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

Visualization Imaging -- Embossing

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

A Vector Visualization Technique: Hedgehog Plots

Terrain Height Bump-mapping

Bump-Mapping for Terrain Visualization

3D Object Silhouettes
Use the GPU to perform nonlinear vertex transformations.

$$\Theta' = \Theta$$

$$R' = \frac{R}{R + K}$$

**Visualization -- Polar Hyperbolic Space**

**Dome Projection for Immersive Visualization**

Use the GPU to perform nonlinear vertex transformations.

**Image Manipulation Example -- Where is it Likely to Snow?**

```cpp
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**

```cpp
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:

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

You can also go the other way:

```
vec3 xyz = -1. + (2. * stp);
```
The Vertex Shader

out vec3 vMC;
void main() {
  vMC = gl_Vertex.xyz;
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

The Fragment Shader

uniform float uMin, uMax;
uniform sampler3D uTexUnit;
in vec3 vMC;
const float SMIN = 0.;
const float SMAX = 100.;
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

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

Uniform Points vs. Jittered Points

"Pointcloud"

"Jittercloud"

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.

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.

Enhanced Point Clouds

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

```
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:

```
fint 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!

Some shapes make better probes than others do. ☝

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
```

OSU
vx
Transfer Function Sculpting Window

Visualization Transfer Function – Relating Display Attributes to the Scalar Value
A Visualization Scenario

Assign colors from temperatures, then interpolate:

Wrong!

Interpolate temperatures first, then assign colors:

Right!

Conclusion: let the rasterizer interpolate your scalar values and let your fragment shader assign colors and alphas to those values.
Volume Rendering – Compositing via Ray Casting

Thinking about it back-to-front:

\[ \text{color}_{o} = \alpha \text{color}_{i} + (1 - \alpha) \text{black}, \]
\[ \text{color}_{n} = \alpha \text{color}_{i} + (1 - \alpha) \text{color}_{o}, \]
\[ \text{color}' = \alpha \text{color}_{o} + (1 - \alpha) \text{color}_{i}. \]

Gives the front-to-back equation:

\[ \text{color}' = \alpha \text{color}_{i} + (1 - \alpha) \text{color}_{o} + (1 - \alpha)(1 - \alpha)(1 - \alpha) \text{black}. \]

Volume Filtering – Median Filter

Visualization by Ankit Khare

Volume Filtering – High Pass Filter Followed by Median Filter

Visualization by Ankit Khare

Volume Visualization with OSU'S College of Vet Medicine

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

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

```glsl
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;

  for( int i = 0; i < uLength; i++ )
  {
    st += uTime*v;
    vec3 new = texture( uImageUnit, st ).rgb;
    color += new;
    count++;
  }

  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

http://hint.fm/wind/

Visualizations by Vasu Lakshmanan