Using Shaders to Enhance Scientific Visualizations

You Can Do Image Processing on Dynamic Scenes with a Two-pass Approach

The negative of a 3D object often reveals details

Embossing

Visualization Imaging -- Sharpening

Visualization Imaging -- Edge Detection
Toon Rendering for Non-Photorealistic Effects

Use the GPU to enhance scientific, engineering, and architectural illustration.

Image Manipulation Example – Where is it Likely to Snow?

If have_clouds && have_a_low_temperature && have_water_vapor
    color = green;
else
    color = from visible map

Visualization – Polar Hyperbolic Space

Use the GPU to perform nonlinear vertex transformations.

Dome Projection for Immersive Visualization

Use the GPU to perform nonlinear vertex transformations.

Placing 3D Point Cloud Data into a Floating-Point Texture for glman

fwrite( &nums, 4, 1, fp );
fwrite( &numt, 4, 1, fp );
fwrite( &nump, 4, 1, fp );
for( int p = 0; p < nump; p++ )
{
    for( int t = 0; t < numt; t++ )
    {
        for( int s = 0; s < nums; s++ )
        {
            float red, green, blue, alpha;
            << assign red, green, blue, alpha >>
            fwrite( &red, 4, 1, fp );
            fwrite( &green, 4, 1, fp );
            fwrite( &blue, 4, 1, fp );
            fwrite( &alpha, 4, 1, fp );
        }
    }
}
Point Cloud from a 3D Texture Dataset

Where to Place the Geometry?

I personally like thinking of the data as living in a cube that ranges from -1. to 1. in X, Y, and Z. It is easy to position geometry in this space and easy to view and transform it. This means that any 3D object in that space, not just a point cloud, can map itself to the 3D texture data space.

So, because the x texture coordinate goes from 0. to 1., then the linear mapping from the physical x coordinate to the texture x coordinate is:

\[ s = \frac{x + 1.}{2}. \]

The same mapping applies to y and z to create the t and p texture coordinates.

In GLSL, this can be done in one line of code:

\[ \text{vec3 stp} = ( \text{xyz} + 1. ) / 2.; \]

You can also go the other way:

\[ \text{vec3 xyz} = -1. + ( 2. * \text{stp} ); \]

The Vertex Shader

```glsl
out vec3 vMC;

void main() {
    vMC = gl_Vertex.xyz;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

The Fragment Shader

```glsl
uniform float uMin, uMax;
uniform sampler3D uTexUnit;
in vec3 vMC;
const float SMIN = 0.;
const float SMAX = 120.;

void main() {
    vec3 stp = ( vMC + 1. ) / 2.;        // maps [-1.,1.] to [0.,1.]
    if( any( lessThan( stp, vec3(0.,0.,0.) ) ) ) discard;
    if( any( greaterThan( stp, vec3(1.,1.,1.) ) ) ) discard;
    float scalar = texture( uTexUnit, stp ).r; // data is hiding in the red component
    if( scalar < uMin || scalar > uMax ) discard;
    float t = ( scalar - SMIN ) / ( SMAX - SMIN );
    vec3 rgb = Rainbow( t );
    gl_FragColor = vec4( rgb, 1. );
}
```

A Problem with Uniform Pointclouds: Row-of-Corn and Moire Patterns

Uniform Points vs. Jittered Points
The shaders can potentially change:

- Color
- Alpha
- Pointsize

Now, change the Point Cloud geometry to a quadrilateral geometry. If we keep the coordinate range from -1. to 1., then the same shader code will work, except that we now want to base the color assignment on Eye Coordinates instead of Model Coordinates:

```glsl
in vec3 vEC;
void main( )
{
  vec3 stp = ( vEC + 1. ) / 2.;
  // maps [-1.,1.] to [0.,1.]
  // Note that the plane can be oriented at any angle because the a-t-p data lookup comes from the x-y-z coordinates of the cutting plane
}
```

This is very much like how we handled rendering a rainbow.

Let's say that we want “contour gaps” at each 10 degrees of temperature. Then the main change to the shader will be that we need to find how close each fragment’s interpolated scalar data value is to an even multiple of 10. To do this, we add this code to the fragment shader:

```glsl
float scalar10 = float( 10*int( (scalar+5.)/10. ) );
if( abs( scalar - scalar10 ) > uTol ) discard;
```

Notice that this uses a uniform variable called `uTol`, which is read from a slider and has a range of 0. to 5. `uTol` is used to determine how close to an even multiple of 10 degrees we will accept, and thus how thick we want the contour gaps to be.

Note that when `uTol=5.`, the `if-statement` always fails, and we end up with the same display as we had with the interpolated colors. Thus, we wouldn't actually need a separate color cutting plane shader at all. Shaders that can do double duty are always appreciated!
An Observation

Note that Point Clouds, Jitter Clouds, Colored Cutting Planes, Contour Cutting Planes, and 3D Data Probes are really all the same technique!
They just vary in what type of geometry the data is mapped to. They use the same shader code, possibly with a switch between model and eye coordinates.

How about something less obvious like a torus?

Visualization Transfer Function — Relating Display Attributes to the Scalar Value

Frequency Histogram

Colors

Opacity

Scalar Value

Visualization — Don’t Send Colored Data to the GPU, Send the Raw Data and a Separate Transfer Function to the Fragment Shader

Use the GPU to turn the data into graphics on-the-fly

A Visualization Scenario

A thermal analysis reveals that a bar has a temperature of 0° at one end and 100° at the other end:

You want to color it with a rainbow scale as follows:

You also want to use smooth shading, so that you can render the bar as a single quadrilateral.

Should you assign colors first then interpolate, or interpolate first then assign colors?

Will it matter? If so, how?

Point Clouds – Three Ways to Assign the Transfer Function

Assign colors from temperatures, then interpolate:

Assign attribute values first

Interpolate temperatures first, then assign colors:

Assigning colors first — problems with interpolation

Conclusion: let the rasterizer interpolate your temperatures and let your fragment shader assign your colors
Point Clouds – A Third Way – I really like this one

```glUseProgram(AssignTransferFunction);
```

```glUseProgram( AssignTransferFunction );
```

```glBegin( GL_POINTS );
```

```glVertex4f( x0, y0, z0, s0 );
```

```glEnd( );
```

*Vertex Shader:

```out float vScalar;
```

```void main() {
```

```vScalar = gl_Vertex.w;
```

```gl_Position = gl_ModelViewProjectionMatrix * vec4(gl_Vertex.xyz, 1.);
```

```}
```

**Hiding** the scalar value in the w component

```Don't want problems with dividing by the wrong w -- replace it before the pipeline uses it
```

```vScalar = gl_Vertex.w;
```

```gl_Position = gl_ModelViewProjectionMatrix * vec4(gl_Vertex.xyz, 1.);
```

```}
```

Volume Rendering – Compositing via Ray Casting

**Thinking about it back-to-front:**

```color2 = a2color2 + (1 - a2)black,
```

```color01 = a1color01 + (1 - a1)color2,
```

```color* = a1color01 + (1 - a1)color01.
```

Gives the front-to-back equation:

```color* = a2color2 + (1 - a2)black + (1 - a2)(1 - a1)color2 + (1 - a2)a1color01 + (1 - a1)(1 - a2)a1color01.
```

```gl_FragColor = vec4( cstar, 1. );
```

Volume Rendering – a different way to think of visualizing 3D Scalar Data

Each voxel has a color and opacity depending on its scalar value

```Scalar Value
```

Volume Filtering – Median Filter

Visualization by Ankit Khare
Volume Filtering – High Pass Filter Followed by Median Filter

Volume Visualization with OSU’S College of Vet Medicine

A Vector Visualization Technique: Hedgehog Plots

Vector Visualization: 2D Line Integral Convolution

At each fragment:
1. Find the flow field velocity vector there
2. Follow that vector in both directions
3. Blend in the colors at the other fragments along that vector

Vector Visualization: 2D Line Integral Convolution

uniform int uLength;
uniform sampler2D uImageUnit;
uniform sampler2D uFlowUnit;
uniform float uTime;
in vec2 vST;
void main( )
{
  ivec2 res = textureSize( uImageUnit, 0 );
  // flow field direction:
  vec2 st = vST;
  vec2 v = texture( uFlowUnit, st ).xy;
  v *= 1./vec2(res);
  st = vST;
  vec3 color = texture( uImageUnit, st ).rgb;
  int count = 1;
  st = vST;
  for( int i = 0; i < uLength; i++ )
  {
    st += uTime*v;
    vec3 new = texture( uImageUnit, st ).rgb;
    color += new;
    count++;
  }
  st = vST;
  for( int i = 0; i < uLength; i++ )
  {
    st -= uTime*v;
    vec3 new = texture( uImageUnit, st ).rgb;
    color += new;
    count++;
  }
  color /= float(count);
gl_FragColor = vec4( color, 1. );
}
Flow around a corner

Flow in a circle

Vector Visualization: 2D Line Integral Convolution

Flow in a circle

http://hint.fm/wind/

Vector Visualization: 2D Line Integral Convolution

Vector Visualization: 3D Line Integral Convolution

Visualizations by Vasu Lakshmanan

Parameterize the shape and recast it into T-N-B coordinates along the flowline, P(t)

Tangent:

$$T(t) = \frac{P(t)}{\|P(t)\|}$$

Binormal:

$$B(t) = \frac{P(t) \times T(t)}{\|P(t) \times T(t)\|}$$

Normal:

$$N(t) = B(t) \times T(t)$$

This are known as the three Frenet Equations and are very useful for geometrically characterizing what is happening on a curve.

Extruding Shapes Along Flow Lines:

As long as you are writing a shader anyway, ...

Add bump-mapping to aid in understanding the orientation.

Add moving “humps” to create a peristaltic effect.

Terrain Height Bump-mapping

No Exaggeration

Exaggerated