Assignment #3:

Scalar Field Visualization 3D: Cutting Plane, Wireframe Iso-surfacing, and Direct Volume Rendering

Due October 9th, before midnight

Goals:

With the results from your assignment#2, the first goal is to get familiar with the slicing technique in 3D volumetric data visualization. You will need to design an interface to allow the volume slicing along the axis directions. The second goal is to learn some simple iso-surfacing like visualization technique which is a natural extension of the iso-contouring. Finally, you will be required to implement a direct volume rendering technique, which will allow you to see more inner structure in the volumetric data. All the visualization in this assignment will re-use the coloring schemes that you have implemented in assignment#2.

Tasks:

In this project, we are going to create an artificial volumetric scalar data, similar to what you have used for assignment#2. It will be great if you can load some real volumetric data for this project but this is not required.

You are going to run a temperature simulation of a 3D room that has 4 heat sources. The function you are going to use is

\[ t(x, y, z) = \sum_{i=0}^{3} A_i e^{-5r_i^2} \]

where \( r_i^2 = (x - X_i)^2 + (y - Y_i)^2 + (z - Z_i)^2 \) and

<table>
<thead>
<tr>
<th>( i )</th>
<th>( X_i )</th>
<th>( Y_i )</th>
<th>( Z_i )</th>
<th>( A_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>90.00</td>
</tr>
<tr>
<td>1</td>
<td>-1.00</td>
<td>0.30</td>
<td>0.00</td>
<td>120.00</td>
</tr>
<tr>
<td>2</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
<td>120.00</td>
</tr>
<tr>
<td>3</td>
<td>0.00</td>
<td>0.40</td>
<td>1.00</td>
<td>170.00</td>
</tr>
</tbody>
</table>

You are given the temperature data from the simulation at each node in a 3D grid. The coordinate range of the data volume is: -1.0 ≤ x,y,z ≤ 1.0. The number of node points you place along each dimension is at least 50 but should also be adjustable by the user.

#define NX ??? // should be larger and equal 50
#define NY ???
For the temperature range of the data, use: \(0.0 \leq t \leq 100.0\). The temperatures defined by the equation actually go higher than 100 degrees in places, but don't worry about it. After computing \(t\), just clamp it to 100:

```c
#define NZ ???
const float TEMPMIN = { 0.f };
const float TEMPMAX = { 100.f };
... if( t > TEMPMAX )
    t = TEMPMAX;
```

You can re-use your data structure that you used for the task2 of assignment #1. The data value computation at each grid point is the same as assignment #1 (please also see the “Suggestions” at the end of this assignment description).

**Task 1: Volume slicing (50 points)**

Use four GLUI range sliders to allow the user to cull the data by displaying a subset in X, Y, Z, and Temperature. You can simply add an additional slider for Z direction based on your assignment#1.

You will need to prepare three orthogonal cutting planes that are parallel to the XY, YZ, and XZ planes, respectively. Use three checkboxes to enable the user to toggle on or off certain cutting planes.

You need to compute the 2D scalar values on each plane and make a color plot for each plane using the color scales you have implemented in assignment#1.

Use a fifth GLUI range slider to allow the user to reduce the data by checking its distance from the center of the cube (0.,0.,0.).

Use a sixth range slider to control the display based on the absolute gradient at each point. The gradient at each point is a 3-component vector: \(\frac{dT}{dx}, \frac{dT}{dy}, \frac{dT}{dz}\). The absolute gradient is \(\sqrt{\left(\frac{dT}{dx}\right)^2 + \left(\frac{dT}{dy}\right)^2 + \left(\frac{dT}{dz}\right)^2}\). This will show where the temperature is changing quickly and where it is changing slowly.

The x gradient at a point is obtained by taking the difference from the point before to the point after. This is called a two-sided gradient (or central difference) computation:

```c
Nodes[i][j][k].dTdx = ( Nodes[i+1][j][k].T - Nodes[i-1][j][k].T ) / ( Nodes[i+1][j][k].x - Nodes[i-1][j][k].x );
```
The $y$ and $z$ gradients are similar. Be sure to take into account when $i$, $j$, or $k$ are either 0 or at their maximum value. This is when you would use a one-sided gradient computation.

2. Wireframe Iso-surfacing (50 points)

Implement the wireframe iso-surfacing technique as introduced in the class (slides 31-32 in lecture 6). This should be done based on the iso-contours computed on those volume slices that are parallel to the XY, YZ, and XZ planes, respectively.

3. Texture-based Direct Volume Rendering (50 points)

Implement the 2D texture approach of the texture-based volume rendering. Here is a more detailed instruction on how you should do.

First, create a volume of nodes

```c
struct node Nodes[NX][NY][NZ];  //Pick NX, NY, and NZ such that each is a power of 2, e.g. 256.
```

For each node structure, compute its scalar value $S$ and its associated $R$, $G$, $B$. Use the same scalar value function and color-mapping scheme that you used in assignment#2.

Create a normal slider that will determine the opacity that the visible $S$ values will have.

![Slider Image]

Maximun opacity: 1.00

Write 3 routines, `CompositeXY()`, `CompositeXZ()`, and `CompositeYZ()`. These will composite the voxels and fill one of 3 Texture arrays:

```c
unsigned char TextureXY[NZ][NX][NY][4];
unsigned char TextureXZ[NY][NX][NZ][4];
unsigned char TextureYZ[NX][NY][NZ][4];
```

These hold 4 components with $R$, $G$, $B$ channels and an alpha channel to control the opacity.

The following routine can be used to composite the texture at XY planes. The texture on XZ and YZ planes can be composited similarly.
void CompositeXY( void )
{
    int x, y, z, zz;
    float alpha;   /* opacity at this voxel */
    float r, g, b;   /* running color composite */

    for( x = 0; x < NX; x++ )
    {
        for( y = 0; y < NY; y++ )
        {
            r = g = b = 0.;
            for( zz = 0; zz < NZ; zz++ )
            {
                /* which direction to composite: */
                if( Zside == PLUS )
                    z = zz;
                else
                    z = ( NZ-1 ) - zz;

                if( ????? ) // determine the whether the value is out of the
                // range set by the range slider
                {
                    r = g = b = 0.;
                    alpha = 0.;
                }
                else
                {
                    r = Nodes[x][y][z].r;
                    g = Nodes[x][y][z].g;
                    b = Nodes[x][y][z].b;
                    alpha = MaxAlpha;
                }

                TextureXY[zz][y][x][0] = (unsigned char) ( 255.*r  + .5 );
                TextureXY[zz][y][x][1] = (unsigned char) ( 255.*g  + .5 );
                TextureXY[zz][y][x][2] = (unsigned char) ( 255.*b  + .5 );
                TextureXY[zz][y][x][3] = (unsigned char) ( 255.*alpha + .5 );
            }
        }
    }
}

Depending on the rotation of the volume, you will want to draw the parallel planes in the X, Y, or Z
direction, and draw them minus-to-PLUS or plus-to-MINUS. The following code tells you which of these
cases is the one you will need.

/* which way is a surface facing: */
const int MINUS = { 0 };
const int PLUS  = { 1 };

```c
/* what is the major direction: */

#define X 0
#define Y 1
#define Z 2

... 

int Major;   /* X, Y, or Z */
int Xside, Yside, Zside; /* which side is visible, PLUS or MINUS */

... 

/**
 ** determine which sides of the cube are facing the user
 ** this assumes that the rotation is being done by:
 **
 ** glRotatef( Yrot, 0., 1., 0. );
 ** glRotatef( Xrot, 1., 0., 0. );
 **
**/

void DetermineVisibility()
{
    float xr, yr;
    float cx, sx;
    float cy, sy;
    float nzx, nzy, nzz; /* z component of normal for x side, y side, and z side */

    xr = Xrot * ( M_PI/180. );
    yr = Yrot * ( M_PI/180. );

    cx = cos( xr );
    sx = sin( xr );
    cy = cos( yr );
    sy = sin( yr );

    nzx = -sy;
    nzy = sx * cy;
    nzz = cx * cy;

    /* which sides of the cube are showing:
    */
    /* the Xside being shown to the user is MINUS or PLUS */

    Xside = ( nzx > 0. ? PLUS : MINUS );
    Yside = ( nzy > 0. ? PLUS : MINUS );
```
Zside = ( nzz > 0. ? PLUS : MINUS );

/* which direction needs to be composited: */
if( fabs(nzx) > fabs(nzy) && fabs(nzx) > fabs(nzz) )
    Major = X;
else if( fabs(nzy) > fabs(nzx) && fabs(nzy) > fabs(nzz) )
    Major = Y;
else
    Major = Z;

To draw the obtained texture arrays using the OpenGL texture mapping functionality. Here is what you will do.

glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP );
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );

filter = GL_NEAREST;
if( Bilinear )
    filter = GL_LINEAR;

glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, filter );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, filter );
glPixelStorei( GL_UNPACK_ALIGNMENT, 1 );
glEnable( GL_TEXTURE_2D );

glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );
glEnable( GL_BLEND );

if( Major == Z )
{
    if( Zside == PLUS )
    {
        z0 = -10.;
        dz = 20. / (float)( NZ - 1 );
    }
    else
    {
        z0 = 10.;
        dz = -20. / (float)( NZ - 1 );
    }

    for( z = 0, zcoord = z0; z < NZ; z++, zcoord += dz )
    {
        glTexImage2D( GL_TEXTURE_2D, 0, 4, NX, NY, 0, GL_RGBA,
                      GL_UNSIGNED_BYTE, &TextureXY[z][0][0][0] );
        glBegin( GL_QUADS );
        glTexCoord2f( 0., 0. );
        }
4. Polygonal iso-surfacing (50 points)

This part is optional. You will get extra credit if you finish it. You can implement either a full Marching cubes algorithm or the simplified iso-surfacing technique as described at the end of the lecture note.

**Grades:**

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Total points</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>(extra)</td>
<td>4</td>
</tr>
</tbody>
</table>

**Suggestions**

Here are some suggestions that you may find helpful for this assignment.

The following code can be used to compute the scalar fields defined on the 3D grids. However, you can always develop your own code.

```c
struct sources
{
  double xc, yc, zc;    // temperature value of the source
}
Sources[] =
{
  {  1.00f,  0.00f,  0.00f,  90.00f },
  { -1.00f, -0.30f,  0.00f, 120.00f },
  {  0.10f,  1.00f,  0.00f, 120.00f },
  {  0.00f,  0.40f,  1.00f, 170.00f },
};

// The following function is going to be used for the next assignment as well
```
double Temperature( double x, double y, double z )
{
    double t = 0.0;
    for( int i = 0; i < 4; i++ )
    {
        double dx = x - Sources[i].xc;
        double dy = y - Sources[i].yc;
        double dz = z - Sources[i].zc;
        double rsqd = dx*dx + dy*dy + dz*dz;
        t += Centers[i].a * exp( -5.*rsqd );
    }
    if( t > TEMPMAX )
        t = TEMPMAX;
    return t;
}

You can use the following data structure to store your 2D and 3D scalar fields defined at the regular grids.

```c
struct node
{
    float x, y, z;          // location
    float T;                // temperature
    float r, g, b;  // the assigned color
    float rad;              // radius
    float dTdx, dTdy, dTdz; // can store these if you want, or not
    float grad;             // total gradient
};
```

The instruction for adding range sliders can be founded in the GLUI online document or the suggestions at the end of the instruction of assignment #1. Have fun!