

Tetrahedral Volume Rendering

Project Report

Presented in Partial Fulfillment of the Requirements for the Degree Master
of Science in the Graduate School of The Ohio State University

By

Musab Fiqi, AS, BS.CSE

Graduate Program in Computer Science and Engineering

The Ohio State University

2024

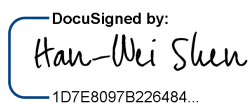
Master's Examination Committee:

Hanqi Guo, Advisor

Han-Wei Shen, Member

DocuSigned by:

68D5A52B16A4497...

DocuSigned by:

1D7E8097B226484...

© Copyright by

Musab Fiqi

2024

Abstract

This project explores the use of WebGPU, a new graphics API, for direct volume visualization of tetrahedral meshes. This project began by the need to efficiently render complex, unstructured datasets using WebGPU's feature set to implement direct volume visualization for a tetrahedral mesh. The renderer traverses the tetrahedral mesh sequentially, accumulating color and alpha values. This continues until the ray exits the mesh. A custom triangle-tetrahedron map data structure facilitates efficient mesh traversal. This report details the development process, including challenges encountered due to WebGPU's changing nature and limitations in its feature set, such as the lack of WebGL's `gl_PrimitiveID` in the fragment shader. The renderer is capable to visualize the turbulence around a golf ball, but currently faces performance limitations when handling large meshes due to CPU-intensive pre-processing steps. Future work will focus on moving these pre-processing steps to WebGPU compute shaders to improve performance and enable the visualization of larger, more complex datasets.

Acknowledgments

Thanks for paying my tuition Mom.

Vita

January 2018 - December 2019 Associates of Science,
Columbus State Community College,
USA.
January 2020 - May 2023 Bachelors of Science,
Computer Science and Engineering,
The Ohio State University, USA.

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

	Page
Abstract	ii
Acknowledgments	iii
Vita	iv
List of Figures	vii
1. Background - Volume Rendering	1
2. Volume Rendering with Structured Grid Data	4
2.1 Motivation	4
2.2 Background	4
2.3 Implementation	5
2.4 Next Steps	7
3. Tetrahedral Volume Rendering of Unstructured Grid Data	8
3.1 Background	8
3.2 Design	8
3.3 Implementation	13
3.4 Issues with Rendering Final Results	15
4. Future Work	17
5. Conclusion	18
5.1 Summary	18

Bibliography 19

List of Figures

Figure	Page
2.1 Voxel Grid	5
3.1 Program Overview	9
3.2 Surface mesh representation of tetrahedral mesh	10
3.3 Coloring a triangles of a single tetrahedron based off of its barycentric coordinates	11
3.4 Barycentric coordinates for each triangle of the entire mesh	12
3.5 Intersections that indicate tetrahedra traversal	12
3.6 Turbulent airflow around a golf ball	15
3.7 Smaller portion of Figure 3.6	16

Chapter 1: Background - Volume Rendering

Volume Rendering

Scientists often want to learn insight about the data they generate, and a popular way to do so is through visualization, specifically volume rendering. Using techniques developed in computer graphics such as ray tracing, global illumination, shadows, etc., scientists can detect useful patterns in their datasets that they wouldn't have known otherwise. Typically, this is done through marching a ray through a structured grid of voxels that make up the volume. For unstructured data, we can use tetrahedral marching, which marches a ray through a tetrahedral mesh accumulating the color and alpha channels of the volume. The mesh is traversed by going from one tetrahedron to the next, until the ray exits the mesh.

WebGPU is a new graphics API for the web, replacing the now aging WebGL. WebGPU is an abstraction layer on top of the newer graphics APIs such as DirectX12, Vulkan, and Metal. Depending on the underlying hardware, WebGPU will default to using one of these three. WebGPU provides compute shaders which allow access to the GPU for non-graphics workloads (GPGPU). The goal of this project is to take advantage of the new features this graphics API can provide to accelerate marching a ray through a tetrahedral mesh.

Volume Rendering Math

We need to model how rays from a light source are absorbed, emitted, and scattered by a medium. To do so, we use what is known as the emission-absorption model. This model only computes lighting for rays that hit the target directly. Rays passing through the volume are weakened due to the medium [3].

$$C(r) = \int_0^L C(s)\mu(s)e^{-\int_0^s \mu(t)dt} ds$$

As the ray passes through the volume, we integrate the emitted color $C(s)$ and absorption $\mu(s)$ at each point s along the ray. The emitted color at each point is weakened as it returns to the eye by the volume's absorption up to that point ($e^{-\int_0^s \mu(t)dt}$) [3].

We approximate the integral using a numeric approximation. This is done by taking a set of N samples along the ray on the interval $s = [0, L]$, each a distance Δs apart, and summing the samples together. The weakening term at each sample point becomes a product series, adding the absorption at previous samples [3].

$$C(r) = \sum_{i=0}^N C(i\Delta s)\mu(i\Delta s)\Delta s \prod_{j=0}^{i-1} e^{-\mu(j\Delta s)\Delta s}$$

We can approximate the attenuation term $e^{-\mu(i\Delta s)\Delta s}$ by its Taylor series. We also introduce alpha $\alpha(i\Delta s) = \mu(i\Delta s)\Delta s$. Front-to-back alpha compositing equation:

$$C(r) = \sum_{i=0}^N C(i\Delta s)\alpha(i\Delta s) \prod_{j=0}^{i-1} (1 - \alpha(j\Delta s))$$

The above equation works like a for loop. It accumulates color and opacity iteratively. The loop continues until the ray either leaves the volume, or the accumulated color has become opaque ($\alpha = 1$). The iterative computation of the sum is done using front-to-back compositing equations [3]:

$$\hat{C}_i = \hat{C}_{i-1} + (1 - \alpha_{i-1})\hat{C}(i\Delta s)$$

$$\alpha_i = \alpha_{i-1} + (1 - \alpha_{i-1})\alpha(i\Delta s)$$

To render an image of the volume, we need to trace a ray from the eye through the volume, and perform the above iteration for each ray intersecting the volume. We implement the ray marching process in the fragment shader where each ray (pixel) is independent of one another.

Chapter 2: Volume Rendering with Structured Grid Data

2.1 Motivation

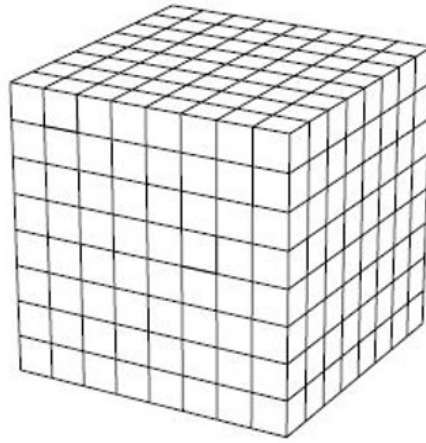
Before working with unstructured grid data, implementing a standard volume renderer with structured data would cover most of the fundamentals behind volume rendering. This includes setting up the WebGPU canvas, scene, and choosing a software design pattern that would be used for the project [1]. The goal of this portion of the project is to have all of the basic systems of volume rendering such as ray-marching, color and alpha accumulation, and overall setup of the scene ready for tetrahedra volume rendering.

2.2 Background

To implement structured volume rendering, you first setup the block of data as a set of voxels that you want to march through. Each voxel (volumetric pixel) contains a data point from the dataset. With the camera position as the origin of the ray, we first detect if that ray intersects with the structured grid data. Once an intersection is detected, we want to march the ray through the cube, gathering information about each voxel as we do so. This information is accumulated to a final color and alpha values which is then returned by the fragment shader.

As we march the ray through the structured volumetric data, we use trilinear interpolation to determine which voxel the ray has collided with. WebGPU supports 3D textures using

the textureSample function, which does the trilinear interpolation for us. We then find the initial color and alpha values based on a given voxel and transfer function. We then blend the total color and alpha values with the color and alpha value at a specific voxel [5].



8x8x8 resolution orthogonal (uniform) voxel g

Figure 2.1: Voxel Grid

2.3 Implementation

Listing 2.1: **Ray Volume Bounds Intersection Test** [5]

```
// Step 1: Normalize the view ray
var rayDir: vec3<f32> = normalize(fragmentInput.ray_direction);

// Step 2: Intersect the ray with the
// volume bounds to find the interval
// along the ray overlapped by the volume.
var t_hit: vec2<f32> =
    intersect_box(fragmentInput.eyePosition, rayDir);
var t_enter = t_hit.x;
var t_exit = t_hit.y;
// So we don't sample voxels behind the eye
t_enter = max(t_enter, 0.0);
```

Listing 2.2: **Compute the step size to march through the grid** [5]

```

var dt_vec: vec3<f32> =
    1.0 / (vec3<f32>(grid_size) * abs(rayDir));
var dt_scale: f32 = 1.0;
var dt: f32 = dt_scale * min(dt_vec.x, min(dt_vec.y, dt_vec.z));

```

Listing 2.3: **Calculate starting point and initial color** [5]

```

var p: vec3<f32> = fragmentInput.eyePosition + t_enter * rayDir;
var color: vec4<f32> = vec4<f32>(0.0,0.0,0.0,0.0);

```

Listing 2.4: **Traversing the volume** [5]

```

for (var t = t_enter; t < t_exit; t += dt) {
    // Sample the volume
    var val: f32 = textureSampleLevel(volume, tex_sampler, p, 0.0).r;
    var val_color: vec4<f32> = vec4<f32>(textureSampleLevel(
        colormap, tex_sampler, vec2<f32>(val, 0.5),0.0).rgb, val);

    // Opacity correction
    val_color.a = 1.0 - pow(1.0 - val_color.a, dt_scale);

    // Accumulate the color and opacity
    var tmp: vec3<f32> = color.rgb +
        (1.0 - color.a) * val_color.a * val_color.xyz;
    color.r = tmp.r;
    color.g = tmp.g;
    color.b = tmp.b;
    color.a += (1.0 - color.a) * val_color.a;

    // Breaks out of the loop when the color is near opaque
    if (color.a >= 0.95) {
        break;
    }
    p += rayDir * dt;
}

return vec4<f32>(textureSample(volume, tex_sampler, p).rgb, 1.0)

```

2.4 Next Steps

With the logic for volume rendering structured data complete, this sets up all the basic components for starting a tetrahedra volume renderer for unstructured data. I've learned how to setup the graphics pipeline in WebGPU. This includes setting up the GPU bindings, vertex buffer, index buffers, and depth buffer. I've also setup the color and alpha accumulation portions and added a simple transfer function.

For a tetrahedral volume renderer however, many things will need to change. The ray marching step will also change from marching through a structured grid of voxels, to connected tetrahedra. To do so, we need to determine the exit face for each tetrahedron and check if that face is connected to another tetrahedron. If so, we need to march towards the next tet until the ray exits the mesh entirely. Lastly, instead of checking for ray-box intersection, we need to check for ray-triangle intersection.

Chapter 3: Tetrahedral Volume Rendering of Unstructured Grid Data

3.1 Background

Depending on the field, most data a scientist can collect can be in the form of unstructured data. Unstructured data is data that does not have any uniform structure. Examples of this kind of data include images, audio, and video files. Because a lot of data a scientist can generate can be unstructured, tetrahedral volume rendering arose as method for this sort of data. Instead of voxels, rays are marched through tetrahedral elements. The fundamentals of volume rendering of structured grid data remain the same.

Tetrahedral marching algorithms leverage the simplicity of tetrahedral elements to efficiently traverse and render the volumetric dataset. This is done by iteratively marching through the tetrahedral elements, computing the intersection between rays and tetrahedra, then accumulating the color and alpha channels along the way [4].

3.2 Design

First, I needed to be able to read the vertices and indices off of a tetrahedral mesh that's stored in a .VTU file that is readable by ParaView. Once I have the basic information of the tetrahedral mesh as a list of vertices and indices, the file is uploaded to a Github data server. My program starts by fetching the required data from Github, the performs the pre-processing steps before finally rendering the result.

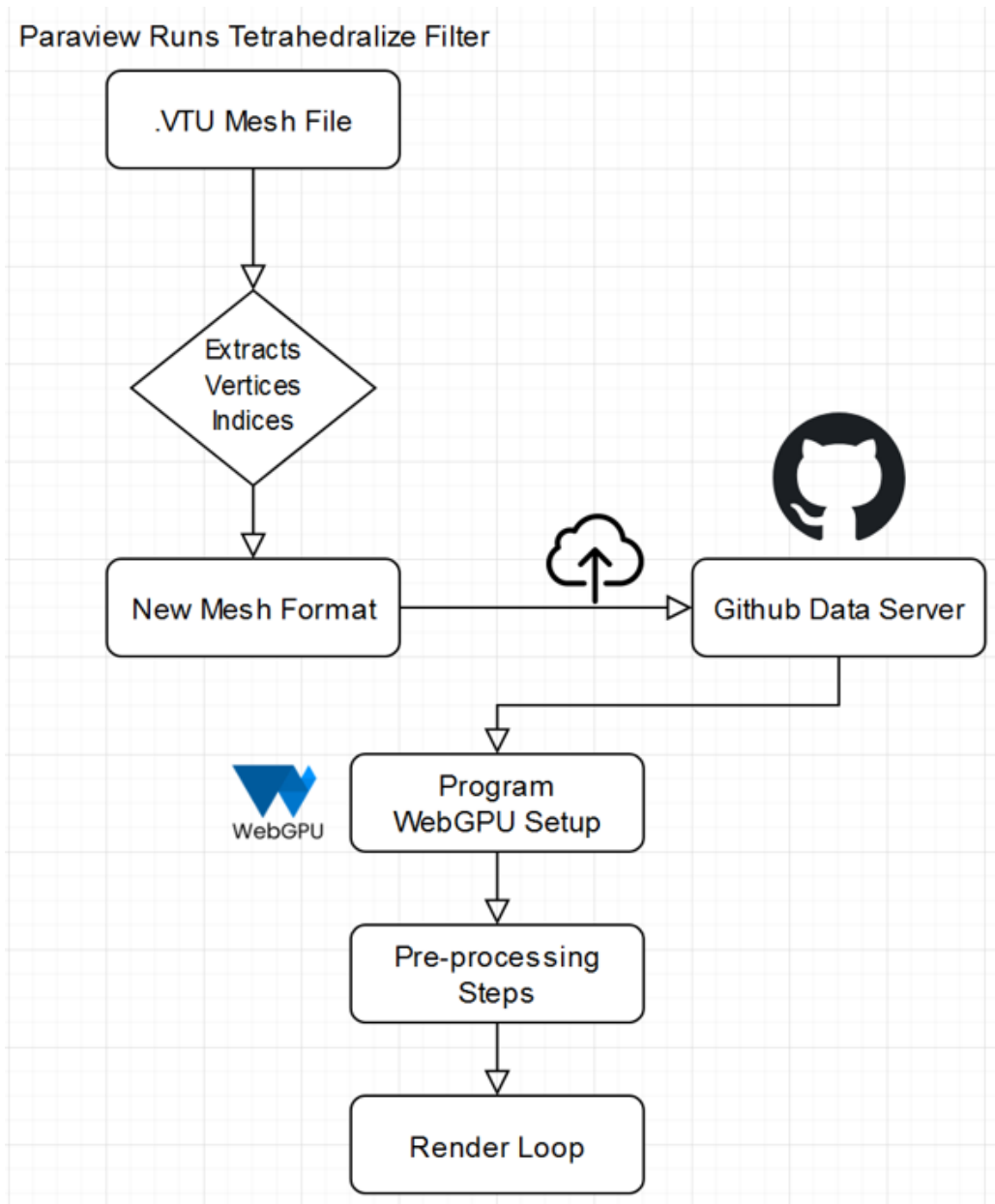


Figure 3.1: Program Overview

To initially visualize the data, the tetrahedral mesh needed to be represented as a triangle mesh. I extracted the shell to render the mesh as a surface, then assigned each triangle a random color to easily differentiate them.

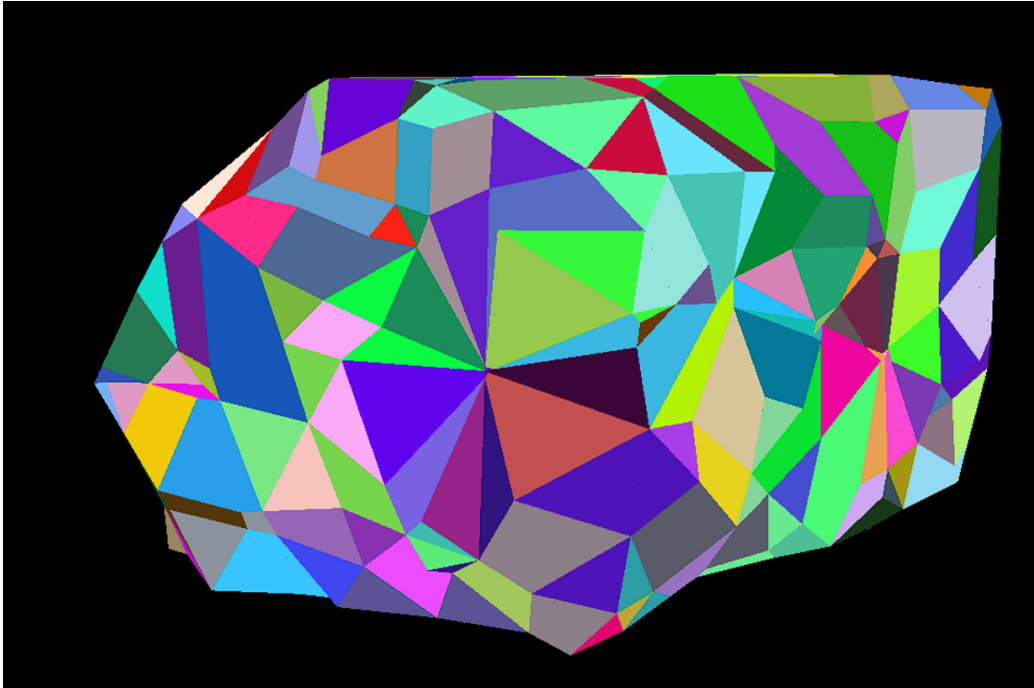


Figure 3.2: Surface mesh representation of tetrahedral mesh

In order to determine the triangle currently the fragment shader is currently working on, I developed a way to determine the triangle based off of the vertex. In order to do so, each vertex must be unique (no sharing of vertices). This would have been a trivial task in WebGL where you have access to the `gl_PrimitiveID` (contains the index of the current primitive).

```
vertexOutput.triangle_id = u32(ceil(f32(vertexInput.v_id) * 0.33333));
```

Once the triangle ID has been determined, it's passed to the fragment shader with no interpolation since we don't want it changing per pixel.

Ray casting along a tetrahedral mesh depends on ray-connectivity. Tetrahedrons are neighbors if they share a single face. This makes a continuous path of tetrahedra to traverse. If they do, we can traverse them until we reach the exit face. If a face does not connect two

tetrahedra, then that face is a shell-face. Once an exit shell-face is detected, the ray exits the mesh completely. Along the way, we accumulate the color and alpha values.

Once you determine an intersection occurs with a face, you calculate the barycentric coordinates. These values are used to interpolate and blend datapoints across the surface of a triangle.

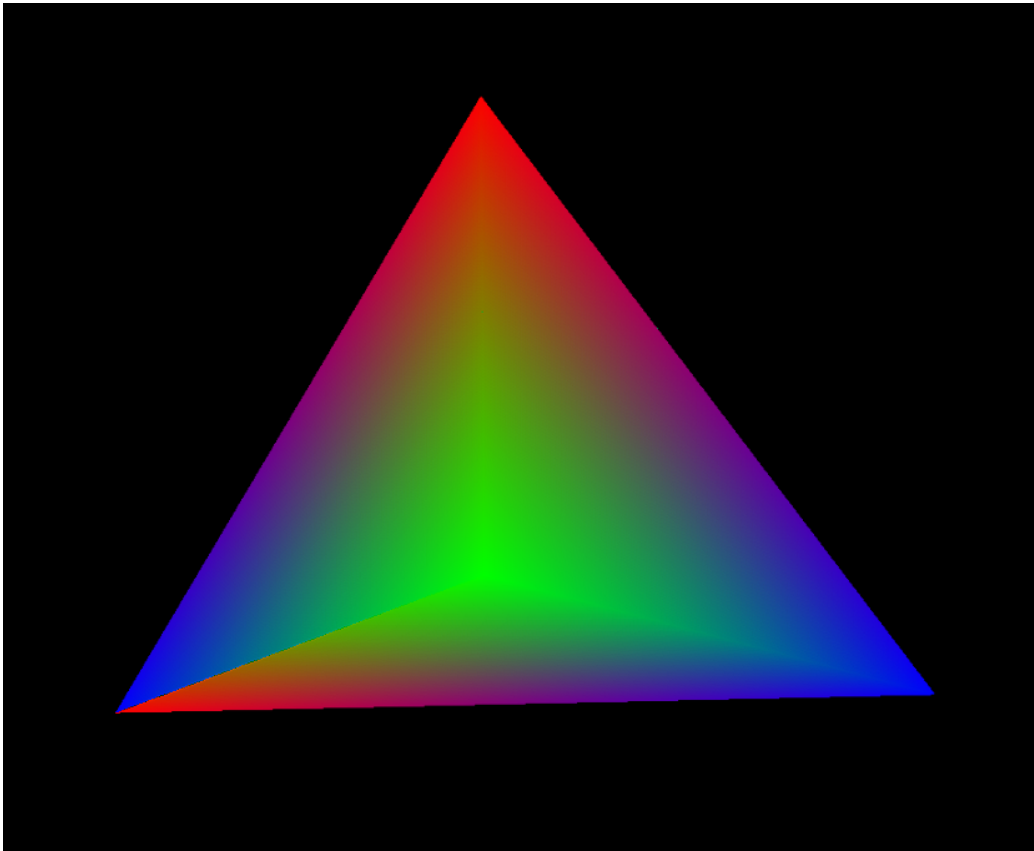


Figure 3.3: Coloring a triangles of a single tetrahedron based off of its barycentric coordinates

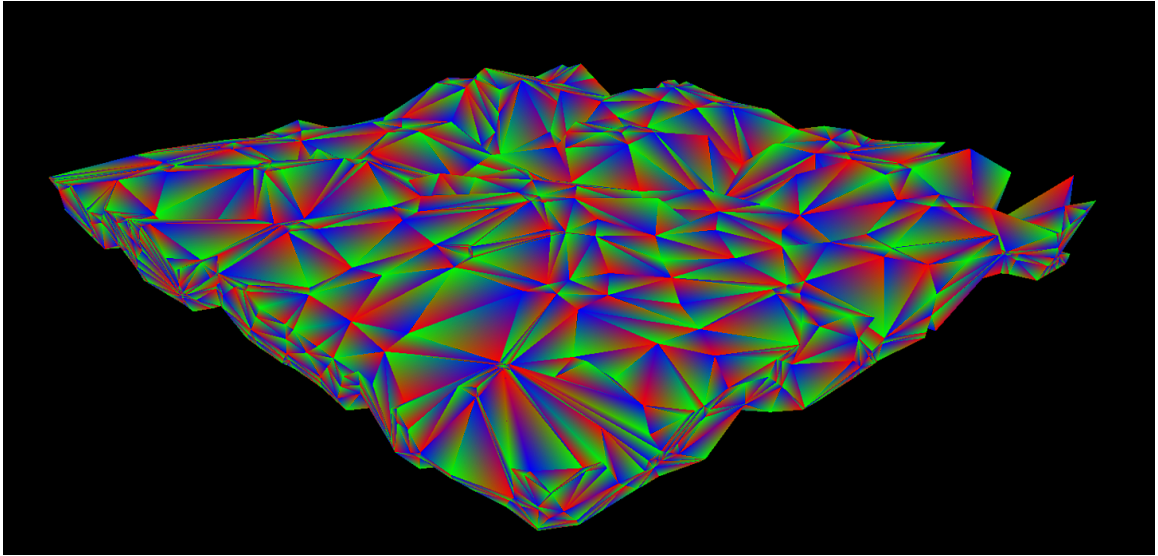


Figure 3.4: Barycentric coordinates for each triangle of the entire mesh

In order to determine if tetrahedra traversal is working correctly, I counted the number of intersections when traversing a tetrahedral mesh.

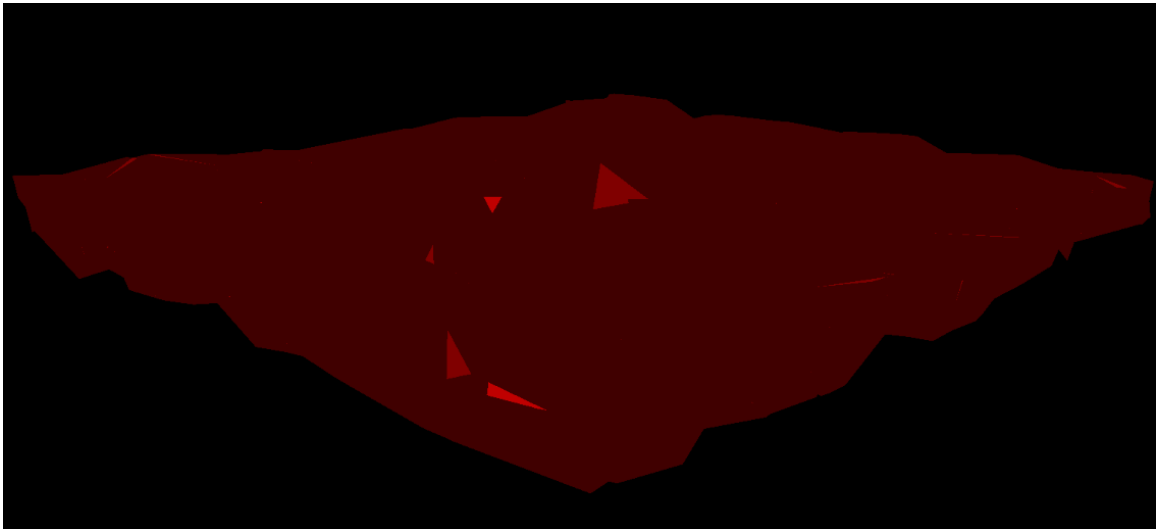


Figure 3.5: Intersections that indicate tetrahedra traversal

3.3 Implementation

In the vertex shader, I calculated the `triangle_ID` based off of the unique vertex.

```
vertexOutput.triangle_id = u32(ceil(f32(vertexInput.v_id) * 0.33333));
```

The `flat` keyword is used because we do not want to interpolate the position of the camera, nor the triangle ID.

Listing 3.1: Input to the fragment shader

```
struct FragmentInput {
    @builtin(position) pixel: vec4<f32>,
    @location(0) ray_direction : vec3<f32>,
    @location(1) @interpolate(flat) eyePosition : vec3<f32>,
    @location(3) @interpolate(flat) triangle_id : u32
};
```

The next step is to calculate the barycentric coordinates based off of the vertices of the triangle, the origin, and the normalized ray direction. The barycentric coordinates are multiplied with the function data. Afterwards, it is passed to a transfer function and we obtain a specific color value for that point in the tetrahedral mesh. We accumulate the color and alpha channels (same as voxel direct volume rendering) while traversing the tetrahedral mesh. The final accumulated color and alpha values are returned and presented in the graphics window.

Listing 3.2: Tetrahedral Mesh Traversal Psuedocode

```

var accum_color: vec3<f32> = vec3<f32>(0.0,0.0,0.0);
var accum_alpha: f32 = 0.0;
while (true) {
    accum_color += calculateColor();
    accum_alpha += calculateAlpha();

    triangle_id =
        find_exit_triangle(triangle_id, tetrahedron_id, O, D);

    // if next tet is -1,
    // the ray has exited the mesh
    var tetID: i32 = find_next_tetrahedron(triangle_id);
    if (tetID == -1) {
        break;
    } else {
        tetrahedron_id = u32(tetID);
    }
}
return vec4<f32>(accum_color, accum_alpha);

```

3.4 Issues with Rendering Final Results

The tetrahedral renderer was developed to visualize the turbulent airflow around a golf ball. One of the biggest issues I've been having with the renderer is to upload the data to my program. Currently, the datasets are all stored on a github repository for my project. It reads the raw text file that my VTK python program generates and extracts the relevant data from it such as vertices, indices, and turbulence values. The bottleneck in this process comes from both reading in the data from the server and doing the pre-processing step where my program ensures each vertex is unique. These pre-processing steps slow down the program significantly.

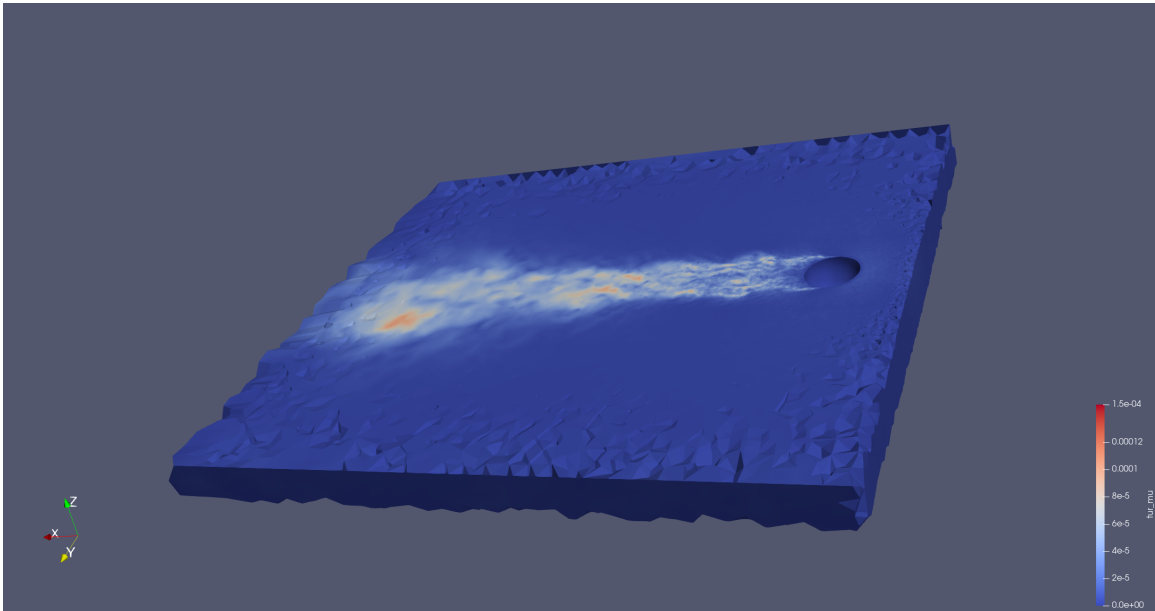


Figure 3.6: Turbulent airflow around a golf ball

I've chosen a much smaller portion of the whole mesh to render as a result of how my program reads the mesh from the repository. As I will mention in the future work portion,

most of the pre-processing steps that get the mesh ready for the ray marching process could be moved to the compute shaders.

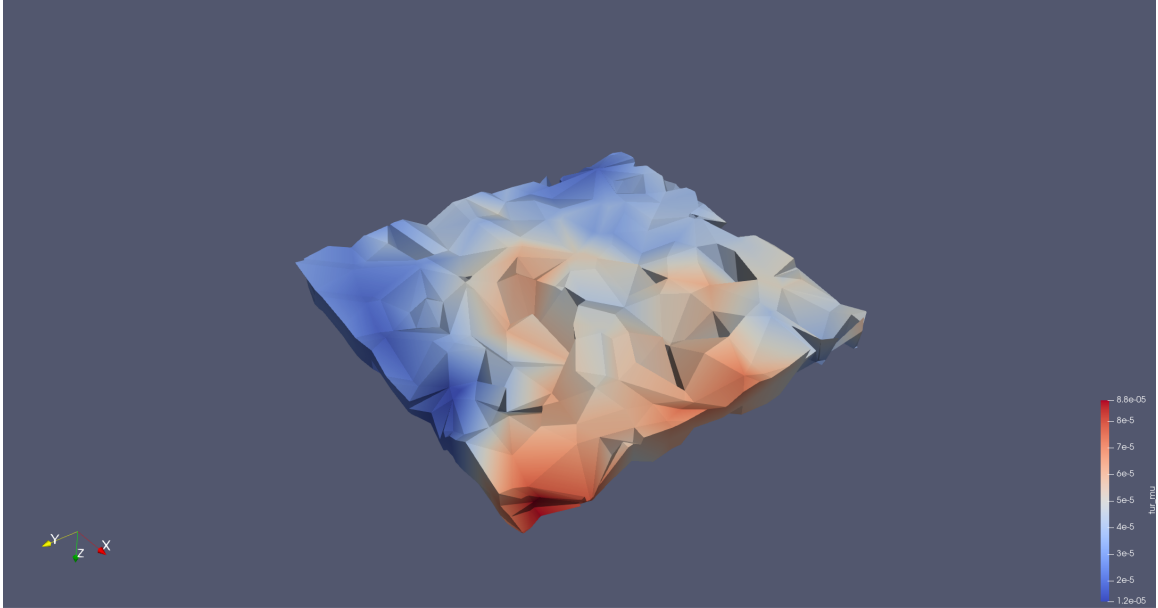


Figure 3.7: Smaller portion of Figure 3.6

Some issues I'm having is how I'm traversing this mesh. This is likely due to how I'm moving from one tetrahedron to the other. Another issue is that this smaller mesh was taking too long for my program to render the final results on screen. I suspect this might be because of how many times the ray triangle intersection test is performed. Including a bounding volume hierarchy is something that could significantly speed up this portion of the program. Once these two issues are resolved, the tetrahedral renderer would be working as expected.

Chapter 4: Future Work

Improving performance would be done by moving setup tasks to the compute shaders. For example, the creation of unique vertices and unique indices could be moved. The function that handles extracting the shell for rendering of the surface of the tetrahedral mesh could also be moved. Adding support for a bounding volume hierarchy (BVH) tree would improve ray-triangle intersection performance.

I would also want to add a light source as to illuminate and make the renderer look more realistic. Once a light source has been added, the next step would be to increase the rendering quality by adding effects such as gradient, volumetric shadows, and ambient occlusion. While my project has been focused on visualizing a static mesh, I wouldn't mind seeing if there's a way I could add a feature for fluid and gas simulations.

Finally I would like to add a feature where scientists would be able to upload their own data and run it through the renderer to see their results. I would include sliders for different features such as turning shadows on and off, increasing light strength, and being able to adjust the color map for the transfer function.

Chapter 5: Conclusion

In this chapter, I summarize the status of the work presented in this report, and outline future plans [2].

5.1 Summary

I implemented a volume rendererer for strucutred data. Marched a ray through a series of voxels, and accumulated the color and alpha channels based off of that data. Once complete, I used it as a template to start developing the tetrahedral volume renderer, but had signi Setting up the data currently requires a series of computationally expensive pre-processing steps. Most of these steps would be resolved by moving them to the compute shaders.

Both datasets I worked with were relatively small, so I wasn't testing performance. The real next step for this project (after dealing with the numerous performance issues) would be to implement one of the efficient ways of encoding the information of a tetrahedron, thus increasing performance [4].

Bibliography

- [1] amengede. webgpu-for-beginners. <https://github.com/amengede/webgpu-for-beginners>.
- [2] Swarnendu Biswas. Ohio State College of Engineering MS/PhD Dissertation Template. <https://github.com/swarnendubiswas/ohio-state-coe-dissertation-template>, August 2016.
- [3] Klaus Engel, Markus Hadwiger, Joe Kniss, Christof Rezk-Salama, and Daniel Weiskopf. Real-time volume graphics. A K Peters, <http://www.real-time-volume-graphics.org/>, July 24, 2006.
- [4] Alper Sahistan, Serkan Demirci, Nathan Morrical, Stefan Zellmann, Aytek Aman, Ingo Wald, and Uğur Gündükbay. Ray-traced shell traversal of tetrahedral meshes for direct volume visualization. Paper, 2021.
- [5] Will Usher. Webgpu volume pathtracer. <https://github.com/Twinklebear/webgpu-volume-pathtracer>.