



Volumetric Splatting

1. [Volume Raycasting](#)
2. [Volumetric Lighting](#)
3. [Reducing Slice Artifacts](#)
4. [3D Gaussian Splat](#)
5. [Flowline Extrusion](#)
6. [Velocity Visualization](#)
7. [Downloads](#)

Instanced rendering turns out to be useful for volumetric graphics. I first came across the concept in [this excellent chapter](#) in *GPU Gems 3*, which briefly mentions that instancing can voxelize a model in only 1 draw call using nothing but a quad. After reading this, I had the idea of using instancing to efficiently render volumetric splats. Splatting is useful for creating distance fields and Voronoi maps. It can also be used to extrude a streamline into a space-filling velocity field.

In this article, I show how volumetric splatting can be implemented efficiently with instancing. I show how to leverage splatting to extrude a circular streamline into a toroidal field of velocity vectors. This technique would allow an artist to design a large velocity field (e.g., for a particle system), simply by specifying a small animation path through space. My article also covers some basics of modern-day volume rendering on the GPU.

Volume Raycasting

Before I show you my raycasting shader, let me show you a neat trick to easily obtain the start and stop points for the rays. It works by drawing a cube into a pair of floating-point RGB surfaces, using a fragment shader that writes out object-space coordinates. Frontfaces go into one color attachment, backfaces in the other. This results in a tidy set of start/stop positions. Here are the shaders:

```
01  -- Vertex
02
03  in vec4 Posi ti on;
04  out vec3 vPosi ti on;
05  uni form mat4 Model vi ewProj ecti on;
06
07  void mai n()
08  {
09      gl_Pos i ti on = Model vi ewProj ecti on * Posi ti on;
10      vPosi ti on = Posi ti on. xyz;
11  }
12
13  -- Fragment
14
15  in vec3 vPosi ti on;
16  out vec3 FragData[2];
17
18  void mai n()
19  {
20      if (gl_FrontFaci ng) {
```

```

21     FragData[0] = 0.5 * (vPosition + 1.0);
22     FragData[1] = vec3(0);
23 } else {
24     FragData[0] = vec3(0);
25     FragData[1] = 0.5 * (vPosition + 1.0);
26 }
27 }

```

Update: I recently realized that the pair of endpoint surfaces can be avoided by performing a quick ray-cube intersection in the fragment shader. I wrote a blog entry about it [here](#).

You'll want to first clear the surfaces to black, and enable simple additive blending for this to work correctly. Note that we're using multiple render targets (MRT) to generate both surfaces in a single pass. To pull this off, you'll need to bind a FBO that has two color attachments, then issue a **glDrawBuffers** command before rendering, like this:

```

1  GLenum renderTargets[2] = {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1};
2  glDrawBuffers(2, &renderTargets[0]);
3  // Render cube...

```



The next step is the actual raycasting, which is done by drawing a fullscreen quad with a longish fragment shader. We need to set up three texture samplers: two for the start/stop surfaces and one for the 3D volume texture itself. The following fragment shader performs raycasts against a single-channel 3D texture, which I used to generate the teapot image to the right. I obtained the scorpion-in-teapot volume data from [this site](#) at the University of Tübingen.

```

01  uniform sampler2D RayStart;
02  uniform sampler2D RayStop;
03  uniform sampler3D Volume;
04
05  out vec4 FragColor;
06  in vec3 vPosition;
07
08  uniform float StepLength = 0.01;
09
10  void main()
11  {
12      vec2 coord = 0.5 * (vPosition.xy + 1.0);
13      vec3 rayStart = texture(RayStart, coord).xyz;
14      vec3 rayStop = texture(RayStop, coord).xyz;
15
16      if (rayStart == rayStop) {
17          discard;
18          return;
19      }
20
21      vec3 ray = rayStop - rayStart;
22      float rayLength = length(ray);
23      vec3 stepVector = StepLength * ray/rayLength;
24
25      vec3 pos = rayStart;
26      vec4 dst = vec4(0);
27      while (dst.a < 1 && rayLength > 0) {
28          float density = texture(Volume, pos).x;
29          vec4 src = vec4(density);
30          src.rgb *= src.a;
31          dst = (1.0 - dst.a) * src + dst;
32          pos += stepVector;

```

```

33     rayLength -= StepLength;
34 }
35
36     FragColor = dst;
37 }

```

Note the front-to-back blending equation inside the **while** loop; Benjamin Supnik has a [good article](#) about front-to-back blending on his blog. One advantage of front-to-back raycasting: it allows you to break out of the loop on fully-opaque voxels.

Volumetric Lighting

You'll often want to create a more traditional lighting effect in your raycaster. For this, you'll need to obtain surface normals somehow. Since we're dealing with volume data, this might seem non-trivial, but it's actually pretty simple.



Really there are two problems to solve: (1) detecting voxels that intersect surfaces in the volume data, and (2) computing the normal vectors at those positions. Turns out both of these problems can be addressed with an essential concept from vector calculus: the **gradient** vector points in the direction of greatest change, and its magnitude represents the amount of change. If we can compute the gradient at a particular location, we can check its magnitude to see if we're crossing a surface. And, conveniently enough, the direction of the gradient is exactly what we want to use for our lighting normal!

The gradient vector is made up of the partial derivatives along the three axes; it can be approximated like this:

```

1  vec3 ComputeGradient(vec3 P)
2  {
3      float L = StepLength;
4      float E = texture(VolumeSampler, P + vec3(L, 0, 0));
5      float N = texture(VolumeSampler, P + vec3(0, L, 0));
6      float U = texture(VolumeSampler, P + vec3(0, 0, L));
7      return vec3(E - V, N - V, U - V);
8  }

```

For the teapot data, we'll compute the gradient for lighting normals only when the current voxel's value is above a certain threshold. This lets us avoid making too many texture lookups. The shader looks like this:

```

01  out vec4 FragColor;
02  in vec3 vPosition;
03
04  uniform sampler2D RayStart;
05  uniform sampler2D RayStop;
06  uniform sampler3D Volume;
07
08  uniform float StepLength = 0.01;
09  uniform float Threshold = 0.45;
10
11  uniform vec3 LightPosition;
12  uniform vec3 DiffuseMaterial;
13  uniform mat4 ModelView;
14  uniform mat3 NormalMatrix;
15
16  float Lookup(vec3 coord)
17  {

```

```

18     return texture(Volume, coord).x;
19 }
20
21 void main()
22 {
23     vec2 coord = 0.5 * (vPosition.xy + 1.0);
24     vec3 rayStart = texture(RayStart, coord).xyz;
25     vec3 rayStop = texture(RayStop, coord).xyz;
26
27     if (rayStart == rayStop) {
28         discard;
29         return;
30     }
31
32     vec3 ray = rayStop - rayStart;
33     float rayLength = length(ray);
34     vec3 stepVector = StepLength * ray / rayLength;
35
36     vec3 pos = rayStart;
37     vec4 dst = vec4(0);
38     while (dst.a < 1 && rayLength > 0) {
39
40         float V = lookup(pos);
41         if (V > Threshold) {
42
43             float L = StepLength;
44             float E = lookup(pos + vec3(L, 0, 0));
45             float N = lookup(pos + vec3(0, L, 0));
46             float U = lookup(pos + vec3(0, 0, L));
47             vec3 normal = normalize(NormalMatrix * vec3(E - V, N - V, U - V));
48             vec3 light = LightPosition;
49
50             float df = abs(dot(normal, light));
51             vec3 color = df * DiffuseMaterial;
52
53             vec4 src = vec4(color, 1.0);
54             dst = (1.0 - dst.a) * src + dst;
55             break;
56         }
57
58         pos += stepVector;
59         rayLength -= StepLength;
60     }
61
62     FragColor = dst;
63 }

```

Reducing Slice Artifacts



When writing your first volume renderer, you'll undoubtedly come across the scourge of “wood grain” artifacts; your data will look like it's made up of a stack of slices (which it is!). Obviously, reducing the raycast step size can help with this, but doing so can be detrimental to performance.

There are a couple popular tricks that can help: (1) re-checking the “solidity” of the voxel by jumping around at half-step intervals, and (2) jittering the ray's starting position along the view direction. I added both of these tricks into our fragment shader; they're highlighted in gray here:

```

01 // ...same as before...
02
03 uniform sampler2D Noise;

```

```

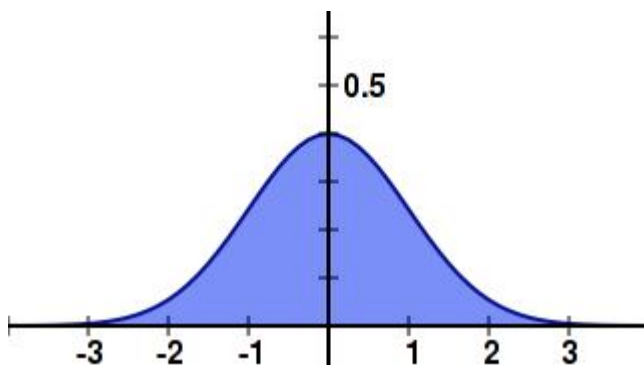
04
05 void main()
06 {
07     // ...same as before...
08
09     rayStart += stepVector * texture(Noise, gl_FragCoord.xy / 256).r;
10
11     vec3 pos = rayStart;
12     vec4 dst = vec4(0);
13     while (dst.a < 1 && rayLength > 0) {
14
15         float V = lookup(pos);
16         if (V > Threshold) {
17
18             vec3 s = -stepVector * 0.5;
19             pos += s; V = lookup(pos);
20             if (V > Threshold) s *= 0.5; else s *= -0.5;
21             pos += s; V = lookup(pos);
22
23             if (V > Threshold) {
24                 // ...same as before...
25             }
26         }
27
28         pos += stepVector;
29         rayLength -= StepLength;
30     }
31
32     FragColor = dst;
33 }

```

3D Gaussian Splat

Now that we've covered the basics of volume rendering, let's come back to the main subject of this article, which deals with the generation of volumetric data using Gaussian splats.

One approach would be evaluating the 3D Gaussian function on the CPU during application initialization, and creating a 3D texture from that. However, I find it to be faster to simply compute the Gaussian in real-time, directly in the fragment shader.



Recall that we're going to use instancing to render all the slices of the splat with only 1 draw call. One awkward aspect of GLSL is that the **gl_InstanceID** input variable is only accessible from the vertex shader, while the **gl_Layer** output variable is only accessible from the geometry shader. It's not difficult to deal with this though! Without further ado, here's the trinity of shaders for 3D Gaussian splatting:

```

01 -- Vertex Shader
02
03 in vec4 Position;
04 out vec2 vPosition;
05 out int vInstance;

```

```

06 uniform vec4 Center;
07
08 void main()
09 {
10     gl_Position = Position + Center;
11     vPosition = Position.xy;
12     vInstance = gl_InstanceID;
13 }
14
15 -- Geometry Shader
16
17 layout(triangles) in;
18 layout(triangle_strip, max_vertices = 3) out;
19
20 in int vInstance[3];
21 in vec2 vPosition[3];
22 out vec3 gPosition;
23
24 uniform float InverseSize;
25
26 void main()
27 {
28     gPosition.z = 1.0 - 2.0 * vInstance[0] * InverseSize;
29     gl_Layer = vInstance[0];
30
31     gPosition.xy = vPosition[0];
32     gl_Position = gl_in[0].gl_Position;
33     EmitVertex();
34
35     gPosition.xy = vPosition[1];
36     gl_Position = gl_in[1].gl_Position;
37     EmitVertex();
38
39     gPosition.xy = vPosition[2];
40     gl_Position = gl_in[2].gl_Position;
41     EmitVertex();
42
43     EndPrimitive();
44 }
45
46 -- Fragment Shader
47
48 in vec3 gPosition;
49 out vec3 FragColor;
50
51 uniform vec3 Color;
52 uniform float InverseVariance;
53 uniform float NormalizationConstant;
54
55 void main()
56 {
57     float r2 = dot(gPosition, gPosition);
58     FragColor = Color * NormalizationConstant * exp(r2 * InverseVariance);
59 }

```

Setting up a 3D texture as a render target might be new to you; here's one way you could set up the FBO: (note that I'm calling **glFramebufferTexture** rather than **glFramebufferTexture{2,3}D**)

```

01 struct Volume {
02     GLuint FboHandle;
03     GLuint TextureHandle;
04 };
05
06 Volume CreateVolume(GLsizei width, GLsizei height, GLsizei depth)
07 {
08     Volume volume;
09     glGenFramebuffers(1, &volume.FboHandle);
10     glBindFramebuffer(GL_FRAMEBUFFER, volume.FboHandle);
11
12     GLuint textureHandle;
13     glGenTextures(1, &textureHandle);
14     glBindTexture(GL_TEXTURE_3D, textureHandle);
15     glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
16     glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
17     glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);

```

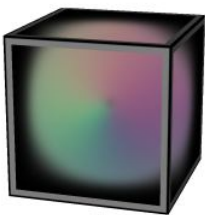
```

18     glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
19     glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
20     glTexImage3D(GL_TEXTURE_3D, 0, GL_RGB16F, width, height, depth, 0,
21                GL_RGB, GL_HALF_FLOAT, 0);
22     volume.TextureHandle = textureHandle;
23
24     GLint miplevel = 0;
25     GLuint colorbuffer;
26     glGenRenderbuffers(1, &colorbuffer);
27     glBindRenderbuffer(GL_RENDERBUFFER, colorbuffer);
28     glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, textureHandle, miplevel);
29
30     return volume;
31 }

```

Note that we created a half-float RGB texture for volume rendering — this might seem like egregious usage of memory, but keep in mind that our end goal is to create a field of velocity vectors.

Flowline Extrusion



Now that we have the splat shaders ready, we can write the C++ code that extrudes a vector of positions into a velocity field. It works by looping over the positions in the path, computing the velocity at that point, and splatting the velocity. The call to **glDrawArraysInstanced** is simply rendering a quad with the instance count set to the depth of the splat.

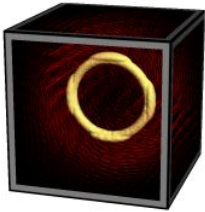
```

01 typedef std::vector<VectorMath::Point3> PointList;
02
03 glEnable(GL_BLEND);
04 glBlendFunc(GL_ONE, GL_ONE);
05
06 PointList::const_iterator i = positions.begin();
07 for (; i != positions.end(); ++i) {
08
09     PointList::const_iterator next = i;
10     if (++next == positions.end())
11         next = positions.begin();
12     VectorMath::Vector3 velocity = (*next - *i);
13
14     GLint center = glGetUniformLocation(program, "Center");
15     glUniform4f(center, i->getX(), i->getY(), i->getZ(), 0);
16
17     GLint color = glGetUniformLocation(program, "Color");
18     glUniform3fv(color, 1, (float*) &velocity);
19
20     glDrawArraysInstanced(GL_TRIANGLE_STRIP, 0, 4, Size);
21 }

```

Blending is essential for this to work correctly. If you want to create a true distance field, you'd want to use **GL_MAX** blending rather than the default blending equation (which is **GL_FUNC_ADD**), and you'd want your fragment shader to use evaluate a linear falloff rather than the Gaussian function.

Velocity Visualization



One popular way to visualize a grid of velocities is via short lines with alpha gradients, as in the image to the right (click to enlarge). This technique is easy to implement with modern OpenGL. Simply populate a VBO with a single point per grid cell, then use the geometry shader to extrude each point into a short line segment whose length and direction reflects the velocity vector in that cell. It's rather beautiful actually! Here's the shader triplet:

```

01  -- Vertex Shader
02
03  in vec4 Position;
04  out vec4 vPosition;
05  uniform mat4 ModelViewProjection;
06
07  void main()
08  {
09      gl_Position = ModelViewProjection * Position;
10      vPosition = Position;
11  }
12
13  -- Geometry Shader
14
15  layout(points) in;
16  layout(line_strip, max_vertices = 2) out;
17  out float gAlpha;
18  uniform mat4 ModelViewProjection;
19  in vec4 vPosition[1];
20  uniform sampler3D Volume;
21
22  void main()
23  {
24      vec3 coord = 0.5 * (vPosition[0].xyz + 1.0);
25      vec4 V = vec4(texture(Volume, coord).xyz, 0.0);
26
27      gAlpha = 0;
28      gl_Position = gl_in[0].gl_Position;
29      EmitVertex();
30
31      gAlpha = 1;
32      gl_Position = ModelViewProjection * (vPosition[0] + V);
33      EmitVertex();
34
35      EndPrimitive();
36  }
37
38  -- Fragment Shader
39
40  out vec4 FragColor;
41  in float gAlpha;
42  uniform float Brightness = 0.5;
43
44  void main()
45  {
46      FragColor = Brightness * vec4(gAlpha, 0, 0, 1);
47  }

```

Downloads

The demo code uses a subset of the [Pez ecosystem](#), which is included in the zip below. It uses [CMake](#) for the build system.

- [Splat.zip](#)
- [Splat.glsl](#)
- [Streamline.glsl](#)
- [Volume.glsl](#)

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

Written by Philip Rideout

November 28th, 2010 at 8:22 pm

Posted in [OpenGL](#)

Tagged with [streamlines](#), [volume rendering](#)

« [Simple Fluid Simulation](#)
[Tron, Volumetric Lines, and Meshless Tubes](#) »

Blog

[Home](#)
[Contact](#)
[Admin](#)

Links

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

Publications

[iPhone 3D](#)
[GPU Pro 2](#)

