Scalar Visualization

Mike Bailey
mjb@cs.oregonstate.edu

Oregon State University

In Visualization, we Use the Concept of a Transfer Function to set Color as a Function of Scalar Value

Color

Scalar Value

A Gallery of Color Scale Transfer Function Possibilities

We will cover this in more detail in the color notes.

Glyphs

Glyphs are small symbols that can be placed at the location of data points. In 2D, we often call this a scatterplot. The glyph itself can convey information using properties such as:

- Type
- Color
- Size
- Orientation
- Transparency
- Features

The OpenDX AutoGlyph function gives you these type options:

OpenDX Scalar Glyphs

<table>
<thead>
<tr>
<th>Square</th>
<th>Circle</th>
<th>Diamond</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LIGO Gravity Glyphs

Can also use shape to convey data-meaning

Hitting the secret Easter Egg key 😊
Using 3D Glyphs is called a Point Cloud

Good for overall patterns – bad for detail

Orthographic Projection results in the row-of-corn problem
Perspective Projection results in the Moiré problem

A Simple Point Cloud Data Structure

```c
struct node
{
  float x, y, z;
  float s;
  float r, g, b;
};
```

```c
struct node Nodes[NX][NY][NZ];
```

In OpenGL . . .

```c
float delx = (XMAX - XMIN) / (float)(NX-1);
float dely = (YMAX - YMIN) / (float)(NY-1);
float delz = (ZMAX - ZMIN) / (float)(NZ-1);
glPointSize( 2.);
glBegin( GL_POINTS );
float x = XMIN;
for( int i=0; i < NX; i++, x += delx )
{
  float y = YMIN;
  for( int j=0; j < NY; j++, y += dely )
  {
    float z = ZMIN;
    for( int k=0; k < NZ; k++, z += delz )
    {
      float scalar = Nodes[i][j][k].s;
      float r = ???;
      float g = ???;
      float b = ???;
      glColor3f( r, g, b );
      glVertex3f( x, y, z );
    }
  }
}
glEnd();
```

Computing x, y, and z

Note that x, y, and z can be computed at each node point by just keeping track of them and incrementing them each time through their respective loop, as shown on the previous page. They can also be computed from the loop index like this:

```c
float x = -1. + 2. * (float)i / (float)(NX-1);
float y = -1. + 2. * (float)j / (float)(NY-1);
float z = -1. + 2. * (float)k / (float)(NZ-1);
```

Jitter Gives a Better Point Cloud Display

Orthographic Projection
Perspective Projection

Point Cloud Culling Using Range Sliders

Low values culled
Using Range Sliders

```c
#define S 0
const char *SFORMAT = { "S: %.3f - %.3f");
float SLowHigh[2];
GLUI_StaticText *SLabel;
slider = Glui->add_slider( true, GLUI_HSLIDER_FLOAT, SLowHigh, S, (GLUI_Update_CB) Sliders );
slider->set_float_limits( SLowHigh[0], SLowHigh[1] );
slider->set_slider_val(   SLowHigh[0], SLowHigh[1] );
slider->set_w( SLIDERWIDTH );

#define S       0
const char *SFORMAT = { "S: %.3f - %.3f" };  
float   SLowHigh[2];
GLUI_StaticText *SLabel;
```

Drawing the Range Slider-Filtered Point Cloud

```c
void Sliders( int id )
{
    char str[256];
    switch( id )
    {
    case S:
        sprintf( str, SFORMAT, SLowHigh[0], SLowHigh[1] );
        SLabel->set_text( str );
        break;
    }
}
```

Enhanced Point Clouds

- Color
- Alpha
- Pointsize

Enhanced Point Clouds are nice, but they only tell us about the gross patterns. We want more detail!

Even though this is a 2D technique, we keep around the X, Y, and Z coordinates so that the grid doesn't have to lie in any particular plane.

2D Interpolated Color Plots

Here's the situation: we have a 2D grid of data points. At each node, we have an X, Y, Z, and a scalar value S. We know Smin, Smax, and the Transfer Function.

We deal with one square of the mesh at a time:

```
2D Interpolated Color Plots

Within that one square, we let OpenGL do the color interpolation for us

```c
void ColorSquare( . . .)
{
    Compute an r, g, b for S0
    glColor3f( r, g, b );
    glVertex3f( X0, Y0, Z0 );

    Compute an r, g, b for S1
    glColor3f( r, g, b );
    glVertex3f( X1, Y1, Z1 );

    Compute an r, g, b for S2
    glColor3f( r, g, b );
    glVertex3f( X2, Y2, Z2 );

    Compute an r, g, b for S3
    glColor3f( r, g, b );
    glVertex3f( X3, Y3, Z3 );
}
```

Note the order: 0-1-3-2 !

Then we loop through all squares:

```c
glShadeModel( GL_SMOOTH );
glBegin( GL_QUADS );
for( int i = 0;  i < numT - 1;  i++ )
{
    for( int j = 0;  j < numU-1;  j++ )
    {
        ColorSquare( i, j, ... );
    }
}
glEnd( );
```

2D Contour Lines

Here's the situation: we have a 2D grid of data points. At each node, we have an X, Y, Z, and a scalar value S. We know the Transfer Function. We also have a particular scalar value, S*, at which we want to draw the contour line(s).

Even though this is a 2D technique, we keep around the X, Y, and Z coordinates so that the grid doesn’t have to lie in any particular plane.

Hiking Maps are a Great Use for Contour Lines

Hiking Maps are a Great Use for Contour Lines

Even though this is a 2D technique, we keep around the X, Y, and Z coordinates so that the grid doesn’t have to lie in any particular plane.

Hiking Maps are a Great Use for Contour Lines

Even though this is a 2D technique, we keep around the X, Y, and Z coordinates so that the grid doesn’t have to lie in any particular plane.

2D Contour Lines: Marching Squares

Rather than deal with the entire grid, we deal with one square at a time, marching through them all. For this reason, this method is called the Marching Squares.
Marching Squares: A Cluster of Connected Line Segments

What's really happening is that we are not creating contours by connecting points into a complete curve. We are creating contours by drawing a collection of 2-point line segments, safe in the knowledge that those line segments will align across square boundaries.

Does $S^*$ cross any edges of this square?

Linearly interpolating the scalar value from node 0 to node 1 gives:

$$S = (1-t)S_0 + tS_1 = S_0 + t(S_1 - S_0)$$

where $0 \leq t \leq 1$.

Setting this interpolated $S$ equal to $S^*$ and solving for $t$ gives:

$$t^* = \frac{S^* - S_0}{S_1 - S_0}$$

Interpreting $t^*$: Where does $S^*$ cross the edge?

If $0 \leq t^* < 1$, then $S^*$ crosses this edge. You can compute where $S^*$ crosses the edge by using the same linear interpolation equation you used to compute $t^*$:

$$X^* = X_0 + t^*(X_1 - X_0)$$
$$Y^* = Y_0 + t^*(Y_1 - Y_0)$$
$$Z^* = Z_0 + t^*(Z_1 - Z_0)$$

If $t^* < 0$, then $S^*$ crosses this edge. If $t^* > 1$, then $S^*$ crosses this edge.

If $0 \leq t^* < 1$, then $S^*$ crosses this edge. If $t^* > 0$, then $S^*$ crosses this edge.

What if $S_1 == S_0$ (i.e., $t^*=?$)

Surprisingly, you just ignore this edge. Why? There are 2 possibilities. Let $S^* = 50$

$S_1 = S_0 = S^*$

- The edges with these intersections create 2 points
- The edges with these intersections create 2 points

Ignore this edge

The 4-intersection Case

If there are 4 edge intersections with $S^*$, then this must mean that, going around the square, the nodes are $>S^*$, $<S^*$, $>S^*$, and $<S^*$ in that order. This gives us a saddle function, shown here in cyan.

If we think of the scalar values as terrain heights, then we can think of $S^*$ as the height of water that is flooding the terrain, as shown here in magenta.
The 4-intersection Case: Computing the middle scalar value, M

Let’s linearly interpolate scalar values along the 0-1 edge, and along the 2-3 edge:

\[ S_{01}(t) = (1-t)S_0 + tS_1 \]
\[ S_{23}(t) = (1-t)S_2 + tS_3 \]

Now linearly interpolate these two linearly interpolated scalar values:

\[ S(t, u) = (1-u)S_{01} + uS_{23} \]

Expanding gives:

\[ S(t, u) = (1-t)(1-u)S_0 + t(1-u)S_1 + (1-t)uS_2 + tuS_3 \]

This is the bilinear interpolation equation. Notice the similarity to the linear equation.

The 4-intersection Case: Overall Logic for a Set of Contour Lines

```c
for( float S* = Smin ;  S* <= Smax ;  S* += ∆S ) {
    Set color for S*
    glBegin( GL_LINES );
        for( int i = 0;  i < numT - 1;  i++ )
            for( int j = 0;  j < numU-1;  j++ )
                Process the square whose corner is at (i,j);
    glEnd( );
}
```

Note that it is bad programming practice to use a floating-point variable to index the S* for-loop! This has been done just to illustrate the concept. Instead do this:

```c
int is;  float S*;
for( is = 0;  S* = Smin;  is = numS;  i = is;  S* += ∆S ) {
    …
}
```
Artifacts?

What if the distribution of scalar values along the square edges isn’t linear?

We have no basis to assume anything, actually. So linear is as good as any other guess, and lets us consider just one square by itself. Some people like looking at adjacent nodes and using quadratic or cubic interpolation on the edge. This is harder to deal with computationally, and is also making an assumption for which there is no evidence.

What if you have a contour that really looks like this?

You’ll never know. We can only deal with what data we’ve been given. There is no substitute for having an adequate number of data points.

What if we subdivide the square and interpolate values? Does that help?

No. We can only deal with what data we’ve been given.

While we’re at it:

Trilinear interpolation

This is useful, for example, if we have passed an oblique cutting plane through a 3D mesh of points and are trying to interpolate scalar values from the 3D mesh to the 2D plane.

Wireframe isosurfaces

Here’s the situation: we have a 3D grid of data points. At each node, we have an X, Y, Z, and a scalar value S. We know the Transfer Function. We also have a particular scalar value, \( S^* \), at which we want to draw the isosurface(s).

Once you have done Marching Squares for contour lines, doing wireframe isosurfaces is amazingly easy. If it had to come up with a name for this, I’d call it Marching Planes.

The strategy is that you pick your \( S^* \), then draw \( S^* \) contours on all the parallel \( XY \) planes. Then draw \( S^* \) contours on all the parallel \( XZ \) planes. And, then you’re done.

What you have looks like it is a connected surface mesh, but in fact it is just independent curves. It is easy to program (once you’ve done Marching Squares at least), and looks good. Also, it’s fast to compute.

And, of course, if you can do it in one plane, you can do it in multiple planes

Remember this! In a moment, we are going to put this to use in a different way, to create wireframe isosurfaces . . .

Overall Logic for a Wireframe Isosurface

```c
// Set color for S*
glColor4f(0.0f, 0.0f, 0.0f, 1.0f);

for(int k = 0; k < numV; k++)
{
    for(int i = 0; i < numT - 1; i++)
    {
        for(int j = 0; j < numU-1; j++)
        {
            Process square whose corner is at (i,j,k) in TU plane
        }
    }
}

for(int i = 0; i < numT ; i++)
{
    for(int k = 0; k < numV - 1; k++)
    {
        for(int j = 0; j < numU-1; j++)
        {
            Process square whose corner is at (i,j,k) in UV plane
        }
    }
}

for(int j = 0; j < numU; j++)
{
    for(int i = 0; i < numT - 1; i++)
    {
        for(int k = 0; k < numV-1; k++)
        {
            Process square whose corner is at (i,j,k) in TV plane
        }
    }
}

glEnd();
```
The original polygonal isosurface Marching Cubes algorithm used the observation that when classifying each corner node as $S^+$ or $S^-$, there were $2^6 = 64$ possible ways that it could happen, but of those 64, there were only 15 unique cases which needed to be handled. Even so, this is difficult, and so we will look at another approach.

Strategy in Polygonal Isosurfaces: Algorithm

1. Look through the $\text{FoundEdgeConnection}[12][12]$ array to find which cube edges have $S^+$ intersections. If no intersections were found anywhere, return.
2. Call $\text{ProcessCubeQuad()}$ 6 times to decide which cube edges will need to connect. This is Marching Squares like we did it before, but it doesn’t need to re-compute intersections on the cube edges in common.
3. Call $\text{DrawCubeTriangles()}$ to create triangles from the connected edges.

Strategy for $\text{DrawCubeTriangles()}$:

1. Look through the $\text{FoundEdgeIntersection}[12]$ array for a Cube Edge #A and a Cube Edge #B.
2. Turn to $\text{NodeIntersection}[12]$ entries for the interpolated $y, x, nx, ny, nz$. If an intersection did occur on edge #i, Node #i will contain the entry $\text{FoundEdgeIntersection}[12][12]$ array filled.
3. Process Cube Edge #A.
5. Draw Cube Triangles. This leaves us with the $\text{FoundEdgeConnection}[12][12]$ array filled.
6. Turn to $\text{NodeIntersection}[12]$ entries from Cube Edge #A to Cube Edge #B.
7. Toggle the $\text{FoundEdgeConnection}[12][12]$ entries from Cube Edge #B to Cube Edge #C.
8. $\text{DrawCubeTriangles()}$ if true, otherwise do nothing. Note that this algorithm will eventually find and properly connect the little triangle in the upper-right corner, even though it has no connection with A-B-C-D.

When we toggle the $\text{FoundEdgeConnection}[12][12]$ entries for AB and BC, they turn from true to false. When we toggle the $\text{FoundEdgeConnection}[12][12]$ for CA, it turns from false to true.

This leaves the $\text{FoundEdgeConnection}[12][12]$ for CA, CD, and AD all set to true, which will cause the algorithm to find them and connect them into a triangle next.

Note that this algorithm will eventually find and properly connect the little triangle in the upper-right corner, even though it has no connection with A-B-C-D.
We would very much like to use lighting when displaying polygonal isosurfaces, but we need surface normals at all the triangle vertices. Because there really isn’t a surface there, this would seem difficult, but it’s not.

Envision a balloon with a dot painted on it. Think of this balloon as an isosurface. Blow up the balloon a little more. This is like changing $S^*$, resulting in a different isosurface. Where does the dot end up?

The dot moves in the direction of the changing isosurface, which is the normal to the balloon surface.

Now, turn that sentence around:

The normal to the isosurface is a vector that shows how the isosurface is changing.

How “something is changing” is called the gradient. So, the surface normal to a volume is:

$$\mathbf{n} = \frac{\mathbf{dS}}{\mathbf{dx}} \frac{\mathbf{dS}}{\mathbf{dy}} \frac{\mathbf{dS}}{\mathbf{dz}} = \nabla S$$

Prior to the isosurface calculation, you compute the surface normals for all the nodes in the 3D mesh. You then interpolate them along the cube edges when you create the isosurface triangle vertices.