California Polytechnic State University, San Luis Obispo

Real-Time Volumetric Cloud Rendering

Jaafer Sheriff

Senior Project

June 2018

Supervisor
Dr. Christian Eckhardt

ABSTRACT

Currently there are almost no video games that include real-time animated clouds. Many games don't value full-blown weather pattern visuals because they don't add as much to the user's experience as other features. Making these features simulated can also become expensive.

Because of this, any game titles opt to visualize the sky or clouds with simple textures and billboards. Most of the time these are static visuals that may be combined with cheap tricks to add a dimension of animation.

We propose a method to cheaply generate animated, realstic clouds in real-time. Our clouds require no precomputation or preprocessing. There is not even any traditional post-processing in our clouds. Using 3D textures, noise generation, and voxel cone tracing, our fully parameterized clouds can be utilized in many applications from video games to physics-based weather simulations.

Contents

1 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.

1.1 Voxelization

Normally a volume is populated using some accompanying static 3D texture data. For our implementation we manually update our 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 is essential for shading the cloud.

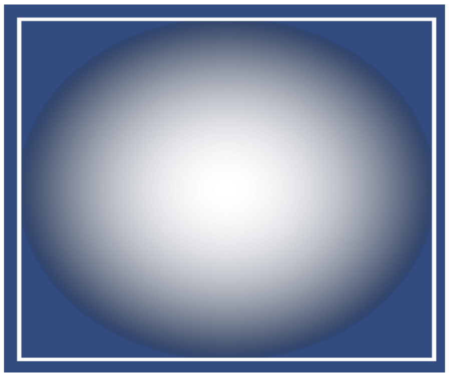
To voxelize we clear the entire volume and rerun our full voxelization algorithm on every frame. This will be necessary if we ever add procedurally animated clouds.

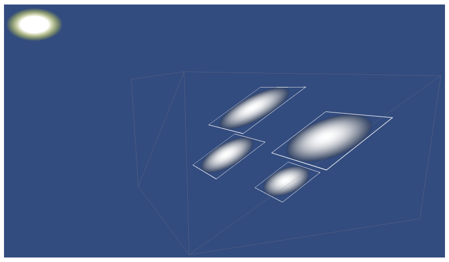
1.1.1 Spherical Billboards

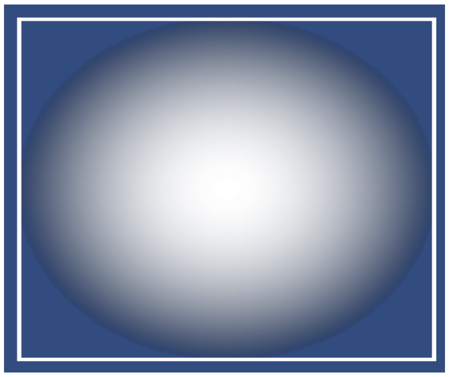
We begin by rendering many billboards from the light's perspective. In the fragment shader we calculate a spherical distribution mapped to the face of the billboard. Using this distribution we populate a sphere within the volume.

```
// 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

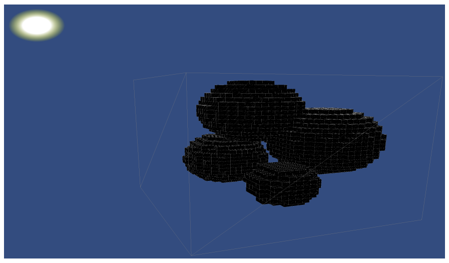

(d) Voxelization of spherical billboards

Figure 1: First voxelization pass of 4 billboards

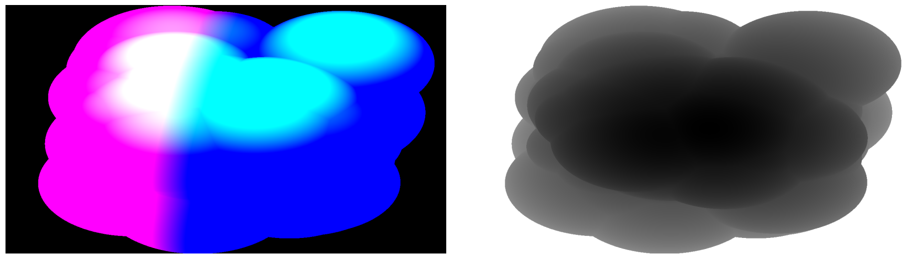
1.1.2 Position Map

We render the billboards 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 simultaneously write out the sphere's world positions to the frame buffer being rendered to. Once all the billboards have been rendered this frame buffer will contain the world positions of the all the voxels nearest to the light source.

This method utilizes the graphics pipeline's depth test to occlude 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.

```glsl
// 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.

1.1.3 Highlighting voxels

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

```glsl
// 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.

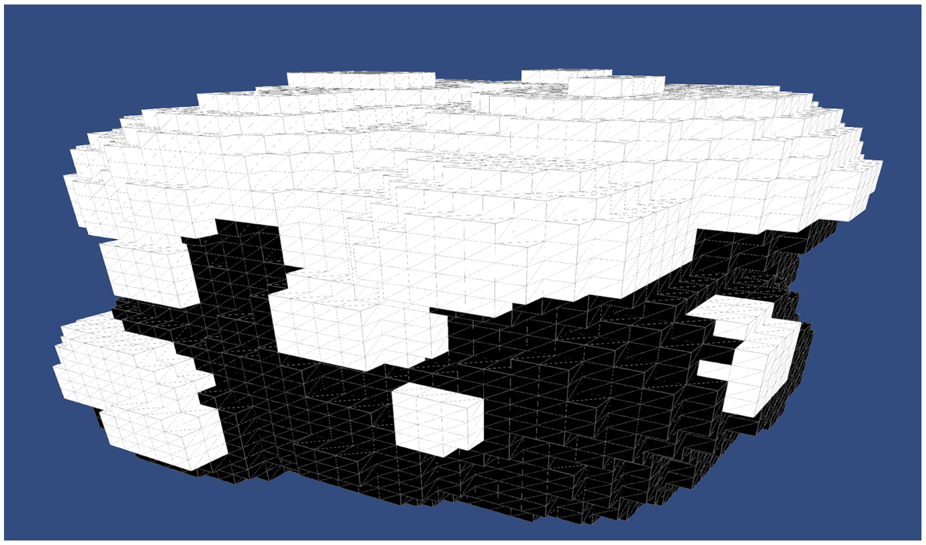


Figure 3: Final result of voxelization

1.2 Voxel Cone Tracing

The next step to render clouds is to ray march through the data. Voxel cone tracing is an existing ray marching technique that utilizes texture mips.

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

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

We cone trace in the direction of the light source, sampling the volume at higher and higher mip levels 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

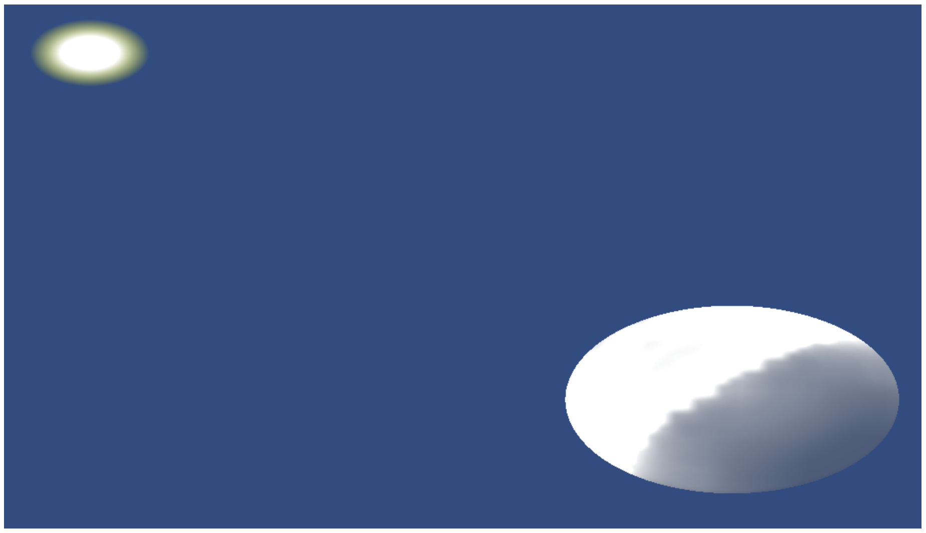


Figure 4: Cone tracing results of a single billboard

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

1.3 Noise Generation

1.3.1 Parameter playing

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

2 Results

3 page result (including pictures).

References

[1] ”Test references”