

Chapter 1

3D geometry III

Now that we have covered the basics of matrix transformations we will see how they are typically used in computer graphics. We will discuss various modeling manipulation and imaging operations.

1.1 World Object and eye frames

Very often, when describing the geometry of a scene, it is useful to have a basic frame \vec{w}^t called the world frame. Other frames can then be described with respect to this world frame.

We also may associate a frame \vec{o}_i^t with every object i in the scene. We can now express the location of parts of the objects using coordinates with respect to the objects coordinate system. To move the object, we can change \vec{o}_i^t and not change any of the coordinate vectors.

The relationship between the object's coordinate system and the world frame is represented by the matrix O_i . That is,

$$\vec{o}_i^t = \vec{w}^t O_i$$

In a computer program, we can not store abstract vectors, or geometric points, we can only store numbers. So in a program, we will store the matrix O_i , with the understanding that it relates the world frame to object i 's coordinate system using the equation above.

In the real world, when we want to create a 2d picture of a 3d environment, we place a camera somewhere in the scene. The position of each object in the picture is based on its 3d relationship to the camera, ie. its coordinates with respect to an appropriate basis. In computer graphics we achieve this by having a frame

$$\vec{e}^t = \vec{w}^t E$$

called the eye frame. Again in a computer program, we will only be able to store the matrix E , but our interpretation will be as above.

Suppose I wish to make a picture of a point $\tilde{p} = \vec{w}^t \mathbf{c}$. I need to have a procedure that takes the information about the point, and uses this to create an image. Such a procedure is available to us in a graphics API such as OpenGL. In a later chapter, we will learn exactly how this procedure works. When we call this procedure, we need to inform it about the location of the point. Again we cannot pass abstract vectors or geometric points to a subroutine, we can only pass its coordinate vector with respect to some agreed upon frame. Ultimately what is important to the image composition is the points relationship to the camera geometry. This suggests that we pass it the coordinates with respect to the eye frame.

$$\tilde{p} = \vec{w}^t \mathbf{c} = \vec{e}^t E^{-1} \mathbf{c}$$

In other words we should pass the coordinates $E^{-1} \mathbf{c}$.

1.1.1 Motion is relative

Suppose I take two pictures, and some point changes its position in the two images. There are two possibilities, the point moved in one way, or the camera moved in some opposite way. Lets see what this means in terms of frames.

For the point to have moved in the image, different coordinates must have been passed to the image making routine. Suppose in the first call the coordinates passed are \mathbf{d} , and in the second call the coordinates passed are $\mathbf{b} = R\mathbf{d}$.

One way to interpret this is that the camera has stayed motionless, and the point has been transformed.

$$\vec{e}^t \mathbf{d} \Rightarrow \vec{e}^t R\mathbf{d}$$

Under this interpretation the point has been transformed by applying the R (say a rotation) operation with respect to the eye frame.

Another way to interpret this is that the point has stayed motionless, and the camera is now using a new eye frame \vec{e}'^t . What is this new eye frame? Since the point hasn't moved,

$$\begin{aligned} \vec{e}^t \mathbf{d} &= \vec{e}'^t \mathbf{b} \\ &= \vec{e}'^t R\mathbf{d} \end{aligned}$$

This implies

$$\vec{e}'^t = \vec{e}^t R^{-1}$$

In english, the new frame is obtained by starting with the original coordinate system, and performing the inverse transformation with respect to the original eye frame.

1.2 Moving Things around

In an interactive 3D program, we will often want to move things, such as objects and even the eye point of view to various locations and orientations. We will now discuss how this is commonly done.

Updates to an objects position and orientaton is implemented by appropriately updating its frame, which is represented by its matrix O_i .

When i say that i wish to rotate or tranlate an object, I must state with respect to which frame I wish to move it. The easiest way to rotate or translate an object is with respect to its own coordiate system. This means that the object will rotate about its own center. The directions of rotation and translations will be determined by its current orientation. Let us call Q the matrix representing the requested rotation/translation. Then

$$\vec{o}_i \Rightarrow \vec{o}_i Q = \vec{w}^t O_i Q \equiv \vec{o}_i^t$$

so $O_i := O_i Q$.

Suppose that instead i wish to transform the operation with respect to the world frame. This means that it will be rotated about the center of the world frame, adn that the directions of the rotation and translation will be determined by the global directions. Here is how we figure out the proper way to update the matrices

$$\vec{o}_i^t = \vec{w}^t O_i \Rightarrow \vec{w}^t Q O_i \equiv \vec{o}_i^t$$

and so $O_i := Q O_i$ In this case we first rewrote \vec{O}_i^t with respect to \vec{w}^t so we could then apply Q using the “left of” rule.

Suppose that instead i wish to perform the transformation operation with repect to the frame of O_2 . Perhas O_2 represents the director of the scene. When he says “left” he means his left. Here is how we figure out the update

removed for assigment 3

1.3 Current Transformation Matrix and Stack

It is not very convenient for a graphics programer to have to mulptly every set of coordinates by some matrix before sending it to the picture making procedure. To help organize this process, openGL provides a help my maintaining something called a current transformation matrix. When the user sets the current transformation matrix, from then on, all coordinate vectors passed to openGL are first multiplied by the CTM before being passed on to the picture making routine. In other words, openGL is expecting coordinates with respect to the frame

$$\vec{e}^t \text{CTM}$$

The CTM starts out being undefined, but can be set to the identity by calling `glLoadIdentity()`. If the CTM is the identity matrix, then all coordinate vectors are unchanged on their trip to the picture maker. In this case we can say that OpenGL is expecting coordinate vectors in the eye frame.

To change the CTM, the user can call `glMultMatrix(glmatrix M)`. This updates the CTM as $CTM := CTM \cdot M$. Sometimes it is convenient to be able to back up and go to some previously defined CTM. To do this, OpenGL maintains a matrix stack. When one calls `glPushMatrix()` a copy of the CTM is made, pushed on a stack, and is then used as the CTM. The user can now change the CTM at will. When the user calls `glPopMatrix()` the top matrix on the stack is popped, and the next matrix down once again becomes the CTM. This allows us to go back to a previous CTM.

At this point we are ready to write a basic subroutine to draw a scene in OpenGL.

```
void display(void)
{
    static GLfloat lightOnePosition[] = {.0, .2, 1, 0.0};

    /* clear the image*/
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glLoadIdentity();
    /* at start we are using the eye cs */

    setupEye();
    /* now we are using the world frame */

    /* tell gl where a light is */
    glLightfv(GL_LIGHT1, GL_POSITION, lightOnePosition);

    glPushMatrix();
    setupObj1();
    /* now we are using the obj1 frame */
    displayObj1();
    glPopMatrix();
    /* now we are using the world frame */

    setupObj2();
    /* now we are using the obj1 frame */
    displayObj2();
}
```

```

    glPopMatrix();
    /* now we are using the world frame */

    /* tell gl we are done with this image*/
    glutSwapBuffers();
}

void setupEye(void)
{
    matrix4 EMatInv = matInverse(EMat);
    mat2GL(EMatInv, glMat); //change data format
    glMultMatrixd(glMat);
}

void setupObj1()
{
    mat2GL(O1Mat, glMat);
    glMultMatrixd(glMat);
}

void displayObj1()
{
    glutSolidCube(2.0);
}

void displayObj1()
{
    glutSolidCube(1.5);
}

```

The first thing we do is clear the screen (and the z-buffer). The next thing we do is set the CTM to be the identity. OpenGL is now expecting coordinate vectors with respect to the eye frame. This is not convenient, since every time we move the eye, we will need to change the coordinate vectors. We may instead wish to be passing coordinate vectors with respect to the world frame. So we will have to multiply these coordinate vectors by E^{-1} . We accomplish this in `setupEye()` which right multiples E^{-1} to the CTM. Now we can send coordinate vectors with respect to the world coordinate system and OpenGL will do the rest. For short, we say we are in the world frame.

Now we want to draw some object. We wish to express the coordinate vectors of the geometric points making up the object with respect to the object frame instead of the world frame. This way if we move the object, we don't need to recalculate the

coordinate vectors representing the points on the object. Given a coordinate vector \mathbf{c} with respect to $\vec{\mathbf{o}}_i^t$, the way to obtain the coordinate vector of the point with respect to the first objects frame $\vec{\mathbf{w}}^t$ is to multiply it by O_i . We accomplish this in `setupObject1()` which right multiplies the CTM with O_i . We can now send coordinate vectors with respect to the objects own frame. Since we are interested in later drawing other objects with their own frames, we wrap the calls by `glPushMatrix()` and `glPopMatrix()`.

The drawn objects are simply cubes. The calls to `glutSolidCube(sideLen)` simply sends the geometry of 12 triangles describing a cube of the desired side length. The coordinate vectors sent to openGL describe a cube that centered around the origin of the object frame. For example, if `sideLen` is `.5`, then all of the coordinate vectors will be of the form $[\pm 1, \pm 1, \pm 1]^t$. When these coordinate vectors are multiplied by the CTM, openGL will obtain the desired coordinate vectors of the points with respect to the eye frame, and the proper picture is made.

If we want to move the first object, we must simply update O_1 appropriately. If we want to move the second object, we must simply update O_2 appropriately. If we want to move the eye point, we must simply update E appropriately.

For completeness, we include here the rest of the code that is necessary to make our images in openGL.

```
#include <stdio.h>
#include <stdlib.h>
#include "GL/glut.h"
#include "coords3.h"
#include "matrix4.h"

GLdouble glMat[16];
matrix4 O1Mat;
matrix4 O2Mat;
matrix4 EMat;

void main(int argc, char **argv) {
    /* glut stuff */
    glutInit(&argc, argv);                /* Initialize GLUT */
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH );
    glutInitWindowSize(400, 200);
    glutCreateWindow("my window");        /* Create window with given ti-
    tle */
    glViewport(0,0,400,200 );

    glutDisplayFunc(display);              /* Set-up callback func-
    tions */
}
```

```
    glutReshapeFunc(reshape);
    glutMotionFunc(mouseMove);

    setupGLstate();
    initializeMatricies();
    glutMainLoop();                               /* Start GLUT event-
processing loop */
}

void setupGLstate(){
    GLfloat lightOneColor[] = {1, 1, 1, 1};
    GLfloat globalAmb[] = {.1, .1, .1, 1};

    // turn on culling of back faces
    glEnable(GL_CULL_FACE);
    glFrontFace(GL_CCW);

    //enable z-buffer
    glEnable(GL_DEPTH_TEST);

    //set the clear color to black
    glClearColor(0,0,0,0);

    //enable smooth shading as opposed to flat
    glShadeModel(GL_SMOOTH);

    //enable lighting
    glEnable(GL_LIGHT1);
    glEnable(GL_LIGHTING);
    glEnable(GL_NORMALIZE);
    glEnable(GL_COLOR_MATERIAL);

    //describe the one point light in the scene
    glLightfv(GL_LIGHT1, GL_DIFFUSE, lightOneColor);

    //describe the ambient light in the scene
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, globalAmb);

    //use the color command to update the ambient and diffuse
    //properties of the upcoming surface material.
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
}

void reshape(int w, int h){
    float ar = (float)(w)/h;
    glViewport(0, 0, w, h);
}
```

```

glMatrixMode(GL_PROJECTION);
glLoadIdentity();

// magic imaging commands
gluPerspective( 20.0, /* field of view in degrees */
ar, /* aspect ratio */
1.0, /* Z near */
100.0 /* Z far */);

glMatrixMode(GL_MODELVIEW);

glutPostRedisplay();
}

void initializeMatricies()
{
    EMat = transMatrix(0,0,10);
    O1Mat = ident();
    O2Mat = transMatrix(3,0,0);
}

void mouseMove(int x, int y)
{
    updateObj1Matrix();
    glutPostRedisplay();
}

void updateObj1Matrix()
{
    obj1Mat = matMatMul(obj1Mat, roty2mat(2));
}

```

We will be working under the assumption that the scene's viewer is positioned at the origin of \vec{e}^t looking down the negative z axis if \vec{e}^t . We will also assume that the top of the viewers head or camera will be in the positive y axis of \vec{e}^t .

In the procedure `initializeMatricies()`, E is set to be a translation by 10 in the 3rd axis (z) direction. E is used to represent the relationship between the world and eye frames: $\vec{e}^t = \vec{w}^t E$, so \vec{e}^t is obtained by starting with \vec{w}^t and translating it by 10 along \vec{w}^t 's z axis. The the viewer will be looking back towards the origin of the world frame.

O_1 is initialized as the identity matrix, so $\vec{o}_1^t = \vec{w}^t$ So the first cube should appear in the center of the obtained image.

We also initialized O_2 using a translation in the positive x direction. O_2 is used to represent the relationship between the world and second object frames: $\vec{o}_2^t = \vec{w}^t O_2$, so \vec{o}_2^t is obtained by starting with \vec{w}^t and translating the it by 3 along \vec{w}^t 's x axis. As a result, the second cube should appear off to the right in the image.

When the user moves the mouse, the first objects matrix is updated as $O_1 := O_1 R$, where R is a rotation about the y axis. As a result, $\vec{o}_1^t := \vec{o}_1^t R$, its frame is rotated about the y axis of \vec{o}_1^t , (its own y axis).

1.4 Hierarchical Modeling

Many objects are naturally modeled hierarchially. In a tree, the trunk starts out at the ground, large branches come off the trunk at various locations that are best described relative to the branch. Small branches come off of the large braches at postions that are best described relative to the large branch. Branches end in leaves whose positions are best described with respect to thier supporting branches. In such a setting it is best to describe each successive "child" frame (trunk, big branch, small branch, leaf) with respect to its previous "parent" frame: $\vec{c}^t = \vec{p}^t M$. Where M is some simple transformation. The sequence of frame changes is easily described by a sequence of right multiplications onto the CTM. Such a hierarchical model can be best programed using nested `glPushMatrix()` and `glPopMatrix()`. To help with this process, `OpenGL` also provides a number of convenient routines.

`glTranslatef(x, y, z)` updates the CTM by right muliplying it with a tranlation matrix.

`glRotatef(deg, axRotX, axRotY, axRotZ)` creates a matrix that rotates `deg` degrees counterclockwise around the axis of rotation described by `axRotX`, `axRotY`, `axRotZ`, and right multiplies it onto the CTM.

`glScalef(sx, sy, sz)` applies (non-uniform) scaling.

Using these commands, we can change our first object's drawing code to be

```
void displayObj1()
{

    glColor3f(1,1,.3);
    glutSolidCube(1);

    glPushMatrix();
    glTranslatef(1.5,0,0);
    glRotated(45, 1,0,0);
    glRotated(30, 0,1,0);
    glColor3f(1,1,.3);
    glutSolidCube(2);
    glPopMatrix();
}
```

```

glPushMatrix();
glScalef(.5,1,1);
glColor3f(1,0,.3);
glutSolidCube(2);
glPopMatrix();
}

```

1.5 The lookat command

Now that we know how to read equations involving frames, we are in a position to directly write down affine matrices to perform desired tasks. One of these tasks is describing the relationship between the world frame and the eye frame.

$$[\vec{x}_e \ \vec{y}_e \ \vec{z}_e \ \tilde{o}_e] = \vec{e}^t = \vec{w}^t E$$

A useful interface to do this using a `lookat` command. In OpenGL the command is

```
gluLookAt( eyex, eyey, eyez, centerx, centery, centerz, upx,
          upy, upz).
```

The three `eye` arguments specify where the origin of the eye frame should be. Specifically it should be

$$\tilde{o}_e = \vec{w}^t \begin{bmatrix} ex \\ ey \\ ez \\ 1 \end{bmatrix}$$

The `center` arguments specify that the eye should be looking along the direction

$$\vec{w}^t \begin{bmatrix} cx - ex \\ cy - ey \\ cz - ez \\ 0 \end{bmatrix}$$

Since our convention is that the eye is looking down its own $-z$ axis, this tells us what the \vec{z} vector, the third vector in the eye frame should be:

$$\vec{z}_e = \vec{w}^t \begin{bmatrix} ex - cx \\ ey - cy \\ ez - cz \\ 0 \end{bmatrix}$$

This is not yet exactly correct. We want the vectors that make up the eye frame to be unit length, and to be orthogonal to each other. So one must first normalize the coordinate vector $[ex - cx, ey - cy, ez - cz]^t$.

We must still specify the \vec{x}_e and \vec{y}_e axes of the eye frame. We do this by specifying which direction should be up.

$$\vec{y}_e = \vec{w}^t \begin{bmatrix} ux \\ uy \\ uz \\ o \end{bmatrix}$$

This is not yet exactly correct either. First of all, the up vector must be normalized. But moreover, what shall we do if the user specifies an up direction that is not orthogonal to \vec{z}_e . The answer is to pick for \vec{y}_e a vector that is orthogonal to \vec{z}_e , and most like the direction requested by the up arguments.

To do this, we will want to project the requested up direction onto the plane perpendicular to \vec{z}_e . This can be done as follows. First we find an \vec{x}_e axis that is perpendicular to both $\vec{u}\vec{p}$ and \vec{z}_e . This is done by using the cross product

$$\vec{x}_e = \vec{u}\vec{p} \times \vec{z}_e$$

We accomplish this by applying the cross product operation from vector algebra on the two coordinate vectors. This gives us the eye frames “rightward” direction. We then find an \vec{y}_e axis that is perpendicular to both \vec{z}_e and \vec{x}_e . This is done by using the cross product

$$\vec{y}_e = \vec{z}_e \times \vec{x}_e$$

Given these computations, we can directly write down the numbers that define the matrix

$$\begin{bmatrix} \vec{x}_e & \vec{y}_e & \vec{z}_e & \vec{o}_e \end{bmatrix} = \begin{bmatrix} \vec{x}_w & \vec{y}_w & \vec{z}_w & \vec{o}_w \end{bmatrix} \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The only time this will fail is if the requested up direction is the same as the requested \vec{z}_e direction. In this case, all we can do is report an error to the caller.

1.6 Transforming Normals

In computer graphics, we will often use the normals of surfaces to determine how a surface point should be shaded. We need to understand how the normal of a surface transforms when the surface points undergo affine transformations.

Lets start with a few examples. Suppose I have a the normal vector to some point, and I rotate all of the geomery, then the normal simply rotates along with the point. But suppose I scale the geometry in the y direction by some value less than one. The object gets shorter and flatter, and the normal now becomes more vertical. In this case, the normal vector gets scaled in the y direction by some value greater than one. Why does this happen?

Let us define the normal to the surface to be a vector that is orthogonal to the tangent plane. The tangent plane is a plane of vectors that are defined by subtracting (infitesimally nearby surface points, and so we have

$$\vec{n} \cdot (\tilde{p}_1 - \tilde{p}_2) = \vec{n} \cdot \vec{dp} = 0$$

In some fixed coordinate system, this could be expressed as

$$\begin{bmatrix} nx & ny & nz & 0 \end{bmatrix} \begin{bmatrix} dx \\ dy \\ dz \\ 0 \end{bmatrix} = 0$$

Suppose I transform all of my points by applying an affine transformation using an affine matrix A . What vector remains orthogonal to any tangent vector? This can be found out as follows

$$\begin{aligned} & \begin{bmatrix} nx' & ny' & nz' & 0 \end{bmatrix} \begin{bmatrix} dx' \\ dy' \\ dz' \\ 0 \end{bmatrix} = 0 \\ & (\begin{bmatrix} nx & ny & nz & 0 \end{bmatrix} \mathbf{A}^{-1}) (\mathbf{A} \begin{bmatrix} dx \\ dy \\ dz \\ 0 \end{bmatrix}) = 0 \end{aligned}$$

and so

$$\begin{bmatrix} nx' \\ ny' \\ nz' \\ 0 \end{bmatrix} = \mathbf{A}^{-t} \begin{bmatrix} nx \\ ny \\ nz \\ 0 \end{bmatrix}$$

So the coordinates of the normal is transformed using the inverse transpose of the matrix A . This rule explains the two examples we looked at above. If A is some rotation, then its inverse transpose is the rotation A . If A is a non uniform scaling matrix, which is a diagonal matrix, then its inverse transpose is the same as its inverse.