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 provided static 3D texture data. For our implementation we manually update individual voxel ourselves.

The goal of our voxelization method is to properly populate our volume. We want the shape of our final voxels represent the billowing, semi-spherical shape of a cloud. Not only do we desire the proper shape, we also aim to highlight the voxels nearest to the light source.

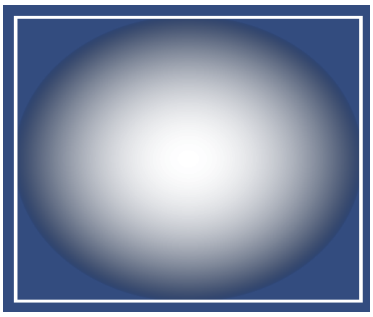
We do our voxelization using billboards and a multi-pass voxelization system. One every frame we clear the entire volume and rerun a full voxelization pass. This will be essential if we ever add procedurally animated clouds.

2.1.1 Spherical Billboards

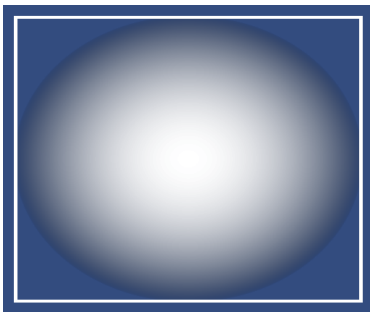
We begin by rendering many billboards from the light's perspective. In the billboard's fragment shader we calculate a spherical distribution mapped to the face of the billboard. Using this spherical distribution, we populate a sphere in the volume.

```
// Calculate distance from the current fragment to billboard's center
float dist = distance(center, fragPos);
// Calculate linear distribution [0, 1]
float distribution = distToCenter / radius;
// Convert 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 front to back, voxelize along the way
for(float length = 0; length < 2 * sphereDistance; length += stepSize) {
   vec3 worldPos = nearPos - billboardNormal * length;
   imageStore(volume, getVoxelIndex(worldPos), vec4(0,0,0,1));
}
```

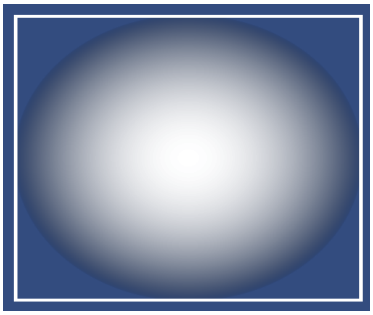
Listing 1: first_voxelize.glsl, 42



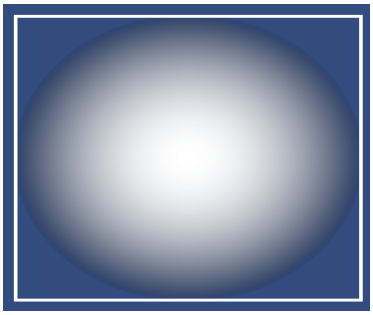
(a) Spherical distribution mapped to a billboard



(b) Diagram of spherical voxelization



(c) Light-facing billboards



(d) Spherical voxelization of billboard

2.1.2 Position Map

When rendering these billboards from the light's perspective because we want to keep track of which voxels are nearest to the light source. While voxelizing, we simulatenously write the world positions of the spherical distributions into a frame buffer. Initially we expected the depth test to manage the occlusion of overlapping spheres. However, the depth test uses rasterized geometry, and our geometry being rendered is the float billboard, not the spherical representations that we want.

To solve this, we manually update the depth buffer by calculating the fragment's spherical depth in respect to the light view's near and far plane. As of nwo this is all done in world-space, but it would be more elegant to do this calculation in clip-space.

```
// Calculate sphere fragment's world position
// This is the position on the sphere nearest to the light source
vec3 nearPos = fragPos + billboardNormal * sphereDistance;
// Calculate sphere fragment's depth within the light's view
float viewSize = distance(lightNearPlane, lightFarPlane);
float depth = distance(lightNearPlane, worldPos) / viewSize;
// Write position and depths to frame buffer
```

```
outColor = vec4(nearPos, 1);
gl_FragDepth = depth;
```

Listing 2: first_voxelize.glsl, 60

2.1.3 Highlighting voxels

The last step for our voxelization technique is to higlight voxels nearest to the camera. Lucky for us, we have exactly what we need in the position map! For our second voxelization pass we render a full screen quad containing the position map. We sample each fragment of the position map to get the world positions of voxels nearest to the light source. We then update the voxel data to highlight it.

Our result is a billowing spherical cloud-like structure with 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).