Scalar Field Visualization I
What is a Scalar Field?

- The approximation of certain scalar function in space \( f(x,y,z) \).
What is a Scalar Field?

• The approximation of certain scalar function in space $f(x,y,z)$.

• Most of time, they come in as some scalar values defined on some sample points.
What is a Scalar Field?

• The approximation of certain scalar function in space $f(x,y,z)$.

• Most of time, they come in as some scalar values defined on some sample points.

• Visualization primitives:
  – Geometry:
    • iso-contours (2D), iso-surfaces (3D),
  – Attributes:
    • colors, transparency, 3D textures.
Generate 2D color plots
Steps of generating 2D color plots

1. Design color transfer function
2. Color interpolation
To create a color plot, we need to define a proper **Transfer Function** to set **Color** as a function of Scalar Value. The following shows a simple transfer function.

Scalar values ->[0,1] -> Colors

In OpenGL, the mapping of 1D texture
To create a color plot, we need to define a proper *Transfer Function* to set *Color* as a function of Scalar Value. The following shows a simple transfer function.

\[
t = \frac{S - S_{\text{min}}}{S_{\text{max}} - S_{\text{min}}}
\]

Scalar values \( \to [0,1] \)
To create a color plot, we need to define a proper **Transfer Function** to set **Color** as a function of Scalar Value. The following shows a simple transfer function.

\[\text{Ex. Rainbow color} \]

0 -> blue (hue=240)
\[t \rightarrow hue = 240 - 240 \cdot t\]
1 -> red (hue=0)

\[[0,1] \rightarrow \text{Colors}\]
To create a color plot, we need to define a proper *Transfer Function* to set *Color* as a function of Scalar Value. The following shows a simple transfer function.

$$Hue = 240. - 240. \frac{S - S_{min}}{S_{max} - S_{min}}$$

Verify low and high!

Scalar values -> [0,1] -> Colors

In OpenGL, the mapping of 1D texture
Use the Right **Transfer Function** Color Scale to Represent a Range of Scalar Values
Rainbow Scale

\[
\text{Hue} = 240. - 240. \frac{S - S_{\text{min}}}{S_{\text{max}} - S_{\text{min}}}
\]
Gray Scale

How to generate gray scale color scheme?
Intensity and Saturation Color Scales

How to achieve the above color schemes?
Two-Color Interpolation

How to achieve the above color scheme?
Heated Object Color Scale

Implementation: add one color component at a time

R+G+B
Add-One-Component at a time -- an extension from the heated object color scale
Blue-White-Red Color Scale

Divergent color scheme!
Blue-White-Red Color Scale

How to achieve it?
(Discrete) Color Scale Contour

Qualitative color scheme

Source: http://glscene.sourceforge.net
Generate 2D color plots

1. Color transfer function
2. Color interpolation
2D Interpolated Color Plots

- How can we turn the discrete samples into a continuous color plot?
- Here’s the input: we have a 2D grid of data points. At each node, we have an X, Y, Z, and a scalar value S. We know $S_{\text{min}}$, $S_{\text{max}}$, and a Transfer Function.

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

- Let us look at one **square (or quad)** of the mesh at a time

For each scalar value at a vertex

```c
float hsv[3], rgb[3];
hsv[0] = 240. - 240. * (s - s_min) / (s_max - s_min);
HsvRgb (hsv, rgb);
```

Convert hsv color to rgb color
2D Interpolated Color Plots

- We let OpenGL deal with the color interpolation

```c
// compute color at V0
glColor3fv (rgb0);
glVertex3f (x0, y0, z0);

// compute color at V1
glColor3fv (rgb1);
glVertex3f (x1, y1, z1);

// compute color at V3
glColor3fv (rgb3);
glVertex3f (x3, y3, z3);

// compute color at V2
glColor3fv (rgb2);
glVertex3f (x2, y2, z2);
```

Choose GL_SMOOTH rendering mode
What if the data is defined on a triangle mesh?
// compute color at V0
glColor3fv (rgb0);
glVertex3f (x0, y0, z0);

// compute color at V1
glColor3fv (rgb1);
glVertex3f (x1, y1, z1);

// compute color at V3
glColor3fv (rgb3);
glVertex3f (x3, y3, z3);
Geometric-Based Scalar Field Visualization
Iso-Contouring and Iso-Surfacing
2D Contour Lines

- **Contour (iso-value) line(s)**
  - Sub-sets of the original data that correlate all the points with the same scalar values.
  - If the 2D scalar field is considered as a height field (2D surface), the contours are the intersections of a moving horizontal plane with this height field.
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 (iso-value) line(s).*
2D Contour Lines: Marching Squares

• Instead of dealing with the entire grid, we once again look at one square at a time, then march through them all in order. For this reason, this method is called the Marching Squares.
Marching Squares

• What’s really going to happen 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.
Marching Squares

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

We have a particular scalar value, $S^*$, at which we want to draw the contour (iso-value) line(s).

Linearly interpolating any scalar value from node0 to node1 gives:

$$S = (1 - t)S_0 + tS_1 = S_0 + t(S_1 - S_0) \text{ where } 0. \leq t \leq 1.$$

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

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

If $0. \leq t^* \leq 1.$, then $S^*$ crosses this edge. You can compute where $S^*$ crosses the edge by using the same linear interpolation equation you used to compute $S^*$. You will need that for later visualization.

The coordinates of the intersection:

$$x^* = (1 - t^*)x_0 + t^*x_1$$
$$y^* = (1 - t^*)y_0 + t^*y_1$$
Marching Squares

• **Do this for all 4 edges** – when you are done, there are 5 possible ways this could have turned out
  
  – # of intersections = 0

  – # of intersections = 2

  – # of intersections = 1

  – # of intersections = 3
Marching Squares

• **Do this for all 4 edges** – when you are done, there are 5 possible ways this could have turned out

  – # of intersections = 0  Do nothing
  
  – # of intersections = 2  Draw a line connecting them
  
  – # of intersections = 1  Error: this means that the contour got into the square and never got out
  
  – # of intersections = 3  Error: this means that the contour got into the square and never got out

Marching Squares

Special cases

What if $S_1 == S_0$ (i.e. $t^* = \infty$) because

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

There are two possibilities.

\[ S_1 == S_0 != S^* \]

Intersections with these edges create 2 points

Ignore this edge

\[ S_1 == S_0 == S^* \]

Intersections with these edges create 2 points
Marching Squares

Special cases

What if there are four intersections

This means 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. The plane in magenta represents the plane with scalar value S*.

The intersection of this plane with the saddle function gives rise to the contours (in orange).
Marching Squares

The 4-intersection case

The exact contour curve is shown in orange. The Marching Squares contour line is shown in green. Notice what happens as we lower $S^*$ -- there is a change in which sides of the square get connected. That change happens when $S^* > M$ becomes $S^* < M$ (where $M$ is the middle scalar value).
Marching Squares

The 4-intersection case: Compute the middle scalar value

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

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

Now linearly interpolate these two linearly-interpolated scalar values:

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

Expand this we get

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

This is the bilinear interpolation equation.
Marching Squares

The 4-intersection case: Compute the middle scalar value

The middle scalar value, $M$, is what you get when you set $t = .5$ and $u = .5$:

$$ S(t, u) = (1 - t)(1 - u)S_0 + t(1 - u)S_1 + (1 - t)uS_2 + tuS_3 $$

$$ M = S(.5, .5) = \frac{1}{4}S_0 + \frac{1}{4}S_1 + \frac{1}{4}S_2 + \frac{1}{4}S_3 = \frac{S_0 + S_1 + S_2 + S_3}{4} $$

Thus, $M$ is the simple average of the four corner scalar values.
Marching Squares

The logic for the 4-intersection case is as follows:

1. Compute M
2. If S0 is on the same side of M as $S^*$ is, then connect the 0-1 and 0-2 intersections, and the 1-3 and 2-3 intersections.
3. Otherwise, connect the 0-1 and 1-3 intersections, and the 0-2 and 2-3 intersections.
Marching Squares Algorithm

Can you summarize the marching squares algorithm based on what we just discussed?
Marching Squares Algorithm

Can you summarize the marching squares algorithm based on what we just discussed?

```c
intersection_counter = 0;
for each edge of a square Si {
  //Determine whether there is an intersection given s*
  if (s0 < s* < s1 || s1 < s* < s0) {
    // compute the intersection
    intersection_counter ++;
  }
}
//According to the value of intersection_counter, connect the intersections
```
Marching Squares Algorithm

Can you summarize the marching squares algorithm based on what we just discussed?

```python
for all squares{
    intersection_counter = 0;
    for each edge of a square Si {
        //Determine whether there is an intersection given s*
        if (s0 < s* < s1 || s1<s*s<s0)
            { // compute the intersection
                intersection_counter ++;
            }
    }
    //According to the value of intersection_counter, connect the intersections
}
```
Marching Squares Algorithm

Can you summarize the marching squares algorithm based on what we just discussed?

for all squares{
    intersection_counter = 0;
    for each edge of a square Si {
        //Determine whether there is an intersection given s*
        if (s0 < s* < s1 || s1<s*<s0)
            { // compute the intersection
                intersection_counter ++;
            }
    }
    //According to the value of intersection_counter, connect the intersections
}

Handling special cases, e.g. s0==s1
Marching Squares Algorithm

Can you summarize the marching squares algorithm based on what we just discussed?

for all squares{
    intersection_counter = 0;
    for each edge of a square $S_i$ {
        //Determine whether there is an intersection given $s^*$
        if $(s_0 < s^* < s_1 || s_1 < s^* < s_0)$
        {
            // compute the intersection
            intersection_counter ++;
        }
    }
    //According to the value of intersection_counter, connect the intersections
}

What is the issue with the above code and how to optimize?
Artifacts?

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

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

\[ S^* = 80 \]
Artifacts?

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

We have no basis to assume *anything*. 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 the data that we’ve been given.

*There is no substitute for having an adequate number of Points data*
And, of course, if you can do it in one plane, you can do it in multiple planes.

And, speaking of contours in multiple planes, this brings us to the topic of wireframe iso-surfaces . . .