## horizontal line



Team LFTCLK Game Engine Technical Design Document

Fall 2022

**─**

Chris Fitch | Producer, Audio & Gameplay

Vance Howald | Lead Developer, Engine Designer

Abhijit Zala | Graphics Engineer

Rohit Punjabi | AI Developer, Scripting, Gameplay

# 

[**Architectural Overview**](#_p9qv3bugyzrt) **2**

[**Graphics Implementation**](#_8ba0x1yghllt) **3**

[**Physics Implementation**](#_6ax5xojjkadw) **4**

[**Player Controls Implementation**](#_ek4q1j5abo0) **4**

[**Behavior Implementation**](#_qg7ea5o0xtp) **4**

[**Debugging**](#_wlwbmmb4aunk) **4**

[**Coding Methods**](#_g5lpw8mv3agx) **5**

[**Version Control**](#_mqgokto4138z) **5**

[**Tools**](#_kh7bfct6os0h) **6**

[**Editor Implementation**](#_2p674djbau6e) **6**

[**Scripting Languages**](#_otw9qn1mz8us) **6**

[**Technical Risks**](#_q0gsmybn6ple) **7**

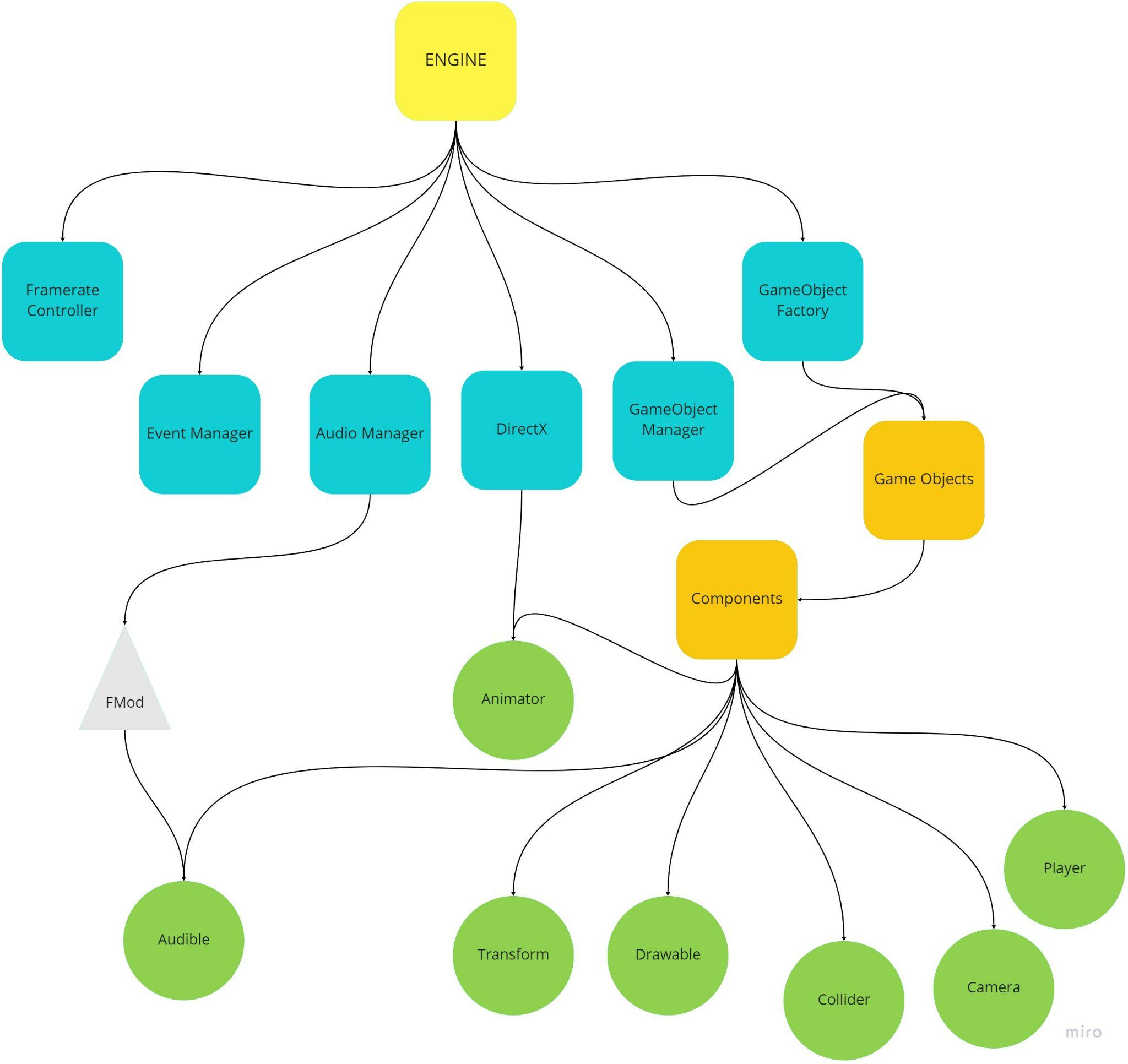
[**Appendices**](#_soymz0iprda0) **7**

[Appendix A: Art Requirements](#_u6w8bxkook8e) 7

[Appendix B: Audio Requirements](#_k50mx0vj9uxw) 7

# Architectural Overview

###### Figure 1 - LFTCLK Engine Technical Architecture Diagram



The game engine initializes in the following order:

1. The game window is created through SDL.
2. The Graphics singleton is initialized and bound to the window using DirectX 11 and DirectXTK.
3. The Audio Management singleton is initialized, which in turn initializes FMOD.
4. ImGUI is initialized.
5. The game loop starts.
6. A JSON containing a list of all prefabricated objects, including their tags and paths to their JSON archetype files, is loaded in and then passed to the game object factory.
7. The game object factory goes through each of the game object’s components and adds it to a new game object instance. The components developed so far are as follows:
   1. The transform component, which stores position and velocity information.
   2. The drawable component, which communicates with the graphical singleton, which in turn communicates with DirectX to create graphics.
   3. The collision component, which detects collision between objects. This collision is used by other components to determine effects, such as stopping an object against a block or causing a sound effect.
   4. The camera component, which controls the camera’s position and rotation.
   5. The sprite animator component, which controls what animation an object is using as well as the speed and what images to use.
   6. The player component, which controls player input.
   7. The audible component, which communicates with the audio management singleton, which in turn communicates with FMOD to control the game’s audio elements.
8. The first time this is done for an object, the prefabs can be cloned using the originally-created object loaded in memory. A major advantage of this is that graphical components of the game object can use the same resource memory with a different constant buffer to reposition the object.
9. Once a scene is over, all objects and related components are deleted in preparation for the new scene.
10. When the game is eventually closed, all resources and managers are released.

# Graphics Implementation

We are using DirectX 11 for rendering and the DirectXTK helper library. The DirectXTK library contains a lot of helper classes and also includes a 2D Texture Loader, which uses Windows Imaging Component (WIC) to load textures. Our models as of now are very simple, so we simply store them as JSON and parse it at runtime. We have a Graphics singleton class which abstracts graphics initialization. Graphics singleton class is accessed by the Drawable component for drawing sprites/meshes. The Graphics engine supports resize functionality and uses 4x MSAA by default.

# Physics Implementation

Physics checking is performed on every object that has subscribed to a collision message event and is checked against every object with a collision box, provided both objects are active. Colliders used are AABB and SAT with point collision being added in the very near future. Collision messages are then broadcasted to the two objects and behavior is decided based on their handling functions. The message is passed through all of the Components of the GameObject so multiple Components can act on the message.

# Player Controls Implementation

To check for the input, SDL passes its input events to the input manager, which checks if the event is valid. If it is, the transform component either moves or rotates the game object. The input manager takes care of keyboard, controller, and mouse events. Components can then interact with the input manager and update their game object’s Transform component to reposition itself.

Currently unplugging the controller throws errors, but this will be fixed shortly.

# Behavior Implementation

Maps will be represented as a list of prefabs with their overrides to reposition the objects. Game objects contain lists of components. Each component contains an update function that is called every frame and a starter function that is called on the parent object’s creation.

Enemies will use A\* for pathfinding and patrol areas until they locate the player, at which point they will chase after them.

# Debugging

The engine uses the ImGui library to print frame rate data to the screen. Apart from this it currently does not have any debug capabilities. Before the game is finished, a level editor and collision/velocity information will be provided for better testing and more efficient workflow when designing levels. There will also be button toggles for all these features.

# Coding Methods

Major engine files are kept in the engine’s root directory, such as the event manager and the game object factory. There are also a number of subdirectories:

* Components
  + Keeps files for all of the component types that are used by the base game object, as well as the base component type, the base game object type, and the game object manager.
* Graphics
  + Keeps files for DirectX 11 integration, used by graphical components.
* LUA
  + Keeps library files for using the Lua Scripting Language. Does not contain Lua scripts the team has created.
* Resources
  + Keeps all assets used by the engine. This includes MP3 files for use by FMOD, JSON files defining object archetypes and level loading orders, spritesheets used to render objects, and all Lua scripts.

Alongside the root directory is a Libraries directory, which contains the necessary external libraries that the engine uses, such as SDL and FMOD.

# Version Control

Git is being used for version control. There have been two types of majorly used branches: development environments named after the team member working on them when they are doing more speculative research, and feature-specific branches for completing grounded project tasks. This will drift in the direction of the latter as the final game concept starts to take shape.

Remotely, code is pushed to GitHub. It will eventually be cloned to DigiPen’s own repository, for tracking by an instructor. Final code in GitHub is kept on the main branch, and commits are generally done by merging any commits on main with a development or feature branch, then once said branch is sufficiently integrated with the changes, the combined code is pushed back into the main branch.

# 

# Tools

The project is created in and relies upon Visual Studio 2019 to function. Our graphics engine uses DirectX 11 and DirectXTK for rendering, with ImGUI for UI elements. SDL is used for window creation and window messages such as resizing and input. Our audio engine utilizes the FMOD Core API.

# Editor Implementation

A custom editor has not yet been made but it will behave as follows:

* Objects can be scaled and moved but likely not rotated as our colliders are AABB.
* ImGUI windows will show an object’s transform properties for easy tweaking and feedback.
* Controls will use hotkeys to shift between object operations similar to Blender.
* Objects will be saved in a level json file which contains a list of prefabs the level uses and a list of clones of the prefabs with override values to reposition them.
* As all prefab instances will behave identically only the transform component will likely need to be modified.
* More complicated modifications can be done on the prefab’s json file manually.
* New clones of prefabs can be made by pressing the A key and then selecting which prefab to use, or by pressing D to duplicate the object.

# Scripting Languages

The engine is embedded with LUA 5.4.2 using a LUA dll and using a .a file that contains all the standard libraries required for the engine. LUA will mostly be used for behavior updates of our enemies, and other LUA libraries may be used depending on what has to be implemented.

# 

# Technical Risks

1. Coding the AI for the enemies to navigate around the level and find the player in a way that’s engaging and interesting could be a challenge.
   1. Our fallback is to simply have the enemies home in on the player.
2. Proper audio could be challenging to put in every aspect of the game. A lot of our idea hinges around the audio and visual effects being immersive enough to provide an atmosphere that is properly gritty.
   1. If this does prove challenging, the only alternative is to do the best we can, visually and audibly.

# Appendices

## Appendix A: Art Requirements

Audio will be recorded by the team either at home or using DigiPen resources. We will use Photoshop, Figma, or Gimp for art. Some art may also be fetched from the DigiPen libraries. Image files will adhere to the following standards:

* Names will be lower-case and multi-word file names will be split using underscores.
* Only basic formats will be used, such as PNG, BMP, etc.
* All image files will be kept in the Resources directory.

## Appendix B: Audio Requirements

Audio will be recorded by the team either at home or using DigiPen resources. Some audio may also be fetched from the DigiPen libraries. It will be put together where needed using FL Studio 20, which the audio engineer has a single-user license to use. Audio files will adhere to the following standards:

* Names will be lower-case and multi-word file names will be split using underscores.
* The only format used will be MP3.
* All files will be kept in the Resources directory.
  + Larger files used for background music will be kept in the music subdirectory.
  + Smaller files used for everything else will be kept in the sounds subdirectory.

Information on how the audio will be used by the game, such as looping, compression, event handling, and more, will be kept in the JSON component descriptions on the individual object archetype files.