

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

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Implementation</b>	<b>5</b>
2.1	Voxelization . . . . .	5
2.1.1	Spherical Billboards . . . . .	5
2.1.2	Position Map . . . . .	6
2.1.3	Highlighting voxels . . . . .	7
2.1.4	Optimizations . . . . .	7
2.2	Voxel Cone Tracing . . . . .	8
2.3	Noise Generation . . . . .	9
2.3.1	Parameter playing . . . . .	9
<b>3</b>	<b>Results</b>	<b>10</b>

# **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

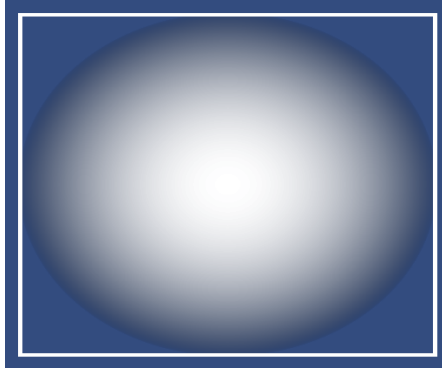
We begin by rendering many billboards from the light's perspective. In the fragment shader we calculate a spherical distribution mapped to the face of the billboard. Using this distribution we populate a sphere within the volume.

---

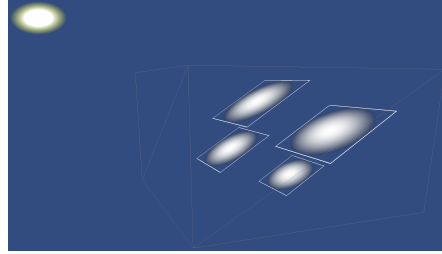
```
// Calculate the distance from the center of the billboard to the
// current fragment
float fragDist = distance(center, fragPos);
// Calculate a linear distribution [0, 1]
float distribution = fragDist / radius;
// Convert the linear distribution to spherical [0, 1]
distribution = sqrt(max(0, 1 - 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));
}
```

---

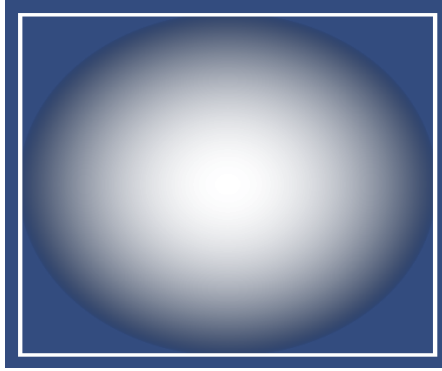
Listing 1: first\_voxelize.glsl, 42



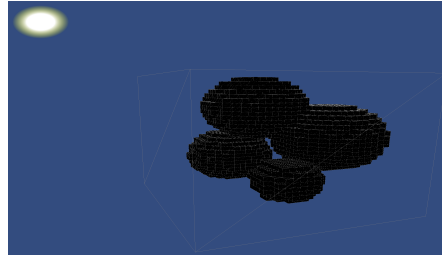
(a) Spherical distribution mapped to a billboard



(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.

---

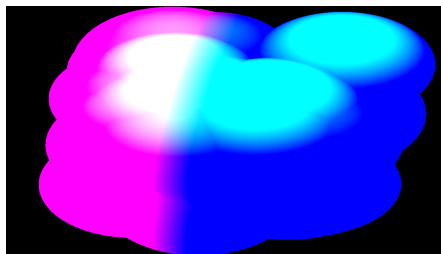
```

// Calculate sphere fragment's world position
// This is the position on this sphere that is 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;

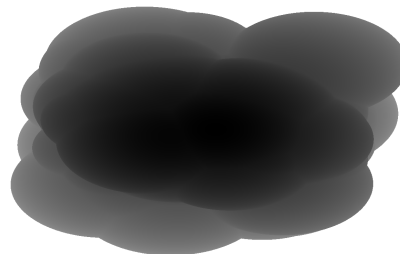
```

---

Listing 2: first\_voxelize.glsl, 60



(a) Color attachment



(b) Depth attachment

Figure 2: Final position map after first pass

### 2.1.3 Highlighting voxels

The last step for our voxelization technique is to highlight voxels nearest to the camera. Lucky for us, the position map contains the world positions of the voxels nearest to the light source! 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 data to highlight that.

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

### 2.1.4 Optimizations

Because of this, our voxels only ever have three states: inactive, active, or highlighted. To optimize our approach, our voxels use R8UI as the internal format. This bits of the red channel are flipped as necessary allowing us to have minimal data representing our volume.

## **2.2 Voxel Cone Tracing**

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



## **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).