Assignment #4

3D Vector Field Visualization

Due Nov.6 before midnight

Goal:

In this assignment, you are required to implement a number of visualization techniques to help explore a number of 3D steady vector fields, including arrow plot, streamlines, stream ribbons, and stream surfaces. You will be given a set of analytic vector fields generated using formula.

Tasks:

Similar to assignment #2, you need to create a regular grid with the range

$$-1 \le x, y, z \le 1$$

Compute a vector at each node point according to the following formulas for the X, Y, and Z components of the vector based on the following formula

$$field\ 1\text{:}\ \begin{cases} v_x(x,y,z) = -3 + 6x - 4x(y+1) - 4z \\ v_y(x,y,z) = 12x - 4x^2 - 12z + 4z^2 \\ v_z(x,y,z) = 3 + 4x - 4x(y+1) - 6z + 4(y+1)z \end{cases}$$

$$field\ 2\text{:}\ \begin{cases} v_x(x,y,z) = Asinz + Ccosy \\ v_y(x,y,z) = Bsinx + Acosz \\ v_z(x,y,z) = Csiny + Bcosx \end{cases} \qquad where\ A = \sqrt{3}, B = \sqrt{2}, C = 1$$

$$field\ 3\text{:}\ \begin{cases} v_x(x,y,z) = -y \\ v_y(x,y,z) = -z \\ v_z(x,y,z) = x \end{cases}$$

The following routine can be used to compute the vector value at any given 3D position for field 1. Note that you will be using it for your streamline computation as well.

```
void
get_vector_field1( float x, float y, float z, float &vxp, float &vyp, float
&vzp )
{
        vxp = -3 + 6.*x - 4.*x*(y+1.) - 4.*z;
        vyp = 12.*x - 4.*x*x - 12.*z + 4.*z*z;
        vzp = 3. + 4.*x - 4.*x*(y+1.) - 6.*z + 4.*(y+1.)*z;
}
```

Similarly, you can set up a routine to compute the vector value for other fields.

1. Direct visualization (20 points)

Assign a 3D arrow for each grid point. The shape of the arrow is up to you. But you can always use the one provided in the skeleton code. Note that you need to scale the arrows uniformly in order to get reasonable visualization. The color of each arrow is determined by the magnitude of the corresponding vector value. You can use any color scales that you have implemented in assignment #1.

2. Streamlines (100 points)

Draw streamlines within the field starting with *at least 10* distinct points. My suggestion is to try to start with a streamline for each grid point and see whether you can get a reasonable visualization. Then, try to place the seeds for those streamlines smartly. How to determine the places of seeding is up to you. Use a *second-order* scheme to draw the streamlines.

Allow the user to move a 3D streamline probe through the volume. Every time the probe moves, its streamline needs to be re-created. How you position the start of the probe in X-Y-Z is up to you. I will suggest starting from the origin (0, 0, 0). To move the starting position, modify the translation widget in the interface. (The sample program uses 3 GLUI Translate-XY widgets to allow translation in XY, XZ, and YZ).

Allow the user to move a 3D Ribbon Trace probe through the volume. Instead of having a single stream coming from the probe, attach a small horizontal line to the probe and have some number of streams coming from different places on the line. Connect them up with GL_QUADS. It is not required to do OpenGL lighting (but it is a nice touch).

3. 3D probe (optional, 30 points)

Instead of having a single stream coming from the probe, have a 3D shape start there and deform. Particle trace each vertex in the object and then reconnect them as they were originally connected. (Hint: you will need some help from the Animate() routine.)

Grades:

Tasks	*Total points*
1	20
2	100
(extra) 3	30

Suggestions:

Draw each streamline until: 1)It leaves the bounding cube given above, or, 2)The change in (x,y,z) becomes very nearly zero, or, 3)You have drawn for a certain number of steps (try a couple hundred)

If you do not have a routine for 3D arrow drawing, you can use the following piece of code to generate 3D arrows by Prof. Mike Bailey.

```
//////////////////Global variables for axes, by Mike Bailey ////////////////////
//////////////
/* size of wings as fraction of length:                                  */
#define WINGS  0.10

/* axes:                                                                 */
#define X          1
#define Y          2
#define Z          3

/* x, y, z, axes:                                  */
static float axx[3] = { 1., 0., 0. };
static float ayy[3] = { 0., 1., 0. };
static float azz[3] = { 0., 0., 1. };

/////Function for axes drawing, borrow from arrow.c//////////////
void       Arrow( float [3], float [3] );
void       cross( float [3], float [3], float [3] );
float      dot( float [3], float [3] );
float      unit( float [3], float [3] );

///The following are the implementations of axes functions////////
void Arrow( float tail[3], float head[3] )
{
        float u[3], v[3], w[3];             /* arrow coordinate system        */
        float d;                    /* wing distance              */
        float x, y, z;                      /* point to plot              */
        float mag;                          /* magnitude of major direction   */
        float f;                    /* fabs of magnitude          */
        int axis;                   /* which axis is the major */

        /* set w direction in u-v-w coordinate system:                       */
        w[0] = head[0] - tail[0];
        w[1] = head[1] - tail[1];
        w[2] = head[2] - tail[2];

        /* determine major direction:                                        */
        axis = X;
        mag = fabs( w[0] );
        if( (f=fabs(w[1]))  > mag )
        {
                axis = Y;
                mag = f;
        }
        if( (f=fabs(w[2]))  > mag )
        {
                axis = Z;
```

```
		mag = f;
}

/* set size of wings and turn w into a unit vector:		*/
d = WINGS * unit( w, w );

/* draw the shaft of the arrow:					*/
glBegin( GL_LINE_STRIP );
		glVertex3fv( tail );
		glVertex3fv( head );
glEnd();

/* draw two sets of wings in the non-major directions:		*/
if( axis != X )
{
		cross( w, axx, v );
		(void) unit( v, v );
		cross( v, w, u  );
		x = head[0] + d * ( u[0] - w[0] );
		y = head[1] + d * ( u[1] - w[1] );
		z = head[2] + d * ( u[2] - w[2] );
		glBegin( GL_LINE_STRIP );
			glVertex3fv( head );
			glVertex3f( x, y, z );
		glEnd();
		x = head[0] + d * ( -u[0] - w[0] );
		y = head[1] + d * ( -u[1] - w[1] );
		z = head[2] + d * ( -u[2] - w[2] );
		glBegin( GL_LINE_STRIP );
			glVertex3fv( head );
			glVertex3f( x, y, z );
		glEnd();
}

if( axis != Y )
{
		cross( w, ayy, v );
		(void) unit( v, v );
		cross( v, w, u  );
		x = head[0] + d * ( u[0] - w[0] );
		y = head[1] + d * ( u[1] - w[1] );
		z = head[2] + d * ( u[2] - w[2] );
		glBegin( GL_LINE_STRIP );
			glVertex3fv( head );
			glVertex3f( x, y, z );
		glEnd();
		x = head[0] + d * ( -u[0] - w[0] );
		y = head[1] + d * ( -u[1] - w[1] );
		z = head[2] + d * ( -u[2] - w[2] );
		glBegin( GL_LINE_STRIP );
			glVertex3fv( head );
			glVertex3f( x, y, z );
		glEnd();
}

if( axis != Z )
{
		cross( w, azz, v );
		(void) unit( v, v );
```

```
                cross( v, w, u  );
                x = head[0] + d * ( u[0] - w[0] );
                y = head[1] + d * ( u[1] - w[1] );
                z = head[2] + d * ( u[2] - w[2] );
                glBegin( GL_LINE_STRIP );
                        glVertex3fv( head );
                        glVertex3f( x, y, z );
                glEnd();
                x = head[0] + d * ( -u[0] - w[0] );
                y = head[1] + d * ( -u[1] - w[1] );
                z = head[2] + d * ( -u[2] - w[2] );
                glBegin( GL_LINE_STRIP );
                        glVertex3fv( head );
                        glVertex3f( x, y, z );
                glEnd();
        }
        /* done:                                                        */
}


///calculate the dot production of two vectors
float dot( float v1[3], float v2[3] )
{
        return( v1[0]*v2[0] + v1[1]*v2[1] + v1[2]*v2[2] );
}


///calculate the cross production of two vectors
void cross( float v1[3], float v2[3], float vout[3] )
{
        float tmp[3];

        tmp[0] = v1[1]*v2[2] - v2[1]*v1[2];
        tmp[1] = v2[0]*v1[2] - v1[0]*v2[2];
        tmp[2] = v1[0]*v2[1] - v2[0]*v1[1];

        vout[0] = tmp[0];
        vout[1] = tmp[1];
        vout[2] = tmp[2];
}


///Normalize vector
float unit( float vin[3], float vout[3] )
{
        float dist, f ;

        dist = vin[0]*vin[0] + vin[1]*vin[1] + vin[2]*vin[2];

        if( dist > 0.0 )
        {
                dist = sqrt( dist );
                f = 1. / dist;
                vout[0] = f * vin[0];
                vout[1] = f * vin[1];
                vout[2] = f * vin[2];
        }
        else
        {
```

```
                vout[0] = vin[0];
                vout[1] = vin[1];
                vout[2] = vin[2];
        }
        return( dist );
}
```

For the probe cube drawing, you can use the following routine. You are welcome to design a different probe rather than a uniform cube.

```
/////Drawing the probe cube/////
void  drawProbeCube(float *p0, float *p1, float *p2, float *p3,
                    float *p4, float *p5, float *p6, float *p7)
{
        glColor3f(0, 1., 0);
        //Front
        glBegin(GL_QUADS);
                glVertex3fv(p0);
                glVertex3fv(p1);
                glVertex3fv(p2);
                glVertex3fv(p3);
        glEnd();

        //Rear
        glBegin(GL_QUADS);
                glVertex3fv(p4);
                glVertex3fv(p5);
                glVertex3fv(p6);
                glVertex3fv(p7);
        glEnd();

        //Left
        glBegin(GL_QUADS);
                glVertex3fv(p0);
                glVertex3fv(p1);
                glVertex3fv(p5);
                glVertex3fv(p4);
        glEnd();

        //Rigth
        glBegin(GL_QUADS);
                glVertex3fv(p3);
                glVertex3fv(p2);
                glVertex3fv(p6);
                glVertex3fv(p7);
        glEnd();

   //Upper
        glBegin(GL_QUADS);
                glVertex3fv(p1);
                glVertex3fv(p5);
                glVertex3fv(p6);
                glVertex3fv(p2);
        glEnd();

        //Buttom
        glBegin(GL_QUADS);
```

```
                glVertex3fv(p0);
                glVertex3fv(p4);
                glVertex3fv(p7);
                glVertex3fv(p3);
        glEnd();
}
```

To add animation for the update of the probe, you can update the Animate() routine as follows.

```
void
Animate( void )
{
        // put animation stuff in here -- change some global variables
        // for Display() to find:

        if(WhichProbe == BLOBE){
           Sleep(300);

                getNewPointinStream(p0);
                getNewPointinStream(p1);
                getNewPointinStream(p2);
                getNewPointinStream(p3);
                getNewPointinStream(p4);
                getNewPointinStream(p5);
                getNewPointinStream(p6);
                getNewPointinStream(p7);


                // force a call to Display() next time it is convenient:

                glutSetWindow( MainWindow );
                glutPostRedisplay();
        }
}
```

Note that the getNewPointinStream(*) is a function that you will implement to advance the front of a streamline (a second order integrator is required).

At the end of the InitGlui() routine, add the following line

```
        GLUI_Master.set_glutIdleFunc( Animate ); // set the graphics window's idle function
```