Modeling with Transformations

Prerequisites

This module requires some understanding of 3D geometry, particularly a sense of how objects can be moved around in 3-space. The student should also have some sense of how stacks can be used in programming.

Introduction

Transformations are probably the key point in creating significant images in any graphics system. It is extremely difficult to model everything in a scene in the place where it is to be placed, and it is even worse if you want to move things around in real time through user control. Transformations let you define each object in a scene in any space that makes sense for that object, and then place it in a scene and move the entire scene around as the scene is actually viewed.

There are several kinds of transformations in computer graphics: projection transformations, viewing transformations, and modeling transformations. Projection transformations are those defined by the system when you choose perspective or orthogonal projections; viewing transformations are those set up when you define your view environment, and modeling transformations are those you use to create the items in your scene. In OpenGL, there are only two kinds of transformations: projection transformations and modelview transformations. The latter includes both the viewing and modeling transformations.

Among the modeling transformations, there are three fundamental kinds: rotations, translations, and scaling. In OpenGL, these are applied with the built-in functions (actually function sets) `glRotate`, `glTranslate`, and `glScale`, respectively. As we have found with other OpenGL function sets, there are several different versions of each of these, varying only in the kind of parameters they take.

The use of simple modeling and transformations allows you to generate more complex graphical objects, but these objects can take significant time to display. You may want to store these objects in pre-compiled display lists that can execute much more quickly.

Definitions

Transformation: a transformation is a function that takes geometry and produces new geometry. The geometry can be anything a computer graphics systems works with — a projection, a view, or an object to be displayed. We have already talked about projections and views, so in this section we will talk about projections as modeling tools. In this case, the basic transformations we work with are the three we mentioned earlier: rotations, translations, and scaling. Below we look at each of these transformations individually, and we will give code examples for these examples later in this module.

Rotations take an object and rotate it around the origin of the geometry in which the object is defined. In fact, it rotates everything in space that is defined while the rotation is in place. As an example, let's take a plane polygon in space (to make it simple, let's take a square two units on a side, defined to be parallel to the Y-Z plane and centered on the point (12.0, 0.0, 0.0), and let's rotate that by 45 degrees at a time around the Z-axis. We could use a more sophisticated figure, but we use a simple square to focus on only the rotations. In the figure below, the original square is colored red and the rotated squares are all colored white, and the axes have red, green, and blue arrows for the X-, Y-, and Z-axis respectively:
Translations take an object and move it, without changing its size or orientation, to another point in space. Again, they move everything in space that is defined while the translation is in place. Choosing our simple square again, and again starting with the square in its original position, let’s translate it two units at a time down the X-axis toward, and through, the origin to its opposite position on the X-axis at (-12.0, 0.0, 0.0).

Scaling takes an object changes its size in space, depending on the scaling factor of each dimension of the object. Again, it changes the size of each dimension of everything that is defined while the scaling is in place. We will again work with our simple square, scaling by fractions from one down to near zero. We scale uniformly — all the dimensions are scaled by the same amount — even though the dimensions can be scaled differently. The main thing you should note is that the square “moves” toward the origin because the X-value of the square is also scaled down as the size is scaled down.
Look at these figures carefully to understand the effect of the transformations; some of the squares may be invisible because they are seen edge-on, or may be difficult to see because of where they are drawn or because of their size. We will explain how the examples are all done later, but it may not be quite as simple as it would look.

Composite transformations: In order to achieve the image you want, you may need to apply more than one simple transformation to achieve what is called a composite transformation. For example, if you want to create a rectangular box with height A, width B, and depth C, with center at (C1,C2,C3), and oriented at a 30 degree angle relative to the Z-axis, starting with a cube one unit on a side and with center at the origin, one would want to apply the following sequence of operations:

- first, scale the cube to the right size to create the rectangular box,
- second, rotate the cube to the right orientation, and
- third, translate the cube to the right position.

This sequence is critical because of the way transformations work in the whole space as illustrated above. For example, if we rotated first and then scaled with different scale factors in each dimension, we would introduce distortions in the box. If we translated first and then rotated, the rotation would move the box to an entirely different place. So the order is very important.

In OpenGL, the sequence of transformations is last-specified, first-applied. Another way to think about this is that the transformations are applied with the transformation closest to the geometry applied first. So the sequence we see above would be achieved by the following sequence of function calls:

```c
glTranslatef( C1, C2, C3);
glRotatef(30.0, 0.0, 0.0, 1.0);
glScalef(A, B, C);
cube();
```

This sequence is exactly the opposite of the sequence noted above, because of the last-specified, first-applied nature of transformations. A word about the rotate function might be in order: note that the first parameter is the angle of rotation in degrees, and the next three parameters are the components of a vector that is the fixed vector for the rotation.

In general, we can think of the transformations applied to a model by considering the total sequence of transformations in the order in which they are specified:

```
P V T0 T1 T2 T3 T4 T5 T6 ... Tn Tn+1 ... Tlast
```

Here, P is the projection transformation, V is the viewing transformation, and T0, T1, ... Tlast are the transformations specified in the program to model the scene, in order (T1 is first, Tlast is last). The projection transformation is defined in the reshape function; the viewing transformation is defined in the init function or at the beginning of the display function so it is defined at the beginning of the modeling process. But the sequence is actually applied in reverse: Tlast is actually applied first, and V and finally P are applied last. You need to understand this sequence very well, because it’s critical to understand how you build complex, hierarchical models.

Transformation stacks and their manipulation: in defining a scene, we often want to define some standard pieces and then assemble them in standard ways, and then use the combined pieces to create additional parts, and go on to use these parts in additional ways. To do this, we need to create individual parts through functions that do not pay any attention to ways the parts will be used later. Let us consider a possible example of a (very simple!!) rabbit’s head. This would have a large ellipsoidal head, two small spherical eyes, and two middle-sized ellipsoidal ears. So we will use the ellipsoid (actually a scaled sphere, for which we will use the gluSphere function) as our basic part and will put it in various places with various orientations as needed.

The key issue is that we will use some transformations when we put a simple part in place, but we need to undo the effect of those transformations when we put the next part in place. In effect, we
need to mark the list of transformations above whenever we start to place a new part, and be able to
return to that marked point in the list (discarding the transformations past that mark) to begin to
place the next part. Note that we are always adding and discarding at the end of the list; this tells
us that this operation has the computational properties of a stack. OpenGL uses stacks to manage
this process as follows:
• as transformations are defined, they are multiplied into the current transformation in the order
noted in the list above, and
• when we want to make a mark in the list of transformations, we save the current version of the
transformation, make a copy, and apply all the subsequent transformations to the copy. We
can then throw away the copy and return to the original transformation and begin to work again
at that point. This is managed by maintaining a stack of transformations and, when we make a
copy, putting that copy on top of the stack. All transformations are applied to the one at the top
of the stack, and when we pop the stack, we return to the original context.
The functions we use to manage the stack are glPushMatrix() and glPopMatrix(),
respectively. Technically, they apply to whatever transformation stack is the current matrix mode,
and the glMatrixMode function with parameters GL_PROJECTION and GL_MODELVIEW sets
that mode. We only rarely want to use a stack of projection transformations (and in fact the stack
of projections can only hold two transformations) so we will almost always work with the stack of
modeling/viewing transformation.

To the example of the rabbit’s head, then, we would apply the following sequence of stack actions:
• push the stack
• apply the transformations to create the head, and define the head
• pop the stack
• push the stack
• apply the transformations that position the left eye relative to the head, and define the eye
• pop the stack
• push the stack
• apply the transformations that position the right eye relative to the head, and define the eye
• pop the stack
• push the stack
• apply the transformations that position the left ear relative to the head, and define the ear
• pop the stack
• push the stack
• apply the transformations that position the right ear relative to the head, and define the ear
• pop the stack
This set of operations will create the head as shown in the figure below. The code for this is given
in the set of code examples below. You should trace the sequence as noted above in the code
below and watch how the head is drawn. Note that if you were to want to put the rabbit’s head on
a body, you would treat this whole set of operations as a single unit inside another pair of
glPushMatrix() and glPopMatrix() functions.

Figure: the rabbit’s head
Display lists: complex OpenGL objects can be rather slow if their geometry must be re-computed every time they are displayed. To save time, you can compile a set of OpenGL instructions into a display list. A display list will be executed more efficiently because it will already contain the final geometry after all the transformations and other operations to create the object have been done. The only negative effect of display lists is that they cannot be changed once they are created, so you must be careful to put only infrequently-changed objects into the list. If you need to change the contents of a list, you must re-compile the list with the changes.

Some examples

Using standard objects to create more complex scenes

The example of transformation stacks is, in fact, a larger example — an example of using standard objects to define a larger object. In a program that defined a scene that needed rabbits, we would create the rabbit head with a function `rabbitHead()` that has the content of the code we used (and that is given below) and would apply whatever transformations would be needed to place a rabbit head properly on each rabbit body. The rabbits themselves could be part of a larger scene, and you could proceed in this way to create however complex a scene as you wish.

Modeling is, in general, a difficult task. There are applications that let you model things visually and then save the model in various file formats. It is possible to write a parser that will read a model file and display the model itself with OpenGL, but this is probably beyond the scope of a first computer graphics course. The closest we get to this is the molecular file reader functions that have been designed to support students from chemistry in the graphics course.

Creating display lists

Display lists are named by nonzero unsigned integer values (technically, GLuint values) and there are several tools available in OpenGL to manage these name values. We will assume in a first graphics course that you will not need many display lists and that you can manage a small number of list names yourself, but if you begin to use a number of display lists in a project, you should look into the `glGenLists`, `glIsList`, and `glDeleteLists` functions to help you manage the lists properly. Sample code is given in the code section below.

A word to the wise...

As we noted above, you must take a great deal of care with transformation order. It can be very difficult to look at an image that has been created with mis-ordered transformations and understand just how that erroneous example happened. In fact, there is a skill in what we might call “visual debugging” — looking at an image and seeing that it is not correct, and figuring out what errors might have caused the image as it is seen. It is important that anyone working with images become skilled in this kind of debugging. However, obviously you cannot tell than an image is wrong unless you know what a correct image should be, so you must know in general what you should be seeing. As an obvious example if you are doing scientific images, you must know the science well enough to know when an image makes no sense.

Another interesting issue with respect to the use of transformations is controlling the display to achieve the effects you want. As an example, consider wanting to control the rotation of an object interactively, so a user can apply rotations in any sequence. Two kinds of rotation are body-centered rotation, where the rotations are all made relative to the body of the object (for example, displaying an airplane as if it were being controlled by a pilot) or world-centered rotation, where all the rotations are based on the world environment (for example, displaying an airplane as if it were part of a world that is being manipulated). We could envision an example that could contain both rotations, if we think of an airplane model in an overall air controller environment; we might want to control an airplane’s motions or we might want to move around the entire display to look at the
airplanes from any vantage point. The processes of implementing each with OpenGL are shown below in the code examples.

**Code examples**

**Simple transformations:**
All the code examples use a standard set of axes, which are not included here, and the following definition of the simple square:

```c
void square (void)
{
    typedef GLfloat point[3];
    point v[8] = {{12.0, -1.0, -1.0},
                  {12.0, -1.0,  1.0},
                  {12.0,  1.0,  1.0},
                  {12.0,  1.0, -1.0}};

gBegin (GL_QUADS);
gVertex3fv(v[0]);
gVertex3fv(v[1]);
gVertex3fv(v[2]);
gVertex3fv(v[3]);
gEnd();
}
```

To display the simple rotations example, we use the following display function:

```c
void display( void )
{
    int i;
    float theta = 0.0;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    axes(10.0);
    for (i=0; i<8; i++) {
        glPushMatrix();
        glRotatef(theta, 0.0, 0.0, 1.0);
        if (i==0) glColor3f(1.0, 0.0, 0.0);
        else  glColor3f(1.0, 1.0, 1.0);
        square();
        theta += 45.0;
        glPopMatrix();
    }
    glutSwapBuffers();
}
```

To display the simple translations example, we use the following display function:

```c
void display( void )
{
    int i;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    axes(10.0);
    for (i=0; i<=12; i++) {
        glPushMatrix();
        glTranslatef(-2.0*(float)i, 0.0, 0.0);
        if (i==0) glColor3f(1.0, 0.0, 0.0);
        else  glColor3f(1.0, 1.0, 1.0);
        square();
        glPopMatrix();
    }
    glutSwapBuffers();
}
```
To display the simple scaling example, we use the following display function:

```c
void display( void )
{
    int i;
    float s;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    axes(10.0);
    for (i=0; i<6; i++) {
        glPushMatrix();
        s = (6.0-(float)i)/6.0;
        glScalef( s, s, s );
        if (i==0) glColor3f(1.0, 0.0, 0.0);
        else  glColor3f(1.0, 1.0, 1.0);
        square();
        glPopMatrix();
    }
    glutSwapBuffers();
}
```

Transformation stacks: The rabbit head example was created with the following display function. This function makes the stack operations more visible by using indentations; this is intended for emphasis in the example only and is not standard programming practice in graphics. Note that we have defined only very simple display properties (just a simple color) for each of the parts; we could in fact have defined a much more complex set of properties and have made the parts much more visually interesting. We could also have used a much more complex object than a simple gluSphere to make the parts much more structurally interesting. The sky’s the limit…

```c
void display( void )
{
    // Indentation level shows the level of the transformation stack
    // The basis for this example is the unit gluSphere; everything else
    // is done by explicit transformations

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    // model the head
    glColor3f(0.4, 0.4, 0.4); // dark gray head
    glScalef(3.0, 1.0, 1.0);
    myQuad = gluNewQuadric();
    gluSphere(myQuad, 1.0, 10, 10);
    glPopMatrix();
    glPushMatrix();
    // model the left eye
    glColor3f(0.0, 0.0, 0.0); // black eyes
    glTranslatef(1.0, -0.7, 0.7);
    glScalef(0.2, 0.2, 0.2);
    myQuad = gluNewQuadric();
    gluSphere(myQuad, 1.0, 10, 10);
    glPopMatrix();
    glPushMatrix();
    // model the right eye
    glTranslatef(1.0, 0.7, 0.7);
    glScalef(0.2, 0.2, 0.2);
    myQuad = gluNewQuadric();
    gluSphere(myQuad, 1.0, 10, 10);
    glPopMatrix();
    glPushMatrix();
    // model the left ear
    glColor3f(1.0, 0.6, 0.6); // pink ears
    glTranslatef(-1.0, -1.0, 1.0);
    ```
Display lists: display lists are relatively easy to create. First, choose an unsigned integer (often you will just use small integer constants, such as 1, 2, ...) to serve as the name of your list. Then before you create the geometry for your list, call the function glNewList. Code whatever geometry you want into the list, and at the end, call the function glEndList. Everything between the new list and the end list functions will be executed whenever you call glCallList with a valid list name as parameter. Because display lists are often defined only once, it is common to create them in the init() function or in a function called from within init(). Some sample code is given below, with most of the content taken out and only the display list operations left.

```c
void Build_lists(void)
{
    glNewList(1, GL_COMPILE);
    glBegin(GL_TRIANGLE_STRIP);
        glNormal3fv(...); glVertex3fv(...);
    ...
    glEnd();
    glEndList();
}

static void Init(void)
{
    ...
    Build_lists();
    ...
}

void Display(void)
{
    ...
    glCallList(1);
    ...
}
```

You will note that the display list was created in GL_COMPILE mode, and it was not executed (the object was not displayed) until the list was called. It is also possible to have the list displayed as it is created if you create the list in GL_COMPILE_AND_EXECUTE mode.

Rotations: body-centered vs world-centered

If you want to rotate an object around one of its own axes, you need only apply the rotation at the very end of the modeling process. That is, you model the object in its context, and at the last
moment before you display the object, you apply the rotation as follows. This is extremely simple,
but you are not doing anything to the object except this simple rotation.

```c
void display( void )
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    rotate(); // generic
drawObject();
    glutSwapBuffers();
}
```

However, if you want to use world-centered rotations for your object, you are faced with a much
more interesting situation. You need to save your current environment, go back to the identity and
start over, start with the rotation you want, multiply back in the saved transformation, draw the
object, and finally pop off the new transformation and restore the original transformation.

```c
static GLfloat saveState[16] = {1.0,0.0,0.0,0.0,
    0.0,1.0,0.0,0.0,
    0.0,0.0,1.0,0.0,
    0.0,0.0,0.0,1.0},
    viewProj[16]; // hold transforms

void display( void )
{
    int i

    // The GL_MODELVIEW_MATRIX starts with the viewing transformation, so we
    // save that, then construct the modeling transformation, then take *that*
    // off, then put them back together.
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Save the original viewing projection in the viewProj array
    glPushMatrix();
    glGetFloatv( GL_MODELVIEW_MATRIX, viewProj );

    // Put identity onto the modelview stack to start the viewing transform
    // and apply a generic rotation function whose content isn't important
    glLoadIdentity();
    rotate();
    // NOW apply the rest of the modeling transformation by POST-multiplying
    // by the saved matrix, then save that and put the identity back on the
    // stack.  *whew*
    glMultMatrixf( saveState );
    glGetFloatv( GL_MODELVIEW_MATRIX, saveState );
    glLoadIdentity();

    // Finally rebuild the overall modelview matrix by multiplying by the
    // viewing transformation, then the modeling transformation, then
    glMultMatrixf( viewProj );
    glMultMatrixf( saveState );
    drawObject();

    // Put back the original GL_MODELVIEW_MATRIX by popping off the new
    // work.
    glPopMatrix();
    glutSwapBuffers();
}
```
Science applications

In many areas, there are standard objects that make up parts of displays in science. An excellent example of this is in chemistry, where there are standard ways to display atoms in molecules, but there are probably many others. Students are urged to look for standard displays in their science field and see how they can be created in OpenGL so they can be used in student-generated displays in the sciences.