
Chapter 2

Making Them Move

In the previous chapter, we saw how to draw basic shapes using the OpenGL graphics library. But we want to be able to do more than just draw shapes: we want to be able to move them around to design and compose our 2D scene. We want to be able to scale the objects to different sizes and orient them differently. The functions used for modifying objects such as translation, rotation, and scaling are called *geometric transformations*.

Why do we care about transformations? Usually we define our shapes in a coordinate system convenient to us. Using transformation equations enables us to easily modify object locations in the world coordinate system. In addition, if we transform our objects and play back the CG images fast enough, our eyes will be tricked to believe we are seeing motion. This principle is used extensively in animation, and we shall look into it in detail in Chapter 7. When we study 3D models, we will also see how we can use transformations to define hierarchical objects and to “transform” the camera position.

This chapter introduces the basic 2D transformations used in CG: translation, rotation, and scaling. These transformations are essential ingredients of all graphics applications.

In this chapter you will learn the following concepts:

- Vectors and matrices
- 2D Transformations: translation, scaling, and rotation
- How to use OpenGL to transform objects
- Composition of transforms

2.1 Vectors and Matrices

Before we jump into the fairly mathematical discussion of transformations, let us first brush up on the basics of vector and matrix math. This math will form the basis for the transformation equations we shall see later in this chapter. We discuss the math involved as applied to a 2D space. The principles are easily extended to 3D by simply adding a third axes, namely, the z -axis.

Vectors

A vector is a quantity that has both direction and length. In CG, a vector represents a directed line segment with a start point (its tail) and an end point (the head, shown typically as an arrow pointed along the direction of the vector). The length of the line is the length of the vector.

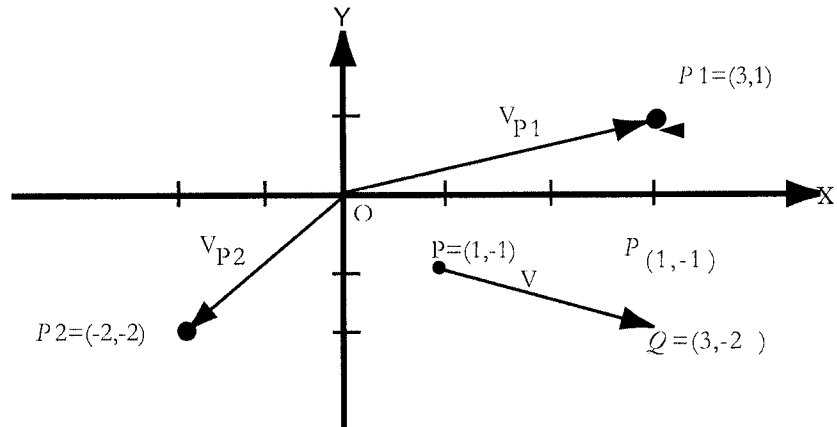


Fig. 2.1: A 2D Vector/Point: A directional line

A 2D vector that has a length of x units along the x -axis and y units along the y -axis is denoted as $\begin{bmatrix} x \\ y \end{bmatrix}$

It is valuable to think of a vector as a displacement from one point to another. Consider two points $P(1,-1)$ and $Q(3,-2)$, as shown in Figure 2.1. The displacement from P to Q is the vector $V = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$, calculated by subtracting the

coordinates of the points individually. What this means is that to get from P to Q , we shift right along the x -axis by two units and down the y -axis by one unit. Interestingly, any point $P1$ with coordinates (x,y) corresponds to the vector V_{P1} , with its head at (x,y) and tail at $(0,0)$ as shown in Figure 2.1. That is, there is a one-to-one correspondence between a vector, with its tail at the origin, and a

point on the plane. This means that we can also represent the point $P(x,y)$ by the vector $\begin{bmatrix} x \\ y \end{bmatrix}$

Often, the math of transformation equations uses the vector representation of points in this manner, so do not let this usage confuse you.

Operations with Vectors

Vectors support some fundamental operations: addition, subtraction, and multiplication with a real number.

Vectors can be added by performing componentwise addition. If V_1 is the vector (x_1, y_1) and V_2 is the vector (x_2, y_2) then $V_1 + V_2$ is

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} x_1 + x_2 \\ y_1 + y_2 \end{bmatrix}$$

Conceptually, adding two vectors results in a third vector which is the addition of one displacement with another.

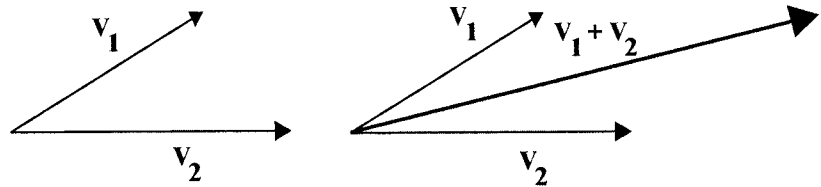


Fig. 2.2: Adding two vectors

Multiplying a vector by a number s results in a vector whose length has been scaled by s . For this reason, the number s is also referred to as a scalar. If s is negative, then this results in a vector whose direction is flipped as well.

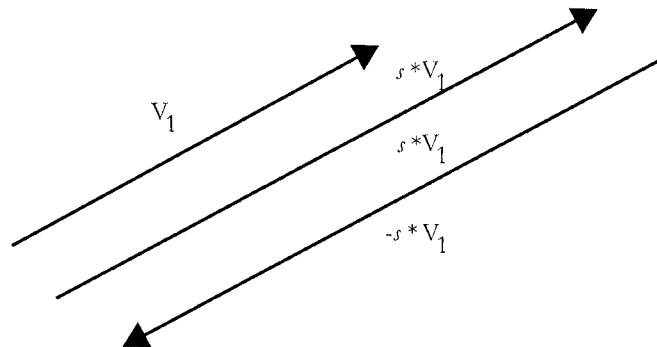


Fig. 2.3: Multiplying a vector with a scalar

Mathematically, the scaled vector $sV_1 = \begin{bmatrix} s \cdot x_1 \\ s \cdot y_1 \end{bmatrix}$

Subtraction follows easily as the addition of a vector that has been flipped: that is $V_1 - V_2 = V_1 + (-V_2)$.

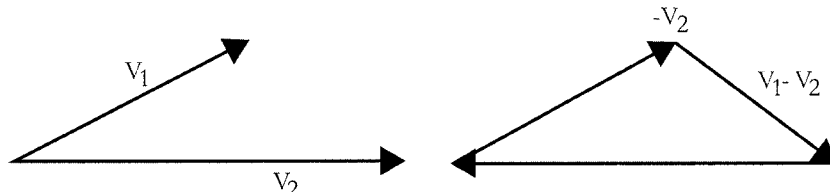


Fig. 2.4: Subtracting two vectors

The Magnitude of a Vector

The length of a vector $V = \begin{bmatrix} x \\ y \end{bmatrix}$

is also referred to as its *magnitude*.

The magnitude of a vector is the distance from the tail to the head of the vector. It is represented as $|V|$ and is equal to $\sqrt{x^2 + y^2}$.

It is very useful to scale a vector so that the resultant vector has a length of 1, with the same direction as the original vector. This process is called *normalizing* a vector. The resultant vector is called a *unit vector*. To normalize a vector V , we simply scale it by the value $1/|V|$. The resultant unit vector is represented as $V = V/|V|$.

Interestingly, a unit vector along the x -axis is quite simply the vector $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$

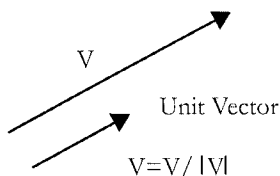


Fig. 2.5: A Unit Vector

and a unit vector along the the y axis is $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$

The Dot Product

The dot product of two vectors is a scalar quantity. The dot product is used to solve a number of important geometric problems in graphics. The dot product is written as $V_1 \cdot V_2$ and is calculated as

$$V_1 \cdot V_2 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \cdot \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = x_1 * x_2 + y_1 * y_2$$

You can find many of these vector functions coded in a utility include file provided by us called *utils.h*.

Matrices

A matrix is an array of numbers. The number of rows (m) and columns (n) in the array defines the cardinality of the matrix ($m \times n$). In reality, a vector is simply a matrix with one column (or a one-dimensional matrix). We saw how displays are just a matrix of pixels. A frame buffer is a matrix of pixel values for each pixel on the display.

Matrices can be added component wise, provided they have the same cardinality:

$$\begin{bmatrix} a11 & a12 \\ a21 & a22 \end{bmatrix} + \begin{bmatrix} b11 & b12 \\ b21 & b22 \end{bmatrix} = \begin{bmatrix} a11+b11 & a12+b12 \\ a21+b21 & a22+b22 \end{bmatrix}$$

Multiplication of a matrix by a scalar is simply the multiplication of its components by this scalar:

$$s * \begin{bmatrix} a11 & a12 \\ a21 & a22 \end{bmatrix} = \begin{bmatrix} s*a11 & s*a12 \\ s*a21 & s*a22 \end{bmatrix}$$

Two matrices can be multiplied if and only if the number of columns of the first matrix ($m \times n$) is equal to the number of rows of the second ($n \times p$). The result is calculated by applying the dot product of each row of the first matrix with each column of the second. The resultant matrix has a cardinality of ($m \times p$).

$$\begin{bmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \end{bmatrix} * \begin{bmatrix} b11 & b12 \\ b21 & b22 \\ b31 & b32 \end{bmatrix} =$$

$$\begin{bmatrix} a11*b11 + a12*b21 + a13*b31 & a11*b12 + a12*b22 + a13*b32 \\ a21*b11 + a22*b21 + a23*b31 & a21*b12 + a22*b22 + a23*b32 \end{bmatrix}$$

We can multiply a vector by a matrix if and only if it satisfies the rule above.

$$\begin{bmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{bmatrix} * \begin{bmatrix} v1 \\ v2 \\ v3 \end{bmatrix} = \begin{bmatrix} a11*v1 + a12*v2 + a13*v3 \\ a21*v1 + a22*v2 + a23*v3 \\ a31*v1 + a32*v2 + a33*v3 \end{bmatrix}$$

An *identity matrix* is a square matrix (an equal number of rows and columns) with all zeroes, except for 1s in its diagonal.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Multiplication of a vector/matrix by an identity matrix has no effect. Prove this by multiplying a vector with the appropriate identity matrix. Refer to LENG93 for more details on vectors and matrices.

Whew! After that fairly exhaustive review of matrices and vectors, we are ready to get into the thick of transformations.

2.2 2D Object Transformations

The functions used for modifying the size, location, and orientation of objects or of the camera in a CG world are called *geometric transformations*. Transformations are applied within a particular space domain, be it an object space or the camera space or a texture space. We shall mainly be discussing transformations applied in the object space, also called *object transformations* or *model transformations*. The mathematics for transformations in other spaces remains the same.

Object Space

Usually, objects are defined in a default coordinate system convenient to the modeler. This coordinate system is called the object coordinate system or object space.

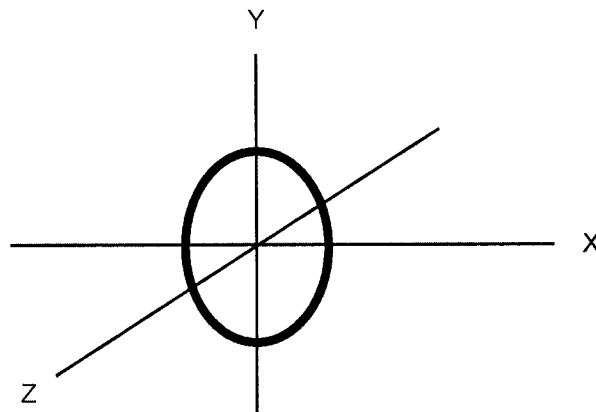


Fig.2.5: Object space

The object coordinate system can be transformed, in order to transform the object within the world coordinate system. An object transformation sets the state of the object space. Internally, this state is represented as a matrix. All the objects drawn on the screen after this state is set are drawn with the new transformations applied. Transformations make it convenient to move objects within the world, without actually having to model the object again!

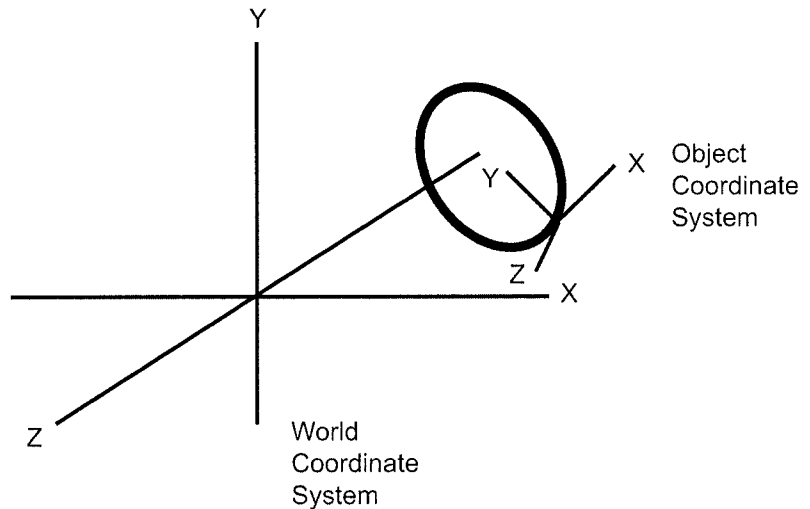


Fig.2.6: Object space is transformed within the world coordinate system.

Let us look into the three kinds of transformations most commonly used in CG: translation, rotation, and scaling.

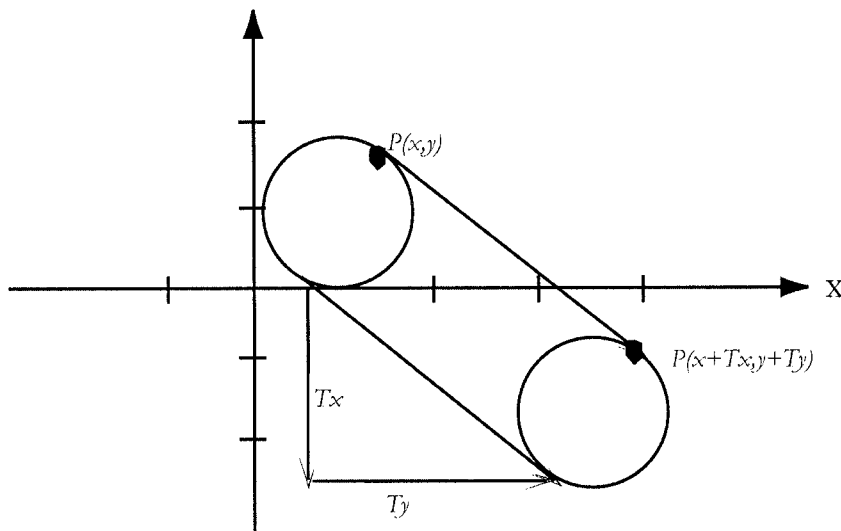


Fig.2.7: Translation along the x- and y-axes

Translation

Translation is the transform applied when we wish to move an object. In a 2D world, we can move the object from left to right (translate along the x -axis) or up and down (translate along the y -axis) as shown in Fig.2.7. The abbreviations for these translations are T_x and T_y .

Consider a circular shape, P , as shown in Figure. If we wish to move P by a distance of (tx, ty) then all points $P(x, y)$ on this shape will move to a new location $P'(x, y) = P(x+tx, y+ty)$.

If we define the vector for a given points $P(x, y)$ as:

$$\mathbf{P} = \begin{bmatrix} x \\ y \end{bmatrix} \text{ and the translation vector as } \mathbf{T} = \begin{bmatrix} T_x \\ T_y \end{bmatrix} \text{ then the resultant}$$

$$\text{transformed point } P' \text{ is the vector represented as } \mathbf{P}' = \begin{bmatrix} x + T_x \\ y + T_y \end{bmatrix}$$

Or more precisely

$$\mathbf{P}' = \mathbf{T} + \mathbf{P}$$

Mathematically, we can translate an object by applying this equation to every point on the object. Usually, we can get away with just applying the translation to the vertices of the shape and then recalculating the translated shape.

Let us visually display this concept with an example of a bouncing ball. From the last chapter, you may recall how to draw a circle:

```
#define PI 3.1415926535898
// cos and sin functions require angles in radians
// recall that 2PI radians = 360 degrees, a full circle

GLint circle_points = 100;
void MyCircle2f(GLfloat centerx, GLfloat centery, GLfloat radius){
    GLint i;
    GLdouble angle;
    glBegin(GL_POLYGON);
    for (i = 0; i < circle_points; i++) {
        angle = 2*PI*i/circle_points; // angle in radians
        glVertex2f(centerx+radius*cos(angle),
                   centery +radius*sin(angle));
    }
    glEnd();
}
```

In *Example2_1*, we use this function to draw a brown ball centered at the origin with the radius of the ball set to be 15 units in our world coordinate system:


```

GLfloat RadiusOfBall = 15.;
// Draw the ball, centered at the origin
void draw_ball() {
    glColor3f(0.6,0.3,0.);
    MyCircle2f(0.,0.,RadiusOfBall);
}

```

We want to bounce this ball up and down in our display window. That is, we wish to translate it along the y -axis. The floating-point routine for performing translations in OpenGL is

```
glTranslatef(Tx, Ty, Tz)
```

It accepts three arguments for translation along the x -, y -, and z -axes respectively. The z -axis is used for 3D worlds, and we can ignore it for now by setting the Tz value to be 0.

Before we apply a new translation, we need to set the transformation state in the object space to be an identity matrix (i.e., no transformation is being applied). The command

```
glLoadIdentity ()
```

clears out the current transformation state with the identity matrix. We do this because we do not wish to reuse old transformation states.

Then, we constantly add (or subtract) to the ty component along the y -axis, and draw the ball in this newly transformed position. The snippet of code to translate an object in a window with the maximum extent of the world coordinates set to (160,120) is shown below. A $ydir$ variable is used to define whether the ball is bouncing up or down (and hence whether we should increment or decrement ty).

```

// 160 is max X value in our world
// Define X position of the ball to be at center of window
xpos = 80.;
// 120 is max Y value in our world
// set Y position to increment 1.5 times the direction of the bounce
ypos = ypos+ydir *1.5;
// If ball touches the top, change direction of ball downwards
if (ypos == 120-RadiusOfBall)
    ydir = -1;
// If ball touches the bottom, change direction of ball downwards
else if (ypos <RadiusOfBall)
    ydir = 1;

```

```
//reset transformation state
glLoadIdentity();
// apply the desired translation
glTranslatef(xpos,ypos, 0.);
```

Camera (or viewing) and object transformations are combined into a single matrix in OpenGL, called the model-view matrix (GL_MODELVIEW). The command

```
glMatrixMode(GL_MODELVIEW)
```

specifies the space where the transformations are being applied. The mode is normally set in the reshape function where we also set the clipping area of the window.

That is, whenever the window is reshaped, resized, or first drawn, we specify any further transformations to be applied in the object space.

If you try to view this animation, we shall see motion that is jerky, and you may even see incomplete images being drawn on the screen. To make this bouncing motion appear smooth, we make use of a concept called *double buffering*.

Double Buffering

Double buffering provides two frame buffers for use in drawing. One buffer, called the foreground buffer is used to display on the screen. The other buffer, called the background buffer, is used to draw into. When the drawing is complete, the two buffers are swapped so that the one that was being viewed is now being used for drawing and vice versa. The swap is almost instantaneous. Since the image is already drawn when we display it on screen, it makes the resulting animation look smooth, and we don't see incomplete images. The only change in the required to activate double buffering is to specify the display mode to be **GLUT_DOUBLE**.

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
```

This call is made in the main function of *Example2_1*. By default, in double buffer mode, OpenGL renders all drawing commands into the background buffer. A call to

```
glutSwapBuffers();
```

will cause the two buffers to be swapped. After they are swapped, we need to inform OpenGL to redraw the window (using the new contents of the foreground buffer).

The function

```
glutPostRedisplay()
```

forces a re-draw of the window.

The code required to display the bouncing ball is as follows:

```
void Display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glClear(GL_COLOR_BUFFER_BIT);
    // 160 is max X value in our world
    // Define X position of the ball to be at center of window
    xpos = 80.;
    // 120 is max Y value in our world
    // set Y position to increment 1.5 times the direction of the bounce
    ypos = ypos+ydir *1.5;
    // If ball touches the top, change direction of ball downwards
    if (ypos == 120-RadiusOfBall)
        ydir = -1;
    // If ball touches the bottom, change direction of ball downwards
    else if (ypos < RadiusOfBall)
        ydir = 1;
    //reset transformation state
    glLoadIdentity();
    // apply the translation
    glTranslatef(xpos,ypos, 0.);
    // draw the ball with the current transformation state applied
    draw_ball();
    // swap the buffers
    glutSwapBuffers();
    // force a redraw using the new contents
    glutPostRedisplay();
}
```

The entire code for this example can be found under *Example2_1/Example2_1.cpp*

Scaling

An object can be scaled (stretched or shrunk) along the x - and y - axis by multiplying all its points by the scale factors S_x and S_y . All points $P=(x,y)$ on the scaled shape will now become $P'=(x',y')$ such that $x'=S_x.x$, $y'=S_y.y$. In matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} S_x & 0 \\ S_y & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \text{ or } P'=S.P$$

In Figure 2.8, we show the circular shape being scaled by $(1/2,2)$ and by $(2,1/2)$.

Notice from the figure that scaling changes the bottom (base) position of the shape. This is because the scaling equation we defined occurs around the origin

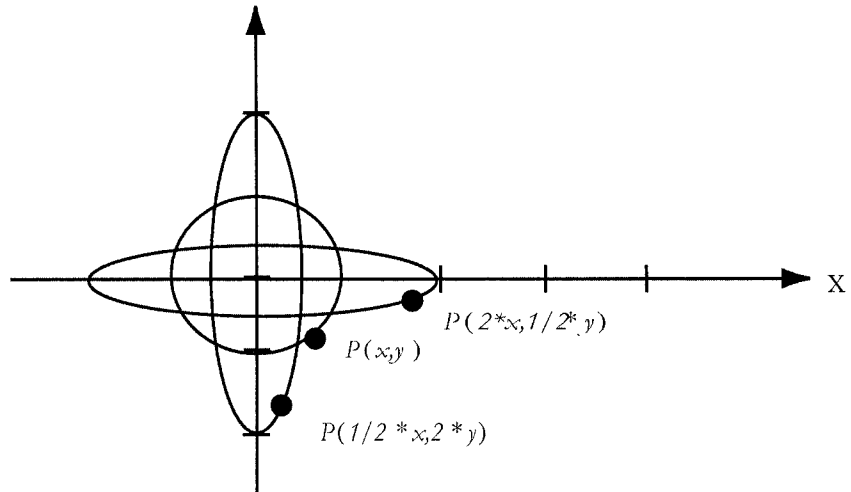


Fig.2.8: Scaling of a circle.

of the world coordinate system. That is, only points at the origin remain unchanged. Many times it is more desirable to scale the shape about some other predefined point. In the case of a bouncing ball, we may prefer to scale about the bottommost point of the ball because we want the base of the shape to remain unchanged. In a later section, we shall see how to scale the shape about a predefined position.

Let us go back to our bouncing ball example. When the ball hits the ground we want it to squash on impact and then stretch back up again into its original shape. This is what we expect from a real (soft) ball bouncing on the ground. The floating point command to scale an object is

glScalef(Sx, Sy, Sz)

which accepts three arguments for scaling along the x -, y - and z -axes. Scaling has no effect when $Sx=Sy=Sz=1$. For now, we will set the Sz to be 1.

We can modify the Display code from the previous example to include scaling of the ball. When the ball hits the ground, and for some time afterwards, we stop translating and squash the shape. That is, we scale down the shape along the y -axis and proportionately scale up along the x -axis. When we reach a predetermined squash, we stretch back up again and restart the translation. Shown below is the code to perform this transformation.

```
// Shape has hit the ground! Stop moving and start squashing down and then back up
if (ypos == RadiusOfBall && ydir == -1 ) {
    sy = sy*squash ;
```

```

        if (sy < 0.8)
            // reached maximum squash, now un-squash back up
            squash = 1.1;
        else if (sy > 1.) {
            // reset squash parameters and bounce ball back upwards
            sy = 1.;
            squash = 0.9;
            ydir = 1;
        }
        sx = 1./sy;
    }
    // 120 is max Y value in our world
    // set Y position to increment 1.5 times the direction of the bounce
    else {
        ypos = ypos + ydir * 1.5 - (1.-sy)*RadiusOfBall;
        if (ypos == 120-RadiusOfBall)
            ydir = -1;
        else if (ypos < RadiusOfBall)
            ydir = 1;
    }
    glLoadIdentity();
    glTranslatef(xpos,ypos, 0.);
    glScalef(sx,sy, 1.);
    draw_ball();

```

The entire code can be found under *Example 2_2/Example 2_2.cpp*. Notice that two transformations are applied to the object—translation and scaling.

When you run the program, you will notice that the ball doesn't stay on the ground when it squashes! It seems to jump up. This happens because the scaling is happening about the origin—which is the center of the ball!

Rotation

A shape can be rotated about any of the three axes. A rotation about the z-axis will actually rotate the shape in the *xy*-plane, which is what we desire in our 2D world. The points are rotated through an angle θ about the world origin as shown in Fig.2.9. Mathematically, a rotation about the z-axis by an angle θ would result in point P (*x,y*) transforming to P' (*x',y'*) as defined below:

$$\begin{aligned}
 x' &= x \cos(\theta) - y \sin(\theta) \\
 y' &= x \sin(\theta) + y \cos(\theta)
 \end{aligned}$$

In matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \text{ or } P = R.P$$

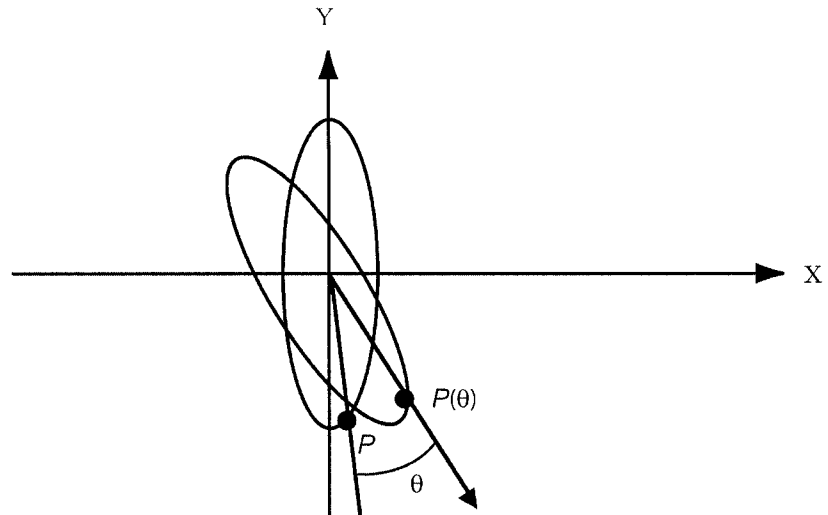


Fig.2.9: Rotation of shape

Where \mathbf{R} is the rotation matrix shown above. Positive angles are measured counterclockwise from x toward y . Just like scaling, rotation occurs about the world origin. Only points at the origin are unchanged after the rotation transformation.

The OpenGL routine for rotation is

glRotatef(Rot, vx, vy, vz)

It expects the angle of rotation in degrees and a nonzero vector value about which you wish the rotation to occur. For rotations in the xy -plane, we would set $vx=vy=0$ and $vz=1$ (i.e., the unit vector along the z -axis)

Let us go back to our bouncing ball example. Of course, rotating a round ball will have no effect on the ball. Let us redefine the `draw_ball` routine to draw an elongated ball.

```
// Draw the ball, centered at the origin and scaled along the X axis
// It's a football!
void draw_ball() {
    glColor3f(0.6,0.3,0.);
    glScalef(1.3,1.,1.);
    MyCircle2f(0.,0.,RadiusOfBall);
}
```

To rotate the ball while it's bouncing, we add the following lines of code to our `Display` function:

```
rot = rot+2.;
```

```
// reset rotation after a full circle
if (rot >= 360)
    rot = 0;
glLoadIdentity();
glTranslatef(xpos,ypos, 0.);
glScalef(sx,sy, 1.);
glRotatef(rot, 0.,0.,1.);
draw_ball();
```

The entire code can be found under *Example 2_3/Example 2_3.cpp*. Note that we apply the transformations in a certain order—first rotate, then scale, and finally translate. The order of transformations does affect the final outcome of the display. Try changing the order in the example and see what happens.

Let us see how transformations are combined together in more detail.

2.3 Homogenous Coordinates and Composition of Matrix Transformations

We have seen the different vector/matrix representations for translation, scaling, and rotation. Unfortunately these all differ in their representations and cannot be combined in a consistent manner. To treat all transformations in a consistent way, the concept of *homogenous coordinates* was borrowed from geometry and applied to CG.

With homogenous coordinates, we add a third coordinate to a (2D) point. Instead of being represented by a pair of numbers (x,y) , each point is now represented by a triplet (x,y,W) , or in vector notation as $\begin{bmatrix} x \\ y \\ W \end{bmatrix}$

To go from homogenous coordinates back to our original non-homogenous world, we just divide the coordinates by W . So the homogenous point represented by: $\begin{bmatrix} x \\ y \\ W \end{bmatrix}$ is equal to the point $(x/W, y/W)$.

Two points of homogenous coordinates (x,y,W) and (x',y',W') are the same point if one is a multiple of the other. So, for example, $(1,2,1)$ and $(2,4,2)$ are the same points represented by different coordinate triples. The points with $W=0$ are points at infinity and will not appear in our discussions.

Because 2D points are now three element column vectors, transformation matrices used to transform a point to a new location must be of cardinality 3×3 .

In this system, the translation matrix **T** is defined as

$$\begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix}$$

Any point $P(x, y, W)$ that is translated by the matrix \mathbf{T} results in the transformed point P' defined by the matrix-vector product of \mathbf{T} and P :

$$P'(x', y', W') = \mathbf{T} \cdot P$$

$$\begin{bmatrix} x' \\ y' \\ W' \end{bmatrix} = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ W \end{bmatrix} = \begin{bmatrix} x + T_x * W \\ y + T_y * W \\ W \end{bmatrix}$$

Satisfy yourself that for $W=1$ this is consistent with what we learned earlier. For any other value of W , convert the coordinates to their non-homogenous form by dividing by W and verify the result as well.

The Scale transformation matrix (\mathbf{S}) in homogenous coordinates is defined as:

$$\mathbf{S} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and the rotation matrix (\mathbf{R}) as:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Each of these matrices can be multiplied successively by the homogenized points of our object to yield the final transformed points. For example, suppose that points on shape P are translated by a translation matrix \mathbf{T}_1 and then by \mathbf{T}_2 . The net result should be the translation $(\mathbf{T}_1 + \mathbf{T}_2)$. Verify that the equation:

$$P' = \mathbf{T}_2 \cdot (\mathbf{T}_1 \cdot P)$$

does indeed result in the desired transformed points.

In fact, it can be shown that this equation can be derived to be:

$$P' = \mathbf{T}_2 \cdot \mathbf{T}_1 \cdot P = (\mathbf{T}_2 \cdot \mathbf{T}_1) \cdot P = \mathbf{T}_{21} \cdot P$$

That is, the matrix product of the two translations produces the desired transformation matrix. This matrix product is referred to as the concatenation or *composition* of the two transformations. Again, please verify for yourself that this is indeed the case. It can be proven mathematically that applying successive transformations to a vertex is equivalent to calculating the product of the transformation matrices first and then multiplying the compounded matrix to the vertex [FOLE95]. That is, we can selectively multiply the fundamental \mathbf{R} , \mathbf{S} and \mathbf{T} matrices to produce desired composite transformation matrices.

The basic purpose of composing transformations is to gain efficiency. Rather than applying a series of transformations one after another, a single composed transformation is applied to the points on our object. At any given time, the current transformation state is represented by the composite matrix of the applied transformations.

Let us apply the composition principle to the ball shape under discussion. Let us go back to *Example2_2*, where we were squashing the ball upon contact with the ground. Remember that the ball seemed to jump when we squashed it. To avoid this jump, we wish to scale the ball about its bottom most point. In other words, we wish the bottommost point to stay at the same place when we squash the ball.

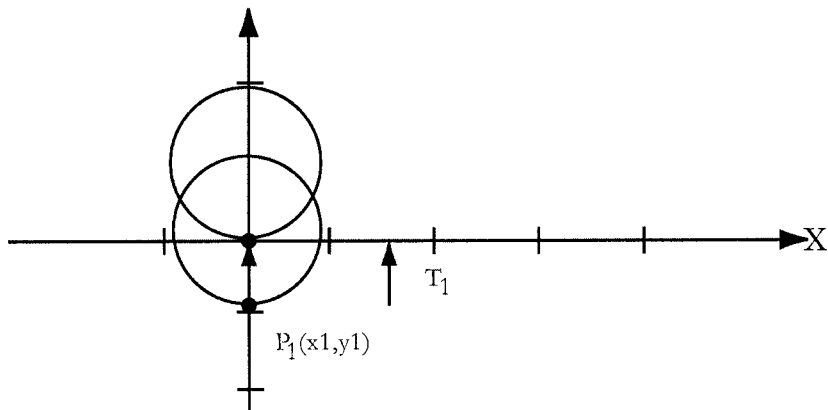


Fig.2.10: Translate shape by T_1

From our study of transformations, we know how to scale the ball about the world origin (which is where the center of the ball is located). To transform the ball about its base, we can convert this problem into three basic transformations:

First, move the ball so that the desired fixed point of scaling (the base in this

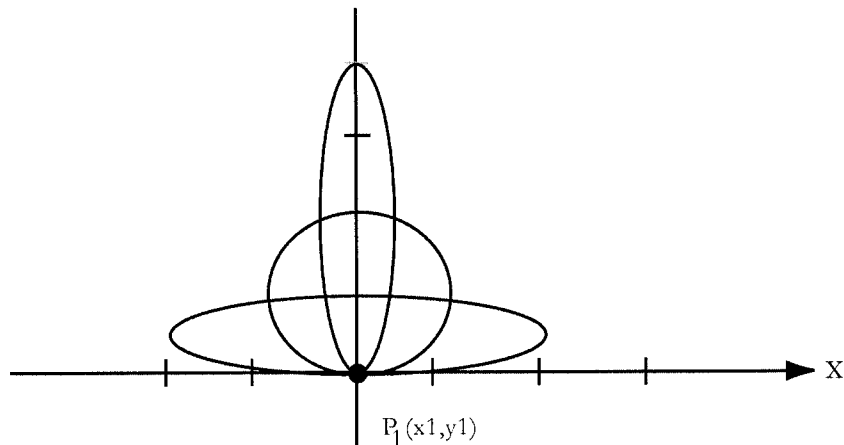


Fig.2.11: Scaling the translated shape

glMultMatrixf(M)

multiplies the current transformation matrix by the specified matrix, **M**. Each successive `glMultMatrix*()` or transformation command multiplies a new 4×4 matrix **M** to the current transformation matrix **C** to yield the (new current) composite matrix **C.M**. In the end, vertices *v* are multiplied by the current matrix to yield the transformed vertex locations. This process means that the last transformation command called in your program is actually the first one applied to the vertices. For this reason, you have to specify the matrices in the reverse order of the transformation applied.

Consider the following code sequence, which draws a single point using three transformations:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(T2);           // apply transformation T2
glMultMatrixf(S);           // apply transformation S
glMultMatrixf(T1);          // apply transformation T1
glBegin(GL_POINTS);
    glVertex3f(v);           // draw transformed vertex v
glEnd();
```

Remember, we do need to set the matrix mode to specify which space the matrix operations are occurring within (in this case, the object space represented by the modelview matrix). With this code, the modelview matrix successively contains **I**, **T₂**, **T₂.S**, and finally **T₂.S.T₁**, where **I** represents the identity matrix. The final vertex transformation applied is equal to: **(T₂.(S.(T₁.v)))** - notice that the transformations to vertex *v* effectively occur in the opposite order in which they were specified.

Coming back to our example, we want to scale the ball about its base. To do this, we translate the ball so that its base lies at the origin. This is done by translating the ball upward by its radius. If we initially define the transformation matrix **T₁** as the identity matrix

```
GLfloat T1[16] = {1.,0.,0.,0.,\
                  0.,1.,0.,0.,\
                  0.,0.,1.,0.,\
                  0.,0.,0.,1.}
```

then to attain the matrix to translate the ball upward by its radius, we can just set

```
T1[13] = RadiusOfBall;
Defining T1 to be {1.,0.,0.,0.,\
                  0.,1.,0.,0.,\
```

```
0.,0.,RadiusOfBall,0.,\
0.,0.,0.,1.}
```

To move the ball back down, we set

```
T1[13] = -RadiusOfBall;
```

If we define the scaling matrix **S** also to initially be the identity matrix, then to squash the ball by *sx* and *sy*, **S** would be set as

```
S[0] = sx;
S[5] = sy;
```

The transformation sequence for scaling and bouncing the ball would be as follows:

```
//Retain the current position of the ball
T[12] = xpos;
T[13] = ypos;
glLoadMatrixf(T);
//Squash the ball about its base

T1[13] = -RadiusOfBall;
// Translate ball back to center
glMultMatrixf(T1);
S[0] = sx;
S[5] = sy;
// Scale the ball about its bottom
glMultMatrixf(S);
T1[13] = RadiusOfBall;
// Translate the ball upward so that it's bottom point is at the origin
glMultMatrixf(T1);
draw_ball();
```

Now you will see that the ball remains on the ground when it is squashed—just as we wanted! The entire code can be found under *Example 2_4*.

An Easier Alternative

The three commands we learned earlier, namely, `glTranslate`, `glScale`, and `glRotate`, are all equivalent to producing an appropriate translation, rotation, or scaling matrix and then calling `glMultMatrix*()` with this matrix. Using the above functions, the same transform sequence for the bouncing ball would be as follows:

```
//reset transformation state
```

```
glLoadIdentity();  
// retain current position  
glTranslatef(xpos,ypos, 0.);  
  
// Translate ball back to center  
glTranslatef(0.,-RadiusOfBall, 0.);  
// Scale ball about its bottom  
glScalef(sx,sy, 1.);  
// Translate the ball upward so that it's bottom is at the origin  
glTranslatef(0.,RadiusOfBall, 0.);  
// draw the ball  
draw_ball();
```

Matrix math is very important to the understanding of graphics routines. But using matrices in actual code is involved and often tedious. It is much easier to use the `glTranslate`, `glScale`, and `glRotate` commands. These commands internally create the appropriate matrix and perform the necessary computations. For the rest of the book, we shall use the above-mentioned OpenGL functions in our coding examples.

Summary

In this chapter, we have covered one of the most mathematical concepts in Computer Graphics: transformations. Transformations are based on vector and matrix math. The basic transformations are of three types: translation, scale, and rotation. These can be composed together to create complex motions. Although the actual matrix math for transformations can be fairly tedious, OpenGL provides a number of functions to easily manipulate and apply transformations to objects. In the next section, we shall extend the concepts of 2D transformations learned here to the more general 3D case. This is when we will really be able to appreciate the true power of transformations.



<http://www.springer.com/978-0-387-95504-9>

Principles of Computer Graphics
Theory and Practice Using OpenGL and Maya®
Govil-Pai, S.
2005, XIV, 296 p. 245 illus., 20 illus. in color., Hardcover
ISBN: 978-0-387-95504-9