CMSC 425: Lecture 5
Drawing in OpenGL: Transformations
Thursday, Feb 7, 2013

Reading: See any standard reference on OpenGL or GLUT.

Complex Drawing in OpenGL: So far we have discussed how to draw simple 2-dimensional objects using OpenGL. Suppose that we want to draw more complex scenes. For example, we want to draw objects that move and rotate or to change the projection. We could do this by computing (ourselves) the coordinates of the transformed vertices. However, this would be inconvenient for us. It would also be inefficient. OpenGL provides methods for downloading large geometric specifications directly to the GPU. However, if the coordinates of these object were changed with each display cycle, this would negate the benefit of loading them just once.

For this reason, OpenGL provides tools to handle transformations and to apply these transformations to the objects we wish to draw.

Affine Transformations: Linear and affine transformations are central to computer graphics. Recall from your linear algebra class that a *linear transformation* is a mapping in a vector space that preserves linear combinations. Such transformations include rotations, scalings, shearings (which stretch rectangles into parallelograms), and combinations thereof.

Linear transformations are limited in that they cannot represent translation. The origin is always mapped to the origin. If you combine linear transformations with translation, you get a broader class of transformations called *affine transformations*. Affine transformations have a number of nice properties:

Preserves linearity: Lines are mapped to lines

Preserves parallelism: Parallel lines remain parallel

Preserves affine combinations: This is harder to explain, but as an example, if p and q are two points and m is their midpoint, and T is an affine transformation, then the midpoint of $T(p)$ and $T(q)$ is $T(m)$.

Points in affine space are represented in *homogeneous coordinates*. This is done by adding an extra coordinate to the end, which is set to 1. For example, in two dimensional space, the point with coordinates (x, y) would be expressed in homogeneous coordinates as the 3-element vector $[x, y, 1]$. Given in this form, any affine transformation can be expressed as the product of such a vector and a 3×3 matrix. In general, a point in standard d-dimensional space can be expressed as a $(d + 1)$-dimensional homogeneous vector, by appending a 1 to the end. Affine transformations can be expressed as the multiplication of this vector times a $(d + 1) \times (d + 1)$ matrix.

How are Affine Transformations Used? Affine transformations have many uses in computer graphics and game programming in general.

Moving Objects: As needed in animations, physical simulations, etc.

Change of Coordinates: This is used when objects that are stored relative to one coordinate frame are to be accessed in a different coordinate frame. One important case of this is that of mapping objects stored in a standard coordinate system to a coordinate system that is associated with the camera (or viewer).

Parallel Camera Projection: Projection is the task of mapping three-dimensional objects onto a two-dimensional image plane. The most common type of projections are called *perspective projections*, where the image rays converge at a fixed focal point point in space. These are *not* affine transformations. However, if the image rays are parallel to each other (they converge at a focal point at infinity), the resulting transformation is affine. An important special case are *orthogonal projections*, where the projection direction is parallel to one of the coordinate axes.

Applying Textures: This is useful when textures are mapped from an image onto an object surface as part of texture mapping process.

OpenGL and Transformations: OpenGL has a very particular model for how transformations are performed. Recall that when drawing, it was convenient for us to first define the drawing attributes (such as color) and then draw a number of objects using that attribute. OpenGL uses much the same model with transformations. You specify a transformation *first*, and then this transformation is automatically applied to every object that is drawn *afterwards*, until the transformation is set again. It is important to keep this in mind, because it implies that you must always set the transformation prior to issuing drawing commands.

Because transformations are used for different purposes, OpenGL maintains three sets of matrices for performing various transformation operations. These are:

Modelview matrix: (GL_MODELVIEW) Used for transforming objects in the scene and for changing the coordinates into a form that is easier for OpenGL to deal with. (It is used for the first two tasks above, moving objects and converting between coordinate systems).

Projection matrix: (GL_PROJECTION) Handles both parallel and perspective projections. (Used for the third task above.)

Texture matrix: (GL_TEXTURE) This is used in specifying how textures are mapped onto objects. (Used for the last task above.)

We will discuss the texture matrix later in the semester, when we talk about texture mapping. There is one more transformation that is not handled by these matrices. This is the transformation that maps the viewport to the display. It is set by glViewport().

Understanding how OpenGL maintains and manipulates transformations through these matrices is central to understanding how OpenGL and other modern immediate-mode rendering systems (such as DirectX) work.

Matrix Stacks: For each matrix type, OpenGL maintains a *stack* of matrices. The *current matrix* is the one on the top of the stack. It is the matrix that is being applied at any given time. The stack mechanism allows you to save the current matrix (by pushing the stack down) and restoring it later (by popping the stack). We will discuss the entire process of implementing affine and projection transformations later in the semester. For now, we'll give just basic information on OpenGL's approach to handling matrices and transformations.

OpenGL has a number of commands for handling matrices. In order to know which matrix (Modelview, Projection, or Texture) to which an operation applies, you can set the current *matrix mode*. This is done with the following command

glMatrixMode(⟨*mode*⟩);

where ⟨*mode*⟩ is either GL_MODELVIEW, GL_PROJECTION, or GL_TEXTURE. The default mode is GL_MODELVIEW.

GL_MODELVIEW is by far the most common mode, the convention in OpenGL programs is to assume that you are always in this mode. If you want to modify the mode for some reason, you first change the mode to the desired mode (GL_PROJECTION or GL_TEXTURE), perform whatever operations you want, and then immediately change the mode back to GL_MODELVIEW.

Once the matrix mode is set, you can perform various operations to the stack. OpenGL has an unintuitive way of handling the stack. Note that most operations below (except glPushMatrix()) alter the contents of the matrix at the top of the stack.

glLoadIdentity(): Sets the current matrix to the identity matrix.

glLoadMatrix*(M): Loads (copies) a given matrix over the current matrix. (The '*' can be either 'f' or 'd' depending on whether the elements of M are GLfloat or GLdouble, respectively.)

glMultMatrix*(M): Post-multiplies the current matrix by a given matrix and replaces the current matrix with this result. Thus, if C is the current matrix on top of the stack, it will be replaced with the matrix product $C \cdot M$. (As above, the '*' can be either 'f' or 'd' depending on M.)

glPushMatrix(): Pushes a copy of the current matrix on top the stack. (Thus the stack now has two copies of the top matrix.)

glPopMatrix(): Pops the current matrix off the stack.

Warning: OpenGL assumes that all matrices are 4×4 homogeneous matrices, stored in column-major order. That is, a matrix is presented as an array of 16 values, where the first four values give column 0 (for x), then column 1 (for y), then column 2 (for z), and finally column 3 (for the homogeneous coordinate, usually called w). For example, given a matrix M and vector v, OpenGL assumes the following representation:

$$M \cdot v = \begin{pmatrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[9] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{pmatrix} \begin{pmatrix} v[0] \\ v[1] \\ v[2] \\ v[3] \end{pmatrix}$$

An example is shown in Fig. 1. We will discuss how matrices like M are presented to OpenGL later in the semester. There are a number of other matrix operations, which we will also discuss later.

Automatic Evaluation and the Transformation Pipeline: Now that we have described the matrix stack, the next question is how do we apply the matrix to some point that we want to transform? Understanding the answer is critical to understanding how OpenGL (and actually display processors) work. The answer is that it happens *automatically*. In particular, *every* vertex (and hence virtually every geometric object that is drawn) is passed through a series of matrices, as shown in Fig. 2. This may

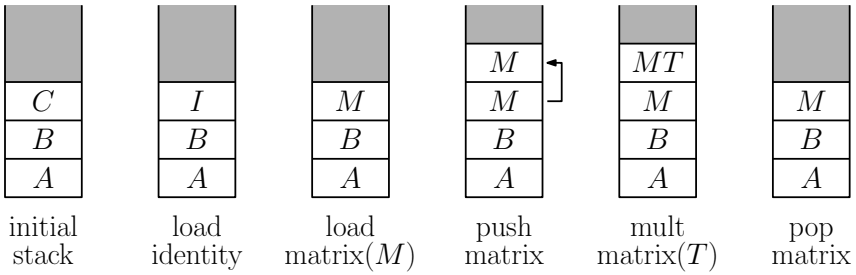

Fig. 1: Matrix stack operations.

seem rather inflexible, but it is because of the simple uniformity of sending every vertex through this transformation sequence that makes it possible for GPUs to run so fast. As mentioned above, these transformations behave much like drawing attributes—you set them, do some drawing, alter them, do more drawing, etc.

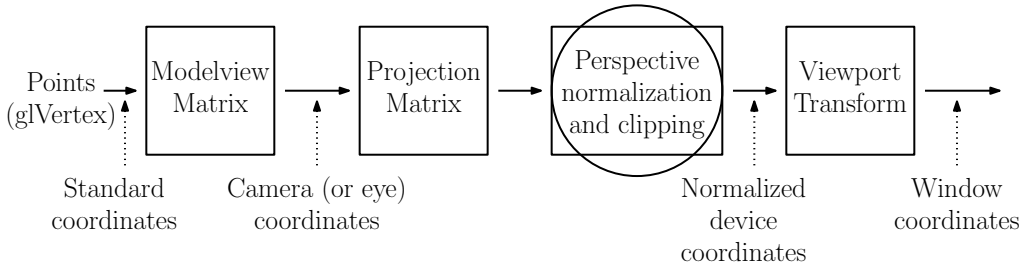

Fig. 2: Transformation pipeline.

A second important thing to understand is that OpenGL's transformations do not alter the state of the objects you are drawing. They simply modify things before they get drawn. For example, suppose that you draw a unit square ($U = [0, 1] \times [0, 1]$) and pass it through a matrix that scales it by a factor of 5. The square U itself has not changed; it is still a unit square. If you wanted to change the actual representation of U to be a 5×5 square, then you need to perform your own modification of U's representation.

You might ask, "what if I do *not* want the current transformation to be applied to some object?" The answer is, "tough luck." There are no exceptions to this rule (other than commands that act directly on the viewport). If you do not want a transformation to be applied, then to achieve this, you load an identity matrix on the top of the transformation stack, then do your (untransformed) drawing, and finally pop the stack.

Example: Rotating a Rectangle (first attempt): The Modelview matrix is useful for applying transformations to objects, which would otherwise require you to perform your own linear algebra. Suppose that rather than drawing a rectangle that is aligned with the coordinate axes, you want to draw a rectangle that is rotated by 20 degrees (counterclockwise) and centered at some point (x, y). The desired result is shown in Fig. 3. Of course, as mentioned above, you could compute the rotated coordinates of the vertices yourself (using the appropriate trigonometric functions), but OpenGL provides a way of doing this transformation more easily.

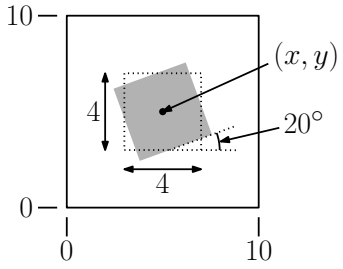

Fig. 3: Desired drawing. (Rotated rectangle is shaded).

Suppose that we are drawing within the square, $0 \leq x, y \leq 10$, and we have a 4×4 sized rectangle to be drawn centered at location (x, y). We could draw an unrotated rectangle with the following command:

```
glRectf(x - 2, y - 2, x + 2, y + 2);
```

Formally, the arguments should be of type GLfloat (2.0f rather than 2), but we will let the compiler cast the integer constants to floating point values for us.

Now let us draw a rotated rectangle. Let us assume that the matrix mode is GL_MODELVIEW (this is the default). Generally, there will be some existing transformation (call it M) currently present in the Modelview matrix. This usually represents some more global transformation, which is to be applied on top of our rotation. For this reason, we will compose our rotation transformation with this existing transformation.

Because the OpenGL rotation function destroys the contents of the Modelview matrix, we will begin by saving it, by using the command glPushMatrix(). Saving the Modelview matrix in this manner is not always required, but it is considered good form. Then we will compose the current matrix M with an appropriate rotation matrix R. Then we draw the rectangle (in upright form). Since all points are transformed by the Modelview matrix prior to projection, this will have the effect of rotating our rectangle. Finally, we will pop off this matrix (so future drawing is not rotated).

To perform the rotation, we will use the command glRotatef(ang, x, y, z). All arguments are GLfloat's. (Or, recalling OpenGL's naming convention, we could use glRotated() which takes GLdouble arguments.) This command constructs a matrix that performs a rotation in 3-dimensional space counterclockwise by angle *ang* degrees, about the vector (x, y, z). It then *composes* (or multiplies) this matrix with the current Modelview matrix. In our case the angle is 20 degrees. To achieve a rotation in the (x, y) plane the vector of rotation would be the z-unit vector, $(0, 0, 1)$. Here is how the code might look (but beware, this conceals a subtle error).

Drawing an Rotated Rectangle (First Attempt)

```
glPushMatrix();                 // save the current matrix
  glRotatef(20, 0, 0, 1);       // rotate by 20 degrees CCW
  glRectf(x-2, y-2, x+2, y+2);  // draw the rectangle
glPopMatrix();                  // restore the old matrix
```

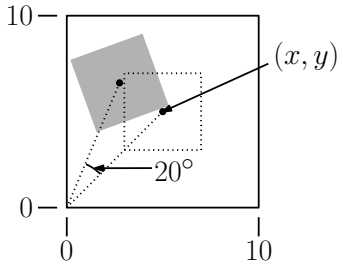

Fig. 4: The actual drawing produced by the previous example. (Rotated rectangle is shaded).

The order of the rotation relative to the drawing command may seem confusing at first. You might think, "Shouldn't we draw the rectangle first and then rotate it?". The key is to remember that whenever you draw (using glRectf() or glBegin()...glEnd()), the points are automatically transformed using the current Modelview matrix. So, in order to do the rotation, we must first modify the Modelview matrix, then draw the rectangle. The rectangle will be automatically transformed into its rotated state. Popping the matrix at the end is important, otherwise future drawing requests would also be subject to the same rotation.

Unfortunately, something is wrong with this example given above. What is it? The answer is that the rotation is performed *about the origin* of the coordinate system, not about the center of the rectangle as we want.

Correct Rotation (Through the Looking Glass): Fortunately, there is an easy fix. Conceptually, we will draw the rectangle centered at the origin, then rotate it by 20 degrees, and finally *translate* (or move) it by the vector (x, y). To do this, we will need to use the command glTranslatef(x, y, z) (see Fig. 5). All three arguments are GLfloat's. (And there is version with GLdouble arguments.) This command creates a matrix which performs a translation by the vector (x, y, z), and then composes (or multiplies) it with the current matrix. Recalling that all 2-dimensional graphics occurs in the $z = 0$ plane, the desired translation vector is $(x, y, 0)$.

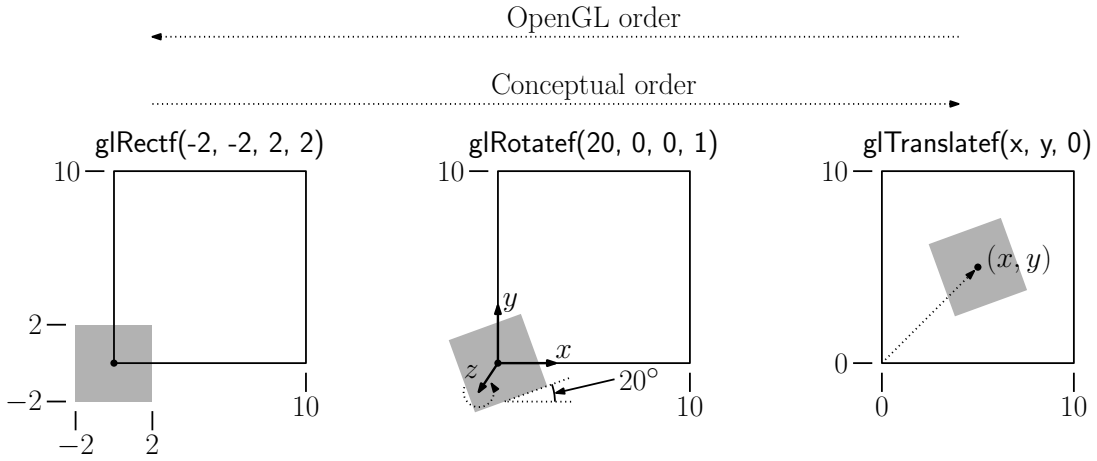

Fig. 5: Final drawing/transformation sequence.

So the conceptual order is (1) draw, (2) rotate, (3) translate. But remember that you need to set up the

transformation matrix *before* you do any drawing. That is, if $\vec{v}$ represents a vertex of the rectangle, and R is the rotation matrix and T is the translation matrix, and M is the current Modelview matrix, then we want to compute the product

$$M(T(R(\vec{v}))) = M \cdot T \cdot R \cdot \vec{v}.$$

Since M is on the top of the stack, we need to first apply translation (T) to M, and then apply rotation (R) to the result, and then do the drawing ($\vec{v}$). Note that the order of application is the exact *reverse* from the conceptual order. This may seems confusing (and it is), so remember the following rule.

Drawing/Transformation Order in OpenGL's

First, conceptualize your intent by drawing about the origin and then applying the appropriate transformations to map your object to its desired location. Then implement this by applying transformations in *reverse order*, and do your drawing. It is always a good idea to enclose everything in a push-matrix and pop-matrix pair.

Although this may seem backwards, it is the way in which almost all object transformations are performed in OpenGL:

(1) Push the matrix stack,

(2) Apply (i.e., multiply) all the desired transformation matrices with the current matrix, but *in the reverse order* from which you would like them to be applied to your object,

(3) Draw your object (the transformations will be applied automatically), and

(4) Pop the matrix stack.

The final and correct fragment of code for the rotation is shown in the code block below.

Drawing an Rotated Rectangle (Correct)

```
glPushMatrix();                 // save the current matrix (M)
  glTranslatef(x, y, 0);        // apply translation (T)
  glRotatef(20, 0, 0, 1);       // apply rotation (R)
  glRectf(-2, -2, 2, 2);        // draw rectangle at the origin
glPopMatrix();                  // restore the old matrix (M)
```

Projection Revisited: Last time we discussed the use of gluOrtho2D() for doing simple 2-dimensional projection. This call does not really do any projection. Rather, it computes the desired projection transformation and multiplies it times whatever is on top of the current matrix stack. So, to use this we need to do a few things. First, set the matrix mode to GL_PROJECTION, load an identity matrix (just for safety), and the call gluOrtho2D(). Because of the convention that the Modelview mode is the default, we will set the mode back when we are done.

If you only set the projection once, then initializing the matrix to the identity is typically redundant (since this is the default value), but it is a good idea to make a habit of loading the identity for safety. If the projection does not change throughout the execution of our program, and so we include this code in our initializations. It might be put in the reshape callback if reshaping the window alters the projection.

```
glMatrixMode(GL_PROJECTION);            // set projection matrix
glLoadIdentity();                       // initialize to identity
gluOrtho2D(left, right, bottom top);    // set the drawing area
glMatrixMode(GL_MODELVIEW);             // restore Modelview mode
```

How is it done (Optional): How does gluOrtho2D() and glViewport() set up the desired transformation from the idealized drawing window to the viewport? Well, actually OpenGL does this in two steps, first mapping from the window to canonical 2×2 window centered about the origin, and then mapping this canonical window to the viewport. The reason for this intermediate mapping is that the clipping algorithms are designed to operate on this fixed sized window (recall the figure given earlier). The intermediate coordinates are often called *normalized device coordinates*.

As an exercise in deriving linear transformations, let us consider doing this all in one shot. Let W denote the idealized drawing window and let V denote the viewport. Let w_r, w_l, w_b, and w_t denote the left, right, bottom and top of the window. Define v_r, v_l, v_b, and v_t similarly for the viewport. We wish to derive a linear transformation that maps a point (x, y) in window coordinates to a point (x', y') in viewport coordinates. See Fig. 6.

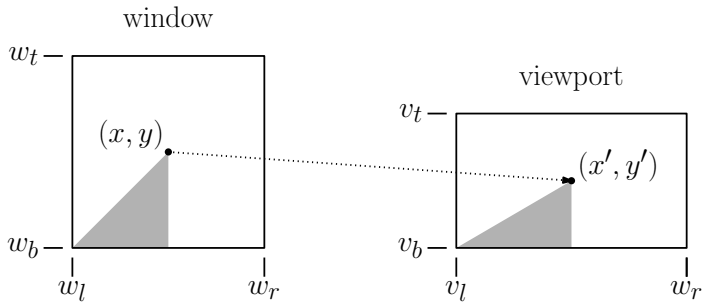

Fig. 6: Window to Viewport transformation.

Let $f(x, y)$ denote the desired transformation. Since the function is linear, and it operates on x and y independently, we have

$$(x', y') = f(x, y) = (s_x x + t_x, s_y y + t_y),$$

where s_x, t_x, s_y and t_y, depend on the window and viewport coordinates. Let's derive what s_x and t_x are using simultaneous equations. We know that the x-coordinates for the left and right sides of the window (w_l and w_r) should map to the left and right sides of the viewport (v_l and v_r). Thus we have

$$s_x w_l + t_x = v_l \qquad \text{and} \qquad s_x w_r + t_x = v_r.$$

We can solve these equations simultaneously. By subtracting them to eliminate t_x we have

$$s_x = \frac{v_r - v_l}{w_r - w_l}.$$

Plugging this back into to either equation and solving for t_x we have

$$t_x \;=\; v_l - s_x w_l \;=\; v_l - \frac{v_r - v_l}{w_r - w_l} w_l \;=\; \frac{v_l w_r - v_r w_l}{w_r - w_l}.$$

A similar derivation for s_y and t_y yields

$$s_y = \frac{v_t - v_b}{w_t - w_b} \qquad t_y = \frac{v_b w_t - v_t w_b}{w_t - w_b}.$$

These four formulas give the desired final transformation.

$$f(x,y) = \left(\frac{(v_r - v_l)x + (v_l w_r - v_r w_l)}{w_r - w_l}, \frac{(v_t - v_b)y + (v_b w_t - v_t w_b)}{w_t - w_b} \right).$$

This can be expressed in matrix form as

$$\begin{pmatrix} \frac{v_r - v_l}{w_r - w_l} & 0 & \frac{v_l w_r - v_r w_l}{w_r - w_l} \\ 0 & \frac{v_t - v_b}{w_t - w_b} & \frac{v_b w_t - v_t w_b}{w_t - w_b} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix},$$

which is essentially what OpenGL stores internally.