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

Visualization Imaging – Sharpening

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

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

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.

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

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 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 \(s\) 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 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() {
  vec3 xyz = 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;
  if (any(lessThan(stp, vec3(0,0,0)))) discard;
  if (any(greaterThan(stp, vec3(1,1,1)))) discard;
  float scalar = texture(uTexUnit, stp).r;
  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
Enhanced Point Clouds

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:

```
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.

Shaders that can do double duty are always appreciated!

Contour Cutting Planes are Also Color 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:

```
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.

3D Data Probe – Mapping the Data to Arbitrary Geometry

Some shapes make better probes than other shapes...
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?

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.

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

Point Clouds – Three Ways to Assign the Scalar Function

1

Assigning colors first – problems with interpolation

2

Assigning attribute values first

Assigning colors from temperatures, then interpolate:

Assign transfer function then draw points:

Wrong!

Interpolate temperatures first, then assign colors:

Correct!

Conclusion: let the rasterizer interpolate your scalar values and let your fragment shader assign colors and alphas to those values.
Point Clouds – A Third Way – I really like this one

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

“Hiding” the scalar value in the w component

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

Vertex Shader:

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

OSU

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

- `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);`
- `if (astar == 0.) break;`

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

Use a vector field equation, or “hide” the velocity field in another texture image: (vx,vy,vz) ≡ (r,g,b)

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.);
}
Computer Graphics

Flow around a corner

Flow in a circle

Vector Visualization: 2D Line Integral Convolution

http://hint.fm/wind/

Vector Visualization: 3D Line Integral Convolution

Bump-Mapping for Terrain Visualization

Terrain Height Bump-mapping

No Exaggeration

Exaggerated

Terrain Height Bump-mapping

Visualization by Vasu Lakshmanan

Visualization by Nick Gebbie
3D Object Silhouettes