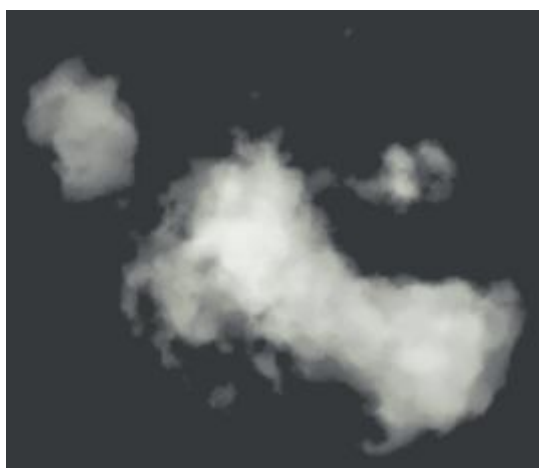




Single-Pass Raycasting



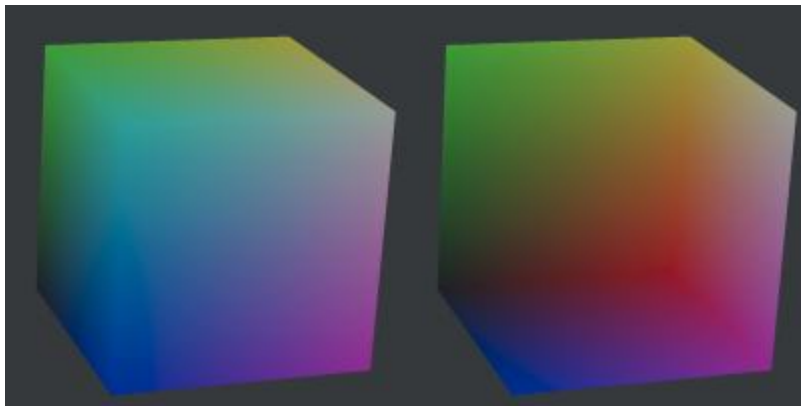
Raycasting over a volume requires start and stop points for your rays. The traditional method for computing these intervals is to draw a cube (with perspective) into two surfaces: one surface has front faces, the other has back faces. By using a fragment shader that writes object-space XYZ into the RGB channels, you get intervals. Your final pass is the actual raycast.

In the original incarnation of this post, I proposed making it into a single pass process by dilating a back-facing triangle from the cube and performing perspective-correct interpolation math in the fragment shader. [Simon Green](#) pointed out that this was a bit silly, since I can simply do a ray-cube intersection. So I rewrote this post, showing how to correlate a field-of-view angle (typically used to generate an OpenGL projection matrix) and focal length (typically used to determine ray direction). This might be useful to you if you need to integrate a raycast volume into an existing 3D scene that uses traditional rendering.

To have something interesting to render in the demo code (download is at the end of the post), I generated a pyroclastic cloud as described in [this amazing PDF on volumetric methods](#) from the 2010 SIGGRAPH course. Miles Macklin has a simply great blog entry about it [here](#).

Recap of Two-Pass Raycasting

Here's a depiction of the usual offscreen surfaces for ray intervals:



Front faces give you the start points and the back faces give you the end points. The usual procedure goes like this:

1. Draw a cube's front faces into surface A and back faces into surface B. This determines ray intervals.
 - Attach two textures to the current FBO to render to both surfaces simultaneously.
 - Use a fragment shader that writes out normalized object-space coordinates to the RGB channels.
2. Draw a full-screen quad to perform the raycast.
 - Bind three textures: the two interval surfaces, and the 3D texture you're raycasting against.
 - Sample the two interval surfaces to obtain ray start and stop points. If they're equal, issue a **discard**.

Making it Single-Pass

To make this a one-pass process and remove two texture lookups from the fragment shader, we can use a procedure like this:

1. Draw a cube's front-faces to perform the raycast.
 - On the CPU, compute the eye position in object space and send it down as a uniform.
 - Also on the CPU, compute a focal length based on the field-of-view that you're using to generate your scene's projection matrix.
 - At the top of the fragment shader, perform a ray-cube intersection.

Raycasting on front-faces instead of a full-screen quad allows you to avoid the need to test for intersection failure. Traditional raycasting shaders issue a **discard** if there's no intersection with the view volume, but since we're guaranteed to hit the viewing volume, so there's no need.

Without further ado, here's my fragment shader using modern GLSL syntax:

```

01 out vec4 FragColor;
02
03 uniform mat4 ModelView;
04 uniform float FocalLength;
05 uniform vec2 WindowSize;
06 uniform vec3 RayOrigin;
07
08 struct Ray {
09     vec3 Origin;
10     vec3 Dir;
11 };
12

```

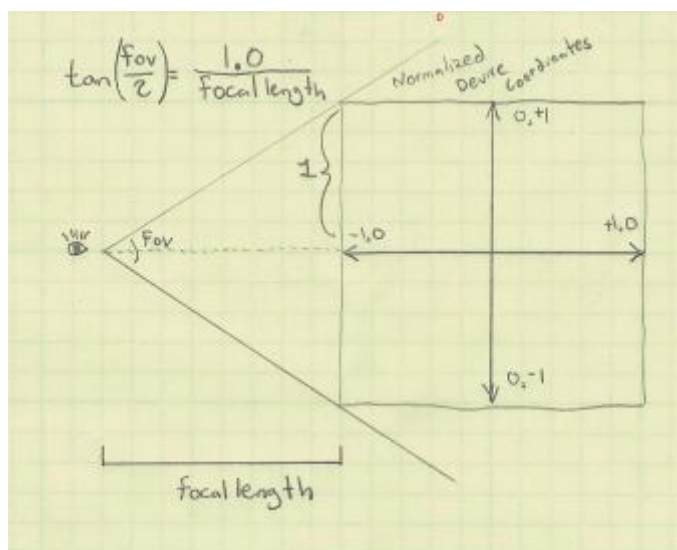
```

13 struct AABB {
14     vec3 Min;
15     vec3 Max;
16 };
17
18 bool IntersectBox(Ray r, AABB aabb, out float t0, out float t1)
19 {
20     vec3 invR = 1.0 / r.Dir;
21     vec3 tbot = invR * (aabb.Min-r.Origin);
22     vec3 ttop = invR * (aabb.Max-r.Origin);
23     vec3 tmin = min(ttop, tbot);
24     vec3 tmax = max(ttop, tbot);
25     vec2 t = max(tmin.xx, tmin.yz);
26     t0 = max(t.x, t.y);
27     t = min(tmax.xx, tmax.yz);
28     t1 = min(t.x, t.y);
29     return t0 <= t1;
30 }
31
32 void main()
33 {
34     vec3 rayDirection;
35     rayDirection.xy = 2.0 * gl_FragCoord.xy / WindowSize - 1.0;
36     rayDirection.z = -FocalLength;
37     rayDirection = (vec4(rayDirection, 0) * Modelview).xyz;
38
39     Ray eye = Ray( RayOrigin, normalize(rayDirection) );
40     AABB aabb = AABB(vec3(-1.0), vec3(+1.0));
41
42     float tnear, tfar;
43     IntersectBox(eye, aabb, tnear, tfar);
44     if (tnear < 0.0) tnear = 0.0;
45
46     vec3 rayStart = eye.Origin + eye.Dir * tnear;
47     vec3 rayStop = eye.Origin + eye.Dir * tfar;
48
49     // Transform from object space to texture coordinate space:
50     rayStart = 0.5 * (rayStart + 1.0);
51     rayStop = 0.5 * (rayStop + 1.0);
52
53     // Perform the ray marching:
54     vec3 pos = rayStart;
55     vec3 step = normalize(rayStop-rayStart) * stepSize;
56     float travel = distance(rayStop, rayStart);
57     for (int i=0; i < MaxSamples && travel > 0.0; ++i, pos += step, travel -= stepSize) {
58
59         // ...lighting and absorption stuff here...
60
61     }

```

The shader works by using **gl_FragCoord** and a given **FocalLength** value to generate a ray direction. Just like a traditional CPU-based raytracer, the appropriate analogy is to imagine holding a square piece of chicken wire in front of you, tracing rays from your eyes through the holes in the mesh.

If you're integrating the raycast volume into an existing scene, computing **FocalLength** and **RayOrigin** can be a little tricky, but it shouldn't be too difficult. Here's a little sketch I made:



In days of yore, most OpenGL programmers would use the **gluPerspective** function to compute a projection matrix, although nowadays you're probably using whatever vector math library you happen to be using. My personal favorite is the simple C++ vector library from Sony that's included in [Bullet](#). Anyway, you're probably calling a function that takes a field-of-view angle as an argument:

```
1 Matrix4 Perspective(float fovy, float aspectRatio, float nearPlane, float farPlane);
```

Based on the above diagram, converting the fov value into a focal length is easy:

```
1 float focalLength = 1.0f / tan(FovView / 2);
```

You're also probably calling function kinda like **gluLookAt** to compute your view matrix:

```
1 Matrix4 LookAt(Point3 eyePosition, Point3 targetPosition, Vector3 up);
```

To compute a ray origin, transform the eye position from world space into object space, relative to the viewing cube.

Downloads

I've tested the code with Visual Studio 2010. It uses [CMake](#) for the build system.

- [raycast.zip](#)
- [SinglePass.glsl](#)
- [Raycast.cpp](#)

I consider this code to be on the public domain. Enjoy!

Written by Philip Rideout

January 16th, 2011 at 2:59 am

Posted in [OpenGL](#)

Tagged with [cloud](#), [opengl](#), [raycasting](#)

« [Noise-Based Particles, Part I](#)

[3D Eulerian Grid](#) »

Blog

[Home](#)

[Contact](#)

[Admin](#)

Links

- [Old Stuff](#)
- [Videos](#)

Publications

[iPhone 3D](#)

[GPU Pro 2](#)

