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. Additionally, creating a simulated weather system can become computationally expensive, and that compute time could be better spent for other game features.

Because of this, many 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 blending tricks to add a loose dimension of animation.

We propose a wholistic 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 required for our clouds. Using 3D textures, 3D noise generation, and voxel cone tracing, our fully parameterized clouds can be utilized in many applications spanning from video games to physics-based weather simulations.

Contents

1 Implementation

Our cloud is represented by a 3D texture and a collection of billboards. To create our clouds we consider 3 major steps. The first is spherical voxelization of the billboards, the second is voxel cone tracing through the volume, and the third is 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 volume data 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 for shading the cloud.

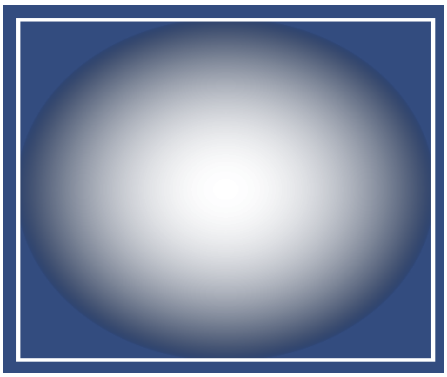
To voxelize we clear the entire volume and rerun our full voxelization algorithm on every frame. This will be necessary if we ever add procedurally animated clouds.

1.1.1 Spherical Billboards

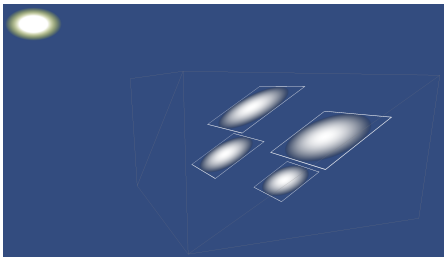
We begin by rendering many billboards from the light's perspective. In the fragment shader we calculate a spherical distribution mapped to the face of the billboard. Using this distribution we populate a sphere within the volume.

```
// Calculate the distance from the center of the billboard to the
    current fragment
float fragDist = distance(center, fragPos);
// Translate that distance into 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 on the sphere respective with 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));
}
```

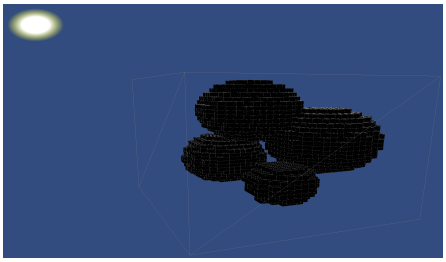
Listing 1: first_voxelize.glsl, 51



(a) Spherical distribution mapped to a billboard



(b) Light-facing billboards



(c) Voxelization of spherical billboards

Figure 1: First voxelization pass of 4 billboards

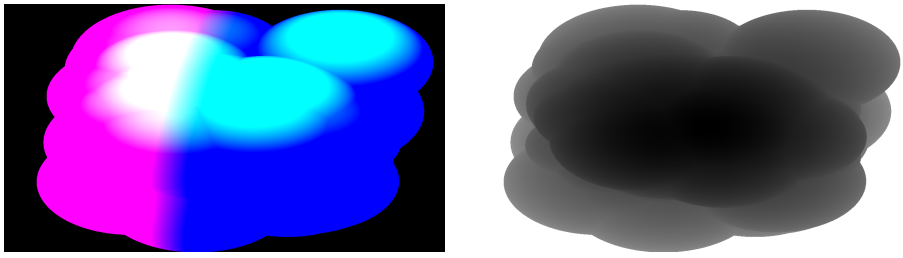
1.1.2 Position Map

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. The geometry being rendered, however, is the flat billboards and 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 this fragment's spherical distribution's world position
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, 61



(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 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.

```
// Get world position of voxels from position map
vec4 worldPos = imageLoad(positionMap, ivec2(gl_FragCoord.xy));
// Only highlight the voxel if the 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, semi-spherical, cloud-like structure with the voxels nearest to the light source being highlighted.

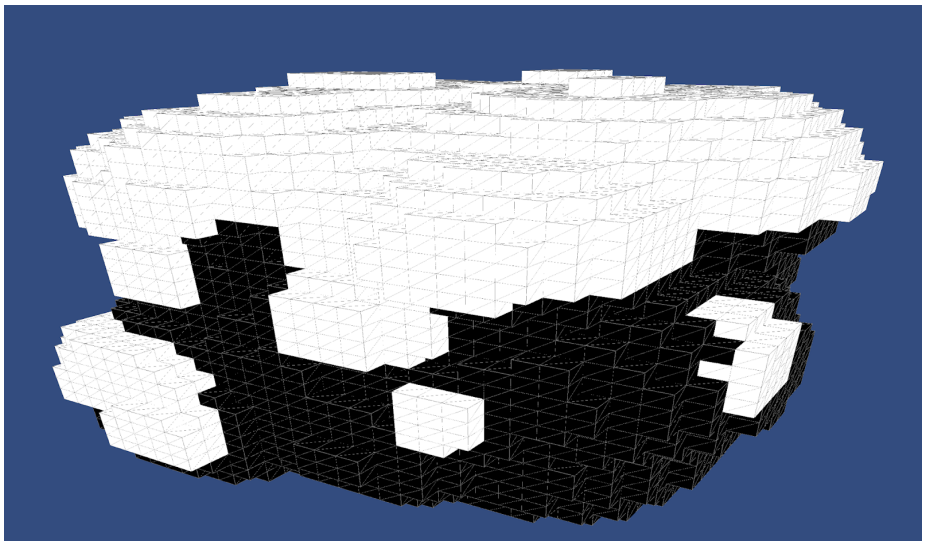


Figure 3: Final result of voxelization

1.2 Voxel Cone Tracing

With the volume data for this frame complete, the next step is to ray march through it. 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.

```
// Start position and direction of ray march
vec3 startPosition = calculateVoxelPosition(sphereSurfacePosition);
vec3 direction = lightPosition - startPosition;

// Scale to volume dimension's
position /= volumeDimension;
direction /= volumeDimension;

// Ray march
vec3 color = vec3(0, 0, 0);
for (int i = 1; i <= steps; i++) {
    // Update the cone radius at this step in the ray march
    // Cone angle is the angle of the cone's tip
    float coneRadius = coneHeight * tan(coneAngle / 2.f);
    // Calculate the mip level using the cone radius
    float lod = log2(max(1.f, 2.f * coneRadius));
    // Sample the volume
    vec4 sampleColor = textureLod(volume, position + coneHeight *
        direction, lod);
    // Update the output if the voxel is valid
    if (sampleColor.a > 0.f) {
        color += sampleColor.rgb;
    }
    coneHeight += coneRadius;
}
return color;
```

Listing 4: conetrace_frag.glsl, 63

Figure 4: Cone tracing results of a single billboard

With added billboards we get the final shading of our clouds. Cone-traced shading of billboards nearer to the light will be brighter than that of billboards that are concealed.

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

To actually achieve the wispy shape of a cloud we use an existing soft-particle noise implementation.

For this implementation we first generate a semi-transparent noise volume using gradient densities. The final result is a signed 3D texture with randomly generated RGB and variable alpha values.

Figure 5: 1 of 32 slices in the noise texture

To represent this noise texture, we follow a similar process as our first pass voxelization: for every billboard we calculate a spherical distribution. Using this distribution we find the near and far positions on the sphere respective to the current fragment. We start at the near position and iterate through the sphere to the back. On each step, we sample the noise texture across several octaves.

When sampling the noise texture, we add an offset that updates with runtime. This causes our sampling coordinates to translate through the noise vol-

ume over time creating an animating effect. The final result is convincing fog.

Figure 6: Noise-based fog using 10 billboards

2 Results

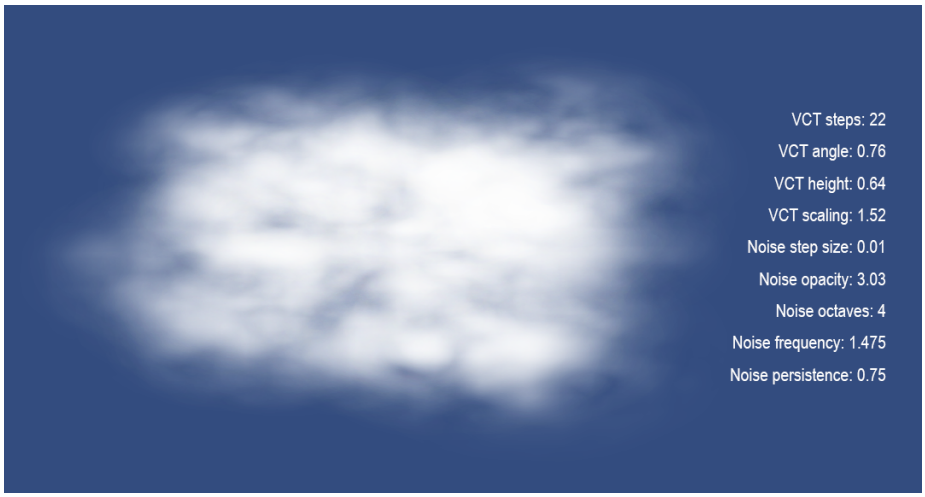
The last step in our process is to combine the noise-generated fog with the voxel cone traced shading.

Our clouds can take many shapes and forms. Because of the flexibility of the parameters, we can create many different types of clouds.

The following clouds were rendered on a GTX 965M at 1280x720 resolution. All the clouds rendered at atleast 60 frames per second.

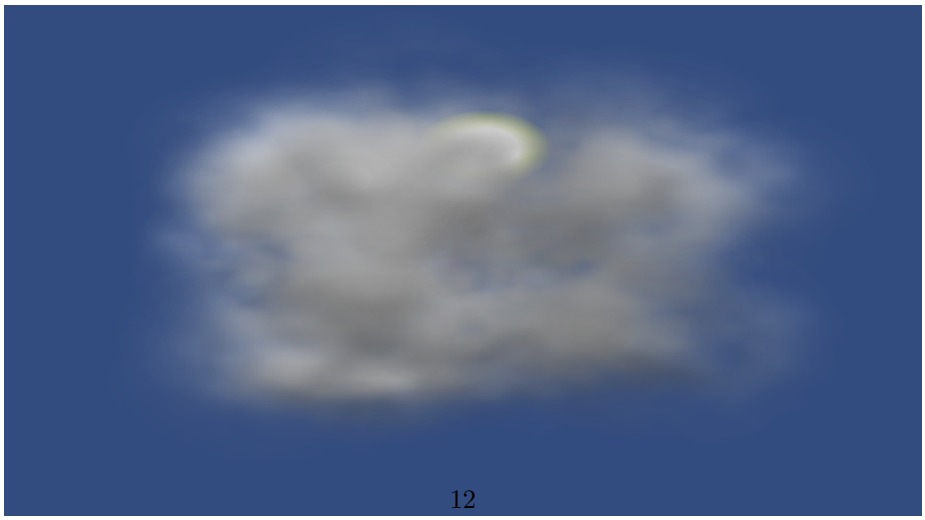
Figure 7: view aligned with sun

Figure 8: view aligned with sun



(a) View aligned with sun

(b) View alongside sun



(c) Sun occluded by cloud

Figure 9: Varied view alignments

References

[1] "Test references"

[2] "Test references"

[3] "Test references"