Using Shaders to Enhance Scientific Visualizations

Mike Bailey
mjb@cs.oregonstate.edu
Oregon State University

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

1. Render a 3D dynamic scene
2. Lighting Shader

Pass #1
- Render a 3D dynamic scene
- Texture

Pass #2
- Render a quadrilateral
- Framebuffer
- Blur Shader
The negative of a 3D object often reveals details.

Embossing

Changing the emboss angle is interesting.
Visualization Imaging -- Sharpening

Visualization Imaging – Edge Detection
Non-Photorealistic Rendering – Toon Rendering

Use the GPU to enhance scientific and engineering 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

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

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

```c
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

Low values culled

Full data
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 $s$ 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 can be done in one line of code:

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

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 = 0.0;
const float SMAX = 120.0;

void main() {
    vec3 stp = (vMC + 1.0) / 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

Perspective

Orthographic
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.]
    ...
}
```

Note that the plane can be oriented at any angle because the s-t-p data lookup comes from the x-y-z coordinates of the cutting plane.

Contour Cutting Planes

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.
Contour Cutting Planes are Also Color Cutting Planes

Note that when uTol=5., the uTol if-statement

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

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

Some shapes make better probes than other 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

OSU vx Transfer Function Sculpting Window
Visualization -- Don't Send Colored Data to the GPU, Send the Raw Data and a Separate Transfer Function

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 0º at one end and 100º at the other end:

0º  100º

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

0º  100º

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:

Wrong!

Interpolate temperatures first, then assign colors:

Right!

Conclusion: let the rasterizer interpolate your temperatures and let your fragment shader assign your colors.

Point Clouds – Three Ways to Assign the Transfer Function

1. `glBegin( GL_POINTS );
   < convert s0 to r0, g0, b0, a0 >
   glColor4f( r0, g0, b0, a0 );
   glPointSize( p0 );
   glVertex3f( x0, y0, z0 );
   . . .
   glEnd( );`

   Assigning colors first – problems with interpolation

2. `glUseProgram( AssignTransferFunction );
   glBegin( GL_POINTS );
   glVertexAttrib1f( location, s0 );
   glVertex3f( x0, y0, z0 );
   . . .
   glEnd( );`

   Assigning attribute values first
Point Clouds – A Third Way – I really like this one

```c
glUseProgram( AssignTransferFunction );
glBegin( GL_POINTS );
glVertex4f( x0, y0, z0, s0 );
...
glEnd();
```

"Hiding" the scalar value in the w component

```c
out float vScalar;

void main( )
{
    vScalar = gl_Vertex.w;
    gl_Position = gl_ModelViewProjectionMatrix * vec4(gl_Vertex.xyz, 1.);
}
```

Don’t want problems with dividing by the wrong w – replace it before the pipeline uses it

Volume Rendering – Compositing via Ray Casting

Thinking about it back-to-front:

\[
\text{color}_{ij} = \alpha_i \text{color}_j + (1 - \alpha_i) \text{black},
\]

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

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

Gives the front-to-back equation:

\[
\text{color}^* = \alpha_i \text{color}_i + (1 - \alpha_i) \alpha_j \text{color}_j + (1 - \alpha_j) \alpha_i \text{color}_i + (1 - \alpha_i)(1 - \alpha_j) \text{color}_i + (1 - \alpha_i)(1 - \alpha_j) \text{black}.
\]

(This is the same as what RenderMan does...)
float astar = 1.;
vec3 cstar = vec3(0., 0., 0.);
for(int i = 0; i < uNumSteps; i++, STP += uDirSTP) {
    if( any( lessThan( STP, vec3(0.,0.,0.) ) ) ) continue;
    if( any( greaterThan( STP, vec3(1.,1.,1.) ) ) ) continue;
    float scalar = texture3D( uTexUnit, STP ).r;
    if( scalar < uMin ) continue;
    if( scalar > uMax ) continue;
    float alpha = uAmax;
    float t = ( scalar - SMIN ) / ( SMAX - SMIN );
    vec3 rgb = Rainbow( t );
    cstar += astar * alpha * rgb;
    astar *= (1. - alpha);
    // break out if the rest of the tracing won't matter:
    if( astar == 0. ) break;
}
gl_FragColor = vec4( cstar, 1. );

Volume Rendering – Compositing via Ray Casting

uMin = minimum scalar value to display
uMax = maximum scalar value to display
uAmax = alpha value to use if this voxel is to be seen
Volume Filtering – Median Filter

Volume Filtering – High Pass Filter Followed by Median Filter
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

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;
}
```

Image + Circular Flow Field = Image

lic2d.frag, I
lic2d.frag. II

```cpp
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. );
```
Visualization: 3D Line Integral Convolution

Visualizations by Vasu Lakshmanan
Extruding Shapes Along Flow Lines

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

- **Tangent:**
  \[
  T(t) = \frac{\ddot{P}(t)}{\|\ddot{P}(t)\|}
  \]

- **Binormal:**
  \[
  B(t) = \frac{\dot{P}(t) \times \ddot{P}(t)}{\|\dot{P}(t) \times \ddot{P}(t)\|}
  \]

- **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

Terrain Height Bump-mapping: Mipmapping Helps Zooming In and Out

Portland
Salem
Corvallis
Eugene
Bump-Mapping for Terrain Visualization

Visualization by Nick Gebbie

3D Object Silhouettes
Hedgehog Plots