**IZMIR UNIVERSITY OF ECONOMICS**

**FACULTY OF ENGINEERING**

**SOFTWARE ENGINEERING**

# FENG 497(498) PROJECT REPORT


LINA ENGINE

Author(s): İnan Evin - Bekir Batuhan Bakır

Supervisor: Kaya Oğuz

# Abstract

The project covers an extensive research on the architectural design of game engines, followed by design and implementation of a game engine, that offers alternative solutions to deficits commonly existing in open source and commercial game engines. The research is motivated by the substantial needs and requirements of the game development community. In the existing market there are state-of-the-art game engines to provide solutions, however they are mostly business-driven with a motive for focusing on the larger audience. This results in heavy structures, forcing the developers using them to include enormous amounts of undesired features and frameworks that are capable of providing many functionalities that most developers do not need in their particular projects in the first place.

Our method of choice is an implementation of an open source game engine, based on the architecture we have designed. By developing an engine that has trivial features to develop a game and demonstrates our proposed architectural solutions, we can establish a basis for the game development community to work upon in order to extend those solutions. We based our methods on an extensive research about available engine sources and design flaws in the architectures of those engines. Implementation of a design that is based on this research and its findings, means that there would be a solid work of an open source engine, that is coupled with its low level dependencies, as weakly as possible, resulting in a highly lightweight architecture that most of the engines in the market lack. As a result, it would open new possibilities in the game community such that the other developers would be encouraged to expand the architecture in order to achieve better engine systems, thus benefiting the whole game community.

# 1. Introduction

Game engines have been a topic of interest since id Software released *DOOM Engine*, as the core technology behind their remarkable and landmark franchise *DOOM* became popular around 1990's. Since then, the realization of the need for a framework to develop games became popular and has been in the radar of many interested game and technology development companies. The idea and genesis of the concept "game engine" was to provide a technology framework in

abstract and polymorphic manner for games development, such that the time, money and effort required for releasing new titles would decrease dramatically, mainly based on the reusability of said technologies. They provided simple libraries and their integration into larger frameworks, to solve problems caused by hard-coded and pre-arranged data in the design, initiation and development process of a game. As the requirements inside a particular game idea got larger, as well as the possibility and the variance of the games that can be targeted, the need for bigger, faster and better engines arised quickly. Especially after 2000's, the release of engines like; id Tech, iw Engine, OGRE, Torque3D and Anvil, has boosted and shaped the market in drastic ways. After 2000's, studios were not only interested in advancing their technologies for their own titles, but they were also keen on developing engines to be sold to other game development stakeholders and parties, which in turn lead to many engines being developed for open community, with various commercial & personal licenses available, thus opening a new era in the means of game development. Nowadays in the game development market, many engines that are capable of providing AAA features are available to open community and independent developers. Engines such as Unreal Engine, Godot, Cocos and Unity can be counted as the largest examples. As we call them modern engines, these softwares deliver highly flexible features to enable any interested individual to prototype and develop any type of game they want in easy, cheap and efficient way. However, these flexibilities provided by this type of modern engines also bring problems along the way in the means of game development. We can refer to the biggest one of these problems, as the fact that the development process got too easy, and the architectures behind them got unstable.

In order to provide ease of use, flexibility, modularity and capability of AAA development, modern engines available to the community had to rearrange their structures in a more openly manner. They had to think about many possible scenarios that a developer would be in, and provide functionality for every single one of them to support wide usage. This idea has brought many structural deficits for those engines as well as implementation problems. It resulted in an iterative process of game engine development where they patch and cover the fundamental errors from the community and keep implementing more features, rather than fixing the current problems. Many of these fundamental problems can be narrowed down into one title: lack of simplicity and lightness. In order to feed the community with the idea of ability to do everything

easily, these engines had to include many built-in libraries and frameworks into their core structure as well as many Application Programming Interfaces (API) to provide communication between high-level and low-level game code. As a result, they enforce such structures to the users that even though it is easy to use and quick to prototype small ideas, as the complexity of the game increases, it gets harder to maintain the systems and architecture inside the game, along with the increased drop in efficiency. Since these engines have high influence on the market, most of what seems to be a new method of engine development, ends up being similar to ones on the market. The resulting phenomenon of engine architecture, along with the development of careless third-party plugins for those architectures, has been a huge problem for teams who aim to develop titles of high complexity and still want to benefit from a power of a game engine which is open to community.

## 1.1 Problem Statement

The problem we focus on is the need for an open source engine, that is trivial to use and dedicated to help solving those issues and design flaws existing in modern game engines. The game development community is bound to use the existing modern game engines because there does not exist many alternatives rather than the ones competing in the market. Even though these engines are well written high level frameworks, their flaws and design issues bring frustration to most developers and this affects especially the independent game development communities. To overcome this, many developers work on projects that are tools, plugins and extensions to these engines. Even some work on developing their own game engines, but most of the projects do not continue due to complication of the subject. The community is in need for an example work of a game engine that presents an architecture which does not include the deficits existing in modern engines, so that developers can influence and base their work upon.

## 1.2 Motivation

The research for generating new techniques for an alternative game engine architecture has the potential to influence the game development community. With this influence, developers can contribute more into the open source engine development and open the gates for possible new technologies and methods to be used in game engine architecture. By compromising the “an

engine for everything" motto, it is possible to extract and eliminate core problems existing in those highly favored engines. Problems such as slow and heavy communication interfaces, enforcement of unmaintainable code architecture due to poor scripting front-end, weakly designed event & messaging systems, entity systems that tend to create code redundancy and duplication can be counted as the most noticeable ones that are seen in the common architectures. It is possible to establish a layout that avoids these problems by using system models that are compatible with each other and consistent in architecture, rather than trying to mix every different types of models into one huge engine to provide vast functionality, which can be seen as one of the biggest mistakes that consumer hungry engines tend to do.

The findings and implementation of this research might influence and encourage many developers to interact with these creative techniques and methods of engine development. The collaboration may produce a community of developers who are able to develop their games with these unique methods of implementation and modular systems to extend the powers of the engine by only using what they really need in their game, thus decreasing the compromise of the efficiency. Such an efficient engine architecture might shape the game development communities' view on the sources for developing real-time applications, thus creating a chance to open the doors for more open source and community driven engines and technologies to develop games, without being forced to stick to the boundaries of those provided by business driven leads.

# 2. Literature Review

Lina Engine adapts good number of principles, designs and ideas, extracted from various academic researches, which have been used by industry developers and academic personnel for research purposes extensively. It is important to point out the main sources that has inspired various designs in Lina Engine, so that we can construct a basis of strong arguments behind the methodologies used in the design of our engine. Due to the significance of concretely pointing the academic sources, this paper is written in a format to mention those sources, analyze and extract the main ideas of them. So that we can specify which parts of those sources inspires Lina Engine, what are the similarities and differences in the techniques discussed and used, along with any comparison traits if viable. Hence, we can form a group of reasoning that we can present as an

argument to support the design choices made throughout the Lina Engine's development lifecycle. This paper's purpose is to give information about these academic materials in question in the means of a review, extract their main ideas along with significant points, put these ideas with the choices made in Lina Engine side by side, to compare them in differences, extensions or similarities, in order to form the basic arguments behind our design. This paper follows a sequential format, in which each research and source is mentioned and review first, then their relationship with Lina Engine is explained.

## 2.1 The Case for Research in Game Engine Architecture

Anderson, Engel, Comninos and McLoughlin (2008) suggest in their research that there exists four significant questions that should be asked while designing a game engine. These questions are; "Where is the boundary between game and game engine?", "How do different genres affect the design of a game engine?", "How do low-level issues affect top-level design?" and lastly "Are there any specific design methods or architectural models that are used, or should be used, for the creation of a game engine?". Anderson et al. (2008) states that a general answer for the first question can be the design border in the engine's architecture in where the integration of components that do not specify game's logic or its environment can be drawn. Even though the general example is too abstract and is not satisfying in the means of concreteness, it is also the best case answer that we can give for the definition of a game engine. This is due to the fact that in most of the time there exists a bigger problem which defines the boundary between the game and the engine, it is the target game genre. This carries us over the second question which is the main topic of it. Currently there are many different game engines available in the market that try to be genre-independent however it is also a fact that in reality many AAA game studios design their engines in specific to their target game genre. They even design different engines, sharing the same methodologies and technical aspects but differ in architecture, in order to be used in the development of games with varying genres. So we can conclusively say that the second question has a varying answer, because it can be the case where the engine is extensively modified for a new genre, or it can be the case where a completely new engine is designed that is genre specific. For the third question, we can say it is less discussed because it is the most obvious one amongst them all. Low-level issues have a huge impact on the general outline of an engine's architecture,

because evolving technology pushes developers to think about constantly changing new low-level designs. Even though a good architecture would be affected less by the changes occuring in the low level systems, it is still pretty clear that eventually a modification would be needed on the top levels as the low-levels alter. For example, in the decade of 2000', most hardware only supported fixed-function pipeline for rendering, so the developers designed their high level architecture assuming that fixed-function pipeline would be commonly used for a long time. As the last years of 2000's, hardwares started to support programmable shaders and that changed a lot. In order to cope with the new technology and compete in the market, most developers had to alter their low-level rendering pipeline, thus altering the high-level elements in the architecture to compensate for the changes. For the last question, based on the ideas of Anderson et al. (2008), we can say that even though there are commonly used principles and methodologies, the design and implementation strategy of a game is selected purely based on the game's and users' requirements, and there is no concrete process that works for multiple scenarios.

As Anderson et al. (2008) stated in their research, the questions mentioned needed to be asked while designing a game engine. In doing so, we have asked these questions and came up with particular answers that would draw our line while designing our architecture. For the definition of a borderline between the engine and the game application, we had decided to not to draw one. Since Lina Engine is an open-source project, we aim to make it highly community encouraged and generic, so that there can exists different instances of Lina Engine each with its own borderline between the game and the engine. Instead of basing the architecture of the engine on certain assumptions, mostly about game's genre and requirements, we based the architecture of the engine on certain principles and methodologies, like open-closed principle widely used in software development world, or the pimpl idiom for the abstraction of Lina classes. In doing so, we let different instances of Lina Engine to exists, serving different purposes. This design then answers the second question automatically, Lina Engine's core design is not based on particular game genres, neither it tries to provide a framework to support all kinds of genres. As the answer for the third question about the impact of low-level design, we can state that Lina Engine's core systems does not suffer a lot from modifications that occur in the low-level systems. The design of Lina Engine allows developers to define various instances of the engine completely focused on their own requirements instead of writing plugins and trying to feed them into the core of Lina

architecture. This is possible due to separated architecture and dependency logic achieved via what we call Package Manager and our gameplay design that is based on pure data driven Entity Component System. As a result of this, low-level issues would not have too much of an impact on the core systems, as there exists a pure abstraction between them and user is actually able to define the limits of this abstraction. Lastly as mentioned earlier, Lina Engine does not follow a definitive and concrete design principle in the means of game engine architecture design, but rather it follows a unique design formed through many different ideas existing in academic researches and proven concepts based on well-known material. Along with those ideas, it follows concrete programming principles to achieve the safety and security of the design basis implemented.

## 2.2 Evolution and Evaluation of the Model-View-Controller Architecture in Games

One of the commonly used game engine architectural patterns for the separation of the gameplay code from the engine code is Model-View-Controller (MVC). Upon conducting an investigation based on the concepts of five different game entity and object models, Olsson, Toll, Wingkvist & Ericsson (2015) finds that the evolution of game architectures differ in quality and in order to increase the software quality drastically one needs to carry out the task of architectural refactoring. In addition to the general statement from the Olsson et al. (2015), he also states that, hardware and software technologies for the game development evolve rapidly and the market demands new technologies to be implemented in games. Thus refactoring the architecture and code of the game engines becomes a must. In addition to this, if the game code and the engine code are highly co-dependent, the transition to a newer technology becomes more difficult. For comparison, Olsson et al. (2015) develops five previously developed games, by implementing the same games with the MVC architectural pattern which raises important question; is there a difference in quality between implementation? For the quality model, Olsson (2015) defined five goals. These goals are:

1. The model should provide easy allocation for the developers according to their expertise, for example rendering expert should work mostly on view components.

2. Developers should be able to work on several projects in a seamless way.
3. Games should be portable to different platforms.
4. The services like rendering used should be able to change and evolve with minimal impact on the game itself.
5. The implementation of the new features should be rapid and developers should be able to try to implement their new design or gameplay ideas quickly.

So in short, according to the Olsson et al. (2015), there should be minimal code duplication and the user interface should not be affected by the changes of the subsystems such as rendering, input or physics.

In comparison with the discussed MVC based architecture, Lina Engine does not implement an MVC based architecture. However, instead its architecture is designed in a way that the definition of quality and the goals are quite similar. While designing the architecture for Lina, we had tried to base our quality measurements based on the quality goal definitions of MVC. The API and subsystem abstraction existing in Lina Engine abides by the first quality role to provide easy allocation for the developers as well as the open-source users. In order to comply with the second rule of being able to work seamlessly on various projects, Lina Engine implements a build system, specifically supported by Premake and CMake build systems. In our design we had definitely considered the portability of the games to different platforms, which is the third goal. However even though the architecture of Lina Engine does not prevent any kind of cross-platform compilation, cross-platform development is not a priority in the roadmap, as it will most likely be handled with the build system designed in the future. The fourth quality goal was the services existing in the engine being able to evolve with minimal or no impact on the games developed. Lina Engine follows this rule thoroughly and it uses strong programming paradigms like static polymorphism over dynamic one, in order to implement its abstraction system for the high-level components and game code. The last quality goal, implementation of new features and prototyping being rapid and easy, is the goal that starts to differentiate the paths between Lina Engine and suggested model by Olsson et al. (2015). Aparting from widely used engines that are

easy to use and specifically developed for fast prototyping, Lina Engine does not aim to give users the ability to implement everything in a quick and easy way. Because designing the engine this way would increase the coupling between high-level systems and low-level frameworks, as well as the game code, which destroys many other features we want to achieve for Lina. This does not mean that the engine would be hard to use, but rather we can say it does not try too hard to be usable by everybody. Of course Lina provides a lot of ease to the users, however it is still expected from the user to put in a little more extra effort when compared with other engines in order to achieve system abstractions, varying engine instances and many more features that makes it much more easy to come up with highly optimized game systems.

## 2.3 An Object-Oriented Graphics Engine

Gingko (Qiu and Chen, 2008), an object-oriented graphics engine, developed by Hang Quiu and Lei-Ting Chen in University of Electronic Science and Technology of China, has several features that are aimed at solving design flaws existing in modern rendering engines. Whilst Lina Engine's main purpose is to provide an architecture aimed at solving design flaws existing in modern engines, Gingko is a great product to base ideas upon in the rendering systems for Lina. Gingko focuses on encapsulation and extension layers to provide rendering functionality, while maintaining an easy to use API for the users. The abstraction and system communication systems that Qiu and Chen has achieved with Gingko allows them to provide efficient and fast rendering, while also having an easy to extend architecture that is hard to exploit. The combination of these features are generally very tricky to implement, as most of the user requirements conflict with each other. When developers are focused on providing efficient and fast operations, they usually have to give up some features that build up easy of use and modularity. This paradigm also occurs vice versa, the more abstracted and virtual a system is, the less efficient it becomes due to propagated function calls and delegations. However, this is not the case for Gingko. They mostly achieve their unique architecture with supervision systems they have built. In Gingko, there exists rendering and scene supervision systems that acts as an extra validation layer for rendering operations, thus they are able to collect information about bugs, deficits or user problems into this layer, as well as supervise rendering operations and leave no room for exceptions. This way, when new rendering operations are implemented, it becomes easier to cover the possible issues

that they might bring as well as validate inputs coming from the user in a centralized system. Instead of making the rendering system pure state and data driven which is efficient, or making it purely based on object-oriented patterns which is modular, extendible but lacks performance, they combine these techniques together. The actual operations are done with data and state driven logic, meanwhile an object-oriented layer is built on top of the rendering engine to provide modularity and ease of use, as well as extendibility.

Rendering engine existing in Lina Engine influences from this supervision system to handle scene and entity rendering logic, thus providing a better underlying basis to solve graphics related problems existing in modern engines. Lina Engine already implements a pure data-driven Entity Component System to handle entity interactions and serializations, which makes it pretty viable to implement the rendering systems in a data-driven way as well. However using a data-driven way means that it would be hard to achieve the singularity and modularity of systems that decreases the coupling between them. In order to overcome this, a similar scene supervision system is used in the highest layers existing in Lina that provides modularity, meanwhile the middle and low layers still act as purely data driven, achieving efficiency.

## 2.4 The Entity System Architecture and Its Application in an Undergraduate Game Development Studio

Gestwicki (2012) presents the idea of Entity Component System to handle entity architecture in a game or an engine. Entities are the main building blocks of a game, as they are the elements or objects existing in the game and building up user interaction. Creating a solid entity architecture is a crucial part of designing a game engine, as it is the main building block that would impact the design decisions of other parts of the engine.

As mentioned by Gestwicki (2012) the most commonly used design pattern for entity systems is the traditional object-oriented entity system. In this approach, entities are objects that hold a list of components. Meanwhile components are object classes that have data declarations along with operations on them. Entities with a has-a relationship to this components, or even more traditionally the ones with is-a relationship to them, can then use the operations existing on these components to derive their own functionality. This approach has a great ease of use, but

introduces many complexities and problems as the number of entities increase. Gestwicki (2012) talks about the data-driven Entity Component System that is the solution to these problems and approaches the entity system in a completely different view.

Gestwicki (2012) mentions that 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 completely follow this ECS principle in order to handle entity architecture. This gives Lina the capability to utilize all the features and advantages of ECS that we mentioned. Moreover, the fact that Lina uses a pure ECS approach in its core systems means that not only the game code can benefit from the ECS logic, but the actual systems like render systems, scene graphs and physics systems can also use an ECS based logic in order to handle data and operations. Since all the entities existing in any particular game instance created with Lina would use this ECS system, it would be amazingly easy and fast to transfer entity data throughout the core systems of Lina engine to handle runtime gameplay operations.

## 2.5 Designing a PC Game Engine

Bishop, Eberly, Whitted, Finch, & Shantz (1998) talks about an important design goal to care about while designing a game engine architecture. In their paper, it is highlighted that a game engine should focus on the importance of providing a public API to the end-users, in order to make the engine and the games developed with it more customizable and unique. No matter how important efficiency and fast low-level operations are, a game engine should be designed in a such way that it needs to provide a public layer on top of the low and high levels in the engine in order to make it possible for users to interact with the engine as much as possible. Implementing an API provides this, at the cost of efficiency mostly due to dynamic polymorphism. This makes it possible for users to use the full features of the engine and create unique content. Bishop et al. (1998) mention that a more moderate approach was taken while designing their engine, to provide a common API to the systems & items existing in the engine with proper optimizations under the hood. This common API works along with the object-oriented structure of the engine, making it easier to achieve modularity. Bishop et al. (1998) also mention the possibility of dispensing the API and providing a generic game executable, however this would limit the possibility of interaction between the user and the engine. Hence, this would result in games generally looking like each other, unless the engine developers provide an extensive

customization to the executable in the means of extensions and plugins, which is a cost at manpower and maintenance.

Bishop et al. (1998) talk about a unique scene graph system that is implemented. It is mentioned that their design of a scene graph does not differ from traditional graphs in the meaning of structure, but rather it implements a unique traversal order to iterate through the objects to be drawn in a particular scene. This mentioned scene graph has a way of indexing particular nodes and their children, anywhere in the whole graph tree and keeping a record of particular segments of nodes inside the graph. When a composite node in the graph needs an update, the update is not given to the whole scene graph, as in oppose to traditional approaches, but rather delegated to the particular segment, that propagates the redraw order down the tree only for the relevant leaf nodes. This provides a significant optimization for an object-oriented based structure to handle render device operations. Bishop et al. (1998) mention more about the technical details provided to handle the graph's implementation, however various programming paradigms and techniques have been developed over the course of past 20 years that makes most of those techniques obsolete. Although, the unique model of the design solution still stays viable.

Lina Engine implements the idea of providing a common API to the users and follows the main principle mentioned by Bishop et al. (1998). It has a common API to be used by the clients of Lina Engine. One point in which Lina starts to differ from the idea of API design of Bishop et al. (1998) is that in addition to this API, there is also a mechanism provided along with the API in order to bypass the general interface and directly access the low-level functionality. This creates a unique structure, in which the user is able to use Lina features extensively and easily through the API but also has the possibility to directly access low-level layers like media management, input, audio and rendering. By accessing these frameworks user can customize their game even more and the possibility of creating unique game graphics and feelings increases. One downside to this approach is the fact that it is also a maintenance nightmare for us the developers. In simpler terms to describe the complexity we can say that the process of achieving this is like a tangled web. We need to manage the low-level structure in a way that it is decoupled with the main systems. However, it also needs to be coupled with the API system, which unfortunately works by delegation on top of the main systems. Handling this kind of a system of systems is hard to achieve, and too much of an overhead in runtime. To fix this issue, we plan to implement static

polymorphism that defines the instances of low-level classes in the build time, before the compilation time. This way, references to these instances would not have to get coupled with the API in the compile time, as they would already be referenced while building the engine. Of course, this means an additional build process that needs to be way too generic, and includes multiple steps of operations defined in various executables in order to link the engine library to a game executable. This would decrease the user friendliness, however the achieved structure would be unique and pretty efficient to use.

In oppose to the unique model for the scene graph given by Bishop et al. (1998), Lina Engine does not use this structure. The main for this is the fact that structure is based on the composition design pattern in object-oriented programming approach. As compared to the mentioned scene graph and other traditional scene graph systems, the object-oriented high-level structure of an engine is adapted into the state machine based rendering frameworks, like OpenGL. There occurs many problems in this context, and the mentioned scene graph is an example of a unique solution that solves one of these problems. However in Lina Engine, the core entity architecture is built upon Entity Component System. Since entity architecture is a crucial design point and has an impact on the design decisions of other systems, we are highly tend to implement a data-driven rendering engine in opposed to an object-oriented traditional one. With a cost of a little overhead, we provide an object-oriented API layer on top of this engine to achieve modularity and ease of use. However the fact that underlying systems works parallel with ECS's logic is a huge advantage for us to achieve a fast communication between entities and high-level systems without any kind of virtual delegation. Due to these reasons, Lina Engine differs from the idea of implementing an object-oriented render logic that implements this scene graph. Instead, Lina would have a basic scene graph implementation that can work along with the ECS logic, which would be pretty enough to achieve the performance goals, since ECS would remove much of the overhead by entities.

## 2.6 Distributed Scene Graph to Enable Thousands of Interacting Users in a Virtual Environment

Lake, Bowman & Liue (2010) talk about the idea of a scene graph that can handle thousands of real-time interacting player avatars in a virtual environment. What is called as a virtual

environment is a virtual world that enables people from all over to world to play an avatar simultaneously, interacting with each other and modifying the virtual world with their own creations as they play. As can be guessed, this kind of a virtual system or a game requires a lot of processing power in the means of rendering, thus it requires a really well optimized render system in order to handle all the avatars. Lake et al. (2010) find out that the main barrier to the optimization of such a scene graph is not the storage or communication or other elements, but it is the actors. The main entities that represent each online player are meant as actors. They mention regardless of the capacity of the central server, there will be some point where the system will be fully loaded and the user experience would be diminished, due to the actor load. In order to solve this problem, two main approaches are generally used; sharding and spatial partitioning. Sharding is the process of creating the copies of the same virtual space on different servers to segregate users into various copies. But with this process, load balancing and quality becomes a problem so that there again has to be some kind of a limitation to the players that try to connect to the game at the same time in order to manage shards, the copies. The other approach is using spatial partitioning, which is the process of divining the virtual space across multiple servers and letting each server deal with the load of a smaller space. But with approach, there is a need for a system that implements communications and player transfer between servers as the players change three dimensional locations throughout the game.

Lake et al. (2010) propose a new architecture which they call Distributed Scene Graph (DSG). It is basically a scene graph system that implements spatial partitioning to handle entity data propagation, instead of spatially partitioning the servers each with a normal scene graph. Instead of using a central server to handle the simulation work, DSG delegates it down to the actors around the scene graph. This allows a general purpose environment that is easily scalable only with a cost of additional hardware. DSG works in a way such that there does not exists a game scene with a centralized logic that has to do object-oriented based singular iteration to simulate the space. Instead of that, the scene is a hub of information that connects the actors together. Now the scene would be seperated from the actors and it can freely focus on only data management, just like the Systems in data-driven Entity Component System.

The model presented by Lake et al. (2010) has many similarities with the scene graph that is implemented in Lina Engine. Lina Engine's scene graph logic differs from traditional

object-oriented graphs in a way that it only care abouts the data, and acts as a system existing in the general Entity Component System hierarchy. The actors in the game scene acts as a collection of data, called components, and in the case of a scene graph this collection of data is a group of drawing and transformation parameters. The scene graph works in a spatially divided way, with multiple regions each representing a particular type of draw operation. Actors that have the same procedure of drawing would be clustered together to form a group to be processed by the scene graph. Lina Engine's scene graph logic is highly inspired by the idea of combining spatial partitioning and region usage for load balancing used in the DSG system created by Lake et al. (2010).

# 3. Methodology

## 3.1 Problem Extraction

Initially, a wide research was carried on the game engines and their architectures. This research has begin from early high level frameworks like OGRE, 3D Studio and Torque3D. We had examined several game and graphics engines that are available as open source, and extracted their architecture to inspect. However, in order to find out deficits in the architectures, it was mostly needed to have prior knowledge about the issues that these applications might derive in real life applications. By using our previous experience, along with a research about the obstacles that developers faced amongst the community, it was possible to start establishing connections between the problems and the architectures extracted. Hence, it was possible to see what kind of design restrictions were the cause of issues that occur while using these engines.

We have continued our research in the same way, but for the modern game engines that are currently popular in the market. Just like we did with the early frameworks, we have inspected these modern engines and examined their designs and architecture.Thus, we would have a basis of understanding about their methods, from an outsider perspective. This information was valuable as a resource to use while working on identifying these engines' problems, as well as differencing between real problems existing in these engines versus soft issues that were consciously not fixed and left for a later date.

After the identification of problems existing in early works and methodologies used in modern engines to overcome these issues, it was possible to start extracting architectures of the modern engines and marking out their deficits so that we can establish a basis for solutions. Extracting architectures and inspecting the design of the modern engines like Unity 3D, Unreal Engine, Godot and CryEngine was significantly simpler than doing the same for early engines. This is due to the fact that we as developers had extensive experience on these modern engines, and there are a lot of open source content available online for these engines. Even the source codes of these engines, except Unity 3D, is available as an open source community work. Along with our experience in game development and knowledge about real life applications of these engines, we were able to repeat the process of identifying issues lying underneath the systems within them. Mostly, problems related with heavyweight libraries, event and communication systems, decay of object oriented structures due to overuse and representation of game entities showed up as the most significant issues. It was relatively easy to overcome most of these issues, but the business driven development techniques of the developers of these engines prevented them from working on and overcoming them. Since we were able to mark out problems, and connect them to their architecture design, due to not having an idea of market competence, it was possible for us to alter these architectures and provide solutions to the problems.

## 3.2 Narrowing Issues

Completing our research about early and modern works of engines has resulted with list of many deficits, design flaws and problems occuring in the development process of real world applications. However, it was not possible to address every single issue we had discovered, as it is not realistically possible to provide a software solution or design that works for all. It was mandatory to cross out elements from the list and eliminate issues that were not related to the main architectural flaws. After that, it was possible to design an architecture that focuses on solving the most significant issues.

Upon inspecting a software solution that is as complicated as a game engine, no matter how many professional developers work on the software, it is highly possible to find out bugs, missing features, incompleted implementations and many other issues that can be considered as problems. However, solving these individual issues that are mostly related to the development cycles were

not our goal. Our goal was to find the core reasons for the flaws, mainly in the high level systems existing in those engines. Thus, we initially divided the issues we had found into 4 categories:

- Architectural Limitations
- Known System Issues
- Incomplete Features
- Minimal Bugs

Minimal bugs were the type of issues that the developers using the engine face daily, and they are usually fixed in a short amount of time with patch updates. They were mostly Editor related issues. A window causing editor shutdown upon a sequence of actions, serialization problems like exceptions in file reading and writing across different operating systems, physics bugs upon combination of particular entity actions can be counted as examples of these bugs.

Incomplete features were mostly caused due to the market competence between these engines. When one of them introduces a feature like GPU instancing in particle systems, the other engines rush into implementing this feature even though it is not in their roadmap or is due for later. This quick implementation results in a feature being released in the next version of the engine, but lacking extended functionality and including a lot of other bugs.

Problems we had put into the known system issues category had importance for our design cause this category was the one where we had started to draw the line. Known system issues are problems that cause inefficiency in messaging between game and high level engine systems or introduce complexity and complications by object oriented approach of entity design as the projects get larger. These problems were the ones we definitely wanted to address.

Lastly, architectural limitation category included the biggest issues we aimed to solve. Low speed communication within engine systems, lack of ability to alter the libraries and dependencies of these engines were examples of these issues. These kind of problems were mostly caused by design flaws inside the architecture of those engines and they are very hard to solve without substituting the core systems existing in engine systems.

After narrowing down the issues we aimed to work on, it was possible for us to design an architecture that focuses on removing them and minimizing the flaws that might cause possible problems in the feature.

## 3.3 Designing the Architecture

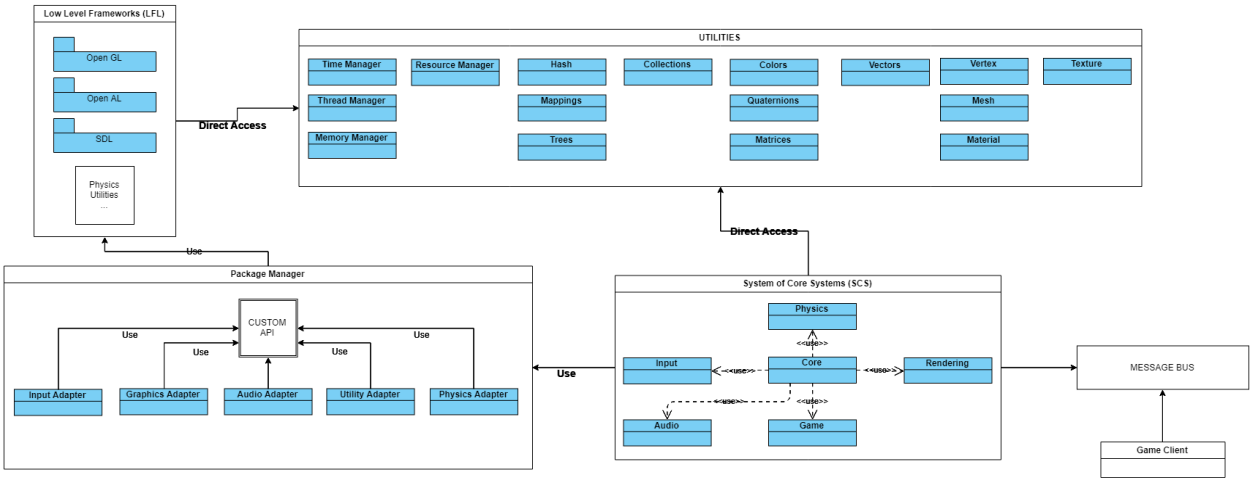
Identifying problems, differentiating between the actual reasons and development cycle based causes for them was a significant and crucial step we had to take. Afterwards, finally narrowing down the problems and extracting the ones that were most crucial was the hardest part of designing our project. After successfully completing these steps, it was fairly easy to do the design, because we had a clear list of issues we wanted to address and proven methodologies that can be used to solve them.

Initially we had thought each major high level system individually, and designed the relationships between the subsystems within them. This way, we would take modularity principles as the lead for the rest of our architecture. Engines existing in the market compete to have the biggest audience, so that the most importing design outcome they focus on is achieving ease of use. This results in systems that are possible to exploit, for the sake of having a large amount of people that are easily able to use those engines. Due to the open source nature of Lina Engine, we do not have any business and audience driven considerations. Hence, we have designed our architecture in a way that is purely solution focused. This way, we were able to come up with systems, that would not have as much ease of use as the engines on the market, but in exchange have solid system structures that were hard to exploit and carry out deficits in the feature.

We have developed several techniques for within engine communication, using our research on game engine architectures, other studies conducted to develop alternative structures, as well as a wide research for the application of various software design patterns. As a result of these suggested techniques, we were able to connect our individual systems within the game engine architecture, by addressing to our biggest goal, which is to achieve lightweight structure.

As another important step, we had designed our data structure for entity representation, using a hybrid approach between delegation based Entity Component System (ECS) and object oriented data communication to achieve a high speed, easy to debug entity system within our core

architecture. Even though most of the engines have plugins and extensions for ECS, there still occurs performance and efficiency related issues while using those plugins because their within engine architecture is based on traditional inheritance based entity representations. We had removed this flaw, by making our engine base on ECS, meaning that even the internal entity relationships between the engine systems would benefit from delegating object and component behaviours into main systems that are responsible for them. This way, using ECS for the game client would not result in incompatible code structures, and we would achieve the efficiency we aimed for the game client code.




In Figure 1, the overview of the architecture we have designed for Lina Engine can be seen. This design suggests us that Lina Engine would be highly modifiable, does not enforce any additional low level frameworks to the users that they do not want to use and implicitly works on ECS systems. After coming up with our design, it was possible for us to start testing possible scenarios on the paper to see if we have achieved what we had aimed for. After finalization, we would be ready to start implementing this design and take initiative on the open source community as a game engine that is specifically developed to solve significant problems.

## 3.4 Technical Details

During the implementation of our design for Lina Engine, there had to be number of technical choices we needed to make. These decisions revolve around many different subjects like build

systems, build toolchains, C++ features, compilation and linking, memory and cpu layouts, cache optimization, hardware buffers, high-level and low-level engine communication and many more. Some of the choices we made are common in most C++ applications, however there are particular combinations of techniques we have used in order to ensure that there will be a pipeline for Lina Engine that is ready to be scaled into an industry level game engine. This section will be highlighting the most important features that we have processed and implemented in Lina Engine in the means of technical details.

### 3.4.1 Static and Dynamic Library

Application wise, Lina Engine is basically a framework, compiled into a library to be used by executable projects. That is one of the nature aspects of game engines, as they are simply libraries that can provide extensive features to develop a game. Widely speaking, there are two ways we can use to build a library. We can either build Lina as a static library, or as a dynamic library. There are pros and cons for both of these choices, along with application cases where both of them can be used. We will shortly discuss the differences between these choices and reason which one of these techniques Lina Engine uses as well as why.

Static libraries are mostly based on compilation time dependencies. They are usually bigger in size, as they include all the value evaluations for a program that can easily be fetched by any executable using it. They are mostly more secure than dynamic libraries, as it is usually harder to decrypt. Nevertheless, this is not an important feature for Lina Engine as it is completely free and open-source. One thing about static libraries that is important is the fact that they are locked into the program using it at the compile time, which requires a recompilation if any changes are made. Even though the recompilation fact is a downside, this compile time dependency usually ensures a better compiler and cpu architecture optimization in most platforms, and results in a faster running executable.

Dynamic libraries on the other hand work differently, as they are dynamically linked into the programs using them at real time. They usually work along with a static library, which is basically the installation of the program, and they provide prototypes and hooks for executables to be used at runtime. Dynamic libraries can be shared between many programs using it at the same time, without creating a copy of the library. This makes dynamic libraries extremely

memory efficient, when compared with static libraries. However, this efficiency is only important in the case where the same library is used by many different programs at the same time, and this is not the case for Lina Engine for now. Theoretically, there would only be a single executable game that is using the Lina Engine. Only one possible case where Lina Engine can be used by multiple programs is using an editor for Lina Engine and also running the game executable at the same time, as they are basically two executables serving different purposes. However, this is not the case for the initial versions of Lina Engine so we do not have to worry about it for now.

Lina Engine currently is build as a static library, as it makes it easier to distribute, more secure and more optimized. Moreover, using a static library ensures that there is minimal amount of risk in the means of our memory manager being robust in various platforms, due to the fact that library is compiled with the same compilation settings and flags with the executable that will be using it. However, there can also be cases where a dynamic library build can be necessary. Especially development wise, it is much faster to compile a dynamic library code than a static library code, so it is much faster and easier to handle the development cycle of a dynamic library. Moreover, as mentioned there can be cases where we would like to distribute render engine, physics engine and the core engine as different dynamic libraries, to be used by various editors made for Lina Engine. For these reasons, we also support dynamic library compilation in Lina Engine. By default it is compiled as a static one, but we also have written an API export feature where all the necessary functions and definitions can be exported outside of Lina Engine, which makes it possible to compile Lina as a dynamic library. So, in short, Lina Engine is build as a static library but also supports dynamic library compilation for scalability, which we hope to be using in the further releases of this project.

### 3.4.2 Vendor Compilation

Lina Engine uses various open-source frameworks and libraries like glfw, OpenGL, glad and Assimp. These frameworks mostly support different compilation features, as they can be used as a static library or a dynamic library at the same time. There are two main ways to use external dependencies like these vendors; compiling them along with the Lina Engine source, or compiling them separately into libraries and using them from inside Lina Engine source.

The former is usually more safer platform wise, as it makes sure that these libraries will be built using the settings of the platform that the source code is currently being compiled on. If you are compiling Lina Engine on Windows and using the former approach, these libraries will be compiled accordingly to Windows requirements, in the means of extensions, compilation flags and target architecture. If we are to switch the platform to Linux, again, the libraries will be compiled accordingly to Linux and the C++ compiler version running on the machine. However, one downside is that this approach is not very user friendly. Once a user builds source files and generates project files for Lina Engine, they will be seeing all these libraries and their relative project files, which can easily extend over 10 different projects for a single library, while developing with Lina Engine. It makes it error prone and confusing, as user would have to deal with compiling these libraries manually themselves.

The latter approach is more user friendly, as no project and extra meta files are included for vendors while compiling Lina Engine from source. In this approach, as the developers we are required to compile these libraries, for different platforms and cpu architectures, as well as different C++ compilers separately. After doing so, we are required to distribute these binaries and explain to the users that they are needed to pick the right version of our distribution depending on their operating system, target cpu architecture and compiler model, as well as the version. This is more of a hard work for us the developers, but it makes it so much easier for end-users since all the building and compilation of vendors would be done by us. Lina Engine currently uses this approach, and in the current stage we are only distributing the vendor libraries for x64 and x86 architectures on Windows machines, build with GCC and Clang compilers.

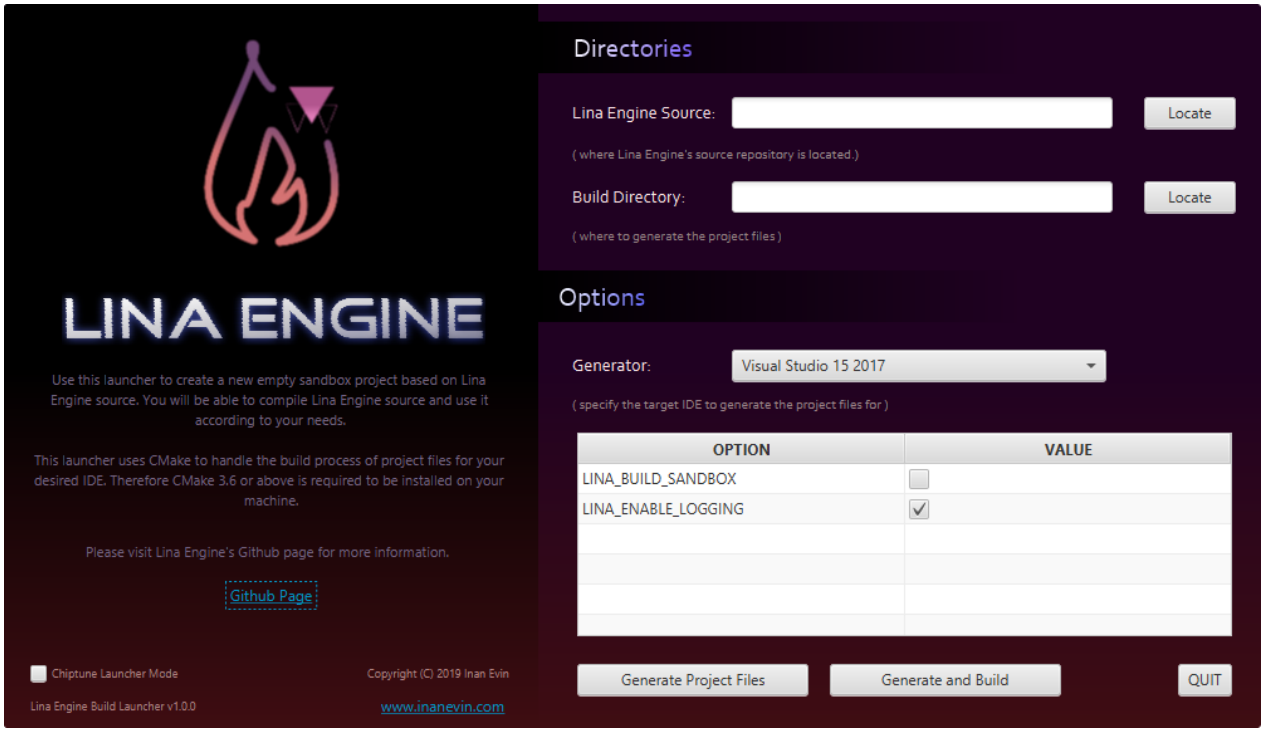
### 3.4.3 Build System - CMake

While developing an open source project like a game engine, there definitely is a need for a build system and toolchain architecture. Otherwise, it would almost be impossible to distribute the sources that can be executable on each platform, as they are many different combinations of operating systems, target architectures and compiler features. In order to avoid this impossibility, build systems like Make and CMake are extensively used by community. By using CMake in Lina Engine, as developers we are able to define how the engine project files should be generated, which compilation flags are needed on different operating systems and architectures,

what kind of libraries are needed to be built and how the directory management, along with file management needs to be handled. We define this by writing different CMake scripts, and requiring users to use CMake which reads these scripts and handles rule management accordingly, in order to generate & compile our source code. This way, no matter which platform, architecture or compiler a user has, we can define universal rules using CMake, and it handles the platform specific build system and toolchain management for us.

### 3.4.4 Lina Build Launcher

Using CMake, as an end-user requires a little bit of knowledge about build systems and CMake commands on the relative terminal. The users are required to use command line or CMake GUI to target Lina Engine source code and build directory, as well as to select different project file generation options we developers define that shapes the Lina Engine source for the desires of the user. Even though learning to use CMake is pretty trivial and easy, we wanted to provide an easier way to use Lina Engine for our end-users. That's why we have developed Lina Build Launcher using JavaFX, for Windows, Mac and Linux.




Lina Engine Build Launcher is a user friendly tool which makes it really easy for users to define the source directories of Lina Engine as well as where they would like to build the source project

files. Users can choose various generators for their project files along with many different build options using an user-friendly interface.

### 3.4.5 C++ 17 Features

Lina Engine makes extensive use of C++ 14 and 17 features. In most systems implemented, like action dispatching systems, event handlers, core engines and entity component systems, Lina Engine uses various C++ 14 and 17 features to provide scalable, robust and optimized code architecture.

Smart pointers are extensively used amongst the systems in Lina Engine in order to handle ownership of object lifetimes. Instead of depending on the raw pointers for ownership, we have chosen to use unique pointers on the objects that would not be shared, like action dispatchers, audio and render devices, physics interaction casters and more. This makes sure that any failure in the destructor call will not cause any memory leaks due to the smart pointer wrappers, as they handle memory clean-up regardless of how the program can fail.

There are many different collection types used in various systems from memory managers, data structure managers, entity components systems and dispatchers. All these devices and systems use hashmaps, arrays, lists and various other data structures. In order to handle proper data transfer between various function calls within the system without extra object copy on the memory, or the creation of dangling pointers, we extensively use move semantics and variadic templates. Usage of move semantics ensures we handle memory blocks efficiently while the variadic templates in most of the classes ensure compile time type checking and ease of use.

Specially in event handling and action dispatchers, we have used constexpr lambdas, folding expressions and lambda expressions of C++17. By using lambdas and various macros along with them, we have created an easy to use API in which the users can easily register callbacks belonging to any class from any instance of it at runtime. Users also able to pass in arguments and specify return types for their own methods to be registered as callbacks upon events occurring within Lina Engine. Thanks to the usage of these features, it was possible to avoid the overhead of std::function objects in most cases.

### 3.4.6 Static Polymorphism in Core Engines

It is no doubt that a game engine framework extensively uses polymorphism to achieve many of the runtime dynamic functionality. This is especially the case for our requirement to be able to extend the engine for multiple platforms and low-level frameworks in the future. For example, Lina Engine currently uses GLFW framework to handle window and context creation, as well as input handling from the hardware. There exists an Input Engine in order to manage input handling. If we were to implement Input Engine purely based on GLFW, it would be really hard to modify the engine to work with SDL, another vendor framework for handling input and media, since it would require deep changes to the engine implementation. To avoid this, obviously we needed to use polymorphism on the Input Engine. Idea is to be able to have many different forms of Input Engine, having the same signature for methods, thus the same API, but implementing those operations differently, completely based on which low-level framework is used. Easiest way to achieve this was to implement dynamic polymorphism using inheritance. However, dynamic polymorphism has an overhead, especially because of what is called vtable lookup. The program needs to go through a lookup table to search for the desired functions and find which instance of a child class of our Input Engine is meant to execute that call. Even though this is a common use case and usually normal to implement, we decided to go a little bit more extreme and optimize this polymorphic structure. Thus, we had implemented static polymorphism.

Core systems in Lina Engine, like Input Engine, Window, Render Devices and more, implement static polymorphism to achieve dependency scalability that is suitable with open-closed principle. We use C++ templates to define super and subclasses, and the types of these templates, meaning what kind of a low-level framework class actually needs to override the operations is resolved at compile time. So the program does not have to worry about lookup tables during runtime, as there are none, all the function calls to the related subclasses will be linked during the compile time. This makes it harder to develop in most IDEs, as they fail to provide hints and features like IntelliSense due to templated class types, however it is worth having a bit more optimized runtime results.

# 4. Results and Discussion

Designing and implementing Lina Engine from scratch has been a tough but rewarding work. We were able to achieve most of our plans when we first designed the architecture. The current open-source version of Lina Engine actually works as intended and in accordance with our design. However, there has been couple of changes to our design we made as we have progressed more in the implementation side of things, mostly due to technical limitations and requirements. In this section those changes would be explained and reasoned, along with some findings related to performance comparison in our Entity Component System.

## 4.1 Design Changes

### 4.1.1 Message Bus

In theory, our design feature to use a message bus within the engine in order to handle in-engine communications as well as low-level and high-level game code communications stays the same. Only difference is that in our design, we had though Message Bus as a central object that would be outliving other system instances and would be decoupled from everything. However, while implementing the engine, we had seen that it was totally unnecessary to define and collect Message Bus operations within a single object as it would introduce unnecessary complexity to the data transportation. Instead, we had implemented the Message Bus operations as a wide network, stretching around the whole engine itself. Various systems carry instances of dispatchers via composition and these dispatchers are used to handle proper communication between various details of game code as mentioned previously in our design. Moreover, important instances of systems and devices are feeded into the game code from the engine itself. This structure acts as a bypasser, as the client game code would not need to use dispatchers and can directly use the references when needed.

### 4.1.2 Package Manager Adapters

Instead of implementing various adapters to provide the main functionality of Package Manager (PAM), we had chosen a more robust approach. Normally, our idea was to integrate low-level

framework management within the engine at compile time via these previously mentioned adapters, however we have seen that again this approach brings a lot of unwanted complexity.

First problem was the extensive use of C++ macros in order to define which low-level frameworks are supposed to be used. All the low-level framework management would depend on the macros, which makes the general system very error prone.

Another problem was static method initialization. In order to implement our initial design, we would have to rely on static methods to instantiate the related objects for the chosen low-level framework settings. Doing so would not have any problems when Lina Engine was built as a static library. However as mentioned before, it is possible to compile and build Lina Engine as both static and dynamic library. In the case of dynamic library build, static methods and the source units that carry their definitions were highly likely to cause problems on different versions of the dynamic library that is installed on the computer. We would have to give too much of an attention to our dynamic library distributions to prevent a DLL Hell, which is the situation created by multiple conflicting DLL files of the same program.

Due to these two main reasons, we had thought about a more safer and robust way that is completely compile time dependent and has no risk of creating a problem during runtime. We use type checking and type definitions in order to define which low-level frameworks are to be compiled and used. Using type definitions, only downside is being have to recompile the engine source whenever a low-level framework dependency has changed. However this is an expected and normal behaviour, as changing the whole media framework of the engine would require a recompilation.

## 4.2 Important Findings

The biggest finding of implementing Lina Engine is the fact that it is 100 percent logical and possible to implement a pure Entity Component System (ECS) based entity structure within a game engine. Most game engines rely on traditional entity-component hierarchy, while providing a backend for ECS type of structure via plugins. We were mostly curious about the main reasons behind it and whether it was possible to bypass the traditional structure. We have found out that it was totally possible to build the whole entity-component hierarchy upon ECS approach and still

have a great performance results even when the number of entities are pretty small. This means that it is completely possible to drive an open source game and engine developer community to contribute into a game engine project, that revolves around new technologies that are coming along with ECS features.

In our machine with RTX 2070 graphics card installed, we were able to achieve stable 60 frames per second on a scene with a single skybox, lightsource and 300.000 cube entities, each having 6 faces, 12 edges and 8 vertices. This is another significant finding for us, as we had only anticipated around 100.000 entities for a stable frame-rate of 60. This proves that it is possible to implement ECS approaches in the Render Engine as well, as this high frame-rate is achieved via making the Render Engine suit to the needs of our Entity Component System architecture running in behind. In practise, this means that we can develop a foundation for an ECS engine that can be scaled up to the industry standards and still have high performance results. Because even though Lina Engine lacks many features when compared to the industry level engines, it achieves this performance levels while running all the necessary subsystems for a game engine. Implementing more features into these subsystems does not mean we would have a reduced performance. Enabling many features at the same time of course would result in a lower performance, however this is a common and expected issue for all real-time graphics applications.

One other important finding is related to our action dispatching and event handling systems. Traditional event and action dispatchers rely on a pure observer design pattern. Event listeners can be attached to and detached from event dispatchers. The only job of the dispatchers is to notify these listeners upon the desired event. The approach has been developed into more sophisticated manners in various applications by creating a pipeline of dispatchers for specific events, so that there will not be an overhead on a central event queue. A widget toolkit developed for C++, called QT, has created a way more clever event dispatching system called Signals and Slots. Again they use the same design approach, however in a much more flexible manner as signals (events) can dynamically change their destination paths (slots) and listeners can define these slots on any object they desire. In Lina Engine, we have created a hybrid event handling system between the traditional approach and QT approach. We used the flexibility of Signals of Slots while keeping the stability and robustness of the traditional approach. Again, we have various dispatchers distributed over different pipelines and they can attach, as well as detach

listeners for themselves. However, instead of sending an event data completely to a particular listener, and letting the listener check for the event and see whether they are interested or not, our system does this check on the dispatchers. The listeners not only tell which events they are interested in, they also give the information of which specific conditions of that particular event they are interested in. For example, an object might be interested in window resize event. Traditionally, whenever the window is resized, an event object would be passed to this listener object. Then the listener is able to check the current size of the window and decide to do something or not. In Lina Engine, listeners can specify the data conditions of these events, meaning that as an example a listener will receive an event callback of window resize event only if current window resolution is bigger than 1024x768. Otherwise, the fact that a window resize event has occurred will not be shared with the listener, providing a more optimized way of sending event data and avoiding unnecessary overhead. We were able to implement this hybir event handling mechanism and use it together with our ECS systems. When we compared with the traditional event handling mechanism, we were able to find out that our system does not introduce any overhead that is more than the traditional one. Our assumption is that it would even be faster than the traditional mechanism as inner-engine communication increases, however in order to prove that we would have to implement more features than we had planned and that is not a case for our initial release. This result means that we still need further tests on our event handling systems in order to see whether it really outperforms the traditional approach or not, however we are sure that in the use case of a game demo, it does not introduce any overhead that is more than the expected.

## 4.2 ECS Comparison

We have compared ECS performance with Unity's ECS and Job System, which are one of the most advertised new features including .NET 4 support for Unity. In Unity scene, we had instantiated 100.000 cubes using ECS, and moved them upwards. Scene does not contain any light source, and the skybox is Unity's default procedural skybox.

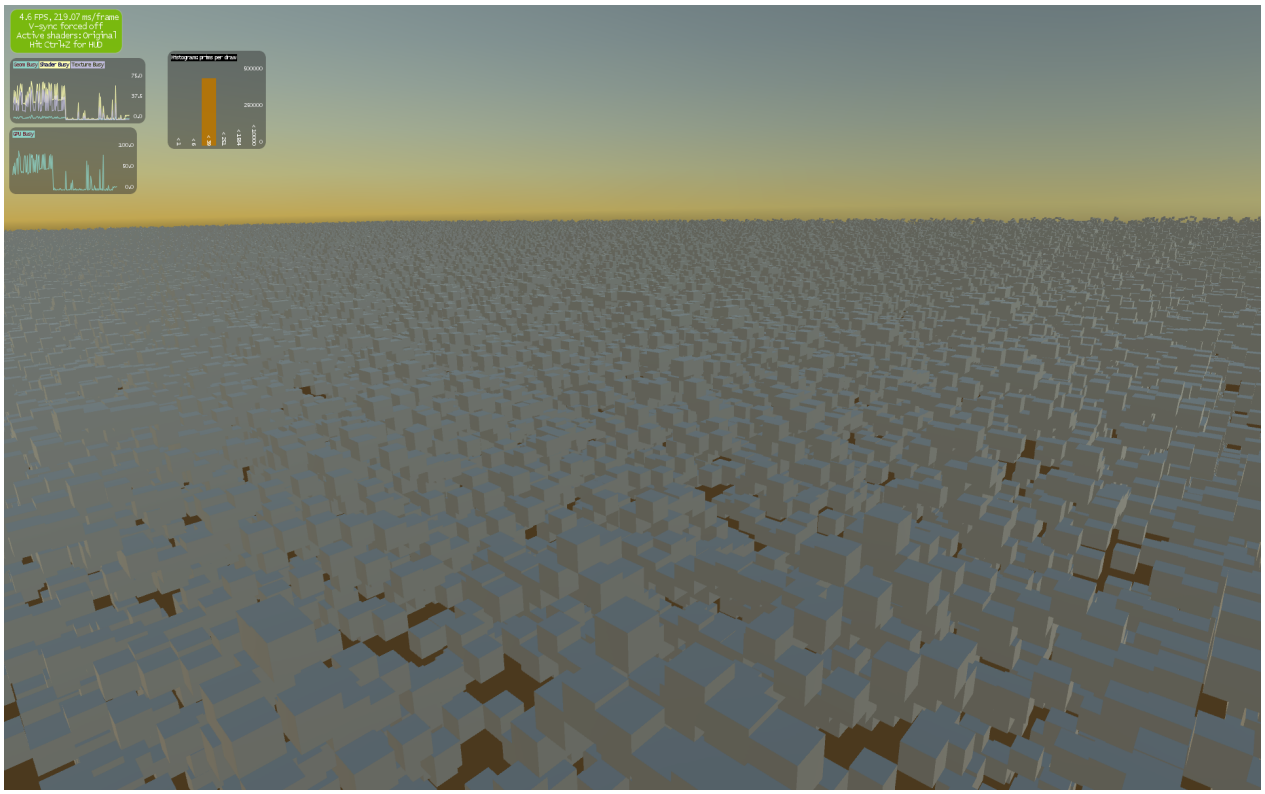



Figure 1 - 100.000 cube entities on Unity Scene

From Figure 1, we can see that at the time of the screenshot the application was running with 4.6 frames per second, 219 ms per frame.

We have built a similar scene using Lina Engine. Only big difference is the cube's movement behaviour and the skybox. We had used a cubemap skybox, which should not differ significant enough to mention from Unity's default procedural skybox. Other than that, instead of moving the cubes upwards, we haved moved the cubes in a forest ruth motion, giving them a constant explosion effect with a particular momentum. This movement is much more CPU heavy as it calculates random interpolation each frame.

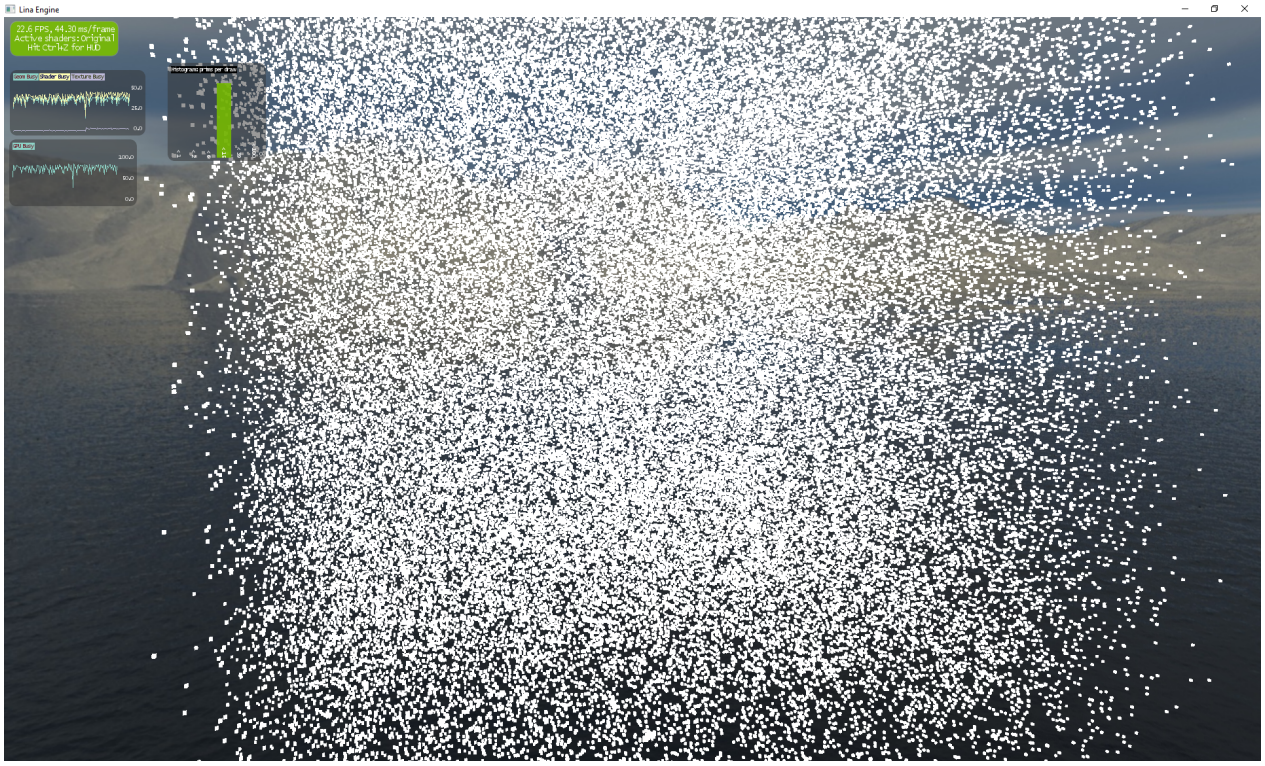



Figure 2 - 100.000 cube entities on Lina Engine scene

From Figure 2 we can see that the application runs with 22.6 frames per second, with 44.3 ms per frame. This is much more optimized and faster than Unity's ECS system, and it was a great achievement for us.

Both applications were debugged using Nvidia NSight Debugger, with the same debugging settings. Both were run on the same computer, using Nvidia GTX 970 graphics card with 4 gigabytes of memory.

# 5. Conclusions

The development journey for the initial version of Lina Engine has taught us about many significant issues. The first one is the importance of architecture design if the case is to develop a project as large as a game engine framework. During the first semester, we had iterated over many different design methodologies and tested various trivial implementations. All these tests have led us with our final design, that we mostly seen as the final architecture. However, even though we had run many iterations it was still a really hard work to carry on with the

implementation that goes alongside the design. From this, we can safely say that most of our time was spent on designing and no matter how many hours were spent the design was still the biggest issue we had faced.

One other important thought of us is the fact that lack of domain knowledge is a huge obstacle for a development process. As the developers of Lina Engine we were previously associated with game development and plugin development for various game engines. Even though we had been closely together with game technology and software, it was still a pretty hard work for us to grasp the concept of game engines and how things work under the hood. We had found out that we had implemented the same system over more than 10 times, each having a better technical quality. This was due to the reason that we had lacked previous experience on engine development and was not aware of the significant techniques used to do so.

For the development of Lina Engine, we have made an extensive literature research, inspected previously related work, current game engines and their sources. Then according to our findings, we had designed an architecture over many iterations, to be implemented with various different techniques. We had compared our implementation techniques and chose the best results according to many factors ranging from performance to user-friendliness. Our methodology was extensive and fit for a game engine development, however we had found out that it is possible to improve this methodology even further. One important thing that can be done was to increase the number of design and implementation iterations. Due to time limits, we were not able to test various final architectural designs. We were only able to implement trivially and test our intermediate architecture designs that has led us to our final design. However, we think that if we were able to come up with two or more architecture designs that differ in the key concepts of user experience, and trivially tests those designs, we believe we would have implemented a more user friendly game engine architecture. Currently, even though Lina Engine offers a vast API for its end-users, it still lacks many user experience features that would make it a fit for community use. We plan to improve this side of Lina and implement more features in order to make it possible for use by larger audience.

# 6. References

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).

M. Doherty. A software architecture for games. University of the Pacific Department of Computer Science Research and Project Journal, 1(1), 2003.

Munro, J., Boldyreff, C., & Capiluppi, A. (2009). Architectural studies of games engines — The quake series. 2009 International IEEE Consumer Electronics Society’s Games Innovations Conference. doi:10.1109/icegic.2009.5293600

“Godot Engine - A look at the GDNative architecture”, as appears on godotengine.org/article/look-gdnative-architecture, accessed on October 26th, 2018.

“Introduction to Godot development” as appears on docs.godotengine.org/en/3.0/development/cpp/introduction_to_godot_development.html, accessed on October 26th, 2018.

“Tombstone Engine” as appears on https://tombstoneengine.com, accessed on October 26th, 2018.

Anderson, E. F., Engel, S., Comninos, P., & McLoughlin, L. (2008). *The case for research in game engine architecture. Proceedings of the 2008 Conference on Future Play Research, Play, Share - Future Play ’08.* doi:10.1145/1496984.1497031

Ollsson, T., Toll, D., Wingkvist, A., & Ericsson, M. (2015). *Evolution and Evaluation of the Model-View-Controller Architecture in Games. 2015 IEEE/ACM 4th International Workshop on Games and Software Engineering.*

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

Gestwicki, P. (2012). *The entity system architecture and its application in an undergraduate game development studio. Proceedings of the International Conference on the Foundations of Digital Games - FDG ’12.*doi:10.1145/2282338.2282356

Lake, D., Bowman, M., & Liu, H. (2010). Distributed scene graph to enable thousands of interacting users in a virtual environment. 2010 9th Annual Workshop on Network and Systems Support for Games.doi:10.1109/netgames.2010.5679669

Kanode, C. M., & Haddad, H. M. (2009). Software Engineering Challenges in Game Development. 2009 Sixth International Conference on Information Technology: New Generations. doi:10.1109/itng.2009.74

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

C. M. Torres-Ferreyros, M. A. Festini-Wendorff and P. N. Shiguihara-Juárez, "Developing a videogame using unreal engine based on a four stages methodology," *2016 IEEE ANDESCON*, Arequipa, 2016, pp. 1-4.

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.

INDRAPRASTHA, Aswin; SHINOZAKI, Michihiko. The Investigation on Using Unity3D Game Engine in Urban Design Study. **Journal of ICT Research and Applications**, [S.l.], v. 3, n. 1, p. 1-18, Sep. 2013. ISSN 2338-5499. Available at: <http://journals.itb.ac.id/index.php/jictra/article/view/180>. Date accessed: 06 Jan. 2019.