California Polytechnic State
University, San Luis Obispo

Real-Time Volumetric Cloud Rendering

Jaafer Sheriff

Senior Project

June 2018

Supervisor
Dr. Christian Eckhardt

ABSTRACT

short abstract

Contents

1 Introduction

introduction/motivation (1 page)

2 Implementation

We considered 3 major steps to create our clouds. The first was spherical voxelization of billboards, the second was voxel cone tracing, and the third was noise generation.

2.1 Voxelization

Normally a volume is populated using some accompanying static 3D texture data. For our implementation we manually update out voxel individually.

We want our final voxels to represent the billowing, semi-spherical shape of a cloud. We also want to highlight the voxels nearest to the light source which will be essential when shading the cloud.

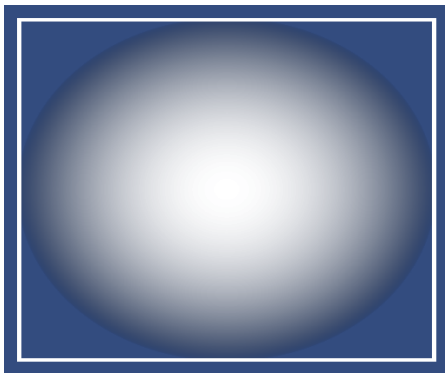
To voxelize we clear the entire volume on every frame and rerun our full voxelization algorithm. This will be necessary if we ever add procedurally animated clouds.

2.1.1 Spherical Billboards

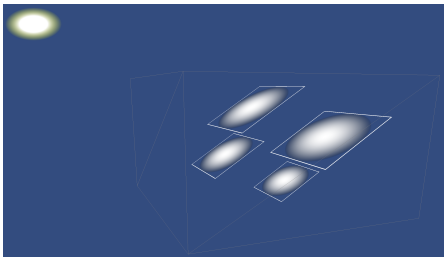
We begin by rendering many billboards from the light's perspective. In the fragment shader we calculate a spherical distribution mapped to the face of the billboard. Using this distribution we populate a sphere within the volume.

```
// Calculate tthe distance from the center of the billboard to the
    current fragment
float fragDist = distance(center, fragPos);
// Calculate a linear distribution [0, 1]
float distribution = fragDist / radius;
// Convert the linear distribution to spherical [0, 1]
distribution = sqrt(max(0, 1 - distribution * distribution));
// Calculate the near point of the sphere intersecting this fragment
float sphereDistance = radius * distribution;
vec3 nearPos = fragPos + billboardNormal * sphereDistance;
// Iterate from the front of the sphere to back
// Set the voxels within this line to black
for(float scale = 0; scale < 2 * sphereDistance; scale += stepSize) {
   vec3 worldPos = nearPos - billboardNormal * scale;
   imageStore(volume, getVoxelIndex(worldPos), vec4(0,0,0,1));
}
```

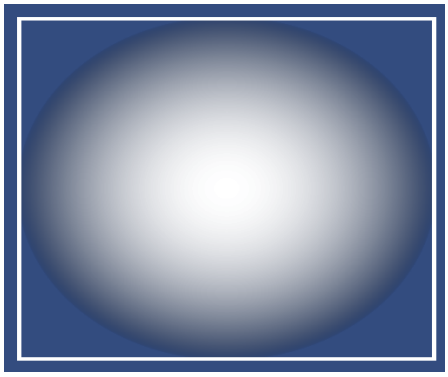
Listing 1: first_voxelize.glsl, 42



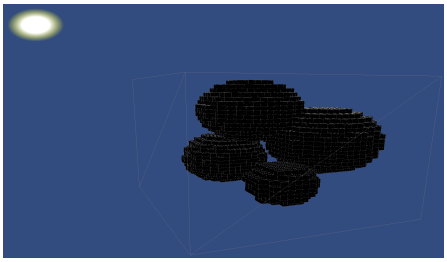
(a) Spherical distribution mapped to a billboard



(b) Light-facing billboards



(c) Diagram of first-voxelization pass



(d) Voxelization of spherical billboards

Figure 1: First voxelization pass of 4 billboards

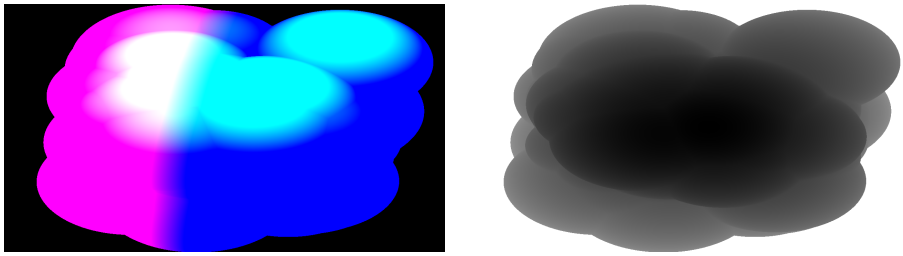
2.1.2 Position Map

We could have rendered these billboard from the camera's perspective and achieved the same voxelization results. We chose to render from the light's perspective because we want to keep track of which voxels are nearest to the light source. During our first voxelization pass we write out the sphere's positions to the frame buffer being rendered to. This frame buffer, once all billboards have been rendered, will contain the world positions of the voxels nearest to the light source.

This method utilizes the graphics pipeline's depth test to occlude spheres any sphere that is hidden by nearer spheres. The depth test, however, relies on rasterized geometry in the z-buffer, and the geometry being rendered is the flat billboards, not the spherical distributions that they represent! To solve this we manually calculate and write the sphere's depth into the z-buffer. We do this in world-space using the light perspective's near and far plane. It would likely be more elegant and efficient to do this calculation in clip-space.

```
// Calculate sphere fragment's world position
// This is the position nearest to the light source at this fragment
vec3 nearPos = fragPos + billboardNormal * sphereDistance;
// Calculate sphere fragment's depth within the light's perspective
float viewSize = distance(lightNearPlane, lightFarPlane);
float depth = distance(lightNearPlane, nearPos) / viewSize;
// Write position and depth to frame buffer
outColor = vec4(nearPos, 1);
gl_FragDepth = depth;
```

Listing 2: first_voxelize.glsl, 60



(a) Color attachment (b) Depth attachment

Figure 2: Final position map after first pass

The final result of the position map is a screen-sized texture containing the world positions of voxels nearest to the camera. This data can be used to highlight those voxels.

2.1.3 Highlighting voxels

For our second voxelization pass we render position map to the screen on a full-sized quad. We sample each fragment of the position map to get the world positions of voxels nearest to the light source, and we then update the voxel at that world position to highlight that it is on the outer layer of the cloud.

```
// Get world position of voxels from position map
vec4 worldPos = imageLoad(positionMap, ivec2(gl_FragCoord.xy));
// Only highlight the voxel if the position map sample is valid
if (worldPos.a >= 0.f) {
    imageStore(volume, calculateVoxelIndex(worldPos.xyz), vec4(1));
}
```

Listing 3: second_voxelize.glsl, 34

The final result of our voxelization algorithm is a billowing spherical cloud-like structure with the voxels nearest to the camera being highlighted.

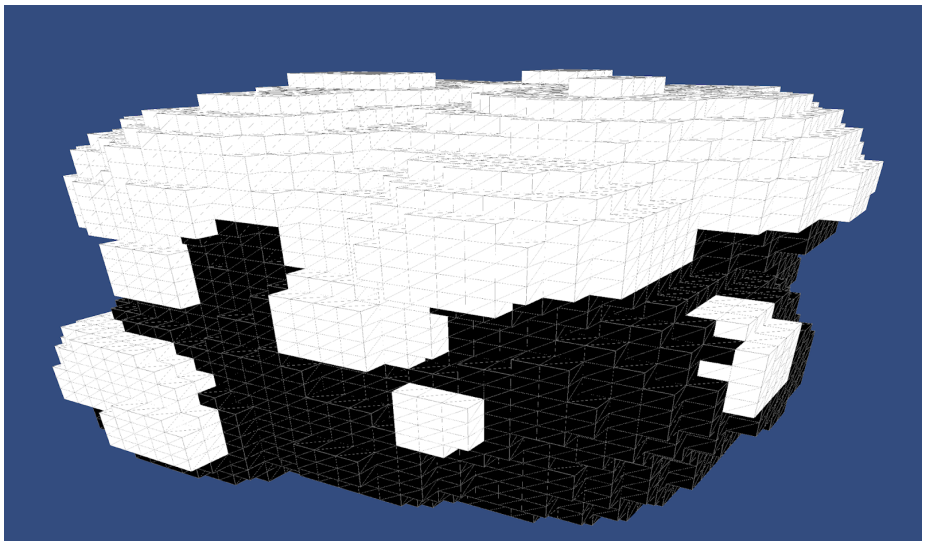


Figure 3: Final result of voxelization

2.2 Voxel Cone Tracing

Now that our voxelization method is complete and our volume contains the necessary data, it is time to ray march through that data. Voxel cone tracing is an existing ray marching technique that utilizes 3D texture mips.

Along with a simple ray march, voxel cone tracing keeps track of the radis of a cone encompassing the ray. That cone radius is used to sample different mip levels within the volume. The further the ray is sampling, the larger the radius, and the higher the mip level is used for sampling.

For our voxel cone tracing method, we render the same billboards as before but this time from the camera's perspective. We use the same spherical distribution calculation to find the surface of the sphere for each billboard. This is where we start our ray march within the volume.

We ray march in the direction of the light source, sampling the volume at the cone radius's respective mips as we go. The end result gives us the final shading for our clouds.

```
vec3 startPosition = calculateVoxelPosition(sphereSurfacePosition);
vec3 direction = lightPosition - startPosition;
float shading = coneTrace(startPosition, direction);
...
float coneTrace(vec3 position, vec3 direction) {
    position /= volumeDimension;
    direction /= volumeDimension;

    float color = 0.f;
    for (int i = 1; i <= steps; i++) {
```

```
        float coneRadius = coneHeight * tan(coneAngle / 2.f);
        float lod = log2(max(1.f, 2.f * coneRadius));
        vec4 sampleColor = textureLod(volume, position + coneHeight *
            direction, lod + vctLodOffset);
        color += sampleColor.r * i/steps; // Down scale sample
        coneHeight += coneRadius;
    }
}
```

Listing 4: conetrace_frag.glsl, 63

Our cone trace method is fully paramterized. The user can change the number of ray march steps, the angle of cone's tip, the start offset of the cone along the ray, and an LOD offset.

2.3 Noise Generation

2.3.1 Parameter playing

implementation (5-6 pages) where you argue in detail how you implemented what i.e. code

3 Results

3 page result (including pictures).

References

[1] "Test references"