CS488 Final Project - Interactive 3D Gaussian Splatting Viewer

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

1 Purpose

The goal of this project was to 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, such as OctTrees, mesh based objects, shadows, etc.

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 (a 3D gaussian is pretty much each just 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, I created a viewer for these splats, which is an interesting computer graphics challenge on its own, and did not consider the actual training of the gaussians.

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 was 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, my work is very novel. 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.

3 Technical Implementation

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, In my viewer, only used the first order spherical harmonic, which is a constant RGB color, this is to increase the rendering speed and to simplify the project. I calculated this 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 frame before we render the splats, we sort the splats from back to front according to their depth from the current camera view matrix. In order to do this fast, I used counting sort to sort the splats, which runs in linear time. By reducing the number of buckets avaible to counting sort, I sped up the sorting, at the expense of slightly less accurate rendering order. See the function `sort_splats_based_on_depth` to see to see the sorting implementation.

3.3 Passing a lot data to the shaders

Originally, I had a buffer for every splat parameter. So one buffer for all the means of the splats, one for all the 3D covariance matrices, one for all the colors, etc. However, because the splats need to be rendered in back to front order, and this order changes every frame, all of these buffers needed to be updated every frame, which I found was incredibly slow, since there are tens of thousands of splats.

To fix this, instead I now only pass a single buffer to the shaders, which contains a list of "indexes" of splats which need to be rendered. For example, the buffer could have [4, 1, 3], which would mean that the splat at index 4 has to be rendered first, then the splat at index 1, then the splat at index 3. Now when I sort, I only need to update this index buffer every frame.

However, the new problem is the OpenGL buffers don't allow random access, so if at the first iteration the index was 4, I couldn't reach into the 4th index of the mean buffer to get the mean of the splat at index 4. Therefore, I instead stored the actual data of all the splats, in data textures, which are just normal textures, where every pixel represents a value of a splat. For example, I have a texture for all the means, and another texture for all the 3D covariance matrices, etc. See the function `update_webgl_textures` in the `renderer.rs` file to see how I send data to these textures. Then, since textures do allow random access, I can use the index (for example 4) to get the value of the mean of the splat at index 4, by looking at pixel 4 in the mean texture.

This way, I am able to reorder the rendering order of the splats every frame, without having to update the buffers, and without having to pay the overhead of updating the textures every frame. Only one buffer (the index buffer) needs to be updated every frame, and the textures only need to be updated when the splats change (which is rare).

3.4 Rendering Gaussians

Each splat is be represented by a quad, which always faces the camera, and is centered at the mean of the splat, and I use 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. This is done in the vertex shader, like follows:

3.4.1 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 $\Sigma_{3D} = RSS^TR^T$

3.4.2 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 calculate the full return value of fragment shader using the following equation:

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

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

3.5 Integrating the guassians with meshes

I have separate vertex and fragment shader for rendering normal objects, and I have implemented phong shading to shade these objects.

One key detail is how to seamlessly combine the guassians with the mesh objects. The way I do this, is to first render the splats and populate the depth buffer. Note that depth testing is disabled when rendering splats, since we do blending when rendering the splats, but we still want to populate the depth buffer. Furthermore, because of the weird way the splats are rendered, I can't just rely on OpenGL to auto populate the depth buffer, and instead have to manually set the depth buffer by setting the `gl_FragDepth` in the splat fragment shader. Then, mesh based objects are drawn second, this time with depth testing enabled,

and thus when a splat has a depth closer to the camera, the mesh object will be occluded. This approach seems to be working well for some scenes, however, in some other scenes, it seems to be that the values that the Splat depth fills in the depth buffer doesn't perfectly line up with the depth when rendering the mesh objects, and this causes some artifacts. Such as the mesh object being partially visible through the splat. I ran out of time to fully debug this issue.

3.6 Storing gaussians in an OctTree

For my next 3 objectives, deleting splats, doing collisions, and casting shadows, its incredibly helpful to be able to check at a specific world position, what are the splats near by. However, normally since the splats are just stored in a list, this operation is $O(n)$ in the number of splats. Which is costly since scenes can have thousands to hundreds of thousands of splats. Therefore, I explored storing all the splats inside of an OctTree, which brings this operation down to $O(\log n)$ in the number of splats, and greatly speeds up those other operations.

The actual OctTree is implemented in the `oct_tree.rs` file. My OctTree essentially treats the splats as points (using only the means), which works since each splat is very small, so we can ignore the 3D shape of them.

In the start, after we load the scene, we insert all of the splats into the OctTree. The OctTree is defined recursively. Every node in the OctTree either is a child, meaning it stores a list of splats, or it is a parent, meaning it stores a list of children. The splats are stored in a simplified manner, with only the mean, opacity, and index in the original scene, to save memory. The function `propogate_splats_to_children` is called on every node. If the node has more than a certain number of splats inside, and it will split the node into 8 children, and recursively propagate the splats to the children. The mean of the Splat is the only parameter that decides which node the splat will be in.

The OctTree has two parameters, `SPLIT_LIMIT`, which is the number of splats a node can have before it splits. So if a node has less than `SPLIT_LIMIT` splats, it will not split into children. The other parameter is `MAX_DEPTH`, which is the maximum depth of the OctTree, and this takes precedence over the `SPLIT_LIMIT`, because I found sometimes, even with many splits, the tree was still too deep, and I ran into recursion stack limits.

3.6.1 Visuzlizing the OctTree

The OctTree is visualized by recursively. If a node has children, then only the children are drawn. Else, the boundary of the node is drawn by drawing lines around the node. These lines are drawn using the normal geometry shader, which I used to draw the normal objects, with depth testing turned off.

3.6.2 Quering the OctTree for collisions

The OctTree is queries with the `find_splats_in_radius` function. This function takes in a center point, and a radius, and returns all the splats that are within that radius from the center point. It does so by checking if the node is outside the radius, and if so, it will not search in that direction, and if the node is inside the radius, it will recursively search its children. If the node is a leaf node, it will check all the splats in that node to see which ones are within the radius. I have a special mode Only Show OctTree at Click, which when enabled, will only draw the OctTree nodes that were touched in the last query (ALT + Click on a splat to query), which shows the power of the OctTree.

3.6.3 Deleting Splats using the OctTree

When the user holds down ALT and clicks on the screen, we send a ray from through the clicked point on the screen. At several points along this ray, we query the OctTree, and see if there are any splats near this point, with a significant opacity. If there is, we consider this point to be the intersection point, and then we query to OctTree for a larger radius of splats near this point. After we have received these splats, we set all of their opacities to 0, and we recreate our data textures.

3.7 Polygon based objects casting a shadow on the Gaussians

When the user clicks the "Cast Shadows" button the function `calculate_shadows` in `scene.rs` is called. The basic ideas is to loop through every single splat, and check if we can shoot a ray from the this splat to the light source, without it being intersected by any of the polygon based objects. To check if the ray intersects with a polygon based object, I reused my ray tracing intersection code from A4. If the ray does interesect with an object before hitting the light, then we make the color of this spalt darker, to signify that it is in the shadow of the polygon based object.

While this first approach worked, it was incredibly slow, since looping through every splat was expensive. However, I realized that since my OctTree nodes were pretty finegrained, if the center of the OctTree was not in a shadow, it was unlikely that any of the splats in that node would be in a shadow. Thus, now, I first query the OctTree for all the nodes that are sufficiently fine grained (have less than 10 splats inside of them). And for these nodes, I only check if the center of this OctTree node is under shadow, and if it is, I darken all the splats in this node. Essentially, we are using the fact that if one splat is under shadow, the splats near it are also likely under shadow. I found that this approach significantly speeds up the shadows (around a 5x speedup), while compromising the quality of the shadows only slightly. However, the shadows are still pretty slow, thus there is room for improvement.

3.8 Polygon based objects not intersecting with the gaussians

To check for collisions between the polygon based objects, whenever a polygon based object moves (which happens if the "Move Down" checkbox is checked or if the Detect Collisions With Gizmo option is checked), we calcualte two bounding points on the object (based on the min vertex of the object and max vertex of the object), and we query the OctTree for all the splats that are within a certain radius of these two points. If there are more than a certain threshold number of splats, which have high opacity, we will consider that the polygon based object is colliding with the splats.

3.9 Combining two scenes

In order to add one splat to another splat, I first load the other splat, and then I apply a transformation to all the means of the new splat. Then I simply append it to the end of the splat list of the first splat. See the `merge_with_other_splatdata` function in `data_objects.rs` for the implementation. and `apply_transformation_to_object` in `scene.rs` for the transformation application. This is what happens when the "Add Shahan" button is clicked.

3.10 Speeding up opening the viewer (Rkyv)

Originally, I read the point cloud file (.ply) file that the splat comes in, inside the viewer. (See the `loader.rs` file). The problem with this is that reading the ply file is incrediblly slow, which greatly slowed down the opening of the scene. To fix this, I now precompute the loaded ply file in a serialized format (see the `local.rs` file). Using the rust library "Rkyv", I serialize and deserialize the actual "SplatData" struct, which contains all the information about a splat in a binary format. So, when the viewer opens, it simply downloads this file, and is ready to go.

3.11 Uploading to web

The code is all written in Rust, however, I wanted the project to be very portable, so I used WebGL to render the project, and webassembly to compile the rust so that it runs in the browser. The actual user interface is written in HTML and CSS, which can be seen in the `index.html` file.

3.12 Picking normal objects

To pick the normal objects and object movement Gizmo, I render a special picking image, similar to A3, see the `draw_picking_image` function in the `renderer.rs` file.

3.13 Code Walkthrough

- `web.rs`: Contains the main code of the project, `start()` is the entry point of the program.
  - See `handle_splat_delete_click` for what happens when you hold down ALT and click on the screen.
- `renderer.rs`: Does all the WebGL rendering.
  - See `draw_splat` for the function that does the actual rendering of the splats.
  - See `update_webgl_textures` for how I update the textures with the splat data.
- `loader.rs`: Contains the code for loading the splat data from the .ply file.
- `oct_tree.rs`: Contains the OctTree data structure, and the functions for querying the OctTree for splats.
- `scene.rs`: Contains the data structure for the scene
  - See `calculate_shadows` for how I calculate the shadows.
  - See `update_gizmo_drag/move_down` for how I check for collisions between the polygon based objects and the splats.
- `data_objects.rs`: Contains the actual data structures for the splats and the polygon based objects.
  - See `sort_splats_based_on_depth` for how I sor the splats in back to front order.
  - See `merge_with_other_splatdata/apply_transformation_to_object` for how I merge two splat scenes together.

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 (Though their approach is very diferent, they actually need to adjust the Splat ML training process to work with the OctTree, while I don't):

Zhang, Hao, et al. "OctTree-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).

I took inspiration from many different online viewers to come up with my rendering approach, especially for **the splat rendering shaders**. Note that none of these viewer have near the amount of features, for example, multiple splats, shadows, collisions, OctTree's, removal of splats, polygon meshes, etc. All of these are unique to my project, I only referenced these for the inspiration on how to do the actual gaussian splatting rendering.

- https://github.com/playcanvas/supersplat
- https://github.com/mkkellogg/GaussianSplats3D

- https://github.com/nerfstudio-project/gsplat
- https://github.com/kishimisu/Gaussian-Splatting-WebGL
- https://github.com/graphdeco-inria/diff-gaussian-rasterization/blob/main/cuda_rasterizer

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.