13

Geometric Transformations

In chapter 12 we discussed a variety of geometric shapes. There are many instances where such shapes by themselves are not sufficient. We want to move them around, make them bigger or smaller, rotate them, flip them different ways, display them on the screen and assemble them into more complex shapes. This chapter describes the basic matrix operations that allow us to do all of these things using a few simple concepts combined in a variety of ways.

Most of what we want to do with our shapes can be composed of three basic transforms: translate, scale and rotate. These three and all of their possible combinations are part of a general class of linear transforms.

Translate

Translation is the transform that moves shapes around. A translation has two parameters D_x and D_y, which are the distance to move the object in X and Y respectively. The equations for this transform are shown in figure 31.1.

$$trans(D_x, D_y)$$

$$X' = X + D_x$$
$$Y' = Y + D_y$$

Figure 13.1 – Translation equations

Scale

Scaling is the transformation that changes the size of shapes. Scaling has two parameters S_x and S_y. These are scale factors that are multiplied times X and Y respectively to change size. The scaling equations are shown in figure 13.2.

$$scale(S_x, S_y)$$

$$X' = X \bullet S_x$$
$$Y' = Y \bullet S_y$$

Figure 13.2 – Scaling equations

Scale factors that are greater than 1.0 will make an object larger. Scale factors between 0.0 and 1.0 will make an object smaller. Negative scale factors will reflect an object across the axes. Figure 13.3 shows examples which transform the rectangles on the left into the rectangles on the right.

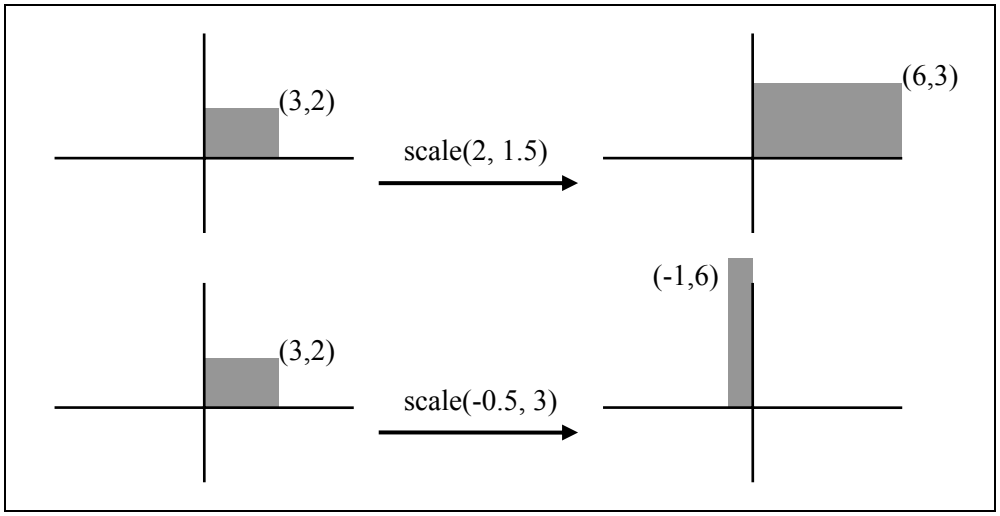

Figure 13.3 – Example scaling transformations

All scaling transformations are relative to the origin. That is only the origin point stays fixed. Any points that are not at the origin will move as a result of scaling. Figure 13.4 shows what will happen to shapes not at the origin when scaling is applied. The object got twice as big, but also moved twice as far away from the origin.

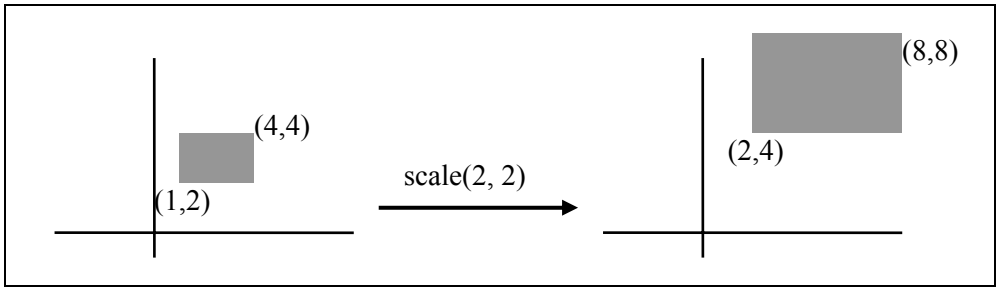

Figure 13.4 – Scaling not at the origin

A common mistake when using the scaling transformation occurs when one does not want the size to change, but the scaling transformation has been programmed as part of the transformation sequence. The temptation is to use *scale(0,0)* for a null transformation. Unfortunately this will transform the entire shape into a single point at the origin. In most applications this appears as a single black pixel in the upper left corner of the window. The correct scaling when no scaling is desired is *scale(1,1)*.

Rotate

Our final transformation is rotation. Rotation is defined by a counterclockwise angle from the positive X axis. This is generally in radians. The rotation equations are shown in figure 13.5. Sometimes rotation is specified not as an angle, but as the sine and cosine of the angle ($rot(sin_A,cos_A)$). In many interactive problems we can more easily compute the sine and cosine than we can the angle and our equations do not actually need the angle.

$$rot(A)$$

$$X' = \cos(A)X - \sin(A)Y$$
$$Y' = \sin(A)X + \cos(A)Y$$

Figure 13.5 – Rotation equations

Rotation is always centered at the origin. Just as with scaling, objects not at the origin will move when rotation is applied. This is shown in the sample rotations in figure 13.6.

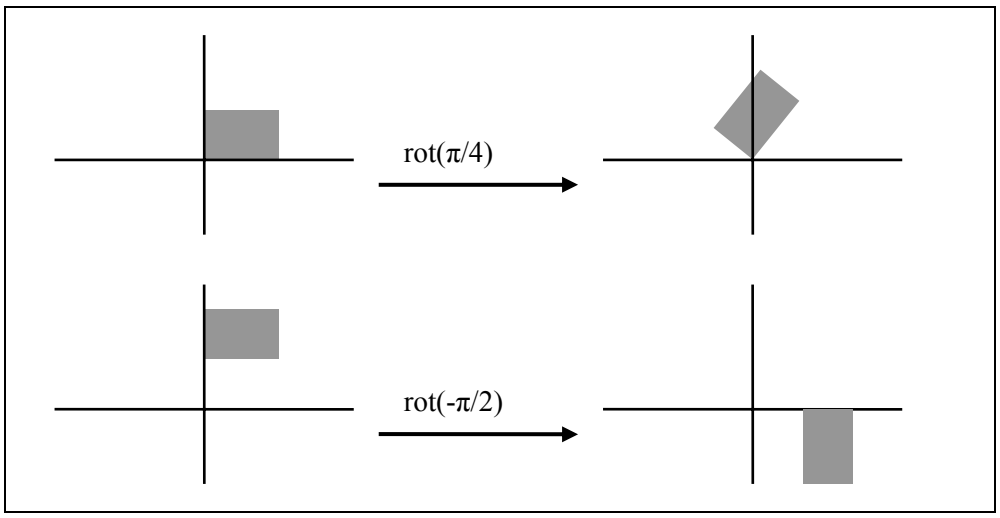

Figure 13.6 – Example rotations

Matrix transformations

The three basic transforms by themselves are not very interesting. We also need to build up complex transforms when we put many shapes together to form complex objects. To accommodate this we need a more flexible model of transformations than the equations themselves. All of our three basic transformations are special cases of the more general linear transformations shown in figure 13.7. Using homogeneous coordinates for our points we can represent all of our transformations as a matrix as shown in figure 13.8.

$$X' = aX + bY + c$$
$$Y' = dX + eY + f$$

$$trans(D_x, D_y) \rightarrow \begin{matrix} X' = 1X + 0Y + D_x \\ Y' = 0X + 1Y + D_y \end{matrix}$$

$$scale(S_x, S_y) \rightarrow \begin{matrix} X' = S_x X + 0Y + 0 \\ Y' = 0X + S_y Y + 0 \end{matrix}$$

$$rot(A) \rightarrow \begin{matrix} X' = \cos(A)X - \sin(A)Y + 0 \\ Y' = \sin(A)X + \cos(A)Y + 0 \end{matrix}$$

Figure 13.7 – General linear transform

$$\begin{matrix} X' = aX + bY + c \\ Y' = dX + eY + f \end{matrix} \Rightarrow \begin{bmatrix} X' \\ Y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

Figure 13.8 – Matrix form for homogeneous linear transforms

Using the matrix form of the general linear transform, we can produce a matrix representation for each of our three primitive transformations. The matrix forms for each transformation are shown in figure 13.9.

$$trans(D_x, D_y) = \begin{bmatrix} 1 & 0 & D_x \\ 0 & 1 & D_y \\ 0 & 0 & 1 \end{bmatrix} \quad scale(S_x, S_y) = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad rot(A) = \begin{bmatrix} \cos(A) & -\sin(A) & 0 \\ \sin(A) & \cos(A) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 13.9 – Matrix representations for translate, scale, rotate

Careful examination of these matrices shows that the third column of the matrix contains only the translation component of the transformation. For scale and rotate these coefficients are zeros. The 1 as the third coordinate of a homogeneous point is multiplied by these, which adds in the translation constants. Remember vectors have direction (rotation) and magnitude (scale) but no position (translation). This is the reason that we represent vectors as [*X Y 0*]. The zero homogeneous coordinate discards the translation information (multiply by zero) while retaining changes in orientation and magnitude.

Concatenation

The real value of the matrix representation comes in the ability to concatenate together a sequence of transformations. Suppose that we want to scale an object to a new size, rotate the object and then move it to some location. The steps for this are shown in figure 13.10. These steps can be collected together into a single expression. Using the associative property of matrix multiplication, we can replace the three steps by multiplying together all of the matrices to produce a single transformation matrix. This single matrix can then be applied to each point in the object to produce the final points from all of the transformations in a single matrix multiply per point. This is much more efficient and much easier than working out the algebra of all of the steps by hand.

$$\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = \begin{bmatrix} X_1 \\ Y_1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos(A) & -\sin(A) & 0 \\ \sin(A) & \cos(A) & 0 \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} X_1 \\ Y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} X_2 \\ Y_2 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & D_x \\ 0 & 1 & D_y \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} X_2 \\ Y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} X_{final} \\ Y_{final} \\ 1 \end{bmatrix}$$

$$trans(D_x, D_y) \bullet \left(rot(A) \bullet \left(scale(S_x, S_y) \bullet \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} \right) \right) = \begin{bmatrix} X_{final} \\ Y_{final} \\ 1 \end{bmatrix}$$

$$\left(trans(D_x, D_y) \bullet rot(A) \bullet scale(S_x, S_y) \right) \bullet \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = \begin{bmatrix} X_{final} \\ Y_{final} \\ 1 \end{bmatrix}$$

$$M \bullet \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = \begin{bmatrix} X_{final} \\ Y_{final} \\ 1 \end{bmatrix}$$

Figure 13.10 – Concatenation of matrix transformations

Any sequence of matrix transformations can be collapsed into a single matrix using matrix multiplication. However, the order is important. Matrix multiply is associative but not commutative. Consider the shape in figure 13.11. In the first case it is scaled and then rotated as shown in the top pair. In the second case it is rotated then scaled. The results are not at all the same. Note also that the order of transformations is read from right to left, not left to right. In figure 13.10 the shape is scaled, then rotated then translated. The algorithm for matrix multiply is given in appendix A1.2b.

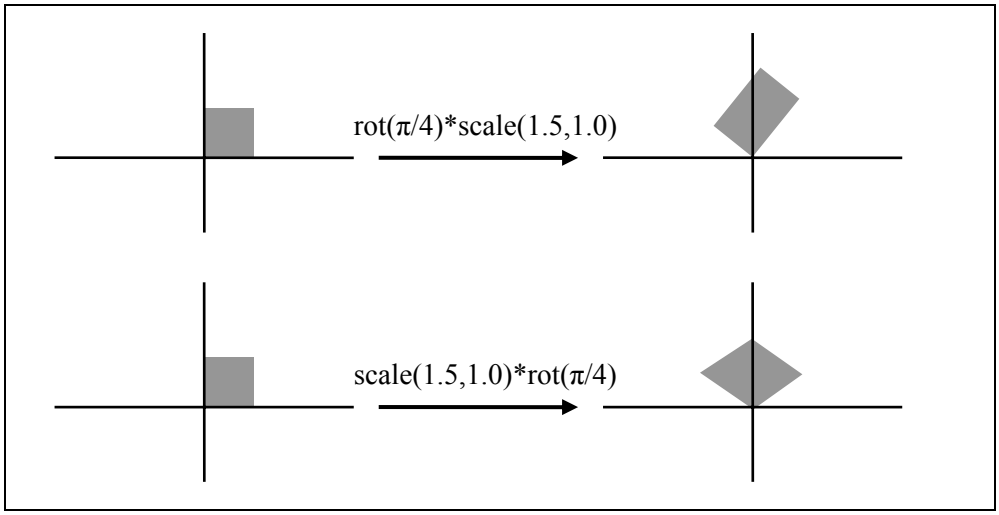

Figure 13.11 – Importance of transformation order

We will frequently not use the matrices in our discussions of transformations. Instead we will use a sequence of transformation functions such as: $trans(14,16) \bullet rotate(-0.3) \bullet scale(0.1,2.4)$. It is important to remember how multiplication of a transformation time a point actually works. Because of the matrix algebra transformation sequences are read in right to left rather than left to right order. In the example the object will first be scaled, then rotated, then translated.

Inverse matrices

It is very important in interactive applications to be able to compute in inverse of a transformation or sequence of transformations. Mapping an object onto a display may involved transformation *M*. Mapping a mouse point that

occurs in display coordinates back to the object will require the transformation M^{-1}.

The inverses of the basic three transformations are quite straightforward and are given in figure 13.12. The inverse of any sequence of transformations is the individual inverses in the opposite order. Sometimes, however, one has just the matrix *M* and wants the inverse. The general matrix inverse algorithm is given in appendix A1.2e. Only square matrices have inverses and not all square matrices have an inverse. However, any matrix built up from the basic three will have an inverse as long as scaling by zero is not one of the transformations.

$$trans(D_x, D_y)^{-1} = trans(-D_x, -D_y)$$

$$scale(S_x, S_y)^{-1} = scale\left(\frac{1}{S_x}, \frac{1}{S_y}\right)$$

$$rot(A)^{-1} = rot(-A)$$

$$(A \bullet B \bullet C)^{-1} = C^{-1} \bullet B^{-1} \bullet A^{-1}$$

Figure 13.12 – Inverses of transformations

Transformation about an arbitrary point

One of the problems that we discussed with scaling and rotation transformations is that they always occur about the origin. This is frequently not very helpful. For example when we have a picture of a car and we want to rotate the wheels, we want them to rotate about their axles and not about the origin, which would move them completely away from the car.

Now that we have concatenation of transformations and inverses of transformations we can operate about any point $[C_x,C_y]$. The technique is to move the point to the origin, do the rotation or scaling and then put it back. Figure 13.13 shows how to rotate about a point. Remember that the transformation sequence reads right to left. Translating by $-C_x$,$-C_y$ moves that point to the origin. We then do the rotate and then translate the origin back to $[C_x,C_y]$. The same can be done for scaling. This is a special case of a general geometric technique that will be heavily used in Chapter 14. We transform the problem to an easier problem (in this case at the origin) do the operation that we need to do and then using the inverse transformations, we put the problem back in its original state.

$$trans(C_x, C_y) \bullet rot(A) \bullet trans(-C_x, -C_y) \bullet \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

Figure 13.13 – Rotation about a point $[C_x, C_y]$

Generalized 3 point transform

There are some cases where we have three points *A, B* and *C* that we want to transform to three new points *A', B'* and *C'*. If points *A, B* and *C* are not collinear then we can compute a general transformation matrix *M* that will accomplish the goal. Using each of the points as columns in a matrix, we formulate the problem as shown at the top of figure 13.14. If we call the matrix of points *A, B* and *C* the *From* matrix and the matrix of *A', B'* and *C'* the *To* matrix. We can easily compute the desired matrix *M* as shown. We might use this as a general transformation technique where three points on an object are dragged to three new positions and an appropriate transform is needed.

$$M \bullet \begin{bmatrix} A & B & C \end{bmatrix} = \begin{bmatrix} A' & B' & C' \end{bmatrix}$$
$$M \bullet From = To$$
$$M \bullet From \bullet From^{-1} = To \bullet From^{-1}$$
$$M = To \bullet From^{-1}$$

Figure 13.14 – General transform from three points to three new points

Shape transforms

Up to this point we have talked about transforming vectors and points. However, what we really want is to transform shapes. Before addressing each of the shapes, we need to consider is the concept of closure of a shape representation under a particular transformation. A shape representation is closed under a transformation if the transformation yields a shape with the same type of representation. This is best illustrated by examples.

Because all of our transformations are linear, the set of lines is closed under any of our transformations. If we take the endpoints of a line segment and transform them with any linear transformation then the resulting endpoints will define a line segment that contains the transformation of any of the original points on the line segment. Thus the set of line shapes is closed under any linear transformation.

Circles are closed for translation. Moving any circle to another place will produce a circle. Circles are closed under rotation. Rotate a circle and you still have a circle. Circles are closed under uniform scaling ($S_x = S_y$). The result will be a circle of a different size. Circles are not closed under non-uniform scaling, the result is an ellipse. Fortunately the result is an axis-aligned ellipse, which is why we use axis-aligned ellipses rather than circles as our basic shape primitive. However, axis-aligned ellipses are not closed under rotation. Rotating an ellipse off of the axes produces an ellipse that cannot be represented simply by its bounding rectangle.

Axis-aligned rectangles of the kind that we commonly used are not closed under rotation. The Xmax, Xmin, Ymax and Ymin representation of a rectangle cannot be rotated without producing a different shape representation. However, if rotation is excluded, transforming the upper left and lower right points of a rectangle will produce a new correct rectangle.

Cubic curves have very nice transformation properties. Consider a curve with coefficient matrix C that is to be transformed by matrix M. What we want is a new curve such that each point on the original curve will be transformed by M. This is shown in figure 13.15. To transform the curve one need only multiply the coefficient matrix by M to produce a new coefficient matrix. By the same reasoning we can transform any spline to a new spline of the same type by multiplying the geometry matrix by M. The set of cubic curves is closed under any linear transform.

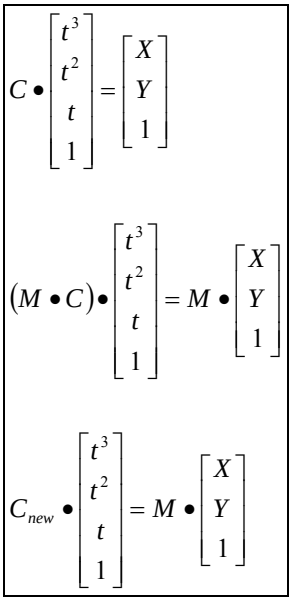

Figure 13.15 – Transforming a cubic curve

A polygon consists of a list of vertices connected by straight lines. By transforming the vertices we transform the polygon and all of its edges. If we want to transform a rectangle we convert it to a four vertex polygon and transform. We lose the efficiency properties of rectangles but the shape is preserved. Because curvilinear shapes consist of vertices and control points connected by lines or cubic curves we simply transform the vertices and control points to produce a new shape. With the exception of rectangles and ellipses we can transform any shape by simply transforming its control points. Many systems such as PostScript and PDF model virtually all shapes using piecewise cubic curves because they are closed under all linear transforms and form a uniform model of virtually anything.

Viewing transforms

We now have mechanisms for transforming shapes in arbitrary ways by composing sequences of linear transformations. We now need to deal with the problems of coordinate systems. In an interactive application there are many possible coordinate systems that are generally related to each other by linear transformations. For an interactive program to work correctly we must understand the coordinate system in which our points are defined and make

certain that we have all points in the same coordinate system when we are working on them. Figure 13.16 shows a diagram of most of the coordinate systems found in interactive applications.

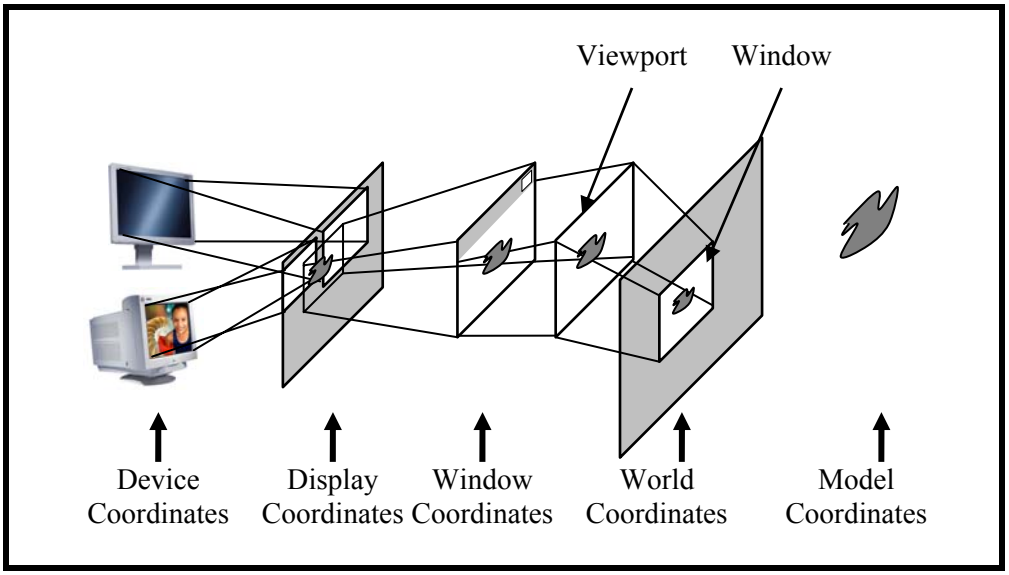

Figure 13.16 – Coordinate Systems

Model coordinates are the coordinates in which objects are defined. These can be anything. They can be parsecs, nanometers, rods or angstroms. It is whatever is appropriate for the model. There is a *modeling transformation* that will transform points from model coordinates to *world coordinates*. There may be many model objects in an application and thus many modeling transformations to bring each model into world coordinates. The modeling transformation may be built up hierarchically with models being transformed into other models and so on until they are transformed to world coordinates. We will discuss hierarchic model transforms in the next section.

World coordinates are a single 2D surface onto which all objects to be presented are transformed. Onto this surface we place a rectangle called the *window*. This is a different window from the one that we normally see on the screen. The world of computer graphics evolved somewhat differently from graphical user interfaces and they use the same words for different concepts. The purpose of the window is to select the region of the world that is to be displayed.

The window is defined in world coordinates. World coordinates are defined in any units suitable to the application.

On the screen window (the one familiar to interactive users) there is another rectangle called the *viewport*. The viewport shows where the selected portion of the world is to be displayed. The viewport also specifies the clipping of the world objects to be drawn. The viewport is defined in screen *window coordinates*. The transformation from the world window to the viewport is called the *viewing transformation*. Window coordinates are defined in pixels.

From the window coordinates, where the viewport is defined, there is a transformation into *display coordinates*. Display coordinates are a uniform coordinate system for all of the display devices connected to a workstation. In our example there are two screens that have been mapped into display coordinates side by side. Screen window positions are defined in display coordinates. This allows the user to move windows anywhere without regard to the boundaries between various display devices. Display coordinates are always in pixels.

From display coordinates there is a transformation to actual *device coordinates*. Device coordinates are always pixels. The mapping from display coordinates to device coordinates is generally hidden from the programmer. Most software defines itself in terms of display coordinates but there are times when more information is needed. For example it is useful to know where display device boundaries are located in display coordinates. It is not very user friendly to pop up a menu right across the boundary between two display devices. Because displays are frequently not the same size, there are often undisplayable gaps in display coordinates. Popping up a window into one of these undisplayable gaps is not good form.

On occasion when printing or when doing very precise work on a screen we need to work in *physical coordinates*. Physical coordinates are the actual physicals sizes of displayed or printed objects. We may, for example, need a shape that is actually 1.2 inches high. This is independent of the number of pixels required to display the shape. When displaying on the screen we generally do not have access to the transform between physical coordinates and device coordinates. However, with most printers we specify drawing in physical coordinates and rely upon the printer to perform the appropriate transformation.

When we are interacting we need to know exactly what coordinates we are working in. Generally mouse points are received in window coordinates. The objects we are manipulating are in model coordinates. Before we can do any

manipulation we must have the mouse points and the object points together in the same coordinate system. Sometimes when we are moving windows around or dragging between windows we work in display coordinates because we are outside of or independent of a particular screen window. It is very important to understand the coordinates that we are working in and the coordinates of all parts of the interaction.

Because all of these transformations are linear transformations we can represent them in a matrix. We can compute a single matrix that transforms points in model coordinates all the way to device coordinates in a single matrix multiply.

Windowing

The viewing transformation, as discussed earlier, maps a portion of world coordinates into a viewport in screen window coordinates. There are a variety of ways in which this is expressed. We will look at the two most popular. The first technique is windowing. This is where a rectangular window in world coordinates is mapped to a rectangular viewport in screen window coordinates. The specification of the problem is shown in figure 13.17.

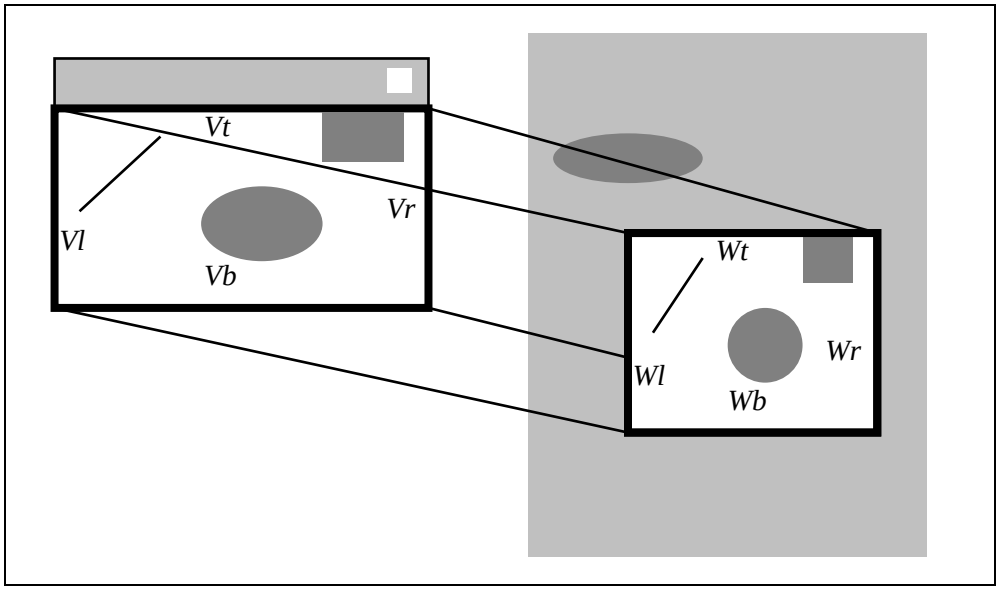

Figure 13.17 – Window Viewing Transformation

Using this representation of the transformation, the programmer can pan across the world by moving the window. Moving the window to the left causes the objects in the viewport to scroll right. Enlarging the window is the same as zooming out and shrinking the window will zoom in. As shown in figure 13.17 the window and the viewport need not have the same aspect ratio. This can produce various scaling effects.

Using the viewport and window parameters shown in figure 13.17 we can develop the transformation sequence in figure 13.18. This first moves the upper left corner of the window to the origin. It then scales the window size to the viewport size and then translates the origin to the upper left corner of the viewport. Note that we could have used any point in the window (center, lower left, etc) as long as we use the same point on the viewport.

$$trans(Vl,Vt) \bullet scale\left(\frac{Vr-Vl}{Wr-Wl},\frac{Vb-Vt}{Wb-Wt}\right) \bullet trans(-Wl,-Wt) \bullet \begin{bmatrix} X_{window} \\ Y_{window} \\ 1 \end{bmatrix} = \begin{bmatrix} X_{viuewport} \\ Y_{viewport} \\ 1 \end{bmatrix}$$

Figure 13.18 – Matrix for windowing transform

If there is a modeling transform *M* that transforms an object from model coordinates to world coordinates we can include it as shown in figure 13.19. Using the associative property we can multiply the modeling transformation and the viewing transformation together into one transformation matrix.

$$M \bullet \begin{bmatrix} X_{model} \\ Y_{model} \\ 1 \end{bmatrix} = \begin{bmatrix} X_{world} \\ Y_{world} \\ 1 \end{bmatrix}$$

$$trans(Vl,Vt) \bullet scale\left(\frac{Vr-Vl}{Wr-Wl},\frac{Vb-Vt}{Wb-Wt}\right) \bullet trans(-Wl,-Wt) \bullet M \bullet \begin{bmatrix} X_{model} \\ Y_{model} \\ 1 \end{bmatrix} = \begin{bmatrix} X_{viuewport} \\ Y_{viewport} \\ 1 \end{bmatrix}$$

Figure 13.19 – Combining modeling and viewing

Point and zoom

One of the problems with the windowing transformation is that a change of aspect ratio is possible in the scaling. This is a flexible model but it provides the user with one more control than is normally desired. An alternative model for the

viewing transformation is "point and zoom". The user specifies a point in the world (*WC*) that should map to the center of the viewport (*VC*) and a zoom or magnification factor. To simplify the user interface the zoom factor (*Z*) is frequently represented as a percentage. Using this representation the transformation is shown in figure 13.20. The window center is moved to the origin, the zoom factor is applied and then the origin is moved to the viewport center. Notice that the scale factor is the same in both *X* and *Y*, which prevents aspect distortion.

$$trans(VC_x, VC_y) \bullet scale\left(\frac{Z}{100}, \frac{Z}{100}\right) \bullet trans(-WC_x, -WC_y) \bullet \begin{bmatrix} X_{window} \\ Y_{window} \\ 1 \end{bmatrix} = \begin{bmatrix} X_{viuewport} \\ Y_{viewport} \\ 1 \end{bmatrix}$$

Figure 13.20 – Point and zoom viewing transformation

Hierarchic models

It is sometimes the case that models are built from parts, which themselves may be built from parts. For example a neighborhood might be built from several houses (which in cheap retirement communities look all alike). Each house is built from doors, windows, chimneys, etc. The door is built of a frame, window and knob. There are model objects and model instances. In a door there is an instance of a knob. In a house there are one or more instances of the door and in a neighborhood there are instances of houses. For each model instance there is a transformation that maps from the model coordinates to instance coordinates. So, for example, there is a transformation *KD* from knob coordinates to door coordinates. There is another transformation *DH* from door coordinates to house coordinates. There are several transformations *HN* from house coordinates to neighborhood coordinates depending on which house we are drawing. Because our neighborhood is our world coordinates, there is a viewing transformation *V* from neighborhood coordinates to screen window coordinates. Transforming a knob from its own coordinates involves the sequence shown in figure 13.21.

$$V \bullet HN \bullet DH \bullet KD \bullet \begin{bmatrix} X_{knob} \\ Y_{knob} \\ 1 \end{bmatrix} = \begin{bmatrix} X_{viewport} \\ Y_{viewport} \\ 1 \end{bmatrix}$$

Figure 13.21 – Transforming a knob into viewport coordinates

To handle this and other transformation tasks it is common for the Graphics object that is passed to the redraw() method to contain a current transformation

(*CT*). Inside the Graphics object each draw method will first transform the shape by *CT* and then draw the shape. When the paint() method is first called *CT* is usually initialized to the viewing transformation (*V*). To manage the current transformation most Graphics objects have a set of methods like those shown in figure 13.22. There is usually a class such as Transform that contains a transformation matrix. The Transform class will usually have methods for multiplication, inverse and construction of the basic transformations. The Transform class usually provides direct access to the six coefficients of the transformation matrix.

```
public class Graphics
{     . . . .
      Transform getCurrentTransform()
            // will return the transformation matrix that is the current transformation
      void setCurrentTransform(Transform newTransform)
            // will change the current transformation
      void translate(float dX, float dY)
            // will change the current transform by CT=CT*trans(dX,dY)
      void rotate(float radians)
            // will change the current transform by CT=CT*rot(radians)
      void scale(float sX, float sY)
            // will change the current transform by CT=CT*scale(sX,sY)
      . . . .
}
```

Figure 13.22 – Current transformation methods

With this facility we can easily construct hierarchic models. What we want is to write methods for each model object that are independent of all other model objects. We do not want to worry about what other parts of the model are doing or what their coordinates may be. This is easily accomplished using the five methods on the Graphics object. The general form for methods to paint an object *O* is shown in figure 13.23.

```
public void paintO(Graphic g)
{
      Transform save = g.getCurrentTransform(); // save the current transformation

      set up instance transformation by calling methods on g in the reverse of the order
            in which they are to be applied.

      draw the object by calling methods on g

      g.setCurrentTransform(save); // restore the current transform to what it was before.
}
```

Figure 13.23 – General paint method for a model object

Suppose that our doorknobs come in three sizes, SMALL, MEDIUM and LARGE and can be placed anywhere on a door. We will develop a knob design with the origin in the center of the knob. We can then write a paintKnob() method as shown in figure 13.24.

```
public enum KnobSize{SMALL, MEDIUM, LARGE};
public void paintKnob(Graphics g, float xLoc, float yLoc, KnobSize size)
{
      Transform save = g.getCurrentTransform();
      g.translate(xLoc, yLoc);

      float s;
      switch (size)
      {
      case KnobSize.SMALL: s=0.75;
      case KnobSize.MEDIUM: s = 1.0;
      case KnobSize.LARGE: s = 1.5;
      }
      g.scale(s, s);

      code to draw the door knob using the Graphics object g

      g.setCurrentTransform(save);
}
```

Figure 13.24 – Paint method for drawing doorknobs

To draw our door knob we first want to scale it to the right size and then translate it to the right location. However, in figures 13.23 and 13.24 these scale() and translate() calls are made in the opposite order. Let us assume that before paintKnob() is called the Graphics object has some current transformation *T*. This may be just the viewing transformation or it may contain transformations that map doors into neighborhoods and then onto the screen or something else. While working with doorknobs we do not care what this transformation is. That is the problem of other parts of our code. Given the transformation matrix *T* let us look at how the current transformation changes as we work through our code (figure 13.25).

1.	at the start	CT=*T*
2.	g.translate(dX,dY)	CT=*T*trans(dX,dY)*
3.	g.scale(s,s)	CT=*T*trans(dX,dY)*scale(s,s)*
4.	g.*drawsomeshape()*	*transform with T*trans(dX,dY)*scale(s,s)*
5.	g.setCurrentTransform(save)	CT=*T*

Figure 13.25 – Building up the current transform

When we start the paintKnob() method, the current transform is *T*. As the two transformations are called in steps 2 and 3, they are multiplied onto the right of the current transformation. When we get to step 4 and want to draw the knob we multiply our points onto the right of the current transformation, which now (reading right to left) has the correct scale/translate sequence followed by whatever *T* may do. Lastly we put the current transformation back the way it was before we exit.

Our doorknob may itself have used a screwHead and a keyHole model object in its drawing. Calling their methods would cause them to change the current transformation, draw themselves and put the transformation back. When screwHead was manipulating the current transformation it would have the knob transformation already built into the current transformation that it received.

Summary

We now have a set of matrix operations that allow us to transform objects in a variety of ways. We have a mechanism to concatenate transformations into sequences. We can use these tools to build up models from parts and to transform a portion of our artificial world onto the a portion of the screen.