Year 2024/25

Polyray Game Engine

A Highly Modular and Performant Game Engine

Abstract

The Polyray Game Engine is written in Java and designed to provide a flexible and high performance framework for creating 2D and 3D games. Due to the modular design of Polyray, more than just basic 2D and 3D is possible, the only limit is imagination. This document outlines its core design, focusing on efficient rendering techniques, modular architecture and design, and support for custom rendering, objects, data, shaders and more. Special emphasis is placed on performance optimizations, such as efficient data structures, shader programming, and effective resource management. Polyray aims to balance ease of use with technical depth, making it suitable for both rapid prototyping and large-scale projects.

The Polyray Game Engine introduces custom new inventions, such as an algorithm for high performance Real-Time spatial audio acoustics and a custom 2D lighting model.

Keywords: Game Engine, Rendering Pipeline, Optimization, Real-Time Graphics, Shader Programming

1. INTRODUCTION

1.1 BACKGROUND

1.1.1 Game Engines

According to the article “Game engine” posted on Wikipedia, last updated February 2018, game engines are the core software that handles most of the work with graphics, physics and audio. They usually come with an easy interfacing framework to get key and button inputs for control. While popular commercial game engines such as Unreal Engine or Unity come with an editor, the game engine itself is what powers the game, whether an editor is involved or not.

While commercial game engines simplify development using intuitive editors and prebuilt systems, this abstraction can lead developers to overlook performance considerations.

By first learning the fundamentals of graphics APIs like OpenGL or Vulkan, developers gain transferable skills that apply to any engine. This foundational knowledge equips developers to make informed decisions about rendering performance, rather than relying solely on engine-specific shortcuts.

More and more game studios are starting to switch from commercial engines to custom made ones. The reason is that because commercial engines need to encapsulate almost all possibilities there are with types of games. A bunch of features are implemented which would never be used in some specific cases and only adds computational cost. Making a custom made engine ensures that the engine is specifically made for the games the studio makes which removes all bloat and unnecessary operations.

And by far the most powerful side effect of making a custom built game engine is that one knows every little detail on how it is implemented and how to use it, and that eliminates the process of learning the engines terminology and use. So one is automatically an expert at it and knows every ins and outs.

1.1.2 History of Game Graphics

In the early days of computer graphics, rendering was limited by the capabilities of processors like the 6502 CPU, which powered early systems like the Atari 2600. Graphics at the time were very limited, simple 2D sprites and tile-based graphics, with developers using low-level assembly code to manually draw each pixel to the screen. This was a long and painstaking process due to the hardware’s limitations, and rendering involved basic raster graphics, where images were drawn into memory line-by-line.

Figure 1. Rasterized Graphics from Microsoft Flight Simulator 1.0

As hardware advanced, particularly with the introduction of more powerful CPUs like the Motorola 68000 and early x86 processors, developers started experimenting with early 3D graphics. These efforts were primarily seen in vector-based games, where simple wireframes represented 3D objects and lines, or vectors connect points in the model. Games that use vector graphics often lack any kind of shading or depth perception beyond basic geometry.

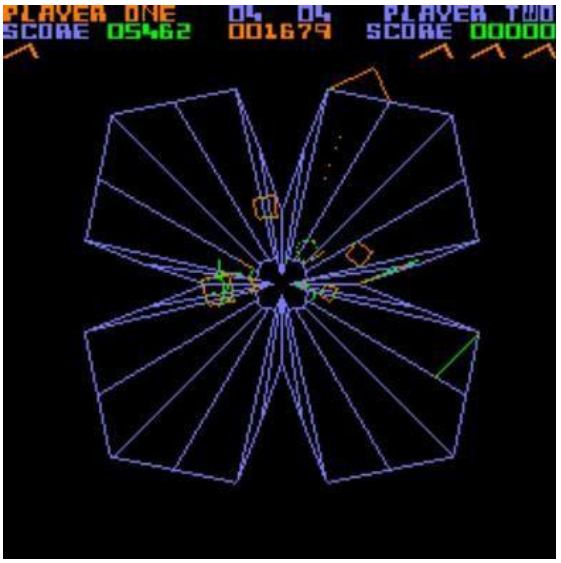

Figure 2. Vector Graphics from the Game Tempest

The true shift to modern 3D rendering began in the late 1980s and early 1990s with the advent of rasterization. This process allowed 3D models to be broken down into individual triangles and projected onto a 2D screen, where they could be textured and lit in Real-Time. The introduction of graphics accelerators, such as the 3dfx Voodoo, revolutionized this process by offloading the computationally expensive task of rendering from the CPU to a dedicated GPU, allowing for complex scenes to be rendered at interactive frame rates.


Figure 3. 3D Graphics with Texture Mapping In the Game Quake 3

1.1.3 Rendering

Rasterization, the process of converting models into a 2D image, has been the cornerstone of Real-Time rendering since its early development. In the late 1980s and early 1990s, as 3D graphics emerged, the need for fast, efficient rendering techniques became crucial, especially for video games. Rasterization works by projecting 3D objects onto a 2D plane (the screen) and determining what color each pixel should be.

The early days of 3D rendering involved simple, flat-shaded polygons and wireframes. As hardware advanced, developers began experimenting with more complex techniques, such as texture mapping and Gouraud shading, to create more detailed and realistic scenes. However, even as these innovations took place, hardware limitations still posed a major challenge to achieving Real-Time performance, leading to the development of various optimizations.

In the 1990s as the demand for more complex and better graphics increased, various optimization techniques were developed. A prime example of this comes from Quake, a game that revolutionized Real-Time rendering through its innovative approaches.
One such breakthrough was the Fast Inverse Square Root algorithm, which significantly improved the performance of lighting calculations and physics by speeding up operations of normalizing vectors. Another crucial optimization was portal-based culling, which reduced the rendering workload by only drawing what was visible to the player. This technique used portals and bounding volumes to determine visibility, a strategy that is still commonly used in modern game engines.

1.2 PURPOSE

The purpose of this project and document is to show that game engines are surprisingly easy to make. This game engine is designed for maximum modularity and flexibility which most commercial engines suffer from not being. Furthermore, performance will be heavily prioritized which might make code more unreadable.

Unity and Unreal are very statically made. The rendering, physics and audio is fully abstracted away from the user and replaced with high-level terminology for them. Anyone using Unity or Unreal may think that they have lots of control, but there are many cool tricks and optimizations that wouldn’t be possible with them which would be obvious and intuitive with a custom made engine.

But even making a custom engine doesn’t ensure full control without constantly changing the engine code to add more features. So this project was an experiment to see how far the modularity of a game engine could be pushed without introducing too much complexity and bottlenecks.

1.3 RESEARCH QUESTIONS

How hard is it really to make a custom game engine using OpenGL or any other Low-Level API?

What is possible to do in a custom made engine that would otherwise be impossible in a commercial engine?

1.4 LIMITATIONS

Due to limited time working before the release of this document, more complex algorithms and optimizations won't be completed, such as occlusion culling, shadow mapping in 3D and other graphics related features and optimizations.

As such, while the engine is capable of rendering detailed scenes and performing essential calculations and handling for game mechanics, certain performance bottlenecks may be present in the rendering pipeline. Developers should be aware that additional steps, like implementing a custom renderer or implementing custom audio effects using Polyray Modular may be necessary for complex environments.

2. METHOD

The development of the Polyray Game Engine was guided by three core principles: Performance, Modularity, and Flexibility while ensuring minimal bloat and unnecessary code. To achieve this, the engine was designed with an efficient architecture that prioritizes control over every aspect.

2.1 Performance Focus

The engine was built with performance as a top priority. Written entirely in Java apart from performance critical code that is written in C++. It uses LWJGL and OpenGL and is carefully made from scratch to minimize unnecessary overhead and bloat, and maximize performance. By avoiding third-party dependencies (except LWJGL), the engine ensures full control over its internal processes, optimizing both performance and resource usage.

Key optimizations include efficient memory management to reduce allocation overhead and low-level control over rendering processes using OpenGL for direct GPU communication.

2.2 Modularity and Flexibility

The engine was developed to support a wide range of game types and mechanics without sacrificing performance. By adopting a modular architecture, individual systems such as rendering, physics, and audio can be modified or replaced without major structural changes. This design allows developers to extend functionality while maintaining consistent performance.

To promote flexibility, the engine features a customizable rendering pipeline that can adapt to different visual styles or performance needs, along with well-defined interfaces that allow developers to extend functionality as required.

2.3 Real-Time Audio Simulation

To enhance audio realism, the engine includes a high-performance Real-Time audio simulation algorithm called DBR (Dynamic Buffer Redistribution). This algorithm simulates audio delay and volume with extreme precision to generate ultra realistic acoustic effects with minimal latency. For advanced spatial audio, an extension called DCDBR (Dual-Channel Dynamic Buffer Redistribution) was developed. This extension enables precise 3D sound positioning, where audio sources can reflect off walls, creating an immersive and lifelike audio experience that adapts to environmental geometry in Real-Time.

2.4 Ensuring Quality Control

By implementing all systems from scratch, the engine maintains complete control over its internal logic. This approach ensures that performance optimizations, bug fixes, and feature implementations can be handled directly, reducing the unnecessary complications with using external libraries or frameworks that may introduce performance bottlenecks, compatibility issues or different naming conventions which only slows development speeds.

With this combination of custom development, efficient algorithms, and a modular structure, the Polyray Game Engine achieves a good balance of performance, flexibility, and minimal bloat.

3. POLYRAY GAME ENGINE

3.1 RENDERING

3.1.1 Window

In Polyray, the window is created and managed by the GLFW library. GLFW is chosen for ease of use, robustness and support for cross-platform compatibility, making it a good choice for Polyray's design aims. The window class is called `GLFWindow` and has a user-friendly interface for window creation and event handling such as keyboard inputs and the ability to hide the cursor.

Initializing a window in Polyray doesn't require much code, which allows for quick setups. The example shown below is an example of how to create a window that is non-exclusive fullscreen and without a title bar.

```java
public GLFWindow w;

public Example() {
    // Create the window object and set the title of the window to "Title"
    this.w = new GLFWindow("Title");

    // Initialize the window with non-exclusive fullscreen and no title bar
    w.createFrame(500, 500, false, true, false);

    // While the window is open, run the game loop
    while(w.isWindowOpen()) {
        // Update the window to process user inputs and update the frame
        w.update();
   }
}
```

Polyray lets developers easily implement handling for key- and mouse events. The example shown below shows how to handle mouse movement and key presses.

```java
this.w = new GLFWindow("Title") {
    // Override keyPress to implement logic for handling key presses
    @Override
    public void keyPress(int key) {
        if(key == GLFW_KEY_SPACE) {
            System.out.println("Jump!");
        }
    }
    // Override mouseMove to implement logic for handling mouse movement
    @Override
    public void mouseMove(float x, float y) {
        System.out.println("Mouse x: " + x + ", y: " + y);
    }
};
```

Polyray also supports other events, letting the user override methods like `keyRelease`, `mousePress`, `mouseRelease`, `mouseDrag`, `scroll`, `effectChanged` (See Section 3.1.10 for more information on post-processing effects) and `windowResized`.

There are other methods that outputs the current state of the window like `isWindowOpen` and `isWindowFocused` which returns whether or not the window is the current one the user is interacting with.

3.1.2 Input

Just like Unity, Polyray has an `Input` class which has one method for getting the current state of a key (if it is pressed or not). It works side by side with the `GLFWindow` class, which updates the key states during runtime. The example below is how usage of the class would look like.

```java
// Get the current state of the 'w' key
if(Input.getKey(GLFW_KEY_W)) {
    // Move forward
}
```

Furthermore, Polyray has built-in support for controllers and can handle up to 16 controllers per device simultaneously. The setup of controller inputs is a bit more tricky though. It supports both Xbox and Playstation controllers with explicit button definitions for both. An example is "`BUTTON_XBOX_A`" and "`BUTTON_PLAYSTATION_X`". The class for managing controllers is called `ControllerInput`. Which, unlike the Input class, doesn't need a window to update the controllers, it is instead done manually as shown in the example below.

```java
public ArrayList<Integer> controllers;

public Example() {
    // Set up controllers
    ControllerInput.setup();
    this.controllers = ControllerInput.getActiveControllers();
}

// Call this method from in the game loop
public void update() {
    ControllerInput.updateControllers();
    if (!controllers.isEmpty()) {
        // Get the first controller
        int c = controllers.get(0);

        // Get forward and backward movement
        float ws =
ControllerInput.getJoystickPosition(c,ControllerInput.JOYSTICK_LEFT_Y_AXIS);

        // Get left and right movement
        float ad =
ControllerInput.getJoystickPosition(c,ControllerInput.JOYSTICK_LEFT_X_AXIS);

        if(ControllerInput.getButton(c, ControllerInput.BUTTON_XBOX_A)) {
            System.out.println("jump!");
        }
    }
}
```

3.1.3 Shader Programs and Preprocessing

In Polyray, Shader Programs are used to manage and execute the graphical operations written in GLSL. The engine provides an abstraction to simplify shader compilation and linking, making it easier to handle various shader stages (vertex, fragment and compute) in Real-Time rendering.

The `ShaderPreprocessor` class in Polyray provides useful utility functions designed to simplify the management and creation of shader code and preprocessor directives. It allows developers to compose shader code from multiple sources and dynamically set values in the shader from Java, creating a flexible and powerful environment for shader management.

One of the most useful features of the `ShaderPreprocessor` is the ability to append code from external files into a base shader file. This helps modularize shader code, promoting reusability and maintainability. For example, a function or a set of common utilities can be placed in a separate file and then appended into the main shader file. The example below illustrates this feature using a utility file “Addition.glsl” being appended to the “Shader.glsl” file.

```C/C++
// In "Addition.glsl"
float add(float a, float b) {
    return a + b;
}

// In "Shader.glsl"
#append "Addition.glsl";

void main() {
    // Uses the add function from the "Addition.glsl" file
    float c = add(10.0, 20.0);
}
```

The `ShaderPreprocessor` class also allows developers to dynamically set values in the shader code, such as constants or buffer bindings. These values can be set from the Java side, allowing for greater flexibility when adjusting shader behavior in Real-Time without modifying the shader code itself or creating several similar shaders for only small differences.

```Java
ShaderPreprocessor proc = ShaderPreprocessor.fromFiles("Shader.glsl");
proc.appendAll(); // Apply the appends
proc.setFloat("A_VALUE", 10.0f); // Set a value for A_VALUE
proc.setFloat("B_VALUE", 20.0f); // Set a value for B_VALUE
ShaderProgram shader = proc.createShader("Name", 0); // Create the shader
```

Polyray has built-in files that can be appended such as “GammaCorrect.glsl”, “PBRLighting.glsl”, “Camera3D.glsl”, “Noise.glsl” and more.

3.1.4 Shader Buffers

In Polyray, The `ShaderBuffer` class abstracts the OpenGL buffer object system, simplifying the process of creating and managing buffer data. The example shown is a shader buffer in GLSL.

```C/C++
layout(std430, binding = VALUE_IDX) buffer ValueBuffer {
    float values[];
};
```

Polyray has another utility class called `BindingRegistry`, which simplifies the management of buffer binding points and helps prevent binding overriding. The example below shows how to properly set the binding point and set the buffers data.

```java
ShaderBuffer valueBuffer = new ShaderBuffer(GL_SHADER_STOGRAGE_BUFFER,
GL_STATIC_DRAW);
// Send data to the GPU
valuebuffer.uploadData(new float[] {0.0f, 1.0f, 2.0f, 3.0f});
// Bind the shader buffer to a binding point
int binding = BindingRegistry.bindBufferBase(valueBuffer);
// Dynamically set the VALUE_IDX to the binding point of the buffer
ShaderPreprocessor proc = ShaderPreprocessor.fromFiles("Shader.glsl");
proc.setInt("VALUE_IDX", binding); // Set the binding point to the shader
ShaderProgram shader = proc.createShader("Name", 0);
```

3.1.5 Vertex Buffer and Instancing

A main aspect of polyray, which sets it apart from other game engines, is the modular design of vertex buffers. Instead of having fixed vertex buffer layouts, Polyray allows the user to easily make vertex buffer layouts using the `VertexBufferTemplate` class. When a template is made, it can be used multiple times to create vertex buffer objects with the same layout of data. Because of how Polyray is designed, such templates are made easily. An example that can be found in Polyray's built-in vertex and instance classes, of how a `VertexBufferTemplate` is set up is shown below.

```java
// In Vertex3D
public static final VertexBufferTemplate VBO_TEMPLATE = new
VertexBufferTemplate(false)
        .addAttribute(VertexAttribute.VEC3)
        .addAttribute(VertexAttribute.VEC3)
        .addAttribute(VertexAttribute.VEC2);
```

A vertex buffer doesn't just store vertex data, it can also hold instance data. The `VertexBufferTemplate` class handles this distinction in an elegant way. With a simple `true` or `false` argument, it can easily switch from holding vertex data to instance data. This makes it easy to define whether or not a template is intended for vertices or for instances. Below is an example of an instance template, notice the `true` argument.

```java
// In Instance3D
public static final VertexBufferTemplate VBO_TEMPLATE = new
VertexBufferTemplate(true)
    .addAttribute(VertexAttribute.VEC4).addAttribute(VertexAttribute.VEC4)
    .addAttribute(VertexAttribute.VEC4).addAttribute(VertexAttribute.VEC4);
```

3.1.6 Textures and Texture Arrays

Polyray supports both Textures and Texture Arrays, making texture management simple. The `TextureUtils` class helps create basic textures easily. Polyray has two texture classes: `Texture`, which manages texture data and integrates with Java's `Graphics2D` for drawing, and `GLTexture`, which handles GPU utilization. `GLTexture` can upload texture data directly from a `Texture` object. The example below shows two ways to create a `GLTexture`.

```java
// Method 1 creating a empty texture of size 128x128
GLTexture tex = new GLTexture(128, 128, GL_RGBA, GL_RGBA8, false, false);
// Method 2 creating a GLTexture from a existing Texture object
Texture texture = new Texture(128, 128); // Example texture
GLTexture tex = new GLTexture(texture, GL_RGBA8, false, flase);
```

There does exist another texture class called `GLTextureMSAA`, however, it is only used with the respective `GLFramebufferMSAA`, it is not meant for usage outside of that.

3.1.7 Framebuffers

Framebuffers in Polyray are easy to use. A framebuffer in Polyray is called `GLFramebuffer` and has similar properties to `GLTexture`. A `GLFramebuffer` has an internal texture for where the framebuffer's color data is stored. That texture could then be used in all kinds of applications such as post-processing, windows inside games etc. The example below shows how a framebuffer is set up and how to access the internal render texture.

```java
GLFramebuffer framebuffer = new GLFramebuffer(128, 128);
GLTexture render = framebuffer.render;
```

For better quality, there exists a `GLFramebufferMSAA` for Anti-Aliasing with a customizable sample count. The example below is how a `GLFramebufferMSAA` is created.

```java
// 16x MSAA
GLFramebufferMSAA framebufferMSAA = new GLFramebufferMSAA(128, 128, 16);
```

3.1.8 Lighting System

Polyray excels in its lightning model. Previously, it used a simple Blinn Phong model, which resulted in a plastic-like look and wasn't that physically accurate. After some experiments, a version of PBR lighting was chosen for its performance and physical accuracy.

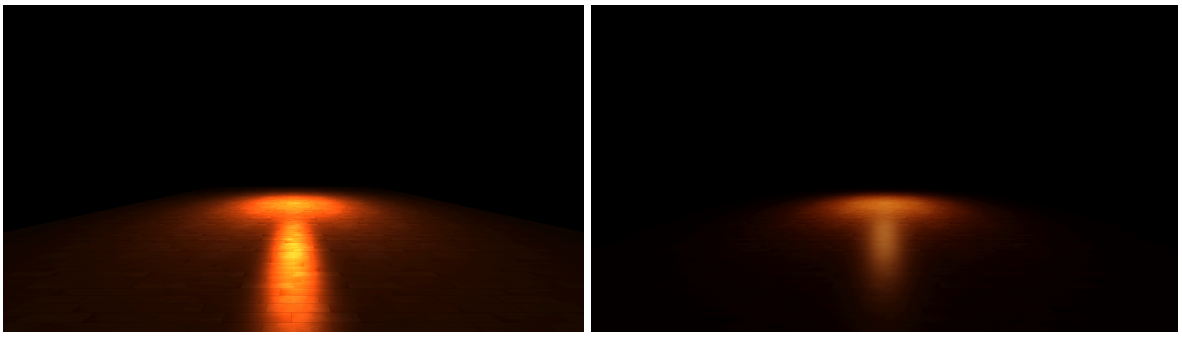

Figure 4. Comparison of Blinn-Phong (left) and PBR (right) lighting models. *Image by Alex Lindgren.*

Another feature with Polyray's implementation of PBR is the inclusion of ACES Tonemapping and Gamma correction. By having those, it prevents excessive bright lights from saturating and instead compresses the color range into HDR so that bloom and other features can be implemented.

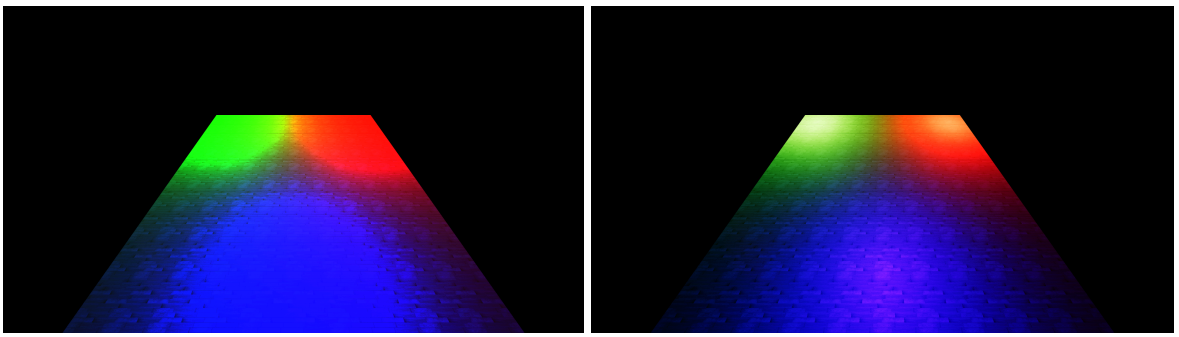

Figure 5. Comparison of without ACES Tonemapping (left) vs. with ACES Tonemapping and gamma correction (right). *Image by Alex Lindgren.*

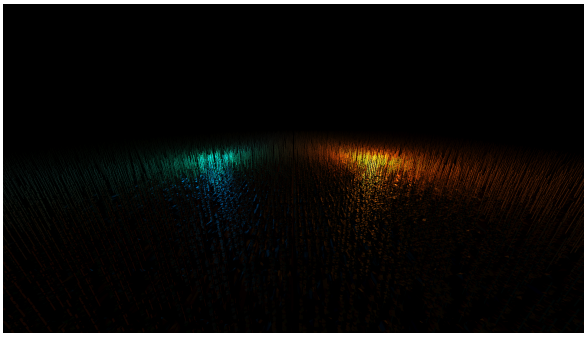

Figure 6. Showcasing 3D Point lights in a Wheat Field where one light's color is the complement of the other. *Image by Alex Lindgren.*

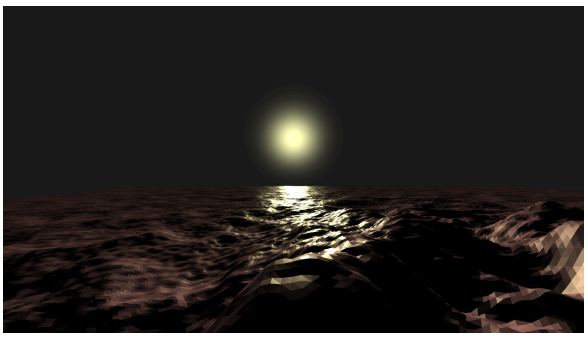

Figure 7. Demonstrating the built-in PBR lighting in a testing scene of the game The Uncles of The Forest. *Courtesy of Polyray Games.*

For 2D, Polyray has a custom implementation for lighting. Although not yet implemented in the main engine, it offers a highly flexible system for different types of lights. The light types implemented include point lights, directional lights, area lights. The lighting also has volumetric lighting and shadow casting.

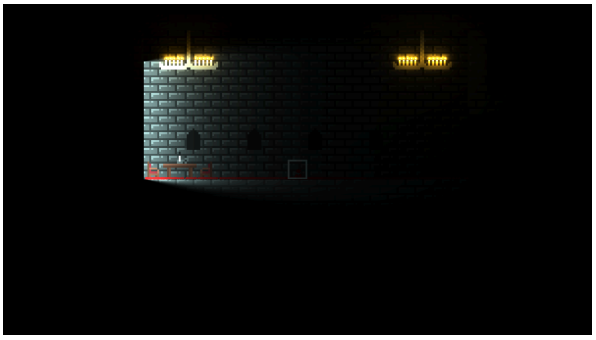

Figure 8. Showcasing Area light and Point lights as a door and candles in Platformer 2D. *Courtesy of Polyray Games.*

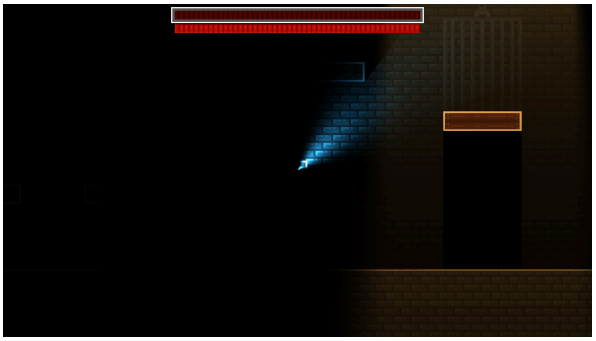

Figure 9. Showcasing shadowcasting using HDDA and a directional light as a flashlight in Platformer 2D. *Courtesy of Polyray Games.*

3.1.9 Render Objects

The `RenderObject` class in Polyray is part of the builtin package which is a simple implementation of the modular `RenderObjectBase` class. The class serves as a foundation for rendering objects in Polyray and brings together `GLTexture`, `VertexBufferTemplate` and `ShaderProgram` classes. The `RenderObject` class takes in two `VertexBufferTemplates`, a `ShaderProgram` and optionally a `GLTexture` or a `Texture`. The example below shows a full implementation of creating a `RenderObject` without a texture.

```java
// u is an instance of a Renderer3D
// get the buffer binding of the camera
int camIdx = u.getCameraTransformBinding();

// get the buffer binding of the environment
Background b = u.getBackground();
int envIdx = BindingRegistry.bindBufferBase(b.environmentBuffer);

// Create a material with the default PBR Lighting model
Material mat = new material(camIdx, envIdx);
mat.setRoughness(0.5f);
mat.setMetallic(0.5f);
mat.setF0(new Vector3f(0.05f, 0.05f, 0.05f));

// The same material can be used in multiple objects with different textures
for example
RenderObject object = new RenderObject(mat.getShader(),
Vertex3D.VBO_TEMPLATE, Instance3D.VBO_INSTANCE);
```

3.1.10 Post-Processing

Post-Processing in Polyray works by rendering the scene on a framebuffer, then the texture is used as the input to the `PostProcessing` class, and the output is then finally rendered as a full screen quad. The current implementation doesn't support any custom post processing effects and only has 19 built-in effects, which will be fixed in later versions though.

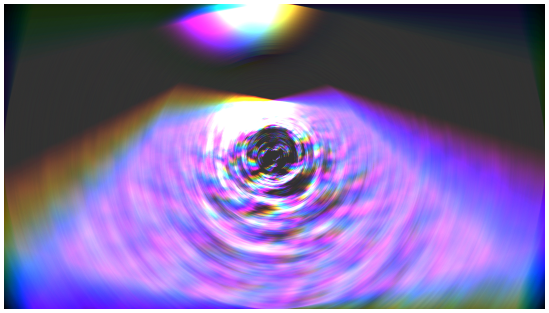

Figure 10. Showcasing one built-in post processing shader effect with maximum strength in The Uncles of The Forest. *Courtesy of Polyray Games.*

3.1.11 Full Rendering Pipeline

Polyray uses a highly efficient rendering pipeline that prioritizes minimal overhead. The pipeline is designed to run in Real-Time, optimizing GPU communication. What follows is a detailed breakdown of how the pipeline functions at runtime for the built-in renderers and render objects.

The beginning of each render begins with uploading global data, such as camera transforms, projection matrices etc. This step is shown below, how global data is uploaded in the `Renderer3D` class.

```java
createPerspectiveMatrix(FOV, (float) width / height, minRendDist,
renderDist, projection);
FloatBuffer cameraData = FloatBuffer.allocate(32);
cameraTransform.toFloatBuffer(cameraData);
cameraData.put(projection.array());
cameraTransformBuffer.uploadData(cameraData.array());
uiTransformBuffer.uploadData(uiTransform.toFloatBuffer(FloatBuffer.allocate(
12), true).array());
```

When rendering the objects the renderer iterates through all objects to see which should be rendered. This is controlled by the `doRender` flag set on each object. If the object's flag is `true` and the object has content to render (not marked as clear or removed), the object is submitted for rendering.

```java
Iterator<RenderObjectBase> iter = objects.iterator();
while (iter.hasNext()) {
    RenderObjectBase object = iter.next();
    if (object.isRemoved()) {
        iter.remove();
        continue;
    }
    if (object.isClear() || !object.doRender) {
        continue;
    }
    ShaderProgram shader = object.getShader();
    shader.use();
    object.render();
    shader.unuse();
}
```

For each valid object that should be rendered, the renderer calls the `object.render()` method. In the built-in implementation, it allows for both textured and untextured objects. A draw call is made to render the object, which in the case of the `RenderObject` class, is `glDrawArraysInstanced`.

```java
@Override
public void render() {
    if (texture != null) {
        glActiveTexture(GL_TEXTURE0);
        texture.bind();
    }
    glBindVertexArray(vao);
    glDrawArraysInstanced(mode, 0, numVertices, numInstances);
}
```

The API calls that get sent to the GPU during runtime would include a header with uploading global data such as UBOs or SSBOs. After that there would be a series of repeating blocks containing each object being rendered, ending with a buffer swap call and an event polling call. What follows is what the complete render would look like.

```c
// A header concisting of uploading UBOs etc. As an example, a single UBO:
glBindBuffer
glBufferData

// Objects made of a optional texture, a shader usage and a vao binding
// As an example, a object without a texture
glUseProgram
glBindVertexArray
glDrawArraysinstanced
glUseProgram

// More objects being rendered here

// End of rendering, swapping buffers and polling events
glfwSwapBuffers
glfwPollEvents
```

3.2 AUDIO

3.2.1 Sound Processing Handler

Polyray uses a modular design for applying audio effects, built around the `SoundEffector` class. This design allows developers to add and combine effects for complex, rich and dynamic audio experiences for true immersiveness.
The core of Polyrays audio system is the `SoundEffector` class, it handles applying effects and encoding and decoding PCM wav data. Using the `addEffect` method present in the `SoundEffector` class, developers are able to add effects to apply.

The example below shows how effects are added and how effects are applied.

```java
SoundEffector effector = new SoundEffector(2, 4);

// Decrease the volume by 50%
effector.addEffect(new VolumeEffect(0.5f), 0);


byte[] buffer = new byte[4096];

// "in" is a AudioInputStream
// "out" is a SourceDataLine
while(in.read(buffer) != -1) {
    out.write(effector.nextBuffer(buffer, 0.0f), 0, buffer.length);
}
```

3.2.2 Sound Effects

Polyray offers a handful of built-in common audio effects which speeds up development. For custom effects, Polyray offers a user-friendly interface called `Effect`. The example below shows how the `Effect` interface looks like and following that is how the built-in `VolumeEffect` implements it.

Java

```java
public interface Effect {
    public void computeRight(int[] channel);
    public void computeLeft(int[] channel);
    public void onStart();
    public void onFinnish();
}
```

Java

```java
public class VolumeEffect implements Effect {
    private float volume, prevVolume;
    public VolumeEffect(float volume) {
        this.volume = volume;
        this.prevVolume = volume;
    }
    public void setVolume(float volume) {
        this.volume = volume;
    }
    @Override
    public void computeRight(int[] channel) {
        compute(channel);
    }
    @Override
    public void computeLeft(int[] channel) {
        compute(channel);
    }
    @Override
    public void onStart() {}
    @Override
    public void onFinnish() {
        this.prevVolume = this.volume;
    }
    private void compute(int[] channel) {
        float delta = volume - prevVolume;
        for(int i = 0; i < channel.length; i++) {
            channel[i] *= prevVolume + (float) i / channel.length * delta;
        }
    }
}
```

The earlier versions of the sound engine didn't have an `onStart` and `onFinish` and they were added solely due to support interpolating values across buffers of audio. Otherwise, if values would change rapidly, loud artifacts or clicks would be present in the audio.

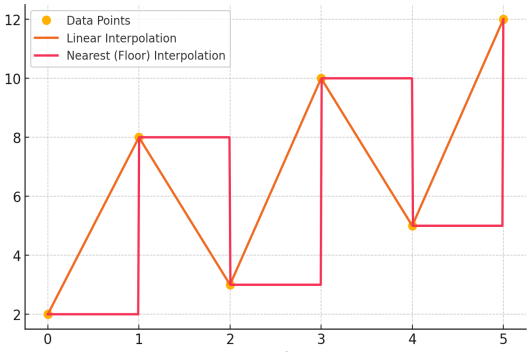

Figure 11. A Visualization of Linear Interpolation vs. Nearest. *Image by Alex Lindgren.*

3.2.3 DBR and DCDBR for Real-Time acoustics

With Polyray comes the invention of a high performance Real-Time acoustics audio effect. The effect is separated into two stages, one stage of collecting samples and the other of processing.

The first step is not included and is instead expected to be implemented by the developers, although files with pre collected samples exist which can be easily loaded in the project using the `loadDBRData` method. Future updates will address that and supply a sample collector.
The audio processing step is included in the engine as the `DBREffect` and `DCDBREffect` classes. The step consists of taking an input audio buffer, distributing them in a larger buffer, summing up the audio, taking a section from the beginning of the larger buffer (which is the output audio) and finally shifting the larger buffer down to fill the section that was taken.

An optimized version of this algorithm in C++ is in the example below using AVX and is integrated into Polyray using JNI. The current implementation with a prebuilt dll only works on x86-64 architecture and is specifically fine tuned for specific hardware.

```C/C++
#include <cstdint>
#include <cstring>
#include <immintrin.h>

void DBR(float* buffer, int& bufferSize, float* outputRight, float*
outputLeft) {
    for(int i = 0; i < sampleCount; i++) {
        int delay = delaysI[i];

        float* dstR = collectorRight + delay;
        float* dstL = collectorLeft + delay;

        __m256 mulR = _mm256_set1_ps(volumesR[i]);
        __m256 mulL = _mm256_set1_ps(volumesL[i]);
        #pragma GCC unroll 8
        for (int i = 0; i < bufferSize; i += 8) {
            float* src = buffer + i;
            __m256 inVec = _mm256_loadu_ps(src);
            _mm256_storeu_ps(dstR, _mm256_fmadd_ps(inVec, mulR,
_mm256_loadu_ps(dstR)));
            _mm256_storeu_ps(dstL, _mm256_fmadd_ps(inVec, mulL,
_mm256_loadu_ps(dstL)));
            dstR += 8;
            dstL += 8;
        }
    }

    int len = bufferSize * sizeof(float);

    std::memcpy(outputRight, collectorRight, len);
    std::memcpy(outputLeft, collectorLeft, len);

    int size = (collectorSize - bufferSize) * sizeof(float);
    std::memmove(collectorRight, collectorRight + bufferSize, size);
    std::memmove(collectorLeft, collectorLeft + bufferSize, size);

    std::memset(collectorRight + (collectorSize - bufferSize), 0, len);
    std::memset(collectorLeft + (collectorSize - bufferSize), 0, len);
}
```

3.3 PHYSICS

Polyray mainly uses point mass physics, the point mass class is called `Particle`. The `Particle` class is an extension of the `PhysicsObject` class that implements mass and force calculations.

Polyray's physics has premade, highly optimized constraints for particle physics which simplifies, optimizes and strengthens the normal way of point masses connected by lines to form structures. Those constraints include `ParticleLine`, `ParticleLineN`, `ParticlePlane` and `ParticleSphere`.

Here is an overview of what the constraints are trying to solve for.

- `ParticleLine` keeps two particles the same distance from each other.
- `ParticleLineN` keeps multiple particles on one line with a custom spacing between each particle.
- `ParticlePlane` keeps all particles on a plane.
- `ParticleSphere` keeps particles on the surface of a sphere.

4. DISCUSSION

Polyray, in its current state, is far from finished. It is in active development, bug patches and feature implementation will be addressed in future updates. By incorporating improved rendering techniques, smarter resource management, and enhanced culling algorithms to reduce unnecessary computations, such limitations could be minimized. Until then, users may need to rely on creative solutions to maintain stable frame rates in much larger projects.

Why make a Custom Engine?

Common engines such as Unity, Unreal Engine and Godot all have one issue, that is how generalized they are. They might have thousands of features that suit most games' needs. However, when it comes to memory usage and performance, due to how generalized they are, they typically run slower and use more memory. That's because all those features and generalizations account for things that might be added in certain games. But for those games that don't use that feature, they still have to do the same calculations for something that isn't necessary.

That is one of the benefits of custom engines. They don't have to over-generalize and account for all possible combinations and outcomes. Instead they can just include what is needed for the specific game which removes the bloat that common engines have.

Why Aren't More Games Running on Custom Game Engines?

Due to how complicated it usually is to create a game engine from scratch, it is often much faster and cheaper to go with premade engines. It is very uncommon for games to use anything other than premade engines. Only in certain cases where the premade ones don't cut it or the game is too complex does studios opt for custom made due to the benefits mentioned before at the cost of time and complexity.

5. REFERENCES

Polyray Game Engine Github Repository The Official Github Repository for the Polyray Game Engine

---

(18 sept. 2024) https://en.wikipedia.org/wiki/MOS_Technology_6502 6504 CPU overview and functionality

(18 sept. 2024) https://sv.wikipedia.org/wiki/Atari_2600 How the Atari 2600 works

(18 sept. 2024) https://en.wikipedia.org/wiki/Assembly_language The assembly language

(18 sept. 2024) https://en.wikipedia.org/wiki/Raster_graphics What rasterized graphics are

(18 sept. 2024) https://en.wikipedia.org/wiki/Motorola_68000 The Motorola 68000 microprocessor architecture

(18 sept. 2024) https://en.wikipedia.org/wiki/X86 x86-64 CPUs

(19 sept. 2024) https://en.wikipedia.org/wiki/Wireframe Wireframes in detail

(19 sept. 2024) https://en.wikipedia.org/wiki/Rasterisation What the process of Rasterization is

(19 sept. 2024) https://en.wikipedia.org/wiki/3dfx The 3D graphics processing unit

(20 sept. 2024) https://en.wikipedia.org/wiki/Central_processing_unit What CPUs are

(20 sept. 2024) https://en.wikipedia.org/wiki/Graphics_processing_unit what GPUs are

(21 sept. 2024) https://en.wikipedia.org/wiki/Real-time_computer_graphics What Real-Time rendering is

(22 sept. 2024) https://en.wikipedia.org/wiki/3D_projection What Projection involves, converting 3D to 2D

(22 sept. 2024) https://en.wikipedia.org/wiki/Texture_mapping How texture mapping works

(27 sept. 2024) https://en.wikipedia.org/wiki/Gouraud_shading What Gouraud Shading is, more commonly known as Smooth Shading

(27 sept. 2024) https://en.wikipedia.org/wiki/Quake_(video_game) The game Quake

(27 sept. 2024) https://en.wikipedia.org/wiki/Fast_inverse_square_root The Fast Inverse Square Root Algorithm, also known as 0x5F3759DF

6. ENGINE SHOWCASE

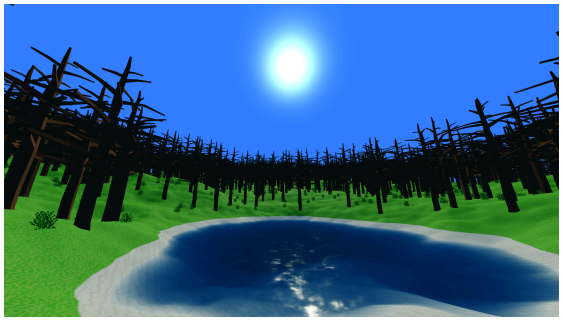

Figure 12. Demonstrating the use of built-in PBR lighting with subsurface water scattering and water reflections in 237. *Courtesy of Polyray Games.*

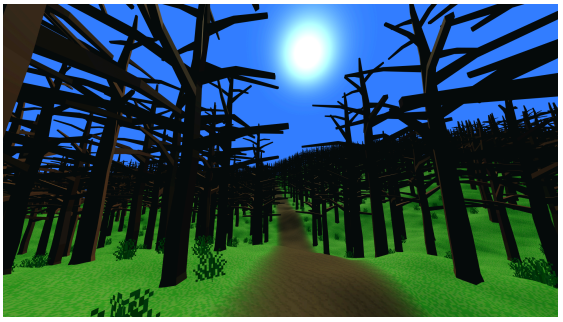

Figure 13. Demonstration of the game engine's capability to render dense forests efficiently in 237. *Courtesy of Polyray Games.*

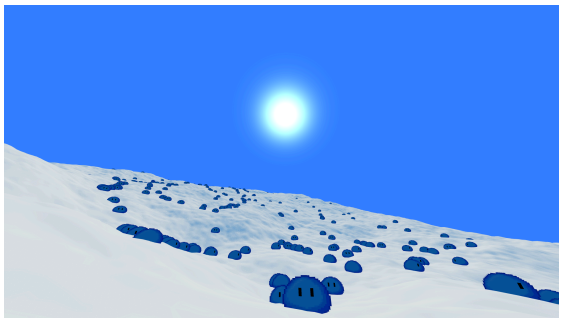

Figure 14. Showcase of realistic subsurface scattering in snow, with deeper creases appearing blue due to sunlight absorption in 237. *Courtesy of Polyray Games.*

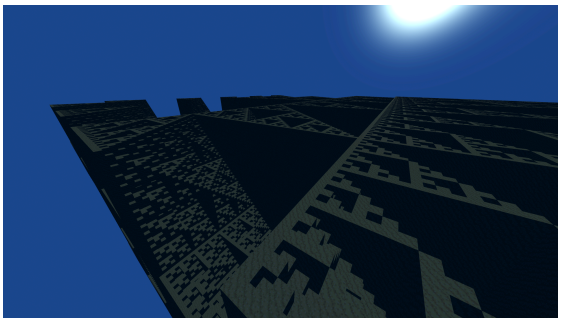

Figure 15. Showcasing a highly efficient greedy meshing in a voxel game with a fractal surface using Polyray. *Image by Alex Lindgren.*

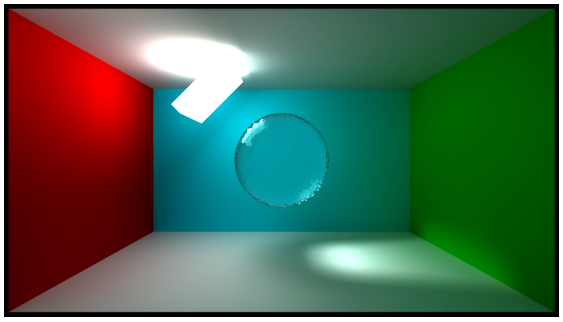

Figure 16. Showcasing a Ray Tracer with transparent glass built using Polyray. *Image by Alex Lindgren.*

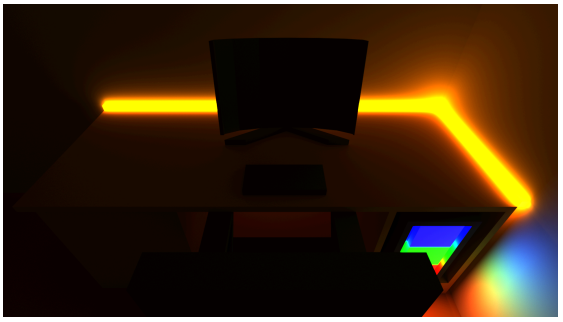

Figure 17. Showcasing a room rendered using a Ray Tracer built using Polyray. *Image by Alex Lindgren.*

.

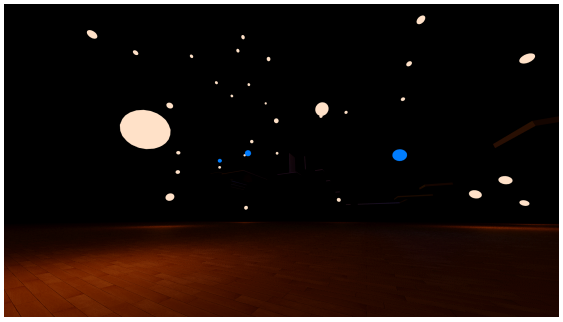

Figure 18. Showcase of many point lights in a platforming game using Polyray. *Image by Alex Lindgren.*

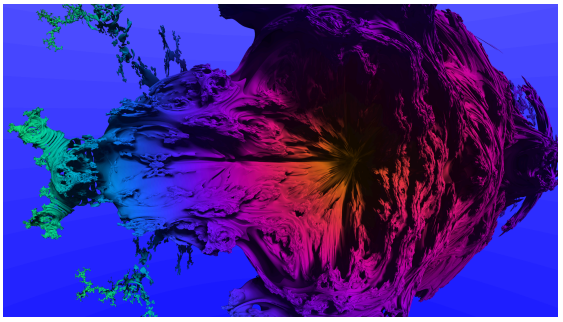

Figure 19. Showcasing a Ray Marcher rendering a mandelbulb using Polyray. *Image by Alex Lindgren.*

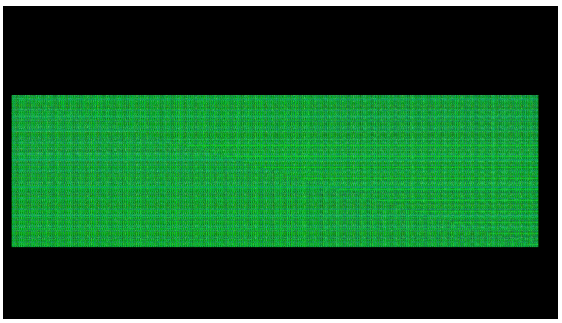

Figure 20. Demonstrating Polyrays performance by rendering 20 Million instances in Real-Time. *Image by Alex Lindgren.*

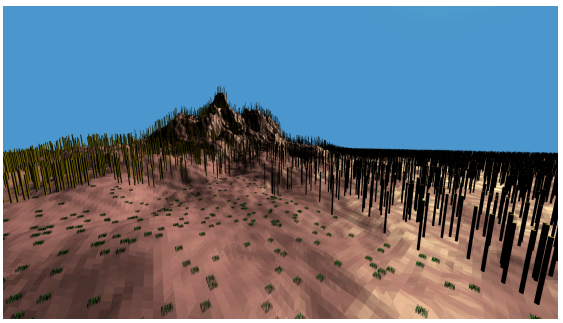

Figure 21. Showcasing an old version of The Uncles of The Forest rendering thousands of instanced trees and grass. *Courtesy of Polyray Games.*

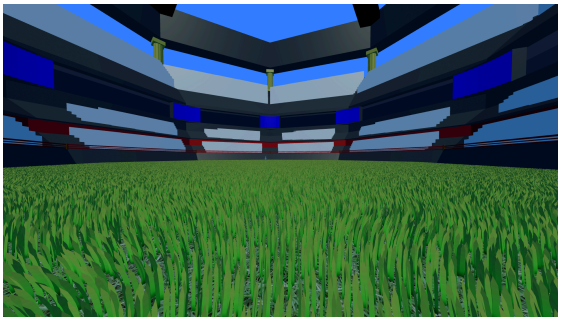

Figure 22. Showcasing 200 000 grass blades being rendered and simulated in Real-Time in Rocket Powered Cripple Arena. *Image by Alex Lindgren.*

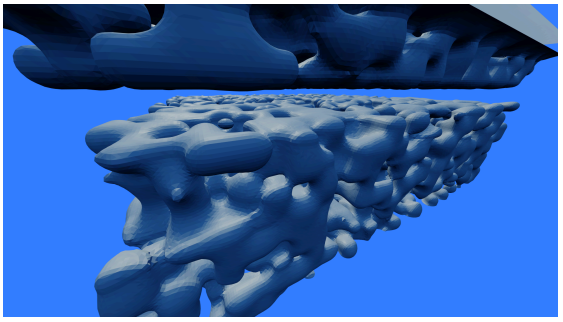

Figure 23. Showcasing Marching Cubes rendering over 2 Million Triangles in Real-Time. *Image by Alex Lindgren.*