BUILDING A HYPER-REALISTIC GRAPHICS RENDERER WITH GAUSSIAN SPLATTING

A Signature Honors Project

Presented to the Honors College

East Carolina University

In Partial Fulfillment of the Requirements for

Graduation with Honors

By

Ryan Hall

December, 2025

Approved By: David Hart, PhD

Department of Computer Science

BUILDING A HYPER-REALISTIC GRAPHICS RENDERER WITH GAUSSIAN SPLATTING

By

Ryan Hall

ABSTRACT

3-Dimensional (3D) graphics is a computer drawing technique in which 3D objects are rendered onto a computer screen. These 3D objects are composed of an array of points that map out the shape of the model. The goal of any graphically intensive program is to provide the user with an experience that is realistic, while maintaining a high frame rate. 3D graphics are typically represented through a collection of triangles which are connected to form a model. These triangles are ideal if geometric shapes are drawn. However, representing smooth shapes requires a large quantity of triangles and puts a processing burden on the computer. While this may not be a problem for models that have 10,000 triangles, more realistic models tend to have polygon counts of over 100,000, causing a severe decrease in frame rate. While other drawing techniques exist that can achieve realism, they typically have low frame rates and are not intended for use in simulations. However, a recent technique, called Gaussian Splatting, is capable of rendering highly realistic graphics at a suitable framerate. Each model rendered through Gaussian Splatting is composed of a sequence of nodes, known as splats, which are morphed through scaling and rotation. These points are morphed and grouped together into point clouds, which depict a highly realistic model.

The goal of this project is to implement Gaussian Splatting in the Open Graphics Library (OpenGL) and test out various effects, such as lighting, that are generally used with triangulated models. Visual comparisons and quantitative frame rate comparisons are made between both techniques. The Gaussian splatting renderer outperforms the triangulated renderer in visual quality while maintaining a high frame rate.

Table of Contents

Chapter 1

Introduction

3D graphics have traditionally been rendered using a long sequence of triangles that are grouped together to create a 3D model. A simple visualization of 3D mesh surface is visualized in Figure 1.1. For each frame, the mathematical calculations for these models are carried out and mapped on the screen to depict an object. With rigid models, such as cubes or pyramids, this drawing style would be sufficient, as they are pointed, geometric objects. However, to render smooth surfaces, more triangles are added to the model. While adding a few triangles may not be a problem, adding a large quantity of triangles to a model will likely influence the framerate, as the computer needs to do calculations for each of those triangles.

The framerate is the time that it takes for all the mathematical calculations and draw commands to be completed for one frame so that the next frame can be rendered. A framerate around 60 FPS (frames per second) is generally preferred, as a lower framerate causes the

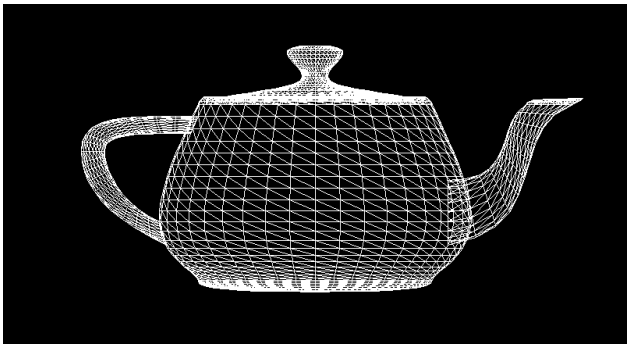

Figure 1.1: A wireframe rendering of a teapot, showcasing the triangular structure that generates the surface.

scene to have a rough, broken appearance. A 3D model of a desk that contains 10,000 triangles may not be a burden for the hardware to process, but scenes with 400,000 triangles could have a severe impact on the framerate. To render realistic 3D models with smooth surfaces at a high framerate, another drawing technique called Gaussian Splatting can be used.

This technique involves the creation of thousands of instanced squares that are positioned, rotated, and morphed to represent a 3D model. These squares are first rendered to be 2D shapes. When these shapes are rendered, they are rotated, scaled, and positioned in 3D space in order to form a shape. Additionally, a gradual translucency fade of the color allows the squares to appear as smooth shapes. Instancing allows for the geometric position data for each of these squares to be copied and rendered a certain number of times; in this case, the amount of squares that will be rendered will be equal to the amount of data points in the Gaussian Splat model's file. An example of a 3D Gaussian Splat model is provided in Figure 1.2.

The aim of this work is to recreate the Gaussian Splat rendering pipeline and additional effects in a highly efficient low-level graphics library. To do so, OpenGL and its Programmable Pipeline will be used. The Programmable Pipeline allows for use of the OpenGL Shading Language (GLSL). The program will be primarily made in C++, with some mathematical functions being handled with Assembly. To access OpenGL's Programmable Pipeline, the OpenGL Extension Wrangler library (GLEW) will be used. Alongside the programmable

Figure 1.2: A Gaussian Splat of a knitted toy.

pipeline, OpenGL also provides a fixed function pipeline for predefined shaders to be used. Using the fixed function pipeline, it is much easier and quicker to have geometry rendered onto the screen. However, it was decided that this pipeline would not be used because of its rendering and performance limitations. For Window drawing commands, the Graphics Library Framework library (GLFW) will be used. Many of the mathematical calculations are handled by the OpenGL Mathematics library (GLM). This library was chosen for its easy access to 3D vector data types and matrix multiplication functions.

For this experiment, lighting, and other effects that are typically used for triangulated 3D models will be simulated. Additionally, the frame rate of a Gaussian Splat are compared to a triangulated model of the same object. It is hypothesized that the Gaussian Splat with lighting effects provides high visual quality while maintaining framerates similar to the triangulated model, as the Gaussian Splat is rendered with a more efficient drawing technique that allows for quicker rendering at a higher framerate.

Chapter 2

Related Work

2.1 Radiance Fields and Gaussian Splatting

Novel view synthesis is a computer vision task that aims to generate a new view of a scene given some nearby views of the same scene. Deep learning approaches utilized structured light field images as input to generate new output images at specific locations [6, 20, 11]. In 2020, Mildenhall *et al.* [12] presented a new approach titled Neural Radiance Fields, where a neural network predicts the radiance at every 3D point in the scene rather than a set of 2D images. This foundational paper provided many advantages to solving the image synthesis task and fundamentally changed the research in this field. Many follow-on papers addressed issues such training speed [4, 14], memory size [1], and generality [10, 19].

One major disadvantage of the neural radiance field representation is that all structure to the scene is embedded within the learned weights of the neural network. This makes it difficult to interpret, evaluate, or edit the end result that is produced. Works such as Plenoxels [4] aimed to give an optimization-based representation of a radiance field that could be interpreted on a 3D grid.

In 2023, Kerbl *et al.* [8] presented Gaussian Splatting, a technique that represents the radiance field as a set of points in 3D space, each with their own color and radial shape. This new representation is easy to interpret and edit, while also being able to render new views quickly and with high resolution detail.

The original Gaussian Splatting method tends to produce a lot of 3D points in order

to represent the full 3D scene. This can lead to a large memory foot print. Many works have aimed to shrink the size of this foot print. In these works, the points are modified and reduced through clustering [17], pruning [3, 9], quantization [17, 15, 9], and parameter compression [13]. All of these approaches, however, rely on post-processing techniques that simply make the output Gaussian Splat smaller. Like these works, this project also aims to effectively render splats, but focuses more on low-level hardware optimizations rather than on reducing the Gaussian Splat file itself.

2.2 Splat Relighting

Gaussian Splats are real objects that are scanned and converted into 3D objects. Each of these 3D objects have their own colors, which may appear lighter or darker, depending on the environment in which the object was scanned. There have been numerous groups that have studied lighting effects with Gaussian Splats [2, 7]. In particular, Gao *et al.* [5] presented a lighting technique that involves the creation of pseudo-normal vectors to simulate light shining on the splat nodes. We build off this technique and implement it within our pipeline.

One key advantage of calculating the normal vectors for each program is that each model will be calculated immediately with normal vectors that are tailored to each splat node. Lighting techniques in computer graphics require the normal vectors at each point. For this work, the normal vectors at each splat node are precomputed ahead of time for computational efficiency. It is much more convenient if the models would be fitted with normal vectors at the start of the program. However, while this method is sufficient it also comes with many disadvantages. For example, it requires many complex mathematical equations to be calculated every frame. While this may not be an issue for some computers, other computers, especially older computers, may struggle under the weight of calculating normals every frame.

Chapter 3

Methodology

In this chapter, we present the 3D Gaussian pipeline. First, we describe the data flow for the OpenGL pipeline. Then, we describe the mathematics of 3D Gaussians for rendering within OpenGL. Finally, we present our lighting setup.

3.1 OpenGL Rendering

OpenGL has two pipelines: the programmable and fixed-function pipeline. The fixed function pipeline provides immediate mode functions, allowing for a graphical program to be created quickly. However, this pipeline has many limitations. One key limitation is that immediate mode does not allow for maximum control over what the computer draws on the screen. All of the geometry and the pixel data is already processed [18, p. 34]. Moreover, this pipeline has been deprecated since version 3.1. The programmable pipeline was chosen in order to provide maximum modifiability for each pixel and splat that is drawn on the screen. With this pipeline, the necessary mathematical equations and color values can be applied to each splat node. The OpenGL programmable pipeline requires the programmer to create a collection of shaders in order to manipulate what is rendered.

There are two main shaders that were used in the creation of this program: the vertex and fragment shader. The vertex shader allows for the vertex data of each triangle to be modified per triangle. The fragment shader allows for color values and other effects, such as transparency, to be applied to that individual vertex. These shaders are then linked to a

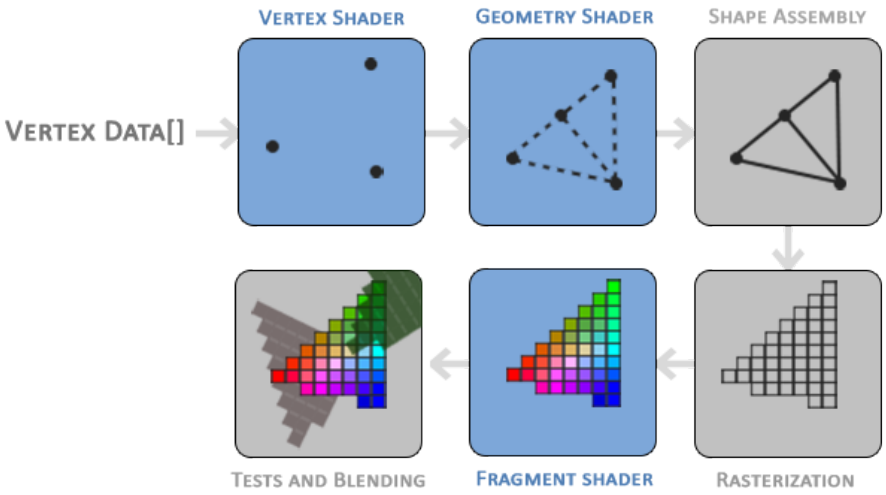

Figure 3.1: The shader pipeline for standard 3D geometry. Source: Copyright Joey de Vries, "Hello Triangle," LearnOpenGL.com (https://learnopengl.com/Getting-started/Hello-Triangle) (CC BY 4.0, https://creativecommons.org/licenses/by/4.0/). Twitter: @JoeyDeVriez.

shader program, allowing for the association between the application and the shaders. This pipeline is shown in Figure 3.2. It is important to note that a vertex shader is required for geometry to be drawn using the programmable pipeline. However, vertex and fragment shaders are typically linked together. For this Gaussian Splat renderer, only one vertex and one fragment shader were used. The shader code is managed by the GPU.

To render geometry to the screen, points have to be added to a Vertex Buffer Object (VBO). These points are then applied to a Vertex Array Object (VAO), which allows the program to send these points to the shaders under an attribute number. After the point data underneath that attribute number is retrieved, then it will go through the Vertex Shader Stage, where its geometry can be modified. After passing through this stage, the Fragment Shader Stage begins, and color values and other effects can be applied to this shape. To render each splat, a group of points were added to create the base splat node. These points, when rendered, would show a single square, a 2-D object. When this object is given 3-D coordinates, then it becomes a flat shape in 3-D space. The view and projection matrix calculations are both carried out on the CPU, and then sent to the vertex shaders to

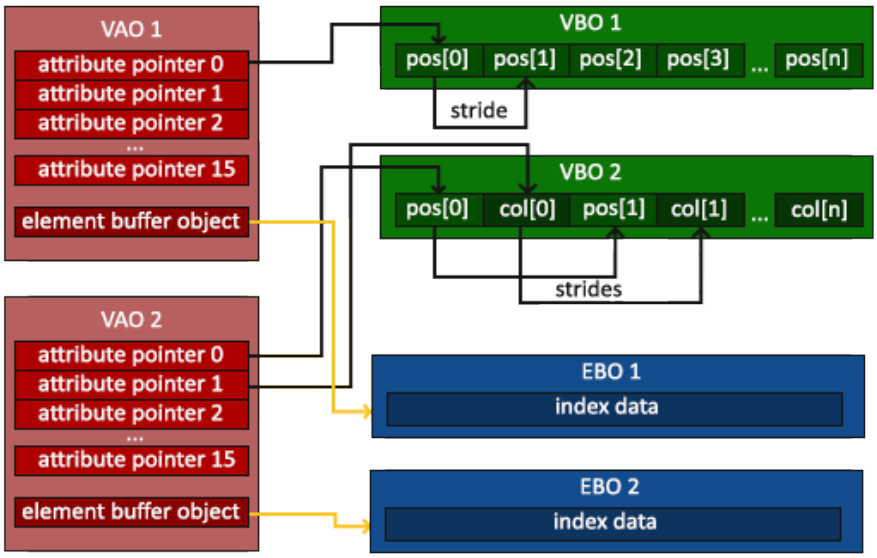

Figure 3.2: The data flow within OpenGL buffers. Source: Copyright Joey de Vries, "Hello Triangle," LearnOpenGL.com (https://learnopengl.com/Getting-started/Hello-Triangle) (CC BY 4.0, https://creativecommons.org/licenses/by/4.0/). Twitter: @JoeyDeVriez.

imitate movement. Gaussian Splats are made up of hundreds of thousands of splat nodes. A splat that is not drawn using efficient techniques will cause a severe decrease in render performance. Therefore, optimization is required to prevent slow framerates. The rendering technique used for this program is called instancing. This procedure allows the programmer to render the same group of points many times in a row [18]. Each splat file contains several data points for each splat node, such as, position, scaling, rotation, color, alpha transparency value, etc. Since instancing allows for a high amount of the same geometry to be drawn, the data from each point in the splat's file can be extracted and applied to its respective splat node, allowing for the model to be drawn in the shape of the splat node on the screen.

3.2 Preparing 3D Gaussian Splats for Rendering

After the splat begins to take shape, more formulae can be easily added to each splat node through the vertex shader. Before the formulae can be explained, it is important to note

that the scaling and rotation values needed to be modified in order for the following steps to be successful. For example, the rotational values calculated by common Gaussian Splat tools like Scaniverse [16] are in quaternions. Unlike Euler rotations, which are measured in degrees, quaternion rotations are 4-D values that represent a rotation. However, the GLM math library contained a function that converted these quaternions to Euler rotations, which were then sent to the vertex shader for each point. Additionally, the scaling values generated by Scaniverse are designed to be exponents of e, e.g., e^{scale_0}, e^{scale_1}, etc. However, as GLSL conveniently provides a function that allows for these values to be calculated, these values were simply calculated on the GPU in the vertex shader.

To start off, the model needs to be rendered as if it were a 2D shape. The necessary mathematical functions are then applied to the resulting model. After these calculations have been completed, the shape is then projected onto a 3D grid, and viewed using a virtual camera. To begin this process, a Jacobian matrix needed to be defined. A Jacobian matrix is a special type of matrix that is useful for coordinate transformations. It simply maps the coordinate conversion between a 2D grid and a 3D grid. The Jacobian matrix can be defined as follows:

$$J = \begin{bmatrix} \frac{f_x}{z} & 0 & -\frac{f_x x}{z^2} \\ 0 & -\frac{f_y}{z} & \frac{f_y y}{z^2} \\ 0 & 0 & 0 \end{bmatrix} \tag{3.1}$$

After this, a transpose variable T was created. This variable was defined as follows:

$$T = W^T J \tag{3.2}$$

where W is the 3x3 viewing matrix. Then, a 3x3 matrix V_{rk} is defined below.

$$V_{\text{rk}} = \begin{bmatrix} \Sigma_0 & \Sigma_1 & \Sigma_2 \\ \Sigma_1 & \Sigma_3 & \Sigma_4 \\ \Sigma_2 & \Sigma_4 & \Sigma_5 \end{bmatrix} \tag{3.3}$$

It is important to note that the matrix Σ is a 3×3 covariance matrix. It is constructed using the equation $\Sigma = MM^\top$, where M is obtained by converting the quaternion rotation values into a rotation matrix, which is then multiplied by the model's scaling values.

With these preliminary 3D matrices defined, then the following calculations and variables are made to represent the 2D shape. A 2D covariance variable cov2d is created. This variable is defined as follows:

$$\text{cov2d} = T^T \, V_{\text{rk}} \, T \tag{3.4}$$

Another variable, called mid, is created at this stage. This variable defines the average between the x and y variances, and is defined as follows:

$$\text{mid} = \frac{\text{cov2d}_{0,0} + \text{cov2d}_{1,1}}{2} \tag{3.5}$$

A radius variable is also necessary. To define this variable, an x and y value are needed. The x value is defined as the difference between the x and y variance variables.

The y value is just the value accessed from $(\text{cov2d})_{0,1}$. The radius is then the vector norm of the x and y values.

$$\text{radius} = \left\| \begin{bmatrix} (\text{cov2d})_{0,0} - (\text{cov2d})_{1,1} \\ (\text{cov2d})_{0,1} \end{bmatrix} \right\| \tag{3.6}$$

After this, two lambda variables need to be defined. The first lambda variable, λ_1, is the sum of the mid variable and the radius variable. The second lambda variable, λ_2, is simply the difference between the mid and radius variables. It is also important that value of the

λ_2 variable not fall lower than 0.0. In the case that this happens, the function is returned, omitting the remainder of the vertex shader code.

$$\lambda_1 = \text{mid} + \text{radius} \qquad \lambda_2 = \text{mid} - \text{radius} \tag{3.7}$$

A diagonal vector then needs to be created. However, this vector needs to be normalized. Fortunately, GLSL provides a normalize function. The normalize function takes in a 2D vector. The x value of this vector will be $\text{cov2d}_{[0][1]}$, while the y value will be the difference between the lamba1 variable and $\text{cov2d}_{[0][1]}$. The diagonal vector variable diagonalVector is equal to the normalized result of this vector.

$$\text{diagonal\_Vector} = \frac{\begin{bmatrix} (\text{cov2d})_{0,1} \\ \lambda_1 - (\text{cov2d})_{0,0} \end{bmatrix}}{\left\| \begin{bmatrix} (\text{cov2d})_{0,1} \\ \lambda_1 - (\text{cov2d})_{0,0} \end{bmatrix} \right\|} \tag{3.8}$$

After the necessary prerequisite variables were defined, the eigen vectors can be defined. For the sake of this project, these vectors were named majorAxis and minorAxis. These two variables utilize GLSL's min function. The min function is equal to the smaller of the two parameter values.

Therefore, the majorAxis and minorAxis variables are defined as follows:

$$\begin{aligned} \text{majorAxis} &= \min\left(\sqrt{0.015 \times \text{lambda1}},\ 1024.0\right) \times \text{diagonalVector}, \\ \text{minorAxis} &= \min\left(\sqrt{0.015 \times \text{lambda2}},\ 1024.0\right) \times (\text{diagonalVector}_x, -\text{diagonalVector}_y) \end{aligned} \tag{3.9}$$

After the eigen values have been defined, a vCenter variable is defined. This variable is the center of the splat in screen space.

$$\text{vCenter} = \frac{\text{vec2(pos2d)}}{\text{pos2d}.w} \tag{3.10}$$

With the creation of the eigen values and the definiton of the splat's center point, the splat node's calculations can finally be assembled. To do this, the gl_Position variable is set to a 4D vector. The points that follow define the shape of the geometry that gets rendered onto the screen. The following is the final definition of the gl_Position variable for each splat node:

$$\text{gl\_Position} = \text{vec4}\left(\text{vCenter} + \frac{\text{position}.x \times \text{majorAxis}}{\text{viewport}} + \frac{\text{position}.y \times \text{minorAxis}}{\text{viewport}},\ 0.0,\ 1.0\right) \tag{3.11}$$

Although the geometry for each splat node has been correctly handled by the vertex shader, this is not sufficient for the Gaussian Splat to render properly. The set of calculations describe the ellipse-like components of the Gaussian Splat, but ellipses are not a native primitive of OpenGL, and typically requires drawing rectangles or quadrilaterals to the screen. To render each circle as a splat would require an astronomical amount of calculations per splat, which could have a drastic impact on the performance of the renderer. The fragment shader is used to overcome this obstacle. On each node, a circle would be rendered onto the rectangular splat node. While a texture could be used to handle this calculation, that comes with a number of limitations. One key limitation is that rendering each circle as a texture would limit the modifiability of each circle. A second key limitation is that a high number of textures rendered could slow down the renderer. A quick and efficient alternative was found. A radius was calculated from the center of each splat node. Any part of the splat node within this radius would have its color rendered, with the rest of the splat node being discarded. What remains are seemingly circular splat nodes, that are translated, rotated, and scaled accordingly.

Although the points are rendered correctly, the problem regarding the order in which

they are rendered still exists. In order for these points to render in the correct order, the splat nodes must be sorted from nearest to farthest with respect to the camera's position. Although instancing helps speed up the raw rendering time of the drawn splat model, doing a sorting algorithm every frame still causes a dramatic drop in frame rate. This drop in frame rate is caused by the sorting algorithm attempting to sort hundreds of thousands of points per frame. In order to fix this, mulithreading was utilized. Multithreading allows for the rendering to be handled in one thread, and the sorting to be calculated in a second thread. This allows for the rendering to be handled without a drop in performance, while also allowing for the points to be appropriately sorted with respect to the camera. However, the sorting algorithm is not calculated every frame on the thread. This calculation is handled on how much the camera is rotated. The difference in the camera's rotation is handled using the dot product. After the dot product is applied, the sorting algorithm is applied on a second thread only when the absolute difference between the dot product and the previous camera position is greater than 0.01. If it is smaller than this number, the function returns, and nothing happens.

3.3 Lighting of 3D Gaussian Splats

For a Gaussian Splat, lighting was handled similarly to how it would be done with a traditional, triangulated model. However, Gaussian Splats scanners typically do not account for lighting effects, and do not provide normal vectors when each splat is processed. Since lighting requires normal vectors to be calculated, a different approach is needed. Originally, a pseudo-lighting system was used based on the position of a single vertex point representing the light. This light value was calculated for each splat node, and their color values were increased or decreased with respect to their position relative to the light. However, this presented some issues. In particular, when the light was supposed to be blocked by the splat, any objects that were near the light still would light up, since the lighting for each point was handled with respect to the position. Objects in the middle were not accounted for. It

was decided that this pseudo-lighting technique had too many limitations, and its functionality was ultimately removed. As an alternative, it was decided that the most practical and efficient path would be to manually generate normal vectors for each point. One method that was originally proposed was to use the K-Nearest Neighbor algorithm to generate these values based on their locations relative to other points. A program that would generate these normal vectors was prepared, but due to hardware limitations, this process would have taken several hours, and any mistakes made would cost nearly an entire day.

After several failed attempts at generating these normal vectors using K-Nearest Neighbor, it was decided that Blender would be the best option. Blender is a free 3D modeling tool commonly used for creating high definition 3D models, animations, and even has a built in game engine. Although Blender does not natively support Gaussian Splats, it sufficiently supports the .PLY file format. After discovering this, the splat was loaded into Blender. Blender has an algorithm that allows for the automatic generation of normal vectors for a 3D model. Although Gaussian Splats are not the same as traditional 3D models, the normal vectors would behave in the same way because both techniques preserve a global 3D structure. Using Blender's automatic normal generation tool, normals were then generated for each splat node. Likewise, the calculations for the lighting of each splat would be the same as a triangulated model.

Lighting is handled through the same calculations with this Gaussian Splatting renderer as a triangulated model. This was possible because all of the splats were treated as their own independent models in this respect, in that they were given their own normal vectors. This allowed the calculations to be done in an identical way as they would be done with a triangulated model. It is important to note that the data for the normal vectors is loaded into the program along with the other data, such as the position and the color, for each point.

Chapter 4

Results

For the following section, the results were divided into two parts: qualitative and quantitative results. The qualitative results will assess the visual quality differences between the Gaussian Splat renderer and its triangulated counterpart. The quantitative results demonstrate performance and the overall rendering speed and usability of the programs.

4.1 Qualitative Results

A set of Gaussian Splat scenes and triangulated meshes were compared in our experiments. Images were taken of both model rendering techniques with and without lighting.

For triangulated meshes, lighting is essential to make realistic looking results. No lighting effects exposes the underlying colors of the model without any variation in appearance. The eye expects shadows and highlights to infer geometry. A demonstration of this principle is shown in Figures 4.1 and 4.2.

By comparison, Gaussian Splat scenes have realistic lighting effects baked into the rendering. The ultra-realistic scenes can be rendered with no additional considerations. The renderer built for this project provides quality results for any Gaussian scene, as demonstrated in Figure 4.3.

Additionally, the built renderer has an option for additional lighting effects to provide more control to the user. This lighting is similar to the techniques used for triangulated mesh. The quality of this additional lighting effect is shown in Figure 4.4.

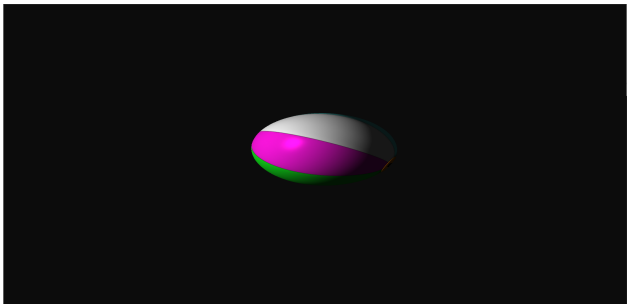

Figure 4.1: The Traditional 3D model rendering program with lighting effects applied.

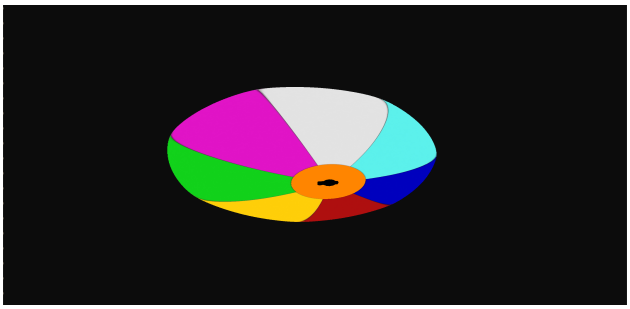

Figure 4.2: The Traditional 3D model rendering program without lighting effects applied.

The two renderer programs are designed with the same lighting techniques. However, it is noticeable that the lighting technique in the first image looks fuzzy compared to the smooth and shiny lighting demonstrated by the second model. However, the model in the first image has a photographic appearance, providing more realism, although it is rather fuzzy in some areas. The beach ball, with its polished and shiny edges, appears almost too perfect. Regardless of its fuzziness, Gaussian Splatting manages to capture a photorealistic scene for a model, a level of realism that can simply not be obtained through 3D triangulated artwork.

4.2 Quantitative Results

The Gaussian Splat renderer and the triangle-based renderer were compared based on their performance with a model that is similar in complexity. The Gaussian Splat model used in the example is a toy that resembles the Tower of Hanoi. The triangle-based renderer uses a 3D model of a beach ball. The machine used for the experiments, by modern standards,

Figure 4.3: The Gaussian Splatting program without lighting effects applied.

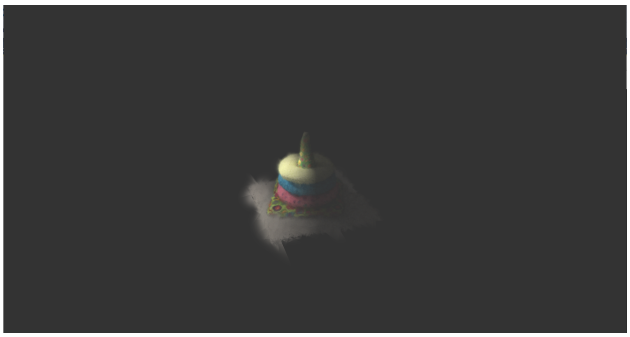

Figure 4.4: The Gaussian Splatting program with lighting effects applied.

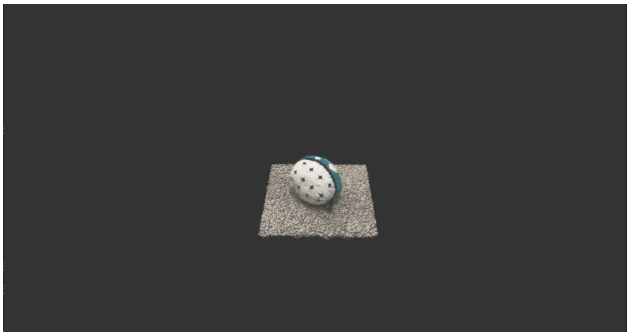

Figure 4.5: A second Gaussian Splat model rendered without lighting effects applied.

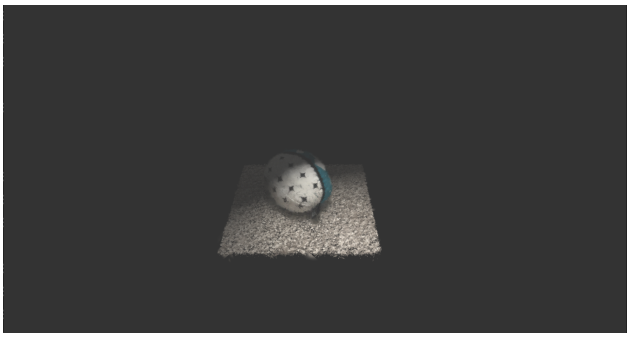

Figure 4.6: A second Gaussian Splat model rendered with lighting effects applied.

is quite underpowered. All results were conducted on a nearly 10 year old machine running Windows 10 via Boot camp. It is equipped with an Intel® Core™ i7-6700K CPU @ 4.00 GHz (4 cores, 8 threads), 8 GB RAM, and an AMD Radeon R9 M395X GPU.

In terms of FPS, the triangle-based renderer performed significantly better than the Gaussian Splat renderer. However, in both cases, the FPS was well above 60, the standard for real-time applications. This is shown in Table 4.1. This is done while maintaining hyper-realism. For decades, developers have attempted to capture a real-life effect. While they succeed in creating life-like graphics, they have many limitations, especially when compared to Gaussian Splatting. One of these limitations is the technique. Computers will never be able to make enough triangles to ensure that a scene will never have a jagged and pointy appearance. If the 3D camera is positioned close enough to a 3D model, the individual triangles will be visible. Another limitation is the fact that 3D models are created by artists. Just as with paintings, 3D models are able to seem life-like, but they will always appear slightly different from the object that it imitates. With Gaussian Splatting, these limitations are easily overcome, allowing for a photorealistic view of a model. Although the Gaussian Splat used for the above figures has some blurry spots near the bottom, this slight speed decrease is a reasonable tradeoff for the improved appearance of the splats compared to their triangulated counterparts.

One area for potential improvement for the renderer is the preprocessing times. The preprocessing times to render Figure 4.3 and Figure 4.4 were significantly higher than their triangulated model counterparts, with the Gaussian Splat renderer completing the task in 50 seconds, significantly higher than the triangulated 3D renderer, which was able to complete the preprocessing stage in less than 10 seconds. However, with a more simple model, like the one shown in Figure 4.5 and Figure 4.6, preprocessing time was reduced to 20 seconds. It is important to note that the lighting is not calculated during this preprocessing step. The lighting is calculated during the fragment shader phase of rendering, and is calculated every frame along with the splat nodes' colors.

Table 4.1: Comparison of the framerate of the Gaussian Splat renderer and the traditional, triangular renderer, both with and without lighting calculations. It should be noted that this rendering approach was tested with relatively smaller scenes, limited to 1-2 objects. Moreover, the addition of the vector lighting calculation had little to no impact on the framerate.

Metric	**Gaussian Splat Renderer**	**Triangle-Based Renderer**
Framerate With Lighting	200 FPS	1900 FPS
Framerate Without Lighting	200 FPS	1900 FPS

Figure 4.7: A Gaussian Splat rendering of a shop.

Next, to test the limits of the gaussian renderer built for this project, we will compare the metrics of these renderers for larger, more detailed scenes. Figure 4.7 shows the rendering of a very large Gaussian Splat scene of a shop. It contained over 100,000 individual points.

While there is no doubt that the scene that is being rendered is, indeed, a shop, the limitations of the Gaussian Splat renderer really seem to manifest with models that are more complex. The preprocessing time for this scene was 5 minutes, and the framerate barely rose above 10 FPS.

There are several possible reasons behind this. One reason could be that computers have been optimized to draw triangle-based geometry. Since Gaussian Splats are not triangle-based, that could lead to the possible significant drop of framerate. Another possible cause is that the Gaussian Splat renderer, although it uses instancing, still has to do more calculations than triangulated models, since the Gaussian Splats have over 100,000 splat nodes, each with its own geometry that has to be calculated. We compare the exact speed with a similarly sized triangular mesh in Table 4.2.

Table 4.2: Comparison of the framerate of the Gaussian Splat renderer and the traditional renderer. This was tested with large scenes. Lighting is not accounted for in this graph due to its insignificant effect on the framerate.

Metric	**Gaussian Splat Renderer**	**Triangle-Based Renderer**
Framerate With Lighting	11 FPS	600[1] FPS
Framerate Without Lighting	11 FPS	600[1] FPS

[1]This metric was obtained from a model that was deemed to be similar enough in detail to the coffee shop Gaussian Splat. This was a free model obtained from the internet. However, the texture mapping format was incompatible with the traditional renderer software, but ignored, since texture mapping has an effect so negligible that it would exceed the precision of this study.

Chapter 5

Conclusion

Gaussian Splatting is a rendering technique that is aimed at providing an image that appears as realistic and detailed as possible. This technique can help produce graphics that are comparable to ray casting graphics, except that they can be rendered in real time, rather than individual frames. They are generally obtained with cameras with LiDAR capabilities. These models are simply a cloud of points in which every point is morphed to form the shape of a model. For this experiment, lighting was added to the models to compare the difference in rendering performance between a Gaussian Splat renderer and a traditional triangle-based renderer. At the end of the experiment, it was discovered that the Gaussian Splat program was significantly outperformed by the triangle-based renderer. However, the Gaussian Splat renderer is capable of providing picture-like graphics, at a high framerate. Although realistic graphics can be produced through using 3D modeling software, the realism and framerate obtained through Gaussian Splatting is unmatched by any other rendering techniques.

BIBLIOGRAPHY

[1] Barron, J. T., Mildenhall, B., Verbin, D., Srinivasan, P. P., and Hedman, P. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. *CVPR* (2022).

[2] Bolduc, C., Hold-Geoffroy, Y., and Lalonde, J.-F. Gaslight: Gaussian splats for spatially-varying lighting in hdr. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)* (October 2025), pp. 29120–29130.

[3] Fan, Z., Wang, K., Wen, K., Zhu, Z., Xu, D., and Wang, Z. Lightgaussian: Unbounded 3d gaussian compression with 15x reduction and 200+ fps, 2023.

[4] Fridovich-Keil, S., Yu, A., Tancik, M., Chen, Q., Recht, B., and Kanazawa, A. Plenoxels: Radiance fields without neural networks. In *CVPR* (2022).

[5] Gao, J., Gu, C., Lin, Y., Zhu, H., Cao, X., Zhang, L., and Yao, Y. Relightable 3d gaussian: Real-time point cloud relighting with brdf decomposition and ray tracing. *arXiv:2311.16043* (2023).

[6] Kalantari, N. K., Wang, T.-C., and Ramamoorthi, R. Learning-based view synthesis for light field cameras. *ACM Trans. Graph. 35*, 6 (dec 2016).

[7] Kaleta, J., Kania, K., Trzciński, T., and Kowalski, M. Lumigauss: Relightable gaussian splatting in the wild. In *2025 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)* (2025), pp. 1–10.

[8] Kerbl, B., Kopanas, G., Leimkühler, T., and Drettakis, G. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics 42*, 4 (July 2023).

[9] Lee, J. C., Rho, D., Sun, X., Ko, J. H., and Park, E. Compact 3d gaussian representation for radiance field. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2024), pp. 21719–21728.

[10] Martin-Brualla, R., Radwan, N., Sajjadi, M. S. M., Barron, J. T., Dosovitskiy, A., and Duckworth, D. NeRF in the Wild: Neural Radiance Fields for Unconstrained Photo Collections. In *CVPR* (2021).

[11] Mildenhall, B., Srinivasan, P. P., Ortiz-Cayon, R., Kalantari, N. K., Ramamoorthi, R., Ng, R., and Kar, A. Local light field fusion: Practical view

synthesis with prescriptive sampling guidelines. *ACM Transactions on Graphics (TOG)* (2019).

[12] Mildenhall, B., Srinivasan, P. P., Tancik, M., Barron, J. T., Ramamoorthi, R., and Ng, R. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV* (2020).

[13] Morgenstern, W., Barthel, F., Hilsmann, A., and Eisert, P. Compact 3d scene representation via self-organizing gaussian grids. *arXiv preprint arXiv:2312.13299* (2023).

[14] Müller, T., Evans, A., Schied, C., and Keller, A. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph. 41*, 4 (July 2022), 102:1–102:15.

[15] Navaneet, K., Meibodi, K. P., Koohpayegani, S. A., and Pirsiavash, H. Compact3d: Smaller and faster gaussian splatting with vector quantization. *arXiv preprint arXiv:2311.18159* (2023).

[16] Niantic, Inc. Scaniverse: 3d scanner app, 2024. Mobile application for iOS and Android.

[17] Niedermayr, S., Stumpfegger, J., and Westermann, R. Compressed 3d gaussian splatting for accelerated novel view synthesis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2024), pp. 10349–10358.

[18] Shreiner, D., Sellers, G., Kessenich, J., and Licea-Kane, B. *OpenGL Programming Guide*, 8 ed. Addison-Wesley Professional, 2013.

[19] Wang, Q., Wang, Z., Genova, K., Srinivasan, P., Zhou, H., Barron, J. T., Martin-Brualla, R., Snavely, N., and Funkhouser, T. Ibrnet: Learning multi-view image-based rendering. In *CVPR* (2021).

[20] Wu, G., Zhao, M., Wang, L., Dai, Q., Chai, T., and Liu, Y. Light field reconstruction using deep convolutional network on epi. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), pp. 1638–1646.