

**NAME** **ROLE** **CHAMPIONING**

Terence Chan Producer AI/Input/Memory

Lim Jia Hao Technical Engineer Physics/Collision/Sound

Javier Foo Graphics Programmer Render/Animation/QA

Loh Kai Yi Tool Programmer Architecture/Editor/Audio

Chong Teck Hsun Lead Design Level Design

Darren Ang Art Lead Mechanics Design

# Team Name:

Stealth Studios

# Game Title:

Exale

Technical Design Document

GAM200F19 – A

FALL 2019

Table of Contents

[Team Name: 0](file:///D:\Downloads\TDD_1.docx#_Toc26044189)

[Game Title: 0](file:///D:\Downloads\TDD_1.docx#_Toc26044190)

[Architecture 4](#_Toc26044191)

[Overview 4](#_Toc26044192)

[Structure 4](#_Toc26044193)

[Graphics Implementation 6](#_Toc26044194)

[Sprite Component 6](#_Toc26044195)

[Texture Manager 6](#_Toc26044196)

[Graphics System 6](#_Toc26044197)

[Future Plans 7](#_Toc26044198)

[Physics Implementation 7](#_Toc26044199)

[Forces and Integration 7](#_Toc26044200)

[Collision Detection 7](#_Toc26044201)

[Contact resolution 8](#_Toc26044202)

[Math library 8](#_Toc26044203)

[Entity Component System 9](#_Toc26044204)

[Entity 9](#_Toc26044205)

[Component 9](#_Toc26044206)

[System 9](#_Toc26044207)

[Coordinator 9](#_Toc26044208)

[Memory Management 9](#_Toc26044209)

[Event System 10](#_Toc26044210)

[Artificial intelligence/game logic 10](#_Toc26044211)

[Movement 10](#_Toc26044212)

[Detection 10](#_Toc26044213)

[Finite State Machine 10](#_Toc26044214)

[Serialisation 11](#_Toc26044215)

[General Purpose 11](#_Toc26044216)

[Loading 11](#_Toc26044217)

[Writing 11](#_Toc26044218)

[Single Player/Multiplayer Implementation 11](#_Toc26044219)

[Coding Methods 11](#_Toc26044220)

[Coding Conventions 11](#_Toc26044221)

[Code Formatting 12](#_Toc26044222)

[File Locations 12](#_Toc26044223)

[Source Control 12](#_Toc26044224)

[Code Documentation 12](#_Toc26044225)

[Debugging 13](#_Toc26044226)

[Visual Studios Debugger 13](#_Toc26044227)

[Assertion System 13](#_Toc26044228)

[Debug Console 13](#_Toc26044229)

[Tools 13](#_Toc26044230)

[Visual Studio 2019 13](#_Toc26044231)

[Unity 13](#_Toc26044232)

[Photoshop 13](#_Toc26044233)

[Clickup 13](#_Toc26044234)

[Musescore 3 14](#_Toc26044235)

[Protools 14](#_Toc26044236)

[Level Editor 14](#_Toc26044237)

[Hierarchy 15](#_Toc26044238)

[Inspector 16](#_Toc26044239)

[Viewport 17](#_Toc26044240)

[Profiler 18](#_Toc26044241)

[File System 18](#_Toc26044242)

[Console 19](#_Toc26044243)

[File loading and saving 19](#_Toc26044244)

[Audio 20](#_Toc26044245)

[EXTERNAL lIBRARIES 20](#_Toc26044246)

[Scripting Languages 20](#_Toc26044247)

[Technical Risks 20](#_Toc26044248)

[Tokenization for saving and loading 20](#_Toc26044249)

[Integration of Scripting languages 21](#_Toc26044250)

[Appendix A 21](#_Toc26044251)

[Flowchart 21](#_Toc26044252)

[Mockups 22](#_Toc26044253)

[Appendix B 24](#_Toc26044254)

[Art Requirements 24](#_Toc26044255)

[Appendix C 24](#_Toc26044256)

[Audio Requirements 24](#_Toc26044257)

# Architecture

### Overview

This project is programmed in C++ using Microsoft Visual Studio 2019, and focuses on building a 2D game engine (*Rogue Engine*) and level editing tools from scratch (with the help of several 3rd party libraries) to produce the game, *Exale*.

Rogue Engine is component-based and its core provides multiple systems required for the game.

### Structure

Rogue Engine can be broken down to these major components:

#### Core

The engine core contains the REEngine object, Entry Point and Entity Component System.

#### Graphics System

The graphics system handles rendering of objects on a window using OpenGL.

#### Physics System

The physics system handles rigidbodies and collisions

#### Math Library

The math library consists of Vector Operations, Transformation Matrix Operations and generic math calculations.

#### Entity Component System (ECS)

The ECS uses a coordinator to handle all entities, components and systems. It uses a signature-based design to determine the components in every entity.

#### Memory Management

The memory manager allocates a memory pool for the objects.

#### Event System

The event manager consists of an even dispatcher and listener that allows systems to receive events and simulate game behaviour.

#### Artificial Intelligence/Game Logic

The AI system would handle the movement behaviours, facilitate player detection and a finite state machine.

#### Serialization

The serializer would handle the loading and saving of game data.

#### Level Editor

The level editor uses the ImGUI 3rd Party Library to support creation of scenes and components. It would be able to play/pause scenes and configure prefabs.

#### Scripting

The script manager would use the LUA Language to allow designers to create scripts for the game.

#### Debugging Tools

The engine uses a logger based on spdlog 3rd party dependency. It also supports debug drawing of meshes and colliders. Asserts are used as much as possible to prevent undefined behaviour, reducing debugging durations.

#### Audio

The engine uses the FMOD 3rd Party Library to handle sound effects and music events in the game.

# Graphics Implementation

The Graphics Implementation for Rogue Engine makes use of the OpenGL application programming interface to handle and render graphics. The implementation makes use of several external libraries, listed in the table below.

|  |  |  |  |
| --- | --- | --- | --- |
| S/N | Library Name | Abbreviation | Usage |
| 1 | OpenGL Extension Wrangler | GLEW | Load OpenGL extensions based on target platform |
| 2 | Simple OpenGL Image Library | SOIL | Texture loading |
| 3 | OpenGL Mathematics | GLM | Math library based on OpenGL Shading Language |

In-game graphics are handled by the Graphics System, whose primary job is to render the data based on the SpriteandTransformcomponents.

### Sprite Component

The Sprite component specifies the texture of the game object. As loading textures is expensive, a Texture Manager is used to ensure that texture loading happens only for textures that have not previously been loaded.

### Texture Manager

The Texture Manager stores a map of loaded textures with their file path as the key. Each time a Sprite component has its texture data set, the manager first searches through the map. If a texture with the same file path already exists, the manager returns the existing texture. Otherwise, it loads the texture using the SOIL library and adds the new texture to the map.

### Graphics System

All sprites are rendered using two OpenGL Triangle primitives arranged into a rectangle. The appearance of non-rectangular shapes, such as the player character, is achieved through the alpha channel of the loaded PNG texture. Initialisation of the Graphics System involves binding a unit primitive to a stored Vertex Array Object (VAO) in the form of two triangles arranged as a rectangle. The same VAO is used for each draw call to render each game object. For easier understanding, the process can be explained in layman terms as such:

Every game object starts out as a rectangle of unit size positioned in the center of the screen.

A texture based on the Sprite component is applied.

The rectangle is then scaled, rotated and translated according to the Transform component.

To achieve the transformations in step 3, a transformation matrix is generated using the GLM library and sent to the vertex shader, which applies the transformation to the game object.

### Future Plans

For better control of draw order and optimisation, depth testing will be used in the rendering process. Support for early depth testing would also allow us to discard fragments that will never be seen (e.g. those covered by other game objects), making for a more efficient rendering process.

## Physics Implementation

### Forces and Integration

The use of forces allows us to simulate collisions and movement more accurately and conveniently. Forces would be useful to resolve contacts or collisions, apply gravity, and simulate game logic and behaviours. The accumulated force of each rigidbody will be exposed to the user to facilitate behaviours using just force. This way, the forces are able to generate acceleration, which is subsequently integrated to compute the velocities and positions (without manually offsetting these values, if possible).

### Collision Detection

The collision detection design would be split into two parts – Broad phase and narrow phase. The broad phase segregates the colliders in the world to smaller subdivisions. The narrow phase would then test for intersection between colliders that are within the same subdivision.

#### Narrow Phase

The key features of narrow phase would be the intersection tests between *Bounding Circles* (BC), *Axis-Aligned Bounding Boxes* (AABB) *and Oriented Bounding Boxes* (OBB). OBB supports polygons with n (any) number of sides, which supports a more appealing level design using entities of different orientations and providing more accurate collisions.

#### Broad Phase

Aside from providing a hierarchy for different bound volumes, several partitioning methods are also considered in the design of collision detection.

Quad Tree

*The Quad Tree* data structure, splits the world into subdivisions, four parts at a time. This way, intersection tests would only be required for colliders in the same region.

Bounding Volume Hierarchy

A simple hierarchy is designed where intersection tests will be filtered from the cheapest to most expensive. It involves checking for overlaps between bounding circles (whether statically or dynamically), followed by the testing of one bounding circle with AABB or OBB, and finally the testing of box/polygon colliders.

Either or both broad phase techniques would be implemented in the engine.

### Contact resolution

If entities collide with each other, a manifold containing contact points and penetration depth would be generated to resolve the constraints. On the more advanced end, forces such as impulse, reflection and/or torque could also be used to simulate the desired collision behaviour.

# Math library

The Math Library mainly consists of functions that will assists in most of the calculations of the Engine for the game editor (e.g. Physics). The library mainly uses a vector2d and matrix3x3 library. Expansions of the vector and matrix library (i.e. templated vector and matrix classes) will be done in GAM 250 to assist in more complex calculations.

# Entity Component System

The engine is based on the Entity Component System (ECS) design. As opposed to Object Oriented Programming (OOP), this design allows users to attach and detach components to/from an entity during level design, allowing tremendous flexibility. The ECS is divided into three parts, with a coordinator that handles all three of them.

### Entity

An entity is simply the number representation of a game object. It is an ID that allows the coordinator to identify its components and its signature.

Every entity would have its own signature, represented as an *std::bitset*. Each signature signifies all the components that each entity has (i.e. every bit represents one component type).

### Component

A component is the property of an entity. It solely contains component data (such as health or velocity). An entity can be attached to multiple components such as *RigidbodyComponent*, *SpriteComponent*, *TransformComponent* and so on. Each contain data that is relevant only to its purpose. Each type of component is stored in a component array.

### System

Every system inherits from the interface *class System* and stores its own *std::set* (i.e. list) of entities. In the initialization, every system will set its own signature to signify the components it will access. This way, only entities with matching signature would be inserted into the list.

### Coordinator

The coordinator handles the entities, components and systems using maps and arrays. Whenever a component is added to an entity, the coordinator will perform operations to register the entity into systems that handle the component.

# Memory Management

A Memory Manager is required to keep track of the space being used for run-time memory allocation and to prevent memory leaks. The Memory Manager will be designed in a memory pool format, allowing for constant execution time and ease of use in allocating and deallocating memory, as it only deals with a singular memory space. As the Component Manager already tracks each component, there is no need for an Object Allocator style memory manager. Smart pointers will also be used when needed, in order to allow better memory management.

# Event System

Event System is used for the systems to communicate between each other, without needing to directly depend on each other. The Event Dispatcher is the messaging system used to do so, by taking in Events and sending them on to the relevant systems. Other systems will only send and receive messages from the Dispatcher, allowing for lesser inter-system dependencies.

# Artificial intelligence/game logic

The AI system for Exale built for all non-user-controlled entities, from props to moving mobs. This system handles two aspects, movement and detection, changing via a finite state machine.

### Movement

Movement of non-user-controlled entities will be controlled via waypoints. While in the patrolling state, these entities will travel from one waypoint to the next, repeating in a closed loop. This will only be executed within a certain range of the player to reduce memory usage. This includes rotation of line of sight, as some AIs are focused more towards being stationary sentinels and looking for the player.

### Detection

Once the player character is detected, AI will break away from their original patrolling or stationary states, following the player if able. This will be done via A\* pathfinding. Once a player is out of range, or if the AI strays too far from its nearest waypoint, the AI will return to its patrolling state.

### Finite State Machine

By implementing a Finite State Machine, the AI will have certain sets of behaviours that will be easily predictable. This Finite State Machine will have a hierarchy to allow each individual type of AI to decide which is the most optimal behaviour at any moment, with a default state to follow.

# Serialisation

### General Purpose

Serialisation is used in Rogue Engine to handle all the data for our components to ensure the configuration and data will be data-driven. The team will be using rapidJson’s library to assist in our serialisation and de-serialisation.

### Loading

All constant data of all components will be stored in their respective json file. When the component is going to be used, the file with the component will be read and initialised and assigned to the variables in our solution file. Also, all constant data in the component must be saved in a single file to prevent name clashing. The json file must be also organised in a way to enhance readability.

### Writing

When the user exits the engine, data that has been modified has to be overwritten in the json so that when the user restarts the engine, they will be able to access their saved data. By wrapping rapidJson’s parse and write functions, we will be able to overwrite those data in the components and save data whenever the user exits the engine.

# Single Player/Multiplayer Implementation

Rogue Engine does not support multiplayer gameplay.

# Coding Methods

### Coding Conventions

The naming of variables should be standardized as follows:

* Global variables will have a “g” in front (gEngine, gDeltaTime)
* Core classes will have “RE” in front (REEngine, RECoordinator, REPhysicsSystem, REVec2)
* Member variables will have “m\_” in front (m\_transform, m\_coordinator)
* The engine will be encapsulated in “*namespace Rogue”*
* All variables and functions will be named after its purpose, as clear as possible, instead of acronyms
* Files will be named in accordance to the system it is under, along with the purpose it serves

### Code Formatting

The format of the code should be standardized as follows:

* Standard proper indentation of C++ code
* Tabs for indentation instead of spaces
* General logic must be commented within functions and classes
* All codes belonging to external libraries are placed in a separate “Dependencies” folder
* When there is a huge body of code, use line breaks or helper functions to indicate what the function is used for or functionality.

### File Locations

The format of file location should be standardized as follows:

* All header files and source files will be stored in folders labelled with its purpose (e.g. “Physics” and “Graphics”)
* All external resources like images, json files and sounds will be saved under the “Assets” folder

### Source Control

Github will be used as our main source control. Everyone has their own branch, and should only push code into their own branches. No one is allowed to touch another person’s code unless they can prove that their own code will only work with that alteration. They are to commit with the amendment in their own branch and alert the author of the other code to update theirs.

### Code Documentation

Every file will contain a file header. The purpose and procedure of each function will be written using end line comments above the function declarations in the header files.

# Debugging

### Visual Studios Debugger

Visual Studios Debugger allows us to set break points and track game flow at runtime. It also contains an error list that displays all errors and warnings that debugging could begin from. Memory leaks will also be checked using the win32 API. This will be our main debugging tool for the engine.

### Assertion System

Besides from the Console Debug System, there is also an Assertion System which tells us that an assertion failed due to an error (e.g. Array out of bounds, or file not found). This allows us to debug line by line on where the error is coming from, and also to prevent undefined behaviour in the code which makes it harder to debug.

### Debug Console

Aside from the debugging tools provided by Visual Studios, the game’s main Integrated Development Environment, Rogue Engine also has its own console debug system which lets us print data and texts for debugging.

# Tools

### Visual Studio 2019

Visual Studios will be the main third-party tool used for our game engine. It provides us the simplest means of compiling and running the executable program.

### Unity

Unity will be used by our game designers for the prototype of our game. It allows us to look at the game mechanics and the technical aspects needed for our game engine.

### Photoshop

Used for editing of sprite sheets and user interface.

### Clickup

A website used for keeping track of tasks.

### Musescore 3

Musescore will be used by our sound designer for scoring and charting for the music of Rogue Engine.

### Protools

Protools will be used by our sound designer for sound engineering and sound mixing for the music for Rogue Engine.

# Level Editor

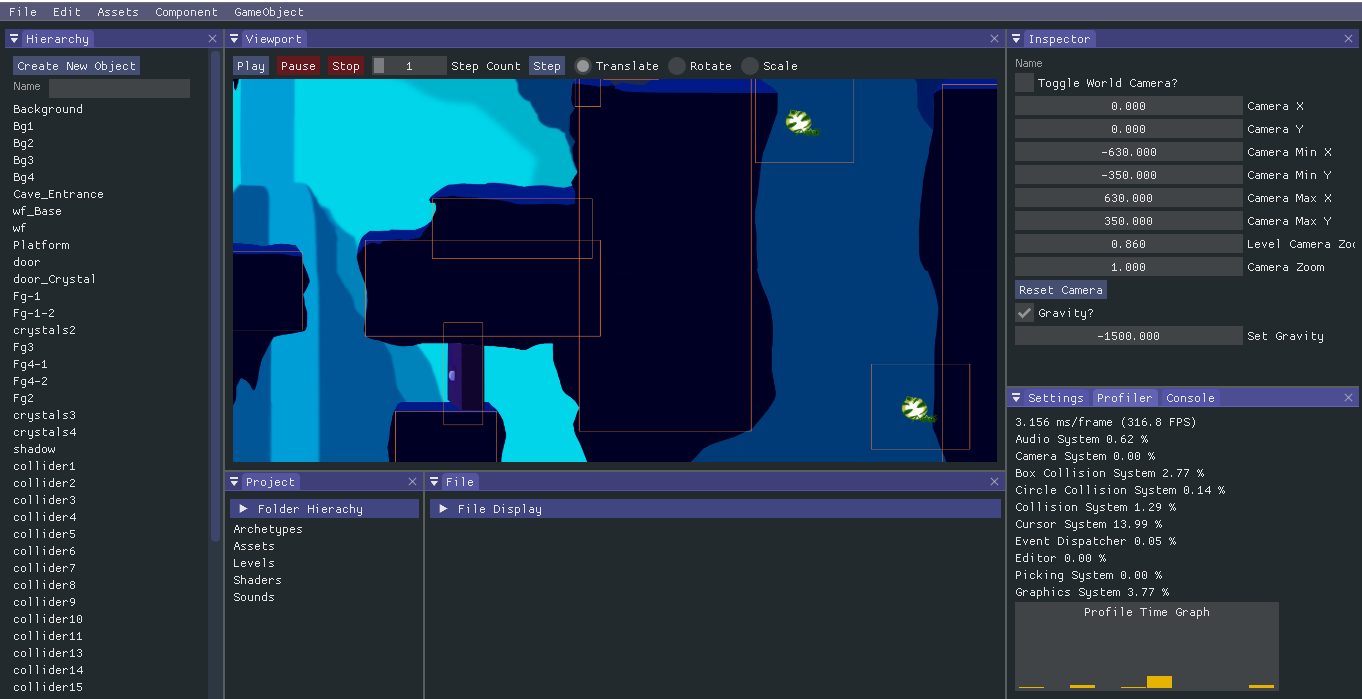


Figure 6: Overall structure of level editor

Rogue Engine has a level editor that is built based on the base game engine. When the solution is launched, the editor will be the main UI that will be shown to the user. The editor will be built based on the ImGui library, which assists in display the graphics UI for the editor.

As there is a huge amount of game objects and components being loaded in the game, the level editor allows our designers to easily access all the game objects that is in the current scene and make amendments based on the editor. The editor can be toggled into non-editor mode which the game can be played.

### Hierarchy

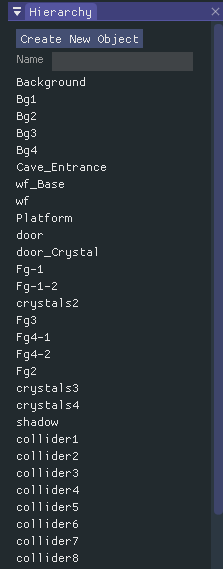


Figure 7: structure of Hierarchy

The Hierarchy contains the list of the current game objects in the scene. The user using the editor will be able to create a new game object and the object will be accessible through the hierarchy. When selected, data will be shown on the viewport and the inspector. The user will also be able to filter the game object through the name.

### Inspector

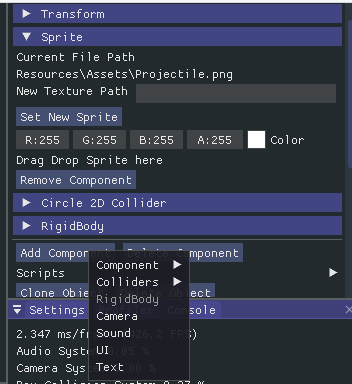


Figure 8: structure of inspector

When a game object is selected through the hierarchy, the inspector will display all the information about the game object. The user will be able to change the name of the object, add a new tag to the object or save the object as a prefab. The user will also be able to add and delete components which will be attached to the game object. The values like its position can be changed via the inspector and will reflect on the viewport. The values that were changed can be saved and loaded as those values are serialised and deserialised from the engine.

### Viewport

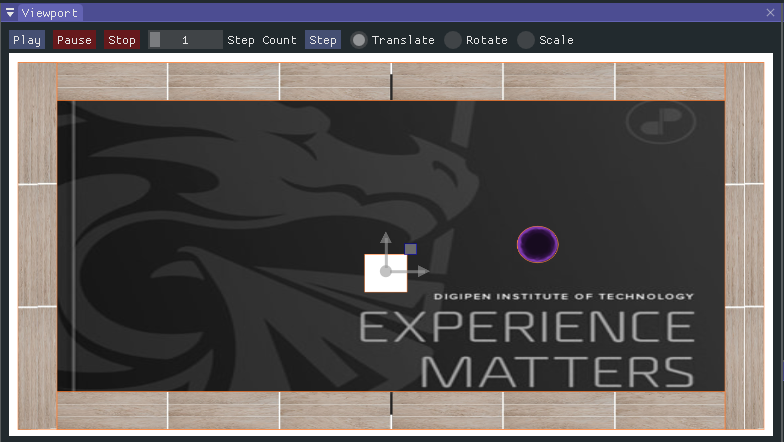


Figure 9: structure of Viewport

The viewport is where the user sees all the objects that are in the scene. When an object is selected, either through the hierarchy or double clicking the object in the viewport, a gizmo would appear on the object and the user will be able to either translate, rotate or scale by dragging in the viewport. The user also can play, pause and stop to scene in order to test out on the gameplay of the scene. There is also a stepping system, when the button is clicked, will go through a few frames depending on the sliding amount. This will assist in the technical side on debugging, especially on the physics.

### Profiler

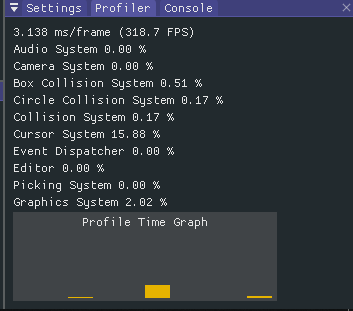


Figure 10: structure of Profiler

The profiler contains information like the current frames the editor is running per second (FPS) and the amount of time each system is current taking to run, totalling up to 100%. This allows us to find out which system is taking up the most in the engine so that we would know which system we should optimise first to get maximum performance increase.

### File System

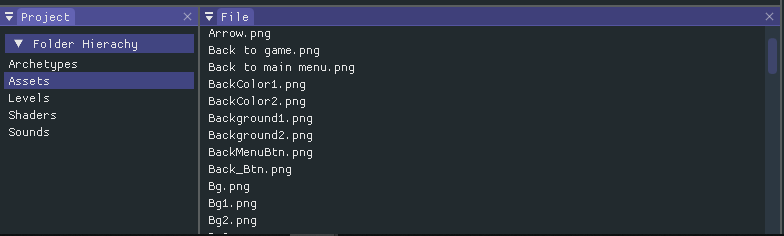


Figure 11: structure of File System

The file system contains all the assets that can be used in the engine. The assets are updated runtime and the files can be drag and dropped to update the scene. The files that are supported are json files to load the level and sounds and png files to load the audio and sprites. Error handling is also provided when the user drags the wrong file format.

### Console

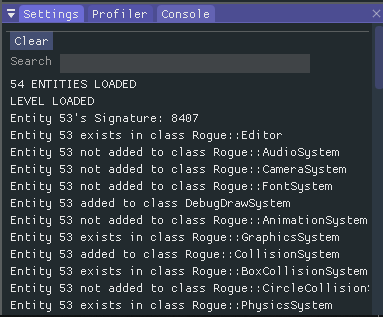


Figure 12: structure of Console

The console contains all the debugging information in the engine and editor. Whenever a user adds a component or game object in the scene, the console will update all the relevant information. This assists in both the technical and the design side in debugging any errors that may occur in the scene.

### Scene loading and saving

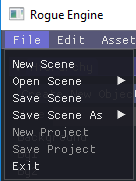


Figure 13: structure of loading and saving of Scene

In the top left corner of the editor, the user will be able to modify the scenes. The user will be able to add and modify the current game scenes. They can also load and current saved scene and save the current edited scene.

# Audio

The sound system handles all the audio input and output and instances where the engine calls for a sound to be played. The audio system consists of different components. For example, the user will be able to attach the audio component to an object and set a distance from where the sound would be played. This allows our designers to set different sounds at different locations of the scene.

# EXTERNAL lIBRARIES

Below are the external libraries that will be used and their usage

|  |  |  |
| --- | --- | --- |
| S/N | Library Name | Usage |
| 1 | FMOD | Audio Loading, Audio Playback |
| 2 | spdlog | For logging of data |
| 3 | GLEW | OpenGL function loader |
| 4 | GLFW | OpenGL wrapper library |
| 5 | SOIL | Library from uploading of textures |
| 6 | Dear ImGUI | Graphical User Interface for level editor |
| 7 | RapidJSON | File serialization |

# Scripting Languages

The scripting languages that we will be using is C++.

# Technical Risks

### Tokenization for saving and loading

One of the risks we might face is the possibility of the player manually editing the data of our components in our json file. Therefore, our team has the intention of running the text through a decoding and encoding system for file reading and writing respectively. This adds an extra layer of protection to our external data. However, due to the complexity the execution of these different elements together posits, it may be executed poorly. If this is not resolved by the second milestone, the encoding and decoding system would need to be omitted entirely.

### Integration of Scripting languages

As the design team is currently only able to provide scripting in C# language, we would need to support scripting for C#. Implementing this would be significantly difficult. Since the LUA language has a fast learning curve and is easier to implement with *luabind* 3rd party library, LUA would be taken into consideration to allow scripting.

# Appendix A

## Flowchart

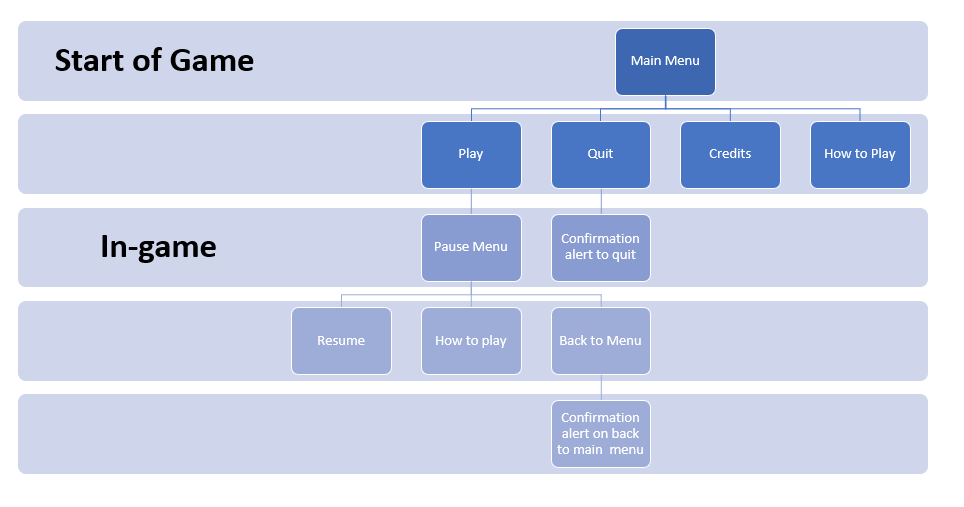


Figure : Game Flow of Exale

## Mockups



Figure : Main Menu Screen

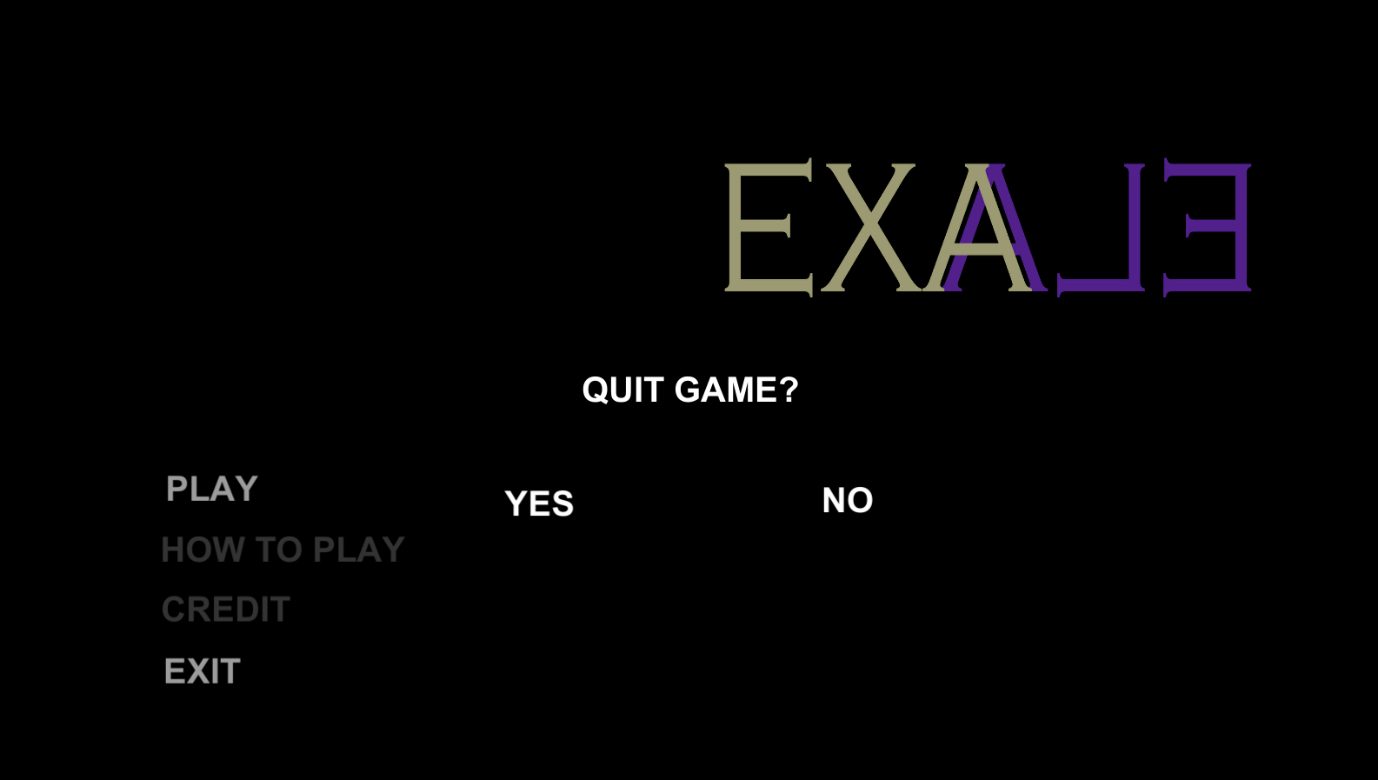


Figure : Exiting from Main Menu

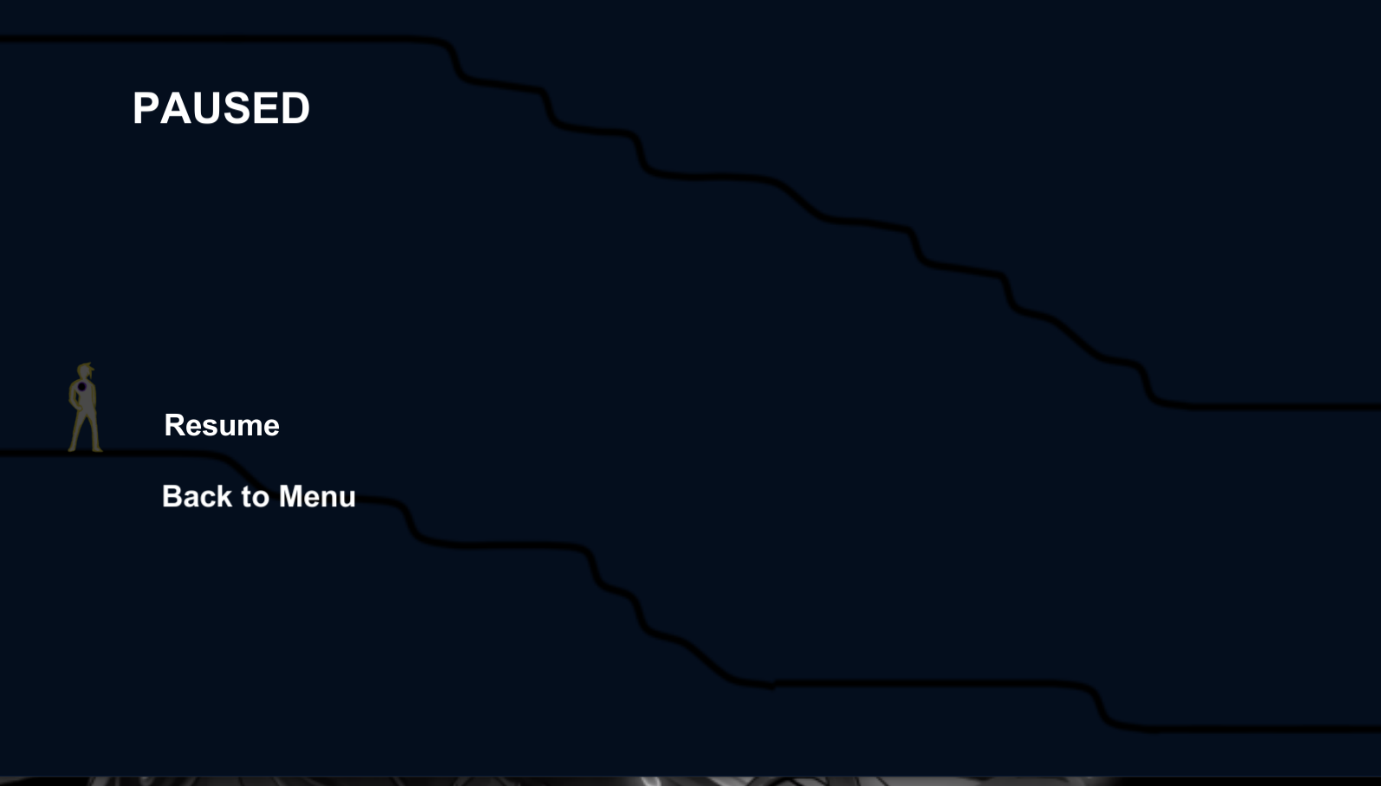


Figure : Game Pause Screen



Figure : Game Pause Screen to Main Menu

# Appendix B

## Art Requirements

All art assets in the game are 2D sprites. These are created by our designers, who will manage the assets accordingly. If art assets are not available at time of testing, especially during the earlier stages, appropriate placeholder sprites will be used in place.

For animated art assets, the asset will be placed into a sprite map for easy reference. If multiple animation assets are needed for a single character, they will be labelled similarly, with an extra segment for animation type.

All art assets will be placed into the Assets folder under subfolder Art. All art assets should be appropriately named and prefixed for ease of search. Files should be compressed if necessary, to avoid an overtly large final executable.

# Appendix C

## Audio Requirements

All audio assets in the game will created by our sound designer.

All audio assets will be placed into the Assets folder under subfolder Audio, before separated into sound effects and music. All audio assets should be appropriately named and prefixed for ease of search. Files should be compressed if necessary, to avoid an overtly large final executable