CS488 - Final Project Report Title: 3D Gaussian Splatting Viewer

Name: Shahan Nedadahandeh
Student ID: 20928490
User ID: snedadah

1 Purpose

Create a viewer for 3D Gaussian Splatting, a new 3D world representation which brings photorealistic quality graphics to real-time rasterization based graphics, and combine this new technique with more traditional computer graphics techniques.

2 Statement

Recently, a new 3D world representation has been created, called 3D gaussian splatting. The idea is instead of meshes, we use a set of 3D gaussian to represent the world, each of which is an ellipsoid. The advantage of this representation, is there are machine learning based approaches to take photos of a scene in the real world, and convert them to a 3D gaussian splat, which look photorealistic, and yet can be rendered in real-time using rasterization. For this project, we will be focusing on how we can view these splats, which is an interesting computer graphics challenge on its own, and not the training side.

The original viewer in the 3D gaussian splatting paper uses a tile base rasterizer implemented in CUDA. However, due to this viewer being implemented in CUDA, it is not portable to different devices (for example, I can't run it in my Macbook Pro). Furthermore, the viewer does not have support for any traditional computer graphics techniques, such as meshes. Therefore, the goal for this project is to create a more cross-platform viewer, using OpenGL, which has support for both meshes, and 3D gaussian splats.

This project is interesting since not much work has been done in combining traditional computer graphics with Gaussian Splatting, and therefore, it will be an interesting exploration. Furthermore, there are many challenges in rendering the gaussian splats. For example, since each splat is translucent, and the order that the splats are rendered in is important, and this order is constantly changing when the camera view changes, performance is. Furthermore, due to the shear number of splats (often hundreds of thousands per scene) and this constant sorting, performance is a concern. Thus, there is a lot for me to learn when doing this project.

3 Technical Outline

3.1 Gaussian Representation of the world

A 3D Gaussian splat is a set of 3D gaussians, $G = \{g_1, g_2, ..., g_n\}$, where each g_i has the following properties:

- A mean $\mu_i \in R^3$
- A 3x3 covariance matrix $\Sigma_i \in R^{3x3}$
- A color $c_i \in R^3$
- An opacity $\alpha_i \in R$

Note that in the original paper, the colors are instead encoded using a set of spherical harmonics, which encodes view dependent effects. However, for this viewer, I will be only using the first order spherical harmonic, which is a constant RGB color, this is to increase the rendering speed and to simplify the project. I will be getting this color, by using the degree-0 spherical harmonics, which is calculated by: RGB $= c_0\sqrt{\frac{1}{4\pi}}$ where c_0 is the first spherical harmonic coefficient.

3.2 Sorting Gaussians according to their depth from the camera

Because splats are translucent, similar to rendering translucent sprites, the order that the splats are rendered in is important. Thus we need to sort the splats according to their depth, from back to front and render them in that order, which is Painters algorithm..

The original implementation uses a CUDA based differentiable rasterizer. However, the reason they do so is because they need their viewer to be differentiable for their training process. Since this project is only a viewer and does not support training (so we don't need our rendering process to be differentiable), we'll be using a more traditional OpenGL based rasterizer.

Every time that the camera view changes, we will be sorting the splats from back to front according to their depth from the current camera view matrix. In order to do this fast, we will to try using Radix sort to speed up the sorting.

3.3 Rendering Gaussians

Each splat will be represented by a quad, which always faces the camera, and is centered at the mean of the splat, and we'll be using OpenGL's instanced rendering feature to render all of the quads at once, without having to do a draw call for every splat.

The first step of the actual rendering, is to project the 3D gaussian (which is a a 3D ellipsoid) which has a 3D covariance matrix and 3D mean, into a 2D gaussian (which is more of 2D oval) which has a 2D covariance matrix and 2D mean. We will be doing this in the vertex shader, like follows:

3.3.1 Projecting the 3D Gaussian into a 2D Gaussian

We first project the mean of the 3D gaussian into 2D screen space:

$$\mu_{2D} = \begin{bmatrix} \frac{f_x x}{z} \\ \frac{f_y y}{z} \end{bmatrix}$$

Then we get the Jacobian of this transformation, which is

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

And we calcualte the following value, where V is the view matrix of the camera:

$$\mathbf{T} = \mathbf{VJ}$$

Finally calcualte the 2D covariance matrix in screen space:

$$\mathbf{\Sigma}_{2D} = \mathbf{T}\mathbf{\Sigma}_{3D}\mathbf{T}^T$$

The Σ_{3D} is the 3D covariance matrix, of the splat. Note that most gaussian training code doesn't output the 3D covariance matrix directly, but instead outputs a scale matrix and a rotation matrix. We will be precomputing the $3D$ covariance matrix from these by using the following $\Sigma_{3D} = RSS^TR^T$

3.3.2 Rendering the 2D Gaussian

Now that we have a 2D gaussian in screen space, in the fragment shader we actually need render the gaussian. More precisely, we will use the opacity (α_i), color (C_{ir}, c_{ig}, c_{ib}), and current (x, y) screen space coordinate to calculate the opacity and color of each pixel. (Note that the pixels will be transparent, and the blending of different gaussians together will be handled by OpenGLs blending).

The first step is to figure out the opacity of the pixel, which depends on the distance from the current pixel coordinate to the mean of the gaussian. Intuitively this makes sense because for (x,y) close to the mean the gaussian will be full opacity, however, as we get further away, the opacity will decrease, and it should be 0 our (x, y) is not in the 2D gaussian at all.

The following quadratic form represents the "Mahalanobis distance" from the current pixel coordiante, to the mean of 2D gaussian, which is what is used in 3D gaussian splatting.

$$Q(x,y) = \begin{bmatrix} x-\mu_x & y-\mu_y \end{bmatrix} \begin{bmatrix} \Sigma_{11}^{-1} & \Sigma_{12}^{-1} \\ \Sigma_{21}^{-1} & \Sigma_{22}^{-1} \end{bmatrix} \begin{bmatrix} x-\mu_x \\ y-\mu_y \end{bmatrix}$$

Next, using that value, we will calculate the full return value of fragment shader using the following equation:

$$\alpha(x,y) = \alpha_i \cdot exp(-\frac{1}{2}Q(x,y))$$

$$\mathbf{C}_{final} = \begin{bmatrix} \alpha \cdot c_r \\ \alpha \cdot c_g \\ \alpha \cdot c_b \\ \alpha \end{bmatrix}$$

,

3.4 Integrating the guassians with meshes

We'll have a seperate vertex and fragment shader for rendering normal objects, and will render them normaly as we have done in the assignments, with phong shading. (note that applying the phong shading to the gaussians is out of the scope of this project).

We will be rendering these objects first, enabling depth testing. Then we'll be rendering the gaussians, again with depth testing enabled, so any gaussians that are occluded by the mesh based objects are not rendered. However, when rendering the gaussians, we'll disable writing to the depth buffer, and instead enable blending, since each splat is translucent, and we need to blend the colors of the gaussians together, but don't want to modify the depth buffer.

3.5 Storing gaussians in an Octtree to allow fast editing

When the user clicks on the screen, we can send a ray from the camera through the clicked point on the screen, and see the closest gaussian in the scene which intersects with the ray. In order to allow fast queries of which gaussians are where, we will be storing all the gaussians in an Octtree, storing only the gaussians mean μ_i. In practice, each guassian is very small, thus we can ignore its covariance matrix when inserting it into the octtree. When computing intersections, as long the mean is within a certain distance of the ray, we can consider the ray intersecting with the gaussian, and select that gaussian, and a group of gaussians around it (again based on the mean distance being less than a certain threshold). After we have selected which gaussians we want to edit, we can edit the gaussian by allowing the user to change their opacity.

3.6 Polygon based objects casting a shadow on the gaussians

We will put an artificial light somewhere in the scene. For every splat, we will shoot a ray to the light source. If the ray intersects with a polygon based object, we will darken that splat, to signify that it is in the shadow of the polygon based object. We will darken the color of the splat by multiplying it by a factor between 0 and 1.

3.7 Polygon based objects not intersecting with the gaussians

We will be checking for intersection between the polygon based objects, and the gaussians. If there is intersection, we will move the gaussian away from the polygon based object by a small distance. We will check our intersections by creating a bounding box around the polygon object, and then querying our octree,

to see if there are any gaussians within that bounding box. Due to each gaussian being small, we will have a threshold, and only if there is over the threshold number of gaussians within the bounding box we will trigger a collision.

3.8 Combining two scenes

We will be loading two different sets of gaussians, and rendering them at the same time. We will treat one scene as the canonical coordinate space, and apply a transformation to all the means of the gaussians in the second scene, to see what the second scene, to move it around relative to the first scene.

4 Bibliography

Kerbl, Bernhard, et al. "3D Gaussian Splatting for Real-Time Radiance Field Rendering." ACM Transactions on Graphics, vol. 42, no. 4, 2023.

Idea of using an Octtree:

Zhang, Hao, et al. "Octree-GS: Towards Consistent Real-time Rendering with LOD-Structured 3D Gaussians." arXiv preprint arXiv:2403.17898 (2024).

While my project doesn't have robotics, I've found this survey paper is very useful for understanding gaussian splatting in general:

Zhu, Siting, et al. "3D Gaussian Splatting in Robotics: A Survey." arXiv preprint arXiv:2410.12262 (2023).

This viewer does a similar rendering process as I want to do, and the github readme was useful for inspiration:

"splat." GitHub, antimatter15, 2024, github.com/antimatter15/splat.

Objectives:

Full UserID: snedadah **Student ID: 20928490**

___ 1: The scene is made of 3D gaussian splats, and there is a toggle which allows showing all the individual splats.

___ 2: The gaussian splats are alpha blended together, and there is toggle which can turn blending on and off.

___ 3: The scene is sorted in back to front order in realtime every time the camera view changes, and there is a toggle which turns this off for the purpose of demonstration.

___ 4: The splats are stored in an octtree, which can be turned on and off. You can see the performance difference of editing splats when having the octtree and not having it, and there is a toggle to visualize the octtree.

___ 5: There is a delete button, which allows the user to click on a certain point on the screen, and delete gaussians close to that point.

___ 6: You can put polygon based meshes into the scene, which are correctly blended with the gaussian splats.

___ 7: Polygon based meshes cast shadows on to the gaussian splats.

___ 8: Polygon based objects will not intersect with the gaussian splat, and there is some visual indication when there is a collision.

___ 9: Two gaussian splat scenes can be combined and rendered at the same time.

___ 10: The starting scene is interesting, having many diverse objects of different scales and colors.