School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Final Report

Rendering 3D Gaussian Splatting scenes

Thomas Chernaik

Submitted in accordance with the requirements for the degree of
Computer Science with High Performance Graphics and Games Engineering MEng, BSc

2023/24

COMP3931 Individual Project

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Final Report	PDF file	Uploaded to Minerva (30/04/2024)
Link to online code repository	URL	Sent to supervisor and assessor (30/04/2024)

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) Thomas Chernaik

© 2023/24 The University of Leeds and Thomas Chernaik

Summary

This paper describes in detail the process of rendering a radiance field represented with 3D Gaussian splats interactively using GPU (Graphical Processing Unit) acceleration and tiled rendering, based on the implementation described in the original paper [1]. It outlines the maths behind the rendering equations, as well as describing how they can be implemented on the GPU. A GPU approach to depth sorting is also described in detail. Results and analysis of my implementation are also provided, where it can be seen that I successfully implemented a rasteriser using OpenGL compute shaders which can render millions of splats interactively.

Acknowledgements

Thanks to my supervisor Markus Billeter for feedback, advice, and help given.
Thanks to my family and friends for listening to me talk about this over and over again.

Contents

Chapter 1

Introduction

A light field is a function which maps from any given point in space, and viewing direction at that point, to the light information you would see when looking from that point in that direction, for a given scene [2]. There are many ways to approximate this function [3, 4, 5, 6], but this paper focuses on rendering a light field stored using a set of 3D Gaussian splats (also referred to as just *splats* or a *splat* throughout the paper) to represent a radiance field. Each scene is made up of hundreds of thousands of discrete splats, which together approximate the light field function. They can be rendered interactively, and in real time, to generate high quality renderings. This paper will describe and justify an implementation that mostly follows the tiled and projected rendering implementation in the seminal paper [1]. The rendering method utilises integration along the viewing ray described in EWA (Elliptical Weighted Average) splatting [7] to project the scene to the viewing plane, before sorting efficiently using a GPU sort [8], and rasterising the image using alpha blending [9]. The rendering method utilises the parallel nature of GPU compute shaders to achieve desirable rendering times. I implemented a renderer using C++ and OpenGL compute shaders which rendered scenes interactively, that could be explored using keyboard controls.

My aim for my project was to render Gaussian splatting scenes and be able to move a camera around the scene to render different views interactively. In order to achieve interactive rendering, GPU acceleration must be utilised. There are several frameworks for programming on the GPU. Some frameworks are designed for general purpose GPU (GP-GPU) programming, such as OpenCL and CUDA, whereas others are designed for graphics applications, such as OpenGL, DirectX, Metal, and Vulkan. I wanted to have interactive rendering, which means my application needs to display the render and take keyboard inputs, so the GP-GPU frameworks were not chosen. DirectX and Metal are only available on Windows and Apple devices respectively, and I preferred a cross-platform option, so these were not chosen. Both modern OpenGL and Vulkan provide the tools I needed - compute shaders and the ability to easily display the output. Vulkan would have likely provided a performance bonus over OpenGL, but there is less abstraction in Vulkan, meaning it is easier to write programs for OpenGL. I am familiar with OpenGL, so it was my chosen framework. I chose to use C++ as the programming language to interact with OpenGL for a number of reasons. C/C++ is the language the OpenGL bindings were written for, and I am familiar with using OpenGL with C++. I used the CLion IDE to write my project, making use of the debugging tools. Git was used for version control, with a remote on GitHub.

Chapter 2

Literature Review

2.1 Light Fields

Plenoptics and light fields are a long studied field. The term light field originates in the 1939 book titled The light field [2]. The book proposes that each point in space, and each direction to each point, can be assigned a quantity representing the light at that point and direction. The book describes a light field as a function mapping from five inputs, the position and direction in 3D space, to the light visible at that viewing vector [2]. This function could be considered as the sum of various *illumination distribution solids* - functions that describe the light emitted by points in space, or the sum of various light fields for each light source [2]. These are unsatisfactory as a representation of a complex real life scene, and the forward of the translation calls for a better representiation, proposing using tensor methods in five dimensions. The fundamental point is that the light field, or plenoptic function, is a five-dimensional function describing the flow of light at every 3D position (x, y, z) for every 2D direction (θ, ϕ) [5].

In 1996, two papers, the Lumigraph [5], and Light Field Rendering [6] proposed methods to represent a light field, and render them to produce new images. They describe capturing a light field from real life. The Lumigraph uses photos taken from known (or calculated) positions, and depth information estimated using knowledge of the geometry of the scene to generate its light field representation. The paper notes that traditional computer graphics techniques would struggle to achieve the complexity of the geometry and lighting effects [5]. This is due to the abstraction of complexity when using real world, complex input data (photographs) to train an approximate representation. Both the Lumigraph and Light Field Rendering do not provide methods to approximate the full 5D function, and put viewing restrictions on the virtual camera. The Light Field Rendering requires viewing rays to pass through two specific planes, and neither allow the camera to explore inside the light field, meaning they can only represent objects, not scenes [5, 6].

Radiance field is a term used to refer to the field of light emitting/reflecting from surfaces. This contrasts to light field, which model the field of light received at a point. Light fields model what you would see with your eyes, so are what we desire to reach to visualise images. Radiance fields contain all the information needed to calculate light field information at a point, and render an image. This can be done by stepping through the radiance field until we accumulate the radiance for that viewing vector. This matches the rendering techniques for modern radiance field methods, such as Neural Radiance Fields (NeRF) and Gaussian splatting, which both can render using either ray marching [3] or direct (approximate) integration [1]. It can be argued that radiance fields are a representation of a light field, as light field data can be directly and precisely computed, and this conversion from radiance field to

light field is needed to render images. The main distinction is that while radiance fields must have some model of the surfaces in a scene, a light field does not necessarily need to know about the geometry of the scene.

Neural Radiance Field rendering (NeRF) was introduced in 2020 as a new method to represent a light field [3]. They represent the full continuous function of the radiance field, able to be sampled from any position, unlike the previously described methods [3, 5, 6]. This is achieved by training a neural network which maps the function from a 5D coordinate (x, y, z, θ, ϕ) to an RGB colour. Unlike traditional light fields, instead of the 5D coordinate being the position of the viewing ray, the 5D coordinate is the vector of light emission (or reflection, which is not different in this representation). The idea to capture the radiance field instead of received light marks a difference from the previously described methods, which attempt to store the light field as how it is received, not emitted. NeRF are rendered using a weighted sum of samples along the camera ray. Much like the previous methods, they are trained using images and camera position information. The quality of NeRF resulted in a large amount of research in the field, with many papers building on the original implementation.

One drawback to NeRF is that in order to convert to the light field, the radiance field must be sampled many times, in a process called ray marching [3]. This is costly, and results in slow rendering times. This is addressed with an alternate method, Neural Light Field (NeLF) rendering. Here, the network only needs to be sampled once for each viewing ray, as the light field is directly stored instead of obtaining it from a radiance field. There are additional benefits to NeLF over NeRF such as better rendering of non-Lambertian effects in practice, but it does not achieve desirable rendering times [10].

Another drawback is that the sampling in ray marching is at discrete points, which can result in aliasing artifacts. Multiscale Representation for Anti-Aliasing Neural Radiance Fields (Mip-NeRF) solves this by encoding the radiance field differently, and rendering by tracing conical frustums, instead of infinitesimally small rays. This results in no more aliasing artifacts. Additionally, it provides significant performance improvements over standard NeRF, partly due to containing a representation for the scene at different scales in one encoding [11]. This method still uses a neural encoding of the radiance field, and requires many samples in the ray marching step of rendering.

One more drawback is that neural representations are not easily editable. Some implementations of light/radiance field rendering propose different encodings of the light/radiance field than neural networks, such as Plenoxels [4], and 3D Gaussian Splatting [1]. These encodings are discrete representations that could be manipulated in 3D space to manually improve, crop, and edit scenes. Much like all the NeRF methods, scenes could be trained using gradient descent from images and camera locations. Plenoxels represent a scene as a sparse 3D grid with spherical harmonics [4], whereas 3D Gaussian splats represented scenes as a set of 3D Gaussian splats [1]. Gaussian Splatting provides better performance with current methods than Plenoxels [1, 4].

2.2 Splatting

Splatting is a volume rendering technique, first proposed by Westover [12]. Westover's later thesis, Splatting [13], describes splatting as throwing splats at a wall, which flatten and spread across the wall. Splats thrown later obscure splats thrown earlier. As more splats are thrown, an image emerges on the wall. Each splat has an opacity and a colour. The opacity determines how much of the light behind it is blocked, and the colour gets blended with the background colour based on the opacity [13]. Westover describes in this paper a method for rendering volumes using splatting. It involves sampling the rendering volume using a splat kernel of choice, and then rendering the splats. The rendering can be performed by calculating the footprint function for a splat, then calculating how it contributes to each pixel in the image. Evaluating the footprint function requires an integration, which varies depending on the kernel function [13]. This method carries through to my implementation, which roughly follows the same steps.

It should be noted that unlike my implementation, the original splatting used splatting kernels to sample an arbitrary representation of a volume, instead of directly representing the volume as splats.

Using a Gaussian Kernel for splatting was proposed in the Westover's Splatting Thesis [13], but it was fully defined in EWA Volume Splatting [14]. EWA volume splatting defines equations for integrating (projecting) the Gaussian kernel, which were used for my implementation and outlined in **Chapter 4**. Using Gaussians as the splat kernel has certain features that are ideal. Gaussians have 2 particular features that make them appropriate for splatting. Gaussiains are closed under affine mappings and convolution, and integrating a 3D Gaussian results in a 2D Gaussian. [14]. These features allow better sampling of the projected splats.

2.3 Radiance Fields with 3D Gaussian Splatting

A Gaussian distribution, or normal distribution, is a continuous distribution, typically representing a probability distribution, described by **Equation 2.1**, and described by Gauss [15].

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{2.1}$$

This can be generalised to a multi-variate Gaussian distribution to represent higher dimensions. $\mathbf{X}$ represents an N-dimensional Gaussian distribution, and

$$\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \tag{2.2}$$

$$\mathcal{N}(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{N/2}} \frac{1}{|\boldsymbol{\Sigma}|^{1/2}} \exp\{-\frac{1}{2}(\mathbf{X}-\boldsymbol{\mu})^{\mathbf{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{X}-\boldsymbol{\mu})\} \tag{2.3}$$

where $\boldsymbol{\mu}$ is an N-dimensional vector representing the mean of the distribution in each dimension, and $\boldsymbol{\Sigma}$ is an $N \times N$ covariance matrix of the distribution [16]. When used in splatting, the value sampled from the covariance matrix at a point represents the density of

the splat - which corresponds to the opacity we will see.

3D Gaussian Splatting for Real-Time Radiance Field Rendering [1] defines a new technique for rendering radiance fields, using splatting with the splats represented as three-dimensional Gaussian distributions. Representing scenes as splats provided faster rendering times than previous methods, such as NeRF, while maintaining scene quality. The models themselves for both NeRF and Gaussians can be created via software that takes in photos and outputs the models. The creation of models for Gaussian Splatting is a faster process than for NeRF, showing another benefit. Splatting scenes can be rendered alias free [17], eliminating the aliasing problems found with NeRF, and solved with Mip-NeRF.

Gaussian Splatting models are generated using gradient descent. A set of splats are initialised from the point field generated by Structure from Motion (SfM). The splats are then optimised iteratively using gradient descent against a set of input images of the scene the splats must represent. Adaptive density control is used where splats can split or combine to reduce or increase the resolution of splats at specific points in the scene where needed. The models I used for my test renders were generated using this technique, and sourced from the internet.

Tiled rendering is used to achieve faster rendering times. Tiled rendering for Gaussian splatting was inspired by the rendering used for Pulsar, a previous rendering technique [18, 1]. It involves splitting up the splats into tiles (a 16x16 grid in the original paper [1]) on the screen they correspond to at render time. This reduces the number of splats that need to be considered for each pixel, resulting in the desired fast rendering times.

3D Gaussian Splatting for Real-Time Radiance Field Rendering [1] concludes that this is the first approach that "that truly allows real-time, high-quality radiance field rendering, in a wide variety of scenes and capture styles, while requiring training times competitive with the fastest previous methods." *Real-time* for the paper is defined as frame rates $\geq$ 30fps, run on an A6000 GPU [1], which is a massively faster device than the device my benchmarks are run on, the integrated graphics of an AMD Ryzen 5 pro 5650u APU, and 16GB DDR4 RAM. The paper shows quality on par or slightly better than Mip-NeRF rendering, making it the state of the art for radiance field rendering [1].

Chapter 3

Methods

3.1 Splats

The splats need to contain the information to calculate the light field data at any point. Splats are aligned roughly to surfaces of geometry, so in order to capture non-Lambertian effects (lighting that means a point on a surface may look different when viewed from different angles), splats must be an anisotropic (different values from different viewing angles) representation. There are three main components to each splat. The first is the *mean*, or central location of the splat. This defines where in the scene it resides, and is isotropic, so can simply be represented by a three-vector. The other two components are the opacity and the colour, which define how the splat contributes to the lighting. In order to make the opacity anisotropic, it is represented as a 3D Gaussian distribution. The weight of the distribution is sampled when rendering to calculate the overall opacity, and sampling a Gaussian distribution is dependent on the incident angle, making it anisotropic. There is also a base opacity, which combines with the Gaussian-sampled opacity to calculate the overall opacity. The colour can also be represented anisotropically with a spherical harmonic, but the anisotropic properties of the opacity are sufficient for a reasonable scene quality, so this is omitted from my project to limit scope. Hence, the colour is simply represented as an RGB value.

3.2 Rendering

Rendering a scene requires sampling every splat for each pixel, in order from front to back, and blending the colour together to work out the value for the pixel. Before sampling, the splats must be transformed from world space to screen space. World space is where things are in the 3D virtual space that maps to the real world, and how they are stored in this case. Screen space is the space where the x-y axis of the scene is the screens' width and height, and the z-axis (depth) is the distance from the camera, normalised between zero and one [19]. There are two separate elements that need transforming from world space to screen space for rendering. The first element is the mean of the splats, which can be transformed using a matrix transformation. The second element is the Gaussian distribution, which gets flattened along the z-axis using an integration [7]. The exact equations for these are described in **Chapter 4**.

To understand how we sample the Gaussian, the Gaussian should be interpreted as a density field, where the weight maps to a density. In order to calculate the transparency along a viewing ray, the density field must be summed up all along the ray. This could be achieved with a ray-tracing based approach, where the density is sampled at intervals along the ray to calculate the overall density. This would be computationally expensive, due to needing to have a calculation at many sample points, for every pixel, for just one splat. This would also introduce aliasing due to the sampling rate. If the sampling rate is less than the Nyquist

frequency (double the frequency of the data being sampled), then we will miss splats in the scene [20]. This can be avoided with sufficiently high sampling rates but this comes with significant performance cost. Instead, we can integrate the 3D Gaussian to sum up the density through the viewing ray in one computation. This produces a 2D Gaussian, which can then be used by each pixel to calculate the contribution of the splat. A ray-tracing approach for one splat would require sampling the 3D Gaussian many times for every pixel, but this approach requires computing a 2D Gaussian once via a numerical method integration, and then sampling that 2D Gaussian once for each pixel [1], providing a significant performance bonus, and eliminating aliasing issues along the z-axis.

Once the opacity for a splat is computed for a pixel, it must be blended (together with the colour) with the opacity and colours of all other splats. The alpha blending technique used is similar to the *GL_ONE_MINUS_SRC_ALPHA* blending mode in OpenGL [9], and updating the destination alpha to the new value. The exact computation can be seen in **Equation 3.1** and **Equation 3.2**. The exact equation for this is described in the implementation. The blending operation is not commutative, so the splats must be blended in a particular order, front to back. This means the splats must be sorted before the blending takes place. An example of a scene rendered without ordering can be seen in **Fig. 3.1**.
$\mathbb{C}$ represents a colour, with $\mathbb{C}_0$ being the original colour, and $\mathbb{C}_1$ being the new colour. A colour has four components - the rgb colours and the α opacity.

$$\mathbb{C}_{1,rgb} = 1 - \mathbb{C}_{1,\alpha} * (1 - \mathbb{C}_{0,\alpha}) * \mathbb{C}_{1,rgb} + \mathbb{C}_{0,rgb} \tag{3.1}$$

$$\mathbb{C}_{1,\alpha} = \alpha + (1 - \mathbb{C}_{0,\alpha}) \tag{3.2}$$

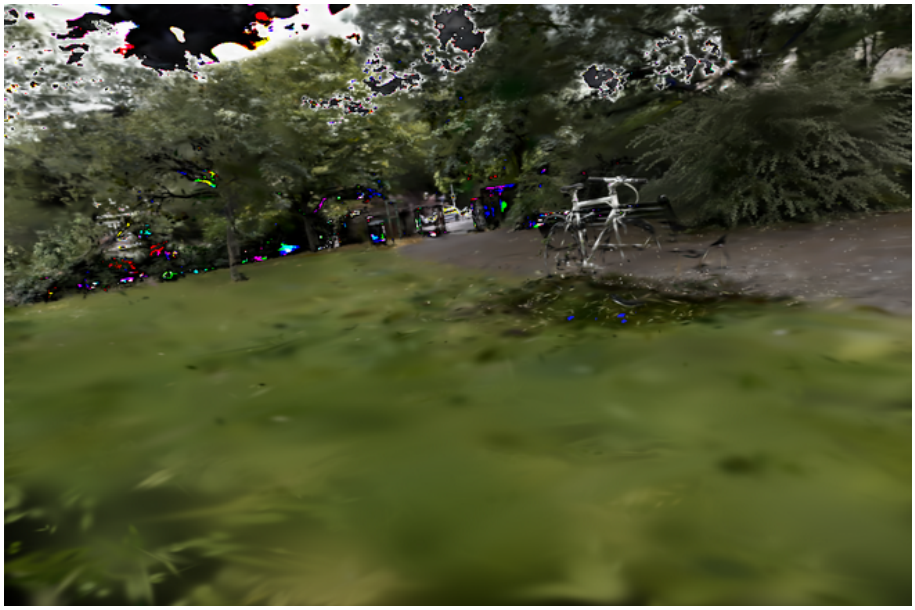

Figure 3.1: Render without depth sorting

3.3 Sorting

Splats are sorted based on their depth from the camera. Splats cannot be sorted before projecting them, as their depth is only known after the projection occurs, meaning they must be ordered at run time. The depth is relative to the camera, so as the camera moves about the depth value changes, and must be recalculated by re-projecting the splats, leading to the choice of sorting the splats each frame. There are far too many splats to sort the splats after render time, using a sort-last approach [21] - an approach where each splat gets rendered individually, and then composited based on sorting. Sorting them individually for each pixel would be wasteful, as the order is shared between each pixel, and the same sort would need to be redundantly performed for each pixel. Thus, the splats should be sorted first, and each pixel deals with pre-sorted splats, so splat contributions can be calculated in order, and blended before the next splat is calculated.

3.4 Tiling

Tiled rendering is the idea of splitting the rendering process into separate sections for different "tiles" on the screen. For the case of splatting rendering, tiled rendering means splitting the screen up into tiles, and only considering splats which are inside the corresponding tile for each pixel. Some splats will be considered for more than one tile, but most splats will not, meaning that the number of splats considered for each pixel is reduced by approximately the number of tiles. This reduces frame times for rendering. There are implementation specific trade-offs, discussed in the implementation section. Gaussians are unbounded, so each splat has some contribution to every point in space, and so will always contribute to every tile.

To solve this problem, a bound is calculated which contains almost all of the density of the Gaussian. This is calculated as a bounding box which contains at least three standard deviations of the Gaussian [22, 7]. There are two different standard deviations for a 2D Gaussian (what we have after projection), aligned to two axis. The larger value of the two can be used as the radius of a bounding circle which will contain both standard deviations. The bounding circle is actually enlarged to a bounding box to simplify calculations as the tiles will also be square, due to the shape of displays. The standard deviation can be calculated as the square root of the larger eigenvalue of the 2D Gaussian. The eigenvalues of a covariance matrix are the variances along their corresponding eigenvectors. Standard deviation is the square root of variance [23]. Three standard deviations is equivalent to 99.7% of the density of a Gaussian distribution, meaning that the contribution of a splat outside the bounding box is at most 0.3%, which is less than the resolution of an RGBA8 output, which has increments of $1/256$, or 3.9%. The 0.3% s derived from the error function $\mathrm{erf}(\frac{3}{\sqrt{2}}) = 0.997$ where three is the number of standard deviations and 99.7% is the amount of domain within those three standard deviations.

Chapter 4

Implementation

The process of rendering a splatting scene takes five stages in my implementation, outlined in **Fig. 4.1**. The pre-process, bin size calculation are defined each in a single compute shader, but the sorting is a several step process.

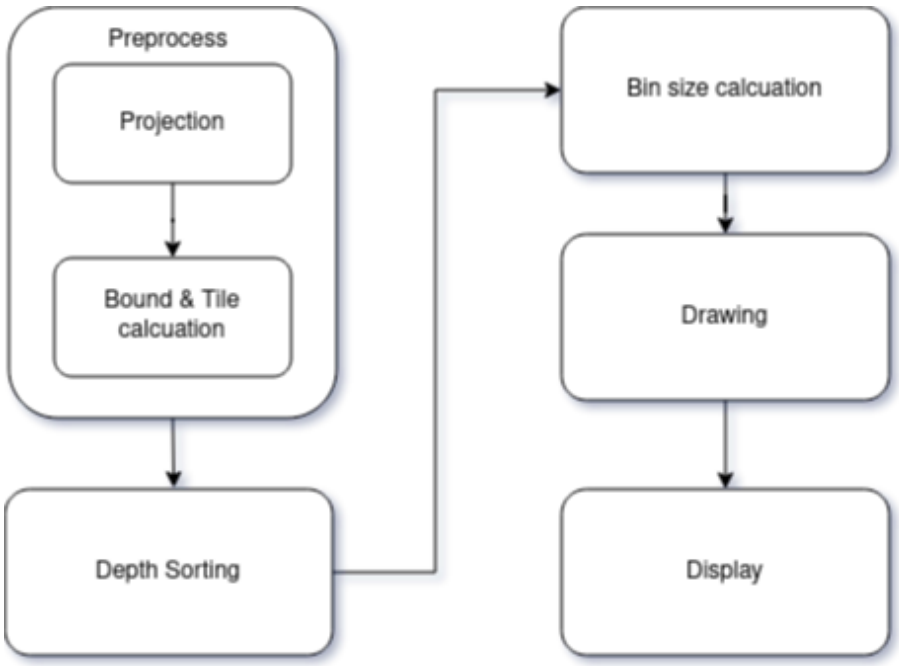

Figure 4.1: Pipeline stages of my Gaussian splatting renderer

4.1 Splats

There is not yet a standard file format for a Gaussian splat file, but the following data must be encoded for every splat:

- The mean (central location) of the splat
- The covariance matrix for the Gaussian distribution of the splat
- The opacity of the splat
- The colour of the splat

In my implementation, there is a buffer for each of these values. There is a buffer of three-vectors for the means, a buffer of floats for the opacities, a buffer of three-vectors for the colours, and a buffer of floats (interpreted as 6-vectors) for the covariance matrix. The covariance matrix is a three by three matrix, however it is symmetrical, meaning only six of the nine elements are stored in the buffer, and the full matrix gets extracted when needed.

4.2 Pre-processing

Each splat must be projected in the pre-processing stage. There are two separate values to project - the mean and the covariance matrix. The mean is transformed using a standard view projection matrix. **Equation 4.1** gives us the mean in clip space. It can then be converted to screen space using **Equation 4.2**. This gives us a three-vector representing the location of the mean, with the first two elements being the location on the screen, which are stored into a projected mean buffer. The third element is the depth, which is used later in this shader. $\boldsymbol{\mu}$ represents the mean of the splat. P represents the projection matrix, and V represents the view matrix. W represents the width of the screen in pixels, and H represents the height of the screen in pixels. Σ represents the 3D covariance matrix, and Σ' represents the integrated 2D covariance matrx.

$$\boldsymbol{\mu}_{clipspace} = P * V * \boldsymbol{\mu}_{worldspace} \tag{4.1}$$

$$\boldsymbol{\mu}_{screenspace} = \begin{pmatrix} \boldsymbol{\mu}_{clipspace,x} * W & \boldsymbol{\mu}_{clipspace,y} * H \end{pmatrix} \tag{4.2}$$

The covariance matrix is not a single point like the mean, and must be integrated to be converted to screen space. An exact integration is not possible, and instead a Taylor expansion approximation is used, using the first two terms [7, 22].

$$\Sigma' = J * V * \Sigma * V^T * J^T \tag{4.3}$$

where J is the Jacobian, defined as follows:

$$J = \begin{pmatrix} \frac{f_x}{t_z} & 0 & -\frac{f_x * t_x}{t_z^2} \\ 0 & \frac{f_y}{t_z} & -\frac{f_y * t_y}{t_z^2} \\ 0 & 0 & 0 \end{pmatrix} \tag{4.4}$$

Equation 4.4 and **Equation 4.3** are defined first by EWA splatting [7, 22]. Here f_x is the focal length of the camera along the x-axis, f_y is the focal length of the camera along the y-axis, and t is the projected mean.

The 2D covariance is a diagonal matrix. This means that it can be compressed from a three by three matrix to a three-vector as follows, where $\Sigma'_{\mathbf{d}}$ denotes the diagonal of the 2D covariance matrix:

$$\Sigma'_{\mathbf{d}} = \begin{pmatrix} \Sigma'_{xx} & \Sigma'_{xy} & \Sigma'_{yy} \end{pmatrix} \tag{4.5}$$

The alignment for a vec3 in an OpenGL buffer object is the same as the alignment for a vec4 in an OpenGL buffer object, both take up 16 bytes [24, 25]. This means that it will take the same memory to store a buffer of vec3s as a buffer of vec4s. To overcome this issue, the 2D covariance diagonals are stored in a vec4, with the first three elements being the diagonal, and the fourth element being the opacity of the splats, which is used together with the 2D covariance when the memory is later read. This means that reading the opacity now comes together with reading the 2D covariance at no extra memory cost.

The covariance is also normalised by dividing it by its determinant, so that it can be sampled later in pixel space.

$$\det(\Sigma') = \Sigma'_{\mathbf{d},x} * \Sigma'_{\mathbf{d},z} - {\Sigma'_{\mathbf{d},y}}^2 \tag{4.6}$$

The final stored value, Σ'_{stored} is

$$\Sigma'_{stored} = \begin{pmatrix} \frac{\Sigma'_{\mathbf{d},z}}{\det(\Sigma')} & -\frac{\Sigma'_{\mathbf{d},\mathbf{y}}}{\det(\Sigma')} & \frac{\Sigma'_{\mathbf{d},x}}{\det(\Sigma')} & \alpha \end{pmatrix} \tag{4.7}$$

where α is the base opacity of the splat.
The bounding box of the splat is then calculated with the eigenvalues of the 2D covariance matrix. *BL* represents the "bounding length" for the bounding box, or half the length of any side of the bounding box.

$$m = \frac{\Sigma'_{\mathbf{d},\mathbf{x}} + \Sigma'_{\mathbf{d},\mathbf{z}}}{2} \tag{4.8}$$

$$\lambda_1 = m + \sqrt{m^2 - \det(\Sigma')} \tag{4.9}$$

$$\lambda_2 = m - \sqrt{m^2 - \det(\Sigma')} \tag{4.10}$$

$$BL = 3 * \sqrt{\max(\lambda_1, \lambda_2)} \tag{4.11}$$

PM is the *projectedMean* and *BB* contains the four corners of the bounding box.

$$BB = \begin{pmatrix} (PM_x - BL, PM_y + BL) & (PM_x + BL, PM_y + BL) \\ (PM_x - BL, PM_y - BL) & (PM_x + BL, PM_y - BL) \end{pmatrix}$$

The tiles the bounding box overlaps with are then calculated. A key for sorting is generated that combines the tile index and the depth value, with the tile index contained in more significant bits than the depth value, as shown in **Fig. 4.2**. Some bounding boxes may overlap with more than one tile. These splats get duplicated so they can be rendered in all relevant tiles. They are duplicated by adding another key to the end of the key array, and adding an index to an index array which points to the correct splat location. The end of the array is tracked by an atomic counter, and the size of the array is capped as it is stored as a buffer. This means that it must be decided before rendering the maximum number of duplicates that will be needed. I chose to leave space for each splat to be duplicated an average of one time, or a buffer length of $2 * n$, where n is the number of splats. There are some viewing angles where the duplicate space becomes saturated, and rendering quality suffers, but these were rare in my experimentations.

There are trade-offs to using tiling. Using an atomic counter to track the end of the array is expensive. Additional memory must be used to store the duplicated splats, and there are up to double the number of keys to sort. For a GPU radix sort, the time complexity is $O(k * n)$, where n is the number of keys to sort, and k is the number of digits in each key. n may be doubled, and k is increased by eight for a case where 256 tiles are used. We go from k at 16, in the case where we encode the depth to 16 bits, to k as 24, or a $1.5*$ increase, and up to a $2*$ increase in n, resulting in up to $3*$ increase in sorting time. The number of tiles should be a power of two, so that it can fit in a discrete number of bits with no redundancies. The more tiles, the more

duplicates, but the fewer tiles, the more splats to consider for each pixel. The number of tiles appropriate for a scene can be discovered experimentally. In my case, it was 256.

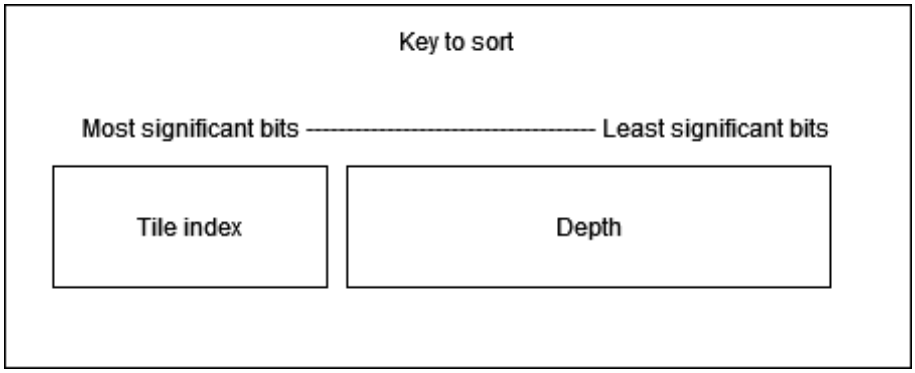

Figure 4.2: Breakdown of the components of the key

4.3 Sorting

After the pre-processing occurs, the sorting must take place. The buffer of keys that is generated is a buffer of floats, and there is an index buffer which are the splat indexes the keys correspond to. The index buffer gets sorted by the order of the key buffer. There is also an auxiliary buffer which is the same size as the index buffer. Between each step of sorting, the index buffer gets moved into the auxiliary buffer so the sorting doesn't need to occur in place. The auxiliary buffer then becomes the index buffer (by swapping by reference) for the next stage of sorting, until the sorting finishes.

Sorting can take place on the GPU or the CPU. Previous papers such as the one by Satish et al. [26] have examined the difference in speed between CPU and GPU sort implementations finding that their CPU radix sort was 20% faster than their GPU radix sort. Despite this, it still makes sense to use a GPU radix sort for the case of splatting. Millions of splats must be sorted, and the splats begin on the GPU. This means to sort on the CPU, the data must be transferred from the GPU to the CPU and then back. The time taken to 1,000,000 floats from the CPU to the GPU takes over 500ms on my target machine from experimentation. 500ms is significantly longer than the time to sort 1,000,000 splats on the GPU directly (82ms). If sorting the splats every frame, then the GPU cannot perform other tasks while the CPU sorts, so there isn't a benefit from the advantage of CPU-GPU asynchronicity. This means that the overhead of transferring the data from the CPU to the GPU makes any performance benefit of a CPU sort over a GPU sort redundant, so a GPU sort should be used.

There are multiple options of sorting algorithms on the GPU such as the bitonic sort [27], and the parallel radix sort [8]. Following the original paper on rendering splatting radiance fields [1], I implemented a parallel radix sort.

Radix sort relies on the principle of sorting on each "digit" in the numbers separately, while remaining stable between each digit's sort. The algorithm works well with using keys to sort,

which is needed. There are three separate sections, split into three different shaders, to a radix sort. The shader invocation flowchart can be viewed in **Fig. 4.3**. In order to effectively parallelise the algorithm, data is split into small sections to be handled by separate threads. Each thread first generates a local histogram for that section in the first step. Then, in the second step, the local histograms are prefixed, first for each thread's local offset within the bucket, and then once for the entire array. This means that the buffer to store the histograms in must be $16 * (N + 1)$ elements long, where N is the number of threads. This also means that there isn't necessarily a speedup if more threads are added, as each thread must have read/writes to memory for the local histograms. Finally, the third step calculates where to place each value, and inserts each value into the correct location in a secondary buffer. This is calculated for each value as the global prefix sum for the bucket, added to the local prefix sum for this thread. The local prefix sum for the bucket for the thread is then incremented.

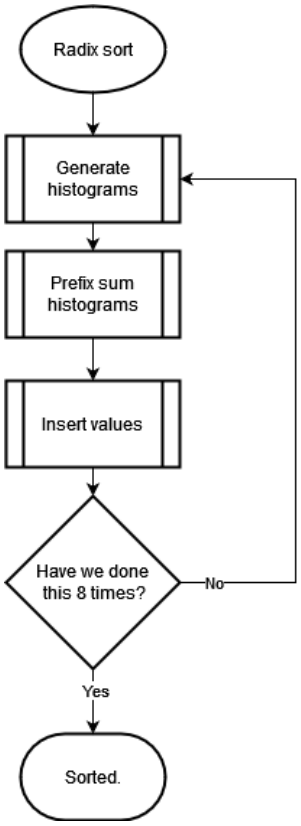

Figure 4.3: Radix sort stage overview

The secondary buffer and original buffer are then swapped by reference, as the secondary buffer contains the correct sorting to this point, and the values in the original buffer are redundant and can now be overwritten in a future stage.

This all results in a stable sort for the digit. When performed for each digit, the array becomes fully sorted, as desired.

There are additional optimisations to make. Where threads need to fetch several identical items of the same data from global memory, it is more efficient for them to distribute the fetching between the threads, utilising shared memory. In the third step, the global prefix sum is fetched into shared memory, one value for each thread, reducing 16 global reads for each thread to one global read, one shared write, and 16 shared reads, resulting in better performance. Another optimisation is to utilise parallel scan algorithms [28] for prefix summing where appropriate, which I use to calculate the global prefix sum.

It should also be noted that while it is possible to perform all these operations in a single shader invocation, there must be synchronisation of threads between steps. My chosen platform, OpenGL compute shaders, does not allow for synchronisation between work groups, so it was necessary to split the steps into separate shader invocations for this case unless I only wanted one work group, but it is not necessary for all cases.

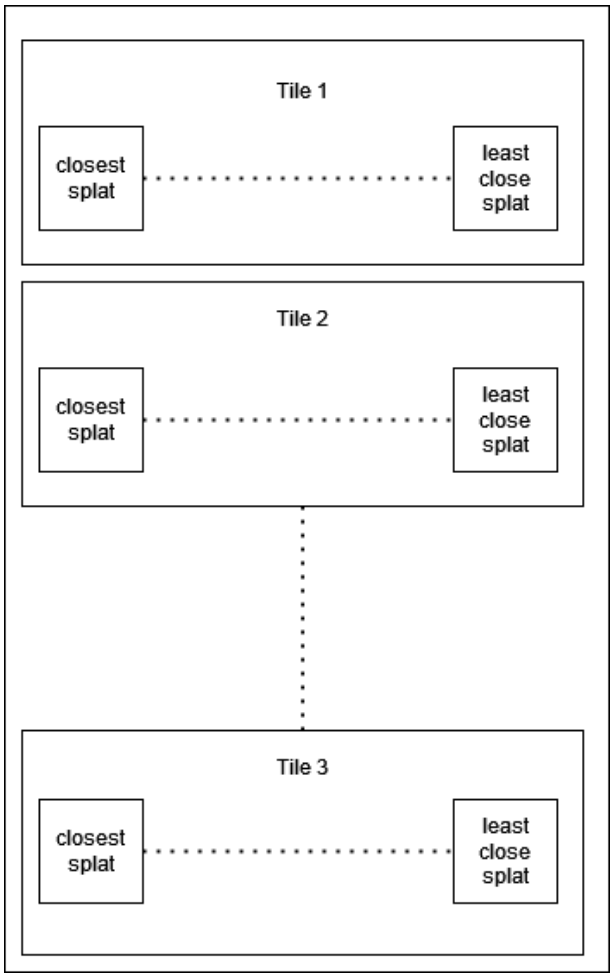

Figure 4.4: Demonstration of how sorting works with tiles

The way space in the sorted buffer corresponds to tiles is demonstrated in **Fig. 4.4**, where each tile's splats are internally sorted in a group for each splat. In order to extract the tile groups from the sorted splats, the sizes of each of the 256 tiles must be calculated, and then

using those sizes to work out the start and end index of each tile section. This corresponds to the *Bin size calculation* stage The algorithm for counting the size of each tile works by initialising an array of length 256 to 0, one element for each tile, then dispatching a thread for each splat, and atomic-incrementing the index for the tile that splat belongs to. Due to the large number of bins (256), the atomic operation is largely not blocking, however, optimisation may be possible by considering more than one splat per thread, or utilising shared memory. The end of each bin can then be calculated by performing a prefix sum over the tile histogram. My implementation utilises a parallel scan algorithm [28] to efficiently distribute the computation across 256 threads. Unfortunately, the drawing stage cannot be executed until this prefix sum is complete, so all other SIMD (Single Instruction Multiple Data) units but the 256 for this operation remain un-utilised during the computation. The start index of each tile is the end index of the previous tile, or 0 for the first tile.

4.4 Draw

Given a set of depth sorted, projected splats, there are multiple ways to rasterise them. I explored two different ways in my solution. Regardless, the same equations can be used to alpha blend, and the same equations to sample the splat. The alpha of the splat is calculated at a pixel using **Equation 4.13** [22].That α value and the colour of the splat *col* are then used to blend with the current colour of the pixel using **Equation 3.1** and **Equation 3.2**, described earlier in the methods section. There is also an early out, where rendering for a pixel is stopped when its alpha value is close to one, so more blending will have no change to the colour as any additional values to blend will be insignificant.
$\mathbf{p}$ represents the pixel position on the screen.

$$\mathbf{d} = \mathbf{p} - \boldsymbol{\mu}_{screenspace} \tag{4.12}$$

$$\alpha = \Sigma'_{stored,z} * e^{0.5 * (\Sigma'_{stored,x} * \mathbf{d}_x^2 + \Sigma'_{stored,z} * \mathbf{d}_y^2) - \Sigma'_{stored,y} * \mathbf{d}_x * \mathbf{d}_y} \tag{4.13}$$

The first way I explored was to rasterise the splats by drawing each splat in order. This meant for each splat, going through each pixel inside the bounding box, and adding that splat to the pixel. This method works quite efficiently in serial, as blending isn't needed for pixels which are outside of these bounding boxes. Unfortunately, each splat must be drawn sequentially due to the need for depth ordering, so drawing must be synchronised between each splat, so this is not an appropriate method for GPU parallel rendering. I used this way to implement a "ground truth" CPU renderer to compare my optimised GPU renderer with. There is no compromise on the quality of the renders when rendering in this method, it precisely renders the splatting scene. I wrote this renderer in C++ using vectors instead of GPU buffers, and did not make use of any parallel processes. Each iteration of the for loop would take the next closest splat, and calculate the contribution the splat made for each pixel in the bounding box, before blending with the pixel value if the contribution was not very small (and would not be visible at all).

The second way I explored was to rasterise the splats by drawing each pixel independently. This means going through each splat in order for every pixel. Parallelising this is trivial, as

each pixel is calculated independently and so can be computed in separate threads with no synchronisation. The draw back of this is that the bounding boxes aren't being fully taken advantage of, even with tiling, and there are thousands of global memory reads per thread.

I wrote this in one shader program, where each thread calculated the colour for one pixel, meaning there was one thread for every pixel. The thread iterated through each splat for the tile, calculating the contribution and blending. When the pixel became saturated ($\alpha > 0.99$), or there were no more splats to iterate over, the thread had finished and the pixel colour was written to a texture. To overcome the bottleneck of many global memory reads, I made use of work groups and shared memory. So long as each thread in a work group corresponds to a pixel in the same tile, the threads need the exact same splats. This means that reading the splats from global memory can be distributed, so that each thread reads one splat at a time to shared memory. The size of the work group is limited by the amount of shared memory available, as each thread needs to be able to read at least one splat. The resolution of the screen must also be chosen such that each work group resides in only one tile, so that the splats are shared. This optimisation resulted in a speedup of over five times for work groups sized 8x8, allowing for real time rendering for small scenes, and interactive rendering for larger scenes.

The *Draw* shader program is a compute shader, and writes to a texture but doesn't have the ability to display the texture to a screen. A *Display* shader program takes this texture and displays it on the screen. It works by rendering one screen-sized quad, which renders out the texture calculated in the *Draw* stage. The entire render pipeline has gone from projection to display without splat data being passed to or from the GPU to the CPU. There is some data that must be passed over, however. The camera location is updated on the CPU. There is a camera C++ class which tracks the cameras transformation. This transformation gets updated by keyboard inputs. The camera has functions to generate the view and projection matrices, which are called to pass as inputs to the pre-process shader. The camera also keeps track of the screen resolution.

4.5 Testing

There are many components to the project which all needed testing. Some of the components could be tested using automated testing, and those tests were written using the GTest framework. The sort was tested by comparing its output to a sort performed by the C++ standard library sort, implemented with GTest. The file loader was also tested using GTest, with a test created that compared the result of the file loader with a set of expected values.

Testing the rendering sections was a lot more difficult. A CPU renderer was implemented, which the different rendering sections could be integrated with. The GPU sections were non-deterministic, so the output couldn't be exactly compared with the CPU output, and automated testing wouldn't work. Instead, I could input a section at a time to the CPU renderer, and see if the results (visual image) were as expected. This is not as effective testing as

automated tests written with GTest, but there was no expected output to compare with. The outputs of each stage were also too abstract to directly inspect, which is why they were integrated with the CPU renderer to inspect the full rendered image. Inspecting the rendered image involved comparing with the output of the CPU renderer to check it matched, and working out what was causing discrepancies when they did not match.

Chapter 5

Results

5.1 Renders

(a) GPU render, with tiling and optimisations (650ms)

(b) CPU render, rendered per splat precisely, ground truth (4015ms)

Figure 5.1: GPU and CPU renders of a large scene (3,616,103 splats)

In **Fig. 5.1** and **Fig. 5.2** there are rendered images from two different scenes. I implemented a CPU renderer which renders an image precisely by drawing each splat one at a time, with no tiling or quality compromising optimisations, which I consider as a ground truth for rendering the splats. It can be seen for both of them the GPU renders are very similar to the ground truth. There are small artifacts visible in **Fig. 5.2a**. There are not similar artifacts visible in **Fig. 5.1a**. The artifacts are temporally consistent, and are most likely due to the different encoding used for the CPU and GPU render, not due to inaccuracies in the rendering. Otherwise, an almost perfectly matched image can be seen, with over six to eight times reduction in rendering time. This shows the optimisations in the GPU render did not sacrifice on the quality of the rendering, but still provided significant performance improvements to achieve interactive

rendering. For even smaller scenes than the bonsai tree **Fig. 5.2**, real time rendering can also be achieved, with 30FPS being reached when scenes are limited to 100,000 splats, and 60 FPS being reached when scenes are limited to 50,000 splats.

However, there is one case where slight compromises are visible. Exemplified in **Fig. 5.3**, when the number of duplicates required is more than the memory allocated to duplicate into, some splats are left un-rendered for some tiles. This results in some visible tile lines. In the case of **Fig. 5.3**, there is memory allocated to allow duplication of 272,956 splats, but there were more duplicates required. This problem can be overcome by a few methods. The first is to simply allocate more memory for duplicating splats than is required, which can be calculated through testing various viewpoints. The other is to reduce the number of duplicates required, which can be done by choosing scenes which don't have such large splats that span across so many tiles.

(a) GPU render, with tiling and optimisations (72ms)

(b) CPU render, rendered per splat precisely, ground truth (588ms)

Figure 5.2: GPU and CPU renders of a smaller scene (272,956 splats)

Figure 5.3: GPU render, with tiling visible due to duplication saturation

5.2 Timings

All timings were performed on the integrated graphics of an AMD Ryzen 5 pro 5650u APU and 16GB DDR4 RAM on Ubuntu at 1024x512 pixels

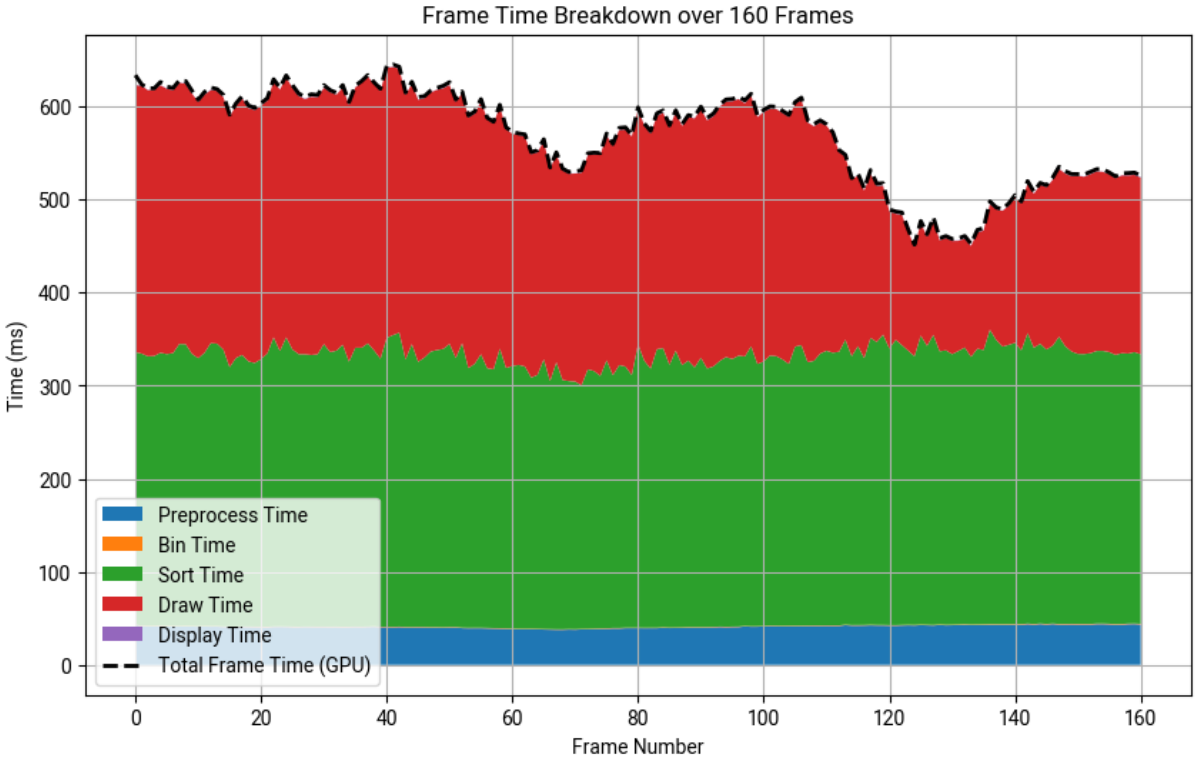

Figure 5.4: Frame time breakdowns for rendering a large scene (3,616,103 splats) across all frames as a stacked line graph

The chart in **Fig. 5.4** shows the frame time breakdown across 160 frames for a large scene. As the frame times were captured, the camera was moved all around the scene, rendering from various viewpoints. It can be seen in **Fig. 5.4** that as the camera moves around, the frame time varies, however the only component that varies is the draw time. It is expected that the draw time would vary as the camera moves around, as different view points will have different numbers of splats culled. Around frame 120, it is likely the viewpoint was moved to a position where less splats were in view. Unfortunately, the other stages cannot take advantage of the culling and so remain relatively stable across all frames.

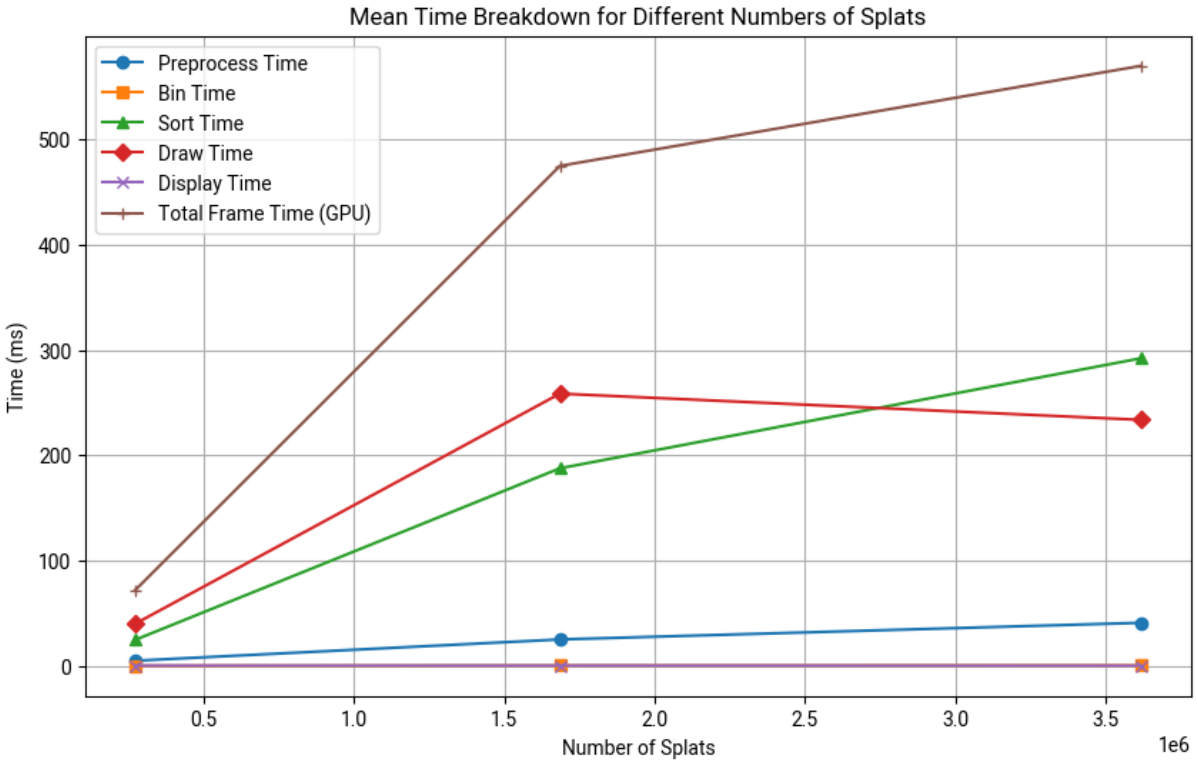

Figure 5.5: Mean frame times for three different scenes with varying numbers of splats.

The most significant stages are the draw and sort, while the bin and display time take an insignificant amount of frame time. The computation in the draw stage is less complex than the computation in the pre-process stage, but the draw stage takes over 5x the frame time. This is because each thread in the draw stage has large numbers of memory reads, which take far longer than the complex maths calculations in the pre-process stage. The sort time's contribution is the expected time to sort millions of keys [26].

Fig. 5.5 shows how the contributions to frame time, and overall frame time, change as the number of splats change. From algorithmic analysis it is expected for the overall rendering to scale with $O(N)$, and the significant components, pre-processing, sorting and drawing, to also scale with $O(N)$, where N is the number of splats. From **Fig. 5.5**, this appears to be true for the pre-processing and sorting stage, which grow linearly between the three scenes. However, the draw stage does not follow a linear growth, it actually takes less time in the largest scene than the second largest scene. This can be explained by a larger amount of culling occurring in the largest scene, which leads to less data to process for the draw stage. The sort stage not following the draw stage in being faster indicates that it is not due to more resources being free on the machine at the time of measurement, and the sort stage is where the culled splats are discarded, so it wouldn't be expected to see any improvement due to culling in the sort stage.

5.3 Reaching real time rendering

5.3.1 Alternate rendering

Figure 5.6: Debug render displaying the number of splats contributing to each pixel. Fully white means 255 or more splats contribute to that pixel. Fully black mean there are no splats contributing to that pixel. The maximum contribution was 271.

It is discussed earlier how tiled rendering can improve frame times by reducing the number of splats considered per pixel. However, there are major drawbacks. When a pixel is outside of the bounds for a splat, there is no need to render it, as it will not be visible on a computer screen. For the large scene, each tile contained 10000 splats, so each pixel was considering that many. It can be seen from **Fig. 5.6** that most pixels have very few splats contributing to them, and the pixel with the highest number only had 271. Considering at least 50x as many splats as needed is not ideal, and alternate methods of rendering should be considered. This consideration of too many splats is somewhat negated by the use of work groups to store splats in shared memory, but there is still a big performance hit. Rendering each splat by drawing to the pixels within its bound is a method that does not result in this massive overdraw. Unfortunately, this is not easily parallelised, as discussed earlier, due to the need to render the splats in order, but each pixel within the bounds for an individual splat can be parallelised. The overhead of calling millions of shader invocations to draw each splat is too much to consider this method with my chosen framework, OpenGL compute shaders, but with an alternate framework where shaders can be invoked directly on the GPU or queued it is worth considering this alternate algorithm.

5.3.2 Taking less time to sort

The time to sort is a significant portion of the rendering time, particularly for larger scenes, as shown in **Fig. 5.4**. In order to achieve real time rendering, this contribution to frame time must be reduced. The chosen sorting algorithm, parallel radix sort, is of time complexity $O(K * N)$

[8]. There are alternate implementations of a parallel radix sort such as the one-sweep [26], which can provide some speedup, but I will focus on how to improve the sorting time for this specific case, rather than improve the sorting algorithm itself.

N is the number of splats to sort, so reducing the number of splats to sort will reduce to the time to sort linearly. In my implementation, no culling occurs before the sort, with the sort moving the culled splats to the end of the array in its process. An implementation which performed culling before the sort stage would reduce the number of splats that needed to be sorted. Timings I took with sample data showed that in practice the sort scales slightly less than linearly. The sort takes 82ms for 1,000,000 splats, 38ms for 500,000 splats, and 16ms for 250,000 splats. This shows that there is an even greater benefit than expected to reducing the number of splats to sort.

K is the number of digits that make up the numbers. In my implementation I sort on a full 32 bit float. The keys can be encoded to fewer bits than this. Eight bits of the number are needed to encode the 256 tiles. The remaining data to encode is the depth. The precision needed to encode the depth is not exactly known, and depends on the quality of output desired. The bottom right tile of renders (indexed 255), has the depth stored in 16 bits, as there are only 16 bits available in a float32 to store numbers between 255 and 256, so its clearly possible to encode the entire key in 24 bits (six 4-bit digits) or less, with eight bits for tile index and 16 bits for the depth. If the key can be encoded into fewer bits, then there will be a linear speedup of the sort. The algorithm takes 82ms to sort 1,000,000 splats with eight 4-bit digits, 68ms with six 4-bit digits, and 42ms with four 4-bit digits. This is the expected linear speedup.

The previous ways to improve sorting still allow for precise sorting each frame. However this is not always needed. With a tiled rendering technique tiles must be sorted into each frame, but this is only a quarter of the sorting time. Additionally, there are alternate rendering methods which do not use tiling but still require the sorting. While rendering with no sorting can have undesirable results, such as **Fig. 3.1**, rendering with an almost sorted list can produce acceptable results. Many implementations such as [29] distribute the sort across many frames, and have acceptable results. My radix sort implementation consists of 24 shader invocations, which could be all distributed across different frames with little overhead. This means I could reduce the frame time of sorting to 1/24 of the current time, with the sacrifice of precision of the sort. This precision is unimportant when the view changes very little, as the depths from the camera do not change too much, however if the view changes significantly between sorts then there will be visible artifacts as the sorting will not be correct for that view point at all [30].

Further research should be carried out to determine how small the keys for sorting can be compressed, how often sorting must occur for reasonably visual quality. The radix sort is a sorting algorithm that sorts an array exactly, however if sorting doesn't need to occur every frame, that means that visual quality is acceptable when splats are mostly sorted, but not entirely. This means it is worth investigating using approximate sorting algorithms that will run

faster than the radix sort to achieve faster frame times.

5.3.3 Rendering with fewer splats, and compressing splats.

Rendering scenes with fewer splats leads to much faster frame times, as evidenced by **Fig. 5.5**. Even with the smallest scenes tested, the frame times in a renderer such as mine which doesn't compromise on precise rendering are not real time. Faster times can be reached without compromising by simply rendering even less splats in the first place. There have been methods created to create these smaller scenes such as [31], which can compress a 1,400,000 splat scene to 500,000 splats, which would result in almost a 3x increase in FPS. Additionally, a large amount of the frame time from the Draw shader comes from fetching the splat data from memory. Methods which allowed nearby splats to share colour data, or other data, could help reduce the time taken to draw, as the data would only be fetched once for multiple splats. Further research should be undertaken to explore whether rendering with fewer splats and compressing splats can result in better frame times, and how best to compress splats for rendering speed. If splats can be significantly compressed and reduced, the memory impact will be reduced, along with noticeably faster frame times, making splatting a more viable option for software like video games which have tight memory and frame time budgets, and allow lower spec devices to run splatting scenes.

5.4 Integrating with traditional rendering

Gaussian splatting can work together with traditional geometry rendering. It is possible to calculate an approximate depth buffer for a splatting render, which can then be used to combine with a traditional render to create a hybrid render. If the splatting model was generated with the same lighting as is present inside the scene, they may combine quite well. I believe the best use case for splatting together with traditional rendering is to replace the high frequency parts of a scene which geometry cannot capture well due to aliasing [20]. However, due to the nature of splats being unbounded they do not suffer from edge aliasing. Additionally, when generated using the method outlined in the original paper [1], they do not suffer from high frequency aliasing when rendered from the distance of the input images. There are also methods of rendering which result in alias free rendering from all viewing positions, which integrate across each pixel to render an exact image [17]. One element of scenes that often suffers from aliasing effects is foliage.

I believe it is worth researching using splatting to render high quality foliage in real time, together with traditional rendering for the less high frequency sections. A pipeline to render splats together with geometry could make use of geometric bounding of splats in the scene to replace the tiling method I used. A splatting scene or object could be divided up into small boxes in world space. If splats were bounded into small enough boxes where each box contained few enough splats that one pixel could manage them, then there would be no need for tiling. This would keep the benefits of tiling, without the need to sort into tiles every frame, and without the overhead of duplicating splats. Splats could also be culled easily before the sorting stage, as splats in boxes which are not visible on screen could simply be discarded.

Performing the draw stage in the fragment shader of the surfaces of the boxes would mean that the fast GPU rasterising hardware would manage what splats need to be considered by what pixels. Such a method would eliminate tiling while keeping the benefits and not the drawbacks, allow for better and easier culling, reduce sorting time as no tiling sorting is needed, and culling occurs before the sorting stage, and simplify managing the splats relevant to each pixel by running it in the fragment shader for the box.

Chapter 6

Conclusion

The literature review outlines the research that led to Gaussian splatting for radiance fields, by going through the history of the two main ideas it combines - splatting and light fields. This gives good context for what the Gaussian splatting scenes are representing, and why the renderer works as it does.

The project was successful in its goal of rendering Gaussian splatting scenes interactively with camera controls. Scenes were rendered with sub-second frame times, and with little compromise on the rendering quality. The pipeline created for rendering was described in detail. Evaluation of the implementation considered the limitations of it, and how other methods may provide improvements to reach real time rendering, suggesting some future research that could be done, including a proposal for a method to implement with traditional rendering. The breadth of options to improve the rendering time show that it is likely possible to implement an effective real-time Gaussian renderer, but the suggestions of these optimisations came from evaluation of my own renderer, so could not be taken advantage of without another large iteration of software development, which would be outside the scope of the project. Optimisations that did not require investigation were made and outlined in the implementation, bringing the renderer from offline timings (several seconds per frame), to interactive frame rates (less than one second per frame). All together, this means that I have implemented a program which takes keyboard inputs, and renders and displays the view from a virtual camera in a Gaussian Splatting scene at an interactive frame rate, utilising GPU acceleration and other optimisations.

References

[1] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. *ACM Transactions on Graphics*, 42:1–14, 07 2023. doi: 10.1145/3592433, [Online]. Available: https://arxiv.org/pdf/2308.04079.pdf (Accessed 2024-03-15).

[2] A. Gershun. The Light Field. *Journal of Mathematics and Physics*, 18:51–151, 04 1939. doi: 10.1002/sapm193918151.

[3] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. NeRF. *Communications of the ACM*, 65:99–106, 01 2022. doi: 10.1145/3503250.

[4] Alex Yu, Sara Fridovich-Keil, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance Fields without Neural Networks. *arXiv:2112.05131 [cs]*, 12 2021, [Online]. Available: https://arxiv.org/abs/2112.05131 (Accessed 2024-03-15).

[5] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques - SIGGRAPH '96*, 1996. doi: 10.1145/237170.237200, [Online]. Available: https://dl.acm.org/doi/10.1145/237170.237200 (Accessed 2024-03-19).

[6] Marc Levoy and Pat Hanrahan. Light field rendering. *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques - SIGGRAPH '96*, 1996. doi: 10.1145/237170.237199, [Online]. Available: https://graphics.stanford.edu/papers/light/light-lores-corrected.pdf (Accessed 2024-03-19).

[7] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. EWA splatting. *IEEE Transactions on Visualization and Computer Graphics*, 8:223–238, 07 2002. doi: 10.1109/tvcg.2002.1021576.

[8] Takahiro Harada and Lee Howes. Introduction to GPU Radix Sort, 2011, [Online]. Available: https://gpuopen.com/download/publications/Introduction_to_GPU_Radix_Sort.pdf (Accessed 2024-03-15).

[9] LearnOpenGL - Blending, [Online]. Available: https://learnopengl.com/Advanced-OpenGL/Blending (Accessed 2024-04-02).

[10] Mohammed Suhail, Carlos Esteves, Leonid Sigal, and Ameesh Makadia. Light Field Neural Rendering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–13, 01 2023. doi: 10.1109/tpami.2023.3316992.

[11] Jonathan T Barron, Ben Mildenhall, Matthew Tancik, Peter Hedman, Ricardo Martin-Brualla, and Pratul P Srinivasan. Mip-NeRF: A Multiscale Representation for

Anti-Aliasing Neural Radiance Fields. *International Conference on Computer Vision*, 10 2021. doi: 10.1109/iccv48922.2021.00580.

[12] Lee Westover. Interactive volume rendering. 01 1989. doi: 10.1145/329129.329138.

[13] Lee Alan Westover. Splatting: a parallel, feed-forward volume rendering algorithm. 07 1991, [Online]. Available: https://www.cs.unc.edu/techreports/91-029.pdf (Accessed 2024-04-20).

[14] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. EWA volume splatting. In *Proceedings Visualization, 2001. VIS '01.*, pages 29–538, 2001. doi: 10.1109/VISUAL.2001.964490.

[15] Carl Friedrich , Charles Henry Davis, and Gerstein University of Toronto. *Theory of the motion of the heavenly bodies moving about the sun in conic sections. A translation of Gauss's "Theoria motus," with an appendix*. Boston Little, Brown, 1857, [Online]. Available: https://archive.org/details/theoryofmotionof00gausuoft/ (Accessed 2024-04-26).

[16] M Jordan, J Kleinberg, and B Schölkopf. Information Science and Statistics, 2006, [Online]. Available: https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf (Accessed 2024-04-24).

[17] Zhihao Liang, Qi Zhang, Wenbo Hu, Ying Feng, Lei Zhu, and Kui Jia. Analytic-Splatting: Anti-Aliased 3D Gaussian Splatting via Analytic Integration. 2024, [Online]. Available: https://arxiv.org/pdf/2403.11056 (Accessed 2024-04-01).

[18] Christoph Lassner and Michael Zollhöfer. Pulsar: Efficient Sphere-based Neural Rendering, 12 2020, [Online]. Available: https://arxiv.org/abs/2004.07484 (Accessed 2024-04-26).

[19] Tomas Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michael Iwanicki, and Sébastien Hillaire. *Real-time rendering*. Crc Press, Taylor Francis Group, 2018. ISBN 9781138627000.

[20] Franklin C. Crow. The aliasing problem in computer-generated shaded images. *Communications of the ACM*, 20:799–805, 11 1977. doi: 10.1145/359863.359869.

[21] S Molnar, Maria Christina Cox, David S Ellsworth, and Harald Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14:23–32, 07 1994. doi: 10.1109/38.291528.

[22] Vickie Ye and Angjoo Kanazawa. Mathematical Supplement for the gsplat Library, 2023, [Online]. Available: https://arxiv.org/pdf/2312.02121.pdf (Accessed 2024-03-15).

[23] Roger A Horn and Charles R Johnson. *Matrix Analysis*. Cambridge University Press, 10 2012. ISBN 0521548233.

[24] Khronos. Buffer Object - OpenGL Wiki, [Online]. Available: https://www.khronos.org/opengl/wiki/Buffer_Object#Alignment (Accessed 2024-04-12).

[25] Interface Block (GLSL) - OpenGL Wiki, [Online]. Available: https://www.khronos.org/opengl/wiki/Interface_Block_(GLSL)#Memory_layout (Accessed 2024-04-22).

[26] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on CPUs and GPUs. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 06 2010. doi: 10.1145/1807167.1807207.

[27] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proceedings of the VLDB Endowment*, 1:1313–1324, 08 2008. doi: 10.14778/1454159.1454171, [Online]. Available: https://www.vldb.org/pvldb/vol1/1454171.pdf (Accessed 2023-11-05).

[28] Hubert Nguyen and Nvidia Corporation. *GPU gems 3*. Addison-Wesley, 2008, [Online]. Available: https://developer.nvidia.com/gpugems/gpugems3/ (Accessed 2024-04-26).

[29] Dylan Ebert. Gsplat Editor - a Hugging Face Space by dylanebert, 12 2023, [Online]. Available: https://huggingface.co/spaces/dylanebert/gsplat-editor (Accessed 2024-04-12).

[30] HubschmanHarold and W ZuckerSteven. Frame-to-frame coherence and the hidden surface computation. *Computer graphics*, 15:45–54, 08 1981. doi: 10.1145/965161.806788.

[31] Zhiwen Fan, Kevin Wang, Kairun Wen, Zehao Zhu, Dejia Xu, and Zhangyang Wang. LightGaussian: Unbounded 3D Gaussian Compression with 15x Reduction and 200+ FPS, 02 2024, [Online]. Available: https://arxiv.org/abs/2311.17245 (Accessed 2024-03-15).

Appendix A

Self-appraisal

A.1 Critical self-evaluation

I am overall pleased with my project. I met my main goal of creating an interactive splatting renderer. A successful splatting renderer was created, and I achieved interactive frame rates on low spec hardware. During my project, I had to change my goals, as I had initially over scoped my project, as I underestimated the difficulty of debugging a project like this. The outputs of each stage being seemingly abstract data stored on the GPU, in very large quantities, meant I struggled to work out what was wrong when things didn't work. Eventually this was overcome by rewriting sections from scratch incrementally so I could ensure they were working.

I made the choice to begin my project by implementing the GPU radix sort. All of the code for the project was written from my understanding of papers and tutorials online, and there are more resources for implementing a radix sort than implementing a splatting renderer. This made it a good starting point, as it was easier to learn how to implement, but still taught me how to understand algorithms from technical papers to transfer to software. It was also a great learning source for how to write the compute shaders, as it was far easier to debug and the processes were less abstract to me. Later, I did come back to the radix sort and improve it, when I knew exactly how it would integrate with my full renderer.

Achieving a working renderer took a lot of time, but by re-scoping my project and originally beginning programming early, I still managed my time well and did not find myself overwhelmed with work for the programming section. I did find I didn't leave as much time to write the report as I would have liked. I was able to write the full report by the deadline without having to compromise on the quality of the report, but I found I was spending a larger proportion of my time in the final two weeks before submission writing the report than I would have liked. To add more depth to the research, and ensure I properly backed my claims with high quality references, I spent more time than I expected reading around the subject, which I didn't mind as I found the subject really interesting, but for example reading a 100 page book set me back about a day.

A.2 Personal reflection and lessons learned

I am very pleased with what I've learned through this project. Not having a way to evaluate if my project was working, and not having an iterative process was a big mistake at the start. After I fixed those errors, things got a lot easier, and I will begin projects with small goals and methods of evaluation in future. I learned a lot about GPU programming through the project. I had used compute shaders for a couple of projects before, but never as effectively as I did

here. I learned how to better make use of the GPU hardware, and also learned good algorithms for parallel programming which I can use again in future, such as the radix sort. It was a good, if tedious, experience debugging such a complex project (once I had a method to do so), and certain things which caused issues with my project may come up again, and I'll be able to spot them much faster.
Through both my research and implementation, I gained a much greater understanding of how radiance fields and Gaussian splatting work. I understand much better what the scenes are actually representing, how the rendering works, and why it runs at the speed it does. I presented my project to prospective students on an open day, and found I could answer every question they had about my project and Gaussian splatting, so I am confident I have gained a good understanding of how it all works.
In future, I will aim to have a better understanding of what my goals involve before I begin to undertake them, so I don't over scope like I did with this project

A.3 Legal, social, ethical and professional issues

A.3.1 Legal issues

The models used for the project were not generated by me, but sourced from the internet. The datasets to train the models are open source, as are the models used by me. I do not distribute the models I used with my source code regardless. Open source code that is not written by me was modified and used to generate tests for the file reading, but the code is distributed with the MIT license. I include the MIT license in my own distribution as required.
External libraries used (GTest, GLEW, GLFW, OpenGL) are open source, and aren't included in my repository.

A.3.2 Social issues

Systems such as Gaussian splatting make it easier to acquire a 3D scene from photographs for non-technical individuals. Scenes could be created which violate an individuals privacy, for example if a scene was created using a video of somebody's home without their consent. A license could be included if the software was distributed that forbids malicious use cases.

A.3.3 Ethical issues

Gaussian splatting rendering is very resource intensive. When more resources are utilised on a computer, more power is drained. If Gaussian splatting was commonly adopted in games or another popular use case, it would mean these software generate more energy than they likely would otherwise. With sufficient users, this extra energy usage could have a genuine impact on the environment. Responsible distribution of software that utilised Gaussian splatting could use money generated from sales to invest in more green energy, so the extra energy used by Gaussian splatting comes from renewable, non-harmful sources.

A.3.4 Professional issues

The project was developed following good software engineering principles. The code is written to be readable, with descriptive variable names and comments. The development was tracked throughout using Git. Licenses have been followed appropriately.

Appendix B

External Material

The images rendered by my software used models I did not create. There were two models used for images in the write up - the bike scene, and the bonsai scene.
Bonsai: `https://huggingface.co/datasets/dylanebert/3dgs/tree/main/bonsai/point_cloud/iteration_7000`
Bike: `https://huggingface.co/datasets/dylanebert/3dgs/tree/main/bicycle/point_cloud/iteration_7000`
All .cpp, .h, and .glsl files are my own code. Python files for generating test data use open source code not written by me. The Python files contain a mix of my own and open source code. The open source code can be found here:
`https://github.com/limacv/GaussianSplattingViewer/blob/main/util_gau.py`.