Refraction

GAM200 Technical Specification

Sophomore Game Project

Fall 2023

Team Vyv

|  |  |
| --- | --- |
| Programmers: | Designers: |
| Tyler Dean (Producer, Graphics Programmer) | Zan Moffat (Design Lead, Puzzle Design) |
| Michael Howard (Tech Lead, Systems Programmer) | Siddharth Mahesh (Level Design) |
| TayLee Young (Engine Programmer) | Songyang Cai (Narrative Design) |
| Wang HeTong Louis (Audio/Physics Programmer) |  |
| Thomas Stephenson (Graphics Programmer) |  |

Contents

[**Architecture Overview** 3](#_Toc149739746)

[**Graphics Overview** 4](#_Toc149739747)

[**Particle System**: There is currently a fully functioning particle system within the graphics pipeline. 4](#_Toc149739748)

[**Physics Overview** 4](#_Toc149739749)

[*Collision* 4](#_Toc149739750)

[**Player Controls** 5](#_Toc149739751)

[**Behavior** 5](#_Toc149739752)

[**Debugging** 5](#_Toc149739753)

[**Coding Methods** 6](#_Toc149739754)

[**Version Control** 7](#_Toc149739755)

[**Tools** 7](#_Toc149739756)

[**Editor Implementation** 7](#_Toc149739757)

[**Scripting Languages** 7](#_Toc149739758)

[**Technical Risks** 7](#_Toc149739759)

[Appendices 8](#_Toc149739760)

[Appendix A: Art Requirements 8](#_Toc149739761)

[Appendix B: Audio Requirements 8](#_Toc149739765)

# **Architecture Overview**

**Engine:** An object that manages BaseSystem and its children.

* **Base System:** A base class that all other systems inherit from, serving as an abstraction for other systems.
  + **Scene System:** Manages all scenes, responsible for initializing, updating, rendering, and all exiting necessary for scenes.
  + **Platform System:** Handles platform-specific operations, such as initializing the SDL2 library and managing the window.
  + **Event System:** Manages and handles all events within the system.
    - **Event:** Contains events that can be observed and dispatched within the entire system.
  + **Level Builder:** Uses serialization to handle level creation, management, and easy manipulation of level data within JSON.
  + **Entity Factory:** A “factory” for creating game entities of specific types.
  + **Component Factory:** A “factory” for creating various types of components that can be attached to game entities.
  + **Laser System:** Responsible for creating, managing, and visualizing laser emitter reflectors.
  + **Particle Manager:** Maintains particle objects.

**File I/O:** Handles file input and parsing of various data formats, such as JSON, tile maps, and light data.

**Font System:** Handles and manages fonts using SDL\_ttf.

**Math:** Contains several general math functions for use, such as clamping, lerp, squaring, etc.

* **Matrix:** Contains math functions for matrix usage and calculations.
* **Vector:** Contains math functions for vector usage and calculations.

**Entity Container:** Manages a collection of all game entities.

* **Entity:** A base class for all game entities with various components.

**Component:** A base class for various component types.

* **Transform:** Responsible for managing the translation, rotation, and scale of game entities.
* **Physics:** Responsible for managing physics-related properties and behaviors for game entities.
* **Sprite:** Responsible for managing sprite and animation components for game entities.
* **Behavior:** Provides a framework for handling various entity behaviors.
  + **Behavior Player:** Manages the behavior of the player character, handling input, character movement, and other gameplay-related logic.
  + **Behavior Door:** Manages the behavior of the doors, such as collision and states.
  + **Behavior Mirror:** Manages the behavior of the mirrors, such as collision and reflection.
  + **Behavior Switch:** Manages the behavior of the switches, such as collision and states.

**Renderer:** The core of our graphics. Responsible for rendering lights, objects, and animations on screen.

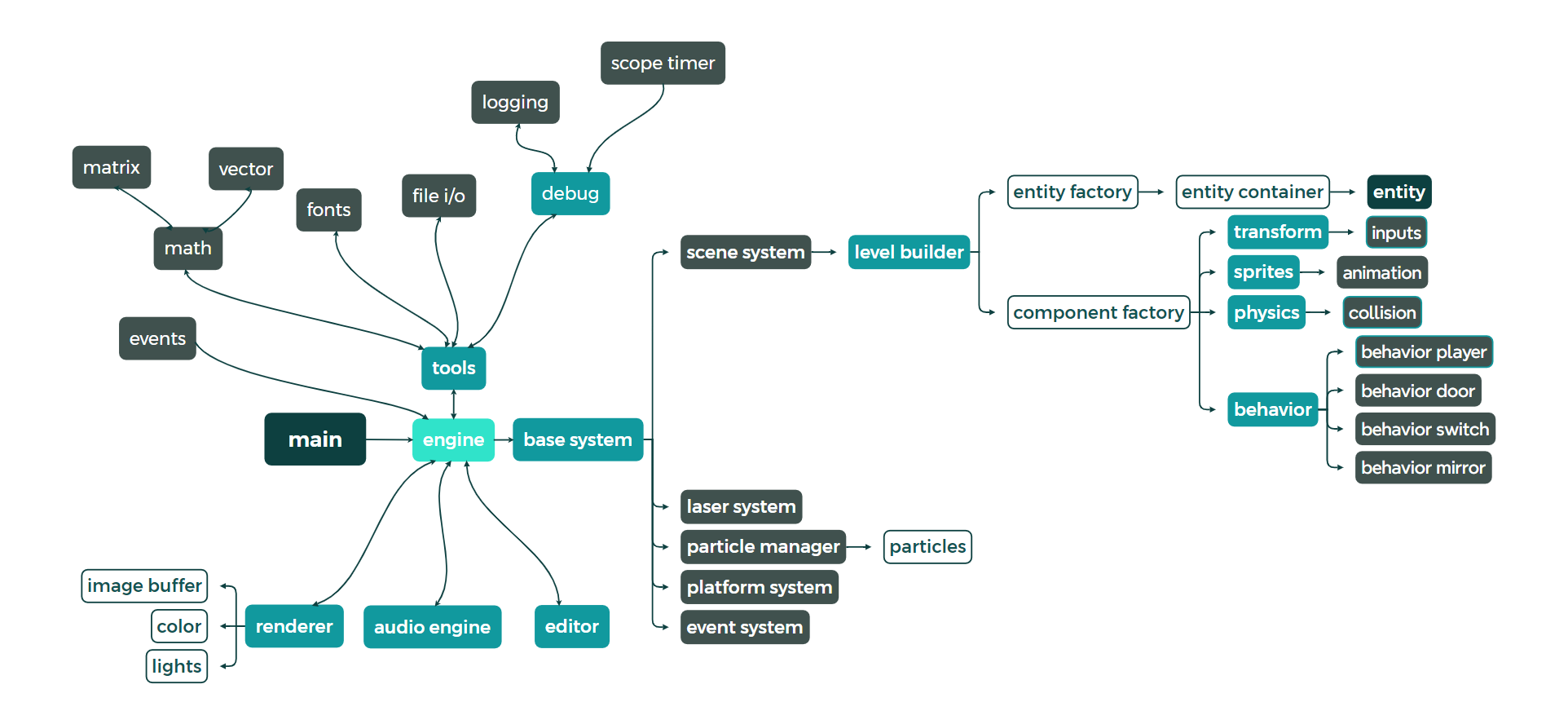
* **Image Buffer:** Responsible for image manipulation.
* **Color:** Allows for setting and getting color components, as well as performing common color operations.
* **Light:** Manages and allows for manipulation of the light sources provided in game.

**Input:** Using SDL2 to handle user input, responsible for checking the state of input devices and responding to input events.

**Audio:** Manages audio-related functions using the FMOD library.

**Editor:** The ImGUI overlay.

* **Input Tracking:** Live overview and tracking of all input events.
* **Entity Tracking:** Overview of existing entities and their variables.
* **Particle Tracking:** Live overview of existing particles, their variables, and active movement.
* **Cheat Codes:** Access to every cheat code from the menu.
* **Scene Management:** Access to the ability to change and reset scenes from the menu.



The core features of the engine and their **logical** attachments.

# **Graphics Overview**

**Graphics Pipeline:**

Each frame a series of color arrays, called buffers, are written to with each buffer representing a layer. The first buffer read in is the tile map buffer. When the scene is loaded the engine reads in the tile set sprite and the tile map data. This data is then used to split up the tile set and load the proper tiles at the proper positions into the tile map buffer. The normal map is then loaded using the same technique and stored in its own buffer. These two buffers remain unchanged for the duration of the scene being loaded. During gameplay the object buffer is cleared and re-written every frame changing the positions of the objects as necessary. As each object is being replaced the engine determines which animation frame to use based on a variable on the animation. Now the engine has reached the lighting stage. First the normal map is translated to be in the correct place relative to the camera. The engine also loops through the particle system and renders each particle at its desired location. Then every buffer is combined starting from the bottom layer to the top layer, with the result being stored in the input buffer. Each pixel in the input buffer is then looped through to calculate the color and intensity of the final lit pixel. The renderer needs to use the light sources, initial color of the pixel, and the normal map to find the correct color and intensity. When the color of the lit pixel is found it is stored in the output buffer. This output buffer is then baked into a texture and sent to the GPU to be rendered by OpenGL with SDL2 handling the window.

**Rendering Pipeline:**

A diagram of a person's face

Description automatically generated

The rendering pipeline is broken up into 3 distinct stages.

**Stage 1:** All of the initial buffers are added together, in the order of lowest to highest in order to achieve a layered effect, then stored into the input buffer.

**Stage 2:** The input buffer is then looped through pixel by pixel. For each pixel the renderer uses each light source, the surface’s color, and normal map the brightness and color of that pixel which are then written to the output buffer.

**Stage 3:** The resultant matrix is then baked into a texture and sent to the GPU to be rendered by OpenGL and SDL2.

# **Physics Overview**

## *Collision*

For Collision, we are using AABB collision. We decided to go for the AABB method because our game is tile-based, with 6 pixels per square. Since our game is already a map of squares, using AABB is the most feasible for our case. Every object has a behavior that corresponds to a reaction. If it collides with something, depending on what the collided object is, a certain response will occur. For example, if the player collides with an object that is considered a “wall”, it will stop moving. AABB checks the min.x, min.y, max.x and max.y of the objects colliding. If the player’s max.x is greater than the wall’s min.x, it means that it is colliding with the wall on the right. If the player’s min.x is smaller than the wall’s max.x, it collides to the left. The same goes for the y-axis.

# **Player Controls**

# Utilizing SDL2, the Inputs class handles and manages various keyboard and mouse inputs. It follows the Singleton design pattern as only one instance exists at any given time and can be considered to be a part of the Observer design pattern as it handles and dispatches input events. It is designed to accommodate a single-player experience.

# **Input Manager**: SDL2 Manages keyboard and mouse events.

# **Behavior**

Uses every part of a game object’s components to give specific responsibilities actions, and reactions.

**BehaviorPlayer:** The core of how the game is perceived and will affect how other game objects will behave. Uses its parent objects to transform and physics to change. Player has collision behavior with objects that has collider in their behaviors. Player is able to collide and move objects, and being able to collide with walls.

**BehaviorSwitch**: Switch will interact with player to move the mirrors. When the player is in the radius of the switch, there will be a prompt appear to indicate that the switch is interactable and “Press E” will appear beside the player. When player press E, a sound will be heard to indicate that the switch is pressed, and the mirror will lerp to its next position.

**BehaviorMirror**: When the player presses E, a bool called OnOff will be switched to true. This variable will be parsed as an argument into a Mirror function called SwitchOn. If OnOff is true, the mirror will lerp to its next position. The positions are stored as an array called pos during initialization, where the values in positions from the JSON file are read and stored in this variable pos.

**BehaviorDoor**: It controls whether the game can be won. When it is closed, player is not able to go through the door and win the level. When it is opened, the player is able to go through the door. To make the door open, player has to fulfill certain conditions.

## **Debugging**

Debugging is incredibly important and used throughout the engine, especially within main systems to ensure clean and bug-free code.

**ImGUI:** Windowed panels display live information of inputs.

**Assertions:** Asserts are implemented within all core systems with proper error checking and logging to verify everything runs smoothly within the underlying engine.

**Console and File Logging:** Both console and file logging can be utilized and customized to display any needed information.

**Profiling:** Scope Timer and the timestamp function in Logging allow for simplistic profiling of code segments.

# **Coding Methods**

**File Naming Conventions:**

Give all files a name that is descriptive and concise.Example *PlayerSprite.*

* **Sprites**: *NameSprite.ppm*
* **Audio Files**: *SoundName.mp3* (or whatever file type is best suited)
* JsonData: *JsonName.json*

**Code Naming Conventions:**

Names that are easy for team members to understand and read ie. no joke names (for final versions). Names must be a one-to-three-word summary of the function, class, or member goals.

**Styling:**

File headers describing the creator(s) intentions and clearly crediting DigiPen and the authors. Comment as necessary and to the author’s future benefit.

**Guidelines:**

* Allow no memory leaks in repository commits.
* Each commit MUST compile cleanly.
* Singleton classes are typically for systems but are allowed in some special cases as long as managed correctly.
* Practice peer reviews.

**Patterns:** Apply patterns for personal practice and wherever beneficial.

# **Version Control**

# **Github:** Github is used by all programmers to push and pull various files that are relevant to the game. Programmers are to ensure that the code is as bug-free as possible, and entirely error-free before pushing any files.

# **SVN:** Used by designers namely to store all assets, files, documents, and anything relevant to the game. All programmers must push to the SVN at least once per week and be encouraged to push after any major changes to the files.

# **Tools**

**Libraries:** FMOD, ImGUI, JSON Modern C++, OpenGL, SDL2

**Other Tools:** Aesprite

# **Editor Implementation**

**In-Game Value Editing:** Input values, entity values, and particle values are monitored using ImGUI.

# **Scripting Languages**

Scripting will be written in C++, as the engine is currently constructed using the same language. No other languages will be required for this project.

# **Technical Risks**

**Performance:** Currently our main tech risk is the performance of the rendering portion of the engine. Currently, it runs well on lower-end PCs, but performance will always remain a challenge. If any performance issues arise, we have various improvements to make. These improvements include more binning, partitioning, and threading of the screen, and in extreme cases migration of the expensive sections to the GPU.

# Appendices

## Appendix A: Art Requirements

## The art assets must be in a PPM format and named in all lowercase with underscores between each word. The art assets must be stored in the assets->data folder.

## Art assets are incorporated into the game by converting to ppm and adding the filename and its position in the game into a JSON folder. For tilemaps, the sprites must be in the form of a sprite sheet and have a corresponding array of the tilemap itself and an array containing what tilemap numbers correspond to tiles with collision.

The art in the game is all from either the Scut tileset\* on itch.io or made by Tyler Dean. We have made sure that the tileset in question is fully free and editable for all non-commercial games. <https://scut.itch.io/7drl-tileset-2018>

## Appendix B: Audio Requirements

Our Audio Engine is using Low-Level FMOD. We have created a folder in Assets that is sorely for audio. In the folder, there are 3 different folders, all for different purposes. We have the Music folder, SFX folder, and Voice-over. Music is for all the background music, that will be looping until we decide to stop it. SFX is for sound effects like footsteps and interactions. Voice-over is for character voice which we might add in GAM250. All the 3 folders will be loaded into the engine separately. They are in .ogg file format rather than .MP3 because of the smaller file size and better sound quality. Our audios are sourced from the DigiPen libraries and converted to .ogg. For all the audio assets, we will parse them into the engine with just one line of code for each audio. For example, if we want to parse in the footsteps sound, we just need to type AudioManager::LoadSFX(“footsteps.ogg”) in the initialization function. As footsteps is an SFX and in the SFX folder, we must call the SFX load function. If the audio is music, the function will be AudioManager::LoadMusic(“music.ogg”). As long as the audio is in the right folders, the AudioManager will be able to find them. Next, to play the sounds, we just have to call AudioManager::Play\_\_\_(“name.ogg”), depending on what type of sound we want to play. Essentially, there are 3 channels: Music, SFX, and Voice. Each channel will play its sound individually, meaning that we can play music and SFX at the same time. Next, there is a stop function, to stop the audio from playing in their respective channels. Lastly, we have a function to set the volume of the audio and increase/decrease of volume for audio. It can be used to set the volume in the settings menu when we implement that in GAM250. Load()->Play()->Stop() We are using the sources from DigiPen Library, in mp3 format and converting them into .ogg format to be used in our game. We have the background noise in a forest environment with dripping of water occasionally. We also chose 9thSense by Sazonoff as the background music because of the instruments and choir. We tweaked it to be of the lowest volume possible so that players will not be distracted by the music as it is supposed to be a background noise.