Geometric Objects and Transformations

Chapter 4

■ Introduction:

- We are now ready to concentrate on three-dimensional graphics
- Much of this chapter is concerned with matters such as
  - how to represent basic geometric types
  - how to convert between various representations
  - and what statements we can make about geometric objects, independent of a particular representation.

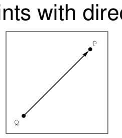
1. Scalars, Points, and Vectors

- We need three basic types to define most geometric objects
  - scalars
  - points
  - and vectors
- We can define each in many ways, so we will look at different ways and compare and contrast them.

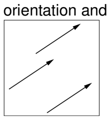
■ 1.1 The Geometric View

- Our fundamental geometric object is a point.
  - In a three-dimensional geometric system, a point is a location in space
  - The only attribute a point possesses is its location in space.
- Our scalars are always real numbers.
  - Scalars lack geometric properties,
  - but we need scalars as units of measurement.

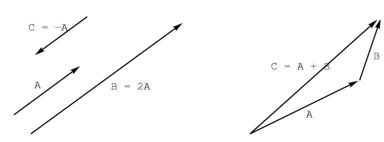
- In computer graphics, we often connect points with directed line segments



- These line segments are our vectors.
- A vector does not have a fixed position
  – hence these segments are identical because their orientation and magnitude are identical.



- Directed line segments can have their lengths and directions changed by real numbers or by combining vectors



■ 1.2 The Mathematical View: Vector and Affine Spaces

- We can regard scalars, points and vectors as members of mathematical sets;
- Then look at a variety of abstract spaces for representing and manipulating these sets of objets.
- The formal definitions of interest to us -- vector spaces, affine spaces, and Euclidean spaces -- are given in Appendix B

- Perhaps the most important mathematical space is the vector space
  - A vector space contains two distinct entities: vectors and scalars.
  - There are rules for combining scalars through two operations: addition and multiplication, to form a scalar field
  - Examples of scalar are:
    - real numbers, complex numbers, and rational functions
  - You can combine scalars and vectors to forma new vector through
    - scalar-vector multiplication and vector-vector addition

- An affine space is an extension of the vector space that includes an additional type of object:
  - The point
- A Euclidean space is an extension that adds a measure of size or distance
- In these spaces, objects can be defined independently of any particular representation
  - But representation provides a tie between abstract objects and their implementation
  - And conversion between representations leads us to geometric transformations

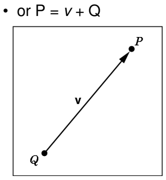
■ 1.3 The Computer Science View

- We prefer to see these objects as ADTs
- Def: ADT
- So, first we define our objects
- Then we look to certain abstract mathematical spaces to help us with the operations among them

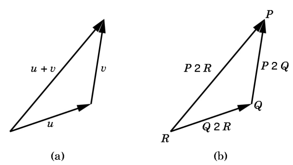
■ 1.4 Geometric ADTs

- Our next step is to show how we can use our types to perform geometrical operations and to form new objects.
- Notation
  - Scalars will be denoted α, β, γ
  - Points will be denoted P, Q, R, ...
  - Vectors will be denoted *u, v, w*, ...
- The magnitude of a vector v is the real number that is denoted by $|v|$
- The operation of vector-scalar multiplication has the property that $|\alpha v| = |\alpha||v|$

- The direction of αv is the same as the direction of v if α is positive.
- We have two equivalent operations that relate points and vectors:
  - First there is the subtraction of two points P and Q. This is an operation that yields a vector
    - $v = P - Q$
    - or $P = v + Q$

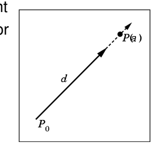


– Second there is the head-to-tail rule that gives us a convenient way of visualizing vector-vector addition.



■ 1.5 Lines

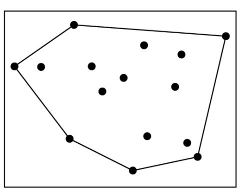
- The sum of a point and a vector leads to the notion of a line in an affine space
  - Consider all points of the form
    - $P(\alpha) = P_0 + \alpha d$
      - P_0 is an arbitrary point
      - d is an arbitrary vector
      - α is a scalar



      - This form is sometimes called the parametric form, because we generate pints by varying the parameter α.

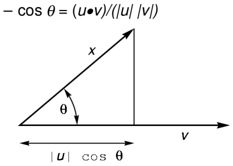
■ 1.7 Convexity

- Def: Convex object
- Def: Convex Hull

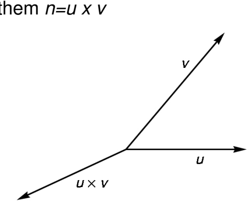


■ 1.8 Dot and Cross Products

- Dot product
  - the dot product of *u* and *v* is written *u•v*
  - If *u•v*=0, then u and v are orthogonal
  - In euclidean space, $|u|^2 = u \bullet u$
  - The angle between two vectors is given by
    - $\cos\theta = (u \bullet v)/(|u|\ |v|)$



- Cross product
  - We can use two non parallel vectors *u* and *v* to determine a third vector *n* that is orthogonal to them *n=u x v*



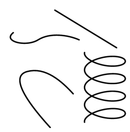
■ 1.9 Planes

- Def: Plane

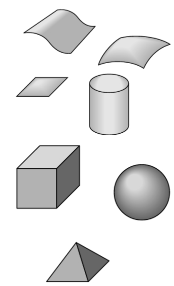
- Def: normal to the plane

2. Three-Dimensional Primitives

- In a three-dimensional world, we can have a far greater variety of geometric objects than we could in two-dimensions.
  - In 2D we had curves,
  - Now we can have curves in space



- In 2D we had objects with interiors, such as polygons,
- Now we have surfaces in space
- In addition, now we can have objects that have volume

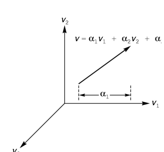


- We face two issues when moving from 2D to 3D.
  - First, the mathematical definitions of these objects becomes complex
  - Second, we are interested in only those objects that lead to efficient implementations in graphic systems
- Three features characterize 3D objects that fit well with existing graphics hardware and software:
  - 1. The objects are described by their surfaces and can be thought of as hollow.
  - 2. The objects can be specified through a s set of vertices in 3D
  - 3. The objects either are composed of or can be approximated by flat convex polygons.

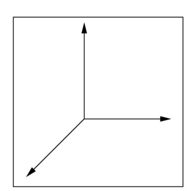
- We can understand why we set these conditions if we consider what most modern graphics systems do best:
  - They render triangles
- Def: tesselate

3. Coordinate Systems and Frames

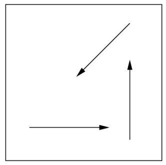
- In a three-dimensional vector space, we can represent any vector w uniquely in terms of any three linearly independent vectors v_1, v_2, and v_3, as
  - $w = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$
- The scalars α_1, α_2, and α_3 are the components of w with respect to the basis functions



- We can write the representation of w with respect to this basis as the column matrix
- We usually think of basis vectors defining a coordinate system



- However, vectors have no position, so this would also be appropriate



  – But a little more confusing.

- Once we fix a reference point (the origin), we will feel more comfortable,
  – because the usual convention for drawing the coordinate axes as emerging from the origin

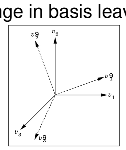
- This representation that requires both the reference point and the basis vectors is called a frame.
  – Loosely, this extension fixes the origin of the vector coordinate system at some point P_0.
  – So, every vector is defined in terms of the basis vectors
  – and every point is defined in terms of the origin and the basis vectors.
  – Thus the representation of either just requires three scalars, so we can use matrix representations

■ 3.1 Changes in Coordinate Systems

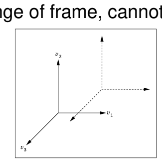
- Frequently, we are required to find how the representation of a vector changes when we change the basis vectors.
  – Suppose $\{v_1, v_2, v_3\}$ and $\{u_1, u_2, u_3\}$ are two bases
  – Each basis vector in the second can be represented in terms of the first basis (and vice versa)
  – Hence:
    - $u_1 = \gamma_{11}v_1+\gamma_{12}v_2+\gamma_{13}v_3$
    - $u_2 = \gamma_{21}v_1+\gamma_{22}v_2+\gamma_{23}v_3$
    - $u_3 = \gamma_{31}v_1+\gamma_{32}v_2+\gamma_{33}v_3$

- The 3x3 matrix is
  – is defined by the scalars and
  – and M tells us how to go one way, and the inverse of M tells us how to go the other way.

- These change in basis leave the origin unchanged



- However, a simple translation of the origin, or change of frame, cannot be represented in this way



■ 3.2 Example of Change of Representation

■ 3.3 Homogeneous Coordinates

■ 3.4 Example of Change in Frames

■ 3.5 Frames and ADTs

- Thus far, our discussion has been mathematical.
- Now we begin to address the question of what this has to do with programming

■ 3.6 Frames in OpenGL

- In OpenGL we use two frames:
  - The camera frame
  - The world frame.
- We can regard the camera frame as fixed
  - (or the world frame if we wish)
  - The model-view matrix positions the world frame relative to the camera frame.
- Thus, the model-view matrix converts the homogeneous-coordinate representations of points and vectors to their representations in the camera frame.

- Because the model-view matrix is part of the state of the system there is always a camera frame and a present-world frame.
- OpenGL provides matrix stacks, so we can store model-view matrices
  - (or equivalently, frames)

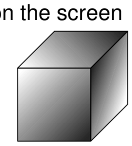
- As we saw in Chapter 2, the camera is at the origin of its frame
  - The three basis vectors correspond to:
    - The up direction of the camera (y)
    - The direction the camera is pointing (-z)
    - and a third orthogonal direction
  - We obtain the other frames (in which to place objects) by performing homogeneous coordinate transformations
    - We will learn how to do this in Section 4.5
    - In Section 5.2 we use them to position the camera relative to our objects.

- Let's look at a simple example:
  - In the default settings, the camera and the world frame coincide with the camera pointing in the negative z direction
  - In many applications it is natural to define objects near the origin.
- If we regard the camera frame as fixed, then the model-view matrix:
  - moves a point (x,y,z) in the world frame to the point (x,y,z-d) in the camera frame.

- Thus by making d a suitably large positive number, we move the objets in front of the camera
- Figure 4.21
- Note that, as far as the user - who is working in world coordinates - is concerned, they are positioning objects as before
- The model-view matrix takes care of the relative positioning of the frames

4. Modeling a Colored Cube

- We now have the tools we need to build a 3D graphical application.
- Consider the problem of drawing a rotating cube on the screen

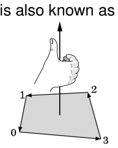


■ 4.1 Modeling of a Cube

- typedef Glfloat point3[3];
- point3 vertices[8] = {{-1.0,-1.0,-1.0}, {1.0,-1.0,-1.0}, {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0}, {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};
- glBegin(GL_POLYGON);
- glVertex3fv(vertices[0]);
- glVertex3fv(vertices[3]);
- glVertex3fv(vertices[2]);
- glVertex3fv(vertices[1]);
- glEnd();

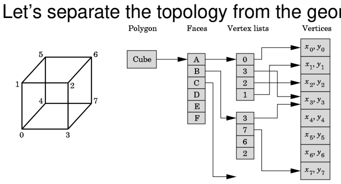
■ 4.2 Inward- and Outward-Pointing Faces

- We have to be careful about the order in which we specify our vertices when we are defining a three-dimensional polygon
- We call a face **outward facing** if the vertices are traversed in a counterclockwise order when the face is viewed from the outside.
  - This is also known as the right-hand rule



■ 4.3 Data Structures for Object Representation

- We could describe our cube
  - glBegin(GL_POLYGON)
    - six times, each followed by four vertices
  - glBegin(GL_QUADS)
    - followed by 24 vertices
- But these repeat data....
- Let's separate the topology from the geometry.



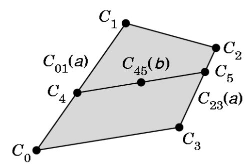
■ 4.4 The Color Cube

- typedef Glfloat point3[3];
- point3 vertices[8] = {{-1.0,-1.0,-1.0}, {1.0,-1.0,-1.0}, {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0}, {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};
- Glfloat colors[8][3]={0.0,0.0,0.0}, {1.0,0.0,0.0}, {1.0, 1.0, 0.0}, {0.0,1.0,0.0}, 0.0,0.0,1.0}, {1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}};

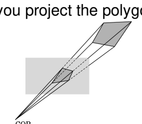
```
void quad(int a, int b, int c, int d)
{
  glBegin(GL_POLYGON);
    glColor3fv(colors[a]); glVertex3fv(vertices[a]);
    glColor3fv(colors[a]); glVertex3fv(vertices[b]);
    glColor3fv(colors[a]); glVertex3fv(vertices[c]);
    glColor3fv(colors[a]); glVertex3fv(vertices[d]);
  glEnd();
}
```


■ 4.5 Bilinear Interpolation

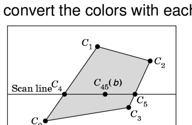
- Although we have specified colors for the vertices of the cube, the graphics system must decide how to use this information to assign colors to points inside the polygon.
- Probably the most common method is bilinear interpolation.



- Another method is Scan-line interpolation
  - pushes the decision off until rasterization.
  - OpenGL uses this method for this as well as other values.
  - First you project the polygon



  - Then convert the colors with each scan line



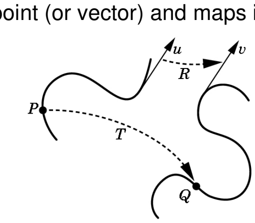
■ 4.6 Vertex Arrays

- Vertex arrays provide a method for encapsulating the information in our data structure such that we could draw polyhedral objects with only a few function calls.
  - Rather than the 60 OpenGL calls:
    - six faces, each of which needs a glBegin, a glEnd, four calls to glColor, and four calls to glVertex.
- There are 3 steps in using vertex arrays
  - First, enable the functionality of vertex arrays.
  - Second, we tell OpenGL where and in what format the arrays are.
  - Third, we render the object

- glEnableClientState(GL_COLOR_ARRAY);
- glEnableClientState(GL_VERTEX_ARRAY);
- glVertexPointer(3, GL_FLOAT, 0, vertices);
- glColorPointer(3, GL_FLOAT, 0, colors);
- Glubyte cubeIndices[24]={0,3,2,1,2,3,7,6,0,4,7,3,1,2,6,5, 4,5,6,7,0,1,5,4};
- We now have a few options regarding how to draw the arrays
  - for(i=0;i<6;i++)
  - glDrawElements(GL_POLYGON, 4, GL_UNSIGNED_BYTE, &cubeIndeces[4*i]);
  - glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, cubeIndeces);

5. Affine Transformations

- A transformation is a function that takes a point (or vector) and maps it into another .



- We can represent this as:
  - Q=T(P) and *v*=R(*u*)
- If we write both the points and vectors in homogeneous coordinates, then we can represent both vectors and points in 4-D column matrices and define a single function
  - q=f(p) and v=f(u)
- This is too general, but if we restrict it to linear functions, then we can always write it as
  - v = Au
  - where v and u are column vectors (or points) and A is a square matrix.

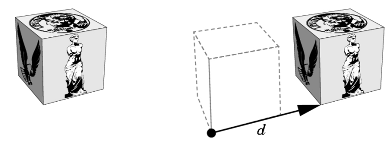
- Therefore,
  - we need only to transform the homogeneous coordinate representation of the endpoints of a line segment to determine completely a transformed line.
  - Thus, we can implement our graphics systems as a pipeline that passes endpoints through affine transformation units, and finally generate the line at rasterization stage.
- Fortunately, most of the transformations that we need in computer graphics are affine.
  - These transformations include rotation, translation, and scaling.
  - With slight modifications, we can also use these results to describe the standard and parallel projections.

6. Rotation, Translation, and Scaling

- In this section,
  - First, we show how we can describe the most important affine transformations independently of any representation.
  - Then we find matrices that describe these transformations
- In section 4.8 we shall see how these transformations are implemented in OpenGL

■ 6.1 Translation

- Translation is an operation that displaces points by a fixed distance in a given direction

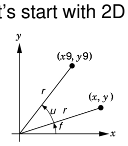


(a) (b)

- To specify a translation, we need only to specify a displacement vector d

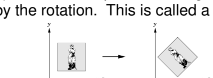
■ 6.2 Rotation

- Rotation is more difficult to specify than translation, because more parameters are involved
- Let's start with 2D rotation about the origin

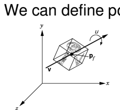


  - These equations can be written in matrix form as:

- We expand this to 3D in Section 4.7
- Note that there are three features of this transformation that extend to other rotations.
  - 1) There is one point - the origin - that is unchanged by the rotation. This is called a fixed point.

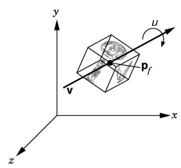


  - 2) We can define positive rotations about other axes

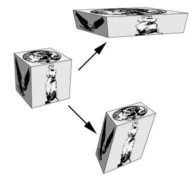


  - 3) 2D rotation in the plane is equivalent to 3D rotation about the x axis.

- We can use these observations to define a general 3D rotation that is independent of the frame.
  - To do this we must specify:
    - a fixed point: P_f
    - a rotation angle: θ
    - a line or vector about which to rotate:

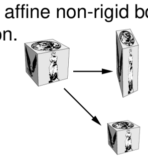


- Rotations and translation are known as rigid-body transformations.
  - No combination can alter the shape of an object,
  - They can alter only the object's location and orientation
- The transformation given in this figure are affine, but they are not rigid-body transformations

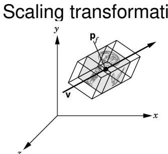


■ 6.3 Scaling

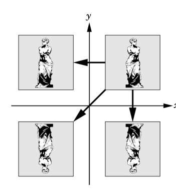
- Scaling is an affine non-rigid body transformation.



- Scaling transformations have a fixed point



- Hence, to specify a scaling,
  - we can specify a fixed point,
  - a direction in which we wish to scale,
  - and a scale factor α.
    - For $\alpha > 1$, the object gets longer
    - for $0 <= \alpha < 1$ the object get smaller
    - Negative values of α give us reflection



7. Transformations in Homogeneous Coordinates

- In most graphics APIs we work with a representation in homogeneous coordinates.
- And each affine transformation is represented by a 4x4 matrix of the form:

■ 7.1 Translation

- Translation displaces points to a new position defined by a displacement vector.
  - If we move p to p' by displacing by a distance d then
    - p' = p + d
    - or p' = Tp
    - where
    - T is called the translation matrix

■ 7.2 Scaling

- A scaling matrix with a fixed point at the origin allows for independent scaling along the coordinate axes.
  - $X' = \beta_x x$
  - $y' = \beta_y y$
  - $z' = \beta_z z$
- These tree can be combined in homogeneous for as
  - $p' = Sp$

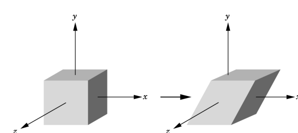
■ 7.3 Rotation

- We first look at rotation with a fixed point at the origin.
  - We can find the matrices for rotation about the individual axes directly from the results of the 2D rotation developed earlier.
  - Thus,
    - $x' = x \cos\theta - y \sin\theta$
    - $y' = x \sin\theta + y \cos\theta$
    - $z' = z$
    - of $p' = R_z p$

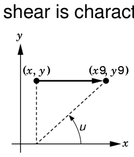
- We can derive the matrices for rotation about the x and y axes through an identical argument.
  - And we can come up with:

■ 7.4 Shear

- Although we can construct any affine transformation from a sequence of rotations, translations, and scaling, there is one more affine transformation that is of such importance that we regard it as a basic type, rather than deriving it from others.



- Using simple trigonometry, we can see that each shear is characterized by a single θ



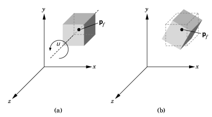
- The equation for this shear is:
  - $x' = x + y \cot\theta$, $y' = y$, $z' = z$
- Leading to the shear matrix :

8. Concatenation of Transformations

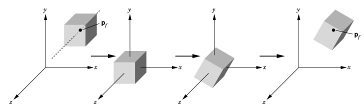
- In this section, we create examples of affine transformations by multiplying together, or concatenating, sequences of basic transformations that we just introduced.
  - This approach fits well with our pipeline architectures for implementing graphics systems.

■ 8.1 Rotation About a Fixed Point

- In order to do this:

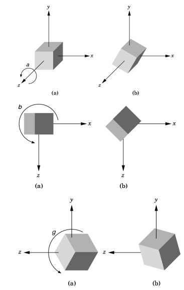


- You do this:



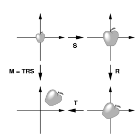
■ 8.2 General Rotation

- An arbitrary rotation about the origin can be composed of three successive rotations about the tree axes.



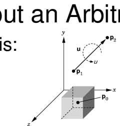
■ 8.3 The Instance Transformation

- Objects are usually defined in their own frames, with the origin at the center of mass, and the sides aligned with the axes.
- The instance transformation is applied as follows:
  - First we scale
  - Then we orient it with a rotation matrix
  - Finally we translate it to the desired orientation.

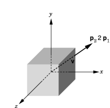


■ 8.4 Rotation About an Arbitrary Axis

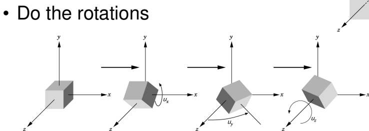
- In order to do this:



- We move the fixed point to the origin



- Do the rotations



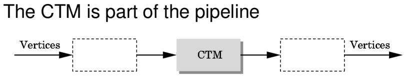
- Translate back

9. OpenGL Transformation Matrices

- In OpenGL there are three matrices that are part of the state.
  - We shall use only the model-view matrix in this chapter.
  - All three can be manipulated by a common set of functions,
  - And we use glMatrixMode to select the matrix to which the operations apply.

■ 9.1 The Current Transformation Matrix

- This is the matrix that is applied to any vertex that is defined subsequent to its setting.
- If we change the CTM we change the state of the system.
- The CTM is part of the pipeline

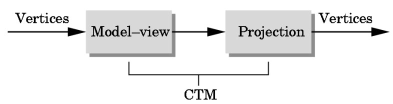


  - Thus, if p is a vertex, the pipeline produces Cp

- The functions that alter the CTM are:
  - Initialization (C ← I)
  - Post-Multiplication (C ← CT)

9.2 Rotation, Translation, and Scaling

- In OpenGL,
  - the matrix that is applied to all primitives is the product of the model-view matrix (GL_MODELVIEW) and the projection matrix (GL_PROJECTION)
  - We can think of the CTM as the product of these two.



- We can load a matrix with
  - glLoadMatrixf(pointer_to_matrix);
- or set it to the identity with
  - glLoadIdentity( );
- Rotation, translation, and scaling are provided through three functions:
  - glRotate(angle, vx, vy, vz);
    - angle is in degrees
    - vx, vy, and vz are the components of a vector about which we wish to rotate.
  - glTranslate(dx, dy, dz);
  - glScale(sx, sy, sz);

9.3 Rotation About a Fixed Point in OpenGL

- In Section 4.8 we showed that we can perform a rotation about a fixed point other than the origin.
  - (translate, rotate, and translate back)
  - Here is how you do it in OpenGL
    - glMatrixMode(GL_MODELVIEW);
    - glLoadIdentity( );
    - glTranslatef(4.0, 5.0, 6.0);
    - glRotatef(45.0, 1.0, 2.0, 3.0);
    - glTranslatef(-4.0, -5.0, -6.0);

9.4 Order of Transformations

- You might be bothered by the apparent reversal of the function calls.
- The rule in OpenGL is this:
  - The transformation specified most recently is the one applied first.
    - $C \leftarrow I$
    - $C \leftarrow CT$
    - $C \leftarrow CR$
    - $C \leftarrow CT$
  - each vertex that is specified after the model-view matrix has been set will be multiplied by C thus forming the new vertex
    - $q = Cp$

9.5 Spinning of the Cube

```
void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity( );
  glRotatef(theta[0], 1.0, 0.0, 0.0);
  glRotatef(theta[1], 0.0, 1.0, 0.0);
  glRotatef(theta[2], 0.0, 0.0, 1.0);
  colorcube( );
  glutSwapBuffers( );
}
```



```
void mouse(int btn, int state, int x, int y)
{
  if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    axis = 0;
  if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN)
    axis = 1;
  if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
    axis = 2;
}
```

- void spinCube( )
- {
- theta[axis] += 2.0;
- if (theta[axis] > 360.0) theta[axis] -= 360.0'
- glutPostRedisplay( );
- }

- void mkey(char key, int mousex, int mousey)
- {
- if(key=='q' || key=='Q')
- exit( );
- }

■ 9.6 Loading, Pushing, and Popping Matrices

- For most purposes, we can use rotation, translation , and scaling to form a desired transformation matrix.
- In some circumstances, however, such as forming a shear matrix, it is easier to set up the matrix directly.
  – glLoadMatrixf(myarray)
- We can also multiply on the right of the current matrix by using
  – glMultMatrixf(myarray);

- Sometimes we want to perform a transformation and then return to the same state as before its execution.
- We can push the current transformation matrix on a stack and recover it later.
- Thus we often see the sequence
  – glPushMatrix( );
  – glTransletef(. . .);
  – glRotatef(. . .);
  – glScalef(. . .);
  – // draw object here
  – glPopMatrix( );

11. Summary

- In this chapter, we have presented two different-but ultimately complementary-points of view regarding the mathematics of Computer Graphics.
  - One is the mathematical abstraction of the objects with which we work in computer graphics is necessary if we are to understand the operations that we carry out in our programs
  - The other is that transformations (and homogeneous coordinates) are the basis for implementations of graphics systems
- Finally we provided the set of affine transformations supported by OpenGL, and discussed ways that we could concatenate them to provide all affine transformations.

12. Suggested Readings

- Homogeneous coordinates arose in Geometry(51) and were alter discovered by the graphics community(81)
  – Their use in hardware started with the SGI Geometry Engine (82)
  – Modern hardware architectures use Application Specific Integrated Circuits (ASIC's) that include homogeneous coordinate transformations
- Quaternions were introduced to computer graphics by Shoemaker(85) for use in animation
- Software tools such as Mathematica(91) and Matlab(95) are excellent aids for learning to manipulate transformation matrices

Exercises -- Due next Monday

- 4.5
- 4.8
- 4.13
- 4.17