Report on

My Game Engine

By Sai Anirudh Yellanki

1 ABSTRACT

A Game Engine is a platform to develop games more effortlessly. This project aims to build one and optimize it for performance. A Game Engine consists of three main components, a Math engine, a Rendering engine, and a Physics engine. The math engine consists of 3d vectors, matrices, and rotation primitives. Optimizing it is crucial for a game engine's performance as it needs to process large amounts of data (Shaders, Vertices, and Math Calculations) in a small amount of time. So, to parallelize the math engine, I used the concept of Single Instruction Multiple Data (SIMD) with the help of Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) Intrinsics. I have built SIMD instructions on top of the serial math library. The SIMD performance improvement helps use underlying hardware more efficiently and reduces power consumption. The Rendering engine has also made a lot of progress with the lighting model setup. I developed the lighting model using Phong Lighting. I plan to improve the performance by using different parallelization techniques such as instruction-level parallelism, data-level parallelism, and task-level parallelism in the rendering engine. I also plan to implement the Physics engine and parallelize that, too, eventually making a full-fletched Parallel Game Engine.

CONTENTS

A Parallel Game Engine

Sai Anirudh Yellanki

Dec 2021

18XJ1A0548

2 Introduction

Nowadays, games are becoming a multi-billion-dollar industry competing with the massive Hollywood film industry in popularity. And the software implemented that drives these three-dimensional worlds - game engines like Electronic Arts' Frostbite Engine, Epic Games' Unreal Engine 5, and Valve's Source engine - have become fully reusable game engines that can be licensed and be used to build almost any game that the game engine can make.

The game engine usually varies a lot in its architecture and implementation. However, a few recognizable patterns emerge through licensing a game engine and their in-house counterparts. Coming down to the core, they all have a set of components, including the rendering engine, the collision and physics engine, the animation engine, the Audio Engine, the game world object model, the artificial intelligence engine, and so on.

The idea to build a game engine is that most of the AAA games have a dedicated game engine just made for that game. This is because a generic game engine made to serve all the games might face the problem of not being good enough for a particular game. Therefore, for a game to be excellent, the game engine is made to satisfy all the technical and logical requirements for that specific game.

Going forward, being one with a big dream, I want to establish a company to help people get into game development. To do so, I need to know how I can make a game engine, and I should be able to make the best games out of it to inspire people. This project can help me grow as a game developer and help me towards my dream.

The goal of this paper is to build a Game Engine. To do so, I started making it component-wise. First, the Math Engine has vector math, matrices, quaternions, and Euler angles. I try to achieve that -

- My math engine performs as well, if not better than the industry standard math engine.
- Implement SIMD into my math engine to get an additional speedup on machines using SIMD.

- Refine my math engine to fit the needs of a Game Engine if needed.

Coming to the Rendering Engine, I intend to -

- Render objects onto the screen.
- Build a lighting system for the scene using the Phong Lighting Model.
- Develop code for directional and omnidirectional shadows in the scene.
- Be able to add at least 20 lights to the scene.
- Be able to control the camera using the keyboard and mouse.

I have the physics engine. I intend to -

- To develop collisions between the objects rendered onto the screen with the help of collision masses.
- Further, expand on it by implementing complex physics calculations like water and cloth simulation, kinematics, etc.

Finally, I need to perfect the low-level systems as they are the foundation for my game engine.

The ultimate goal of this paper is to build a game using the developed game engine. While this is still far from achieving that, I believe that this would mark the completion of this project. I am willing to continue to work on it.

3 Problem Statement

To make a Game Engine from scratch. First, build all the Game Engine components like Math Engine, Rendering Engine, Physics Engine, and Low-level systems. Then, parallelize the math engine using SIMD Intrinsics and the rendering engine using techniques like instruction-level parallelism, data-level parallelism, and task-level parallelism to improve performance.

3.1 Problems I look to answer

- How fast is the serial 3d math library, and how much speedup can we achieve?

- Can the math and rendering engine handle the amount of data that a modern game engine can run?

- How much benefit can we get from parallelizing the rendering engine?

- Can we make games in this Game Engine?

Building a Game Engine is not an easy task as very often, there are hundreds of people working on one game engine. As mentioned before, we need a way to implement the Engine efficiently. When the game engine cannot exploit parallelization in some cases, it should still match industry standards. As 3d math and low-level systems are the core for any game engine, we need to get its implementation logically correct and computationally less expensive.

When we have an efficient implementation of them, we need to parallelize the code, more like, develop a parallel code so that, when the Engine can use parallelization, it will speed up the code. Using SIMD, theoretically, we observe a 4x times speedup, but due to hardware constraints, we should see at least 3x times speedup. We need to answer whether we will be able to achieve it.

Implementing the math and low-level systems, we go on to the rendering engine. Considering the number of calculations needed in the rendering engine, we need to ensure that combined math and rendering Engine can handle the amount of data a modern game engine can. The insurance could be the difference between a game struggling to run at 20 frames per second and running effortlessly at 60 frames per second.

When parallelizing the rendering engine, we need to ensure that we avoid unnecessary use of resources and, more importantly, make sure the program doesn't crash. Then comes the physics. In a high-end game, physics calculations are the most computationally intensive. Therefore, writing lousy code would mean that our game is sure to lag.

Getting all these implemented, we need to make a game out of it because the primary purpose of a game engine is to make games. For that to happen, we need to develop functionality to let the user quickly build a game world and code the logic for the game. Then, the user should ship it to PCs or consoles for others to play the game. That would mark the end of this project.

4 Background and Related Work

The following are a few of the references I used to understand more about the core of a Game Engine. These range from math to rendering to parallelization techniques. These helped us a lot to get started and make things concrete as I was progressing through our project.

4.1 Math Libraries like GLM

GLM is the OpenGL math library that every regarded as one of the faster math libraries for 3d math and rendering. It consists of 3d vector math and matrix math that can be used on the go by just including a header file in your project. GLM has some SIMD optimizations based on compiler optimization techniques. These optimizations will be automatically applied. For example, if we compile a program using /arch: AVX, GLM will automatically detect it and generate a SIMD code using SSE/AVX instructions when a system can utilize SIMD.

Implementing SIMD has improved its performance a lot, and I did so too. So I will surely keep improving the code and improve the performance.

While this library is something that I could have used, it is essential to note that a 3d math library for a game engine is much more than just 3d vector, matrix, and camera math. It involves primitives like rays and also physics calculations. So using this library was an option, but when I develop my math library, I get to know the intricacies of each function, and I can build on it or change it as I want it rather than getting stuck when I want to change it if I use GLM. And also, it is better to keep the math code in one place rather than importing a few functions and coding a few functions.

4.2 3D Rendering software

There are a lot of 3D rendering software out there that provide high-level graphics and allow us to model objects and texture them. While these might give high-level graphics, they don't allow us to implement logic to build a game. Even though these applications are not intended for making games primarily, I believe this level is something I would want to achieve while providing the functionality to make games. One of the rendering software is Blender 3D.

4.2.1 Blender

Blender is a free and open-source 3D graphics software. It supports the whole 3D pipeline-modeling, rigging, animation, simulation, rendering, etc. Blender renders using the render engine called Cycles. It is a physically-based renderer developed by the Blender Organization. It is designed to provide physically-based results out-of-the-box, with artistic control and flexible shading nodes.

4.3 A Study on SIMD Architecture

In this paper, the writers studied the use of SIMD architectures and learned their effects on the performance of specific applications. They chose to test matrix multiplication and Advanced Encryption Standard (AES) encryption algorithms and modify them to exploit SIMD instructions. The performance improvements using the SIMD instructions are analyzed and validated by the experimental study.

4.3.1 SIMD Architecture

In the paper, the writers talk about the famous architecture of AltiVec, which is used by many companies, including Apple and Motorola. AltiVec has source and destination registers to store source and destination vectors. Source registers hold the operands, while the destination registers hold the value of the result. A Vector permutation is performed on the registers.

In the paper, the writers then briefly went over the MMX/SSE architecture introduced by Intel. It is capable of SIMD operations for integers and floating points with

the new MMX series of processors. Finally, they used Katmai's new instructions (KNI) for the new processors. This is the architecture I used in my Math library.

4.3.2 Matrix Multiplication

In the paper, the writers first implemented naive matrix multiplication. Then, applying SIMD instructions could reduce the number of executed instructions. Also, they achieved noticeable performance improvement. It was about two times faster than the original algorithm.

4.3.3 AES Encryption Algorithm

The writers took a cipher of variable block and key length. The cipher block can extend both the block length and key length to multiples of 32 bits. They implemented the AES encryption by coding the algorithm serially in C++ and then changing the instructions to SIMD and observing speedups. The optimization of the AES algorithm helped them observe a massive 23 times speedup.

This paper shows us how to exploit SIMD to speed up the code. Unfortunately, while this implements SIMD and offers a way to help us implement it, it doesn't apply to our problem of making the math library use SIMD.

4.4 Unreal Engine

Unreal Engine is a top-tier game engine that helps you make a wide variety of games, starting from First-person shooters to Role-playing games. It is made up of several components that work together to make a game works. Its massive tools and editors allow you to organize and manipulate them to create the gameplay for your game.

Unreal Engine includes a rendering engine, sound engine, physics engine, input and gameplay framework, and an online module.

4.4.1 Rendering Engine

The graphics engine then undergoes massive calculations in the background using all this information before it can output the final pixel information to the screen. The power of a graphics engine affects how realistic your scene can look. The Unreal graphics engine can output photorealistic qualities for your game. Its ability to optimize the scene and process massive calculations for real-time lighting allows users to create realistic objects in the game. Users can make use of this engine to create games for all platforms. It supports DirectX 11, OpenGL, and JavaScript/WebGL rendering.

4.4.2 Sound Engine

The sound engine is used for managing the music in a game. It ensures that the music and audio files are handled properly. It allows you to play various sound files to add realism and imbibe a mood to the game. Sound is one of the most important things in a game. For example, sounds are constantly in the background. Sound effects can be played repeatedly when needed, or specific events in the game trigger one-off and are triggered by specific events in the game.

4.4.3 Physics Engine

The Unreal physics engine uses the PhysX engine developed by NVIDIA to perform calculations for physics calculations that need to be performed to maintain that realism. Its presence keeps out the necessity for users to work on physics rather than develop the game.

4.4.4 Input and Gameplay Framework

Unreal Engine has an input system to handle inputs given by the player and translate them into something like character movement. The Gameplay framework in Unreal contains the functionality to track game progress and control the game's rules. User interfaces (UIs) are part of the Gameplay framework to provide feedback to the player during the game. Gameplay classes such as GameMode, and PlayerState set the rules and control the state of the game. The in-game characters are handled either by players or AI. Whether influenced by the player or AI, in-game characters are part of a base class known as the Pawn class. The Character class is a child of the Pawn class that is made explicitly for player representation that is vertical, for example, a human.

4.4.5 Lighting and shadows

Unreal Engine 4 enables you to add a set of basic lights that could be added to your game level. The set of lights is directional lights, point lights, spotlights, and skylights. Directional light produces beams of parallel lights, Point Light produces light like a light bulb, Spot Light produces light in a conical shape, and Sky Light produces a light similar to light from the sky on the objects in a scene.

In a scene, you can have two types of shadows in the game world. The static and dynamic shadows. While, Static shadows can be added to the scene beforehand into the scene, so, they are easy and fast to render, Dynamic shadows change during runtime, making them more expensive to render.

4.4.6 Artificial Intelligence

Unreal Engine 4 provides NPCs with a basic AI that lays the foundation for the users to improve the AI of the NPCs while customizing it in your game. NPCs can be given some form of AI to interact with them. For example, it can provide NPCs with the ability to find a sweet spot to attack. If attacked, they will run, hide, and find a better position to fight back.

4.4.7 Online and Multiplatform

Unreal Engine 4 provides the compatibility to create games for many platforms. If you make a game using Unreal Engine 4, it is portable to different Web, iOS, and Android platforms.

There are tons more things that Unreal Engine has, and I am just getting started. Our goal is to try and get as close as possible to this Engine in terms of the sheer performance that this Engine offers. This is considered one of the best open-source engines there is. One could argue that you want to make a game engine when one already exists. That is because

Unreal Engine and any open-source engines are designed to build specific games. For example, Unreal can create games with high-end graphics and is more suited for RPG-style games. Unity is versatile but cannot produce as much realism as Unreal Engine. Therefore, to make your own game with preset functionalities to fit your style, you need a Game Engine developed by yourself. This is precisely why I decided to build my own Game Engine.

4.5 An introduction to parallel rendering

This paper shows a way to render graphics parallelly. It talks about the issues related to the design of parallel renderers and explains how to use parallelization in rendering applications.

4.5.1 Parallelism in the rendering process

The writers in the paper propose a few different ways in how parallelism can be achieved. Firstly, functional parallelism exploits the functional structure of the engine. Then comes Data parallelism that deals with splitting the data and working on it simultaneously. Lastly, temporal parallelism is where different frames are rendered by different processors.

4.5.1.1 Functional Parallelism

The paper talks about how we can use the concept of function level parallelism. A rendering pipeline is formed if a processing unit is assigned to each function and a data path is provided from one unit to the next, like modeling transformations, back-face culling, lighting calculations, viewing transformations, etc. As a processing unit completes work on one data item, it forwards it to the next unit and receives a new item from its upstream neighbor. Once the pipeline is filled, the degree of parallelism achieved is proportional to the number of functional units.

The paper also points out two significant limitations in this approach. First, the overall speed of the pipeline is limited by its slowest stage, so functional units must be designed carefully to avoid bottlenecks. And secondly, the available performance gain due to parallelism is limited to the number of steps in the pipeline.

4.5.1.2 Data Parallelism

In this approach, the data is split into multiple streams, and the processors operate on several items simultaneously by replication several identical rendering units. There are two different places where this type of parallelism can be used, and they need to be balanced in order for us to observe a significant speedup. Object parallelism refers to operations performed independently on the geometric primitives that comprise objects in a scene. For example, if there are two cubes in a scene, one processor works on one cube, and another processor works on the other one. Image parallelism occurs in the later stages of the rendering pipeline and includes the operations used to compute individual pixel values. For example, the screen may be split into several parts, and each processor works on one part simultaneously. Obtaining the proper balance between these two phases of the computation is difficult since the workloads involved at each level are highly dependent on factors such

as the scene complexity, average screen area of transformed geometric primitives, pixel sampling factor, and image resolution.

4.5.1.3 Temporal Parallelism

In this type of parallelism, the problem is converted to the time domain, and each processor is given a set number of frames to process rather than splitting data or objects in each frame. We say that all processors take almost the same amount of time when the frames are temporally parallel, that is when the image that is being rendered doesn't change too much.

This paper provided us insight into how we can go forward to implement a parallel rendering engine. We will most probably try and use the techniques mentioned in this paper going forward

5 Implementation

Implementation of a Game Engine deals with implementing 5 major components. They are the Low-level systems, Math Engine, Rendering Engine, Physics Engine, and User Interface. While there are many other components like Artificial Intelligence systems, Sound Engine, Animation systems, etc. The five mentioned are the core components for a Game Engine to work. I have successfully implemented the Math Engine and am working on the rendering engine. I look to parallelize the rendering engine as I move forward. While doing so, I will continue to build on the low-level systems and optimize them. Once these are ready, then comes the User interface followed by the Physics engine. Here is a brief about how they work and are implemented.

5.1 Low-Level Systems

The foundation for any program is low-level systems like logging, debugging, profiling, asset manager, file manager, event system, etc. Implementing them is essential for a game engine.

5.1.1 Logging System

A logger or a logging system logs out useful messages to inform the user about some valuable information or any error. The logger helps identify any potential issues with the code or the Engine itself. A Logging system is critical to keep the engine code intact and make sure the user is informed about any changes and errors with the Engine.

- **Info** - Designates informational messages that highlight the application's progress at a coarse-grained level.
- **Trace** - Designates fine-grained informational events that are most useful to debug an application.
- **Warning** - Designates potentially harmful situations.
- **Error** - Designates error events that might still allow the application to continue running.
- **Fatal** - Designates very severe error events that lead the application to abort.

I plan on using a logging library called spdlog. It satisfies all the requirements and also will exploit multi-threading when possible. This is considering one of the fastest open-source logging libraries.

5.1.2 File and Asset Manager

Games are, by nature, fantastic multimedia experiences that get the player immersed in the magic. Therefore, a game engine needs to be able to load and manage a wide variety of media - texture maps, 3D mesh data, animations, collision and physics data, game world layouts, and the list goes on. Moreover, the memory in a game engine is usually limited. Therefore, it needs to save a file reference rather than make copies of the object.

A game engine's file system API typically addresses the following areas of functionality:

- Manipulating file names and paths,
- Opening, closing reading, and writing individual files,
- Scanning the contents of a directory,
- Handling asynchronous I/O requests

My Game engine is yet to include a file manager as it is still in the initial stages of development.

5.1.3 Event System

For every application, there needs to be a way to handle events like click or key events. Game Engine is similar. We need to handle events like

- Window events - Include closing, resizing, sizes, etc.
- Mouse Events - Include Pressed, released, right-click, double click, hold, etc.
- Keyboard events - Include Key pressed and released, Key codes, etc.

The way my Game Engine works is by Input polling. It means, the application keeps looking for any events and when it encounters an event, it has the code to handle it.

5.1.4 Support systems (Scheduling)

A Game Engine needs to schedule tasks at a rapid rate like an operating system, as it needs to handle a lot of data and process it while handling all the interrupts and input events.

For scheduling, we follow a simple brute force method to start up or shut down subsystems like the RenderManager which is responsible for rendering. The start-up and shut down functions take the place of the constructor and destructor, and in fact, we should arrange for the constructor and destructor to do absolutely nothing else. That way, the start-up, and shut-down functions can be explicitly called in the required order from within main().

```
class RenderManager {

public:

    RenderManager() { // do nothing }
```

```
~RenderManager() { // do nothing }
void startUp() { // startup the manager... }
void shutDown() { // shut down the manager... }
}
```

5.1.5 Profiler

Games are real-time systems, so maintaining a high frame rate per second is essential. Therefore, a heads-up display is provided, which shows up-to-date execution times for each code block. The profiler provides the data in various forms, including a number of cycles, execution times, and percentages relative to the execution time of the frame. A profiler for my Engine has not been implemented yet, but here is an image from Unreal Engine.

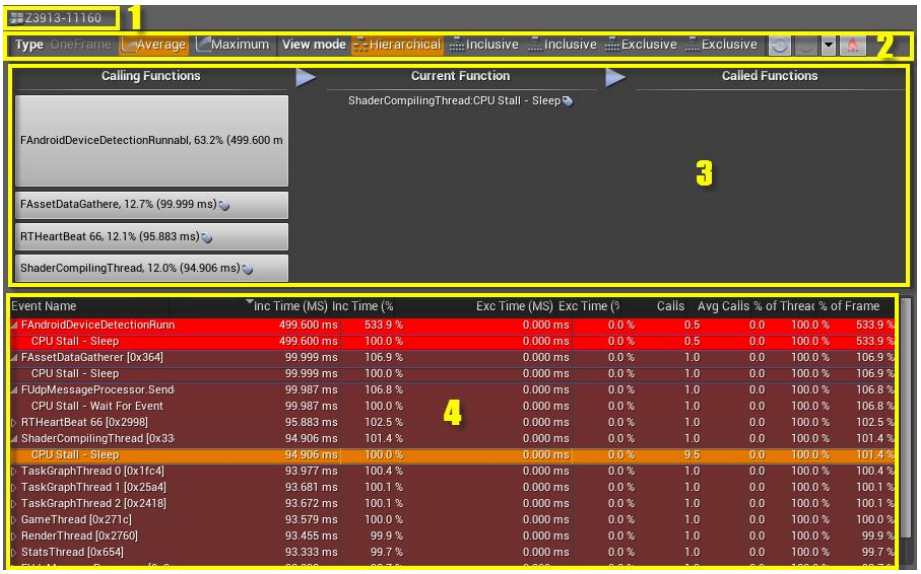

Figure 1: Unreal Engine Profiling Tool

5.2 Math Engine

3D Math is the core of any game engine as we need to handle vectors that represent points on an object inside a game. Most modern 3D games are made up of three-dimensional objects in a virtual world. A game engine needs to process all the positions, orientations, and scales of all these objects, transform them into screen space to be rendered on screen. In games, 3D objects are mostly made up of triangles, which are represented by points, aka vectors.

5.2.1 Vectors

A vector is a quantity that has both a magnitude and a direction in n-dimensional

space. For example, a 3D vector can be represented by a triple of scalars (x, y, z), and its unit vector form represents direction, while the magnitude represents its length. A Vector represents every point in a scene. Implementation of a Vector class requires some operations.

- **Basic Operations (Addition, Subtraction, Scalar Multiplication):** These operations are defined on an element-wise basis.

  $$a + b = [\ (a.x + b.x), (a.y + b.y), (a.z + b.z)\ ]$$

  $$a - b = [\ (a.x - b.x), (a.y - b.y), (a.z - b.z)\ ]$$

  $$c * a = [\ (c * a.x), (c * a.y), (c * a.z)\ ]$$

- **Dot Product and Cross Product:** Dot products can be used for projections, knowing the orientation of vectors, the angle between vectors, etc. Vector Products are used to calculate normals (Perpendicular vectors of a plane) which will be required for various functions like lighting and shading.

  $$a \cdot b = a.x * b.x + a.y * b.y + a.z * b.z = \text{scalar (Dot Product)}$$

  $$a \times b = (a.y * b.z - a.z * b.y)i + (a.z * b.x - a.x * b.z)j + (a.x * b.y - a.y * b.x)k$$
  (Cross Product)

5.2.2 Matrices

Matrices used inside game engines are usually 3x3 (Rotation Matrix) and 4x4 (Transformation Matrix). They help in translating, rotating, or/and scaling a vector. This means a whole object can be transformed using operations on each vertex in the object. Implementation of a Matrix class requires some operations.

- **Basic Operations (Addition, Subtraction, Multiplication, Matrix-Vector Multiplication)**

- **Translation, Rotation, and Scale -** A matrix can be used to "transform" a point/vector. This is called a transformation matrix. It is usually a 4x4 matrix to hold data for rotation, translation, and scale.

- **Applying transformations -** Applying transformations such as Translate(1,1,1) and then Rotate(90,45,30) is as simple as multiplying the matrices containing those transforms together. to do the above transformation, RotMatrix(90,45,30) should be on the left-hand side in the multiplication. This means the matrices have to be multiplied in the reverse order to properly apply the transformations.

5.2.3 Euler Angles

It is used to represent the rotation of a vector. A rotation defined via Euler angles consists of three scalar values: yaw, pitch, and roll. While they are easy to visualize and use, Euler angles are prone to a condition known as gimbal lock. A 90-degree rotation causes one of the three axes to merge into the central axis. For example, a rotation by 90 degrees

about the y-axis, the z-axis merges onto the x-axis. This prevents any further rotations about the original z-axis because rotations about z and x have effectively become equivalent.

5.2.4 Quaternions

We've seen that a 3 × 3 matrix can be used to represent an arbitrary rotation in three dimensions. However, a matrix is not always an ideal representation of a rotation, for a number of reasons:

- In a matrix, we need 9 floating-point values when we actually have only three degrees of freedom, roll, pitch, and yaw.
- When rotating a vector using matrices, we would be needing 3 dot products, nine multiplications, and six additions. That is a lot considering the number of operations usually performed in a rendering engine.
- We need some way to smoothly interpolate between two points. For example, suppose we want a way to animate a camera from some starting orientation A to some final position B over the course of a few seconds. In that case, we need to be able to find lots of intermediate rotations between A and B throughout the animation using matrices.
- Also, when it comes to Euler angles, they are prone to gimbal lock.

A Quaternion overcomes all these four issues. A quaternion looks a lot like a four-dimensional vector, but it behaves quite differently.

$$q = [\ q.x\ q.y\ q.z\ q.w\ ].$$

- **Basic Operations (Addition, Subtraction, Dot product)**
- **Quaternion Product -** Quaternion product is defined a bit differently compared to a vector or a matrix product. Here is the equation.
- **Conjugate and Inverse -** Inverse is defined as conjugate divided by magnitude squared. Conjugate is usually denoted q* and it is defined as follows:

  The Inverse is usually denoted q^-1 and it is defined as follows:
- **Rotating a vector -** $v' = \text{rotate}(q, v) = (q)\ (v)\ (q^{-1})$**.**

5.2.5 SIMD Intrinsics

SIMD stands for Single Instruction Multiple Data. This concept helps us perform a single mathematical operation on multiple data items in parallel, using a single instruction. SIMD is used in game engine math libraries because it permits vector operations such as dot products and matrix multiplication to be performed very fast. I have used SIMD to speed up the calculation.

- **SSE Registers -** SSE is the most commonly used mode for performing these calculations. These registers have four 32-bit floating-point values packed into a 128-bit SSE register.

- **__m128 variable -** __m128 is a predefined variable in Visual Studio Code that mimics the SSE registers. It has 32-bit floating-point values packed into a 128-bit SSE register.
- **Alignment of __m128 variables -** When an __m128 variable is stored in RAM, it is the programmer's responsibility to ensure that the variable is aligned to a 16-byte address boundary.
- **Sample Vector Addition code -**

```
__m128 addWithAssembly(
    __m128 a,
    __m128 b)
{
    __m128 r;
    __asm
    {
       movaps xmm0,
        xmmword ptr [a]
       movaps xmm1,
        xmmword ptr [b]
       addps  xmm0, xmm1
       movaps xmmword ptr [r],
        xmm0
    }
    return r;
}
```

```
__m128 addWithIntrinsics(
    __m128 a,
    __m128 b)
{
    __m128 r =
      _mm_add_ps(a, b);
    return r;
}
```

Figure 2: Vector Addition with SIMD

I are done Implementing the whole math engine and I am starting the implemention SIMD Intrinsics.

5.3 Rendering Engine

For an image to show up on the screen, it has to be rendered onto your display monitor. The graphics engine is responsible for the display on the screen by taking in information about the entire scene such as color, texture, geometry, the shadow of an individual object and lighting, the viewpoint of a scene.

5.3.1 OpenGL

I am using OpenGL to render graphics onto the screen. I chose to do so because it is open-source, cross-platform, and easy to use. I have built a lighting system using the Phong Lighting system. I have also developed the code for shadows.

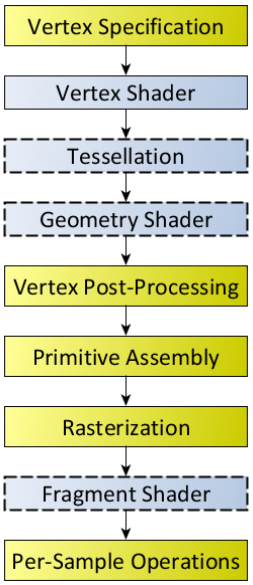

Figure 3: OpenGL Rendering Pipeline

5.3.2 Lighting (Phong Lighting Model)

Components of Phong lighting model using RGB model: OpenGL allows us to break this light's emitted intensity into three parts: ambient, diffuse, and specular. Each type of light component consists of three major color components.

Ambient light is the easiest to implement. Let I_a denote the intensity of ambient light.
For each surface, let

$$0 \leq k_a \leq 1$$

denote the surface's coefficient of ambient reflection, that is, the fraction of the ambient light that is reflected from the surface.

The ambient component of illumination is

$$I_a = k_a L_a$$

Note that this is a vector equation (whose components are RGB).

The key parameter of the surface that controls diffuse reflection is k , the surface's coefficient of diffuse reflection. Let I denote the diffuse reflection component of the light source.
Assume $\bar{l}$ and $\bar{a}$ are normalized, then $\cos\theta = (\bar{a} \cdot \bar{l})$. If $(\bar{a} \cdot \bar{l}) < 0$, then the point is on the dark side of the object.
The diffuse component to illumination is

$$I \quad = k \quad \max(0, \bar{a} \cdot \bar{l})L$$

The parameters of surface that control specular reflection under the Phong model, are k , the surface's coefficient of specular reflection, and s, shininess. The formula for the specular component is

$$I \quad = k \quad (\bar{a} \cdot \bar{l}) \quad L$$

The total equation for lighting calculation will be

$$I = I_e + I_a + \frac{1}{a + bd + cd^2}(I_d + I_s)$$

Figure 4: Phong Lighting Model Equation

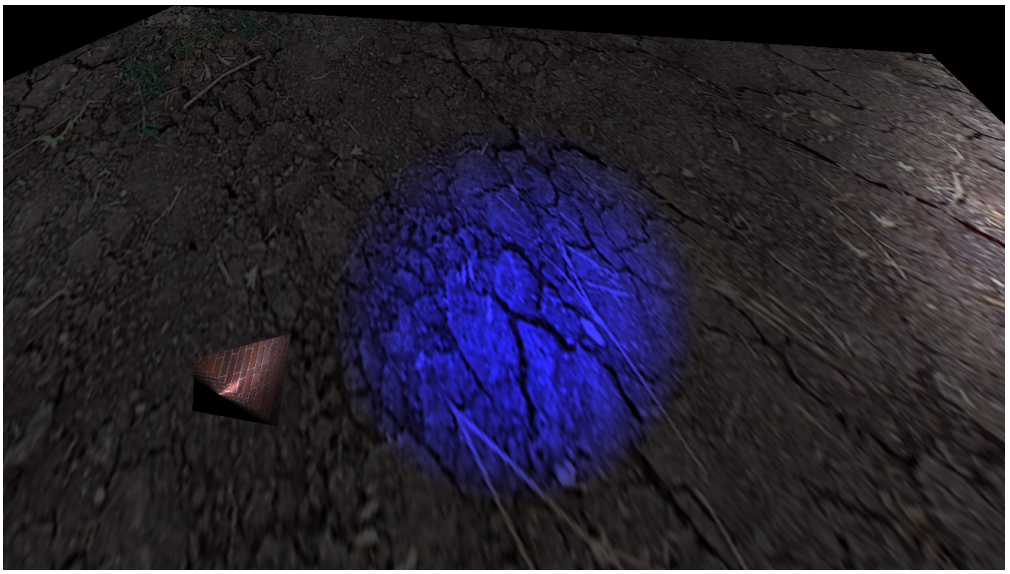

Figure 5: A Scene in our Game Engine using the Phong Lighting Model

5.4 Physics Engine

In the real world, objects are governed by the laws of physics. For example, objects collide and are set in motion according to Newton's laws of motion. In the game world, for objects to act similar to the real world, it has to have the same system built through programming.

I am still yet to implement the physics engine.

5.5 User Interface

User Interface is essential for a game engine as the user should be able to navigate through the Engine without any hassle. To satisfy this purpose, I plan to use a Graphical user interface framework called ImGui for the GUI implementation. However, I am yet to implement it.

6 Results

I managed to incorporate SIMD into my Math Engine. I am yet to implement it fully. I have observed 3x times speedup using the SIMD architecture. This speedup will surely help improve the performance. The rendering engine is being implemented now. I am done with the lighting system using the Phong Lighting model, and I also developed code for calculating shadows. We can add 20 lights to the scene at once. I have made the logging

library using spdlog and event system to handle key events. With this said, there is still a long way to go in building a Game Engine, and I hope to continue working on it.

7 Conclusion

In this paper, I started building a Game Engine from scratch. I introduced methods to construct a parallel game engine from using SIMD to parallelize the Math Engine to Parallelization Techniques like data parallelism and functional parallelism to parallelize the rendering engine. With the help of all the references, books, and open-source tools, I was able to implement the Math Engine, Logging library and am on the way to implementing the Rendering Engine.

There is still a lot to do. First, I need to implement the Rendering engine fully and parallelize it. Then, implement the rest of the low-level systems, implement physics for my Game Engine, and finally make games using the Engine. This would mark the completion of the project. As soon as possible, I look to implementing this and finishing my game engine.

8 Open Source Tools Used

- OpenGL
- spdlog
- ImGui

9 References

[1] I. Corp., “Streaming SIMD extensions - matrix multiplication.” AP-930, June 1999.

[2] I. Corp., “Intel advanced vector extensions programming reference.” June 2011.

[3] An introduction to parallel rendering Thomas W. Crockett

[4] K. Akeley, RealityEngine graphics, in Comp. Graphics Proc. Ann. Conf. Series, ACM SIGGRAPH, 1993, pp. 109-116.

[5] J. Clark, The geometry engine: A VLSI geometry system for graphics, Comput. Graph. 16 (3) (1982) 127-133.

[6] C. Mueller, The sort-first rendering architecture for high-performance graphics, in Proc. 1995 Symp. Interactive 3D Graphics, ACM SIGGRAPH, 1995, pp. 75-84

[7] The Art of Multiprocessor Programming by Maurice Herlihy and Nir Shavit.

[8] Game Engine Architecture by Jason Gregory.

[9] 3D Math Primer for Graphics and Game Development by F.Dunn and I.Parberry.

[10] Rotation Transforms for Computer Graphics by John Vince.

[11] Designing a modern rendering engine by Matthias Bauchinger.

[12] Foundations of 3D computer graphics by Gortler, Steven Jacob.

[13] Unreal Engine Math - https://docs.unrealengine.com/4.27/en-US/API/Runtime/Core/Math/.

[14] Unreal Engine Documentation - https://docs.unrealengine.com/4.27/en-US/

[15] A study on SIMD architecture.