Using Shaders to Enhance Scientific Visualizations

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

Visualization Imaging -- Sharpening

The negative of a 3D object often reveals details
Changing the emboss angle is interesting.

Using the GPU to enhance scientific, engineering, and architectural illustration.
Visualization -- Polar Hyperbolic Space

Use the GPU to perform nonlinear vertex transformations

\[ \Theta' = \Theta \]
\[ R' = \frac{R}{R + K} \]

Dome Projection for Immersive Visualization

Use the GPU to perform nonlinear vertex transformations

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

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 );
        }
    }
}
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 straightforward 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 $s$ texture coordinate goes from 0. to 1., then the linear mapping from the physical $x$ coordinate to the texture $s$ 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 conversion can be done in one line of code using the vec3:

```glsl
vec3 xyz = ??? . . .
vec3 stp = ( xyz + 1. ) / 2.;
```

You can also go the other way:

```glsl
vec3 xyz = -1. + ( 2. * 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. );
}'''

SIMD functions to help GLSL if-tests

```glsl
vec3 smp = vec3(0.); // maps [-1.,1.] to [0.,1.]
```
A Problem with Uniform Pointclouds: Row-of-Corn and Moire Patterns

Uniform Points vs. Jittered Points

Uniform Points vs. Jittered Points

Enhanced Point Clouds

The shaders can potentially change:
- Color
- Alpha
- Pointsize

Color Cutting Planes

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.]
  // ...

  Eye (transformed) coordinates are being used here because the cutting plane is moving through the data.

  Note that the plane can be oriented at any angle because the s-t-p data lookup comes from the transformed x-y-z coordinates of the cutting plane.
```
The cutting plane is actually just being used as a **fragment-generator**. Each fragment is then being asked “what data value lives at the same place you live”?

```glsl
in vec3 vEC;
void main( )
{
  vec3 stp = ( vEC + 1. ) / 2.; // maps [-1.,1.] to [0.,1.]
  . .
}
```

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 discretization 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 `uTol` 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!

3D Data Probe – Mapping the Data to Arbitrary Geometry

The cutting plane is actually being used as a fragment-generator. Each fragment is then being asked “what data value lives at the same place you live”?

Some shapes make better probes than others do...
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 colored 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?
A Visualization Scenario

Assign colors from temperatures, then interpolate:

Interpolate temperatures first, then assign colors:

Conclusion: let the rasterizer interpolate your scalar values and let your fragment shader assign colors and alphas to those values

Point Clouds – Three Ways to Assign the Scalar Function

1. Assigning colors first – problems with interpolation
2. Assigning attribute values first

Point Clouds – A Third Way – I really like this one

3. “Hiding” the scalar value in the w component

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

Each voxel has a color and opacity depending on its scalar value
Thinking about it back-to-front:

\[ \text{color}_{12} = \alpha_2 \text{color}_2 + (1 - \alpha_2) \text{black}, \]

\[ \text{color}_{01} = \alpha_1 \text{color}_1 + (1 - \alpha_1) \text{color}_{12}, \]

\[ \text{color}^* = \alpha_0 \text{color}_0 + (1 - \alpha_0) \text{color}_{01}. \]

Gives the front-to-back equation:

\[ \text{color}^* = \alpha_0 \text{color}_0 + (1 - \alpha_0) \text{color}_1 + (1 - \alpha_0)(1 - \alpha_1) \text{color}_2 + (1 - \alpha_0)(1 - \alpha_1)(1 - \alpha_2) \text{black}. \]

\[
\text{float\hspace{1em}astar} = 1.;
\text{vec3\hspace{1em}cstar} = \text{vec3}(0.,0.,0.);
\text{for( int \hspace{1em}i \hspace{1em}= \hspace{1em}0; \hspace{1em}i < \text{uNumSteps}; \hspace{1em}i++, \hspace{1em}STP += \text{uDirSTP}) \{ 
\text{if( \hspace{1em}any( \hspace{1em}\text{lessThan}( \hspace{1em}STP, \hspace{1em}\text{vec3}(0.,0.,0.))) \hspace{1em})
\text{continue;}
\text{if( \hspace{1em}any( \hspace{1em}\text{greaterThan}( \hspace{1em}STP, \hspace{1em}\text{vec3}(1.,1.,1.))) \hspace{1em})
\text{continue;}
\text{float\hspace{1em}scalar} = \text{texture3D}( \text{uTexUnit}, \text{STP}).r;
\text{if( \hspace{1em}scalar < \text{uMin} \hspace{1em})
\text{continue;}
\text{if( \hspace{1em}scalar > \text{uMax} \hspace{1em})
\text{continue;}
\text{float\hspace{1em}alpha} = \text{uAmax};
\text{float\hspace{1em}t} = ( \hspace{1em}\text{scalar} - \text{SMIN} \hspace{1em}) / ( \text{SMAX} - \text{SMIN} \hspace{1em});
\text{vec3\hspace{1em}rgb} = \text{Rainbow}(\hspace{1em}t\hspace{1em});
\text{cstar} += \text{astar} * \text{alpha} * \text{rgb};
\text{astar} *= ( \hspace{1em}1. - \text{alpha} \hspace{1em});
\text{// break out if the rest of the tracing won't matter:}
\text{if( \hspace{1em}astar == 0. \hspace{1em})
\text{break;}
\}
\text{gl\_FragColor} = \text{vec4}(\text{cstar}, 1.).\]
Volume Filtering – High Pass Filter Followed by Median Filter

Volume Visualization with OSU’S College of Vet Medicine

Visualization by Ankit Khare

Visualization by Chris Schultz

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

Use a vector field equation, or "hide" the velocity field in another texture image: \((v_x,v_y,v_z) \equiv (r,g,b)\)

Image

Circular Flow Field

```
ivec2 res = textureSize( uImageUnit, 0 );
vec2 st = vST;
vec2 v = texture( uFlowUnit, st ).xy;
v *= 1./vec2(res);
st = vST;
vec3 color = texture( uImageUnit, st ).rgb;
int count = 1;
```
Vector Visualization: 2D Line Integral Convolution

lic2d.frag, II

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. );

Vector Visualization: 2D Line Integral Convolution

Flow around a corner

Flow in a circle

Vector Visualization: a Cool 2D Line Integral Convolution Example

http://hint.fm/wind/

Vector Visualization: 3D Line Integral Convolution

Visualizations by Vasu Lakshmanan