IZMIR UNIVERSITY OF ECONOMICS
FACULTY OF ENGINEERING
SOFTWARE ENGINEERING

FENG 497 SOFTWARE DESIGN DOCUMENT

Lina Engine

Authors: Inan Evin & Bekir Batuhan Bakır

Supervisor: Kaya Oguz

1. INTRODUCTION

1.1 Purpose

This software design document describes the architecture, system and data design of Lina Engine. It is targeted to provide rationale for the decisions made regarding the system design, as well as the necessary background information. The intended information is given with a hierarchical view of the system of systems, subsystems, modules and the flow of data. The goal is to generate a notion and a general understanding of how Lina Engine works. This notion would consist of the backbone systems and modules inside the engine, overview of the architecture, as well as the data flow amongst them.

1.2 Scope

The scope of the project covers the design and implementation of the core architecture of a game engine. The implementation is intended to be powerful enough to demonstrate the systems that are provided as solutions to known problems, within their mere power. The crucial parts of this project would be the designed subsystems that are aimed to be the alternatives and extensions of the existing solutions, as well as how those subsystems are connected well within the whole system architecture. These subsystems differ from existing solutions and have advantage over them, because they sacrifice ease of use in exchange for a lightweight structure. Hence, the primary goal is the identification of deficits existing in the current game engines and establishing a basis for a lightweight game engine which includes advisable solutions to the said problems.

Another significant purpose of the project is to contribute the open source community with the implementation of the proposed solutions so that the project may be the basis for unique ideas and the development of more complex problems. Achieving this purpose will provide new perspectives on the existing problems and take the initiative for solving them, so that the game development community might increment upon these points.

1.3 Overview

This document includes several sections in order to powerfully grasp the fundamental concepts of Lina Engine. Initially, information about the high level system architecture is given.This information consists of the main systems, their communication and the data flow between them. Furthermore, the design rationale is provided to explain the decisions taken in order to generate the particular design. The information is both given as text, and UML diagrams for better visualization. Afterwards, the descriptions of the necessary data and their flow, as well as the component design is given. Throughout all these titles, the most crucial systems are pointed out with explicit titles for them to better explain their role.

2. SYSTEM OVERVIEW

In order to start describing the architecture, it is necessary to provide background information of the functionalities, context and the design of the project.

Lina Engine consists of of several high level systems that are capable of constructing a solid main game loop structure. The targeted implementation shall cover all the basic functionalities such as, intermediate rendering features like forward and deferred rendering, global illumination, trivial shading, audio and input system integrations, as well as a base physics engine to be used within component flow that would demonstrate how powerful the architecture works. The main functionality of Lina Engine is its folded out layout, which enables the developers to develop games running on a very lightweight framework structure.

Dependency injections through the subsystems of modern game engines is an existing architectural deficit in the majority of open source game engines. Eventually, this causes heavyweight libraries to be forcibly fed into the developers' project environment. This problem is aimed to be removed with the mentioned lightweight framework structure. Thus, the developers would not need to include libraries and frameworks that they don't want to use inside their project. The engine framework itself would run the main game loop at the lowest cost at all the possible times.

Another significant feature that is designed is the plug and play capabilities of the architecture. As mentioned for the lightweightness, it is crucial to involve the developers only with the frameworks that they intend to use. Due to this, Lina Engine would work on the basis of a package manager system. This package management would be implemented on the very low level structure in the architecture, meaning that even the most important systems like rendering engine would be replaceable with an alternative engine if desired. The examples of this package management system can be observed in popular game engines nowadays, however most of them are only implemented at high level gameplay related sides of the architecture. In other words, pluggable packages provided are only simple libraries that would help the developers to implement various gameplay features, the low level subsystems still stay humble with their constantly disturbed architecture due to gradually added new features for market competence.

Lastly, other crucial feature implemented in Lina Engine would be it's wholesome take on game entities. Unlike many other game engines that are aimed at providing game entity systems that are based on easy to develop hard to maintain structures, Lina Engine provides an Entity Component System (ECS) that works on a different basis. Most other entity systems are designed with very fundamentals of OOP structures to provide ease of use to most possible amount of users due to gain more audience. Lina Engine provides a ECS system that does not follow the main principles like encapsulation or direct inheritance, but rather requires an alternative way of development which requires treating game entities as bundles of data rather than individual objects. While many game engines provide grounds to develop this style of entity systems, their pure nature and architectures are not based on this, but rather on the traditional OOP approach. Lina Engine bases it's whole game level architecture on ECS, making it a perfect fit for highly optimized game client that is a complete fit to the frameworks that the client lies upon.

3. SYSTEM ARCHITECTURE

3.1 Architectural Design

Initially, the systems on the high level should be introduced to proceed furthermore into the architecture. These are given in the Figure 1. At the bottom level, Lina Engine has System of Core Systems (SCS). SCS is responsible for initializing and running game instances supplied by the game client, handling the event and time loops within the core, and successful termination. The core systems are mainly connected to MEB (Message Bus), which is responsible for handling the connection between the systems and game client. In order for SCS to successfully respond to the requests coming from the game client, it needs to have a structured access to low level implementations for audio, physics, drawing. All the low level systems are collected into Low Level Framework (LLF) and the SCS communicates with this framework via PAM (Package & API Manager) to increase the modularity of the architecture. All the external utilities needed for LLF and SCS are directly accessed by them through a collection of utility classes.

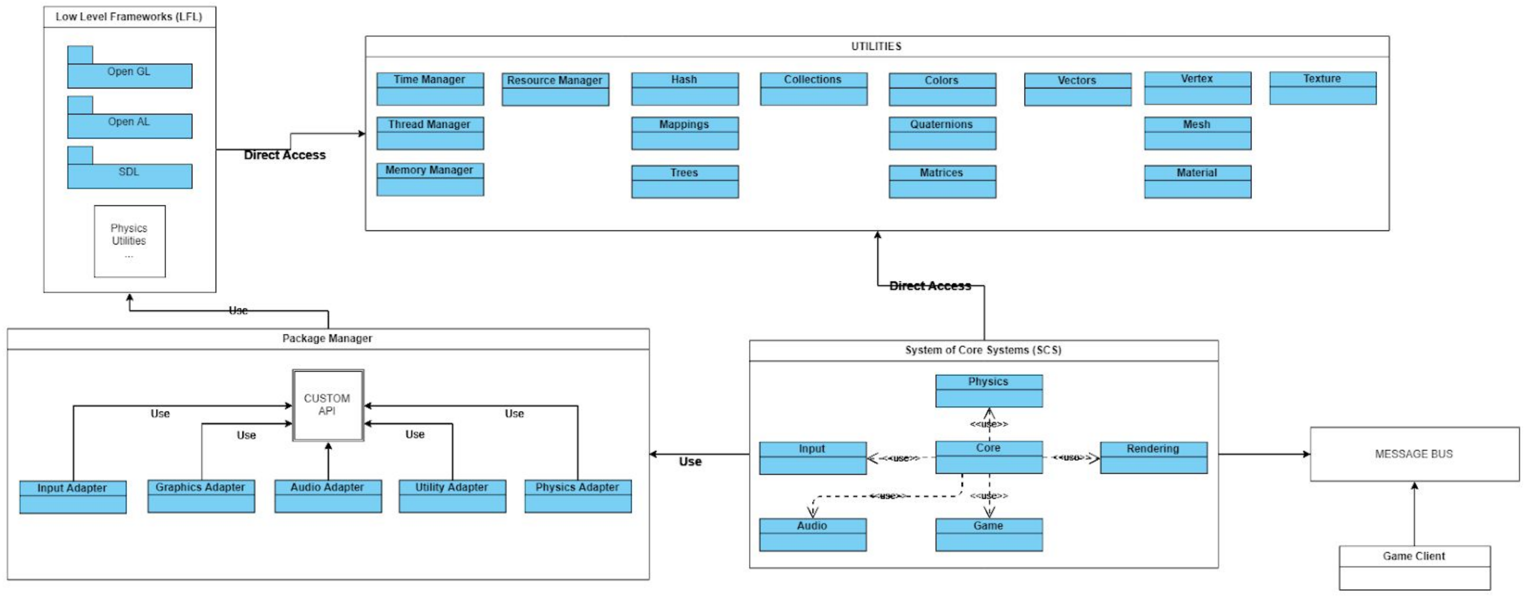

Figure 1 - Architectural View

As can be seen from the Figure 1, numerous number of systems are used within their contained scope in order to run and maintain the game client.

Utilities are consist of subsystems and data classes which are related to data structures, process managers like thread and memory managers, as well as collection items like trees, hash sets and more. LLF and SCS have direct access to these utility classes.

Low Level Framework consists of libraries and other frameworks that are used in order to perform graphics, input, audio and physics operations, as well as other utilities and their libraries related to the operating system connections.

PAM includes Custom API's, which will be responsible for supplying the right API to its managers like input, graphics, audio, physics and utility managers, so that they can perform operations on the API of the selected framework when requested by SCS. Utility Adapter in PAM is not to be confused with Utilities, as it is an adapter for using user written libraries inside SCS, if desired.

SCS consists of high level systems that are the backbone of the game instance process. The Core system is responsible for initializing, looping and cleaning other systems, as well as handling event and game loops, thread separation and memory handling. Input system is responsible for providing operations for the user written code to perform on player input. Audio system's task is to provide audio streams and spatial audio effects. Physics will be used for simulating physics upon game entities. Rendering system is responsible for handling drawing operations, together with the scene supervision system to provide optimized game scenes. Game system handles all entity, game data and logic processes.

Ideally, soft and hard dependency injections exist between the systems inside the SCS. However, these injections do not affect the lightweight structure of the systems, because they can be replaced with dummy injections for the systems user does not want to use, determined by the game options, explained later on. The communication of the systems are done with direct references and no additional messenger is used to provide maximum efficiency.

All the communication between user written game client and the SCS is supplied by the message bus.

3.2 Decomposition

3.2.1 Core System

Core System is the central system that manages intersystem communication and data flow within SCS. The communication of SCS with game client is handled with the Message Bus, while the intercommunication of SCS is handled via direct dependencies, using the Core System as the central node. The Core System is also responsible for validating these dependencies for any exception case.

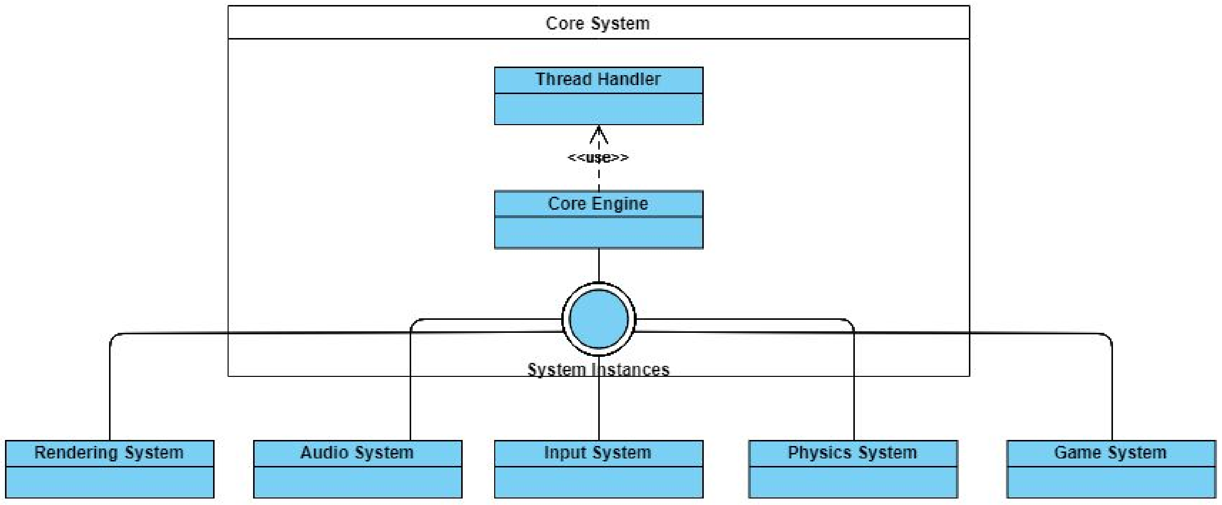

Figure 2 - Core System Architecture

As can be seen from Figure 2, Core System consists of a Core Engine, that uses System Instances interface to manage the instantiation, initialization, maintenance and validation of dependencies with other systems within SCS. Core Engine also handles these operations for the main game loop. Important events like time processing, rendering, physics and input commands, they all get ordered and executed inside the Core Engine. For ease of use, there also exists a

Thread Handler, which deals with the thread handling along with the Thread Manager utility system.

3.2.2 Game Core

The Game Core system is a crucial part of the architecture of Lina Engine. It includes necessary subsystems like ECS and defines their connections in order to process proper game logic, and handles the reading process of scene data.

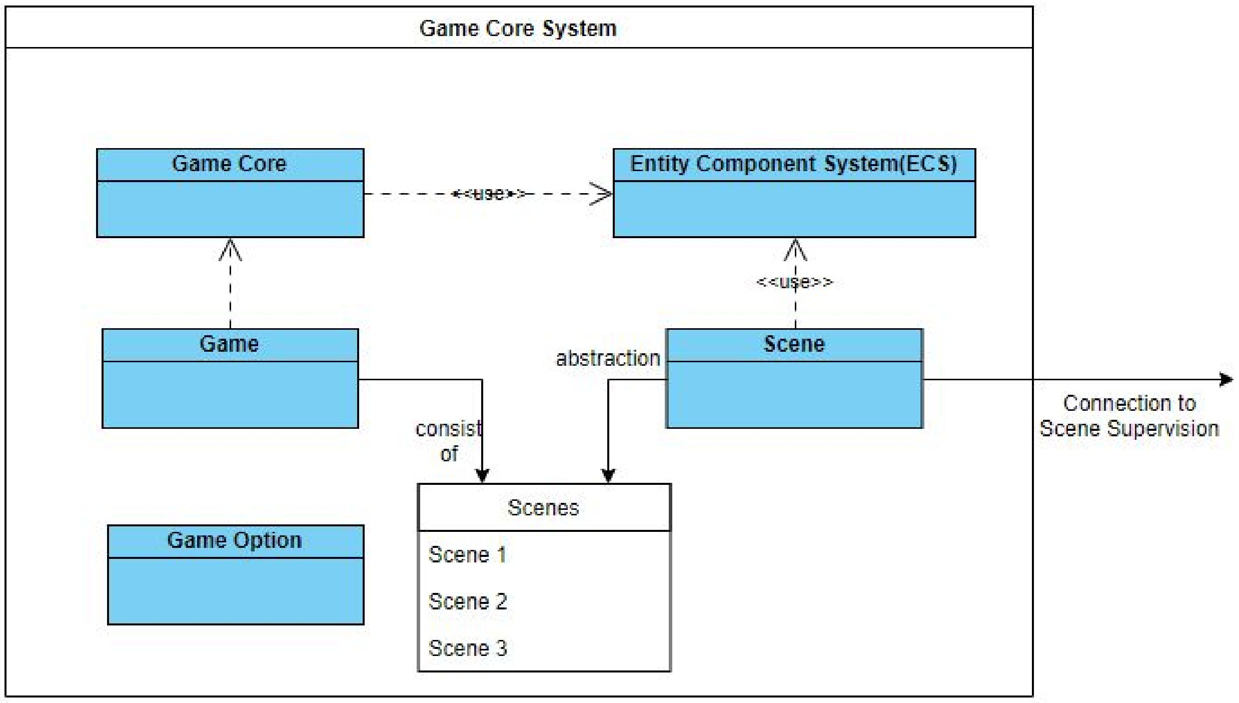

Figure 4 - Game Core System Architecture

Game Core System is mainly based on Game Core class and ECS. Game Core System connects a Game instance with the Core Engine, and Game is the abstract game class. Each Game instance is created with a particular Game Option, that works similar to the policies in rendering supervision systems. A Game Option is a set of rules defining the low level dependencies of the game. Given Game Option is then fed into the PAM to match with relative libraries in order to remove inconsistencies between a game's dependencies and PAM. As an example for the inconsistencies, the case where user chooses to include touch input modules in a Game Option but no touch control library is included in LLF can be given. PAM checks the match between the a Game Option and the implementation of adapters to construct a prior exception handling and ensure the correct form on Game Options. Scenes use ECS in order to define its rules and hierarchy, and it is also fed into scene supervision system of Rendering Engine. Thus, Rendering Engine will be able to perform operations and set the order of render according the hierarchy altered by ECS.

3.2.3 Rendering System

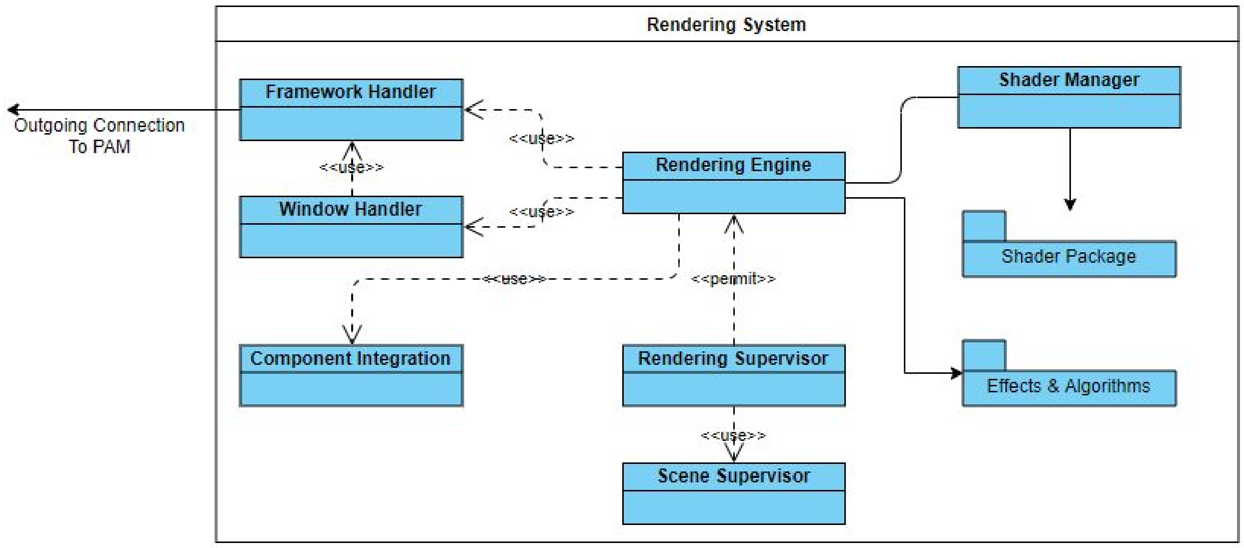

Figure 3 - Rendering System Architecture

Rendering Engine is the control block for the rendering operations. It is responsible for managing instances of windows, defining types of rendering like deferred or forward, working in correlation with the shader manager to handle single or multi-pass shading, as well as containing the definitions of particular rendering operations. It uses Framework Handler, which is responsible for the outgoing connection from rendering system into the PAM. Another

]

subsystem it uses is Window Handler, which is responsible for handling window events and actions and transferring necessary messages to the render engine via actions. Rendering Engine works in correlation with the Shader Manager, which is responsible for initializing, registering and cleaning instances of various shaders. Rendering Engine uses the operations from Shader Manager to push or pull necessary shader instances for multi pass rendering. Additionally, it triggers particular shaders at different stages of rendering if single rendering is chosen. Apart from these choices, if no particular change is desired by the user, all of the rendering operations can be done with the default pipeline and shaders provided by the Rendering Engine.

In order for Rendering Engine to enforce commands and operations, it needs to have particular set of permissions and policies to be active at the current rendering phase. The need for these is due to the initialization structure of Lina Engine, which aims to exclude unnecessary operations and data to increase efficiency. These permissions and policies are set by the Rendering Supervisor. The supervisor provides necessary permissions to rendering engine and enforces a particular form for the rendering engine to use. It works along with the Scene Supervisor which defines and enforces a form for the scene instances. It sets the rules within scenes and particular rendering order of the entities contained in the scene.

Rendering System also includes packages for shaders as well as effects and rendering algorithms which it uses for it’s logic. There also exists a component integration system that is responsible for connecting Entity Component System(ECS) into the engine. ECS will be discussed in the next chapters.

3.2.4 Input Engine

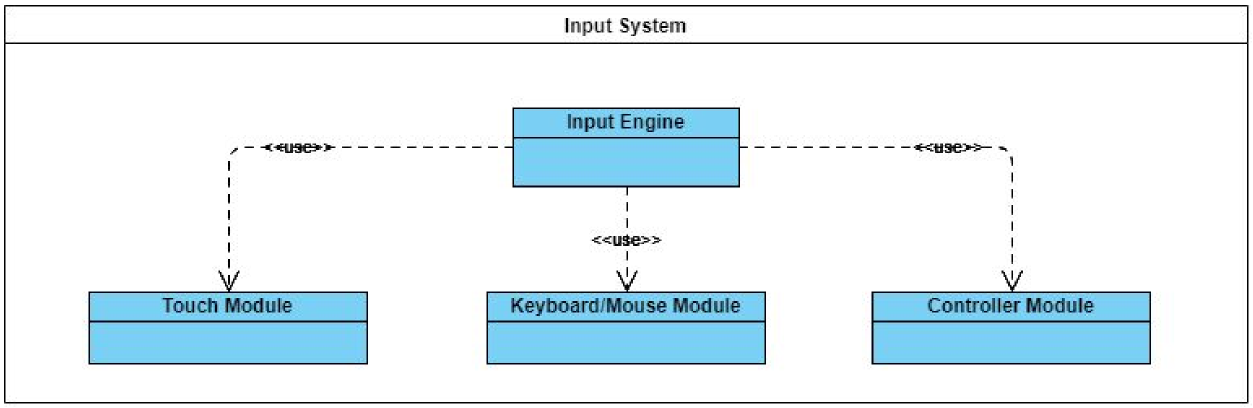

Figure 5 - Input System Architecture

Input System includes a central Input Engine, which is responsible for checking input values on the device when requested and returning the value on the desired format. It uses three different modules for Touch, Keyboard/Mouse and Controller inputs.

3.2.5 Physics and Audio

Similar to the Input System design, the Physics and Audio Systems are trivial in Lina Engine. Most of the goals that are sought to be achieved with the system design and implementation of Lina Engine revolve around other systems. That is why, for the sake of project schedule, the Physics, Input and Audio systems are kept as trivial as possible.

Physics System is only responsible for simulating basic 3D physics using a number of libraries. It has necessary output in order to feed the functionality of itself into the Entity Component System.

Audio System is trivial, as it is only able to deliver the functionalities necessary to stream and play audio. The only planned non-trivial functionality in Audio System is the 3D Spatial sound extension.

3.2.6 Message Bus

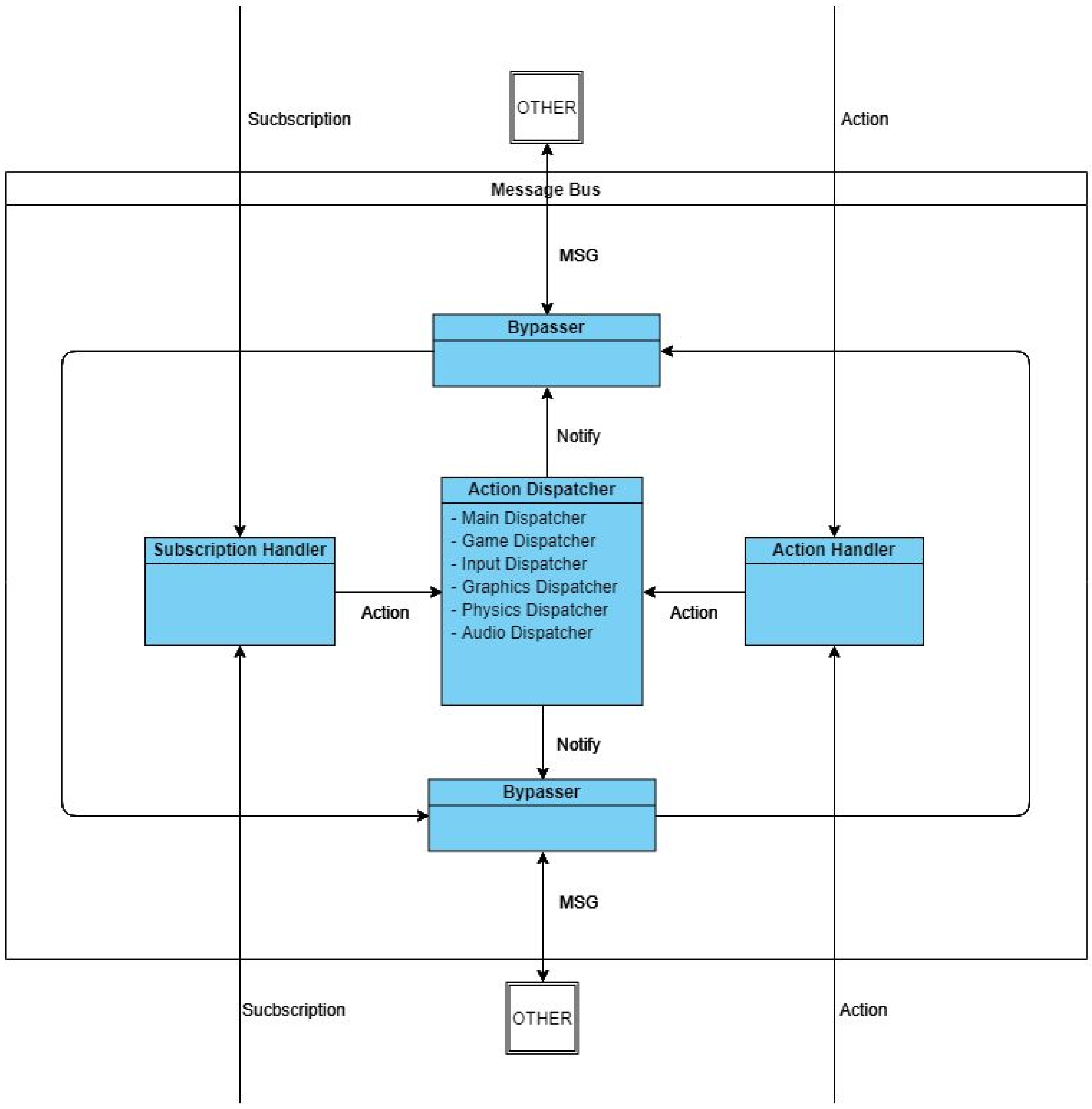

Figure 6 - Message Bus Architecture

Message Bus is a significant system existing in Lina. It is responsible for the communication of game client with the engine systems. It does this in two ways.

Actions: Message Bus includes an action system. There is an action handler, which receives actions from both ends and is responsible for collecting and distributing them to their correct collections in the main dispatcher. Subscription Handler works in the same manner, but is for the handling of subscriptions from both ends. There is a main action dispatcher, which is responsible for managing the collection of subscriptions and actions in their respective categories, and is the system that decides whether a particular action matches with the requirements of a subscription. When an action is sent to the queue, Action Dispatcher looks for a waiting subscription, and fires off a notification if there exists one. It sends the notification via Bypasser.

Bypasser: Bypasser is the system that is responsible for the direct communication of both ends. When one system in the game client high speed communication with the engine API, this communication can be done via the bypasser. Bypasser achieves this by skipping the action dispatchers and their queue, transferring the message directly to its receiver. Bypasser is also used to transfer notifications of the actions to their respective subscriptions. It is the gateway that receives an address with a message and delivers it, unlike the action dispatching system, it doesn’t hold a queue or is not capable of handling threads.

3.2.7 Package Manager

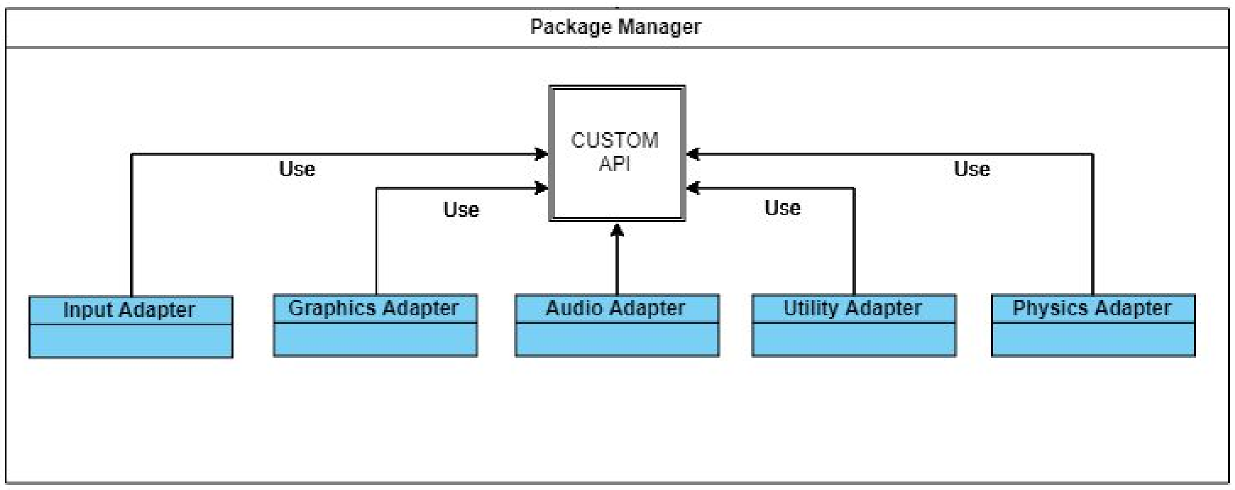

Figure 7 - Package Manager (PAM) Architecture

Package Manager is the system that handles the framework usage for the SCS. It is an extra layer of connection between SCS and LLF that provides validation and modularity for library usage. It is consisted of adapters for the separate systems inside SCS. These adapters are responsible for giving the correct response as data structures to the systems in SCS when requested. In this context, the correct response is the message returned from whatever library that is connected to the package manager. For example, SFML and SDL are two different frameworks that operate as a multi-media library. They are capable of receiving inputs from the device through the operating system, and they can tell us whether a particular key is pressed at the time *t*. Package Manager acts as a layer between SCS and these libraries, so Input System inside SCS does not need to know about the API of SDL or SFML. It sends a request to the Input Adapter inside PAM, and Input Manager is responsible for returning the request from whichever library is used. Here, the different wrappers for SFML and SDL are written as Custom API's that are abstractions of Input Adapter.

3.3 Rationale

From the design purposes and overview of architecture so far, we can say that PAM and ECS are the most significant features present in Lina Engine. These features aim to solve number of architectural problems existing in other game engines. In addition to these features, there are several important decision points taken while designing the architecture of Lina Engine. In this section, these points will be explained.

3.3.1 Direct Access Within SCS

We mentioned that within SCS, there is a direct access between systems. In other words, there exists a strong coupling between the systems present under SCS. This is due to the fact that these are the most crucial systems existing in Lina, as they are the ones performing the fundamental operations like main loop, rendering, audio streaming, physics, input receiving and running game logic. Ideally, since the Core System is the central and also the manager for other systems, it would be feasible to build a communication through Core System if any other system wanted to communicate with each other. However, this would add a small additional step of communication, that is unnecessary. Since the number of systems inside SCS are small, we can get along with a direct access and strong coupling existing among these systems. Since all these systems will be initialized and deinitialized all together, we can define a sequence of ordering for these processes, and there would not be any problems existing due to these dependency injections ideally.

3.3.2 Render and Scene Supervisors

In order to provide a wide range of rendering functionality to the users, we need to implement different rendering algorithms and sequences. This will result in many different options that user can choose from in order to render a particular scene. These options can vary from lighting and shading options, to choice of effects like fog, post processing or particle rendering. Additionally, most of these options are likely to be tweakable at runtime. Thus, it is inevitable to have many different combinations of options being selected and unselected, while doing a rendering job at any time *t*. Since the implementation procedure of rendering will vary depending on the active options, it would be really hard to keep track of where and how to proceed performing the operations if the options change. That is why, we need a set of constraints, defining what the rendering engine can do and can not do at the particular state, determined by the options. Since implementing these constraints in a highly variant environment with the options that they act upon can be very hard to maintain, Lina Engine hands over the responsibilities to supervisors. The Rendering Supervisor is responsible for enforcing the correct rendering procedures dependant on the constraints defined for current active options. In addition to this supervisor, we had separated the constraints on the rendering procedures with the constraints on the scene ordering. Options mentioned above like fog, skybox alterations, effects and particles are highly related with scene hierarchy. Thus it is feasible to also have a Scene Supervisor that sets these constraints only for the scene architecture, working along with and under the Rendering Supervisor.

3.3.3 Game Options

Similar to the set of constraints mentioned in Render & Scene Supervisors, a Game Option is a set of constraints on a game instance itself. It defines the choices for dependent libraries for a particular game instance. These options can be the input devices allowed, rendering techniques that are permitted and used or constraints on the frame rate for rendering and physics. Due to the nature of Lina Engine, it is possible to initialize the game systems specifically with a selected subset of systems. By collecting and defining the game options the engine systems can start only with the truly needed functionality. This process would result in a better optimization for the main game loop.

3.3.4 Message Bus

As mentioned in the overview of architecture, we use a message bus in order to handle communication of game client with engine systems. The main reason to use the message bus is to increase maintainability and modularity of the engine itself. When there is no message bus between two multifunction systems like SCS and game client, building a strong coupling between these systems becomes inevitable. This is because each system needs to communicate with other systems frequently. The game client needs to know about input, render system needs to know about game client to display anything meaningful. The straightforward way of doing this is to establish dependencies and direct references between the system. However, these references will make us end up with a lot of redundancies when the framework and utility references joins this communication circle as well. If we use a message bus, then most of these problems can be avoided. The message bus would be the center of communication, who holds the references and delivers messages to the correct place. Thus, we would have a better organized structure for communication between two ends. . Having a centralized system will reduce redundancies, thus making problem detection, implementation alteration and maintenance easier to perform. The weight that this centralized system is also avoided using the Bypasser inside the message bus, which would be explained below. In addition to that, while writing game client, it will provide an easier way to obtain encapsulation for the engine classes, since the operations that can be accessed by the game client would be strictly defined by the API that the message bus provides. This definition in message bus is done via Actions.

There exists an action system within message bus as explained in section 3.2.6. This system is able to elaborate incoming actions and subscriptors and distribute messages correctly to their listeners. However, this comes with an overhead. Initially, for the games where the entities and operations performed by them are not many, this overhead would not be noticeable with today's hardware. But, it can be a problem where large number of entities perform messaging with the engine classes, like strategy or simulation games. In order to overcome this overhead, we had also implemented a Bypasser System in the message bus. This system does perform outside of the boundaries of the action system, which is dependent on threads and queues for the actions.

The bypasser would allow some sort of direct communication, giving away encapsulation and organization of structure, in exchange for speed. The game client will be able to access to operations from the action system, as well as the operations given by the bypasser. Moreover, the action system is able to use the bypasser whenever it needs to empty up some space, dispatching events faster.

3.3.5 Package Manager

PAM is an extra step in communication of SCS with the LLF. This need for an extra step is highly related with the goal of providing a *lightweight* engine.

When we look at the some other open source game engines in the market, we can easily see that most of the frameworks and libraries that those engines use are strongly coupled with the main system of engines. In other words, these engines enforce their users to use the libraries that they choose. Of course, users are able to write plugins for extra libraries, but since most of the systems are highly dependent on the ones that were included at the beginning, the users still have to suffer from the weight of those libraries that they do not use. The range of these libraries can differ, from media libraries to drawing frameworks, from 2D and 3D utilities to audio systems. One of our main goals is to make Lina Engine as light as possible. So, we are aimed to solve this enforcement and heavy weight problem by making Lina with a such architecture that users would be able to choose the most low level libraries themselves. Of course, we would provide implementation for the libraries that we choose initially, but they would not be necessary for engine to run, as long as a replacement by the user is placed.

The way PAM works is, instead of asking for a direct command from the libraries, SCS asks PAM for the commands that it requires. SCS then provides these commands, under the adapter specifically written as an API wrapper for the target library. This way, an adapter pattern would be used to link the SCS to the targeted library. Making the initial libraries that we had provided totally removed from the systems.

4. DATA ARCHITECTURE

4.1 Data Description

4.1.1 High Level Data Structures

In order to handle operations throughout the systems of Lina Engine while running a particular game client, there needs to be a constant way of flowing the data. Lina Engine achieves this by collecting the data into relevant data structures, as it is the traditional way of handling data. There are utilities for handling groups of data.

The most significant utility is Lina Engine's collections classes. They consist of linked lists, array lists, sets, hash sets, linked hash sets, tree sets, maps and hash maps. The way Java handles collections is the source of influence, due to its convenience.

In Lina, there exists collections for storing rendering related structures like vertices, indices and meshes. In addition to these basic rendering structures, data sets for storing lighting, shadowing and terrain data is to be implemented for future expandability.

Lina Engine includes a range of functionality inside its Math library, which also contains data structures for storing colors, vectors, quaternions and matrices.

4.1.2 Data Structures For Engine Configurations

In order to handle engine configurations and operating system arguments, Lina Engine uses simple configuration files, in plain text and XML format. These configurations are shared between the editor and the initialization stage of the engine's main loop.

4.1.3 Player Data

One of the most crucial elements to be used inside a game is the storage of the player data. Lina Engine provides basic storage operations to store custom user data into the hard drive of the running device. The most common operations provided are described below.

Saving and Loading Basic Data Structures: Lina provides an interface for the users to save the values of basic data structures like int, float, double and boolean. The user is able to use an API provided similar to Player Prefs given by Unity. Also, it is aimed to provide structures for handling the save and load of other data structures like lists and sets as a whole.

Saving and Loading Objects: Users are able to serialize entity states as objects and deserialize them using object streams. This way user can load a game's state through particular number of iterations on the loaded object streams.

Saving and Loading Configurations: As can be deducted from the separation of data structures and objects, it can be said that Lina provides different levels of user friendliness in the means of storing data. In addition to ability to save data into an object stream, there also exists operations to save game configurations separately. Meaning that the user will be able to save game configurations related mostly with hardware options in separate serializable structures.

While doing these saving and loading operations, Lina provides a simple XOR encryption to the data, in order to provide a trivial sense of security.

4.2 Data Flows

4.2.1 SCS - Initialization

Initialization of the systems inside SCS are given in Figure 8.

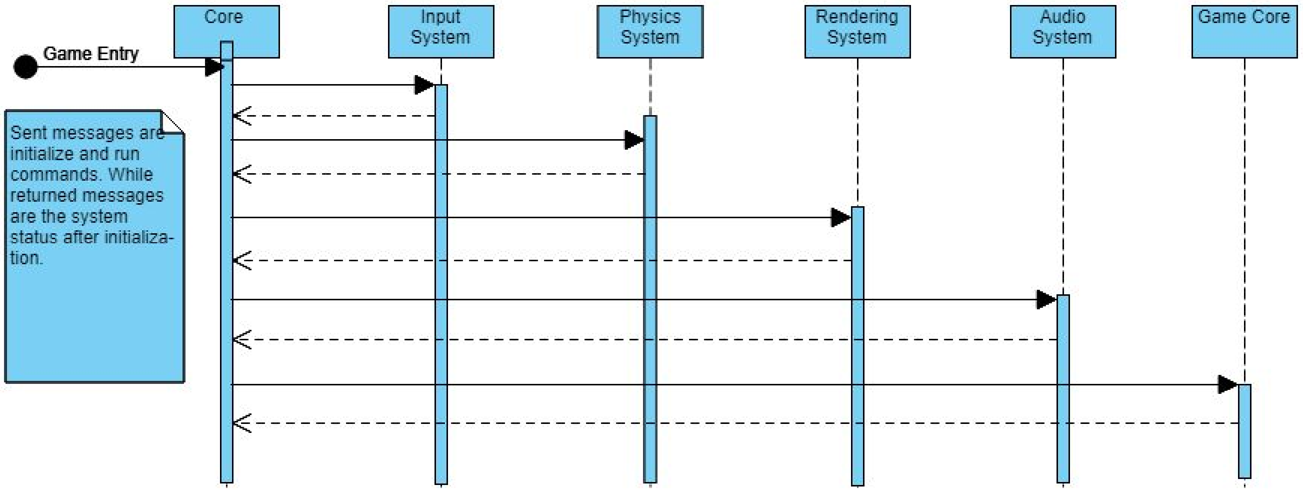
Core
Input System
Physics System
Rendering System
Audio System
Game Core
Game Entry
Sent messages are initialize and run commands. While returned messages are the system status after initialization.

Figure 8 - Initialization Sequence Diagram

The messages that are sent from the Core are “Initialize and Run” commands with various arguments. These commands initialize the systems and makes them run a dummy frame for the status check. The returned messages are the states of the system status. If any state rather than “OK” is received, the core does not execute the sequence anymore and falls into the exception handling category.

Figure 8 can be thought as waking the systems, after all systems are awaken, the same process is repeated for their “start” operations. After these operations return “OK” status, we continue to frame iteration. As in the initialization, any system returning any status rather than “OK” will cause the main to stop the execution and fall into exceptional states.

4.2.2 SCS - Main Loop

We can see the sequence diagram for the main loop in Figure 9.

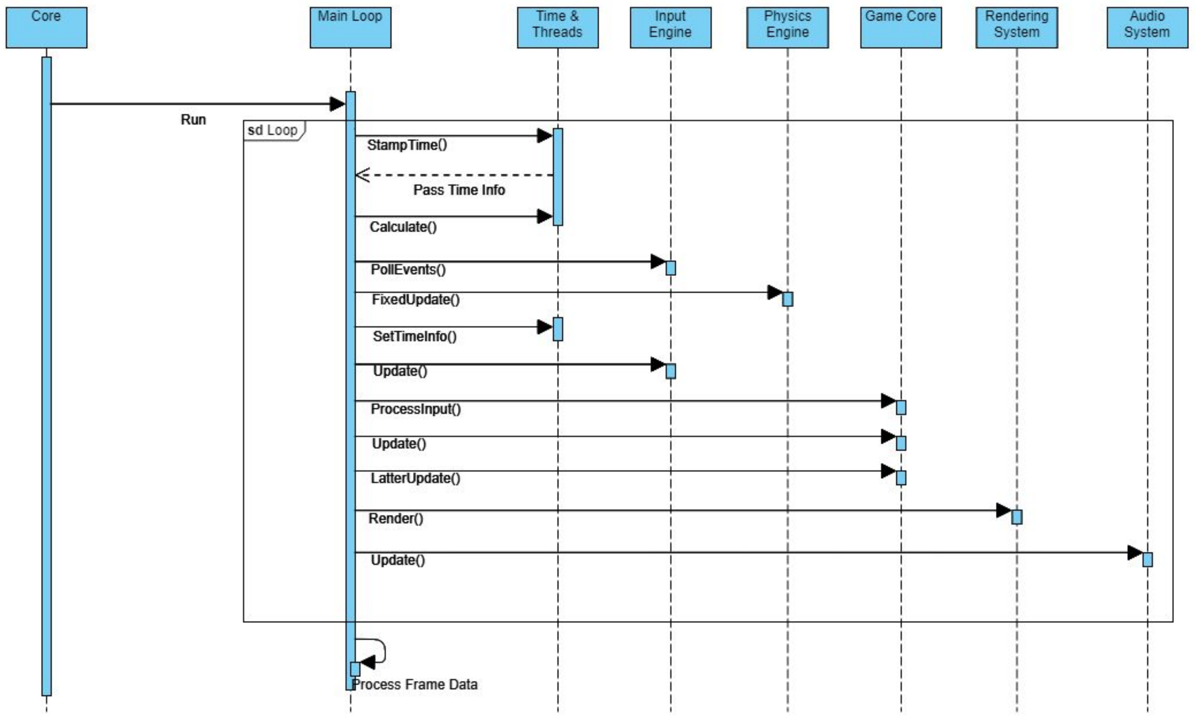

Figure 9 - Main Loop Sequence Diagram

As can be seen from Figure 9, there are several update runs for different systems inside SCS. The order of these operations definitely do matter, as most of them requires information to be processed and calculated before them. Especially in the Input System, polling input events must occur before updating Input System's state. Same with the Game Core, processing input must done before updating the Game Core, so that the user input would be captured within the frame.

For updating purposes, three kinds of tick methods are used; Fixed Update, Update and Latter Update. Fixed Update is processed at the physics time and in a fixed rate, that is why time information is set after it is performed. Update is our main tick event in which the user input and entity update will occur. Latter Update is an another tick event which will be performed after the Update is done, so that more detailed functionality based on the state of entities previously calculated in the latest update can be implemented in the game client.

4.2.3 SCS - Termination

Below we can see the process after the main loop is done.

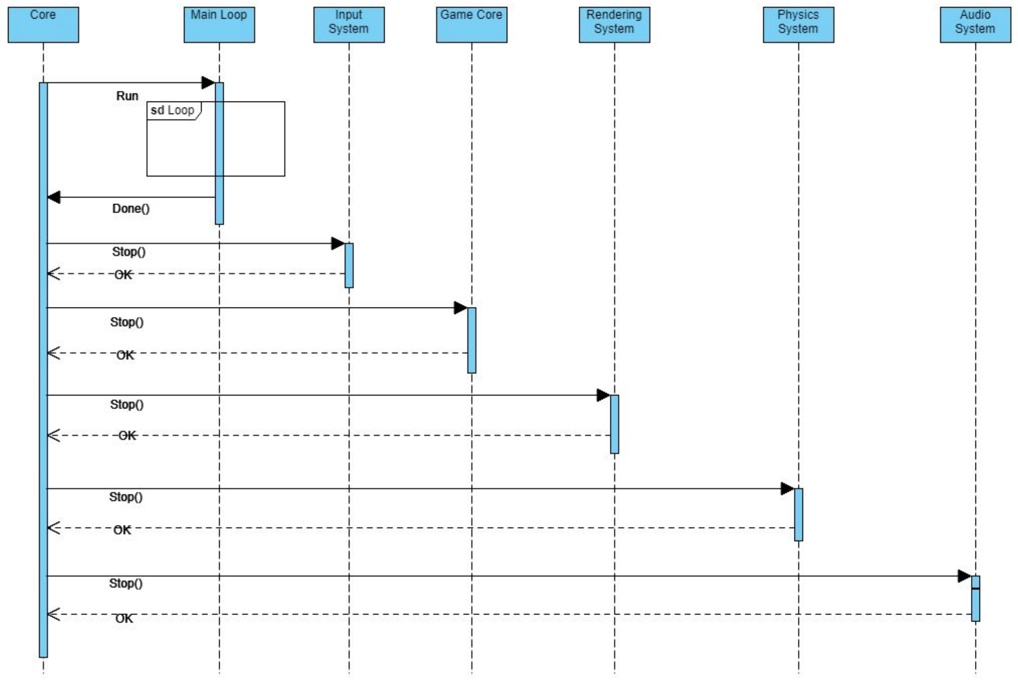

Figure 10 - Termination Process Sequence Diagram

The condition for the main loop to be completed can emerge from numerous events; window operations, proper termination of game state or even an unexpected and unhandled error. Just like the initialization, the systems are stopped sequentially, and just like in all the sequences so far the ordering important. Some systems requires other systems to stop running before being able to properly stop their process. After the stop command has reached around and the systems are terminated successfully, process of cleaning up the systems comes into play. There is no diagram drawn for cleaning up, as the process is identical with the stop process. Only this time, in the implementations, systems clean up their unwanted and unneeded data, then perform a memory optimization for proper deinitialization to be completed.

5. ENTITY COMPONENT SYSTEM

5.1 Entities and Components

One of our biggest goals in Lina Engine to achieve fast and reliable systems to build game logic and process game states. In order to do this, we need to have a solid system of entity management. An entity is any object, property or a being inside the game world, that is able to carry specific behaviours in order to build, control or maintain game mechanics. An entity can be rendered, it can be controlled by player or it can be an AI. Basically any container that provides functionality to the game is can be called as a *game entity*. When we group the behaviours which these entities can perform into their respective classes, we can call these classes as *components.* Building a solid system to handle entities and components is a crucial step in game engine development, as this system will enforce form on how the game will be built by the users. Moreover, the performance of this system will have high effect on the performance of the game created, as entities are the building blocks of a game.

5.2 Inheritance Approach

Providing behaviour to entities is a problem that can be solved using various ways. The most common way is using an object oriented approach to designate behaviours on the game entities. This approach is inheritance based, and is used commonly throughout many game engines that are in the market today. The main idea is to implement behaviour as components, and enforce the game entities to inherit operations, attributes and states from these components. An entity then would have the functionality of inherited component. For an example, an entity can be a *DamageableActor Component* which will result in the entity to be able perform operations of an

Actor and receive damage as well, since *Actor* would be the superclass of *DamageableActor*. This way of building entity logic architecture is fairly straightforward and easy to use in the cases where number of components are low. However, it is not the approach we used in Lina Engine. There are number of reasons for this.

First, as mentioned, this approach is inheritance based. Over time, when the number of entities inheriting from various components increase, the overall coupling of the classes begin to rise dramatically. For example, in order to create a robot that the user is able to control and a robot that is controlled by the game, we would need to implement a *RobotActor* class, as well as a *RobotAI* class, that both inherit the features written in the *Robot* class. Now, any change to the *Robot* will affect both classes. As the number of classes and subclasses increase, the modularity of the systems begin to decrease dramatically.

Second, due to the inheritance, there occurs a tendency to push common behaviours between different classes of components into more generalized subclasses. This behaviour ironically reduces the efficiency of the whole class hierarchy. For example, we can write a base *Enemy* class to define behaviour of enemies. To extend it, we can write a *ZombieEnemy* class that derives from it. If we want actors that can implement different kinds of operation environments, we have to write *GroundZombieEnemy* and *SkyZombieEnemy* separately. Of course we can add ground or sky as an attribute, but we have to add different kinds of enumerations each time we want a new type of environment. In other words, each kind of functionality that is big enough to exclude from the class has to become a new class or an interface, disrupting the hierarchy and hardening the maintenance.

Moreover, as the type of components increase, the inheritance chains for the components begin to get very long. When talking about hundreds of different components, distributed amongst different systems, we will have a huge ambiguity between different components due to the inheritance chain.

As the number of components increase, the traditional approach becomes really hard to maintain and keep its performance at a steady level. Furthermore, in the development process of a game, as the game gets bigger and bigger, it becomes very hard to maintain game entities and groupings of them, due to the problems in the entity architecture explained above.

5.3 Composition Approach

One another method to provide entity logic is the composition architecture. Instead of using inheritance, this architecture follows composition over inheritance principle. This will eliminate most of the ambiguity and coupling problems. In this approach, entities do not derive from the components, but rather are composites of multiple components. So that an entity can have an *Actor* component, it can have a *Damageable* component. Creating entities as compositions of various components is even better than traditional approach, since it is easier to use and eliminates most of the problems in traditional approach. Moreover, it is also easier to maintain and explain, and used widely in popular game engines like Unity. However it has some performance costs and the development process of a game using this architecture might get over complicated if the number of components and entities gets enormously big.

In Lina Engine, one of our biggest purpose is to eliminate common deficits in modern game engines' architectures. That is why, in order to eliminate this object oriented, inheritance and composition based entity relationship problems , we have decided to implement a new kind of a system to handle entity logic.

5.4 ECS Approach

Entity Component System (ECS) is an alternative architecture to handle game entities and their behaviours. Rather than using object oriented approach, we use a delegation based architecture in ECS. In ECS, a game entity, does not inherit or perform the behaviour of a component. But rather, an entity is nothing but a collection of components. It is only a logic object in the game world, that holds components, but do not perform any operations that are designated for those components. The behaviour of entities are strictly defined by the components that they aggregate. In other words, rather than inheriting behaviours, the delegations to cohesive components are used. The components in ECS are purely nothing but raw data. They are the collection of attributes and their states, and they do not include or perform any operations, but only keep the information necessary to perform operations. When we look at ECS for now, we have components, which do not have any behaviour but only raw data, and entities that are the collections of those components. The behaviour is performed as follows. The systems in ECS comes into play. There are various systems operating over various components, mostly with efficient rules of iteration. All the behaviour logic is implemented within systems, and systems use the information on the components and apply the corresponding behaviours to the entities that are paired with those components. In this process, systems do not hold any references to entities or components. They discover entities by using managers that manage the queries of the components. This way, in order to alter or fix behaviours, we only need to find system-component pairs, so the problem of increasing complexity in handling entities would be removed, as they only collect components, not cast their behaviours.

In ECS, there occurs a procedural design, in which the components are nothing but data transfer objects. In other words, as mentioned above, components do not carry any behaviour, and the implementation of behaviours are isolated from the systems. This way, we can create various modules to implement any kind of system in the way that we want.

ECS provides high amount of modularity, along with a solid and high performance. Due to entities not being the center of implementation, if we can perform efficient implementation of behaviours in the systems, it is possible to manage millions of entities with a constant frame-rate throughout countless frames. Millions of entities, that are rendered and perform operations, would be almost impossible to process with a constant performance using other approaches.

In Lina Engine, we use ECS with additional object oriented features added in order to make testing easier and use the advantages that ECS provides.

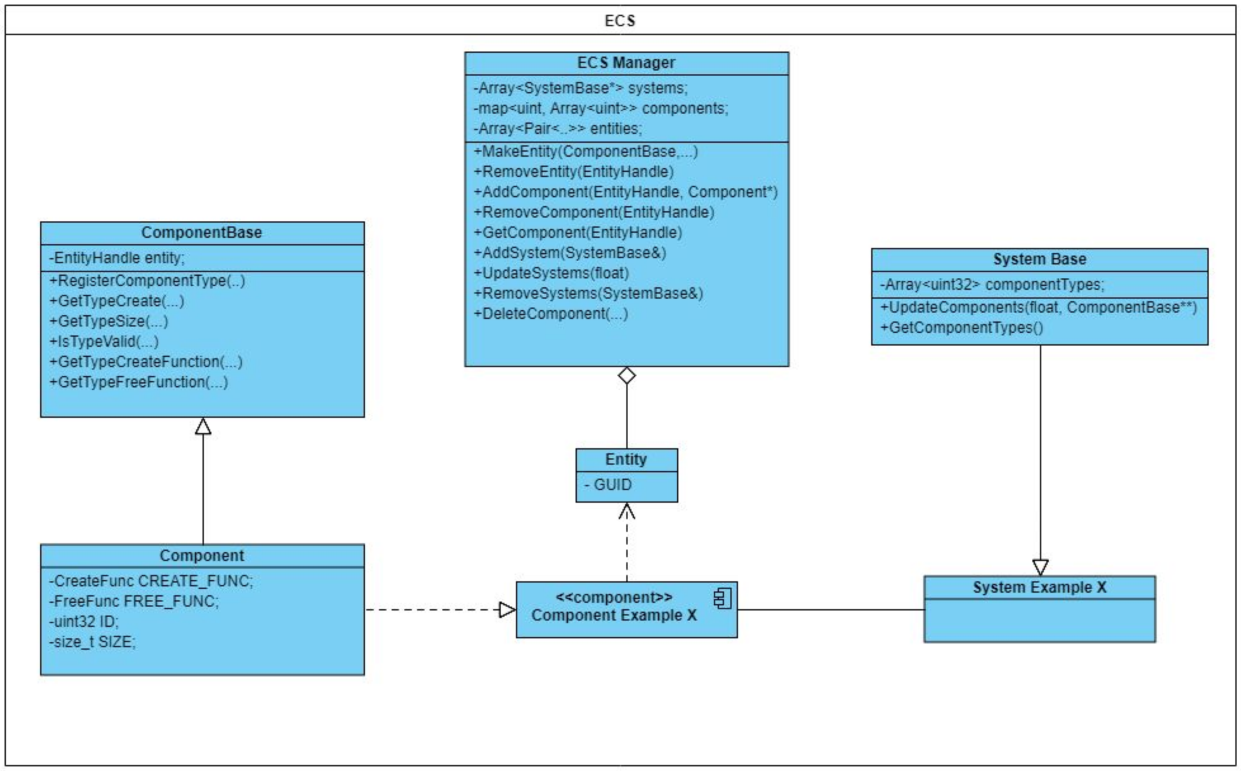
ECS
ECS Manager
-Array<SystemBase*> systems;
-map<uint, Array<uint>> components;
-Array<Pair<..>> entities;
+MakeEntity(ComponentBase,...)
+RemoveEntity(EntityHandle)
+AddComponent(EntityHandle, Component*)
+RemoveComponent(EntityHandle)
+GetComponent(EntityHandle)
+AddSystem(SystemBase&)
+UpdateSystems(float)
+RemoveSystems(SystemBase&)
+DeleteComponent(...)
ComponentBase
-EntityHandle entity;
+RegisterComponentType(..)
+GetTypeCreate(...)
+GetTypeSize(...)
+IsTypeValid(...)
+GetTypeCreateFunction(...)
+GetTypeFreeFunction(...)
System Base
-Array<uint32> componentTypes;
+UpdateComponents(float, ComponentBase**)
+GetComponentTypes()
Entity
- GUID
Component
-CreateFunc CREATE_FUNC;
-FreeFunc FREE_FUNC;
-uint32 ID;
-size_t SIZE;
<<component>>
Component Example X
System Example X

Figure 11 - Entity Component System (ECS) Architecture

We can see the implementation of ECS approach that is explained in Lina Engine in Figure 11. This ECS System is part of the Game Core system, mentioned in section 3.2.3. As can be seen from the figure, we have implementations for components and systems. As well as a global ECS Manager, which is responsible for the queries involving these components and systems as well as component-entity pairs.

The Zombie & Enemy example in inheritance approach now can be implemented in ECS in a more elegant manner. A zombie would be a simple entity, that holds *SkyMovement* or *GroundMovement* components that are operated by the *Motion System.* It can also hold *ZombieEnemy* component operated by the *Behaviour System*. This way, the only hierarchy would be between the components, and that is only if a hybrid approach including OOP is desired. We can extend the enemy behaviour into *Enemy System* and thus do not need a base *Enemy* and sub *ZombieEnemy* components. When a problem regarding enemy's movement occurs, we would only have to look at the *Motion System*, not the entity or not the component itself, as it holds holds raw data. This way, modularity would be increased and debugging can be much easier.

6. HUMAN INTERFACE DESIGN

As for the user interface, since it is already a lightweight game engine, Lina Engine is planned to have a level editor instead of creating a complex game engine editor like Unreal Engine 4 and Unity that has lots of buttons and panels. While not including the complex parts of the popular heavyweight game engines, Lina Engine Level Editor has convenient game engine features such as loading resources through the editor, supporting multiple scenes per project and sharing resources between scenes of a project.

6.1 Overview of User Interface

Because we do not want to create an overwhelming user interface, in the design of the level editor we follow the WYSIWYG(What You See Is What You Get) principle. That is why, Lina Engine provides an editor that built on top of the sample game that is built with the engine as the user interface. The sample game scene will be an entry point whenever the user creates a new project by using the editor. The user can create a new scene and edit the scene by using the tools that editor provides. Provided tools by the editor are, translate, rotate and scale the objects which is loaded by using the editors resource loader tool. For each project there is a project folder that contains scenes, engine configuration files and player data. Furthermore, for each scene, there is a scene folder which stores assets in the scene. Since the assets are saved in the scenes, the users can use and share the same assets in different scenes by using the import asset functionality. While working on the project, users will have a view all of the assets in all of the scenes of the active project. After working on the project, the users can save the project for testing or working on the project later.

Lina Engine Level Editor has basic required features such as, scenes, scene selection, saving, loading, exporting game data, importing a resource and sharing resource between scenes at the same project.

Because the game is already built on top of the working game engine components, building the editor on top of the sample game provides three major benefits. The First one is, after working on the scene, the testing time will be minimal since the editor is built on top of the working engine components, given that the engine components that are used in the editor are stable. That means there is not going to be a weird geometry error if it is not seen in the editor while editing the scene. Second benefit is, after editing the scene, the user can directly save and test the game without losing time because while editing the scene, the user also edits the game as well. Last major benefit is that, since the users work on the scenes and each scene has seperate folder, scene folders can easily shared between developers.

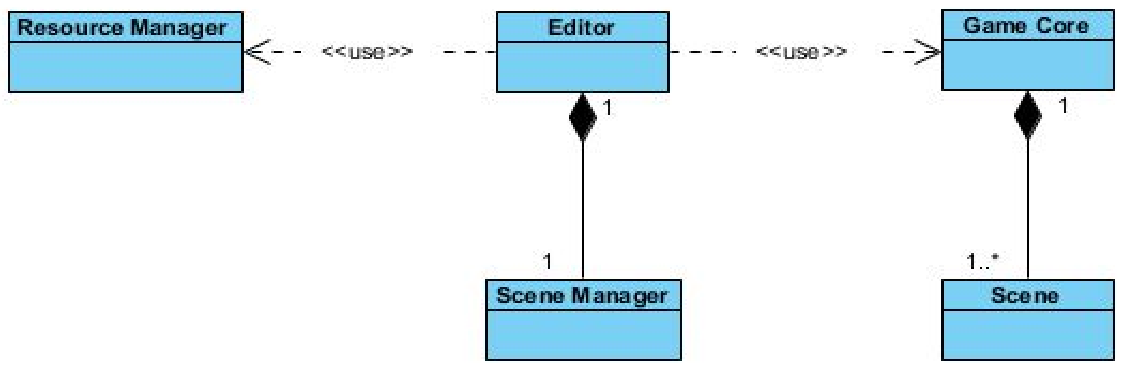

Figure 12 - Class Diagram for Lina Editor

As it can be seen in Figure 12, the main editor class the Editor, only aggregates the well established parts of the Lina Engine such as, Resource Loader and Game Core. In addition to that, there is an active Scene Manager in the Editor for managing the scenes in the active project.

7. REFERENCE MATERIALS

L. Bishop, D. Eberly, T. Whitted, M. Finch, M.Shantz(1998). “Designing a PC game engine” IEEE Computer Graphics and Applications (Volume: 18, Issue: 1, Jan/Feb 1998).

Vahldick, Adilson & Mendes, Antonio & José Marcelino, Maria. (2016). Towards a Constructionist Serious Game Engine. 10.1145/2983468.2983526.

H. Qiu and L. Chen, "An Object-Oriented Graphics Engine," *2008 International Conference on Computer Science and Software Engineering*, Hubei, 2008, pp. 1027-1030

“Definition of ES”, as appears on http://entity-systems.wikidot.com/, accessed on December 3rd, 2018.

Gestwicki, Paul. (2012). The entity system architecture and its application in an undergraduate game development studio. Foundations of Digital Games 2012, FDG 2012 - Conference Program. 10.1145/2282338.2282356.

D. Maggiorini, L. A. Ripamonti, E. Zanon, A. Bujari and C. E. Palazzi, "SMASH: A distributed game engine architecture," *2016 IEEE Symposium on Computers and Communication (ISCC)*, Messina, 2016, pp. 196-201.