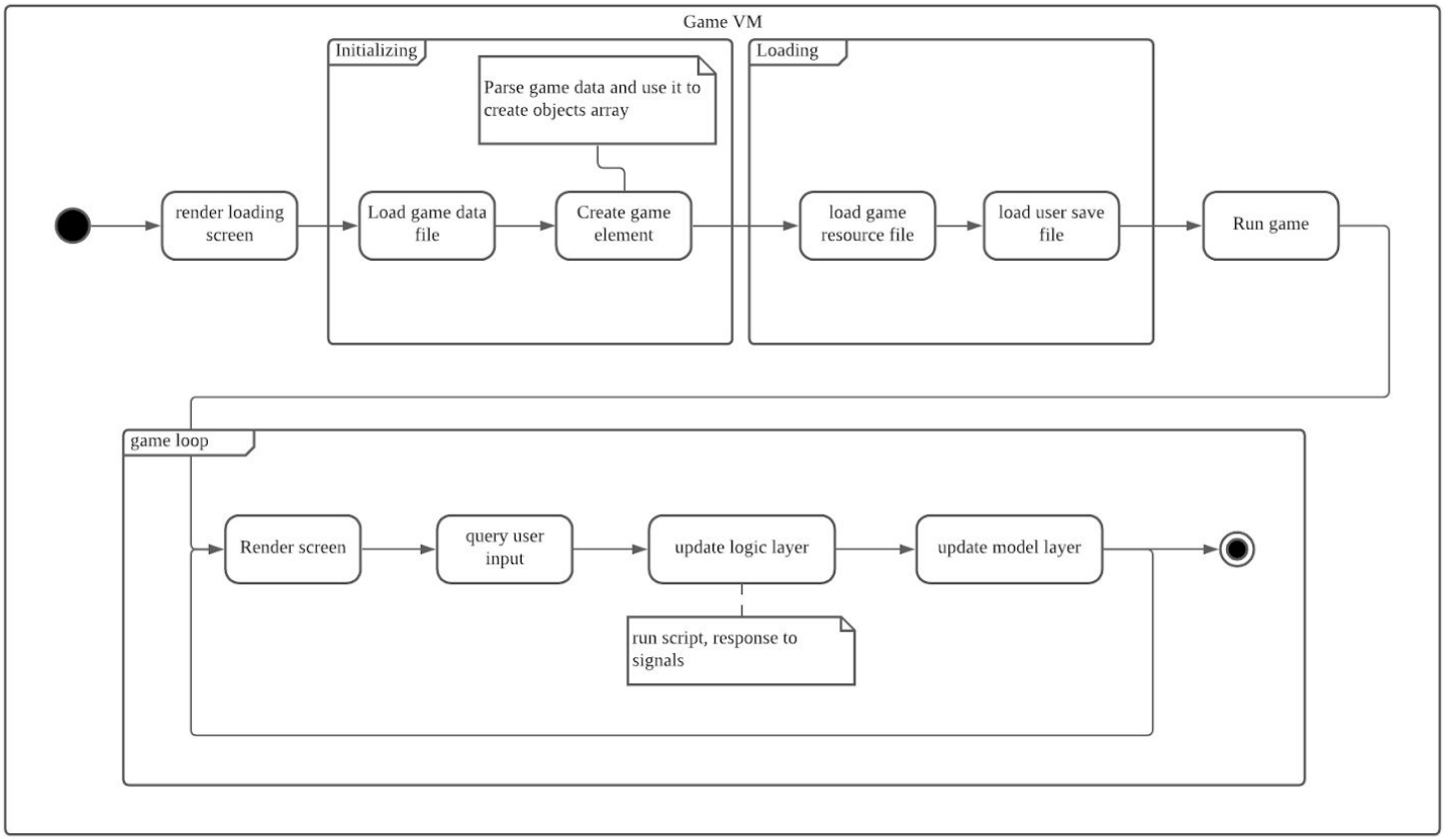
a. A room or game level, organized as an n x m grid.
b. Entities and tiles are found on maps.

5. Sprite
    a. An image for an entity or tile.

6. Page
    a. A rendered screen (e.g. welcome screen, main menu, game scene, score screen).

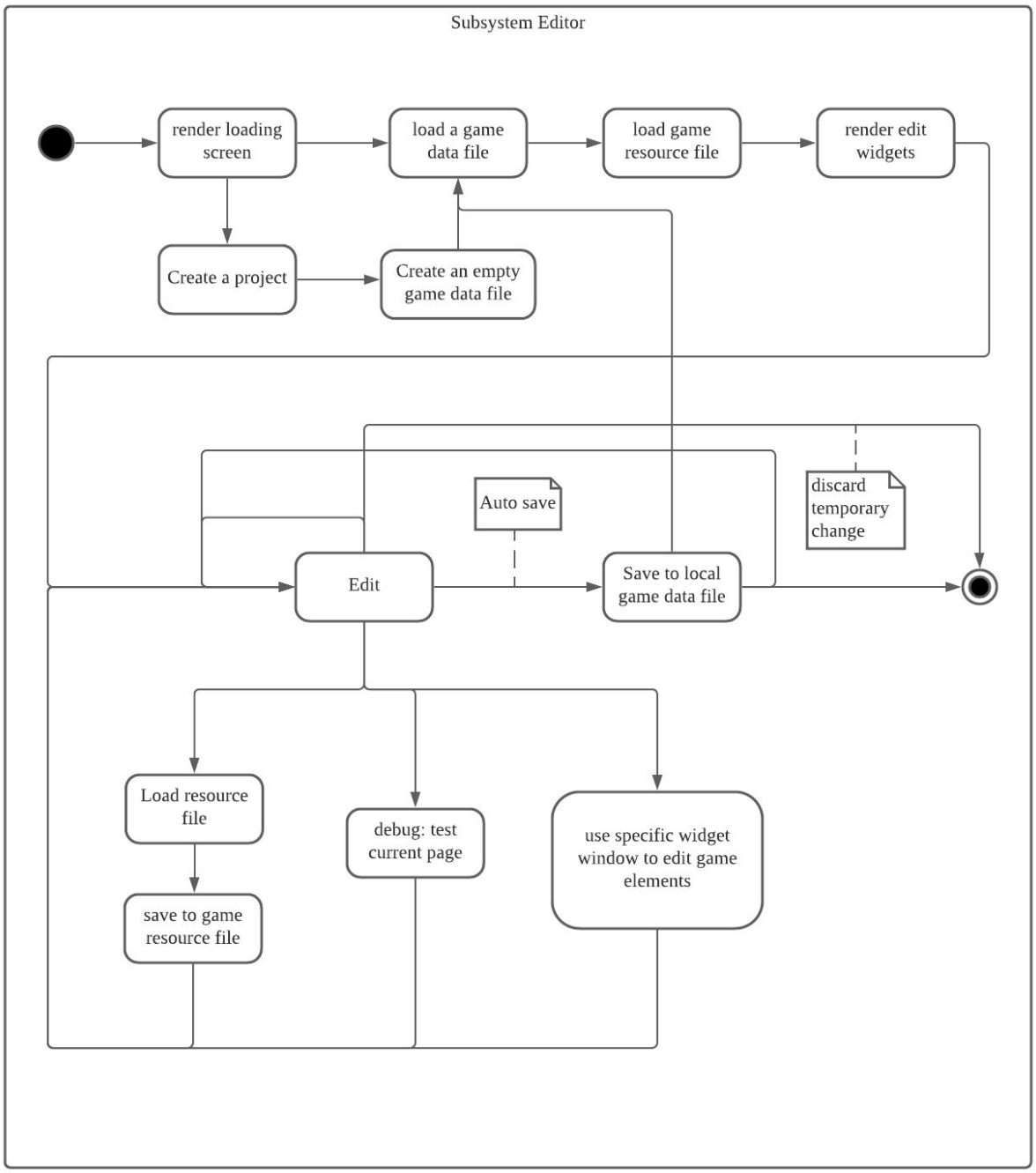
Game VM Structure Diagram

The Game Virtual Machine (GVM) starts by rendering a loading screen and then enters the initializing phase where it loads a game data file, parses game data to create an array of objects, and creates game elements based on what is found in the game data file. Following the initialization phase, the game's resources are loaded, including the user's most recent save file, and the game begins to run. This begins the game loop, a process by which screens are rendered, logic is updated based on provided user input, and the model layer is updated as a result of changes in game logic.



Subsystem Editor Structure Diagram

The subsystem editor allows users to edit their game or create a new project. Once the edit widgets are rendered, a user is able to edit their game and the engine autosaves their editing progress to the local game data file, which is later read by the game VM and “translated” into an executable file.



Design Issues

Functional Issues

1. How does the logic between entities work?
    a. Option 1: Entities activate signals in a large signal array that is being checked by an event loop, which runs logic whenever a signal is found to be activated. These logics are also stored in a large logic array and accessed from the top level.
    b. Option 2: Whenever an entity acts, the game loop queries the logic layer for results.
    c. Option 3: For each in-game time tick, the game loop queries every entity's actions to see if any logic action should be run.
    d. Option 4: Store a local signal and logic array in each page, and a global signal and logic array in the game object at the highest level.
    e. Option 5: Store the logic action data into the entity class and register every entity logic into a global logic array. When the entity acts, call the corresponding logic.

    Choice: Option 1
    Justification: With option 2, we foresaw difficulties debugging problems because of a lack of information between the actor, action, and receiver. This may also hinder extensibility.
    Option 3 is the most accurate way of calculation, but it is also the least efficient.
    Option 4 is a good practice, because it has the information of the signal and action and also maintains an efficient way of determining the state of all entities. But, as we do a mainly object-oriented programming practice, the scope of objects and functions affect the performance of the logic actions. It is complicated from a memory management perspective to manage so many separate arrays.
    Option 5 ignores the signal and only requires information of the actor (group) and actions. However, as it works in an efficient way, it hardly supports multi-relational logic actions (for example, several switches connected to 1 elevator).

    Option 1 completely separates the signal and signal handler. The member functions (map, entities) only need to send signals as it is predefined and know nothing about the handler. Although the signal array is stored in the highest level, we use wrapper functions in the namespace to give accessibility to member functions. The logic actions are defined in the prototypes and managed by the game object. All logic actions are stored into one huge array, so that we can efficiently access and manage memory. All the signals and logics are sent and called using the array index.

2. On which component level will the engine render the screen?
    a. Option 1: Entity/Map tile level
    b. Option 2: Page level
    c. Option 3: Game level

Choice: Option 2
Justification: Our render system will be multi-layered, which means we can support the cool effect of a HUD over the map and allow the pause menu to float over the map. To achieve this functionality, we will use the new feature of OpenGL - context rendering. We will render different pages in different contexts. Then we switch the contexts and stack them to display the multi-layer effect.

3. How do we manage the map and entities on the map?
    a. Option 1: Map and entities are irregular and there's no limitation on them.
    b. Option 2: Map and entities are all regulated and must be managed in square tiles.
    c. Option 3: Maps are managed as square tiles and the entities are freely placed and managed. The picture of tiles or entities are managed in the same way as "sprite".

Choice: Option 3
Justification: The first way is cool, but the later two options work more effectively. The regulated entities are indeed efficient and easy to manage, but choosing this option limits the choice of the users and narrows our deliverability. Option 3 is the best choice as it provides the support of pixel art and simple map management as well as the freedom of placing entities. Meanwhile, the unified structure of both map tile sprite and entity sprite give a better view of the resources.

4. How should we format our game data file?
    a. Option 1: JSON
    b. Option 2: XML
    c. Option 3: Custom binary format

Choice: Option 1
Justification: Representing the game data file as a JSON file would be the simplest option since JSON often stores data in readable text and is the easiest to deal with. XML would be a good option as well for similar reasons, however we ultimately decided on JSON for convenience.

5. Does a user need login information to access Parchment?
    a. Option 1: Require account creation to access the engine.
    b. Option 2: Do not require account creation to access the engine.
    c. Option 3: Leave account creation as optional.

Choice: Option 2
Justification: We will not initially provide account creation as it seems to be an extraneous feature that can be added in later releases (possibly closer to the end of development) if it is deemed necessary. Adding it in the early stages of development would take valuable coding time away from more important features and we wanted to shift such a feature towards the end of development when we could make account creation optional, if it becomes a feature at all.

6. How does the user export their game from the editor?
    a. Option 1: Export a separate Game File with the Game VM executable
    b. Option 2: Pack Game File and Game VM into a single executable file

Choice: Option 1

Justification: We want our game to be both easily shared and modified. So we designed a Virtual Machine-File-Editor structure and let the VM be completely separate from the game designing. It only reads the data file and acts like a translator which parses all data to memory and constructs a game object (.exe) to run. Users can copy both the game data file and game resource file (if necessary) to run with any copy of the game virtual machine or to be edited in any copy of the game editor.

7. In which level should we separate the classes between the Subsystem Editor and Game VM?
    a. Option 1: Make it as one.
    b. Option 2: Make them completely separate.
    c. Option 3: Make them partially separate.
    d. Option 4: Make them partially separate, but make some overlap functions as well.

Choice: Option 2

Justification: There are three main parts of Parchment: The game VM, game data file, and game engine. We need to include the game class in both the VM executable and the editor executable. Firstly, we thought about writing a single executable and letting users choose modes at the beginning of the program. However, this is an undesirable choice. We want to release a real functional user-friendly engine instead of a rough “college project”. We debated on Option 3 and Option 4 a lot. The partially separated structure is better at implementation and support, but requires more redundant work. If we overlap some of the functions, which most likely involves the game class, we have to deal with a scope issue in the case of object-oriented programming -- how to access the member of the game from both the grand child of the game object and the member of VM or editor file. Meanwhile, we have to keep the dignity of the game data file to be separable for sharing. As we were trying to make the balance between the separation and overlap, we came up with an idea for a virtual machine. By using a game VM that takes a game data file and parses the data from it to create an executable file, this eliminates the aforementioned problems, making Option 2 the most attractive choice for our development purposes.

Non-Functional Issues

1. What language should the engine be coded in?
    a. Option 1: C
    b. Option 2: C++
    c. Option 3: Java

d. Option 4: JavaScript

Choice: Option 2

Justification: C++ is an industry standard for game and game engine development, meaning there is a wider variety of resources and literature on the subject of adapting the language to solve game engine problems. Also, because the engine itself is written in C++, it makes sense to support C++ code.

2. How should we create the GUI?
    a. Option 1: Qt
    b. Option 2: ImGui
    c. Option 3: Roll our own with OpenGL drawing

Choice: Option 2

Justification: ImGui is built on top of OpenGL and is intuitive to code menus with. The immediate GUI model it uses bypasses the need for any sort of event listeners or delegation of events, making it easy to rapidly develop new GUI components and features. Therefore, it is very evident that ImGui is the best option.

3. What languages should be supported for user generated code?
    a. Option 1: C++
    b. Option 2: Python
    c. Option 3: Lua
    d. Option 4: a custom node-based or otherwise graphical language

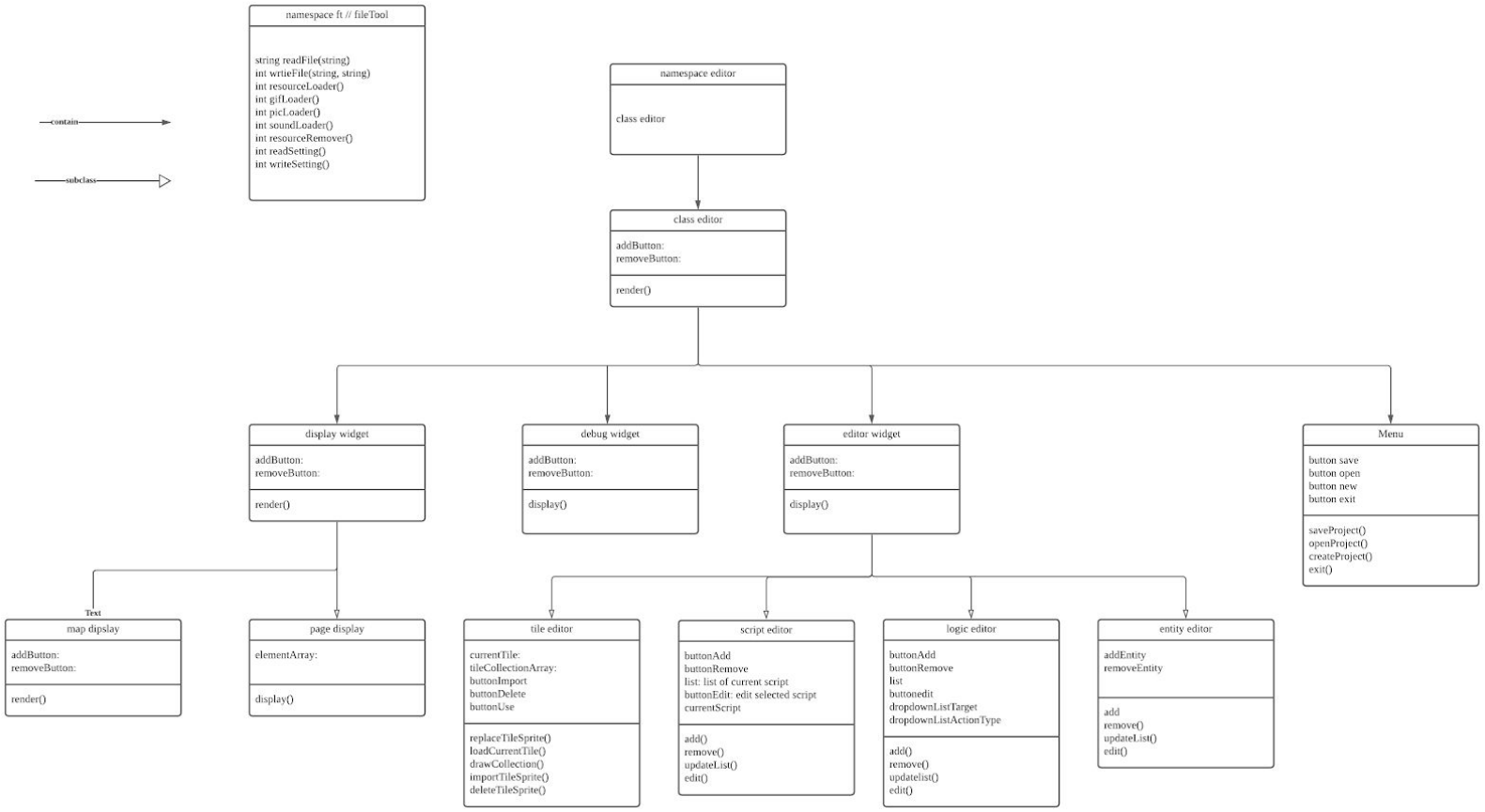
Choice: Option 1

Justification: We can support override of our member functions, so C++ is the straightforward solution for user support language. Since our engine is mainly built for non-programmer users, we will assume most users will not touch the coding. As a result, we don't need to support common languages like Python for better accessibility. The reason we decide to support such functionality (user code support) is because we can let the users specialize the engine for their desire if they wish to do so. And since the engine is primarily coded in C++, it makes sense for C++ to be the primary language for user code support.

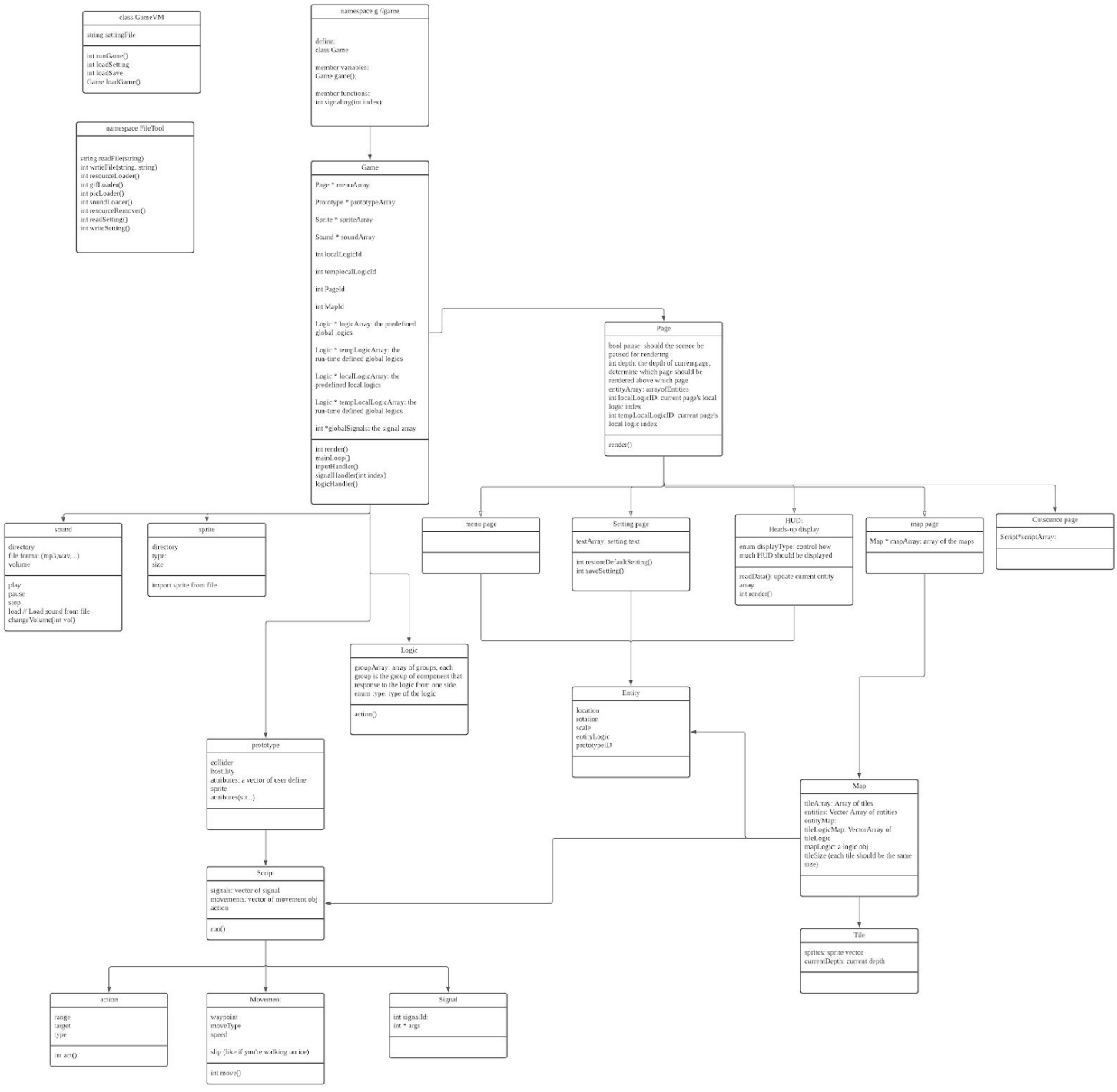
Design Details

Class Diagrams

The following two class diagrams highlight classes and their relationships with one another in the game editor and the game virtual machine (VM). The first diagram involves the game editor while the second involves the game virtual machine.



If you have difficulty reading the class diagrams above and below, they are also accessible via this Imgur link: https://imgur.com/a/I9HFqbA


class GameVM
string settingFile
int runGame()
int loadSetting
int loadSave
Game loadGame()
namespace g //game
define:
class Game
member variables:
Game game();
member functions:
int signaling(int index);
namespace FileTool
string readFile(string)
int wrtieFile(string, string)
int resourceLoader()
int gifLoader()
int picLoader()
int soundLoader()
int resourceRemover()
int readSetting()
int writeSetting()
Game
Page * menuArray
Prototype * prototypeArray
Sprite * spriteArray
Sound * soundArray
int localLogicId
int templocalLogicId
int PageId
int MapId
Logic * logicArray: the predefined global logics
Logic * tempLogicArray: the run-time defined global logics
Logic * localLogicArray: the predefined local logics
Logic * tempLocalLogicArray: the run-time defined global logics
int *globalSignals: the signal array
int render()
mainLoop()
inputHandler()
signalHandler(int index)
logicHandler()
Page
bool pause: should the scence be paused for rendering
int depth: the depth of currentpage, determine which page should be rendered above which page
entityArray: arrayofEntities
int localLogicID: current page's local logic index
int tempLocalLogicID: current page's local logic index
render()
sound
directory
file format (mp3,wav,...)
volume
play
pause
stop
load // Load sound from file
changeVolume(int vol)
sprite
directory
type:
size
import sprite from file
menu page
Setting page
textArray: setting text
int restoreDefaultSetting()
int saveSetting()
HUD
Heads-up display
enum displayType: control how much HUD should be displayed
readData(): update current entity array
int render()
map page
Map * mapArray: array of the maps
Cutscence page
Script*scriptArray:
Logic
groupArray: array of groups, each group is the group of component that response to the logic from one side.
enum type: type of the logic
action()
Entity
location
rotation
scale
entityLogic
prototypeID
prototype
collider
hostility
attributes: a vector of user define
sprite
attributes(str...)
Map
tileArray: Array of tiles
entities: Vector Array of entities
entityMap:
tileLogicMap: VectorArray of tileLogic
mapLogic: a logic obj
tileSize (each tile should be the same size)
Script
signals: vector of signal
movements: vector of movement obj
action
run()
Tile
sprites: sprite vector
currentDepth: current depth
action
range
target
type
int act()
Movement
waypoint
moveType
speed
slip (like if you're walking on ice)
int move()
Signal
int signalId:
int * args

Class Descriptions

Game Framework

1. Game
    - This class will maintain much of the information related to the game, such as the pages of the game, the images, audio, and logic information. When you want to add something to the game itself, it will get added here in some way.
    - The game class also takes care of rendering the actual game because it contains all of the necessary data needed to do so.

2. Sound
    - These objects can be created to represent a unique sound. This class will be used primarily within the Game class.
    - Sounds are sound files that have been added to the engine and can be played, paused, and stopped by calling the respective functions of the class instance.

3. Sprite
    - Sprites are essentially images that can be used anywhere within the engine.
    - Tiles have sprites, entities have sprites, items have sprites, players have sprites. Anything with a visual component contains a sprite.

4. Prototype
    - The prototype class contains information that can be used to instance a certain kind of entity. This information can be things like HP, strength, friendliness, or other quantifiable properties.

5. Script
    - Scripts are essentially just a sequence of signals, where a signal can be something like movement, for example. A script for moving an entity could be a sequence of “Movement” instructions in different directions. Any entity can contain a script.

6. Logic
    - At its most basic level, Logic is something that happens as a result of something happening. For example, moving the player to a certain tile could cause an avalanche to occur, or shooting an object could signal an explosion to occur.
    - Objects such as Entities and Maps can include logic

7. Map

- This is the “map” of the game, or the world that the player exists on. Maps contain, in general, a 2D array of tiles, which determines the layout of the world.
- The Map class will also keep track of tileLogic and any entities on the current map.

8. Tile
   - A tile is a basic unit of the map. Each tile in a map has the same size.
   - Each tile contains a corresponding sprite and a depth. The depth allows objects to be placed above or below other objects on the map.

9. Entity
   - Entities are anything that is non-static on the map. Any object that moves is an entity, any object with logic is an entity, any object with scaling or rotation is an entity.
   - The entity class contains all the possible modifiable parameters of an entity, such as location, rotation, and scale.
   - Entities can be assigned logic or a prototype.

Editor Subsystem

1. Page
   - Pages are analogous to website pages. They contain all the information for a particular “screen” of the game. For example, a “settings” page would contain data for the components to adjust the volume, full-screen, brightness, etc., while a “menu” page would contain the data needed to display buttons for “Create a new game”, “Open an existing game project”, etc.

2. Editor
   - The Editor class is essentially the heart of the creation side of the engine. The editor maintains the Display widget, the Editor widgets, and the menu widget.
   - These widgets compose the GUI side of the engine, and interacting with the widgets will change the makeup of the game you are making.

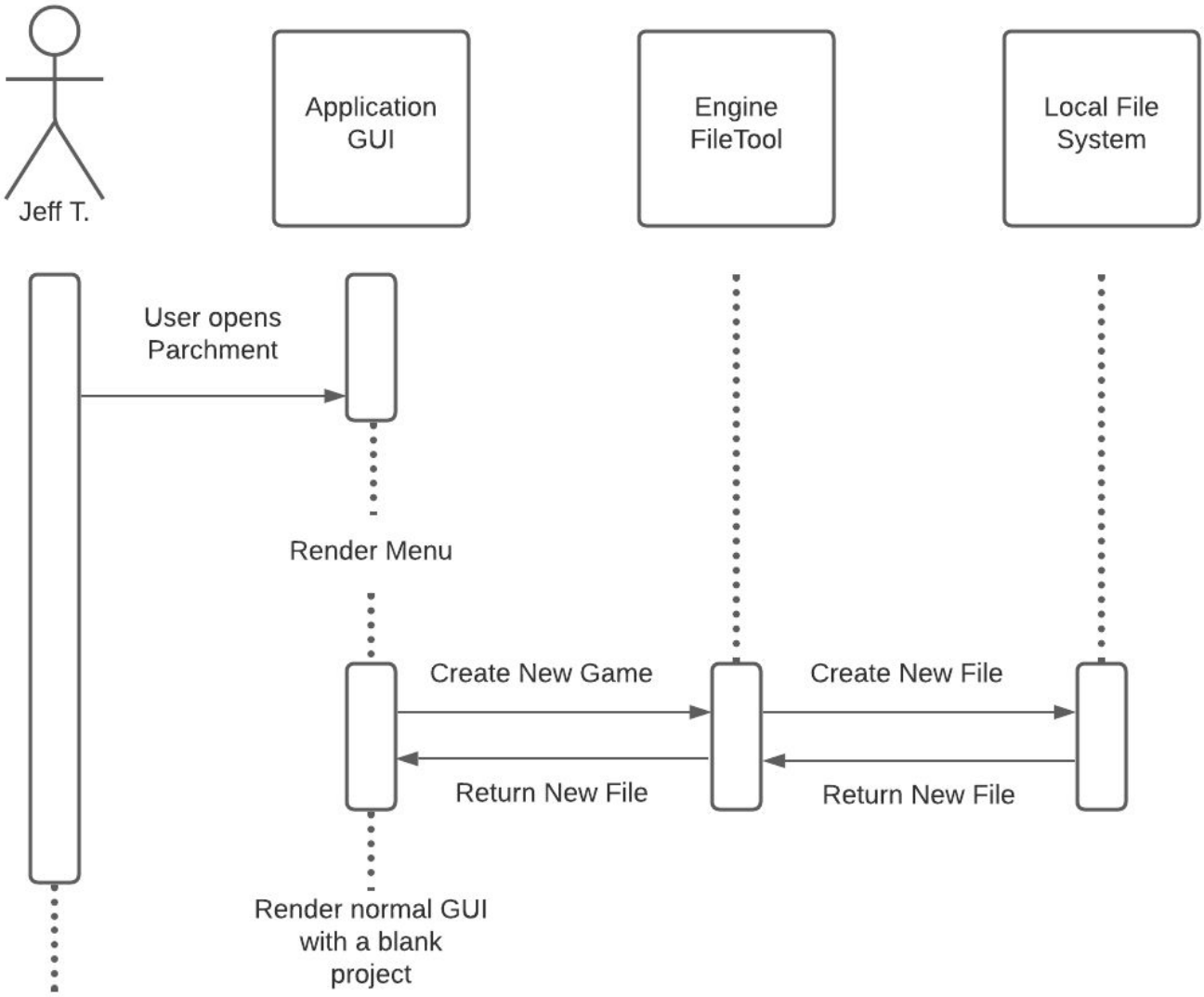
3. Widget
   - Display Widget - This is where the Game you are making is shown. This could show the map if you are currently working on the map, or a specific page.
   - Editor Widget- Provides a few different types of editors, the “Tile Editor”, “Script Editor”, “Logic Editor”, “Sprite Editor”, and the “Entity Editor”

4. Menu Widget

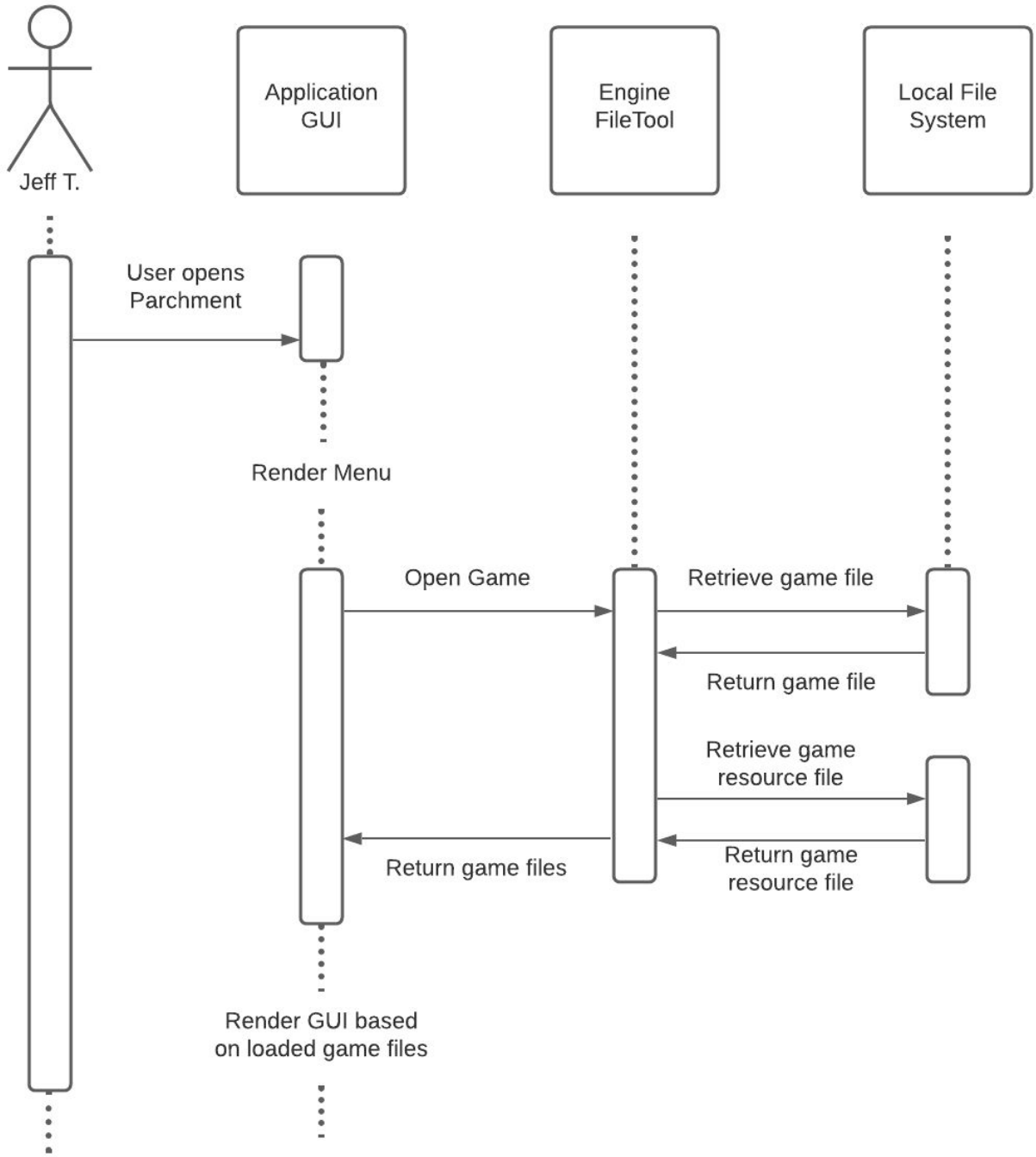
- Provides the user with game management options, such as “New Game”, “Open Game”, and “Save Game” much like the File tab of a program such as Microsoft Word.

Sequence Diagrams

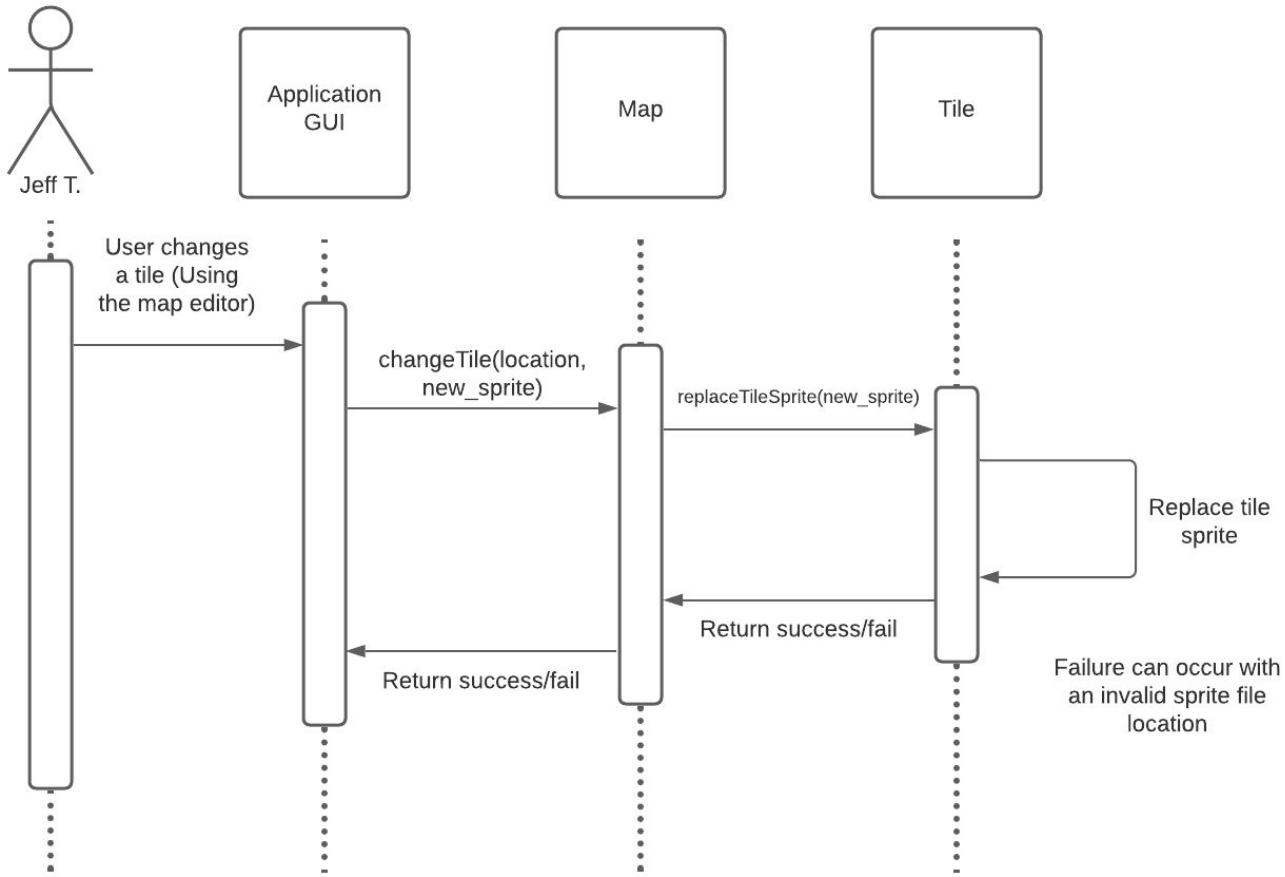
1. Sequence of events when users create a new game



2. Sequence of events when users open a game



3. Sequence of events when users edit the tiles of a map



Navigation Flow Map

The diagram below shows the ease of access Parchment provides, and walks a user through the most typical actions that can be performed in the engine. While not a complete view of the engine and its abilities, this rudimentary flow map can provide a general idea of what to expect when launching and using Parchment.