Contents

1 Introduction

2 Related Works

2.1 Light Fields and Novel View Synthesis

In the fields of computer graphics and computer vision, novel view synthesis refers to the problem of, given a relatively small set of model images from different camera positions, generating an image representing the model from a point of view different from the input model images. The main advantage of this approach is to avoid the computationally expensive tasks of creating a 3D model, texturing, and rendering. In traditional computer vision, this task was performed by manipulating the input images through techniques such as flow-based interpolation and mosaic compositing. The main drawbacks of these techniques are the large amount of input images necessary to obtain quality results, and the need for significant overlap between model images [**?**].

One idealized concept in vision that would allow the representation of a scene from any point and any orientation is the plenoptic function. A three-dimensional color lightfield is defined by the 6-dimensional plenoptic function $P(\theta, \phi, \lambda, x, y, z)$. This function P denotes the light intensity with wavelength λ passing through the point (x, y, z) through a ray direction parameterized by the spherical coordinates (θ, ϕ) [**?**]. If the plenoptic function corresponding to an environment is known at all points and viewing directions, then the task of generating a novel view becomes as trivial as performing an angular integration at all camera pixel positions over all incident rays [**?**].

However, being an idealized concept, it cannot be completely specified for a natural scene as it would require the measurement of light intensity at infinite points in space from infinite directions, which is impossible in practice. However, multiple views of a model can help build an approximation of a discretized plenoptic function, and it hints to its ability to be used as an implicit representation of an environment, which will be later leveraged by Neural Radiance Fields to encode the illumination of a scene.

2.2 Structure from Motion

Before diving into an analysis of modern approaches to model reconstruction from 2D images, it is worth going over the Structure from Motion (SfM) algorithm, which serves as a starting point for all of the methods I will discuss later. The main problem this algorithm solves is inferring the 3D structure and motion of objects from the 2D transformation of their projected images when no other spatial information is given. The algorithm is based on the "structure from motion" theorem, which states that, given 3 orthographic projections, the structure of 4 non-coplanar points in space can be recovered [**?**]. For modern applications, the problem actually becomes retrieving 3D information about a scene from a set of unordered 2D images. COLMAP [**?**, **?**] is a pipeline implementing the SfM algorithm for this purpose and is used as an incipient step by all environment reconstruction models that I will present in the later sections.

The COLMAP algorithm is divided into two stages: Correspondence Search and Incremental Reconstruction. The **Correspondence Search** step takes as input the set of unordered images and outputs a set of geometrically-verified image pairs and a graph of image correspondences. First, for each image, a set of geometrically invariant features is generated, which will serve as a basis for finding correspondences between images in the initial set. Then, by leveraging these feature descriptors, the algorithm finds images that see the same parts of the scene. This first mapping is only computed based on appearances, so there is no guarantee that the features actually map the same scene point. In order to verify this match, the algorithm tries to find a homography that maps the features between the two images. If enough features match after applying the transformation to one of the image planes, then the image correspondence is added to the scene graph. The second stage, **Incremental Reconstruction**, takes as input the scene graph and outputs estimated poses for each image and a point cloud reconstruction of the scene. SfM initializes the model from a two-view reconstruction, which has to be carefully selected, as it can heavily influence the quality of the result. Then, in an iterative manner, more images can be added to the model. The criterion for selection is that a newly registered image must observe existing points in the model. This allows to determine camera parameters relative to the existing model, and also to triangulate the positions of the new feature points the image adds to the model. However, new points can be triangulated only when they are seen from two distinct perspectives. Even though registration and

triangulation are highly correlated, incremental errors from both processes result in the reconstruction drifting, so a bundle adjustment step is necessary. This is a non-linear refinement of camera parameters and feature point parameters in order to minimize the reprojection error. An overview of the complete pipeline can be seen in figure 1.

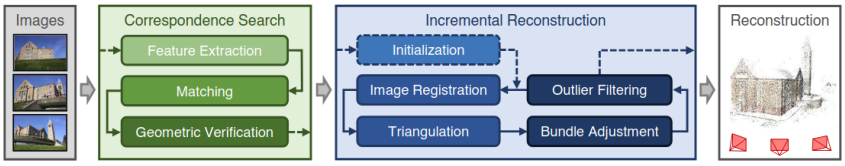

Figure 1: Complete SfM pipeline. Taken from [**?**].

After this pipeline is executed, each input image will have associated a pose and camera intrinsics which approximate the relative position of the camera where the image was taken. Moreover, a reconstructed point cloud is provided, which can serve as a starting point for environment reconstruction algorithms.

2.3 Neural Radiance Fields

The first NeRF model was published in 2020, and since then a lot of variations have emerged in an attempt to address some of its issues and improve its performance metrics. I will first go over the initial implementation, as it serves as a base for its follow-ups and illustrates the fundamentals of neural scene encodings.

2.3.1 NeRF

The main idea behind NeRF is to encode a static scene using a fully connected deep neural model [**?**]. The input of the model is a five-dimensional vector representing a position in space $\mathbf{x}$ and a viewing direction $\mathbf{d}$ and outputs the volume density at that point σ and the view-dependent radiance $\mathbf{c}$. This is quite similar to the formulation of the plenoptic function of a light field, however, the neural model can only provide an approximation of it through the following function $F_\Theta : (\mathbf{x}, \mathbf{d}) \rightarrow (\mathbf{c}, \sigma)$. Generating a novel view is done by querying 5D points along camera rays and accumulating the density and emitted color, just like any volumetric renderer. Since this raymarching method for rendering the images is fully differentiable, gradient descent can be used to optimize the model by minimizing the difference between the reference images and the renders obtained from querying the model. An overview of the training process can be seen in figure 2.

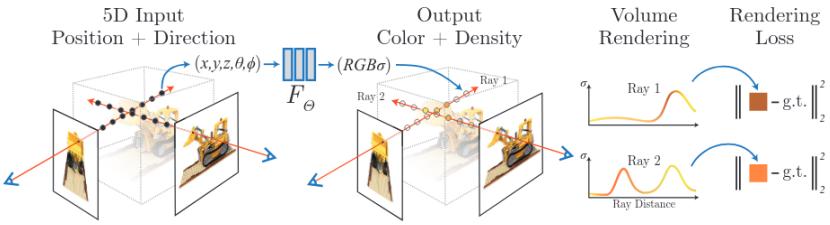

Figure 2: NeRF training process, from [**?**].

Even though the neural network is used as a generic function approximator, it tends to bias towards favoring lower frequency information. To address this issue, the input query points are first mapped into a higher dimensional space using higher frequency functions, thus creating a positional encoding, which promotes the learning of high-frequency features. In order to optimize the ray marching component, two models are trained for the same scene, one "coarse" and one "fine". The coarse model is sampled first to determine regions that require more sampling in order to allocate more sampling points to areas that are expected to have more impact on the final render. Using these strategies, they were able to surpass in terms of quality existing volumetric scene reconstruction implementations.

2.3.2 Mip-NeRF

An immediate follow-up to the original implementation of NeRF is Mip-NeRF [**?**]. It addresses the multi-resolution issues in NeRF, where a model can be rendered with high quality only when the scale is the same as that of the training images. This happens because of the ray sampling strategy, where rays are evaluated at individual points, so as the model gets further away from the camera, the volume gets more sparsely sampled and aliasing artifacts start to appear. Also, for synthetic scenes, the reference cameras are all at the same distance from the model, so a single-resolution model cannot solve a multi-resolution problem efficiently. The solution to this issue is inspired by the mipmapping algorithm for texture sampling, where lower-resolution textures are prefiltered, and then can be interpolated between levels to obtain the desired resolution. To achieve this effect, the ray sampling is changed from point sampling to volumetric sampling through integrated positional encodings. Computing and sampling a cone around a ray is computationally expensive, so around each sample point, a multivariate Gaussian is fitted in order to approximate the sampling volume. Figure 3 shows the difference in model sampling between the reference NeRF implementation and Mip-NeRF. As the sample point gets further away from the camera, the sampled volume gets bigger. This can be achieved by shrinking the high-frequency positional encodings, thus sampling the lower-frequency features, which achieves the same effect as prefiltering. The advantage of this approach is that the quality of the high-resolution renders remains completely unaffected, while lower-resolution renders become more photorealistic. This allows zooming into models without losing quality and also seamlessly transitioning between different levels of detail.

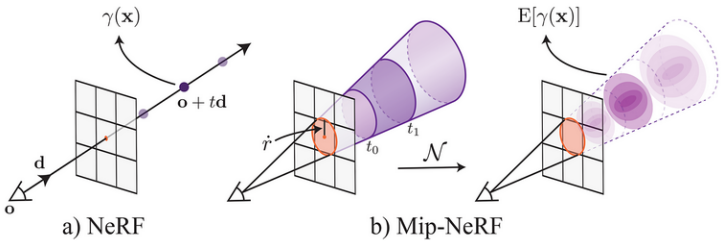

Figure 3: Differences in model sampling between NeRF (left) and MipNeRF (right), from [**?**].

2.3.3 FastNeRF

Besides addressing the image quality, some implementations try to improve the rendering speed of the original NeRF, which is far from being able to be used in real-time rendering. By building a cache structure of the radiance map represented by the model and querying it instead of the neural network, FastNeRF achieves a roughly 3000x increase in rendering performance without sacrificing quality [**?**]. The main idea of the cache is to take as input the position and orientation vectors and produce the estimated density and illumination values in a roughly constant time. However, building a cache for a 5-dimensional input is very taxing in terms of required space, and even moderate resolutions would require unreasonable amounts of storage space. In order to avoid the issues of high polynomial increase of storage requirements with resolution, the authors propose a separation of the model into a position cache and an orientation cache. A simplified representation of the architecture can be seen in figure 4.

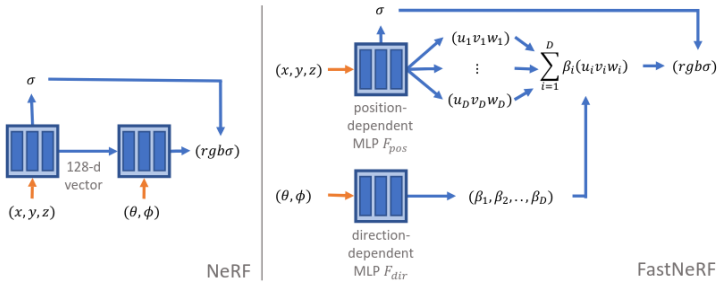

Figure 4: Reference NeRF model (left) and FastNeRF split architecture (right) [**?**].

The estimated density is only a function of position, but the radiance depends on both position and viewing direction. To overcome this parameter coupling, the position-queried model produces a set of deep radiance maps for each point in space, and the direction-based model produces a set of weights corresponding to each of the deep irradiance maps. This approach is similar to the use of spherical harmonics to estimate directional illumination. By taking the dot product between the deep irradiance map and the weight vector, the local irradiance can be estimated while also taking into account the viewing direction. By building two separate caches following this strategy, the required storage space can be reduced to reasonable values. Also, by adjusting the dimension of the cache, the tradeoff between quality and storage can be balanced depending on the application. However, the main drawback of this implementation is that in order to achieve comparable quality metrics to the original NeRF, the model size is drastically increased, but in turn, enables the use of NeRF for real-time scenarios.

2.3.4 Mip-NeRF360

All of the NeRF methods presented above are trained and evaluated on scenes containing one central model and a black background. This is because these models are confined to a bounded volume for their representation and cannot fit large scenes efficiently, as the background could be very distant. Moreover, the disparity between the detail in the foreground and the background introduces floating artifacts in the scene, as the representation becomes ambiguous in areas that are seen by few camera positions. The architecture of Mip-NeRF360 proposes a solution to this issue [**?**]. To address the first problem of the scene being unbounded, the authors propose a reparameterization of the coordinate vector through a strategy similar to an Extended Kalman filter. In this approach, the coordinates inside the unit sphere remain unchanged, and the rest of the coordinates outside the unit sphere are remapped to the sphere of radius 2. This means that also the volumetric Gaussians used for sampling but Mip-NeRF will get distorted as the scene elements get further away from the center point of the scene, as can be seen in figure 5.

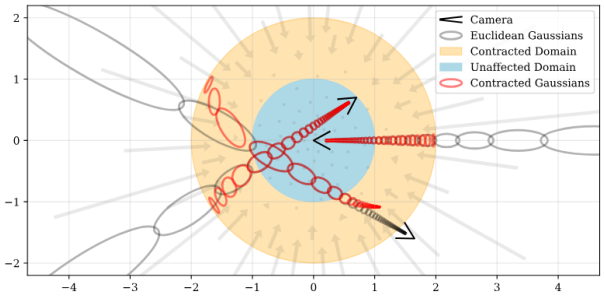

Figure 5: Mip-NeRF360 scene compression [**?**].

The original Mip-NeRF runs both coarse and fine evaluations along the rays using the same MLP, where zones of interest are determined by the coarse sampling and they determine where the finer sampling should be performed. However, this is wasteful in terms of computation time, since only the density is needed in the coarse sampling. This new architecture introduces a second MLP which acts as a proposal model, and which only outputs a density distribution along the ray in order to determine the finer sampling of the actual NeRF MLP. This can be thought of as an online distillation method since both models are trained together. The proposal model is not trained directly, as it is only constrained that the density histogram it emits is consistent with the histogram of the NeRF MLP since they represent the density distribution along the same ray. The last improvement proposed by the authors for this model is the introduction of a regularization step which is applied to the weighted density distribution along each ray. In short, the regularizer is trying to minimize the total distance between pairs of sequential points along the ray, as shown in figure 6. This minimum can be achieved only when the weights are 0, which means that the ray would be empty. However, in the case of non-empty rays, it is minimized by consolidating the weights into a region as small as possible. This in turn has the effect of removing floating artifacts, which are introduced by distant elements of the scene by effectively "pulling" them towards their correct position.

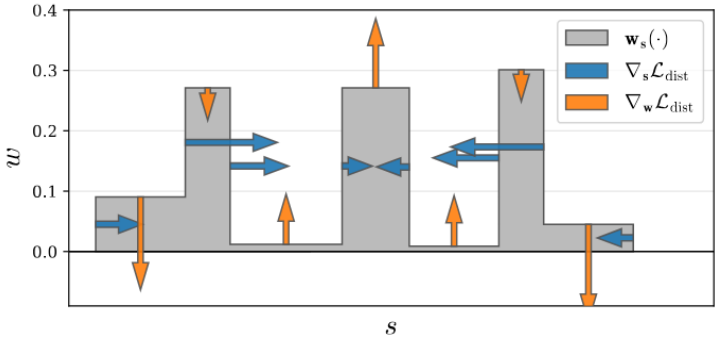

Figure 6: Regularization function and its gradients [**?**].

Using these 3 improvements, Mip-NeRF360 can efficiently model unbounded scenes and greatly surpasses the models preceding it in terms of visual quality.

2.4 Plenoxels

Plenoxels is a photorealistic view synthesis system that adopts the idea of differentiable rendering from NeRF, but it attempts to encode a scene representation without an MLP [**?**]. Instead, this model uses a sparse 3D grid of voxels called plenoptic volume elements, which store spherical harmonic information and density on their vertices. Then, the color and density at any arbitrary point can be determined by trilinear interpolation of the values at the corners of the voxel containing said point. This allows for simple rendering based on raycasting, similar to any other volumetric renderer. Using this interpolation approach, the model can define a continuous plenoptic function throughout the whole model. During model optimization, the spherical harmonic coefficients and opacities are updated with respect to the mean square error between the rendered image and the reference images, as well as using a total variation regularization term. Optimization is done in a coarse-to-fine manner. Going over the coarse voxel grid, unnecessary voxels are pruned and the voxels in detailed areas are subdivided, then the optimization is performed on the finer grid. The total variation term in the cost function has the purpose of removing high-frequency noise in the reconstruction and is balanced with the image quality metric through the regularization weight λ_{TV}. Figure 7 shows an overview of the plenoxels scene optimization pipeline.

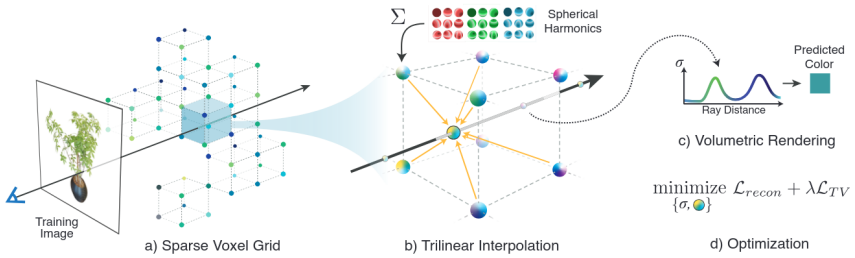

Figure 7: Plenoxels model optimization pipeline [**?**].

This model can achieve up to 100x improvement in training time compared to typical NeRF implementations while maintaining similar image quality metrics, and with minimal changes can be adapted to unbounded scenes. For real unbounded scenes, the sparse volume is surrounded by a set of background spheres, which in turn represent sparse voxel grids that will model the background, interpolating both within each sphere and between adjacent spheres.

2.5 Splatting for Volume Rendering

Traditional volumetric rendering techniques such as ray casting, which is the primary rendering method in NeRF-like models, offer very good quality as they evaluate the path of multiple rays through the volume

and accumulate light information in a realistic manner, they are very compute-intensive and sometimes not fit for real-time applications. The first approach of accelerating volume rendering through a forward-mapping algorithm was proposed by Westover L. in 1989 [**?**]. In order to accelerate rendering, the initial volume is sampled along a regular grid at the desired resolution. These samples are then considered particles that emit or absorb light and influence the final image. Line integrals are computed across each particle to determine its footprint on the camera plane. This removes the issue of raycasting, as now instead of determining which sample each pixel "sees", the problem becomes determining which pixels each sample influences, which reduces the complexity since volume samples are considered to be simple volumetric primitives. Reconstructing a continuous signal from discrete signals is done by convolution with a reconstruction kernel. For band-limited signals, a perfect reconstruction can be achieved using the **sinc** function as a convolution kernel. However, the volume samples have a limited volumetric span, and the sampling frequency in the initial grid is dictated by performance requirements so it may not follow the Nyquist-Shannon sampling theorem, so the samples are convoluted with discs whose properties vary in a Gaussian manner. After the image signal is reconstructed from the discrete samples, the color on each pixel is blended using a back-to-forward or forward-to-back traversal over the samples that influence it. This way, rendering volumetric data can be performed significantly faster, albeit at a cost to image quality.

Even though it went through numerous changes, splatting has become popular recently as an alternative to NeRFs, where the scenes are represented explicitly through Gaussian primitives. Even though the current implementation follows the general direction of the original, it is no longer a method for approximating known volumetric data for the purpose of rendering, but the splats themselves are the base representation of the data. In the following subchapters, I will go over the original implementation of Gaussian splatting for novel view synthesis, as well as some variations of it that are relevant to my work.

2.5.1 3D Gaussian Splatting for Real-Time Radiance Field Rendering

Radiance Field methods have brought many advancements to environment reconstruction by encoding scenes through a Multi-Layer Perceptron network. However, these methods sacrifice rendering speed for quality, especially in complex or unbounded scenes, as continuously evaluating the neural network during rendering significantly limits frame times. However, an alternative to this is brought by 3D Gaussian Splatting models [**?**], which represent the scene explicitly through Gaussian primitives, which allows for achieving real-time rendering speeds. This implementation provides a methodology of using 3D Gaussians to represent a continuous radiance field, optimizing the scene, and rendering the scenes using a differentiable visibility-aware process that allows it to achieve real-time frame times.

Just like the NeRF models, this implementation also starts with processing the input images through a Structure from Motion pipeline. This will output a set of calibrated camera positions, and also a point cloud of the scene features, which is also used by this algorithm, as each point in this initial cloud will initialize a Gaussian at its center. However, these Gaussian primitives are not only defined by their center, but also by a 3x3 covariance matrix Σ, an opacity α, and a set of spherical harmonic coefficients which compose the color taking into account camera direction to model specular properties and possible reflections.

The next step of this process is iterative optimization, which is applied to all the properties of the Gaussians in the scene. All the primitives are initialized with an isotropic covariance with axes equal to the average distance to the three closest points. During optimization, the splats are projected to the screen and alpha blending is used to determine the final pixel color. The exact process of projecting the covariance matrix will be detailed in a later portion of this document. The loss function that is optimized is a weighted combination of the $\mathcal{L}_1$ norm of the image difference and the structural dissimilarity metric $\mathcal{L}_{D-SSIM}$. Gradients for all parameters are derived explicitly to avoid the overhead of automatic differentiation, so the adjustments needed for all parameters can be easily computed. The use of anisotropic covariance matrices allows splats to model a variety of features and is especially useful for thin and long scene components, where using isotropic covariances would require significantly more primitives. The complete scene optimization pipeline is shown in figure 8.

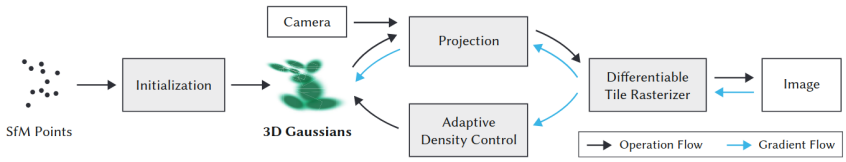

Figure 8: Overview of the 3DGS scene optimization [**?**].

However, even though anisotropic splats are flexible in terms of the features they can model, the Gaussians fitted to the initial point cloud from SfM are usually not enough to get a good representation of the scene. This is why the authors propose a method for the adaptive control of gaussians which allows the control of the number of splats in the scene, as well as their spatial density. This mechanism identifies under-reconstructed or over-reconstructed regions through the presence of large view-space positional gradients since those areas are lacking in quality and the optimizer tries to move gaussians there to increase detail. In the case of under-reconstruction, where a single Gaussian fails to reconstruct a feature and new geometry is needed, the Gaussian is cloned and the clone is moved in the direction of the positional gradient. For over-reconstruction, usually, a Gaussian has become too big trying to cover a geometric feature, but the feature would be better modeled by a set of smaller Gaussians. In this case, the volume should be preserved but the number of entities has to increase, so the initial Gaussian is split into two smaller ones that cover the same space. In both cases, additional optimization steps are necessary to ensure that the adaptive control, has the desired effect. The disadvantage of this mechanism is that it can produce floating artifacts close to the camera, so this issue has to be addressed by pruning splats with very low opacities periodically. The two densification cases can be seen in figure 9.

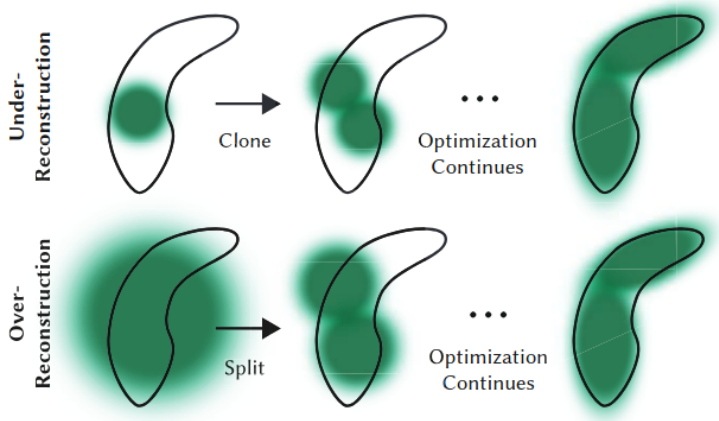

Figure 9: Cases for Gaussian densification [**?**].

This implementation achieves image quality metrics close to state-of-the-art, while also using a significantly faster rendering process. Additionally, since rendering is also a key part of the optimization loop, this means that the scene training times are in turn reduced compared to neural-based radiance field scene representations.

2.5.2 Gaussian Pruning and Compression

The quality and rendering speed advantages of representing scenes using 3D Gaussians come with the drawback of high requirements for storage space, which is a few orders of magnitude higher than that of neural representations. LightGaussian is a method that addresses this issue through Gaussian pruning, knowledge distillation, and vector quantization of Gaussian properties in order to significantly reduce the storage requirements of these scenes with minimal impact on the rendering quality [**?**].

The standard 3DGS optimization process tends to produce dense scenes with many redundant Gaussians, which negatively impact both storage and rendering speed. Taking inspiration from neural network

pruning, which removes neurons that have a low impact on the output, this method tries to identify Gaussians with a minimal contribution to the rendered images and removes them during training. Using a simplistic criterion for identifying insignificant splats, such as opacity, results in a quick degradation of the images and the loss of fine details. The authors propose a global significance score derived from the splat volume, opacity, and the number of pixels influenced over all training views. The volume is normalized by the largest 90% of all Gaussians, otherwise, the large splats making up the background would get an exaggerated importance score. After applying the pruning process, the remaining splats will continue to be optimized, but the adaptive densification is disabled, so the number of Gaussians will not increase again.

The second strategy for reducing the size is lowering the number of spherical harmonic coefficients. In a full representation, these make up 81.3% of the stored data. Removing them completely would decrease image quality, as they encode specular details, but in many cases, the full set of coefficients is not needed to encode all the available information. To balance model size and quality, the authors propose a knowledge distillation process, where high-degree SHs transfer the information to a lower-degree representation. The supervision of this training step is based on the difference in predicted pixel values between the two models. To increase the robustness of this process, the SH models are also sampled from synthetically generated pseudo-views, placed around the original camera positions, and following a normal distribution.

The last step proposed in this method is the Vector Quantization of spherical harmonic coefficients. This procedure is based on the assumption that a subgroup of Gaussians will exhibit a similar appearance, so they can be represented by a single encoding. After choosing the amount of desired entries in the codebook, k-means is used to create a mapping between Gaussians and their codebook entries, based on Euclidean distance in the SH vector space. Then, the codebook entries are refined without changing the mapping in order to increase visual quality. To ensure that significant details are not lost, Gaussians with a high precomputed visual importance score will not be compressed. Figure 10 shows an overview of the methods implemented in the LightGaussian solution.

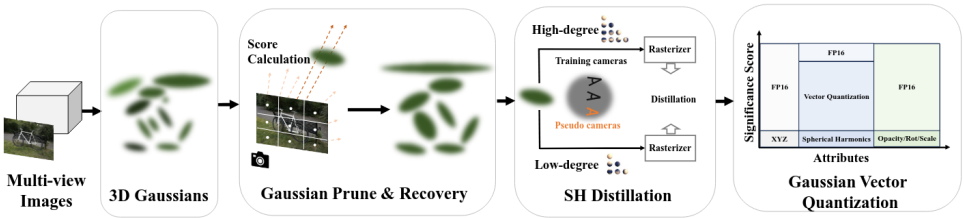

Figure 10: Methods implemented by LightGaussian for primitive pruning and feature compression [**?**].

Using this approach, the authors achieve an approximated 15x reduction in storage space and over 75% increase in FPS, while maintaining a reduction in quality under 0.4 dB PSNR. Note that the increase in rendering speed is only a result of the scene containing fewer Gaussians, as no changes have been made to optimize the renderer.

Another publication on Gaussian scene compression achieves even better results, especially in the rendering time improvement by combining the pruning and compression with a modified rendering pipeline [**?**]. Unlike the previous implementation, this method does not imply a reduction in the number of Gaussians in the scene. The first step is a vector quantization of the SH coefficients and shape parameters (i. e. scale and rotation).

Determining the quantized vectors is done by K-means clustering on the color and shape parameters separately. However, instead of using a simple Euclidean distance metric for clustering, the authors introduce a sensitivity metric. The sensitivity of a parameter is described as the variation in image energy when a small change to the parameter is applied, summed over all the training images. This computes the gradient of image energy with respect to all parameters of all Gaussians, and it can be performed in one single backward pass. When clustering, the Euclidean distance is multiplied by the parameter's sensitivity, thus artificially pulling apart features of high sensitivity, and giving lower importance to features with low sensitivity. For color information, the top 5% of splats in terms of sensitivity contribute

most to the image, so they will not be clustered. The rest will be clustered using vector quantization. In the case of shape parameters, the clustering is done on the covariance matrices, which are afterward decomposed in a scale and a rotation matrix. Over the tested scenes, an average of 15% of Gaussians present zero sensitivity, which means they have no contribution to the rendered image, so can in turn be pruned.

Since quantizing parameter values comes with a cost to image quality, the scene goes through an additional stage of fine-tuning. However, instead of optimizing the individual Gaussians' parameters, the gradients are accumulated per codebook entry, and after each iteration, the quantized entries in the SH and shape codebooks are updated instead. Moreover, all Gaussian parameters except the position are quantized further to 8-bit values using the Min-Max scheme, except for position, which shows severe degradation if quantized with fewer than 16 bits.

The last step of the compression strategy is to take advantage of the spatial coherency of reconstructed scenes, where Gaussians in close proximity to one another are expected to have similar, or even the same, properties. By ordering the Gaussians according to a Z-order curve in Morton order, the performance of the LZ77 run-length encoding can be improved. Since the Gaussian properties have been quantized, the final compression also uses Huffman coding to take advantage of the lower entropy of the information.

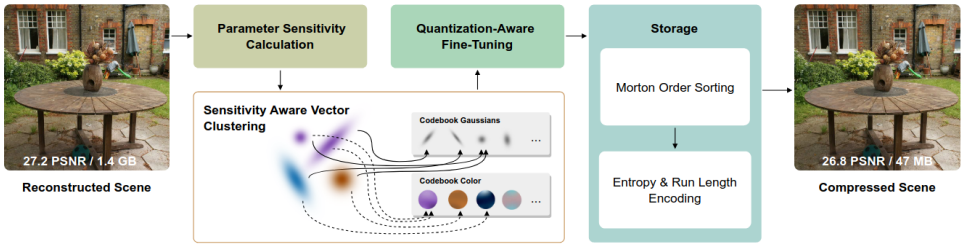

Figure 11: Gaussian compression pipeline [**?**].

Besides the compression strategy shown in figure 11, this method also proposes a modified rendering pipeline. The first step remains the same as in the original 3DGS implementation, projecting the Gaussian into splats on screen, eliminating splats that are not visible, computing the color from the SH coefficients, and ordering them by depth. However, instead of using a tiling software rasterizer, this implementation uses the traditional GPU rendering pipeline. For each splat on the screen, a quad made up of two triangles is instantiated. The vertex shader computes for each splat the vertex position, such as the quad covers the 99% confidence area of the Gaussian. Then, it outputs to the fragment shader the solid splat color and the Gaussian's center. Then, the pixel shader uses the distance from the splat center to compute the exponential color and opacity falloff and perform the blending into the framebuffer.

Using this strategy, the method achieves an average 26x compression over all scenes with an average quality loss of 0.26dB PSNR. The rasterization pipeline also sees a 4x improvement in speed. Approximately a 2x increase in rendering performance can be attributed to the lower bandwidth requirements of the compressed splat representation, and the additional improvement is a result of using the more efficient hardware rasterization pipeline, instead of using a software rasterizer.

Another attempt at making Gaussian scenes smaller uses a learnable approach to both pruning and color encoding [**?**]. The number of Gaussians in the scene is controlled by a learnable mask. Instead of waiting for the entire training process to end before pruning, this method eliminates splats based on a volume mask after each densification step. The learnable mask is based on the volume and opacity of Gaussians, since these two metrics define a Gaussian's expected contribution to the rendered image. The balance between the eliminated Gaussians and the rendering quality is maintained by introducing an additional masking loss term in the optimization function. The advantage of this masking procedure is that it also reduces the number of primitives during training, resulting in lower memory requirements compared to the original 3DGS. Encoding the geometric properties of scale and rotation quaternions is done using residual vector quantization, where the number of cascading stages is chosen to balance performance and quality.

Instead of encoding the color information in a similar codebook, this implementation uses a hash grid followed by a small multi-layer perceptron model to estimate color from the viewing direction. The Gaussian center is fed as input to the hash grid, then the resulting features and the camera viewing direction are used as input to the MLP to get the color estimate. Of course, the unbounded coordinates of the scene have to be bounded first using a technique similar to Mip-NeRF360. This implicit representation allows for very good compression since the SH coefficients take up most of the storage space.

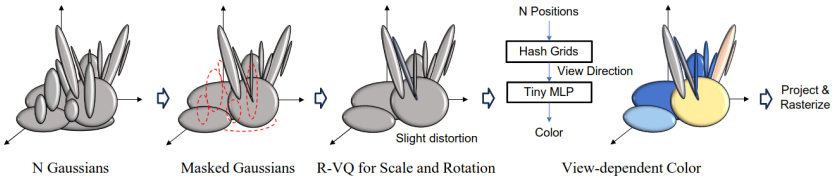

Figure 12: Details of the compact Gaussian architecture [**?**].

Using this masking and encoding approach shown in figure 12, this method achieves up to 22x storage compression, but usually without a loss in quality, and many times with an increase in observed PSNR compared to the reference implementation, hinting that the masking process might also be beneficial for reconstruction quality, not only for space optimization.

Scaffold-GS introduces a hybrid scene representation by implicit encoding of Gaussian properties through an MLP and an explicit representation of feature anchor points in the scene for Gaussian distribution [**?**]. The initial point cloud produced by COLMAP is used to produce a sparse grid of anchor points, where each anchor tethers a set of neural Gaussians with learnable properties. This approach leverages the structural information given by the point cloud by allowing the Gaussians to only optimize locally, thus reducing drift and "floater" artifacts.

For each anchor point, k neural Gaussians are spawned, each being defined by a 3D offset from the anchor point location. Each anchor is also assigned a feature bank of 32 components and a scaling factor. A set of very small MLPs is then optimized to estimate opacity, color, quaternions, and scales for each of the k spawned Gaussians based on the viewing direction, camera distance, and the feature bank specific to each anchor. The offsets and scaling factors for each anchor are also learnable parameters. Only visible anchor points are evaluated through the MLP, thus reducing the overhead. It is worth mentioning that, even though there are separate MLPs for estimating the different properties, these are global to the scene, and not instantiated per anchor point.

The point cloud from SfM gives a good starting point for creating the anchor grid cells. However, some areas of the scene need more detail than can be provided by the set of k Gaussians that can be produced by a single anchor. To determine such cases, neural Gaussian gradients for each cell are accumulated over multiple training iterations, and if they exceed a set threshold, the cell will go through a densification process. This involves spawning new anchor points in a multi-resolution grid based on the initial scaffold cells. To regulate this densification process, trivial Gaussians are identified by their opacity. If an anchor cannot produce Gaussians with opacity high enough, the respective anchor is pruned from the structure.

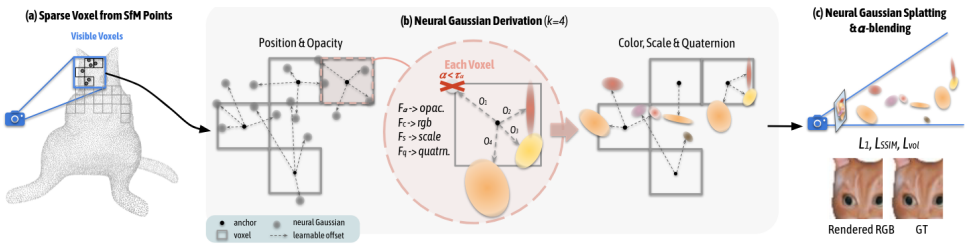

Figure 13: Scaffold-GS optimization pipeline [**?**].

This implementation, also shown in figure 13 achieves a slight increase in rendering quality and speed

compared to the reference 3DGS, and a reduction between 3.8x to 10.2x in required memory thanks to the implicit representation of Gaussian parameters. Additionally, it showcases better view adaptability to camera positions outside of the training poses.

2.5.3 Gaussian Splatting Anti-Aliasing

While a lot of research goes into optimizing the storage size and rendering speeds of Gaussian models, some implementations focus more on the image quality aspects of this kind of environment representation. One clear issue is the lack of regularization with respect to the sampling frequency during training, which leads to aliasing artifacts and high-frequency noise [**?**]. The MipSplatting implementation aims to address these issues through two separate mechanisms. Their approach is mainly based on the Nyquist-Shannon sampling theorem [**?**], which states that in order to correctly reconstruct a continuous signal from discrete samples without losing information, the original signal has to be band-limited, and the sampling frequency should be at least twice the maximum frequency of the continuous signal. In the case of 3DGS scenes, the spatial sampling frequency is given by the camera intrinsics, as well as its position relative to the scene. This means that Gaussians are sampled differently depending on the view, and the reconstruction does not account for views outside the training camera positions. In the reference implementation, projected splats that are thinner than one pixel are dilated by an arbitrarily chosen kernel, which ensures that all visible splats have a contribution on screen. However, this leads the optimizer to favor the creation of thin Gaussians and underestimates their real scale. This works for the training images but leads to erosion and dilation artifacts when the camera moves closer or further away from the scene, or the sensor resolution changes. Figure 14 illustrates these kinds of artifacts.

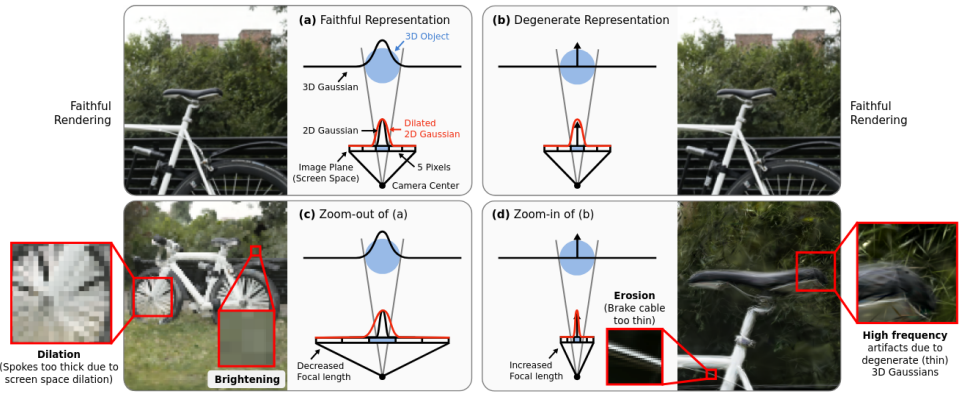

Figure 14: Dilation and erosion artifacts appear when the sampling frequency differs from the training cameras [**?**].

The first change the authors propose to mitigate these issues is regularizing the Gaussian spatial frequency during training. The first step is to determine the maximal sampling frequency of each Gaussian, taking into account all training images, distance to the camera, and focal length. This process is done at set intervals during training. Then, for each Gaussian, a 3D low-pass filter is applied, with the cutoff frequency set at the maximal sampling frequency for that Gaussian. This operation is done before projecting to screen-space, by the convolution of the initial Gaussian with the Gaussian low-pass filter. Using this filter solves the issue of high-frequency artifacts in reference images, but aliasing still appears when rendering the scene in lower resolution.

The second modification comes in the form of replacing the dilation mechanism in the final pre-processing stage with a 2D Mip filter. This is based on the physical principle of a camera capturing light, where the photons are integrated over the area of a pixel. A good estimation of this process would be applying a 2D box filter in image space, but this would require filtering all projected splats after they are rasterized, but before they are blended into the framebuffer to produce the final image. A more efficient way to achieve a similar effect is to a 2D Gaussian filter to each splat before rasterization, as this process only involves operating on the covariance matrix.

Applying the steps above to the training and rendering routines respectively, this implementation achieves slightly improved quality metrics when generating full-resolution images, but it retains more detail and achieves an increase of around 1 to 2 dB PSNR when decreasing the sampling resolution.

Another proposed solution to the issues above is to create a multi-scale representation of the scene, and select the Gaussians to be rendered based on the estimated sampling frequency [**?**]. The implementation creates a 4-tier representation, where each level is optimized at 1x, 4x, 16x, and 64x downsampled resolution respectively. The Gaussians at coarser levels are created by merging fine-level Gaussians, and at render-time, the selection for render is done by each primitive's pixel coverage. The pixel coverage metric is defined by the length, in pixels, of the shortest axis of the Gaussian.

The scene is initialized using the same strategy as the reference implementation and it goes through the same optimization process in the first phase, including the densification process. Then, the images are rendered at the downsampled resolutions, and the splats that fall below a predefined pixel coverage metric are marked for aggregation into the next level. The aggregation process is done by dividing the space into a grid sized according to the downsampling rate, and merging the Gaussians in each cell in order to form a new Gaussian for the next resolution level. The aggregation is done by average pooling of all the parameters that define the primitives. After all levels are generated, the optimization process continues using images at all resolutions mentioned above, in order to fit all the multi-resolution levels to the reference images. For each Gaussian, a minimum and maximum pixel coverage threshold is stored, which are then used in rendering to decide which Gaussians in the hierarchy should be passed for rasterization. Figure 15 shows an overview of this pipeline.

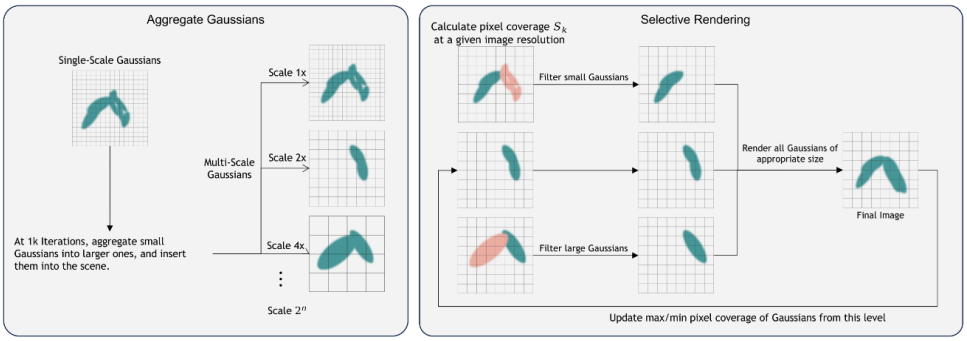

Figure 15: Overview of the training pipeline, including the multiscale aggregation of primitives and the selective rendering process [**?**].

Since not all primitives are small enough to be aggregated, the whole process increases the Gaussian count by an average of 5% in order to create the three additional resolution levels. The render quality benefits of this method increase as the render resolution decreases, offering an increase of between 13% and 66% in PSNR and 140% to 2400% improvement in rendering time. The improvement in rasterization speed comes from the implementation of the tiling rasterizer: at lower resolution, each tile overlaps more splats, so the same thread workgroup has to process more primitives. In the reference implementation, decreasing the rendering resolution or moving away from the scene in such a way that the scene occupies less space on the screen results in an increase in rendering times, which this implementation solves by reducing the number of primitives as they occupy less space on the screen.

2.5.4 Multi-resolution Gaussian Representations

As discussed before, Gaussian scenes have the disadvantage of large memory requirements when compared to neural representations. The research discussed above focuses on reducing this requirement for benchmark scenes, however, another problem is posed by the reconstruction of very large environments, such as cityscapes, which many times will not fit the memory limitations of even workstation-grade GPUs. To overcome this limitation, most implementations use a combination of space partitioning schemes and multi-resolution representation to enable training and rendering of these massive scenes.

GityGaussian [?] proposes an implementation based on a divide-and-conquer strategy and multiple levels of detail to facilitate training large-scale 3DGS environments. The scene is first partitioned into adjacent blocks that can be optimized in parallel, thus reducing the memory strain on each GPU. Individual block training, however, poses the issue of "floater" artifacts which try to represent the space outside the training block that is seen by the cameras. This leads to inaccurate representation and makes combining the blocks after training more difficult. This is why the initial point cloud produced by COLMAP is used to produce a scene prior of the entire environment, which is a coarse Gaussian representation of the model. Because the number of Gaussians remains relatively low, this training step can be done before the partitioning. Using a scene prior proved to produce much better reconstructions and allows for seamless merging of blocks. Because the scenes this method is aimed at are usually unbounded, a linear contraction of the space allows for a more even distribution of primitives inside blocks and avoids almost empty blocks. Then, for each block, the camera poses that capture that block need to be registered. To determine this registration, an SSIM loss is computed between the fully rendered image and the image rendered without that block, and if the loss exceeds some threshold, the block is considered to have a considerable contribution to that pose. Then, each block is trained individually from its respective camera poses, using the scene prior for rendering but only performing fine-tuning on the primitives inside the block. Then, the complete fine-tuned model can be obtained from the direct concatenation of the blocks. Figure 16 shows the block training process, including the partitioning and the scene prior.

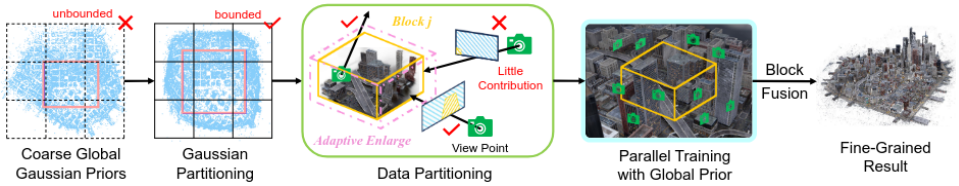

Figure 16: CityGS block optimization pipeline [?].

The level-of-detail component of this implementation uses the same mechanism as the one proposed in LightGaussian [?], in order to reduce the number of primitives successively across multiple detail levels. Because lover-detail levels are used for blocks further away from the camera, where high-frequency details become more insignificant, the decrease in quality is less noticeable. During rendering, the selection of which level to rasterize is done based on each block's distance to the camera, only for the blocks that intersect the frustum, as shown in figure 17. In practice, the extent of many blocks is artificially enlarged by floaters, so the authors propose an approach based on the Median Absolute Deviation algorithm [?] to compute block bounds more accurately.

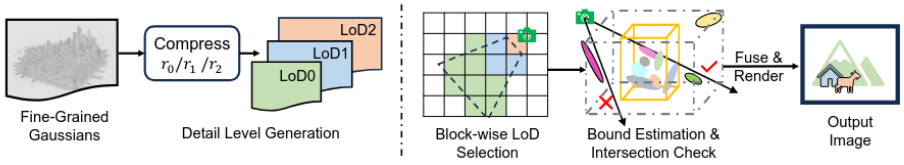

Figure 17: CityGS rendering process, including the selection of the appropriate LoD [?].

The tests were performed on the *MatrixCity* [?] dataset, which is a synthetic, large-scale collection of cityscape images. Compared to the reference 3DGS implementation, CityGaussian achieves an increase of 3.79 dB PSNR, as it is able to generate more Gaussian primitives for a better scene representation, and more than doubles the rendering speed through the use of the LoD structure.

Coming as an extension to Scaffold-GS, OctreeGS takes advantage of the multiresolution grids that are generated in the densification process when more anchors are spawned in order to build an octree structure [?]. The octree is not generated from a single root node, but it starts from the grid generated on the SfM point cloud. In this implementation, a level of detail corresponds to a level in the octree.

Initially, the anchors are associated with all LoDs, and each anchor can be rendered at a different detail level. The selection for the appropriate level during rendering is based on the degree of complexity in that area of the scene and the distance between the anchor point and the camera.

Similar to the method that it is based on, this implementation uses grid cell gradients to drive anchor densification. Even though all detail levels start off with the same set of anchors, the densification will be performed differently based on the level. Starting from the gradient threshold defined by Scaffold-GS, a set of additional, increasingly higher thresholds are generated. These values are used to decide, based on the gradient magnitude, which level the new anchor should be spawned in. Higher gradients mean that the anchor will be assigned to a finer detail level. This restricts the anchors from growing too aggressively into the finer levels. Anchor pruning uses the same strategy as Scaffold-GS, removing anchors that fail to produce sufficiently opaque Gaussians.

Training is done in a coarse to fine manner, first optimizing the lowest level of detail, and then enabling the optimizer to spawn anchors into the higher levels one by one. As mentioned before, besides camera distance, scene detail is also a driving factor for LoD selection. To encode this metric, a learnable render bias is assigned to each anchor, which then influences the level selection. Using this bias, anchors close to the camera can be assigned a low detail level if they do not represent complex features, and anchors further away can get a higher level if they contain a lot of detail that contributes a lot to the render. However, these biases are learnable parameters that are optimized to lower the training objective function, so they might not be exactly and only determined by anchor detail complexity.

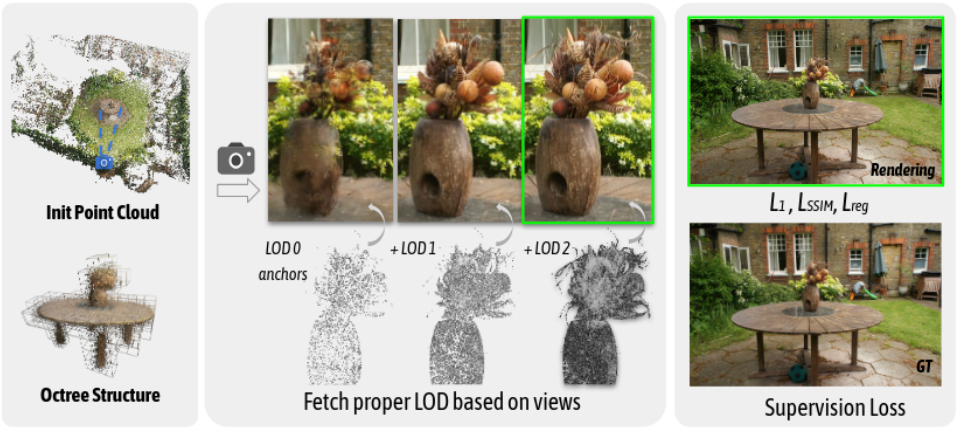

Figure 18: Traning pipeline starting from the sparse point data, to LoD generation and selection [**?**]. The procedure inside each node is similar to ScaffoldGS.

This method, also illustrated in figure 18, provides an image quality increase when the scene is rendered at a lower resolution (i.e., the camera is far away from part of the scene), however, its main advantage is the ability to render complex scenes at real-time 45-47 FPS, while other methods such as Mip-Splatting (10-12 FPS) and Scaffold-GS (3-5 FPS) fail to reach a 30FPS consistent rendering speed.

The authors of the reference 3DGS implementation also provide a solution for optimizing and rendering massive environments through a hierarchical LoD structure [**?**]. Creating a tree-based hierarchy for the levels of detail allows different parts of the environment to be rendered at different complexities while also having good granularity when transitioning between the levels. The proposed structure contains the original Gaussians as the leaves of the tree, and merged primitives in the intermediary nodes. In order to not overly complicate the existing pipeline, the merged primitives are also 3D Gaussians. This poses the problem of merging two Gaussians into one while maintaining the aspect as close as possible to the original. The proposed solution uses a weighted average for the mean and SH coefficients, while the merged covariance takes into account both the initial covariances and the means. The weights are defined by each Gaussian's contribution to the image, which is given by the opacity and the surface area of the ellipsoid defined by the Gaussian distribution. The opacity of a merged Gaussian sometimes has a slower fall-off, so it is allowed to go over 1 and is only clamped at 1 during rendering.

Having the strategy for merging two 3D Gaussians, the next problem is finding candidates for merging. The implementation proposes a BVH partitioning of the space starting from the axis-aligned bounding box of the entire scene as the root node. Then, the volume is divided into two children by a median split. This means that the Gaussians are projected on the longest axis of the bounding box, then the split is performed at the median projection, such that the two resulting will have an equal number of Gaussians (pr differ by one in the case of an odd number of primitives). This process is performed recursively until each bounding box only contains one Gaussian. Then, starting from the leaves, the Gaussians are merged in the interior nodes towards the root. Because this is a binary tree, primitives will always be merged two at a time, even if they are in turn merged representations of other Gaussians.

During rendering, a target node granularity is set depending on the required quality. Then, the problem of selecting the proper level for rendering becomes one of finding a graph cut where all the nodes satisfy the granularity condition, which is determined by evaluating the size of the node's bounding box projection on the screen, as shown in figure 19. Traversing the structure from the root, when a parent node does not satisfy the granularity, but the child satisfies it, the child will be selected into the graph cut. For smooth transitions, the authors propose an interpolation method between parent and children primitives to reduce popping artifacts when transitioning between levels.

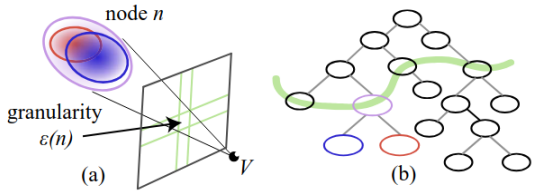

Figure 19: Node granularity computation and the respective graph cut for a target granularity [**?**].

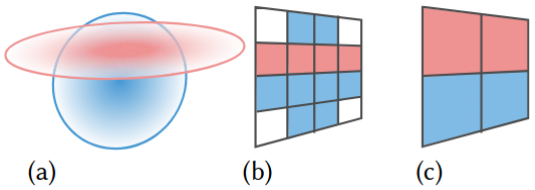

Figure 20: Two nodes in the scene tree (a), and their rasterization at two different target granularities (b and c).

For training, the scene is optimized at multiple granularity levels, as illustrated in figure 20, which allows the merged intermediary Gaussians to also be trained against the full-resolution reference images. This solution is particularly useful when dealing with large scenes. In that case, the scene is split into chunks, each one having its individual LoD hierarchy. A coarse scene prior is first trained and used to represent the environment outside each chunk. The chunks are then consolidated in the final scene.

When evaluated against other methods on chunk-sized scenes, the quality improvements over the reference implementation are small. However, on the SmallCity and Campus datasets, it manages to achieve a consistent 30+ FPS at a 3-pixel granularity, while those scenes would not even be able to be optimized or rendered by the original 3DGS due to their size. However, these results were collected on an NVIDIA A6000 GPU.

3 Overview

In this project, I will present a new approach for accelerating Gaussian splatting rendering through a hierarchical Level-of-Detail structure. This is intended for consumer applications on consumer hardware, where the full scene cannot be rendered in real-time, and detail levels have not been provided with the optimized scene, so the simplifications have to be generated locally. All of the implementations presented in the previous chapter incorporate the levels of detail into the training algorithm, which allows them to optimize all of the levels, thus creating representations with very good quality. However, this requires a significant amount of resources that are not readily available on consumer hardware, so the availability of those LoDs depends on whether they were pretrained alongside the scene or not. Moreover, they also require the initial images for training the detail levels, which might not be readily available in consumer applications.

The method I will present in the following chapters only requires the pretrained scene, from which it can generate an arbitrary number of levels without requiring additional training. Then, at render time, the detail levels for different parts of the scene can be selected based on the camera position and orientation, and the available hardware performance. The implementation takes advantage of the existing rasterization pipeline for 3DGS and introduces minimal changes to the render loop. Figure 21 shows a graphical representation of the pipeline I implemented in this project, highlighting in different colors the pipeline elements existing in the reference 3DGS implementation, and the additional pipeline steps introduced by my implementation.

Figure 21: Overview of the implemented system.

The acceleration structure I propose in this project is built on a hybrid hierarchical space partitioning scheme based on insights from previous works on the topic. The root node of the scene represents the entire scene, which is incrementally subdivided into an octree up to a specified depth. This allows for an even distribution of nodes in the scene, and the maximum octree depth defines the lowest detail level of the simplification. Then, each octree leaf becomes the root node of a binary partitioning tree which will hold the merged and simplified Gaussians. Details on the partitioning structure will be presented in chapter 6.

I will also propose a new partitioning strategy for the Gaussians in the deeper nodes of the tree based on feature clustering, which, for this use case, performs better than previously proposed solutions that are solely based on Gaussian position. Also, because this method does not involve any training or fine-tuning, I will also propose a method for merging the Gaussians to create simplified representations and a comparison to the other methods in the literature. Details on this aspect will be discussed in chapter 5 of this document.

Then, I will discuss the method I used for combining the merging and partitioning algorithm to obtain a hierarchical level of detail structure, and how the appropriate levels are selected at runtime.

In Chapter 8 I will perform an analysis of the performance considerations taken into account when implementing this structure, how it fits into the existing 3DGS rasterization pipeline, and profiling the algorithm. Also, I will investigate its potential for further acceleration through earlier frustum culling.

Lastly, I will present the experimental results in terms of image quality, rasterization speed, and required resources compared to the reference 3DGS implementation. Note that for this project, all of the experiments have been done on consumer hardware, as this is the intended use of this method, and not on workstation-grade GPUs like the other previously presented works.

4 Rendering

In this chapter, I will present the details of the rasterization pipeline for 3DGS, as introduced in the reference implementation. Because the Gaussians are rendered directly without any intermediate representation through other standard primitives, the process cannot take advantage of the existing geometry pipelines implemented on GPUs. In turn, it is based upon a tiling software rasterizer implemented as a CUDA kernel.

The render loop is made up of two main routines: the preprocessing stage and the rasterization routine, with a few intermediary steps in between for splat duplication, sorting, and assignment to tiles. This process takes as input the camera transformation and the Gaussian data and outputs the correct pixel color to a pixel buffer that allows the interoperability between CUDA and OpenGL. The only render call made to OpenGL is for a textured quad that fills the whole frame and is textured with the rasterized image.

4.1 Preprocessing

As discussed previously, Gaussian primitives in 3DGS are defined by the following properties: mean $\boldsymbol{\mu} \in \mathbb{R}^3$, scale $\boldsymbol{S} \in \mathbb{R}^3$, rotation quaternion $\boldsymbol{q} \in \mathbb{R}^4$, opacity $\alpha \in \mathbb{R}$, and a set of spherical harmonics coefficients represented as an array of 48 floating point values, out of which 3 represent the base color, and the rest the specular details. Using this formulation, the distribution of a 3D Gaussian at any point in space $\boldsymbol{x}$ is the following [**?**]:

$$G(\boldsymbol{x} - \boldsymbol{\mu}) = e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \Sigma^{-1} (\boldsymbol{x}-\boldsymbol{\mu})}$$

This value is multiplied by the specific opacity of each Gaussian before being used for the last alpha-blending step.

4.1.1 Gaussian Covariance

We can build the covariance matrix from the scaling matrix $S = diag(\boldsymbol{s}) \in \mathbb{R}^{3\times3}$ and the rotation matrix derived from the quaternion $\boldsymbol{q} = (x, y, z, w)$ as [**?**]:

$$\boldsymbol{R} = \begin{bmatrix} 1 - 2 \cdot (y^2 + z^2) & 2 \cdot (xy - wz) & 2 \cdot (xz + wy) \\ 2 \cdot (xy + wz) & 1 - 2 \cdot (x^2 - z^2) & 2 \cdot (yz - wx) \\ 2 \cdot (xz - wy) & 2 \cdot (yz + wx) & 1 - 2 \cdot (x^2 + y^2) \end{bmatrix}$$

Then, we can build the 3D covariance matrix as $\Sigma = RSS^TR^T$. As the matrix is symmetrical, only the upper triangular region is stored. Now, the 3D covariance has to be projected to screen-space into a 2D covariance. A perspective transformation does not map a 3D Gaussian into a 2D Gaussian on the screen. For simplicity, the EWA splatting algorithm [**?**] is used to approximate the perspective transformation by an affine local transformation using the first-order Taylor expansion at point $\boldsymbol{t}$, where $\boldsymbol{t}$ is the projected mean point of a Gaussian through the camera extrinsic matrix. Let (f_x, f_y) be the focal lengths of the camera. Then we can obtain the Jacobian matrix of the perspective projection mapping at $\boldsymbol{t}$:

$$\boldsymbol{J} = \begin{bmatrix} f_x/t_z & 0 & f_x \cdot t_x/t_z^2 \\ 0 & f_y/t_z & f_y \cdot t_y/t_z^2 \end{bmatrix} \in \mathbb{R}^{2\times3}$$

If the camera extrinsic matrix is:

$$T_{cam} = \begin{bmatrix} R_{cam} & t_{cam} \\ 0 & 1 \end{bmatrix} \in \mathbb{R}^{4\times4}$$

then we can finally compute the 2D covariance matrix of the projected Gaussian as:

$$\Sigma_{2D} = \boldsymbol{J} R_{cam} \Sigma R_{cam}^T \boldsymbol{J}^T \in \mathbb{R}^{2\times2}$$

At this point in the pipeline, the low-pass filter is applied to the covariance by ensuring that splats are at least one pixel in each direction using the following formula:

$$\Sigma_{screen} = \Sigma_{2D} + \begin{bmatrix} 0.3 & 0 \\ 0 & 0.3 \end{bmatrix}$$

The value of 0.3 is chosen somewhat arbitrarily, the reasoning being that when evaluating the ellipse of 99% confidence interval of a Gaussian, which is at three standard deviations from the mean, the value of 0.3 would roughly translate to a 1-pixel dilation in both directions of the variance of the Gaussian. Other methods, such as the Mip-Splatting presented earlier, use values that are influenced by the Gaussian's dimensions and the frequency band limitations of the resolution the scene is rendered at.

4.1.2 Splat geometric properties

The inverse of the 2D covariance matrix Σ_{2D} is a conic matrix that will be used later in the render routine to compute a Gaussian's influence at multiple locations on the screen.

From the eigenvalues of the 2D covariance λ_1, λ_2 where $\lambda_1 > \lambda_2$, we can then determine the minimum radius of a circle centered at the Gaussian mean that contains the 99% confidence of the interval as $r = 3\sqrt{\lambda_1}$. Knowing the radius and the projected location of the Gaussian's center, we can then compute a screen-space bounding rectangle for the splat.

From this bounding rectangle and knowing the dimensions in pixels of each tile in the rasterizer, we get the number of overlapped tiles, determining how many times the splat will have to be duplicated in the next step. This value, alongside the conic matrix, radius, projected center, and splat depth (the value of the Z coordinate in the viewport position) will be stored in global memory and passed to the next step.

4.1.3 Splat color

In order to represent variations in color based on different viewing directions on a single splat, the 3DGS implementation uses spherical harmonics. These are a set of functions defined on the surface of a sphere, which form an orthonormal basis, so any function defined on the surface of a sphere can be decomposed into a series of spherical harmonics of multiple degrees [**?**]. For performance and storage considerations, this implementation uses harmonics up to degree $l = 3$, as additional levels require more storage and only improve high-frequency details. For each splat, the viewing direction vector is normalized, then its components are used as input to the spherical harmonics function. The standard formulation for the spherical harmonics with the Condon–Shortley phase is used, and the coefficients are precomputed and stored in a table. When compositing the color for each channel, each term of the harmonic expansion is also multiplied by a learned coefficient which is produced by the scene optimization procedure. This allows control of various detail frequencies across the four harmonics degrees and the three color channels.

4.2 Splat duplication and sorting

In order for the tiling rasterizer to execute efficiently, each tile workgroup needs to have all the necessary data in a contiguous location in memory. Since splats can overlap multiple tiles, this introduces the need to duplicate splat data for each tile and arrange it appropriately in the device memory. To execute some of these operations, the implementation uses the CUB library, which provides state-of-the-art parallel primitives for the CUDA programming model.

The first step is to determine the total number of splats after duplication, and the array offsets for duplicating in memory. This is done using the overlapped tile values computed in the previous step, on which is applied an inclusive prefix sum scan. This computes, for each element, the sum of all previous values in the array, including itself, and stores the sum in a new results array. This means that given an array of integers of size n $\boldsymbol{x} \in \mathbb{N}^n$, the resulting inclusive prefix sum array $\boldsymbol{s}$ is:

$$\boldsymbol{s}_k = \sum_{i=0}^{k} \boldsymbol{x}_i$$

The values in this array will then provide the necessary offsets for duplicating the splat data in memory, and the amount of memory that has to be allocated. The formulation above, implemented on the CPU, has a time complexity of $\mathcal{O}(n)$. However, taking advantage of the fact that the data is stored on the GPU, the implementation uses the parallel version of this prefix scan routine [**?**], which takes advantage of the massively parallel capabilities of GPUs and can perform this sum in $\mathcal{O}(\log_2 n)$ steps by aggregating partial sums.

The next step is duplicating the splat IDs and preparing them for sorting. The tiling rasterizer needs the splats to be ordered by depth, and the data in global memory needs to be contiguous for each tile. To achieve this, the duplicated IDs will be sorted by a set of precomputed keys. During duplication, keys containing the overlapped tile ID and the splat depth are generated for each duplicated splat ID. This means that each duplicated splat will be associated with a unique splat. We wish the arrangement in memory to be done first by splat ID, so splats overlapping the same chunk are contiguous in memory, and then by depth inside each block, so the rasterizer does not have to perform any further sorting. Thus, the keys are built as 64-bit values, where the higher 32 bits represent the overlapped splat ID, and the lower 32 bits represent the splat depth, as shown in figure 22. Because the tile ID occupies the higher-significance bits, it will have priority during sorting.

Figure 22: Sorting key structure for splat ordering.

After sorting, the keys will have a structure as seen in figure 23, where the first half of each cell is color-coded by tile ID, and the gradient of the second half shows the depth. To sort the values in device memory by keys, the most efficient method is to use RadixSort [**?**], also implemented in the CUB library. It performs a least significant digit sorting, so the values are sorted in multiple passes, from the least significant digit to the most significant digit. Even though the depth component of the key is a floating point number, it will be interpreted as an integer, which is not an issue since the depth always has the same sign, so ordering as an integer representation produces the same result. The number of passes necessary for sorting depends on the number of digits of the key with the highest value, and it scales linearly with the number of sorted elements and performs very efficiently on the GPU architecture. Because usually, the value for the tile ID does not take up the whole 32 bits allocated for it in the sorting keys, we can determine the highest non-zero bit across all keys and terminate the sorting there, which might reduce the number of necessary sorting passes, depending on the stored values. In my testing, this extra check eliminates one of the passes, resulting in a small performance improvement.

The last step before calling the rasterization routine is to determine, for each tile, the start and end range of its data in the global splat ID array. Ranges are also shown in figure 23.

Figure 23: Sorted keys showcasing tile ranges.

4.3 Splat Rasterization

The rasterizer used in the reference 3DGS implementation cannot take advantage of the hardware graphics pipeline for processing splat primitives, so a tiled software rasterizer is used instead. This means that the rasterization is done through a programmable CUDA kernel. The screen is split into a grid of 16×16 pixels each, and the image on each grid block, or tile, is generated independently of the other tiles and composed in the end in the output buffer, thus the need for splat duplication done in the previous step. The render kernel is launched as a grid of blocks, and each block processes one screen tile and is made up of 16×16 threads. This means that there is one thread processing each pixel on the final render.

Inside every block, all threads will process all the splats assigned to the tile. Global memory accesses are costly and can significantly stall the kernel's execution and would be very wasteful especially if all threads need to access the same data. To avoid this memory access overhead we can take advantage of the local shared memory of each Streaming Multiprocessor [**?**], which is however much smaller than global memory and cannot fit all splat data at once. To go around this limitation, the image can be composed in multiple passes, each pass processing 256 splats assigned to the tile. At the start of a pass, each thread loads into local shared memory (L1 cache) the screen position, conic matrix, and splat ID. Then, a synchronization barrier is used to ensure all data is loaded before proceeding to the next step. After synchronization, each thread in the block processes all the splats collected in local memory and computes their influence on their assigned pixel in the render. After the color blending is done, the block loads and processes the next batch of 256 splats and the process repeats until all splats assigned to the tile have been rasterized.

The influence of a Gaussian on a specific pixel is determined by the distance between the 2D Gaussian on the screen and the pixel location, which is passed through an exponential falloff function. Because the EWA volume splatting algorithm projects 3D Gaussians to screen as ellipses, we can define the following radial basis function:

$$r(\boldsymbol{d}) = \boldsymbol{d}^T \boldsymbol{Q} \boldsymbol{d}$$

where $\boldsymbol{d} \in \mathbb{R}^2$ is the component-wise distance vector between the center of a splat and the pixel position and $\boldsymbol{Q} \in \mathbb{R}^{2\times 2}$ is the conic matrix of the splat (i.e. the inverse of the 2D covariance matrix). Then, opacity of a splat centered at point $\boldsymbol{s} = (s_x, s_y)$ evaluated at a pixel with center $\boldsymbol{p} = (p_x, p_y)$ is:

$$\alpha_{\boldsymbol{p}} = \alpha \cdot e^{-\frac{1}{2} r(\boldsymbol{s}-\boldsymbol{p})}$$

where α is the learned base opacity of the Gaussian (i.e. the opacity at the center of the splat).

Splats are processed front to back, and the contribution of each splat is accumulated for each pixel following an alpha-blending compositing formula. In the end, the color of a pixel k is given by the following formula:

$$\boldsymbol{C}_k = \sum_{n<N} \boldsymbol{c}_n \cdot \alpha_n \cdot T_n \text{ where } T_n = \prod_{i<n} (1 - \alpha_i)$$

Here, N designates the number of splats overlapped by the tile that the thread processing pixel k is assigned to, and $\boldsymbol{c}_n$ is the color of splat n. To ensure better performance splats with computed opacity lower than $\frac{1}{255}$ are skipped, and the color compositing can be terminated early if the remaining transmittance coefficient T_n is lower than 10^{-4}, as the pixel is considered to have reached "full" opacity and any further contributions are insignificant.

4.4 Performance profiling

The scope of this project is to propose an acceleration structure for 3DGS rendering, so after explaining the rendering pipeline, it makes sense to also perform a short performance analysis in order to identify potential bottlenecks and the routines that would most benefit from potential speedups. The profiling has been performed on the "Train" scene of the *Tanks and Temples* dataset [**?**], using one of the standard training camera positions. Data has been collected using NVIDIA Nsight Compute, which offers launch statistics, timing, and bottleneck statistics on profiled CUDA kernel launches, and will be an indispensable tool for this project. All the experiments for this project have been performed on a laptop with an NVIDIA RTX 3050 Ti GPU with 4GB of VRAM running at 35W.

Figure 24 shows the render pipeline profile. The final render routine takes up around 65% of the execution time of the pipeline, indicating that it would be the best candidate for optimization, followed by the sorting at 13.7% and 6.17% for the duplication and key generation. However, Nsight Compute reports a compute throughput of over 90% for the render routine, so optimizing the code will most likely result in minimal improvements, and the RadixSort routine is highly optimized already for the CUDA architecture. This indicates that if we wish to maintain the same pipeline for rendering, the most obvious way to increase the performance is to render fewer Gaussians in each frame.

Figure 24: Render pipeline routine duration.

Figure 25: Render routine speedup when reducing the number of rendered Gaussians.

To investigate the potential improvements of reducing the number of rendered Gaussians, I profiled the render routine on the same scene, rendering all the Gaussians, half of the Gaussians, and lastly a quarter of the Gaussians. Figure 25 shows the speedup obtained by reducing the number of Gaussians in the scene, indicating an almost linear relation between render time and the number of primitives. Of course, the relationship is not always linear and further decreasing the number of primitives produces diminishing returns, as kernel launch overheads become more relevant. However, just arbitrarily removing Gaussians from the scene is obviously not a good solution, so there is a need to create a scene simplification structure, which I will present in the following chapters.

5 Gaussian Merging

As discussed in the previous chapter, one way to accelerate 3DGS rendering is by reducing the number of primitives in the scene, which can be done using simplified representations, similar to levels-of-detail on triangle meshes. Some implementations presented in the literature review propose very simple methods for Gaussian merging, such as averaging all properties. This technique is not accurate for all properties, but in those cases, it is a good starting point, as the simplified representations are also trained during the scene optimization process. However, for this implementation, I need to find an approach that produces quality results without further fine-tuning, because the simplifications are used directly as they are generated at runtime. In this chapter, I will present the approach used in this implementation for merging two Gaussian into one primitive in order to obtain scene representations with fewer Gaussians.

5.1 Spherical Harmonics

The main use of the merged Gaussians is to replace primitives far away from the camera with a simplified representation in order to increase performance by decreasing detail in areas where the loss in quality is not that noticeable. Thus, the intended use of the simplification is distant areas from the camera. In this situation, the original Gaussians will only take up a reduced area on the screen, which would be equivalent to rendering them at a lower resolution if they were closer to the camera. Using the insight from studies on antialiasing for Gaussian splatting, a good option for band-limiting the rendered primitives is to apply a box blur filter that acts as a low-pass filter. Applying a 2D box blur filter to a patch of pixels on the screen is done by convolution of the initial image with a 3×3 kernel with the following formulation when applied in image space [**?**]:

$$H = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

As expected, the blur used to subsample the projected primitives to improve aliasing takes the average of neighboring pixels. That indicates that in order to get a similar representation from a single primitive, a good choice for the color is the mean of the initial primitives. However, taking a simple average of the color would ignore the different contributions different Gaussians have to the final image on the screen. Some implementations propose determining the visual importance of a Gaussian as the amount of overlapped pixels over all training images. While this is a relevant metric, the method I am proposing is intended for generic use on 3DGS scenes, not necessarily aiming for the highest metrics on a set of predetermined camera positions. For this reason, I chose to consider the relative contribution of a Gaussian as a function of its learned opacity and volume. Thus, I chose the following formulation to define the weight of a Gaussian i: $w_i = \alpha_i \cdot V_i$, where $V_i = s_x \cdot s_y \cdot s_z$. Because the actual color is computed in every frame depending on the camera position, I will blend the Gaussians using a weighted mean on the arrays of SH coefficients. The coefficients are separated by color, so the weighted mean can be applied to each channel individually and will provide the desired result when recomposing the color from the new SH coefficients and viewing direction.

5.2 Opacity

In order to compute the opacity of the merged Gaussian, I am using an approach similar to the one in [**?**]. One thing to note is that the opacity property of Gaussians defines the maximum opacity at the center of the splat and is the value from which the splat's opacity falls following a Gaussian curve towards the edges of the splat. However, when two Gaussians projected to the screen overlap, the perceived opacity across them does not follow a Gaussian distribution, as the two opacities are composed over the overlapping region, so the falloff is slower than a Gaussian.

It is important to model this behavior, otherwise the merged primitive will have lower perceived opacity, and applying the merging procedure hierarchically would result in significant opacity loss across the scene. Consider two partially overlapping Gaussians projected on the screen as in figure 26. A scanline plotting the

Figure 26: Scanlines across the initial Gaussians and the merged one.

opacities across the space occupied by the two splats will produce the profiles shown below, where figure 27 shows the sum of opacities generated by the distributions, and figure 28 shows the total opacity saturated at 1, as that is the maximum opacity accepted by the rasterization routine. Note the plateau around the center of the two splats generated by the sum of the two decaying opacities, which cannot be directly modeled by a Gaussian distribution.

However, the shape of the function above the $\alpha = 1$ line is not important, as the opacity will be saturated at 1. This means that this behavior can be modeled by a single Gaussian distribution with a peak higher than 1, as shown in figure 29. This is an oversimplification of the potential situations for Gaussian opacity merging, and the height of the distribution would depend on many variables such as the distance between primitives and their scales. However, it showcases the need for primitives after merging to have an opacity greater than 1 to simulate a slower fall-off.

Figure 27: Scanline opacities.

Figure 28: Perceived opacity.

Figure 29: Merged opacity model.

I chose to use a similar approach to the *Hierarchical 3D Gaussian Representation* paper from INRIA, but changing the splat surface to Gaussian volume, as it would be a more reliable metric for any arbitrary view in the scene. Thus, the opacity of a merged Gaussian is the following:

$$\alpha_m = \frac{\sum_i^N w_i}{V_m}$$

where V_m is the volume of the new Gaussian from its covariance matrix, as will be explained in the next subchapter, the weights w_i are the same as for the color merging.

5.3 Mean and Covariance

The most difficult properties to merge for Gaussian primitives are the geometric ones, such as mean and covariance. Especially the covariance, it defines both the spread of the distribution and its directions, i.e. the orientation of the Gaussian in space. A lot of research has gone into reducing the dimensionality of Gaussian Mixture Models, both from a purely analytical perspective [**?**] and for use in hierarchical photon mapping [**?**]. Considering any arbitrary normalized weights w_i' assigned to a set of N Gaussians, the mean and covariance of the merged Gaussian are the following:

$$\overline{\mu} = \sum_{i \leq N} w_i' \mu_i$$

$$\overline{\Sigma} = \sum_{i \leq N} w_i' (\Sigma_i + (\mu_i - \overline{\mu})(\mu_i - \overline{\mu})^T)$$

This method aims to minimize the Kullback–Leibler divergence, which measures how much two probability distributions differ. The formulas above are also used by [**?**] to generate their intermediary nodes, but note that the scene training algorithm then optimizes the generated nodes. During testing, I implemented this merging approach but found that a direct implementation of the formulas above results in Gaussians that have lower volume than expected, so applying the same approach on multiple levels resulted in significant gaps appearing in the scene. Note that at the time of writing, the implementation of [**?**] was not published, so there was no reference to any pre- or post-processing they are doing to achieve the results described. Also, their very good quality metrics are very likely to be due to the extended training process after the merged Gaussians are generated.

In order to test an alternative to the method above, I chose to implement a covariance merging approach based on the volumetric coverage of the combined Gaussians. The challenge when merging primitives is to get a similar shape to the initial distributions and fill in the same space occupied by them. To achieve this, I chose to follow an approach that analyzes the volumetric extent of the composed Gaussians and tries to reproduce the same volumetric coverage using only one distribution.

Given a set of N 3D Gaussian distributions, the first step is to determine their volumetric coverage, which is the space they occupy inside their 99% confidence interval. I chose to use this interval as it is

the same one that is used after the primitives are projected to determine their on-screen bounds. From the eigenvectors of the covariance matrix, we can determine the axes of the confidence ellipsoid, and by scaling these vectors by a factor of 3, we get the extent needed to cover the desired confidence interval. Given the scaled vectors $\boldsymbol{s_1}, \boldsymbol{s_2}, \boldsymbol{s_3}$ that represent the ellipsoid axes and the center point of the Gaussian $\boldsymbol{\mu}$, I can generate a coverage point cloud P defined as

$$P = \{\boldsymbol{\mu}, \boldsymbol{\mu} + \boldsymbol{s_1}, \boldsymbol{\mu} + \boldsymbol{s_2}, \boldsymbol{\mu} + \boldsymbol{s_3}, \boldsymbol{\mu} - \boldsymbol{s_1}, \boldsymbol{\mu} - \boldsymbol{s_2}, \boldsymbol{\mu} - \boldsymbol{s_3}\}$$

which contains the center point and the ellipsoid vertices. This procedure is applied to all the primitives that are going to be merged, then we concatenate the sets of points generated by them:

$$Q = \bigcup_{i=1}^{N} P_i$$

The final volumetric coverage point set will contain $7N$ points. To get the merged Gaussian mean, we will take the weighted average of these positions, the weights being the same as for the other properties before:

$$\overline{\boldsymbol{\mu}} = \frac{1}{\sum_i^{7N} w_i} \sum_{\boldsymbol{p} \in Q} w_p \boldsymbol{p}$$

The weights for all 7 points generated by one Gaussian are the same. Then, to compute the equivalent Gaussian spread, we only need to compute the covariance of the point set. Let the matrix $D \in \mathbb{R}^{7N \times 3}$ be defined as:

$$D_i = Q_i - \overline{\boldsymbol{\mu}}, 1 \leq i \leq 7N$$

and the diagonal weight matrix $W = diag(\boldsymbol{w}) \in \mathbb{R}^{7N \times 7N}$. Then, the weighted covariance matrix [**?**] is defined as:

$$\Sigma = \boldsymbol{D W D^T}$$

With the two formulas above we have the necessary information to describe the new distribution. Figure 1 shows a simplified 2D representation of the coverage points and the computed resulting distribution for four Gaussians placed in different configurations in space.

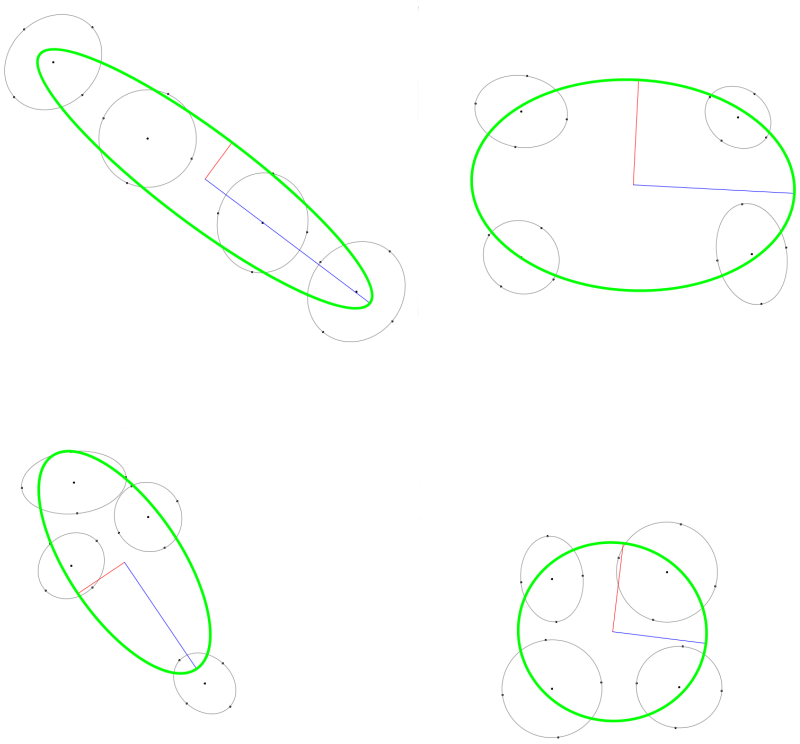

Figure 30: Examples of computed point spread.

The gray lines show the confidence ellipse of the initial Gaussians, and the dots show the distribution centers and ellipse vertices. The green line represents the confidence ellipse of the merged Gaussian, and the blue and red lines represent the covariance matrix eigenvalues. We can see that for Gaussians that are close together, the distribution spread follows the general shape. However, when the primitives are spread apart, the resulting merged Gaussian covers a significant amount of empty space. This highlights the necessity of a good selection mechanism for choosing the candidate Gaussians for merging. Ideally, we would like the primitive to be compact and without empty spaces between them.

6 Spatial Partitioning

Since in the previous chapter I presented the method for merging 3D Gaussians and outlined some potential issues, the focus of this chapter is to discuss scene partitioning methods, which in the end will be the criterion for selecting groups of primitives for merging. I will first go over two well-known methods in computer graphics that have been explored by other publications on this topic, then present a novel approach and justify its advantages for this application in particular.

6.1 Octrees

Octrees are one of the simplest forms of space partitioning in computer graphics. Usually, the root node of the tree represents the entire axis-aligned bounding box of the model or scene to be partitioned. Then, the node is split into $2 \times 2 \times 2$ nodes, each axis of the box being split simultaneously at its midpoint, resulting in 8 children nodes of equal sizes. The procedure then repeats recursively for all newly generated nodes until a stopping criterion is met. The criterion for stopping the subdivision could be reaching a maximum depth in the tree, reaching a minimum dimension of the leaf nodes, or it could depend on the number of primitives in the leaves since at some point subdividing further would not be beneficial for the application. Every time a node is subdivided, the primitives contained by the parent will be distributed between the children based on the bounding box intersection. Usually, primitives are stored only in the leaf nodes, and the set of primitives contained by an intermediary node can be determined by traversing its subtree.

The octree can be used either as a structure for partitioning point data, such as vertices in a triangle mesh or a point cloud, or they could be the implicit representation of the data, for cases in which the information is better displayed as voxels than point geometry. Moreover, this kind of structure can be useful for queries in occlusion tests, collision detection or ray tracing.

Being a regular structure, octrees have some advantages. For example, the locations of the splitting planes are predetermined, and the size of the node can be determined during traversal from the size of the root node and the path taken through the tree [**?**]. However, their regularity comes at a cost if the scene or model does not have a somewhat uniform distribution of primitives inside its bounding box. Since nodes are always generated at half the dimensions of the parent and at predetermined positions, many generated nodes may be empty. Moreover, clusters of very close-by primitives may take a large amount of recursion steps to be split into different nodes, because the splitting planes are generated at specific positions and do not adapt to the data. Figure 31 shows two examples of octree subdivisions for a set of Gaussians (shown in 2D as a quadtree for easier representation), the illustration on the right highlights the inefficiencies of regular space division, as many nodes end up empty.

Figure 31: Octree (quadtree for simplicity) examples for different cases of primitive organization in space. Empty nodes are shown in red.

To determine the repartition of Gaussians inside nodes, I am using only the center point of the Gaussian, and determining in which one of the eight children each primitive should be distributed. I experimented with considering the bounding box of the confidence ellipsoid and duplicating Gaussians where more nodes overlap, but this leads to an exponential increase of primitives in the acceleration structure. The method I have settled with is using the center point for node assignment and the AABB of the confidence ellipsoids to determine node bounding boxes used in rendering and generating the dynamic LoD. Differences between the node bounds and the computed bounding box for rendering can be seen in Figure 32.

Figure 32: Octree (quadtree for simplicity) showing node bounds and the node AABB computed from confidence ellipsoids.

6.2 BSP Trees

Binary space partitioning trees are similar to octrees, the main difference being that each node is split into only two child nodes. Also, research on space partitioning for 3DGS suggests that a median split works well for this kind of scene. This approach implies splitting each node along the longest axis of its bounding box. As before, bounding boxes are axis-aligned, just like the splitting planes. To find the splitting plane, all the primitive centers are projected on the longest axis, sorted, and then the median position can be found, through which we will split the node, as shown in figure 33. Then, given that the projected positions are already computed, it is easy to distribute the primitives between the two children. Similarly to the octree, primitives are only stored in the leaf nodes.

Figure 33: Node subdivision for BSP tree showing projected centers on the longest axis and the split plane with a red dotted line.

Because the positions of the splitting planes are not predetermined, the data structure needs additional storage to hold the information about its volume. On the other side, using a median splitting plane ensures that the final structure is a balanced tree, and there are no empty nodes. Even though BSP trees might need more depth than an octree to partition all primitives, this highly depends on the distribution of the primitives, as can be seen in figure 34.

Figure 34: BSP tree examples for different cases of primitive organization in space. No empty nodes compared to octree.

Other than the splitting method, for this implementation I am using the same approach as with the octrees, distributing primitives based on center position, and computing an additional bounding box based on the confidence ellipsoid. From my experiments, I have found that using a BSP strategy for generating the LODs gives more natural results since all leaf nodes differ in depth by one at most, so when generating a cut in the tree for creating the appropriate LOD, primitives tend to be more uniformly simplified, instead of having high-frequency Gaussians stuck in higher levels in the tree, which would require more simplification steps to be reached. Also, this structure allows for finer transitions between levels, since only two Gaussians are merged at a time, instead of eight.

6.3 Hybrid Partitioning

In this last section, I will present a hybrid space partitioning algorithm that I have implemented for this application, taking into account the advantages and disadvantages observed in the two approaches presented above. This method proved to produce better quality results, which will be discussed in a later chapter.

The first part of this approach involves the initial space subdivision, which is intended to result in a uniform distribution of nodes throughout the scene. Because these nodes are very coarse and contain a high count of primitives, they will not be used for generating the levels of detail. Instead, they are used as a starting point for the second subdivision step. This is why for this initial step I chose to use an octree partitioning approach, up to a specified depth, which is determined usually by the scene complexity.

For the second part of the partitioning, the primitives have to be more carefully grouped into nodes. When generating the LoDs, the distribution of Gaussians in nodes determines the order in which they will be merged, and because there are no further scene optimization steps applied after this, creating quality merged primitives is essential for reliable LoD representation. As discussed above, binary space partitioning is advantageous when considering that the stored Gaussians have to be merged, as we have more flexibility in determining the distribution of primitives in nodes. Moreover, binary splitting allows for finer transitions and more possible intermediate representations, as we have more interior nodes than an octree.

When merging the primitives as explained in the previous chapter, the best results for merged Gaussians are obtained when the initial primitives have similar properties, whether it is spatial coherency or

color similarity. This is because, for Gaussians clustered in space, the resulting distribution will approximate well the volumetric coverage, and for color similarity the resulting color will not deviate too much from the appearance of the initial Gaussians. While the median split method works well in evenly clustering spatial structures, in this implementation I will explore clustering-based methods for distributing Gaussians in the children nodes of the tree.

What I want to achieve in this step of space subdivision is a distribution in which, for a parent that is split into two children, the variance of the primitives inside nodes is low, while the variance between nodes is high. Because both position and color should be taken into consideration, each Gaussian is described by a feature vector $\boldsymbol{v} = (x, y, z, r, g, b)$, where the color is taken as the first spherical harmonic coefficient on each channel, as that is the one describing the base color of the primitive, and introducing multiple degrees of the harmonics in the clustering would both be inefficient in terms of computation complexity and would bias the clustering towards color and variations based on viewing position. Also, it is worth mentioning that the position is relative to the node center and normalized to the extent of the node's bounding box, so the clustering would not be biased towards position in large nodes or nodes far from the world origin.

K-means clustering [**?**] is a very popular clustering algorithm that partitions the observation points in k groups, where k is specified at the beginning. The algorithm is based on a heuristic approach and minimizes within-group variance, which is one of the goals for this step. However, it does not take into account the specific variations of position and color, so these should be first weighted based on their importance in the clustered set. For example, if the spatial distribution is uniform, the clustering should prioritize color, and if the color is uniform the clustering should prioritize spatial features.

Spectral clustering [**?**] makes use of the eigenvectors of the similarity matrix to project the points in a lower-dimensional space, and the clustering is then performed in this space. This ensures that only the components with the highest observed variation in the dataset are used as a basis for clustering. This method is used widely in graph partitioning, as the similarity matrix is built on the edge costs for traversing the graph. Even without connectivity information, the similarity matrix of a point set can be built based on the Euclidean distance of the feature vectors assigned to the points. While this works for small sets of points, the similarity matrix of N points is of size $N \times N$, and computing its eigenvalues becomes too inefficient for this application considering the amount of Gaussians in the scene.

DBSCAN [**?**] is another clustering method that allows partitioning clusters that are not linearly separable, and the point allocation is done through a method similar to region-growing, where points are added to clusters based on the local spatial density. However, this method has the same issue as k-means, where weights for positions and color would have to be computed apriori if we wish to distinguish between features, but an even greater issue is the concept of data noise, which allows this method to leave outliers uncategorized. This does not work for this application, as we wish all the Gaussians to be distributed in a node and take part in the merging process.

Given the insights from these methods, I chose to implement a dimensionality reduction algorithm similar to spectral clustering. However, in order to reduce the computational strain of eigendecomposition for large matrices, I am using the covariance matrix of the feature vectors. Given an intermediary node containing N primitives, we first compute the average feature vector $\overline{\boldsymbol{v}}$:

$$\overline{\boldsymbol{v}} = \frac{1}{N} \sum_{i}^{N} \boldsymbol{v}_i$$

then the matrix of feature vector variance $D \in \mathbb{R}^{N \times 6}$:

$$D_i = \boldsymbol{v}_i - \overline{\boldsymbol{v}}, i \in [1, N]$$

and we can obtain the covariance matrix $\Sigma \in \mathbb{R}^{6 \times 6}$ as:

$$\Sigma = D^T D$$

Lastly, through eigendecomposition, we obtain the set of six eigenvectors. Let $(\boldsymbol{e}_1, \boldsymbol{e}_2)$ be the two eigenvectors with the highest associated eigenvalues. This means that the highest amount of variation in the data appears along these two directions. Now, we will project the 6-dimensional data on this 2-dimensional

space, resulting in the set of new feature vectors $\boldsymbol{u}$:

$$\boldsymbol{u}_i = ((\boldsymbol{v}_i - \overline{\boldsymbol{v}}) \cdot \boldsymbol{e}_1, (\boldsymbol{v}_i - \overline{\boldsymbol{v}}) \cdot \boldsymbol{e}_2) \in \mathbb{R}^2$$

The last clustering step is to apply k-means on the data points using the new feature vectors to measure point distance and variation from the mean. This method allows us to select the features where the highest variance is observed, automatically bias toward them when clustering, and ideally obtain clearer separation in the new 2-dimensional space. The choice to use only two eigenvectors comes from the fact that we are only clustering the points into two partitions, and using a higher dimensionality did not bring any improvements in my tests.

To have a visual explanation of this algorithm, figure 35 shows a set of 2D points with color information, where a cluster of one color is surrounded by points of a different color. These clusters are not linearly separable, so k-means would not be able to determine the clusters by creating a separating plane in the image space. However, after applying the algorithm above, we observe a clear separation of the points, which can then be easily clustered to obtain a better separation.

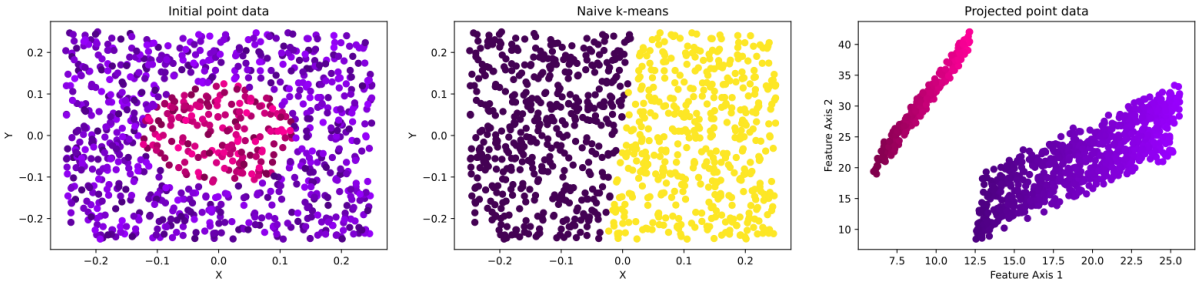

Figure 35: Example of color feature separation in uniform distribution of points in space.

In order to showcase the spatial feature separation, figure 36 shows two parallel lines with noise, which are close enough together that k-means would not be able to distinguish between them. However, after projecting to a lower-dimensional space based on feature significance, we get a more distinguished separation. These two cases could be also directly solved by k-means by scaling the data and choosing an appropriate separating axis in the 6-dimensional space, but this algorithm is more robust since the choices for transforming the data are based on the distribution of data in the initial point set.

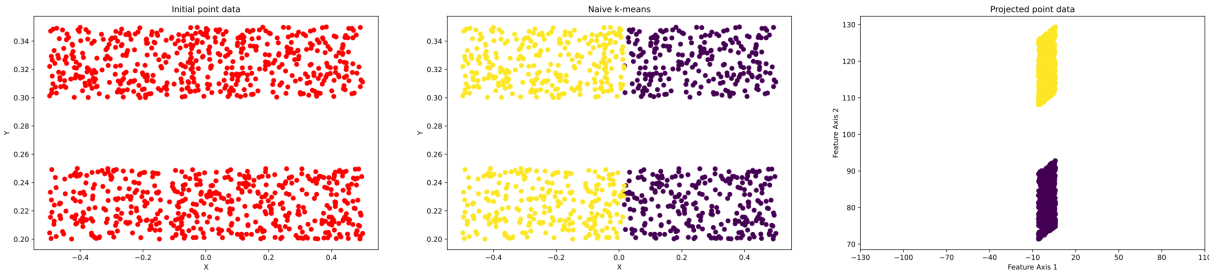

Figure 36: Example of spatial feature separation with uniform color.

Note that this partitioning approach for the BSP component of the tree does not result in volumetric separation of the primitives, and in some cases, the bounding boxes of the resulting children nodes might have significant overlap. However, this produces significantly better results in terms of image quality. More examples using actual 3DGS scenes will be shown in the Results chapter.

7 Level of Detail Generation and Selection

Now that I have presented the methods implemented for space partitioning and Gaussian merging in the previous two chapters, I will discuss how these two concepts are combined to create the acceleration structure. The solution I propose in this project is a view-dependent continuous LoD structure [**?**] since the amount of detail is selected dynamically for each frame depending on the camera position relative to scene elements and the desired granularity.

7.1 Generating the Level of Detail

The level of detail representation is generated in an incipient step when the scene is loaded into memory. The scene partitioning algorithm begins from the bounding box of the entire scene, generated from the confidence ellipsoids of all Gaussians. The octree component of the hybrid partitioning is built without holding any intermediate information in the nodes, except the connectivity information to the children, as no simplification takes place at this level. For most scenes in my tests, I am using an octree depth of 12, which provides good node uniformity across the scene while leaving enough space for multiple levels of detail in the BSP part. The primitives contained in each node are exclusively passed down to the children based on their location inside the parent node.

Figure 37: Simplified representation of scene tree showing the different types of nodes and representatives allocation.

When transitioning into the BSP component, the Gaussians are merged from the coarsest nodes to the finer nodes. When a BSP node is processed, all the Gaussians contained in the node are merged using the method presented earlier. The new primitive is added to the list of scene primitives, and its index is stored in the intermediary node. Also at this point, we use the primitives contained in the node to determine the actual bounding box of the node, which is used later for level selection. Then, we determine the two distinct clusters in the data, and the Gaussians are passed down to the two children depending on their respective clusters. The process repeats recursively until we reach the leaf nodes of the tree, which only contain one primitive, and all the intermediary BSP nodes have a reference to their merged Gaussian. From now on, I will refer to merged primitives of interior nodes as *representatives*, as their intended use is to provide a good representation for a set of multiple Gaussians. Compared to other implementations, which build the representations from the finer level to coarse levels by only merging two primitives at a time, I have found that using all Gaussians included in the subtree of a node provides better results when no further training is involved. This means that all representatives are built directly from the original Gaussians, instead of deep intermediary nodes being built by merging two child representatives. Figure 37 shows a simplified representation of the hybrid scene tree.

7.2 Level Selection

The second part of the acceleration structure is the level selection. This is done by setting a desired primitive granularity, which dictates which nodes will be passed for render. A larger granularity means that larger intermediary nodes will rasterize their representative, instead of allowing traversal to their children, while a small granularity ensures that the traversal will reach the finer nodes and more primitives will be rendered, resulting in higher quality. The granularity of a node is defined as the approximated area on the screen when the primitive is projected. Because projecting representatives is somewhat costly to be done for all nodes, I am using the bounding box of the node. However, instead of projecting all 8 points of the box and then determining the area, I only compute the projection of its diagonal as if it was viewed perpendicular to the diagonal axis. This metric significantly reduces the overhead when searching for node render candidates and gives a good estimation of the perceived size of a node when projected. Tests using the full box projection showed no improvement in image quality, but the processing time increased significantly.

Given a node with the diagonal of length d, the projected size on the screen d_p, as defined above, is

computed as:

$$d_p = \frac{d}{D} \frac{W_{screen}}{FOV_y}$$

where D is the distance from the camera to the node, W_{screen} is the width of the viewport in pixels, and FOV_y is the horizontal field of view of the camera. Figure 38 shows a simplified view of this process.

Figure 38: Node granularity computation. Node diagonal is shown in magenta, and its projected length on the image plane is shown in red.

The tree traversal starts from the octree leaves, as these mark the roots of the set of BSP subtrees. The tree structure is traversed in a depth-first manner. At each node, we compute the length of its projected diagonal. If the size is larger than the target granularity, the traversal continues to its children, as the node is too large to be rendered directly through its representative. If the projected size is smaller than the granularity, it means that the node satisfies the criterion, its representative is marked for render, and the traversal of that path is terminated (i.e. the node will not add its children to the node processing queue). In case the traversal reaches a leaf, its assigned primitive will be automatically marked for render. This means that setting a target granularity of 0 tells the LoD selection algorithm to render the scene at the highest quality possible. This effectively creates a cut in the tree that marks the nodes that have a granularity smaller than the target granularity, and their immediate parents have a granularity higher than the target granularity.

The intuition behind this approach is that parts of the scene closer to the camera will have a higher projection length, prompting the traversal to advance further into the tree representation and allocate more primitives for rendering that part. Conversely, parts farther away from the camera will have a smaller projection, so they will be simplified more, since the loss of detail in the background is less noticeable, especially when the primitives would only rasterize to a few pixels. However, the higher the detail level, the deeper the traversal has to go inside the scene tree, which means there are higher overheads. These details will be discussed in the Results section.

This section concludes the discussion about generating and using the dynamic level of detail, as I have presented my approach to scene partitioning, Gaussian merging, and lastly how these two combine to create the acceleration structure.

8 Implementation and Performance Considerations

This section is dedicated to discussing some implementation details, as well as some performance aspects of the implementation considering the target hardware. The implementation of GPU device functions has to take into account overheads introduced by uncoalesced memory accesses, maximizing compute throughput and some other limitations characteristic of the CUDA architecture.

The implementation of the subdivision algorithm and the LoD generation is straightforward, and I did not implement any heavy optimizations because the process only runs once when the scene is loaded, and then the structure is never modified during rendering. This also means that the tree structure can be serialized and written to disk, to facilitate faster loading on subsequent runs. The process could potentially be optimized to run on multiple threads, however, the creation of additional scene primitives implies potentially conflicting writes to the global scene primitive array. There are workarounds to this, like processing subtrees in parallel, and then concatenating their merged primitive vectors, however, this was not the scope of the project, as the overhead can be easily avoided by writing the tree information to disk.

The tree traversal algorithm, however, runs at the beginning of every frame in order to mark the primitives that are eligible for rendering. This means that I have to take special consideration of the performance aspects of the implementation, as overheads that are too large may render this solution ineffective. Because the scene tree is generated in a CPU function call and new nodes are allocated dynamically, we cannot ensure that all the tree data is in a contiguous block of memory after it is generated, as it depends on how the Operating System handles dynamic allocations. After the scene tree is built, I allocate a sufficiently large buffer, traverse the tree, and insert the node information in the buffer, replacing node pointers with buffer indices. This first traversal is also done depth-first, as this is the access pattern when the tree is traversed in the GPU kernel. Now, the tree buffer can be transferred to GPU memory in only one call ensuring memory continuity.

Another significant challenge is the actual tree traversal in the GPU. The CUDA programming model performs well on tasks that perform the same operation on a very large amount of data, and the instructions executed by each thread are mostly the same. Traversing a very deep tree poses problems, as threads in a warp will start to diverge in their execution, which leads to stalls. Also, traversing the entire tree from its root node will pose some issues, as the execution is not parallelizable due to the low amount of concurrent data. To address this issue, I am "splitting" the scene tree at the octree leaf nodes. This means that each subtree starting from an octree leaf node will become an independent structure which will be processed in its entirety by one thread. This results in a large set of shallower trees that are completely independent of one another, so they can be traversed in parallel without issues, and the result of the traversal will be the same as a sequential traversal from the scene root. This optimization takes advantage of the fact that the octree component holds no primitive information and is only used for connectivity information, so it can be ignored in the traversal and we can start directly from the BSP roots.

The changes above ensure that the tree can be traversed in parallel by a large number of threads, but I still have to discuss the actual traversal implementation. Tree and graph-like structures in general are notoriously hard to process in GPU kernels, as there is no effective way to ensure coalesced memory accesses and avoid random memory accesses, which come at a great overhead. Moreover, for this application, the computational load is quite small for every node, as we only compute the length of the projected diagonal, and then decide whether to mark the primitive for render or continue the traversal to the children. This means that memory access overheads will be significant and the compute throughput relatively low. Generally, DFS graph traversals are done with recursive function calls, as the implementation is more elegant and CPUs deal well with relatively large function call stacks. The CUDA framework also allows recursivity through its Dynamic Parallelism functionality. However, this is mostly used to launch a computationally-heavy kernel from an already running kernel. In the traversal case, launching multiple kernels recursively will incur a massive amount of launch overhead for a very short computation. Alternatively, the kernel can be used as a wrapper to a device function implementing recursive calls, but in my experiments, this also performed poorly, and extra care has to be taken not to overflow the call stack.

The most efficient DFS implementation I found for this application is to perform the traversal in a loop, using a stack to keep track of the traversed path. Because the trees are quite shallow in the kernel

traversal, allocating a stack of 64 elements is enough to ensure there is no overflow, while also being small enough to fit in the local thread memory for fast access. Using the NVIDIA profiling tools, this method showed the lowest percentage of cache misses and the highest compute throughput (even though the value is lower than ideal in order to take full advantage of the architecture).

The scope of this project is to provide an extension to the existing render pipeline without introducing extensive changes. The existing pipeline remains mostly unmodified, and the traversal is done in a separate kernel call before the pre-processing routine. Marking the primitives eligible for render is done through a boolean mask situated in global GPU memory. The scene tree traversal populates the mask with values, then the mask is passed to the pre-processing routine. Then, the pre-processing will quickly discard the primitives that are not marked for rendering in the mask, and the rest of the pipeline remains unchanged.

Another optimization I wanted to explore was performing frustum culling during the traversal. This can easily be performed by intersecting the node bounding box with the frustum planes and eliminating nodes that are out of view by terminating their traversal. The performance obtained from this change is quite mixed and heavily depends on what parts of the scene are in view. First of all, even though frustum intersection is trivial to compute, its computational load inside the kernel is quite significant relative to the other operations that are performed, incurring an increasing cost as the traversal goes deeper into the tree. Secondly, primitives that are out of view are removed early in the pipeline by the pre-processing routine by simply projecting the Gaussian centers to the camera plane and using a heuristic to determine if the projected centers are far enough outside the image plane to be discarded.

On camera positions that contained the most detailed parts of the scene in view, culling in the traversal did not bring any improvements, and in some cases even incurred a small overhead. The only situation when it is beneficial is when complex parts of the scene are outside the frustum, so many nodes and primitives can be removed early from the traversal by culling large nodes close to the root. Performance metrics and examples of this are discussed in the Results chapter.

9 Experimental Results

In this chapter, I will discuss the results obtained using the system developed in this project. First, I will go over intermediate results obtained during development and discuss how these influenced the decisions I took, then present the performance of the final system on multiple scenes. All the experiments that will be presented have been performed on a system running Ubuntu 23.10, with an AMD Ryzen 7 5800H, 16 GB of LPDDR4X RAM, and an NVIDIA RTX 3050 Ti Mobile GPU with 4 GB of VRAM and 2560 CUDA Cores running at 35W. The system configuration is relevant, especially the GPU used, as the performance of the renderer highly depends on the computational capabilities of the graphics card, and the size of some scenes may prevent them from being properly loaded into memory. For reference, most 3DGS publications use the NVIDIA RTX A6000 to test their implementation and generate results. That GPU has an FP32 performance of 38.71 TFLOPS, compared to the hardware I used which only achieves a theoretical maximum of 5.299 TFLOPS, and features 12x more video memory.

10 Conclusion

References

[1] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire. *Real-Time Rendering 4th Edition.* A K Peters/CRC Press, Boca Raton, FL, USA, 2018.

[2] S. Avidan and A. Shashua. Novel view synthesis in tensor space. In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1034–1040, 1997.

[3] 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. *ICCV*, 2021.

[4] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. *CVPR*, 2022.

[5] M. R. B. Clarke. Algorithm as 41: Updating the sample mean and dispersion matrix. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 20(2):206–209, 1971.

[6] Y. Dodge. *The Concise Encyclopedia of Statistics.* The Concise Encyclopedia of Statistics. Springer New York, 2008.

[7] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[8] Zhiwen Fan, Kevin Wang, Kairun Wen, Zehao Zhu, Dejia Xu, and Zhangyang Wang. Lightgaussian: Unbounded 3d gaussian compression with 15x reduction and 200+ fps, 2023.

[9] Stephan J Garbin, Marek Kowalski, Matthew Johnson, Jamie Shotton, and Julien Valentin. Fastnerf: High-fidelity neural rendering at 200fps. *arXiv preprint arXiv:2103.10380*, 2021.

[10] Hans J. Weber George B. Arfken and Frank E. Harris. *Mathematical Methods for Physicists.* Academic Press, Elsevier, Inc., 2013.

[11] Pascal Getreuer. A Survey of Gaussian Convolution Algorithms. *Image Processing On Line*, 3:286–310, 2013.

[12] Jacob Goldberger and Sam Roweis. Hierarchical clustering of a mixture model. In L. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems*, volume 17. MIT Press, 2004.

[13] Wenzel Jakob, Christian Regg, and Wojciech Jarosz. Progressive Expectation–Maximization for hierarchical volumetric photon mapping. *Computer Graphics Forum (Proceedings of EGSR)*, 30(4), June 2011.

[14] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4), July 2023.

[15] Bernhard Kerbl, Andreas Meuleman, Georgios Kopanas, Michael Wimmer, Alexandre Lanvin, and George Drettakis. A hierarchical 3d gaussian representation for real-time rendering of very large datasets. *ACM Trans. Graph.*, 43(4), jul 2024.

[16] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and temples: Benchmarking large-scale scene reconstruction. *ACM Transactions on Graphics*, 36(4), 2017.

[17] Michael Landy and J. Anthony Movshon. *The Plenoptic Function and the Elements of Early Vision*, pages 3–20. MIT Press, 1991.

[18] Joo Chan Lee, Daniel Rho, Xiangyu Sun, Jong Hwan Ko, and Eunbyung Park. Compact 3d gaussian representation for radiance field. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 21719–21728, 2024.

[19] Yixuan Li, Lihan Jiang, Linning Xu, Yuanbo Xiangli, Zhenzhi Wang, Dahua Lin, and Bo Dai. Matrixcity: A large-scale city dataset for city-scale neural rendering and beyond. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3205–3215, 2023.

[20] Yang Liu, He Guan, Chuanchen Luo, Lue Fan, Naiyan Wang, Junran Peng, and Zhaoxiang Zhang. Citygaussian: Real-time high-quality large-scale scene rendering with gaussians. *arXiv preprint arXiv:2404.01133*, 2024.

[21] Tao Lu, Mulin Yu, Linning Xu, Yuanbo Xiangli, Limin Wang, Dahua Lin, and Bo Dai. Scaffold-gs: Structured 3d gaussians for view-adaptive rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 20654–20664, 2024.

[22] David Luebke, Benjamin Watson, Jonathan D. Cohen, Martin Reddy, and Amitabh Varshney. *Level of Detail for 3D Graphics*. Elsevier Science Inc., 2002.

[23] J. MacQueen. Some methods for classification and analysis of multivariate observations. Proc. 5th Berkeley Symp. Math. Stat. Probab., Univ. Calif. 1965/66, 1, 281-297 (1967)., 1967.

[24] Manuel Martínez-Corral and Bahram Javidi. Fundamentals of 3d imaging and displays: a tutorial on integral imaging, light-field, and plenoptic systems. *Adv. Opt. Photon.*, 10(3):512–566, Sep 2018.

[25] Duane Merrill and Michael Garland. Single-pass parallel prefix scan with decoupled lookback. 2016.

[26] Duane Merrill and Andrew Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters*, 21(02):245–272, 2011.

[27] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020.

[28] Andrew Ng, Michael Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14. MIT Press, 2001.

[29] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, mar 2008.

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

[31] H. Nyquist. Certain topics in telegraph transmission theory. *Transactions of the American Institute of Electrical Engineers*, 47(2):617–644, 1928.

[32] Kerui Ren, Lihan Jiang, Tao Lu, Mulin Yu, Linning Xu, Zhangkai Ni, and Bo Dai. Octree-gs: Towards consistent real-time rendering with lod-structured 3d gaussians, 2024.

[33] Sara Fridovich-Keil and Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance fields without neural networks. In *CVPR*, 2022.

[34] Johannes Lutz Schönberger and Jan-Michael Frahm. Structure-from-motion revisited. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[35] Johannes Lutz Schönberger, Enliang Zheng, Marc Pollefeys, and Jan-Michael Frahm. Pixelwise view selection for unstructured multi-view stereo. In *European Conference on Computer Vision (ECCV)*, 2016.

[36] S. Ullman and Sydney Brenner. The interpretation of structure from motion. *Proceedings of the Royal Society of London. Series B. Biological Sciences*, 203(1153):405–426, 1979.

[37] Lee Westover. Interactive volume rendering. In *Proceedings of the 1989 Chapel Hill Workshop on Volume Visualization*, VVS '89, page 9–16, New York, NY, USA, 1989. Association for Computing Machinery.

[38] Zhiwen Yan, Weng Fei Low, Yu Chen, and Gim Hee Lee. Multi-scale 3d gaussian splatting for anti-aliased rendering, 2024.

[39] Vickie Ye and Angjoo Kanazawa. Mathematical supplement for the `gsplat` library, 2023.

[40] Zehao Yu, Anpei Chen, Binbin Huang, Torsten Sattler, and Andreas Geiger. Mip-splatting: Alias-free 3d gaussian splatting. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 19447–19456, June 2024.

[41] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Ewa volume splatting. In *Proceedings Visualization, 2001. VIS '01.*, pages 29–538, 2001.