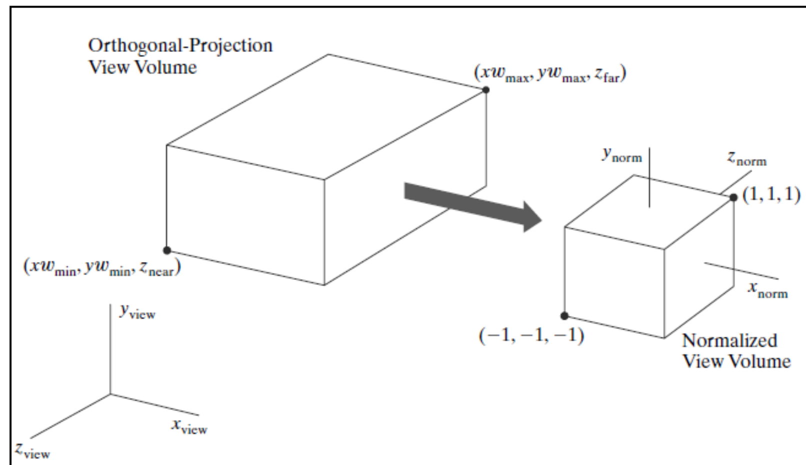
Computer Graphics
IAT 3 Solution 2018

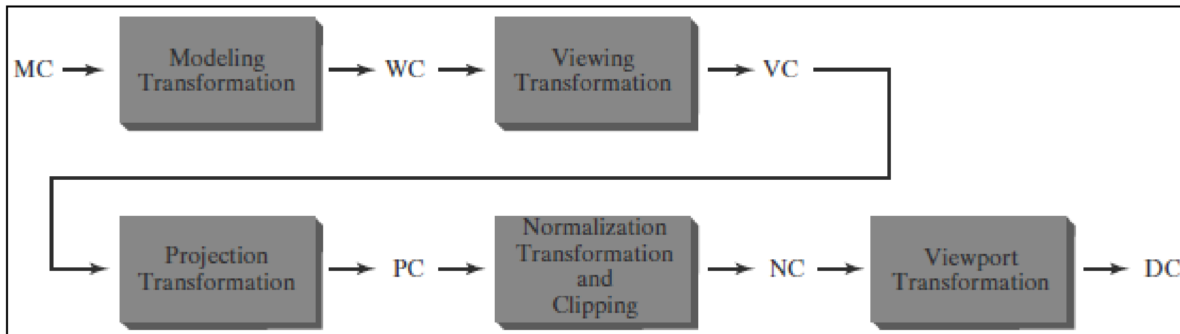
1a) Derive normalization transformation matrix for Orthogonal projection.



- Coordinate descriptions inside this rectangular parallelepiped are the projection coordinates, and they can be mapped into a normalized view volume without any further projection processing.
- As the screen coordinates are often specified in a left-handed reference frame (above Figure), normalized coordinates also are often specified in a left-handed system.
- We can convert projection coordinates into positions within a left-handed normalized-coordinate reference frame, and these coordinate positions will then be transferred to left handed screen coordinates by the viewport transformation.
- To illustrate the normalization transformation, we assume that the orthogonal-projection view volume is to be mapped into the symmetric normalization cube with coordinates in the range from −1 to 1 within a left-handed reference frame. z-coordinate positions for the near and far planes are denoted as Z_{near} and Z_{far}, respectively
- Position (Xmin, Ymin, Znear) is mapped to the normalized position (−1, −1, −1), and position (Xmax, Ymax, Zfar) is mapped to (1, 1, 1).
- Transforming the rectangular-parallelepiped view volume to a normalized cube is similar to the methods for converting the clipping window into the normalized symmetric square.
- The normalization transformation for the orthogonal view volume is given as below.

$$
\mathbf{M}_{\text{ortho,norm}} = \begin{bmatrix} \dfrac{2}{xw_{\max} - xw_{\min}} & 0 & 0 & -\dfrac{xw_{\max} + xw_{\min}}{xw_{\max} - xw_{\min}} \\ 0 & \dfrac{2}{yw_{\max} - yw_{\min}} & 0 & -\dfrac{yw_{\max} + yw_{\min}}{yw_{\max} - yw_{\min}} \\ 0 & 0 & \dfrac{-2}{z_{\text{near}} - z_{\text{far}}} & \dfrac{z_{\text{near}} + z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

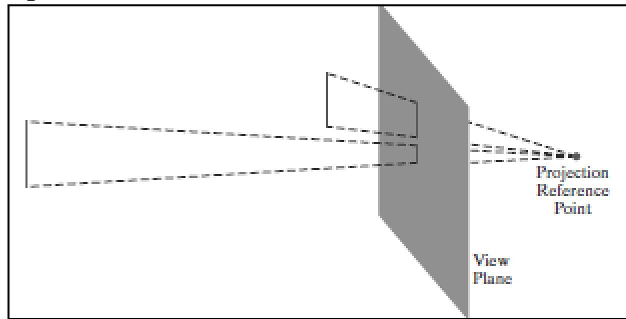
b) Briefly explain 3D viewing pipeline with neat block diagram.



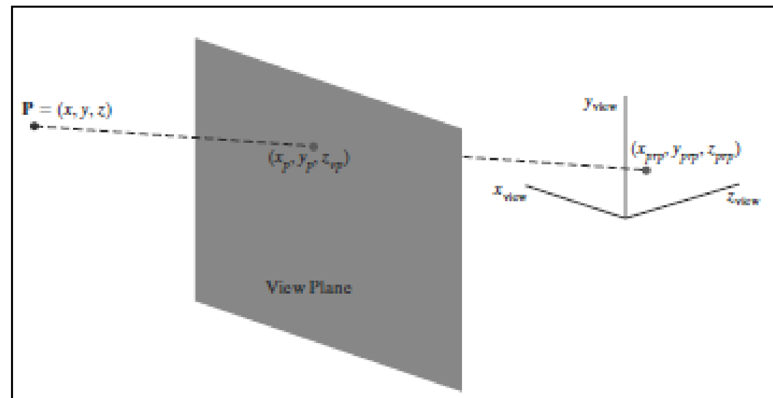
- Above figure shows the general processing steps for creating and transforming a three-dimensional scene to device coordinates.
- Once the scene has been modeled in world coordinates, a viewing-coordinate system is selected and the description of the scene is converted to viewing coordinates.
- The viewing coordinate system defines the viewing parameters, including the position and orientation of the projection plane (view plane), which we can think of as the camera film plane.
- A two-dimensional clipping window, corresponding to a selected camera lens, is defined on the projection plane, and a three-dimensional clipping region is established. This clipping region is called the view volume, and its shape and size depends on the dimensions of the clipping window, the type of projection we choose, and the selected limiting positions along the viewing direction.
- Projection operations are performed to convert the viewing-coordinate description of the scene to coordinate positions on the projection plane.
- Objects are mapped to normalized coordinates, and all parts of the scene outside the view volume are clipped off. The clipping operations can be applied after all device-independent coordinate transformations (from world coordinates to normalized coordinates) are completed.
- The final step is to map viewport coordinates to device coordinates within a selected display window. Scene descriptions in device coordinates are sometimes expressed in a left-handed reference frame so that positive distances from the display screen can be used to measure depth values in the scene.

2) Explain perspective projection with near diagrams? Derive perspective projection transformation coordinates.

- Perspective projection is a projection which approximates the geometric-optics effect by projecting objects to the view plane along converging paths to a position called the projection reference point (or center of projection).
- We can sometimes select the projection reference point as another viewing parameter in a graphics package, but some systems place this convergence point at a fixed position, such as at the view point.



- Consider a perspective projection where projection a spatial position (x, y, z) is projected at a projection reference point at (Xprp, Yprp, Zprp) as shown in the below figure.
- The projection line intersects the view plane at the coordinate position (Xp, Yyp, Zvp), where Zvp is some selected position for the view plane on the Zview axis.



We can write equations describing coordinate positions along this perspective-projection line in parametric form as below,

$$\begin{aligned} x' &= x - (x - x_{prp})u \\ y' &= y - (y - y_{prp})u \qquad 0 \le u \le 1 \\ z' &= z - (z - z_{prp})u \end{aligned}$$

Coordinate position (x', y', z') represents any point along the projection line. When u = 0, we are at position P = (x, y, z). At the other end of the line, u = 1 and we have the projection reference-point coordinates (Xprp, Yprp, Zprp).

On the view plane, z' = Zvp and we can solve the z' equation for parameter u at this position along the projection line as:

$$u = \frac{z_{vp} - z}{z_{prp} - z}$$

Substituting this value of u into the equations for x' and y' and solving them, we obtain the general perspective-transformation equations as,

$$x_p = x\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) + x_{prp}\left(\frac{z_{vp} - z}{z_{prp} - z}\right)$$

$$y_p = y\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) + y_{prp}\left(\frac{z_{vp} - z}{z_{prp} - z}\right)$$

3) Write a program to animate a flag using Bezier Curve algorithm.

```
#include<GL/glut.h>
#include<stdio.h>
#include<math.h>
int i,k,u;
int flag;
float cp[4][2]={{20,70},{30,80},{40,60},{50,75}};
float cp1[4][2]={{20,60},{30,70},{40,50},{50,65}};
float cp2[4][2]={{20,50},{30,60},{40,40},{50,55}};

void beziercoeff(int n, float *c)
{
	for(k=0;k<=n;k++)
	{
		c[k]=1;
		for(i=n;i>=k+1;i--)
		{
			c[k]=c[k]*i;
		}
		for(i=n-k;i>=1;i--)
		{
			c[k]=c[k]/i;
		}
	}
}

void init()
{

	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluOrtho2D(0,100,0,100);
	glMatrixMode(GL_MODELVIEW);
}
```

```
void draw(float cpp[][2])
{
	float blend,u,x,y;
	int n=3;
	float c[4];
	beziercoeff(n,c);
	glLineWidth(1000);
	glBegin(GL_LINE_STRIP);
		for(u=0;u<=1;u+=0.01)
		{
			x=0,y=0;
			for(k=0;k<=n;k++)
			{
				blend=c[k]*pow(u,k)*pow(1-u,n-k);
				x+=cpp[k][0]*blend;
				y+=cpp[k][1]*blend;
			}
			glVertex2f(x,y);
		}
	glEnd();
}


void flagu()
{
      glColor3f(1,0.25,0.25);

	draw(cp);

	glColor3f(0,0,1);
	draw(cp1);

	glColor3f(0,1,0);
	draw(cp2);
	glColor3f(0,0,0);
	glBegin(GL_LINES);
		glVertex2f(20,50);
		glVertex2f(20,70);
		glVertex2f(50,55);
		glVertex2f(50,75);
	glEnd();
	glLineWidth(300);
	glColor3f(1,1,0);
	glBegin(GL_LINES);
		glVertex2f(20,72);
		glVertex2f(20,10);
	glEnd();
	glBegin(GL_POLYGON);
		glVertex2f(20.5,72);
		glVertex2f(19.5,74);
```

```
			glVertex2f(20.5,76);
			glVertex2f(21.5,74);
		glEnd();
}
void scan_menu(int id)
{
	if(id == 1)
		flag = 1;
	else if(id == 2)
		flag = 2;
	else
		exit(0);
		glutPostRedisplay();

}
void display()
{
	glClearColor(1,1,1,1);
		glClear(GL_COLOR_BUFFER_BIT);
		if(flag==1)
		{
			glColor3f(1,0,0);
			float cp23[4][2]={{12,10},{22,48},{32,5},{42,50}};
			draw(cp23);
		}
		else if(flag==2)
		{
			int iterations,max;
			float yCHANGE=0.01;//0.01;
			//while(temp--)
			//for(temp=1000;temp>0;temp--)
			{
				for(iterations=0; iterations<2000; iterations++)
				{
					cp[1][1]-=yCHANGE;
					cp[2][1]+=yCHANGE;
					cp1[1][1]-=yCHANGE;
					cp1[2][1]+=yCHANGE;
					cp2[1][1]-=yCHANGE;
					cp2[2][1]+=yCHANGE;
					flagu();
					glClear(GL_COLOR_BUFFER_BIT);
					glutPostRedisplay();
					for(int temp=1000;temp>0;temp--);
				}
				for(iterations=0; iterations<2000; iterations++)
				{
					cp[1][1]+=yCHANGE;
					cp[2][1]-=yCHANGE;
					cp1[1][1]+=yCHANGE;
```

```
                        cp1[2][1]-=yCHANGE;
                        cp2[1][1]+=yCHANGE;
                        cp2[2][1]-=yCHANGE;
                        flagu();
                        glClear(GL_COLOR_BUFFER_BIT);
                        glutPostRedisplay();
                        for(int temp=1000;temp>0;temp--);
                    }
                }
        }

        glFlush();

}
void main(int argc,char** argv)
{
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);
        glutInitWindowPosition(100,100);
        glutInitWindowSize(500,500);
        glutCreateWindow("Bezier's Curve");
        init();
        glutDisplayFunc(display);
        glutCreateMenu(scan_menu);
        glutAddMenuEntry("Draw Single Curve",1);
        glutAddMenuEntry("Draw Flag",2);
        glutAddMenuEntry("exit",0);
        glutAttachMenu(GLUT_RIGHT_BUTTON);
        glutMainLoop();
}
```

4) Explain with a simple program how an event driven input can be programmed for a keyboard and mouse device.

A mouse event occurs when one of the mouse buttons is either depressed or released. When a button is depressed, the action generates a mouse down event. When it is released, a mouse up event is generated. The information returned includes the button that generated the event, the state of the button after the event (up or down), and the position of the cursor tracking the mouse in window coordinates (with the origin in the upper-left corner of the window). We register the mouse callback function, usually in the main function, by means of the GLUT function.

glutMouseFunc(myMouse);

The mouse callback must have the form

void myMouse(int button, int state, int x, int y);

It is provided by the application programmer. Within the callback function, we define the actions that we want to take place if the specified event occurs

For our simple example, we want the depression of the left mouse button to terminate the program. The required callback is the single-line function

```
void myMouse(int button, int state, int x, int y)
{
        if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        exit(0);
}
```

Keyboard events can be generated when the mouse is in the window and one of the keys is depressed or released. The GLUT function glutKeyboardFunc is the callback for events generated by depressing a key, whereas glutKeyboardUpFunc is the callback for events generated by release of a key.

When a keyboard event occurs, the ASCII code for the key that generated the event and the location of the mouse are returned. All the key-press callbacks are registered in a single callback function, such as

glutKeyboardFunc(myKey);

For example, if we wish to use the keyboard only to exit the program, we can use the callback function

```
void myKey(unsigned char key, int x, int y)
{
        if(key=='q' || key == 'Q') exit(0);
}
```

Example:

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include<math.h>
#include<stdio.h>


GLfloat theta,thetar;
void display()
{
   glClear(GL_COLOR_BUFFER_BIT);
   glBegin(GL_POLYGON);
   thetar=theta/(3.14159/180.0);        //convert theta in degrees to radians
   glVertex2f(cos(thetar),sin(thetar));
   glVertex2f(-sin(thetar),cos(thetar));
   glVertex2f(-cos(thetar),-sin(thetar));
   glVertex2f(sin(thetar),-cos(thetar));
   glEnd();
   glFlush();
   glutSwapBuffers();
}
void idle()
{
   theta+=2;
   if(theta>=360.0) theta-=360.0;
```

```
   glutPostRedisplay();
}

void mouse(int button,int state,int x,int y)      //  change idle function based on
        //  mouse button pressed
{
   if(button==GLUT_LEFT_BUTTON&&state==GLUT_DOWN)
   glutIdleFunc(idle);
   if(button==GLUT_RIGHT_BUTTON&&state==GLUT_DOWN)
      glutIdleFunc(NULL);
}
void keyboard(unsigned char key,int x,int y)      //  change idle function based on
        //  mouse button pressed
{
   if(key=='p')
   glutIdleFunc(idle);
   if(key=='r')
      glutIdleFunc(NULL);
}

int main(int argc,char **argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
   glutCreateWindow("Rotating Square");
   glutDisplayFunc(display);
   glutMouseFunc(mouse);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

5) List the three input modes and explain them with the neat block diagrams.

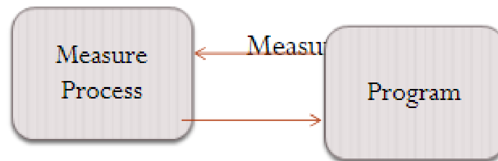
The manner by which physical and logical input devices provide input to an application program can be described in terms of two entities: a measure process and a device trigger. The measure of a device is what the device returns to the user program. The trigger of a device is a physical input on the device with which the user can signal the computer.

We can obtain the measure of a device in three distinct modes. Each mode is defined by the relationship between the measure process and the trigger. Once the measure process is started, the measure is taken and placed in a buffer, even though the contents of the buffer may not yet be available to the program. For example, the position of a mouse is tracked continuously by the underlying window system, regardless of whether the application program needs mouse input.

Request mode: In request mode, the measure of the device is not returned to the program until the device is triggered. This input mode is standard in non graphical applications. For example, if a typical C program requires character input, we use a function such as scanf. When the program needs the input, it halts when it encounters the scanf statement and waits while we

type characters at our terminal. In request mode, the measure of the device is not returned to the program until the device is triggered. This input mode is standard in non graphical applications. For example, if a typical C program requires character input, we use a function such as scanf. When the program needs the input, it halts when it encounters the scanf statement and waits while we type characters at our terminal. Request mode can be represented as below using block diagram.

Sample-mode: In this mode the input is immediate. As soon as the function call in the application program is encountered, the measure is returned. In sample mode, the user must have positioned the pointing device or entered data using the keyboard before the function call, because the measure is extracted immediately from the buffer. Sample mode can be represented as below using block diagram.



One characteristic of both request- and sample-mode input in APIs that support them is that the user must identify which device is to provide the input. Consequently, we ignore any other information that becomes available from any input device other than the one specified. Both request and sample modes are useful for situations where the program guides the user, but they are not useful in applications where the user controls the flow of the program.

Event mode: This mode can handle an environment with multiple input devices, each with its own trigger and each running a measure process. Each time that a device is triggered, an event is generated. The device measure, including the identifier for the device, is placed in an event queue. This process of placing events in the event queue is completely independent of what the application program does with these events. The user program can examine the front event in the queue or, if the queue is empty, can wait for an event to occur. If there is an event in the queue, the program can look at the event's type and then decide what to do. Event mode can be represented as below using block diagram.

6) What is a display list? Give the open GL code segment that generates a display list defining a red triangle with vertices at (50, 50), (150, 50) and (100, 150).

Display list is a structure which can be used multiple times with different display operations. On a network, a display list describing a scene is stored on the server machine, which eliminates the need to transmit the commands in the list each time the scene is to be displayed. We can also set up a display list so that it is saved for later execution, or we can specify that the commands in the list be executed immediately. And display lists are particularly useful for hierarchical modeling, where a complex object can be described with a set of simpler subparts.

A set of OpenGL commands is formed into a display list by enclosing the commands within the glNewList/glEndList pair of functions. For example,

```
glNewList (listID, listMode};
...
glEndList ( );
```

This structure forms a display list with a positive integer value assigned to parameter listID as the name for the list. Parameter listMode is assigned an OpenGL symbolic constant that can be either GL COMPILE or GL COMPILE AND EXECUTE. If we want to save the list for later execution, we use GL COMPILE. Otherwise, the commands are executed as they are placed into the list, in addition to allowing us to execute the list again at a later time.

```
#define triangle 1
glNewList (triangle, GL_COMPILE);
glBegin (GL_TRIANGLES);
        glVertex2f(50,50);
        glVertex2f(150,50);
        glVertex2f(100,150);
glEnd ( );
glEndList ( );
glCallList (triangle);
```

7) Explain selection mode and picking with a code snippet.

To use OpenGL's selection mechanism, you first draw your scene into the frame buffer, and then you enter selection mode and redraw the scene. However, once you're in selection mode, the contents of the frame buffer don't change until you exit selection mode. When you exit selection mode, OpenGL returns a list of the primitives that intersect the viewing volume. Each primitive that intersects the viewing volume causes a selection hit. The list of primitives is actually returned as an array of integer-valued names and related data - the hit records - that correspond to the current contents of the name stack. You construct the name stack by loading names onto it as you issue primitive drawing commands while in selection mode. Thus, when the list of names is returned, you can use it to determine which primitives might have been selected on the screen by the user.

To use the selection mechanism, you need to perform the following steps.

- Specify the array to be used for the returned hit records with glSelectBuffer().
- Enter selection mode by specifying GL_SELECT with glRenderMode().
- Initialize the name stack using glInitNames() and glPushName().
- Define the viewing volume you want to use for selection. Usually this is different from the viewing volume you originally used to draw the scene, so you probably want to save and then restore the current transformation state with glPushMatrix() and glPopMatrix().
- Alternately issue primitive drawing commands and commands to manipulate the name stack so that each primitive of interest has an appropriate name assigned.
- Exit selection mode and process the returned selection data (the hit records).

As an extension of the process described above, we can use selection mode to determine if objects are picked. To do this, you use a special picking matrix in conjunction with the

projection matrix to restrict drawing to a small region of the viewport, typically near the cursor. Then you allow some form of input, such as clicking a mouse button, to initiate selection mode. With selection mode established and with the special picking matrix used, objects that are drawn near the cursor causes selection hits. Thus, during picking you're typically determining which objects are drawn near the cursor.

Picking is set up almost exactly like regular selection mode is, with the following major differences.

- Picking is usually triggered by an input device. In the following code examples, pressing the left mouse button invokes a function that performs picking.
- You use the utility routine gluPickMatrix() to multiply a special picking matrix onto the current projection matrix. This routine should be called prior to multiplying a standard projection matrix (such as gluPerspective() or glOrtho()).

```
#include<stdlib.h>
#include<stdio.h>
#include<GL/gl.h>
#include<GL/glu.h>
#include<GL/glut.h>
void draw()
{
   glColor3f(1,0,0);
   glRectf(100,100,200,200);
   glColor3f(1,1,0);
   glRectf(300,300,400,400);
}
void process(int hit,int buffer[])
{
        int i,j,*ptr,names;
        printf("hits  %d\n",hit);
        ptr=buffer;

        for(i=0;i<hit;i++)
        {
        names=*ptr;
                ptr+=3;
        for(j=0;j<names;j++)
        {
                        if(*ptr==11)printf("Yellow square\n");
                        else if(*ptr==10)printf("Red square\n");
                        ptr++;
                }
        }
}

void select(int x,int y)
{
        int hit,buffer[100],viewport[4];
        glGetIntegerv (GL_VIEWPORT, viewport);
        glSelectBuffer(100,(GLuint*)buffer);
```

```
	glInitNames();
	glPushName(0);
	glRenderMode(GL_SELECT);
	glPushMatrix();
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluPickMatrix(x,viewport[3]-y,50,50,viewport);
	gluOrtho2D(0,500,0,500);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
	glPushName(10);
	glRectf(100,100,200,200);
	glLoadName(11);
	glRectf(300,300,400,400);
	glPopMatrix();
	hit=glRenderMode(GL_RENDER);
	process(hit,buffer);
	glutPostRedisplay();
}

void mouse(int button,int state,int x,int y)
{
	if(button==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
		select(x,y);
	if(button==GLUT_RIGHT_BUTTON && state==GLUT_DOWN)
	{
		glViewport(0,0,500,500);
		glMatrixMode(GL_PROJECTION);
		glLoadIdentity();
	gluOrtho2D(0,500,0,500);
		glMatrixMode(GL_MODELVIEW);
		glutPostRedisplay();
	}
}
void display()
{
	glClearColor(1.0,1.0,1.0,1.0);
	glClear(GL_COLOR_BUFFER_BIT);
	draw();
	glFlush();
}
void myReshape(int w,int h)
{
	glViewport(0,0,w,h);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluOrtho2D(0,500,0,500);
	glMatrixMode(GL_MODELVIEW);
}
int main(int argc,char **argv)
```

```
{
	glutInit(&argc,argv);
	glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
	glutInitWindowSize(500,500);
	glutCreateWindow("3d Gasket");
	glutReshapeFunc(myReshape);
	glutDisplayFunc(display);
	glutMouseFunc(mouse);
	glutMainLoop();
	return 0;
}
```