Feline Felony

Technical Design Document
CSD2401AF23
Fall 2023
Jelyjacs

Student Name	Roles	Champion
CHEN Guo	Programmer	Animation editor and Rendering
GOH Luke Weng Ee	Technical Lead	Engine and Debugging tools
KOH Ashley Hui Xuan	Art Lead	Art Environment
TAN Yee Ann	Product Manager	Physics and Collision
TAN Zi Xuan Elizabeth	Artist	Art Animation
TAY Sen Chuan	Programmer	Serialization, Asset Manager
WOO Jia How, Jonathan	Design Lead	Level Design and Level Editor
YEO Jia Ming	Programmer	Input and Gameplay

Contents

Overall Project

Game Summary

Feline Felony is a 2D Puzzle Platformer that has 2 characters for the player to control. The player has to switch between the 2 characters to clear the obstacles and proceed to the next level until the end of the game.

Overview

The project is written in C++17 to be able to use language features such as inheritance and polymorphism. This project incorporates the component based architecture which makes it easier to include additional features and make changes to the code without many bugs. In this architecture, we use our systems to update each component that the object can have per game loop. We used inheritance as many of our classes are using the same functions, so we have a few base classes and we built upon them.

Engine Elements

Input Implementation

Responsible for handling keyboard and mouse input, offers a user-friendly interface that allows easy access and retrieval of this information for those interested.

Graphics Implementation

Responsible for Image rendering, which is used to bring visual enjoyment to users

Physics & Collision Implementation

Responsible for simulating the physics of all objects and calculating whether objects are colliding.

Windows Implementation

Responsible for initialization window and resolution changes.

Messaging/Event Implementation
Responsible for allowing different parts of the engine to be able to communicate with one another

Assets Implementation
Responsible for loading and unloading all the assets into the game

Logic Implementation
Responsible for every logic of the game such as behaviors of the objects.

Game State Implementation
Responsible for uploading the scene and objects according to the state

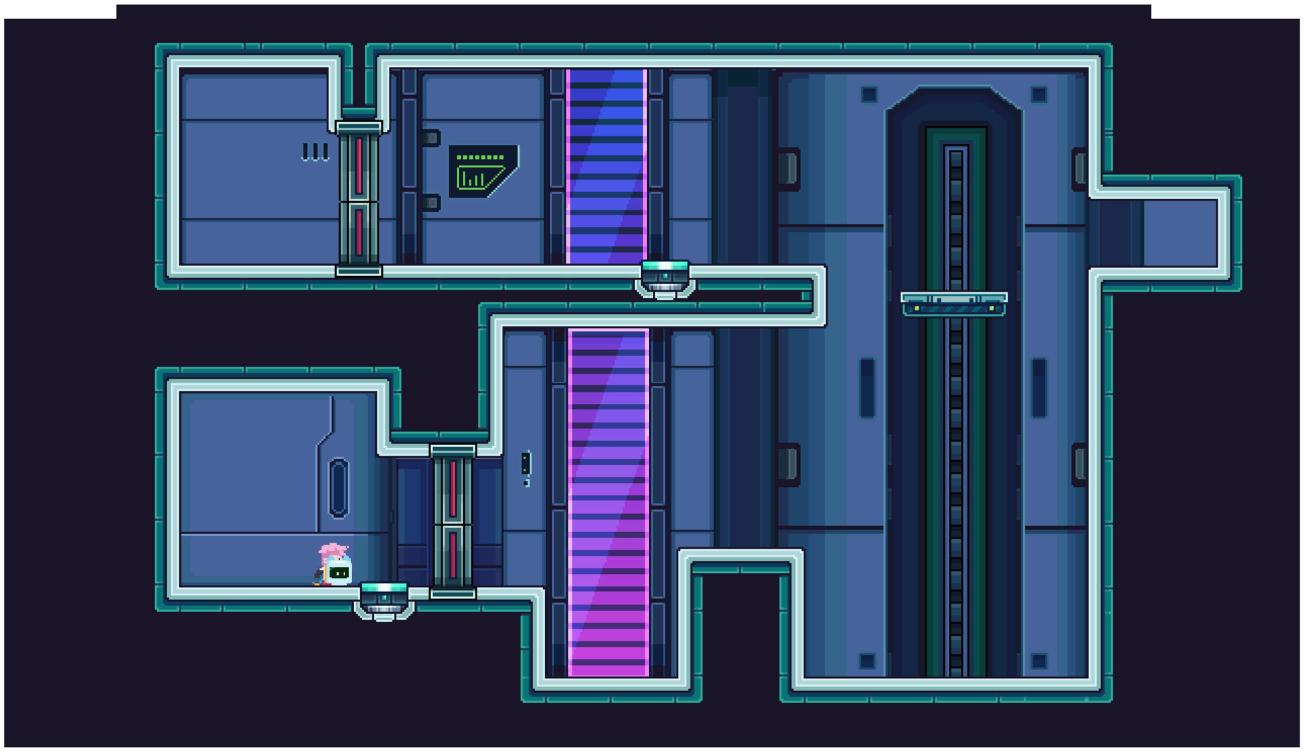
Audio Implementation
Responsible for handling and managing everything audio related such as loading and managing audio files, playing audio, volume control among others through utilizing the FMOD library

Level Editor
Responsible for allowing users to add and remove game objects within the game scene, the creation of new game scenes and the editing of the properties of current game objects, including the logic script attached to them, their textures, their transform position, scale and rotation and the different assets that are used for the game, including art, scripts and prefabs.

Graphics Implementation
In our graphic and window rendering setup, we utilize several key libraries: GLEW, GLFW, and STB. These libraries play crucial roles in the overall functionality of our application. GLEW ensures OpenGL compatibility and functionality, GLFW streamlines window management and input handling, while STB simplifies image loading, making our application more efficient and versatile in creating visually engaging graphics and interactive user interfaces.

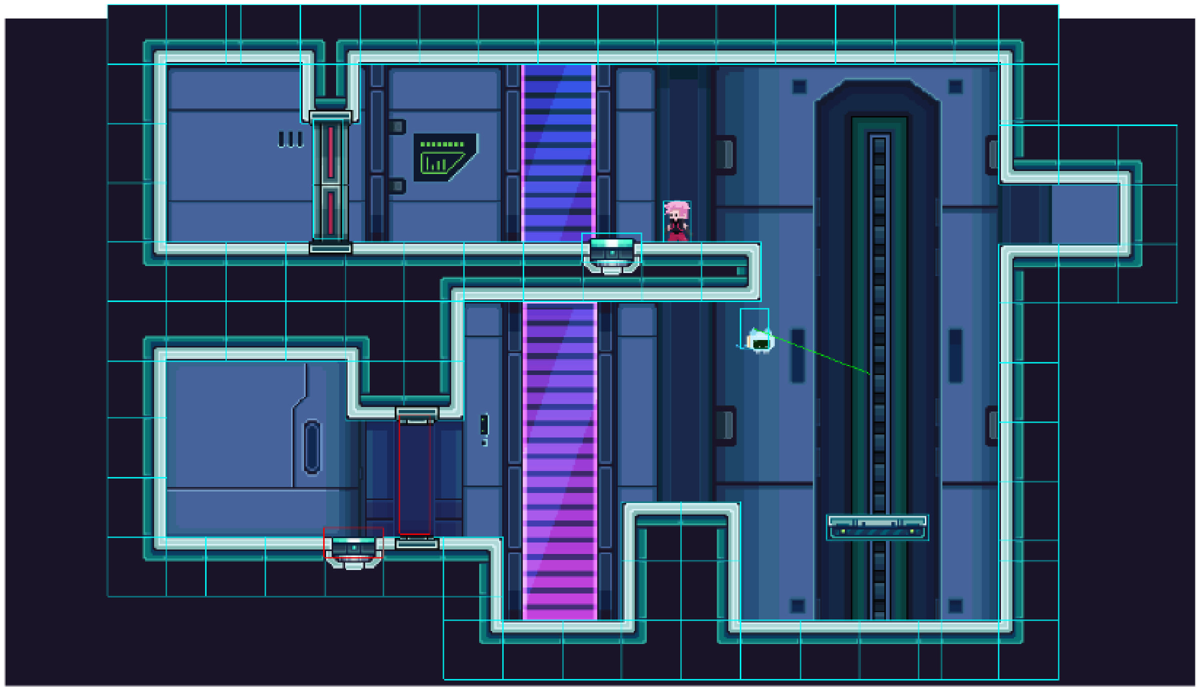
Graphics System
Our 2D game predominantly relies on 2D graphics to facilitate a streamlined and efficient gaming environment. This choice aligns with our objective of delivering an immersive 2D gaming experience. To maintain a consistent presentation, we have opted for an orthographic view, ensuring that objects retain their relative sizes regardless of their position within the game world. The orthographic projection simplifies rendering and aligns well with the 2D nature of our game. For the foundation of our graphics system, we have chosen OpenGL 4.5, a versatile and robust graphics API. This choice empowers us with modern hardware support and feature-rich capabilities, allowing us to deliver high-quality visuals and performance.

The visuals of our game are brought to life through careful asset management. We load background images as textures and render them as sizable quads, creating the backdrop for our game scenes. We also have 2D models, which represent various in-game elements, are loaded as textures and efficiently rendered within the scene.



(background with objects with 2D model)

We have engineered specific components to augment our rendering capabilities. Texture and transform are used to render images in correct position with designed scale, physics and body are used to check object movement and collision in debug mode.

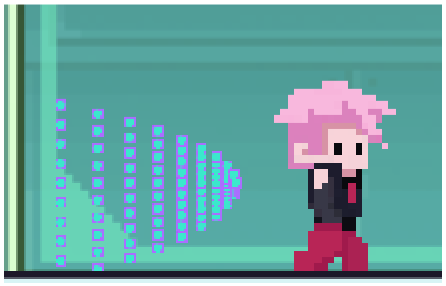


(In debug mode, objects are rendered with rectangles sized to detect collisions and the lines represent velocity)

Our vertex format for rendering 2D graphics is carefully structured, it includes the position, texture coordinates and color. Rendering pipeline encompasses vertex shader and fragment shader.

(Sprite sheet for Finn)

For 2D games, sprite sheet animations are highly effective. These involve creating a single image containing all frames of an animation, which our game will cycle through to create movement.



(Particle effect for movement)

Instancing technique is used for drawing many (equal mesh data) objects at once with a single render call. This will save us all the CPU -> GPU communications each time we need to render a particle system.

At the same time, we also have a camera system to track the manipulated objects.

Transform Component

The Transform Component is essential for controlling the position, rotation, and scale of game objects within the game world. It acts as the cornerstone for placing and moving objects in the game's 2D space.

Texture Component

This component is designed to hold image data that can be mapped onto 2D sprites within the game. Its primary function is to manage and apply these graphical textures to enhance the visual elements of the game's sprites.

Animation Component

This component manages animation by storing texture coordinates for different frames on a sprite sheet, categorized by various animation types. It also includes logic for determining when to halt specific animations, like during actions such as jumping, ensuring smooth and contextually appropriate animation transitions.

Physics Implementation

Physics/Collision System

The Physics system contains functions that handle the physics calculations and collision, such as updating the velocity, acceleration, collision boxes, etc. of each object and checking to see if an object is colliding with another object. The techniques that have been implemented are AABB collision and hotspot. Also, there is a uniform grid implemented to significantly reduce the number of objects to calculate collisions with, lowering the total computational time required. The uniform grid is implemented using 2D std::vector to allow easy resizing of the uniform grid. Inside, each grid contains another std::vector to store pointers to the game objects that are calculated to be inside that particular grid.

Physics Component

The Physics component contains variables to store velocity, mass and a flag to set whether the object is affected by gravity.

Body Component

The Body component contains variables to store the width and the height of the AABB box. If this component is detected, the system will calculate and store the AABB data inside this component using the current position of the object along with the width and height of the AABB box.

Math

The math library is able to create and process 2D vectors, 3x3 matrices, lines, rectangles and circles

Windows Implementation

Window System

The window is created through the use of the GLFW API, which initializes OpenGL and GLEW to provide a valuable graphics context. Additionally, it processes the user input and window closure.

It can also change the window mode and window resolution during the runtime.

Logic Implementation

Game Logic System

The Game Logic System loops through all the objects with behavior components and updates them every frame. It also has cheat code added for debugging purposes.

Base Script Class/Script Manager

The Script Manager is the base class for all our scripts. When a Script is created, the constructor is called and the script is added into the map in the Game Logic System, where it updates all the objects that are using the scripts as logic.

Behaviour Component

Objects that require a logic to be set will have a behavior component that stores the behavior name of the script they need for the object.

Scripts

Scripts inherit from the base Script class which contains the 3 functions, Start(), Update, and Shutdown(). Each type of Object has its own Script and many objects can be using the same script at one time.

Messaging/Event Implementation

Messaging System

The Messaging System allows Systems to communicate with one another by using the Broadcast and the MessageRelay Function.

Event Component

This component handles the game objects that need to be associated in the game logic.

Assets Implementation

File I/O

There are currently two ways of reading from files. One is designed to read text files (.txt) and the other is for reading json files (.json). In our engine, we will mostly be reading json files for our data. The engine is able to read text files through a serialization class using standard library fstream, which will allow other codes to extract a section of text separated by space from the text file. It is also able to utilize the json serialization class which uses the library json.cpp and will allow users to read json files through the class.

There are also two ways of writing to files. Similar to reading from files, the text serialization class and json serialization class are both capable of writing to files as well using the same libraries mentioned above.

Assets Manager

The Asset Manager is capable of reading through a directory and loading all the .png files within as textures. It is also able to load animations (.png), audio (.wav) and prefabs (.json).

 The textures are then loaded and buffered in the GPU which is assigned a unique identifier which points to the texture. This identifier is then stored together with the name of the texture e.g. <Sample.png, id>, in an std::map which will be looked through during texture retrieval when necessary.

The animations work similarly to the textures which use the same principle of storing the data. THe main difference is that these (.png) files are sprite sheets instead of one single texture.

The audio is stored in an std::map which contains a string linked to a FMOD::Sound object. This object is created through the FMOD library which contains the loaded sound (.wav) and stored for later use when necessary.

The prefabs are stored in a std::map which stores a string linked to an object created through the factory for later use when necessary.

Game State Implementation

Scene Manager

The scenes are loaded and saved through the file Scene.cpp. This file contains the functions LoadScene and SaveScene.

LoadScene is a function which uses the json serialization class to read through a json file containing all the list of objects for a given scene. A sample level data in json file are listed in the following way:

```
{
  "SceneName": "example name",
  "Size": {
    "startX": -672.0,
    "startY": -352.0,
    "width": 2176.0,
    "height": 1344.0
  },
  "Objects": [
    {
      "Prefabs": "exampleObject.json",
      "Type": "Transform",
      "Position": {
        "x": -128.0,
        "y": 352.0
      },
      "linkedevent": 8
    }, etc..
  ]
}
```

The scene/level file should contain the minimum. “SceneName” which contains the name of the level, “Size” … which will contain the data for the starting coordinations and level size, and any further list of “Objects” as necessary. The minimum data an object should contain is the “Prefabs”. Any further data is used to modify the clone copy of the prefabs previously stored in the asset manager that will be created and modified from here.

SaveScene is a function that does the reverse of what load scene does. Instead of reading from a json file, it will read the necessary data from the other systems and write it similar to what is shown above into a json file, such that it can be loaded properly again with LoadScene in the future.

These functions are called when necessary, such as loading of a new scene when the player reaches the end of the current level or when a level is selected through the level editor.

There is a SceneManager class that manages the scene state. This class allows the engine to play/pause the game and/or restarts the scene to its initial state by copying the initial state of the game objects back to the object map . It will also be able to change to another selected scene.

Object Factory

The object factory is a class responsible for the creation and management of the objects used in the game. The main functions of the factory are the createObject and cloneObject function. These functions are used for the creation of objects in the game. createObject function is mainly used during the creation of the prefab lists in the asset manager while the cloneObject function is used almost everywhere else when object creation is necessary. The createObject function works by reading a .json file and creating the components necessary for the object, while the cloneObject function will look through the asset manager's prefab list and copy the object stored there.

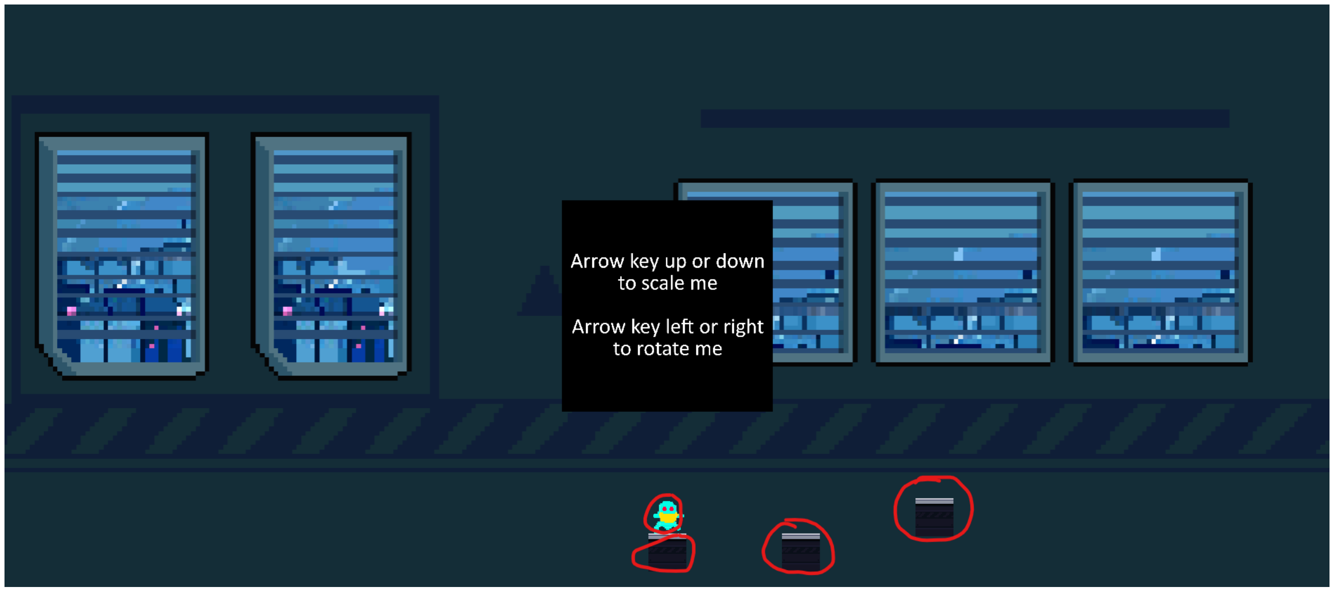
There are also other helpful functions such as the Update function which maintains the order of ID in the list of objects and getPlayerObject which allows other systems to easily access the player object and more.

Game Object

A game object is a collection of components. Components are classes that inherit from the same Component class that give every game object properties in the game with the private variables of the component classes being edited to fit the needs of the game. For example, if we add the Texture component to a game object, we can use that game object to draw a sprite in the game. A game object is referred to as an Object class instance in code. Every game object has a private map that contains all the components that make up that game object, a private long that represents the id of that object so that it can be more easily called upon, a GetComponent function to return a pointer to a certain kind of component if it is present in the private map which allows for the variables of that particular component class instance to be both read and edited, an AddComponent function to add new components to itself, an Initialize function that will set all the private variables of all the components to default values and a GetId function that returns the id of that particular game object.

When the game loop starts, we add component creators to the game object factory. A component creator is a class that helps to create new pointers to existing components with said pointer being used by the game factory to add the game component to the private component map of the game object.

Once we have a game object, we can use the values of those game objects to detect collision, draw sprites, move sprites according to player input, etc etc.


Arrow key up or down to scale me
Arrow key left or right to rotate me

As an example, all the sprites circled in red as well as the background here are game objects that have been created with the Texture component that contains the name of the png that will be used to draw them.

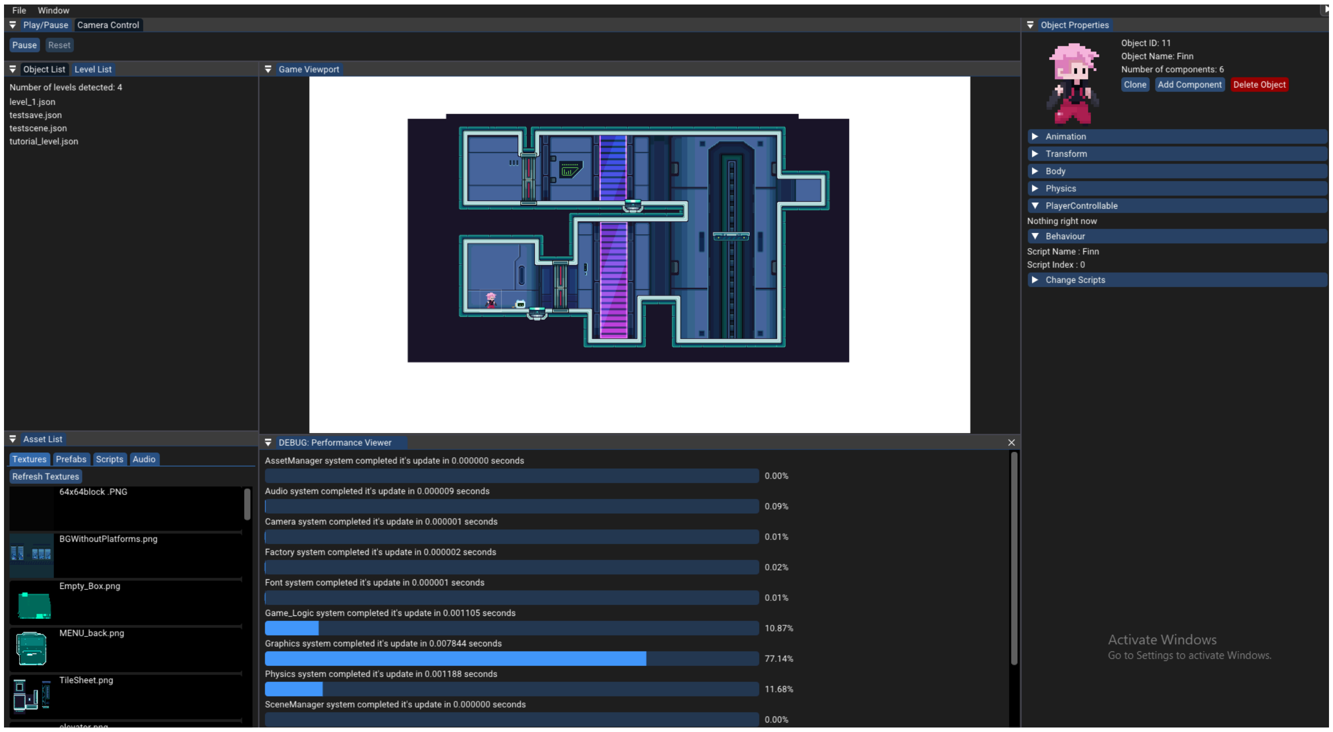
Input Implementation

Input Manager

The input manager uses the GLFW API to poll for key presses, and mouse callbacks which uses an event based system. An array is used to keep track of the states of all keys(pressed/released/held down) that we chose to define. The input system is updated at the start of the game loop to poll for key presses. For every key pressed for the game loop, the key state will be updated into the array, so that users of the input manager can simply check for whatever key that they are interested in if it is pressed/released/held down. The position of the mouse is also updated whenever it moves.

Level Editor Implementation

Level Editor



Number of levels detected: 4
level_1.json
testsave.json
testscene.json
tutorial_level.json

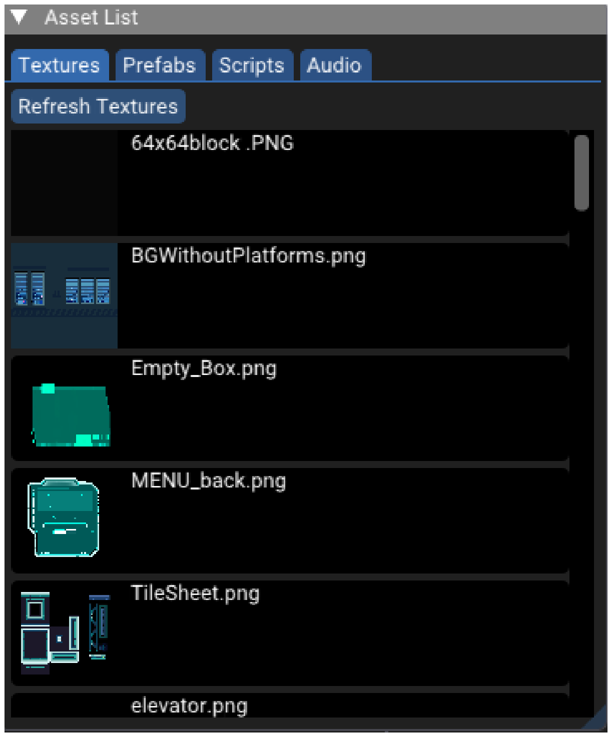
Object ID: 11
Object Name: Finn
Number of components: 6

Nothing right now

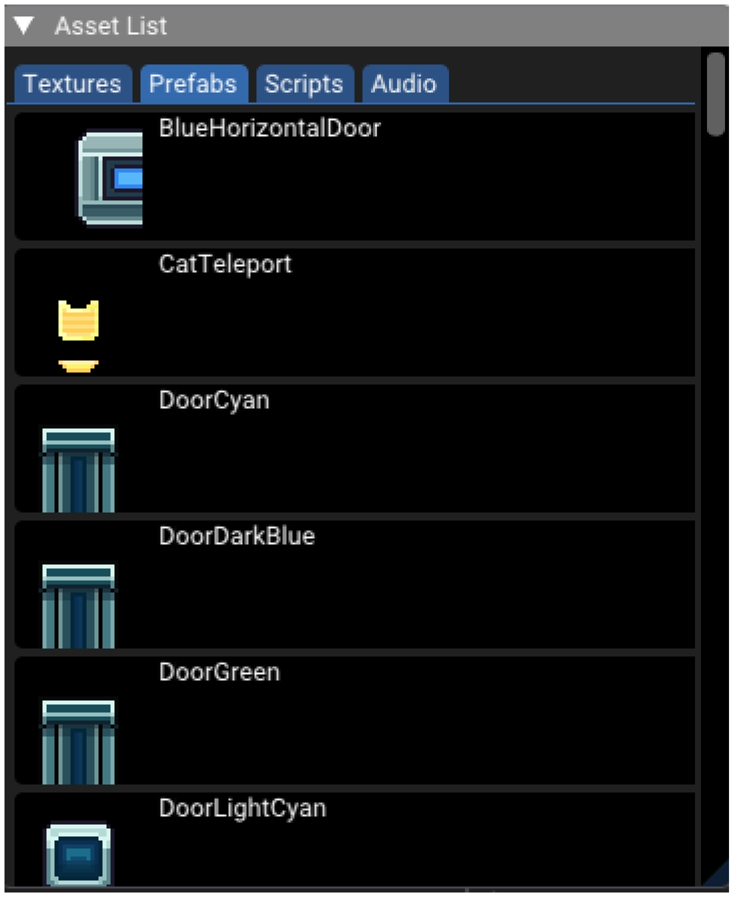
Script Name : Finn
Script Index : 0

```
AssetManager system completed it's update in 0.000000 seconds
                                                        0.00%
Audio system completed it's update in 0.000009 seconds
                                                        0.09%
Camera system completed it's update in 0.000001 seconds
                                                        0.01%
Factory system completed it's update in 0.000002 seconds
                                                        0.02%
Font system completed it's update in 0.000001 seconds
                                                        0.01%
Game_Logic system completed it's update in 0.001105 seconds
                                                        10.87%
Graphics system completed it's update in 0.007844 seconds
                                                        77.14%
Physics system completed it's update in 0.001188 seconds
                                                        11.68%
SceneManager system completed it's update in 0.000000 seconds
                                                        0.00%
```

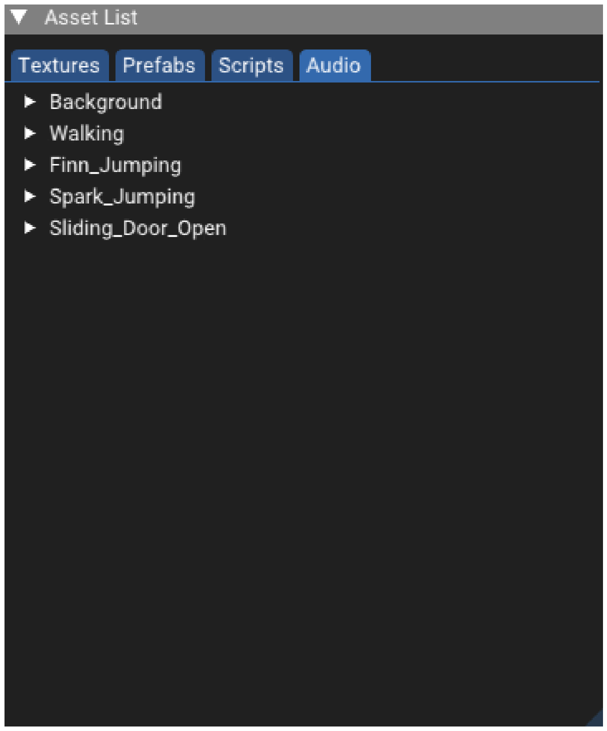
Using the ImGui library, we created menus that display various kinds of information necessary for the creation of a game and editing the game objects within the scene itself.



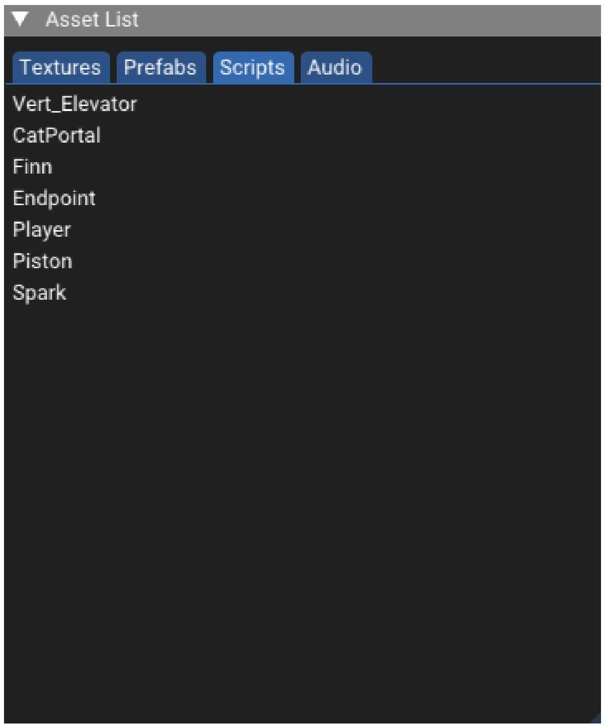
This Asset List menu lists all of the different types of assets used in the creation of a level. Textures refers to pngs that the game objects are drawn as in the game scene. It is possible to change the texture of a game object, thereby changing how it looks in the game scene. This can be done by pressing the change texture button on a Texture component for a game object



Prefabs are pre-done game objects that can be easily added to the game scene by clicking and dragging them into the scene. The object will be instantly created once done.

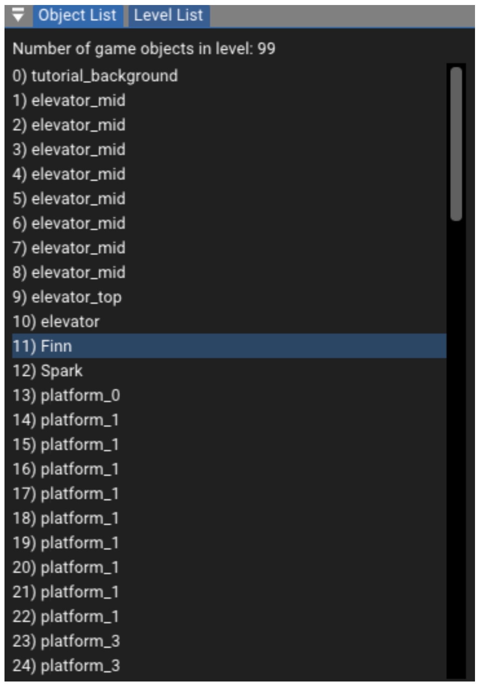


- Background
- Walking
- Finn_Jumping
- Spark_Jumping
- Sliding_Door_Open

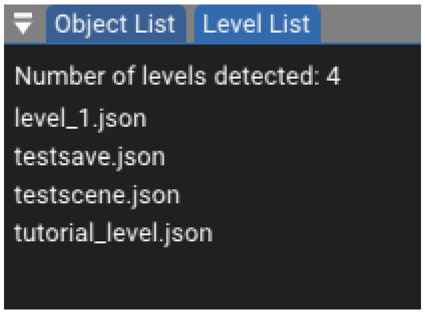


Vert_Elevator
CatPortal
Finn
Endpoint
Player
Piston
Spark

Scripts is the code that certain game objects will follow once attached to it. Audio represents wav files that will be used in the game in various ways like for SFX or BGMs.

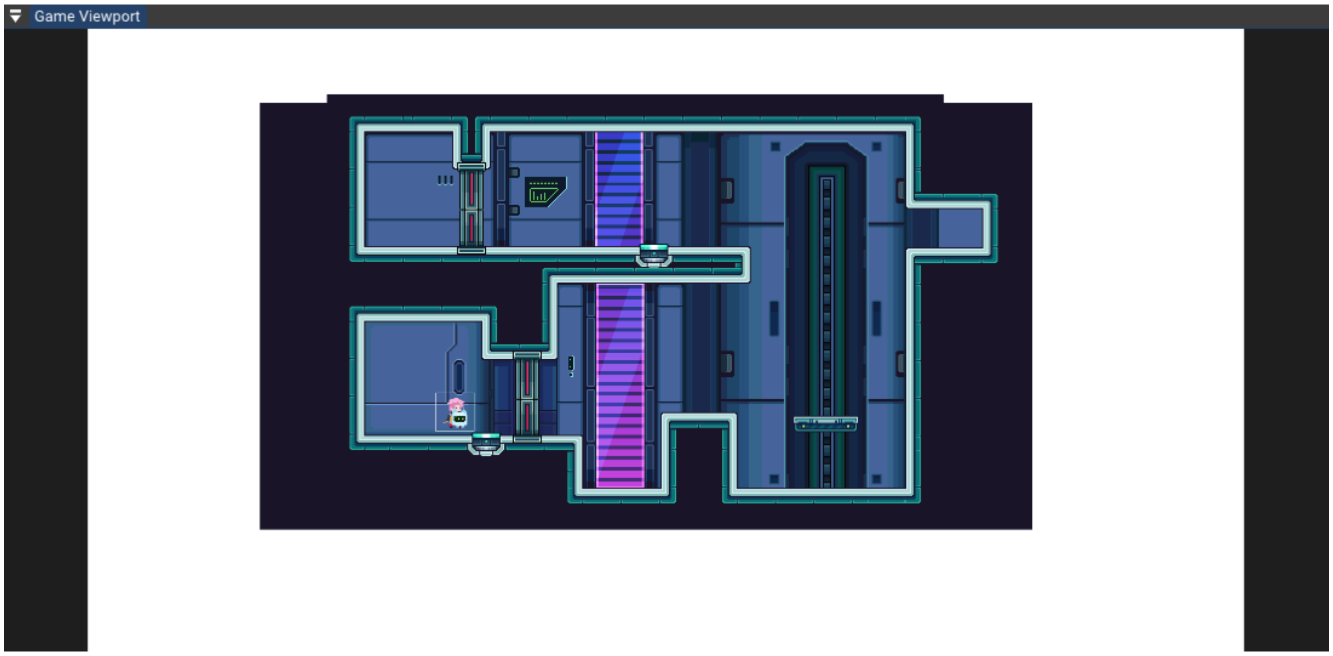


```
Number of game objects in level: 99
0) tutorial_background
1) elevator_mid
2) elevator_mid
3) elevator_mid
4) elevator_mid
5) elevator_mid
6) elevator_mid
7) elevator_mid
8) elevator_mid
9) elevator_top
10) elevator
11) Finn
12) Spark
13) platform_0
14) platform_1
15) platform_1
16) platform_1
17) platform_1
18) platform_1
19) platform_1
20) platform_1
21) platform_1
22) platform_1
23) platform_3
24) platform_3
```

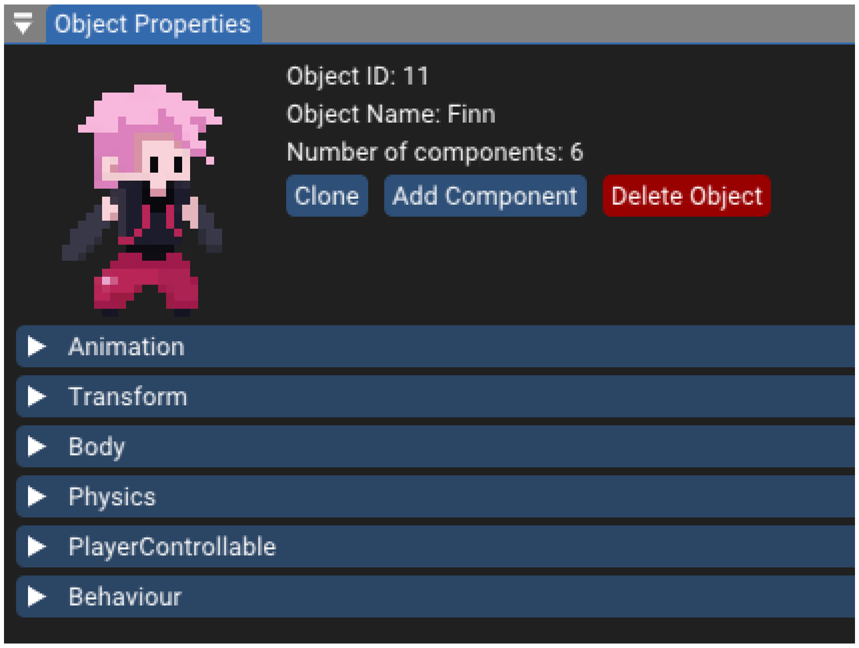


```
Number of levels detected: 4
level_1.json
testsave.json
testscene.json
tutorial_level.json
```

The Object List menu shows all the game objects that are loaded into the current game scene. Users can select them with a mouse and view their components. The Level List menu shows all the json files that are used to load levels into the engine. By clicking on one of the json files there, it is possible to load a new level into the engine.



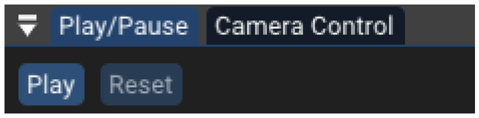
The Game Viewport menu shows a level of the game. Users can play through the game through it, the effects of game objects interacting as they change their properties and components and the structure of levels loaded from json files.

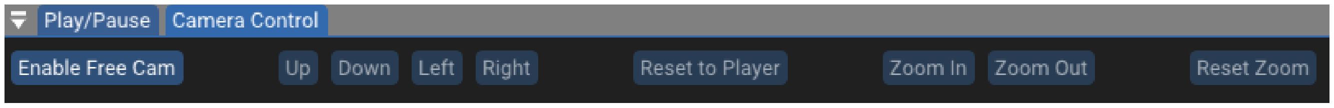


Object ID: 11
Object Name: Finn
Number of components: 6

The Object Properties menu only displays information when a game object is selected. Once a game object is selected, the menu will display the game object's ID, name and the number of components that object possesses.

In addition, users can click on the drop down menus with the name of a particular component in order to view the values of that particular component and change them. By pressing the correct button, it is also possible to clone a game object, add more components to the object or delete the object from the scene entirely.





The Play/Pause menu allows the user to pause a game scene in play and play a game scene that's paused. The Camera Control menu allows a user to move the camera within the game scene itself. Enabling free cam allows the camera to move from its fixed position with the rest of the buttons in that menu.

Coding Methods

We did not set in place any naming conventions. For locations of where each file should go, we split them between header files and source files. From there we classify them into their groups such as components, scripts eg.

Debugging

For Debugging, we print output on to console such as initializing information or some updating information to check on whether the aspect of the engine is working and passing information properly. We also can see the performance of each system in the editor.

Third-Party Tools

Visual Studio 2022

Visual Studio is a robust development tool, offering a one-stop solution for the entire software development lifecycle. It's an all-encompassing integrated development environment (IDE) that enables you to write, modify, debug, compile, and deploy your applications. Beyond basic code editing and debugging functionalities, Visual Studio comes equipped with compilers, autocomplete features for coding, version control systems, various extensions, and numerous other tools designed to improve each phase of the software creation process.

Github

Used for Version Control as well as a backup repository for the project

OpenGL

Used for rendering 2D vector graphics. It's widely used in applications ranging from video games to CAD software.

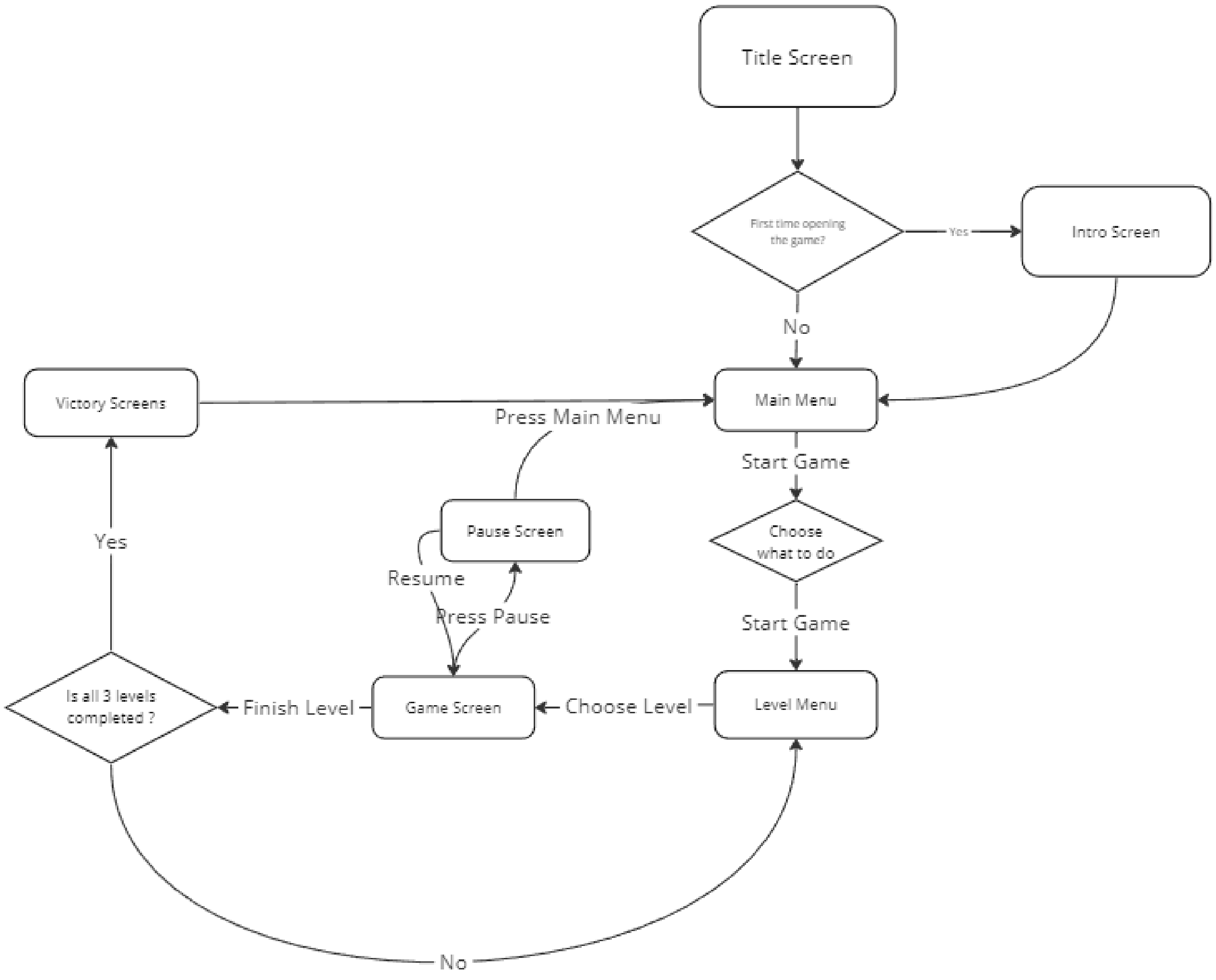
OpenGL Extension Wrangler Library

GLEW is a cross-platform open-source C/C++ extension loading library. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. It helps in managing OpenGL extensions in a more straightforward manner.

Appendices

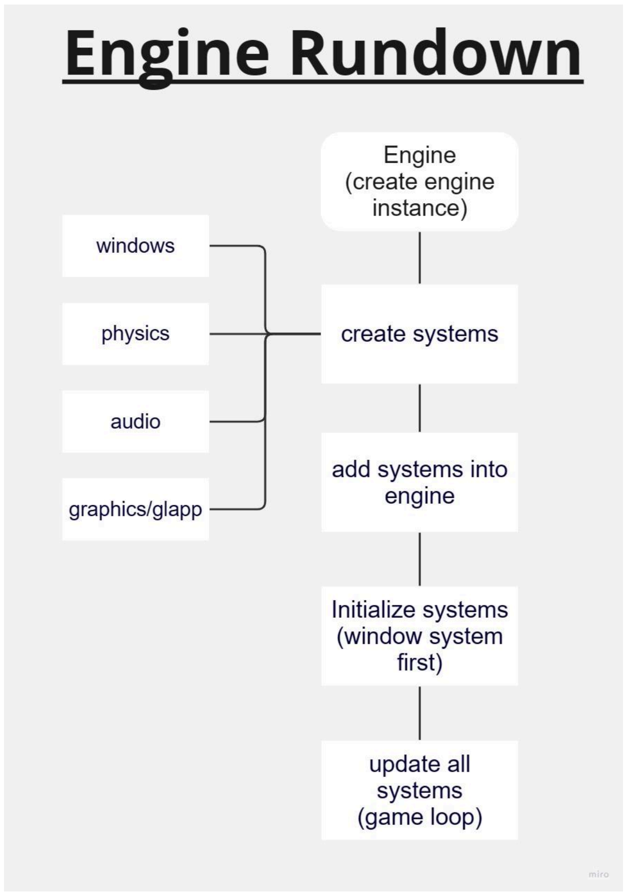
Appendix A: Interface Flow

Interface Flowchart

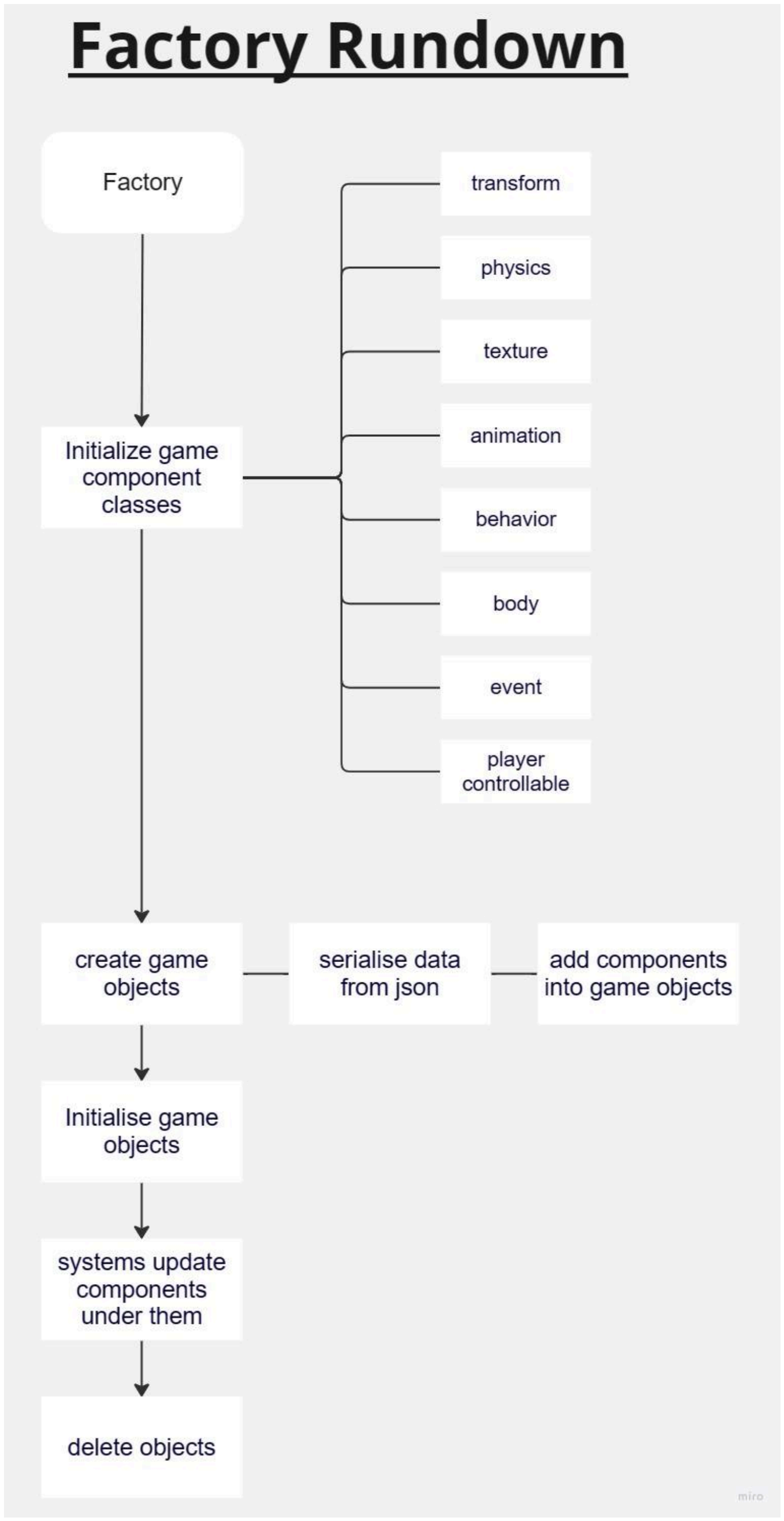


Appendix B: Engine Architecture Flow

Engine Flowchart


Engine Rundown
Engine (create engine instance)
windows
physics
audio
graphics/glapp
create systems
add systems into engine
Initialize systems (window system first)
update all systems (game loop)
miro

Object Creation Flowchart



Appendix C: Art Requirements

Naming Convention
Due to the editor, there is no real need for any naming convention.

Accepted File Format
Only PNG Files accepted

Additional File Requirements
PNG size is in multiplier of 64 pixels (Tiles used in the game are 64x64 pixels). In a sprite sheet of example: 192 x 64, the texture in each of the 3 frames of 64, should be centered. If a sprite sheet contains an opening animation, the animation should go from close to open (left to right)

Sources
Our team's BFA students, Elizabeth and Ashley.

Appendix D: Audio Requirements

Naming Convention
There are no naming convention, just try to name the audio as close to what it contains as possible.

Accepted File Format
Only files in .wav are accepted

Sources
Metadigger, and our Sound Professors, Prof. Vuk Krakovic and Prof. Gavin Parker