Using Shaders to Enhance Scientific Visualizations

Mike Bailey
mjb@cs.oregonstate.edu

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

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

Pass #2
- Render a quadrilateral
- Framebuffer
- Sharpening Shader

Visualization Imaging – Sharpening

The negative of a 3D object often reveals details

Visualization Imaging – Edge Detection

Embossing

Changing the emboss angle is interesting

Visualization Imaging – Embossing

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

Using the GPU to enhance scientific, engineering, and architectural illustration

A Vector Visualization Technique: Hedgehog Plots

Terrain Height Bump-mapping

No Exaggeration

Exaggerated

Terrain Height Bump-mapping

3D Object Silhouettes

Visualization by Nick Gebbie
Use the GPU to perform nonlinear vertex transformations.

Use the GPU to perform nonlinear vertex transformations.

Use the GPU to perform nonlinear vertex transformations.

Use the GPU to perform nonlinear vertex transformations.

Use the GPU to perform nonlinear vertex transformations.

Use the GPU to perform nonlinear vertex transformations.

Use the GPU to perform nonlinear vertex transformations.

Use the GPU to perform nonlinear vertex transformations.

Use the GPU to perform nonlinear vertex transformations.

Use the GPU to perform nonlinear vertex transformations.

Use the GPU to perform nonlinear vertex transformations.

Use the GPU to perform nonlinear vertex transformations.

Use the GPU to perform nonlinear vertex transformations.

Use the GPU to perform nonlinear vertex transformations.
The Vertex Shader

cut 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 = 0.0;
const float SMAX = 120.0;

void main() {
    vec3 stp = (vMC + 1.0) / 2.0;  // 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, 1.0);
}

SIMD functions to help GLSL if-tests

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

Uniform Points vs. Jittered Points

"Pointcloud"

"Jittercloud"

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.

In vec3 vEC;
void main() {
    vec3 stp = (vEC + 1.0) / 2.0;  // 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?"

```cpp
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:

```cpp
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!

Contour Cutting Planes are Also Color Cutting Planes

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?"

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
Computer Graphics

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

0º 100º

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.

0º 100º

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:

WRONG!

Assign attribute values first:

RIGHT!

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

Assigning colors first – problems with interpolation

Assigning attribute values first

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

glUseProgram( AssignTransferFunction );

glBegin( GL_POINTS );

glVertex4f( x0, y0, z0, s0 );

...;

glEnd( );

“Hiding” the scalar value in the w component

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

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

Each voxel has a color and opacity depending on its scalar value.
Volume Rendering – Compositing via Ray Casting

Thinking about it back-to-front:

\[ \text{color}_{i+1} = \alpha \text{color}_i + (1-\alpha) \text{black}, \]
\[ \text{color}_0 = \alpha \text{color}_i + (1-\alpha) \text{color}_{i-1}, \]
\[ \text{color}^* = \alpha \text{color}_0 + (1-\alpha) \text{color}_{i-1}. \]

Gives the front-to-back equation:

\[ \text{color}^* = \alpha \text{color}_{i-1} + (1-\alpha) \text{color}_i + (1-\alpha)(1-\alpha) \text{color}_{i-2} + (1-\alpha)(1-\alpha) \text{color}_{i-3} + (1-\alpha)(1-\alpha)(1-\alpha) \text{black}. \]
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

```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++;
    }
    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.);
}
```

Compound Flow Field

Flow around a corner
Flow in a circle

Vector Visualization: 3D Line Integral Convolution

Visualizations by Vasu Lakshmanan