14

Interacting with Geometry

In this chapter we bring together input syntax using PPS, the geometric properties of shapes and our geometric transforms to create a variety of standard interactive techniques. We want a standard set of techniques to create a uniform feel for all of our interactions. We will first discuss the choice of coordinates for interaction. We will then discuss how to create, select and manipulate shapes using control points. Lastly we will discuss how to interactively express transformations.

Interactive Coordinates

Before we can begin we need to make certain that the shapes we are manipulating and the mouse input points are in the same coordinate system. Most widget packages will deliver mouse events with the mouse location in window coordinates. Our model could be defined in world coordinates or in a hierarchy of model coordinates.

Window/World coordinates

If our entire model is represented in world coordinates, we need only worry about the viewing transformation *V* that transforms points from world to window coordinates. This case arises in most drawing packages, painting of images, word processors and a number of other applications. With our model in world coordinates and our mouse inputs in window coordinates, we can choose to work in either coordinate system.

interact in world coordinates

If we choose to work in the world coordinates of our model, then we need to multiply the mouse input points by V^{-1}. This will bring the mouse point into world coordinates and we can work there. The advantage of this approach is that only one transformation of the mouse position is required and when we are done with any manipulation of geometry, our geometry is already in the world coordinates of the model.

The disadvantage is when we are selecting shapes. We want to select by being near a shape rather than exactly on it. This is why we did the distance from a point to a shape calculation. However, we want that distance to be defined in pixels not in world coordinates. Suppose we decide that 3 pixels is close enough for a selection and that we have a zoom factor of 1.0. Transforming the mouse point into world coordinates will still leave the selection distance at 3 pixels. Suppose, however, that the zoom factor is 2.0. A distance of 3 pixels in the world is now 6 pixels on the screen. One of the reasons that we zoom in is the get greater precision, but this destroys that advantage. Suppose that the zoom factor is 0.1. This means that a 3 pixel distance in window coordinates becomes a 0.3 pixel distance in world coordinates. Because mouse points are only in pixels, the round-off error may make it impossible to select a shape at all. If our viewing transformation only supports uniform scaling, as with the point and zoom technique, we can transform our selection distance from window to world coordinates by dividing by the zoom factor. Remember that we are transforming a distance, which is neither a point nor a vector. If scaling is non-uniform in X and Y, there is no good way to transform the selection distance from window to world coordinates.

interact in window coordinates

An alternative is to interact in window coordinates. For this technique we transform our shapes by V before comparing them with the mouse position. As we manipulate and transform the shapes we first transform them into window coordinates by V, work on them to get a new shape and then transform them back into world coordinates using V^{-1}. This is a more expensive option because we must transform every shape for selection, and manipulation. If any change is made to a shape it must be transformed back into world coordinates using V^{-1}.

We can also mix the coordinates that we use. For example we can perform selection in window coordinates where the distance matters and then perform other operations in world coordinates. The choice is up to the programmer so long as both the mouse point and the shape are in the same coordinate system while we are working on them.

Window/Model interaction

In some cases the model is composed of many coordinate systems with instance transformations mapping models into the coordinates of other models and then finally into world coordinates. In this case the choice of coordinate system is more complicated because of the number of coordinates from which we can choose. In our doorknob example we could interact in any of the following coordinates: (window, world/neighborhood, house, door, knob).

We can simplify this problem somewhat by first deciding which object we are interacting with. This is part of the selection problem that we will discuss later. Assuming that we have decided that we are manipulating the door, we really have four choices in which to work: (window, world, house or door). Working in world coordinates is probably the least advantageous because we must transform the mouse into world coordinates (with the same problems discussed before) and we must also transform the door and all of its instances of other models into world coordinates.

The choice of coordinates is further complicated by the fact that we frequently will be manipulating the instance transformation rather than the object. For example when we move the door we are probably changing the transformation from door to house coordinates rather than actually modifying the door. It is easiest to work with this instance transformation of the door in house coordinates. However, if we are moving a piece of decorative molding around on the door we are actually modifying the model in door coordinates.

interacting in window coordinates

Window coordinates still have the advantage of allowing selection to operate in the coordinate system of the user. They also have the advantage of unifying the problem no matter what the coordinates. To make this work we must transform the door and all of its instances (such as knobs) into window coordinates. Fortunately this is the same transformation used when drawing the door. Having manipulated the door in window coordinates we now must make a decision. Are we changing the door (moving molding) or changing the door’s instance transformation? If we are changing the door then the shape is transformed by $DH^{-1} \bullet HN^{-1} \bullet V^{-1}$. If, however, we are changing the instance transformation then we transform the new information by $HN^{-1} \bullet V^{-1}$ and then compute a new *DH*.

interacting in model coordinates

It is frequently easier when actually manipulating objects to transform the mouse position into the coordinates where the manipulation should occur. With this technique, all of the manipulations in the rest of the chapter can be performed exactly as described. This also reduces the number of points that must be transformed.

A mixed approach is probably best. Selection and snapping a point to the nearest shape are best done in window coordinates where the distances are consistent with what the user perceives. Once selection is done, all manipulating can be performed by transforming the mouse point back into model coordinates and working there. If any snapping to nearest shape is to be done then this can be done in window coordinates and then the new "snapped" mouse points can be transformed back into model coordinates for the shape manipulations.

For the remainder of this chapter we will assume that the mouse input points and the shape representations are in the same coordinate system. This will simplify the discussions and also make these techniques independent of whatever coordinate system choices are made. Note that specialized coordinate systems such as polar, latitude/longitude or logarithmic are not addressed by any of these discussions. They will need special application-specific attention.

Creating Shapes

Our first interactive task is to interactively create the shapes discussed in chapter 12. In creating shapes we would like a uniform feel (input syntax) that is easily remembered and reused in a variety of situations. All of the shapes discussed in chapter 12 can be created in one of two ways: a single mouse drag or sequence of points.

Single drag shapes

The syntax for a mouse drag is mouse-down, zero or more mouse moves and mouse up. The interactive technique used is called *rubberbanding*. When the mouse goes down the shape is started at that point and then the shape is continuously drawn after every mouse movement as if the mouse would go up at the current mouse position. This provides the user with continuous feedback of what the shape will be like. This technique works for lines, circles, rectangles and axis-aligned ellipses. All of these shapes can be defined by two points (down and up).

Figure 14.1 shows a PPS specification for creating such shapes. We have our basic three inputs as well as the modifiers for the left mouse button. We have a Mode field for the drawing mode selected by the user. This field would be set by some other part of our user interface such as a menu item or palette button. The actions handle the semantics of getting objects drawn on the screen and entered into the model. The State field remembers that we are dragging out a shape. We will expand the role of this field in later sections.

```
Input {*MD,      // mouse down
       *MM,      // mouse move
       *MU}      // mouse up
LeftButton {LD*,      // left down
       LU*}      // left up
Mode { line,     //drawing a line
       rect,     // drawing a rectangle
       ellipse}  // drawing an ellipse
State { idle, dragging }
Actions { !startLine,         // save the mouse position as the line's starting point
       !moveLine,             // redraw the line from the start point to the new mouse position
       !endLine,              // add a line to the model from start point to current mouse point
       !startRect, !moveRect, !endRect,
       !startEllipse, !moveEllipse, !endEllipse }

line *MD LD* idle -> !startLine(mX,mY) dragging
line *MM LD* dragging -> !moveLine(mX,mY)
line *MU LU* dragging -> !endLine(mX,mY) idle

rect *MD LD* idle -> !startRect(mX,mY) dragging
rect *MM LD* dragging -> !moveRect (mX,mY)
rect *MU LU* dragging -> !endRect (mX,mY) idle

ellipse *MD LD* idle -> !startEllipse (mX,mY) dragging
ellipse *MM LD* dragging -> !moveEllipse (mX,mY)
ellipse *MU LU* dragging -> !endEllipse (mX,mY) idle
```

Figure 14.1 – PPS for single drag drawing

Multipoint shapes

There are a variety of shapes such as polylines, curves, polygons, closed curves and curvilinear shapes that require multiple points. The simple down, move, up sequence is not sufficient. Most systems allow the user to click on a set of points and after each click they construct the shape with the points created so far. It also becomes important to decide if each point is specified on mouse-down or mouse-up. It makes a difference because between mouse-down and mouse-up the user may move. In our PPS we will use the mouse-down convention and generally ignore mouse-up. Our rubberbanding will occur on mouse movement regardless of whether the mouse button is up or not.

In many drawing systems the question of an open or closed shape is not decided until all of the points are entered. If the last point is near the first point, then the shape is closed. If the last point is near the previous point then the shape is terminated as an open polyline or curve rather than a closed shape.

```
Input {*MD,     // mouse down
      *MM,      // mouse move
      *MU}      // mouse up
LeftButton {LD*,     // left down
      LU*}      // left up
Mode { line,    //drawing a line
      rect,     // drawing a rectangle
      ellipse,  // drawing an ellipse
      polyline, // drawing a polyline or polygon
      curve,    // drawing an open or closed curve
      }
State { idle, dragging, points }
Actions { !startLine,        // save the mouse position as the line's starting point
      !moveLine,             // redraw the line from the start point to the new mouse position
      !endLine,              // add a line to the model from start point to current mouse point
      !startRect, !moveRect, !endRect,
      !startEllipse, !moveEllipse, !endEllipse
      !startPoly, !movePoly, !clickPoly, !endPolyLine, !closePolygon,
      !startCurve, !moveCurve, !clickCurve, !endCurve, !closeCurve
      }
MouseLoc { ?nearFirstPoint,       // mouse point is near the first entered point
      ?nearPrevPoint,             // mouse point is near the previous point entered
      ?farAway }                  // not near any points

      . . .  other rules for lines, rectangles and ellipses

polyline *MD LD* idle -> !startPoly(mX, mY) points
polyline *MM points -> !movePoly(mX, mY)
polyline *MD LD* points ?farAway -> !clickPoly(mX, mY)
polyline *MD LD* points ?nearPrevPoint -> !endPolyLine idle
polyline *MD LD* points ?nearFirstPoint -> !closePolygon idle

curve *MD LD* idle -> !startCurve(mX, mY) points
curve *MM points -> !moveCurve(mX, mY)
curve *MD LD* points ?farAway -> !clickCurve(mX, mY)
curve *MD LD* points ?nearPrevPoint -> !endCurve idle
curve *MD LD* points ?nearFirstPoint -> !closeCurve idle
```

Figure 14.2 – PPS for multipoint shapes

Figure 14.2 shows a PPS that deals with all of these issues. We added new Mode choices for polylines and curves. We also added new actions for starting a list of points, moving around, clicking a new vertex point and ending an open or a closed shape. We have also added the query field MouseLoc that gives us the information about where the mouse click has occurred relative to the previous points.

Selecting shapes

Having created a variety of shapes, we also need to be able to select them so that we can change them in various ways. There are four types of selection. There is single object selection involving just a click to designate the selected object or shift clicking for multiple objects. There is rectangle selection and lastly there is lasso selection. We will address each of these in turn.

Click selection

Simple click selection involves taking the mouse point on mouse-up and comparing it to all objects in the model. Our paint() method draws object in back to front order so that the front most objects appear on top. Our selection method iterates through the objects in front to back order and returns the first object that is selected by the point.

To select a path shape such as a line, curve or edge of a closed shape, we use the distance from point to shape methods from chapter 12. To select filled shapes such as ellipses, rectangles or polygons, we use their inside/outside tests.

Since the first Apple Macintosh, most systems have used shift clicking to select multiple objects. If the shift key is down during a selection, objects are added to the selected set if they are not already in the set, or removed if already in the selected set. The methods to select from a mouse-point are the same. Figure 14.3 shows a PPS with click selection added to it. We add a select condition to the Mode field to indicate that we are selecting rather than creating. The Selection query field tells us if a shape has been hit by the selection. Clicking where there are no shapes causes all selected shapes to be unselected.

```
Input {*MD, *MM, *MU}
LeftButton {LD*, LU*}
ShiftButton {SD*,SU* }
Mode { select, line, rect, ellipse, polyline, curve}
State { idle, dragging, points }
Actions { !startLine, !moveLine, !endLine, !startRect, !moveRect, !endRect,
    !startEllipse, !moveEllipse, !endEllipse
    !startPoly, !movePoly, !clickPoly, !endPolyLine, !closePolygon,
    !startCurve, !moveCurve, !clickCurve, !endCurve, !closeCurve,
    !simpleSelect, !shiftSelect, !deselectAll
    }
MouseLoc { ?nearFirstPoint, ?nearPrevPoint, ?farAway }
Selection( ?objectSelected, ?noObjectSelected }

select *MU LU* SU* ?objectSelected(O) -> !simpleSelect(O)
select *MU LU* SD* ?objectSelected(O) -> !shiftSelect(O)
select *MU LU* ?noObjectSelected -> !deselectAll

    . . .  other rules for lines, rectangles, ellipses, polylines and curves
```

Figure 14.3 – Click and shift-click selection

Rectangle selection

One technique for selecting a group of shapes all at once is to drag out a rubberband rectangle around all of the shapes that should belong to the group. Once we have the selection rectangle, we select all shapes whose tight bounding box (not the loose convex hull bounding box) is completely inside of the selection rectangle. Using the shift key we can add or remove elements from the selected set by using clicking or rectangle selection. Some implementations of rectangle selection add any shape whose bounding box intersects the selection rectangle. This make it easier to select more objects but in practice is harder to control to get exactly the right set.

Rectangle selection is useful in the presence of hierarchic groups of objects. If the user draws a rectangle around a doorknob it is clear that the knob rather than the door or the house is being selected. This is much less ambiguous that selection by clicking. The technique is to select the largest object groups that fit completely within the selection rectangle.

Lasso selection

A problem with rectangle selection is that the set of objects to be selected may not always be selectable by a rectangle. Consider the task of selecting the black shapes and not the gray shapes in figure 14.4. There is no rectangle that can accomplish this task. The user can, however, draw a "lasso" around the desired shapes and excluding the others as shown. The lasso selection can also be augmented by shift-select techniques.

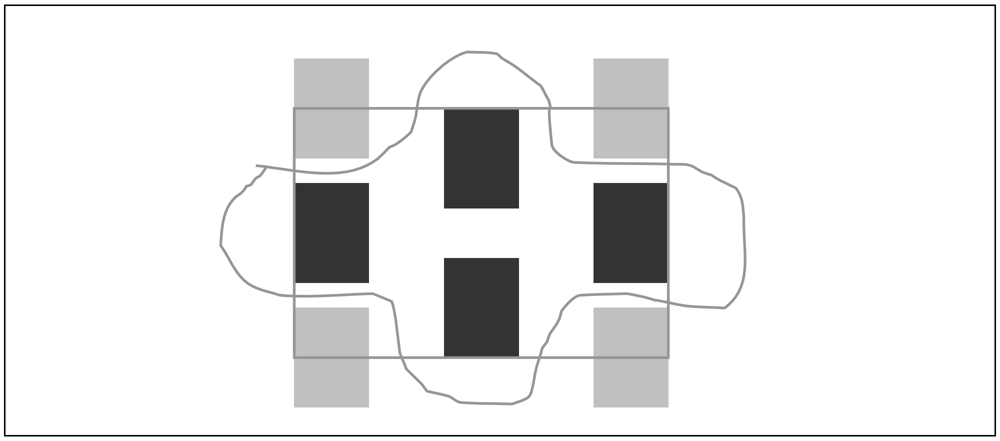

Figure 14.4 – Rectangle and Lasso selection

The selection problem is to decide which shapes are inside and outside of the lasso. Lassos are never built from cubic curves, which make the curve inside/outside techniques unusable. One simple technique for lasso selection is to ensure that the entire shape is closed by drawing lines between the mouse movement points when the lasso is drawn as well as a line from the first to last point. We draw the lasso in black into a blank (all white) image. We then perform a black flood fill using a starting point known to be inside of the lasso. This produces a mask of all pixels that are inside of the shape. This is done in any pixel coordinates, but usually screen window coordinates.

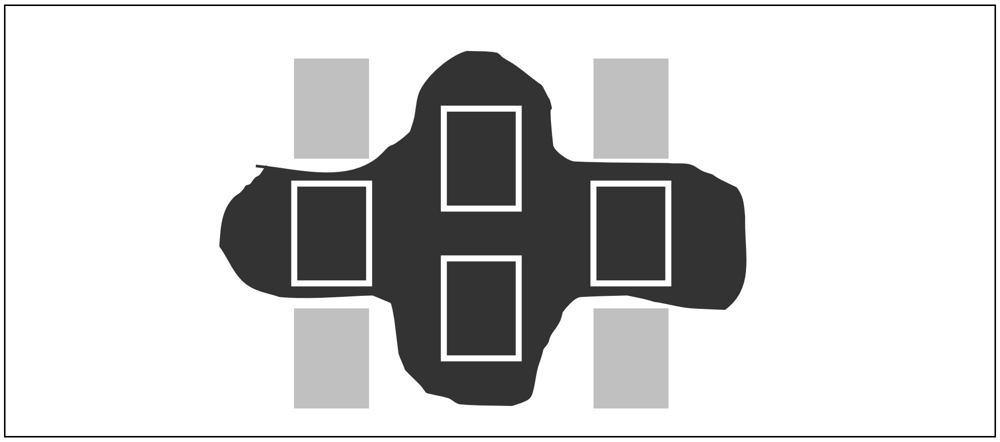

Figure 14.5 – Selection mask

Using the mask, shown in figure 14.5, we must decide which pixels are in or out. There are two techniques for this depending upon our assumptions about user selection. The fastest technique is to take the "principle points" for each shape and checking to see if they all fall on a black pixel in the mask. If they do, then the shape is selected. For lines we use the end points, for ellipses the north, east, south and west points, for curves the end points and the X/Y maxima and minima, and for rectangles we use the corner points.

Checking principle points is very fast, but relies upon the assumption that the selection is not crossing shape boundaries in arbitrary ways. In most, but not all, situations this is a good assumption. An alternative technique is to "draw" each shape into the mask image using the scan-conversion algorithm for that shape. A shape is selected if every point on the shape is drawn into a black mask pixel. For filled shapes, only the boundaries need be drawn. This pseudo-drawing technique gives an exact, pixel accurate, selection.

In most systems the user must specify whether click, rectangle or lasso selection is desired. Saund and Lank[1] have proposed that the user need not make such a distinction and that it can be inferred from the movement of the mouse.

- If the mouse goes down and then up without ever moving more than a few pixels from the down point, then it is a click.
- If the mouse goes down and then up at some distance from the down point then the selection is a rectangle defined by the down and up point at opposite corners.
- If the mouse moves some distance away from the down point but eventually the up point is near the down point, then it was a lasso selection.

The only problem with this technique is providing an appropriate echo while the mouse is moving. They propose that both the lasso and the rectangle be echoed until it is obvious from the mouse movement the approach that is intended. All of these selection techniques can be encoded in the PPS shown in figure 14.6. We now must remember the start point, what kind of echo we are doing and check where the up point is relative to the starting position.

```
Input {*MD, *MM, *MU}
LeftButton {LD*, LU*}
ShiftButton {SD*,SU* }
Mode { select, line, rect, ellipse, polyline, curve}
State { idle, dragging, points, clickSelecting, rectLassoSelecting }
Actions {  . . . drawing actions . . .
      !startSelect, !clickSelect, !shiftClickSel, !rectSelect, !shiftRectSelect,
      !lassoSelect, !shiftLassoSelect, !rectLassoEcho
      }
MouseLoc { ?nearFirstPoint, ?nearPrevPoint, ?farAway }
Selection( ?objectSelected, ?noObjectSelected }

idle select *MD LD* -> clickSelecting !startSelect(mX,mY)
clickSelecting *MM LD* ?nearFirstPoint -> clickSelecting
clickSelecting *MM LD* ?farAway -> rectLassoSelecting !rectLassoEcho
clickSelecting *MU LU* SU* -> idle !clickSelect
clickSelecting *MU LU* SD* -> idle !shiftClickSel
rectLassoSelection *MM LD* -> !rectLassoEcho
rectLassoSelection *MU LU* ?nearFirstPoint SU* -> idle !lassoSelect
rectLassoSelection *MU LU* ?nearFirstPoint SD* -> idle !shiftLassoSelect
rectLassoSelection *MU LU* ?farAway SU* -> idle !rectSelect
rectLassoSelection *MU LU* ?farAway SD* -> idle !shiftRectSelect

      . . .  other rules for lines, rectangles, ellipses, polylines and curves
```

Figure 14.6 – All selection PPS

Manipulating Control Points

Once an object or set of objects has been selected we must give the user feedback on what is selected. We must also allow the user to manipulate the object. In many systems both goals are accomplished by displaying a set of control points for the selected object(s), as shown in figure 14.17. This not only identifies the selected object but provides "handles" that the user can use to manipulate the object.

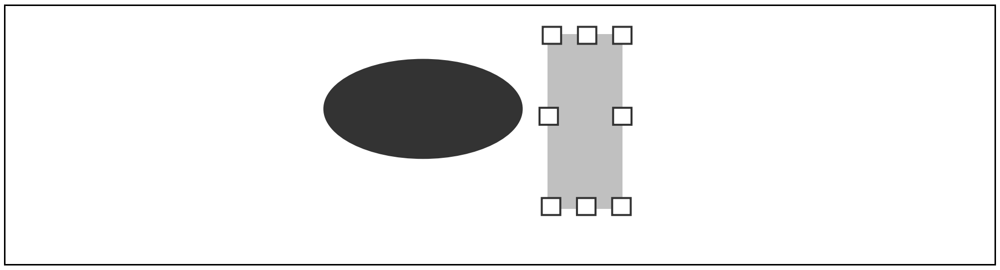

Figure 14.17 – Control points

The manipulation of control points can have a variety of meanings depending upon the shape and the kind of manipulation to be performed. The value of the control-point model is that it is a very concrete way to offer the user an opportunity to change some aspect of the shape. In terms of common look and feel most computer users believe that by dragging a control point the selected shape will be changed in some way. This provides an immediate affordance for the user to manipulate the shape.

Because the manipulation of control points is so pervasive and because we have so many meanings for what effect a control point will have, we need to use an abstract-model architecture as described in chapter 7. The abstract model will allow us to implement any meaning we desire while placing all of the control point manipulation in the widget where it can be reused in many situations. We accomplish this with the interfaces shown in figure 14.18.

The ControlPointModel can be implemented by any model. All it does is provide the widget with a list of the control points for the selected objects. It is up to the model to decide what has been selected and what control points should be provided. Based on this interface the widget can paint all of the model presentation and then ask the model for the list of control points. Using the getPoint() method on each point the widget can draw the control handles over the top of all other model information. The widget also has all of the information that it needs to allow the user to drag the point. The widget then calls setPoint() so that the model can do what it wants with the dragged control point.

```
public interface ControlPoint
{
        Point getPoint();       // return the control point location in window coordinates
        void setPoint(Point newLoc);   // set the control point location in window coordinates
}
public interface ControlPointModel
{
        ControlPoint [] getSelectedControls();
                        // return all of the control points for selected objects in the model
}
```

Figure 14.18 – Abstract model for control point interaction

In many cases a control point is constrained as to where it can be placed. For example in figure 14.17, the control point on the right edge of the rectangle should remain centered vertically between the top and bottom of the rectangle while freely moving horizontally. When the widget controller calls setPoint() for this control point, the model can actually place the control point wherever it wants. In this case the model ignores the *Y* component of the newLoc. For this to work correctly the model should perform a damage on the entire shape and the view should redraw all of the control points from the model. This keeps the presentation consistent with the model rather than with the controller. With control points on the screen we must also modify our PPS to accommodate the dragging of control points. This is shown in figure 14.19.

```
Input {*MD, *MM, *MU}
LeftButton {LD*, LU*}
ShiftButton {SD*,SU* }
Mode { select, line, rect, ellipse, polyline, curve}
State { idle, dragging, points, clickSelecting, rectLassoSelecting, controlDragging }
Actions {  . . . drawing actions . . .
                . . . select actions . . .
        !controlSelected, !moveControl
        }
MouseLoc { ?nearFirstPoint, ?nearPrevPoint, ?farAway }
Selection( ?objectSelected, ?noObjectSelected}
Control {?controlSelected, ?noControl }

idle select *MD LD* ?noControl -> clickSelecting !startSelect(mX,mY)
idle select *MD LD* ?controlSelected(C) -> controlDragging !selectControl(C)
controlDragging *MM LD* -> !moveControl
controlDragging *MU LU* -> !moveControl idle
        . . . other rules for selection
        . . .  other rules for lines, rectangles, ellipses, polylines and curves
```

Figure 14.19 – Control point dragging

Transformations

Now that we can create objects, select objects and change objects, we want to transform objects using our three basic transformations. With a generic set of interactive techniques for transformations we will have a uniform feel for a large variety of interactive behaviors.

Translation

Interactively expressing a translation is called *dragging* because we are dragging a shape around the screen. In the model, this is generally manipulated in one of three ways. The simplest is that every model object has a position that is changed as a result of dragging. The second way is that the dragging distance (dX, dY) is added to every point of the shape and the new position is encoded in the shape points. The last approach is that there is an instance transformation *T* for the object and a new transformation is computed as *T=trans(dX,dY)*T*.

What we need is to get (dX, dY) from the mouse position while dragging. One approach is to save the last mouse position *(lX, lY)* and then compute the difference between the current position and the last position. For translation this will generally work. However, for the other transformations incrementally computing the transformation on each mouse move can have inaccuracy problems due to the accumulation of round-off error. Remember that world coordinates are frequently in floating point and mouse coordinates are always in integers. To eliminate this problem we always perform our transformations relative to the mouse-down point rather than relative to the last mouse point.

Interactively we have a problem because most systems do not require the user to specify “I am going to drag a shape now”. In most systems there is only a “select” mode and the controller must distinguish between click selection, dragging an object, rectangle selection, lasso selection and control point dragging by watching what the user actually does. This complicates our controller code.

For translation we must distinguish between click selection and dragging. One of the first techniques is to perform a selection on mouse-down if the mouse is over an object. The selection must be done whether dragging is intended or not. We can put off the dragging decision until later. One may want to infer a dragging intention whenever a mouse-move is received. However, most users’ hands will shake slightly when pressing the mouse button and this may or may not cause a mouse-move event. If we move the object every time the user’s hand shakes, there will be frustration. The appropriate technique is to assume clicking until the mouse is no longer near the down point and then assume dragging.

```
Input {*MD, *MM, *MU}
LeftButton {LD*, LU*}
ShiftButton {SD*,SU* }
Mode { select, line, rect, ellipse, polyline, curve}
State { idle, dragging, points, clickSelecting,
        rectLassoSelecting, controlDragging, objDragging }
Actions {  . . . drawing actions . . .
       !startSelect, !clickSelect, !shiftClickSel, !rectSelect, !shiftRectSelect,
       !lassoSelect, !shiftLassoSelect, !rectLassoEcho,
       !controlSelected, !moveControl, !moveSelected, !setSelected
       }
MouseLoc { ?nearFirstPoint, ?nearPrevPoint, ?farAway }
Selection( ?objectSelected, ?noObjectSelected}
Control {?controlSelected, ?noControl }

select idle *MD LD* ?noObjectSelected ?noControl -> rectLassoSelecting !startSelect
select idle *MD LD* ?controlSelected -> controlDragging !controlSelected
select idle *MD LD* SU* ?objectSelected ?noControlSelected -> clickSelecting
              !clickSelect  !startSelect
select idle *MD LD* SD* ?objectSelected ?noControlSelected -> clickSelecting
              !shiftClickSelect !startSelect

clickSelecting *MM LD* ?nearFirstPoint -> clickSelecting
clickSelecting *MM LD* ?farAway -> objDragging !moveObj
clickSelecting *MU LU* SU* -> !clickSelect idle
clickSelecting *MU LU* SD* -> !shiftClickSelect idle

rectLassoSelecting *MM LD* -> !rectLassoEcho
rectLassoSelecting *MU LU* SU* ?nearFirstPoint -> !lassoSelect idle
rectLassoSelecting *MU LU* SD* ?nearFirstPoint -> !shiftlassoSelect idle
rectLassoSelecting *MU LU* SU* ?farAway -> !rectSelect idle
rectLassoSelecting *MU LU* SD* ?farAway -> !shiftRectSelect idle

controlDragging *MM LD* -> !moveControl
controlDragging *MU LU* -> !moveControl idle

objDragging *MM LD* -> !moveSelected
objDragging *MU LU* -> !setSelected idle
       . . .  other rules for lines, rectangles, ellipses, polylines and curves
```

Figure 14.20 – Full PPS for select and drag

In figure 14.20 we show the PPS that brings together all of the elements of selection and dragging. The reason this has become so complex is because we desire the user interface to behave naturally and smoothly to the user with a minimum amount of extra button clicking. When the mouse first goes down we must consider whether a control point, an object or nothing is under the mouse and act accordingly. If the mouse is over an object then we must perform the selection or shift selection on mouse-down so that if we are dragging, the appropriate objects are selected for dragging. The PPS assumes that we are clicking until the mouse moves away from the start point and then we switch to the objDragging state. Once in the objDragging state, the user can move the object back very close to the original point. This technique accommodates hand tremor on clicking while providing for very small movements when desired.

If we add the actions and rules for shape drawing back into figure 14.20 we have a complete specification of our controller for interactive manipulating geometry. The remaining techniques in this chapter all rely upon control point manipulation. The resulting PPS fits on a single page and can be studied, discussed and modified. From this we can carefully derive the code for the mouseDown(), mouseMove() and mouseUp() event methods. Notice in figure 14.20 we have organized the rules according to the tasks that we are trying to accomplish. This helps us think about what we are trying to do while designing. However, the implementation is organized around event methods and we must now use our PPS techniques from chapter 11 to derive an implementation equivalent to our design.

Scaling

We use the scaling transformation to change the size of objects. The scaling transformation always functions relative to the origin and relative to the *X* and *Y* axes. We learned in chapter 13 how to scale relative to any point by translating that point to the origin first. We can also scale relative to any axis by rotating that axis to the *X* axis first. What we want is to assemble this geometry together so that we can express scaling using dragging of a few control points. We will discuss three such interactive techniques: rectangle scaling, point scaling and point-axis scaling.

The most common scaling technique is to bound a shape with a rectangle and use the 8 rectangle control points to express a scaling transformation. This is shown in figure 14.21.

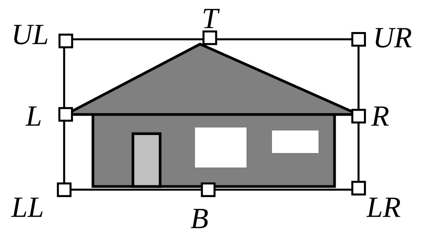

Figure 14.21 – Rectangle Scaling Controls

For each control point in figure 14.21 the opposite control point forms its center of scaling. So for example if the user grabbed the point *UR* and dragged it to a new position *NUR*, the transformation sequence would be that shown in figure 14.22. If the user had grabbed *T* and dragged it to *NT* then the transformation sequence would be that shown in figure 14.23. All of the other control points have similar transformation sequences. There are three main advantages to rectangle scaling: 1) the control points naturally follow selection of the object without introducing any new menu modes, 2) the control points form a natural affordance, and 3) the center of scaling is automatically inferred.

$$\boxed{trans(LL_x, LL_y) \bullet scale\left(\frac{NUR_x - LL_x}{UR_x - LL_x}, \frac{NUR_y - LL_y}{UR_y - LL_y}\right) \bullet trans(-LL_x, -LL_y)}$$

Figure 14.22 – Scaling by dragging *UR* to *NUR*

$$\boxed{trans(B_x, B_y) \bullet scale\left(1, \frac{NT_y - B_y}{T_y - B_y}\right) \bullet trans(-B_x, -B_y)}$$

Figure 14.23 – Scaling by dragging *T* to *NT*

Sometimes we want to grab a corner control point and scale the object, but we do not want to change the object's aspect ratio. We want it bigger or smaller, but still the same shape. For this we need a transformation that is uniform in *X* and *Y*. Such a transformation is shown in figure 14.24. We chose the axis that has the most change and scale both axes by that amount.

$$S = \begin{cases} \left|(NUR_x - UR_x) > (NUR_y - UR_y)\right| \Rightarrow \dfrac{NUR_x - LL_x}{UR_x - LL_x} \\[2ex] \left|(NUR_x - UR_x) \le (NUR_y - UR_y)\right| \Rightarrow \dfrac{NUR_y - LL_y}{UR_y - LL_y} \end{cases}$$

$$trans(LL_x, LL_y) \bullet scale(S, S) \bullet trans(-LL_x, -LL_y)$$

Figure 14.24 – Uniform scaling by dragging *UR* to *NUR*

Suppose in figure 14.21 we want to scale our home so that the lower left corner of the foundation remains in location but the right roof/eve point is scaled to a particular position. This is not very realistic for houses but does happen in other situations. To accomplish this task we want the lower left foundation point rather than the point *LL* to stay fixed. For this task we need three points: the center of scaling *C*, a current point *P* on the object, and a new point *NP* for where *P* should end up. All of our dragging techniques only give us two points. Generally, arbitrary scaling is performed by a special “scaling” mode of the interface rather than integrated with the selection. When a user selects “Scale” they then click on the center of scaling point followed by dragging point *P* to point *NP*. The resulting transformation sequence is shown in figure 14.25.

$$\boxed{trans(C_x, C_y) \bullet scale\left(\frac{NP_x - C_x}{P_x - C_x}, \frac{NP_y - C_y}{P_y - C_y}\right) \bullet trans(-C_x, -C_y)}$$

Figure 14.25 – Scaling relative to arbitrary points

These two scaling techniques still defined relative to the *X* and *Y* axes. Suppose we have the house on the left in figure 14.25 and we want to make it wider but not taller.

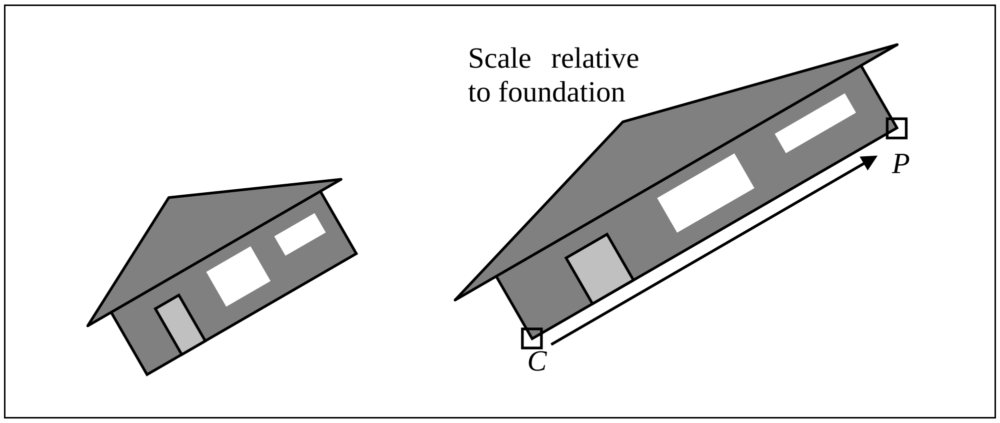

Figure 14.25 – Scaling on a new axis

In this situation we want scaling only along a particular axis, but the axis is not parallel to the *X* or *Y* axis. We solve this problem by rotating to the *X* axis before scaling and then putting it back. The challenge is computing the angle of rotation. However, remember that we do not need the angle of rotation. What we need are the sine and cosine of that angle. We can compute that directly from points *P* and *C*. Note that we first want to rotate by the negative of the angle to get it to the *X* axis. Remember that *sin(-A)=-sin(A)* and that *cos(-A)=cos(A)*. This process is complicated by the fact that we want to only scale along the axis *C->P*, but our user will wiggle the point *NP* around off of that axis. Therefore we need to transform *P* and *NP* into the new rotated coordinates *P'* and *NP'* before we can compute the scale factor. The transformation matrix *T* that will perform all of this is shown in figure 14.26. The steps are shown visually in figure 14.27.

$$H = \sqrt{(P_x - C_x)^2 + (P_y - C_y)^2}$$

$$SIN = \frac{P_y - C_y}{H} \qquad COS = \frac{P_x - C_x}{H}$$

$$M = rot(-SIN, COS) \bullet trans(-C_x, -C_y)$$

$$P' = M \bullet P \qquad NP' = M \bullet NP$$

$$S = \frac{NP'_x}{P'_x}$$

$$T = M^{-1} \bullet scale(S,1) \bullet M$$

Figure 14.26 – Scaling of *C->P* to *NP*

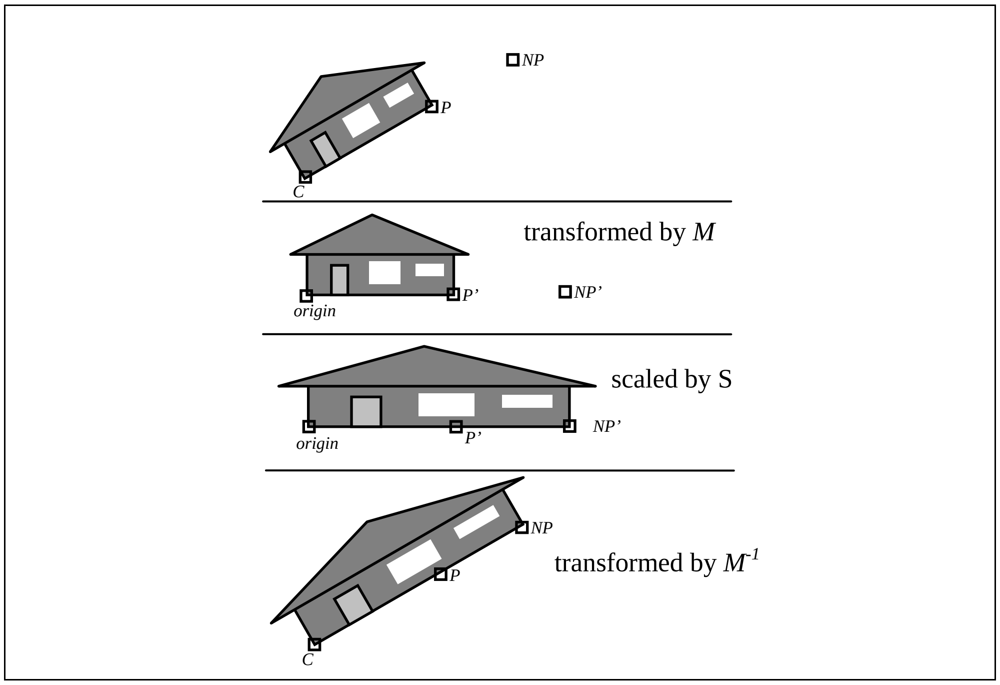

Figure 14.27 – Transformation steps scaling *C->P* to *NP*

Rotation

Rotation, like scaling, needs a center of rotation. Thus we will need three points to perform a rotation (*C*, *P* and *NP*). When the user specifies rotation mode they then select the object to rotate. We can get the center of rotation from the selection point or from the "center of mass" of the shape. The center of mass is usually calculated as the average of all the points that define the shape. This tends to place the center of rotation near the center of the shape. In some systems this is displayed as a special control point as shown in figure 14.28. Grabbing any point *P* on the shape the user can specify a rotation by dragging to *NP*. The user can also drag point *C* to change the center of rotation.

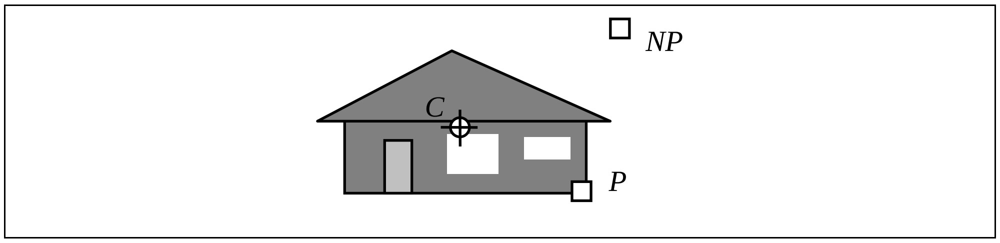

Figure 14.28 – Rotation with 3 points

To perform this rotation we rotate point *P* to the *X* axis and then rotate the *X* axis to point *NP*. The transformation matrix *T* is derived in figure 14.29.

$$HP = \sqrt{(P_x - C_x)^2 + (P_y - C_y)^2}$$

$$SINP = \frac{(P_y - C_y)}{HP} \qquad COSP = \frac{(P_x - C_x)}{HP}$$

$$HNP = \sqrt{(NP_x - C_x)^2 + (NP_y - C_y)^2}$$

$$SINNP = \frac{(NP_y - C_y)}{HNP} \qquad COSP = \frac{(NP_x - C_x)}{HNP}$$

$$T = trans(C_x, C_y) \bullet rot(SINNP, COSNP) \bullet rot(-SINP, COSP) \bullet trans(-C_x, -C_y)$$

Figure 14.29 – Rotating from *P* to *NP* about *C*

9 Point transformation frame

We can interactively accomplish almost all of these transformations using the 9 control points shown in figure 14.30. In this formulation the object has a natural center that we use as a center of rotation. The object also has a current rotation. As shown, the bounding box frame for scaling is oriented according to the rotation angle. This allows scaling to be expressed relative to the object's natural axes rather than the X and Y axes. The center of rotation is shown. The extra handle extending above the bounding box is the rotation handle.

All transformations are performed by dragging. Dragging any point on the object will translate the object. Dragging any of the 8 control points on the rectangular frame will scale the object relative to the frame. Dragging the rotation handle will rotate the object about its center. Dragging the center point will change the center of rotation.

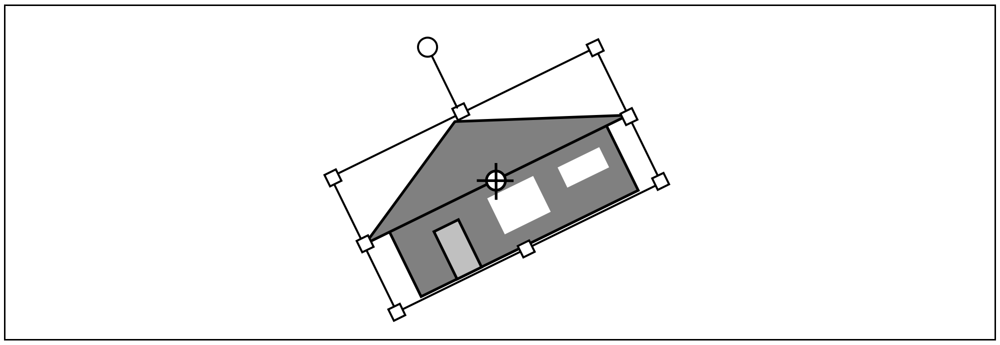

Figure 14.30 – 9 Point transformation frame

[1] Saund, E. and Lank, E., "Stylus Input and Editing Without Prior Selection of Mode", *ACM Symposium on User Interface Software and Technology (UIST '03)*, ACM, (Oct 2003), pp. 213-216.