

Sciophobia

Gam200 Technical Specification

Fall 2025

Team 2B Determined:

Xander Boosinger – Dear ImGui General

Zach Rojas – Dear ImGui General

Veronica Stever – General Programmer

Tyler Andrews – General Programmer

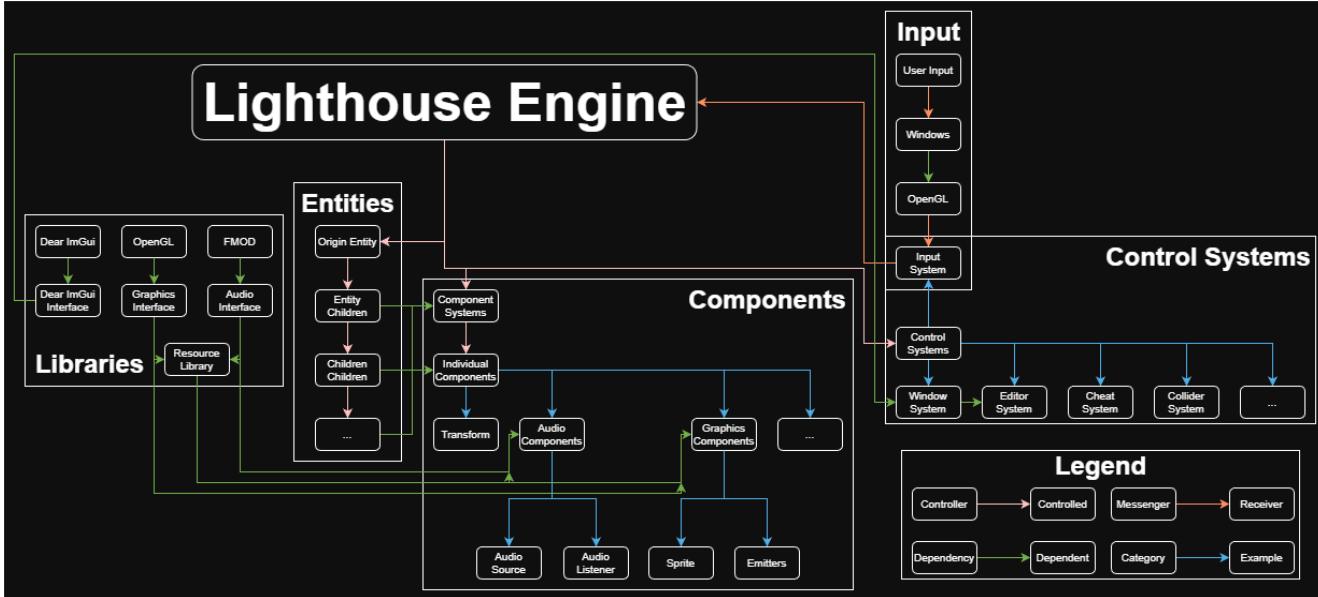
Jackson Miller – Producer

Table of Contents

Architecture Overview	2
Graphics Implementation	3
Physics Implementation	4
Player Controls Implementation	4
Behavior Implementation	4
Debugging	4
Coding Methods	5
Version Control	5
Tools	5
Editor Implementation	6
Scripting Languages	6
Technical Risks	6
Appendices	6
Appendix A: Art Requirements	6
Appendix B: Audio Requirements	7

Architecture Overview

- **Engine:** The ultimate controller of all game objects and systems.
 - **Entities:** Purely relational constructs; contain the information of how components are related through parenting and references to components.
 - **Component List Systems:** Actual owners of components; calls all management functions (Messages, Updates, Renders, etc.) on the contained components, as well as managing lifetimes.
 - **Components:** The containers of game object functionality.
 - **Transform:** Position data and parent-child position inheritance.
 - **Audio Components:** Components that control audio behavior.
 - **Graphics Components:** Components that control graphics.
 - **Behavior Components:** Anomalous components that do anything from walking to firing bullets. The bulk of the actual game logic.
 - **Particle Emitter Components:** Allows for particles to be drawn relative to the owner Entity's Transform component.
 - **Control Systems:** Systems which Control non-Component behavior, such as the editor windows, debug windows, the cheat system, collider system collision checking, etc..
 - **Window Creation System:** Allows for simplified creation of ImGui-based windows.
 - **Editor System:** All in-game objects are displayed and editable through the hierarchy & inspector windows via their serialization/deserialization functions.
 - **Input System:** Receives keycodes from Windows/OpenGL and translates that into Inputs such as Jump or Shoot in order to dictate game events via messaging.
 - **Serialization:** All in-game objects can be serialized and deserialized from file in JSON format.
 - JsonLibrary is the best in-code tool for easily instantiating entities into the scene.
- **Libraries:** Outside code which implements certain features.
 - **Graphics library:** OpenGL/GLFW.
 - **Audio library:** FMOD Core.
 - **GUI library:** Dear ImGui.
 - **JSON library:** JSON for modern C++.
- **Resource Libraries:** Template class that allows for shared referencing of named singleton objects, such as Meshes, Textures, Shaders, or Sounds.



Graphics Implementation

Our engine is a fairly standard 2D game engine, using OpenGL v3.3 and GLFW v3 for our graphics rendering. OpenGL uses buffer drawing to render Sprites, which will be bundles of data containing an instance's mesh, texture, shader, color, and current texture offsets.

Meshes, textures, and shaders are all loaded as unique Resources, and do not live on-Sprite. They are deleted only at the end of the program.

Animations are handled via spritesheets, and assume an appropriate mesh is being used with an appropriate texture.

A major component of our engine is the lighting system, which uses a special shader combined with an Emitter component to handle our dynamic lighting.

Physics Implementation

Player movement is handled with the physics system. When movement inputs are pressed the velocity of the player object is set to a specific value. The physics system uses the Improved Euler method of integration to calculate position, velocity, and acceleration.

Collision detection is handled with the collider system and supports line and circle colliders. Collision handling is done through an event system, where the collision system detects the collisions and then sends an event to the involved entities which then passes that event to their components to handle specific collision resolution. For collision resolution physics, the objects are pushed away from each other until they no longer collide. Spatial partitioning will be used to handle collision detection with many colliders.

Player Controls Implementation

Currently only keyboard and mouse controls are supported.

We use Windows routed through OpenGL/GLFW for our inputs, which are received by InputSystem and converted into an Input, which is an enum representing a particular action, such as “jump”. The mappings from keycodes to Inputs can be found in “DefaultInputMappings.txt”. This Input is then passed to each system as a message, to do with as they will.

There is no multiplayer planned for this project, and the only thing which will happen when an input device is disconnected is that that device will cease communicating with the program.

Behavior Implementation

All behaviours are components of entities, but specifically are for gameplay-based purposes. Our behaviours are basic if-statement chains; pathfinding is done by having the tilemap declare given tiles as “walkable” (or not) and the direction from which it came from (the player being the source); then, whatever object is trying to pathfind simply references this map and follows the arrows. There are no current plans for scripted sequences.

Some simple behaviors we have are PlayerInput (to handle player movement, as well as marking which entities are “players”), and Tracker, which allows the entity it is on to move towards a target transform.

Complex behaviors for enemies/bullets are planned, although a concrete solution is not fully thought out yet. These will most likely be implemented by using either a finite state machine or behavior trees.

Debugging

There is no in-game debug console. There is debug drawing as well as a trace log with message levels, and we simply use `<cassert>` for our asserts. Our in-game performance viewing is only stopwatches tracking how long each system is taking to update. Most variables can be viewed and edited in real time in the runtime editor. Most debug controls we currently have are also enabled and disabled through similar windows, as well as keybinds.

Coding Methods

All game files go in the NightLight folder.

All component files go into the NightLight/Source/Components folder.

All enum files go into the NightLight/Source/Enums folder.

All semi-reflective files go into the Pre-Build folder.

All files which both NightLight and Pre-Build reference go into the SharedDependencies folder.

All cpp/h file pairs should be named the same, and should prefer code to go into the cpp file.

All other file-related concerns are up to the individual.

Version Control

We use GitHub for our version control, with our branches being LastKnownGoodBuild->main->dev->[individual production branches] in decreasing order of security; LastKnownGoodBuild should be as clean as can be, and carefully kept taken care of. Main should almost always be correct and safe, but may have not been vetted fully. Dev should be buildable, but not necessarily correctly functioning or crash-safe. Individual production branches can do whatever they want, but should seek to synchronize with dev often.

Tools

Editor: Visual Studio 2022, visual studio code for JSON editing.

Other tools and programs: Tiled for map creation, Blender for art assets, RenderDoc for debugging, Reaper and MuseScore to edit sound effects.

Libraries: IMGUI for editor windows & widgets, FMOD for audio, Soundly for sound effects, OpenGL and GLFW for graphics.

Editor Implementation

We use IMGUI to spawn editor elements, but access the API through the in-house WindowCreationSystem, which stores window state between frames and manages more complex versions of the base IMGUI widgets known as WindowObjects.

Entities and components are created by typing in their filepath/name in the appropriate input box in the Inspector window. Entities currently cannot be deleted in the editor, but components have a deletion input box similar to and beside the box to add components. The only copying capabilities at the moment are those afforded by serialization and deserialization.

Entities can be moved or rotated by typing in the Transform section in the Inspector window, in the same way as any other component field. There is currently no archetype system, other than that of different entities being able to be found in separate json files.

Scripting Languages

Our project is built purely in C++. JSON files are used only for storing data. We do not have any plans to further support any kind of scripting.

Technical Risks

1. Advanced lighting system. We plan to implement an advanced lighting system with shadow casting that interacts with the environment. To mitigate this risk, we plan to seek help soon if progress is slow, and if we are unable to make sufficient progress we will drop this system and stay with the current lighting system.
2. Spacial Audio. We plan to implement directional and spatial audio. To mitigate this risk, we plan to seek help soon if progress is slow, and if we are unable to make sufficient progress we will drop this system.

Appendices

Appendix A: Art Requirements

Art assets have no current naming conventions, although only PNGs are tested and known to work. Assets are created using whatever the individual has at hand.

Sciophobia

To add an asset, simply place it into the Assets folder, in the appropriate subfolder (Textures). Then, either enter its name (with extension) into a location that supports as such, or in code call
ResourceLibrary<assetType (Texture)>::Get([assetName])

We have no plans to request for a BFA to help with art assets and are giving the responsibility to one of our programmers.

Appendix B: Audio Requirements

Audio assets need only be placed in the Assets/Sounds folder, and be accessed via
ResourceLibrary<Sound>::Get([filename excluding Assets/Sounds]) in order to be used. The file formats should be .wav or .ogg.

The creation of audio assets could take many forms, be it homemade or a library sound, and will be processed (if needed) most likely in Reaper.