Contact me in order to access the whole complete document. Email: solution9159@gmail.com
WhatsApp: https://wa.me/message/2H3BV2L5TTSUF1 Telegram: https://t.me/solutionmanual

Chapter 1S

Introduction

There are no written exercises for this chapter.

Chapter 2S

Introduction to 2D Graphics using WPF

There are no written exercises for this chapter.

Chapter 3S

Chapter 3 solutions

Inline Exercise 3.1S: Imagine that you can move the lute in Dürer's woodcut. (a) How could you move it to ensure that "touches boundary of frame" is true for each iteration of the inner loop, thus resulting in almost all the work of the algorithm (aside from setup costs) falling into that clause? (b) How would you move it to ensure that no work fell into that clause? (Your answers should be of the form "Move it closer to the frame and lift it up a little," i.e., they should describe motions of the lute within the room.)

Solution:

(a) Move the lute far to the right or left.

(b) Move it far from the eyebolt; have a friend stand with his eye near the eyebolt and verify that the whole lute is visible within the bounds of the frame.

Inline Exercise 3.2S: Imagine that instead of moving the pointer at the end of the string to points of the lute and then seeing where they pass through the paper, that the two men start with a piece of graph paper. For each square on the graph paper, the one with a pencil holds it at the center of the square; the "frame" is then opened, and the man with the pointer moves it so that the string passes by the pencil point, *and* so that the end of the string is touching either the lute, or the table, or the wall, etc. The object being touched is noted, and when the frame is closed again, the man with the pencil fills in the grid-square with some gray-level, corresponding to how the touched point looks: if it appears dark, the grid-square is filled in darkly. If it's light, the grid square is left empty. If it's partly dark, the grid-square is partly shaded in. What sort of picture results? This approach (working "per pixel" and finding out what's seen through that pixel) is the essence of raytracing, as discussed in Chapter 14. A slightly different version of it, also developed by Dürer, is shown in Figure 3.2: the graph paper is on the table; the corresponding grid in the frame consists of wires stretched across the frame, or semi-transparent graph paper, and the string-and-pointer are replaced by the line-of-sight through a small hole in front of the artist.

Solution:

The resulting picture is pixellated, like a checkerboard. Straight lines between dark and light areas in the scene become staircase-like patterns in the picture unless they happen to be horizontal or vertical. Small objects in the scene may not appear in the picture, because no ray through a pixel-center happens to hit them.

Inline Exercise 3.3S: What are the coordinates of the other two corners of the frame?

Solution:

They are $(x_{\text{min}}, y_{\text{max}}, 1)$ and $(x_{\text{max}}, y_{\text{min}}, 1)$.

Inline Exercise 3.4S: The previous paragraph suggests that we can summarize by saying "perspective projection from space to a plane takes lines to lines" or "takes line segments to line segments." Think carefully about the first claim and find a counterexample. Hint: is our perspective projection defined for every point in space?

Solution:

If we're performing perspective projection from an eye-point E to a plane P not containing E, then any line ℓ that passes through E will generally project to either a single point of P (namely $\ell \cap P$) or to nothing at all, if ℓ is perpendicular to the normal vector for the plane P.

If we say that P' is the plane parallel to P but containing E, then for any line m that passes through P' but does not contain E, a finite segment s of m that meets P' will generally project to a pair of rays in P, with the intersection point $s \cap P'$ being sent to infinity.

Inline Exercise 3.5S: Verify, in Listing 3.5, that a vertex that happens to be located at the lower left corner of the view square, i.e., $(xmin, ymin, 1)$, does indeed transform to the lower left corner of the final picture, i.e., the corresponding *pictureVertex* is $(0,1)$; similarly verify that $(xmax, ymax, 1)$ transforms to $(1,0)$.

Solution:

Suppose `vertex[0]` has coordinate $(xmin, ymin, 1)$. In lines 7 and 8, this makes `x` and `y` be $xmin/1$ and $ymin/1$, respectively. Thus in line 10, `x - xmin` is zero, as is `y - ymin`, so the new `PictureVertex` entry is initialized with `Point(1 - 0, 0)`, which is the point $(1,0)$.

For the point `xmax, ymax`, we have each of the fractions in lines 10 and 11 turning into 1, so that the initializer is `Point(1-1, 1) = Point(0, 1)`.

Exercise 3.1S: Suppose that the apprentice holding the pencil in the Dürer drawing not only made a mark, but also wrote next to it the height (above the floor) of the weight at the end of the string. That number would be the distance from the eye to the object (plus some constant). Now suppose we make a first drawing-with-distances of a lute, whose owner then takes it home. We later decide that we'd really like to have a candlestick in the picture, in front of the lute (i.e., closer to the eye). (a) If we place the candlestick on the now-empty table, but in the correct position, and make a second drawing-with-distances, describe how one might algorithmically combine the two to make a composite drawing showing the candlestick in front of the lute. This idea of "depth compositing" is one of many applications of the z-buffer, which you'll encounter again in Chapters 14, 32, 36 and 38. The recording of depths at each point is rather like the values stored in a z-buffer, although not identical. (b) Try to think of other things you might be able to do if each point of an image were annotated with its distance from the viewpoint.

Solution:

(a) At each point of the two drawings-with-distances, copy, to a new drawing, the one whose distance is smaller. Since the only points that have distances are those corresponding to marks, you have to make a guess at the distance for unmarked points, perhaps by interpolating nearby distances. Thus, for instance, a point of the lute that happens to fall in the image at the same location as the middle of the candlestick will be left undrawn in the composite image, because the depth for the mid-candlestick location (in the candlestick image) can be estimated from the known depths of the nearby edges of the candle.

(b) You might be able to make a new image with some degree of focal blur, which arises in cameras that use a lens rather than a pinhole. If you decided to imitate a camera whose lens was focused at 2 m, you could then blur each mark whose distance was *not* 2 m by an amount that depended on how far the mark's depth was from the two-meter focal plane.

Exercise 3.2S: The dots at the corners of the rendered cube in Figure 3.8 appear behind the edges, which doesn't look all that natural; alter the program to draw the dots *after* the segments so that it looks better. Alter it again to not draw the dots at all, and draw only the segments.

Solution:
This is a purely programming exercise.

Exercise 3.3S: We can represent a shape by faces instead of edges; the cube in the Dürer program, for instance, might be represented by 6 square faces rather than its 12 edges. We could then choose to draw a face only if it faces towards the eye. "Drawing," in this case, might consist of just drawing the edges of the face. The result is a rendering of a wire-frame object, but with only the visible faces shown. If the object is convex, the rendering is correct; if it's not, then one face may partly obscure another. For a convex shape like a cube, with the property that the first two edges of any face are not parallel, it's fairly easy to determine whether a face with vertices $(P_0, P_1, P_2, \ldots)$ is visible: you compute the cross-product[1] $\mathbf{w} = (P_2 - P_1) \times (P_1 - P_0)$ of the vectors, and compare it to the vector $\mathbf{v}$ from the eye, E, to P_0, i.e., $\mathbf{v} = P_0 - E$. If the dot product of $\mathbf{v}$ and $\mathbf{w}$ is negative, the face is visible. This rule relies on ordering the vertices of each face so that the cross-product $\mathbf{w}$ is a vector that, if it were placed at the face's center, would point into free space rather than into the object. (a) Write down a list of faces for the cube, being careful to order them so that the computed "normal vector" $\mathbf{w}$ for each face points outwards. (b) Adjust your program to compute visibility for each face, and draw only the visible faces. (c) This approach fails for more complex shapes, but later in the book we'll see more sophisticated methods for determining whether a face is visible. For now, give an example of an eye-point, a shape. and a face of that shape with the property that (i) the dot product of $\mathbf{v}$ and $\mathbf{w}$ is negative, and (ii) the face is not visible from the eye. You may describe the shape informally. Hint: your object will have to be non-convex!

Solution:
(a) 1562, 5476, 4037, 0123, 2673, 0451.
(b) This is a purely programming exercise.
(c) In the xy-plane, draw a polygon with vertices $(0,0)$, $(1,1)$, $(-1,0)$, $(1,-1)$, and $(0,0)$. Extrude this polygon along the z-axis to form a surface. Place the eye at $(0,-10)$. The extrusion of the edge from $(0,0)$ to $(1,1)$ has a normal that's in the

opposite half-plane from the view vector, but nonetheless that face is not visible — it's obscured by the extrusion of the edge from $(-1,0)$ to $(1,-1)$.

Exercise 3.4S: As in the previous exercise, we can alter a wire-frame drawing to indicate front and back objects in other ways. We can, for instance, consider all the lines of the object (the edges of the cube, in our example) and sort them from back to front. If two line segments do not cross (as seen from the viewpoint) then we can draw them in either order. If they *do* cross, we draw the one further from the eye first. Furthermore, to draw a line segment (in black on a white background), we first draw a thicker version of the line segment in white, and then the segment itself in black at ordinary thickness. The result is that nearer lines "cross over and hide" farther lines. (a) Draw an example of this on paper, using an eraser to simulate laying down the wide white strip. (b) Think about how the lines will appear at their endpoints — will the white strips cause problems? (c) Suppose two lines meet at a vertex but not at any other point. Does the order in which they're drawn matter? This "haloed line" approach to creating wire-frame images that indicate depth was used in early graphics systems, when drawing filled polygons was slow and expensive, and even later to help show internal structures of objects.

Solution:

(a) Figure 3.1S shows an example of the original line drawing and the result of erasing around the line as you draw it.

(b) If one line meets another in the middle, as in the letter "y", the white strip for the short segment may obscure part of the long segment.

(c) Yes, the white strips will cause problems — at a sharp corner the centerline of one segment may be obscured by the white strip for the next segment, unless the white strips are "beveled" at the corners, which can lead to its own problems.

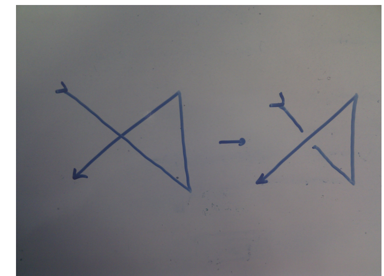

Figure 3.1S: A haloed line drawing

fig:3-4

Exercise 3.5S: Create several simple models, such as a triangular prism, a tetrahedron, and a $1 \times 2 \times 3$ box, and experiment with them in the rendering program.

Solution:

This is a purely programming exercise.

Exercise 3.6S: Enhance the program by adding a "Load model" button, which opens a file-loading dialog, lets the user pick a model file, loads that model, and makes a picture of it.

Solution:

This is a purely programming exercise.

Exercise 3.7S: Implement the suggestion about displaying a rotating cube in the program. Add a button that, when the cube is loaded, can update the locations of the cube vertices by computing them with a new value of t, the amount to rotate. To make the animation look smooth, try changing t by $.05$ radians per button-press.

Solution:

This is a purely programming exercise.

Chapter 4S

A 2D graphics testbed

There are no written exercises for this chapter.

Chapter 5S

An introduction to human visual perception

There are no written exercises for this chapter.

Chapter 6S

Introduction to Fixed-Function 3D Graphics and Hierarchical Modeling

Most of the exercises for this chapter are invitations for you to perform guided activities using a particular module of this chapter's associated software laboratory; they generally pose no particular question and thus do not have an associated solution presented here.

Inline exercise 6.9 refers to exercises that are presented in the online resources, and the associated solutions are presented in that same online-resource document, not here.

Let's now consider inline exercise 6.15:

Inline Exercise 6.1S: 6.15: How can we make the caravan's movement across the desert scalable? Is there a scene graph that would allow a single instance trans- form to move the entire caravan, without the loss of the scalable knee/hip con- trol we just designed? What loss in realism would occur with such a plan?

Consider the scene graph shown in Figure 6.52. By applying a single modeling transform to the node representing the caravan, we can ensure the movement of the caravan across the scene. But here again, this simplicity produces an unnatural end result, all of the camels moving in perfect unison, with the distance between any two pair of camels being perfectly constant. How can we achieve more realism in effecting the movement of the caravan?

One quite-scalable solution is to effect random variations on the relative position of each camel in the caravan. Let's take a closer look at this strategy. The sim-

plest way to form the caravan is to apply distinct (but constant) instance transforms to each instantiation of the reuseable camel model. For example, this pseudocode creates three camels that form a line parallel to the X axis:

```
1 Create a Caravan node, composed of:
2    Instantiation of Camel with modeling transform: translate
         (1500, 0, 3000)
3    Instantiation of Camel with modeling transform: translate
         (3000, 0, 3000)
4    Instantiation of Camel with modeling transform: translate
         (4500, 0, 3000)
```

We can effect the movement of this caravan by animating an instance transform on the Caravan node, but the three camels' movement will be unnatural because of their constant Z value in the local coordinate system of the caravan. That problem can be resolved by adding random 2D "error transforms" to the three instance transforms. In pseudocode, this new strategy might look like this:

```
1 Create a Caravan node, composed of:
2    Instantiation of Camel with modeling transform group:
3           1) translate(1500, 0, 3000)
4           2) additionally randomly translate by at most
               (+/-150, 0, +/- 85)
5    ...
```

Here, we are requesting that imprecision to be added to the instance transform for the first camel, on both the X and Z axes. (We don't want our camels to appear to have the benefit of flight skills, so we are requesting the Y value stay constant.)

Obviously, the error transform must be expressed as a function of time, to ensure the camel is appearing to simply "meander" from its optimal path. Let's simplify the problem and worry only about applying an error to the camel's X position. Let's have our "perfectly drunken" camel wander on the X axis in a perfect sine wave having period T and peak amplitude of (as specified for camel number 1) 150 units:

```
1 At t=0, the error is X=0
2 At t=T/4, the error is X=150
3 At t=T/2, the error is X=0
4 At t=3T/4, the error is X=-150
5    etc.
```

Similarly, for the Z axis, we could establish another sinusoidal error transform with a different period and peak amplitude. In tandem, the two error transforms establish a fairly natural "drunken" meandering of the camel from its "optimum" location within the caravan.

Multiply this across the entire caravan – with each camel meandering on two axes using sine waves with randomly-generated periods and peak amplitudes. The end result will have a bit more naturalism in its caravan movement.

Even more "randomness" can be effected by adding slight errors to the orientation (rotation) of each camel as well, or by using less predictable parametric curves (e.g., Bézier curves).