BASIC CONCEPTS FOR COMPUTER GRAPHICS

AMOS JOHNSON

Expand Knowledge Publications
www.expandknowledge.net

The programs in this book are included for their instructional value. They have been tested with care but are not guaranteed for any particular purpose. The author does not offer any warranties or representations, nor does he accept any liabilities with respect to the programs.

This book is dedicated to the Glory of God

ABOUT THE AUTHOR

Amos Y. Johnson, Jr. was born in 1973 in Augusta, Georgia. He received his B.S. in General Science from Morehouse College (1996), and received another B.S. degree in Electrical Engineering from Georgia Institute of Technology (1996). He earned a M.S. degree in Electrical and Computer Engineering (1998) from Georgia Institute of Technology, and his Ph.D. in Electrical Engineering from Georgia Institute of Technology (2002). Currently he is an Assistant Professor at Morehouse College in the Computer Science Department. He has published papers in the fields of computer vision and artificial intelligence with a particular slant toward automatic visual surveillance.

PREFACE

Computer graphics may be defined as: the creation and display of pictures with a computer, the process to produce images, generation of images of virtual scenes, drawings on a computer screen, etc ... Any of these definitions will do; however, in this book, computer graphics is defined as *algorithms to construct images*. An algorithm is just a set of steps to do something; like a recipe.

Computer graphics unites math, science, programming, and creativity for a *real purpose*: constructing images. For example, to create line drawings, the equation of a line must be understood and manipulated – see Figure 1(a) – and to produce images with radiant colors, the science of how light reflects off surfaces must be grasped – see Figure 1(b). After an algorithm is designed to construct an image, a computer program is developed to implement the algorithm. Underlining this whole process is your creativity to envision an image and your ability to create algorithms to bring the image from your mind to a computer screen.

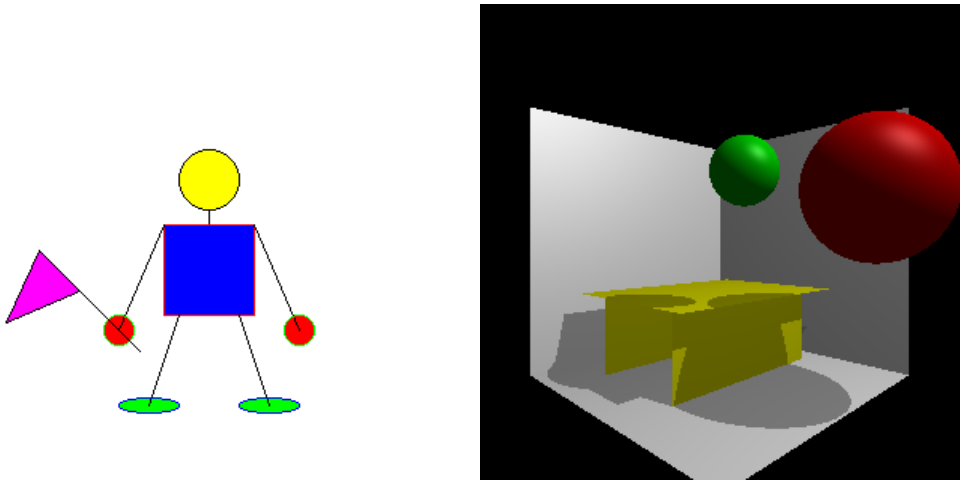
This book is an introduction to computer graphics and can be done within one semester. The basic concepts of computer graphics are taught from the ground up instead of starting with a high-level graphics package. However in practice you will work with a high-level graphics package to create images, but one should know the fundamentals before moving to a high-level graphics package. If you don't know the fundamentals, you are held hostage to the high-level graphics package because you will only be able to do what the package allows you to do. If you know the fundamentals, you can do a lot more than the package allows.

Prerequisites

This book can be used without any background in computer graphics and with limited knowledge in science, but basic programming and math knowledge is needed. [1]

Throughout the chapters, algorithms are developed to create basic images. You must implement these algorithms (*with a programming language of your choice*). I believe a great language to use is Python. In Appendix A the basic of programming is discussed and a brief introduction to python is given. Also

[1] Algebra, Geometry, Trigonometry, and Linear Algebra



(a) (b)

Figure 1. Image Examples (a)Line Drawing (b) Color Drawing

Appendix B gives the programming code (implemented in Python) for the first chapter. The programming code is given to get you started programming.

Chapter Overview

There are four parts to this book: Introduction to image formation (chapter 1), 2D graphics (chapters 2 and 3), 3D graphics (chapters 4, 5, and 6), and Animation (chapter 7).

Chapter 1: Image Basics defines images based on their structure and color content, and describes a method to store the structural and color information of an image.

Chapter 2: 2D Primitives begins the focus on 2D graphics by defining algorithms to draw 2D primitives such as lines, circles, ellipses, and polygons. In addition, algorithms to fill these primitives with color is given.

Chapter 3: 2D Line Transformations presents three general 2D transformations (translation, rotation, and scale) for images composed of lines.

Chapter 4: Viewing 3D Environments begins the focus on 3D graphics by discussing the transformations needed to display a 3D wire-frame environment on a 2D image grid.

Chapter 5: Lighting 3D Environments introduces solid 3D environments and processes needed to illuminate and display them on a 2D image grid.

Chapter 6: 3D Object Transformations defines how to transform 3D objects by translation, rotation, and scale.

Chapter 7: Computer Animation moves beyond computer graphics and introduces animation based on 2D and 3D graphics concepts.

Teacher's Guide

This course is design to be taught in one semester and the focus is on quality rather than quantity. Therefore, a student should be able to finish this entire book and write computer programs for all programming exercises. The concepts and computer programs build upon each other, so the chapters must be taught in order. For example, a student will not be able to complete a programming exercise from Chapter 6, if they have not completed a programming exercise from Chapter 2. When a student is finish with this book, they will have enough code to create their own 2D and 3D graphics engine.

Since this book is design to be a basic book on the concepts of computer graphics, I have decided not to include transformation matrices in this book. Instead, transformation matrices are replaced with there equivalent set of equations. However, since this book is focused on quality rather than quantity, you may teach the course faster than the students can program all the exercises and thus may have some extra time at the end of the semester. During this time, it may be good to introduce the concepts of transformation matrices and relate it back to the set of equations used previously – see Appendix C.

Chapter 7 is considered an extra chapter, but must be taught at the end of the course because it builds on the concepts of Chapters 1 to 6. After teaching this course, I have found that the next logical question students ask is "what about animation." Students are curious about how to animate their creations. Therefore, this chapter is included to satisfy their curiosity. This chapter is only an introduction to animation, but it gives students a taste of what can be done with computer graphics.

Lastly, as the teacher, you may obtain the programming code (in Python) for all the programming exercises by contacting the publisher, Expand Knowledge, by email at info@expandknowledge.net.

Concepts

Here are lists of math concepts and science concepts used to create algorithms to construct graphical images. In addition, a list of programming concepts needed to implement the algorithms is included.

Math Concepts

01. Equations
02. XY-Coordinate System
03. Line Equation
04. Circle Equation
05. Ellipse Equation
06. Circle Symmetry
07. Ellipse Symmetry
08. Polygons
09. Solving Equations
10. Line Intersections
11. Sine/Cosine Functions
12. Right Triangles
13. Double Angle Formulas
14. Equation Substitution
15. XYZ-Coord. System
16. Multi. Coord. Systems
17. Planes
18. Projections
19. Similar Triangles
20. Vectors
21. Rotating Coordinate Systems
22. Spheres
23. 3D Polygons
24. Plane Equation
25. Normal Vector
26. Cross Product
27. Dot Product
28. Parametric Line-Equation
29. 3D Lines
30. Sphere/3D-Line Intersections
31. Quadratic Formula
32. Real and Imaginary Numbers
33. 3D Polygons/3D-Line Intersections
34. Distance between Points
35. Magnitude of a Vector
36. Cosine of the Angle between Vectors

Science Concepts

01. Light
02. Visible Light Spectrum
03. Primary Colors
04. Ambient Light
05. Diffuse Reflections
06. Specular Reflections

Programming Concepts

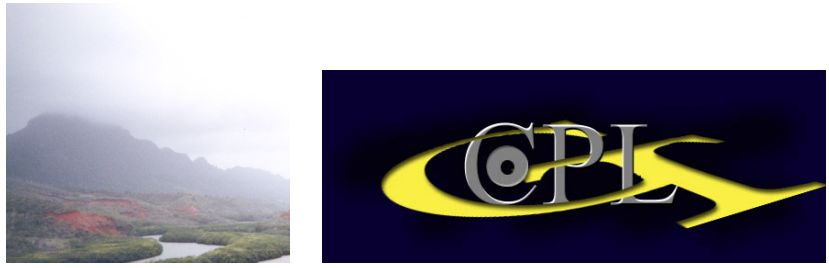
---

01. Writing to Files	
02. Variables	
03. Data Types →	integers, floats, char, strings
04. if-else Statements	
05. For loops	
06. While loops	
07. Functions	
08. Combining Functions	
09. Passing data between functions	
10. Data Structures →	list, arrays, etc...
11. Pointers	
12. Copying Memory location contents	
13. Importing Functions	
14. Counters	
15. Integer vs. Floating point division	
16. Using Comments	
17. Debugging	

TABLE OF CONTENTS

I IMAGE BASICS

IMAGES are pictures of natural or real objects or computer generated objects. Figure 1.1(a) is an example of a natural environment captured with a camera, Figure 1.1(b) is an example of a computer generated image created by an artist. Either way, all images have a basic structure. In this chapter, images are defined based on their structure and color content. Finally, a method to store the structural and color information of an image is given.

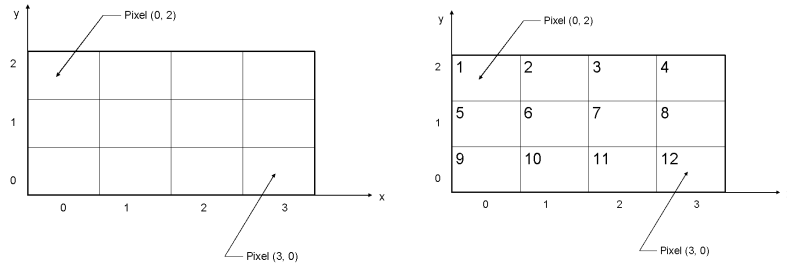


(a) (b)

Figure 1.1. Example images (a) Natural image taken from a camera (b) Computer generated images created by an artist.

1.1 Image Structure

An image is a rectangular grid of cells called pixels. Each pixel on the grid has color information, which identifies the color of the pixel. An image is characterized by the size of the grid and the color information per pixel.



(a) (b)

Figure 1.2. (a) 4x3 Image (b) 4x3 Image with pixel numbers

Figure 1.2(a) is an example of a 4x3 image with an x-dimension (x_d) of 4 and y-dimension (y_d) of 3. It has 12 pixels and each pixel has a location in the XY coordinate system of the grid. For example, location (0, 2) is the pixel in the top left corner and location (3, 0) is the pixel in the bottom right corner. Location (0, 2) is pixel number 1 and location (3, 0) is pixel number 12. Pixel numbers 1 to 12 are numbered left to right and row to row, starting at the beginning of each row. For example, location (0, 0) is pixel number 9 (see Figure 1.2(a)). The equation

$$I = x + x_d(y_d - y - 1) + 1 \tag{1.1}$$

is an index translation equation that takes an x-y location and converts it to a pixel number. For example, location (3, 0) using this equation is

$$I = 3 + 4(3 - 0 - 1) + 1 = 12,$$

which gives us pixel number 12 as expected.

1.2 Color Information

Each pixel in the grid contains color information, but what is color? However, before this question is addressed, let's talk about Light. Light is an electromagnetic wave; electromagnetic waves exist over many frequencies called the electromagnetic spectrum. The entire spectrum is not visible to the human eye; however, a small part of the spectrum called the visible light spectrum can be seen by people.

The visible light spectrum is continuous (see Figure 1.3(a)); if the spectrum is passed through a prism (see Figure 1.3(b)), seven distinct colors are separated from this spectrum: *red, orange, yellow, green, blue, indigo,* and *violet.* If all seven of these visible lights are seen by your eye at the same time, then you perceive a white color. In turn, if no visible light enters your eye, then you have the perception of black or darkness.

(a) (b)

Figure 1.3. (a) Visible light spectrum (b) Visible light through a prism

Interestingly, if the human eye just views red, green, and blue light together, then it also perceives white. For this reason, red (R), green (G), and blue (B),

are primary colors [1] . A mix of these colors with varying intensities produce the seven colors in the visible light spectrum and numerous other colors. The intensity of a color is like a dimmer switch. When the dimmer switch is all the way off ($intensity = 0$), light can't be seen. If the dimmer switch is turned all the way up, then light is seen at full intensity.

In computer graphics, the intensity change of a color is in 256 discrete levels, [0, 1, 2, . . ., 254, 255]. An RGB color space for graphics is created by varying the intensity levels of the red, green, and blue colors. Since each color has 256 different levels there are a total of 256^3 (16,777,216) possible colors.

The color information contained at each pixel in the image grid is an RGB triple (r, g, b) [2] . For example, a pixel may contain the color information [255, 0, 0]. This RGB triple produces the color red because the intensity of the red color is turned all the way up to 255, and the intensities of the green and blue colors are turned all the way down to 0. Table 1.1 shows eight common colors created from RGB triples.

Red	Green	Blue	Color	
0	0	0	Black	
0	0	255	Blue	
0	255	0	Green	
255	0	0	Red	
0	255	255	Cyan	
255	255	0	Yellow	
255	0	255	Magenta	
255	255	255	White	

Table 1.1. Eight common colors using the RGB color triples

1.3 Storing Images

Images are stored in *files* that describe the content of the image. Remember from the previous discussion in section 1.1, an image is just a grid of pixels with

[1] These are not the only set of primary colors. Any subset of the seven colors that produce white when added together are primary colors.

[2] There are others ways to store color information, but for this book and commonly in computer graphics this is the way.

color information associated with each pixel. The creator of an image must store (write out) this information to a file in a standard format. A standard format allows other programs to read the file and display the image as intended by the creator of the image.

There are many different file formats such as: bitmaps, JPEG, TIFF, PPM, GIF, etc . . . Each of these formats store images differently depending on what is desired. For example, if you want to reduce the amount of information required to store an image, then JPEG would be a good format. JPEG compresses an image, which reduces the amount of hard drive space needed to store the image.

However, given the many different formats that could be used, this book chooses to focus on the PPM file format because of its simplicity; PPM stands for Portable Pixel Map. A simple program can easily be created to store images in this format. To demonstrate the PPM format, Figure 1.4(b) is used as an example image. It is a 4 x 3 image with the eight common colors from Table 1.1.

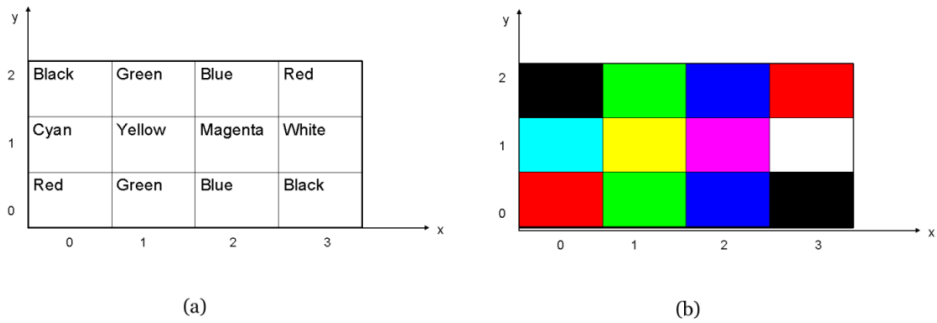


Figure 1.4. Example 4 x 3 Image: (a) Colors written (b) Actual Colors

The PPM file consists of two parts: a header and the image data. The header describes how to read the image data. The header is made of three parts, which are written on separate lines in the file. The first part is the "magic number" P3, which indicates that the image information is written as plain text. Second, the width and height are written. Using our image example, the width is 4 and the height is 3. Third, the maximal intensity of a color is given, and for our example that number is 255. Also, "#" followed by text represents comments, which contain any extra information the artist wants to include; for example, their name.

The image section of the file lists the RGB triples for each pixel in the image. For our example, there are 12 pixels, so there will be 12 sets of RGB triples in the image section. The information is written in the file from the first pixel to

the last pixel [3] . Figure 1.5 gives an example of a PPM file for Figure 1.4(b). Remember, pixel numbers are labeled left to right and row to row, starting at the beginning of each row – Figure 1.2(a).

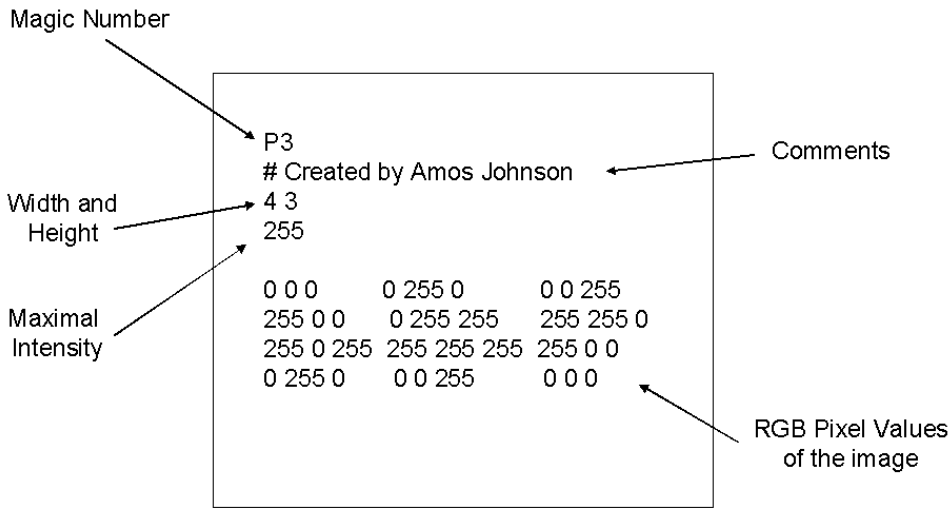


Figure 1.5. PPM file for example image in Figure 1.4(b)

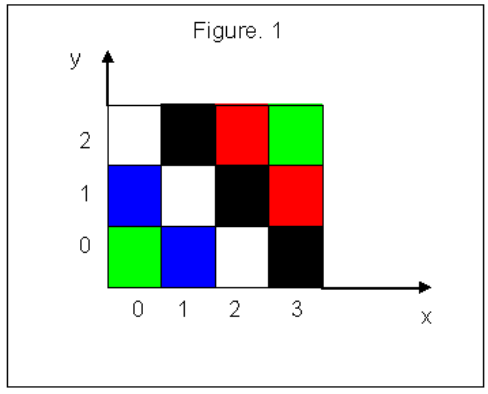
Finally, the name of the image should be named as "*anyname*.ppm". The ".ppm" extension needs to appear at the end of the image's name. Images crafted from the algorithms presented in this book can be easily stored as PPM files. The free program *IrfanView* – http://www.irfanview.com/ – may be used to view the PPM files as images.

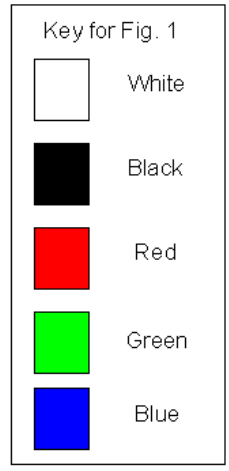
[3] A line should not contain more than 70 characters.

1.4 Review Questions

1. Find the index for the xy location (25, 70) on an image grid of height 240 and width 320.

2. Given the RGB image in figure 1, what is it resulting PPM file?



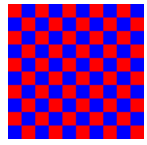


PROGRAMMING EXERCISES

1. Create a PPM image of size 320 x 240. The color of the image is to be gray [127, 127, 127], and the maximal intensity is 255. (*see example below*)

2. Create a PPM image of size 320 x 240. The color of the image is to be red [255, 0, 0], and the maximal intensity is 255. (*see example below*)

3. Create a PPM image of size 100 x 100. Each 10 x 10 block of pixels is to alternate between blue [0, 0, 255] and red [255, 0, 0] to form a checker broad pattern. The maximal intensity is 255. (*see example below*)



II 2D PRIMITIVES

2D graphics is the focus of this chapter and the next chapter. 2D graphics is the creation of images in an XY coordinate system. These images are composed of 2D primitives, such as, lines, circles, ellipses, and polygons. The following sections discuss algorithms to draw these 2D primitives and algorithms to fill (with color) the areas enclosed by the 2D primitives. The line primitive is the only primitive that can't be filled because it doesn't enclose an area.

2.1 Drawing Lines

The formula for a line in an XY coordinate system is

$$y = m \cdot x + b, \tag{2.1}$$

where m is the slope and b is the y intercept of the line. Given any two points – Figure 2.1 – the m and b parameters can be computed for the line that passes through those points. For two arbitrary points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, m and b are computed by

$$m = \frac{y_2 - y_1}{x_2 - x_1} \tag{2.2}$$

$$b = [-\frac{y_2 - y_1}{x_2 - x_1}]x_1 + y_1. \tag{2.3}$$

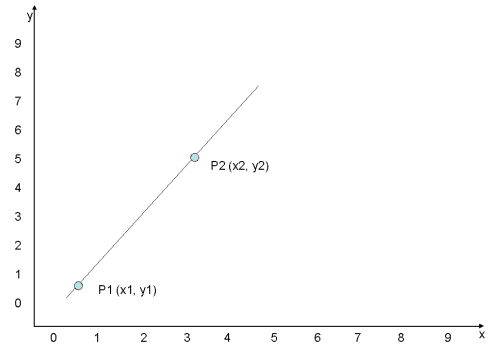


Figure 2.1. Line example

With the m and b parameters calculated, all points that lie on the line passing through points p_1 and p_2 can be computed with Equation 2.1.

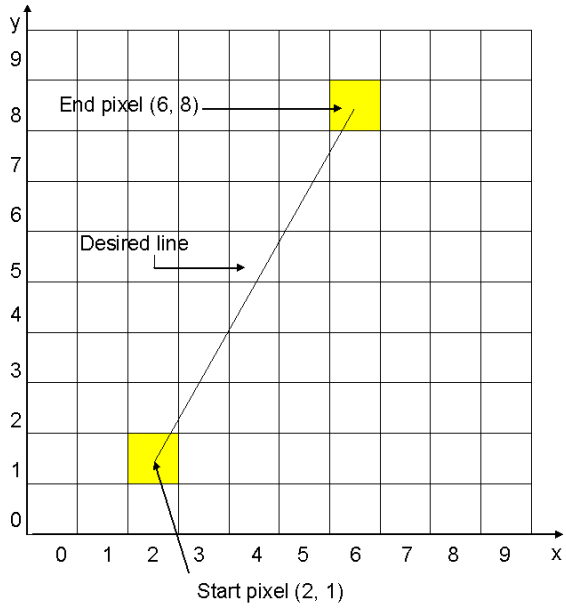


Figure 2.2. Example Line drawing

Equation 2.1 is the standard line formula, but it generates an error when $x_2 - x_1 = 0$. This situation corresponds to a vertical line with $m = b = \infty$. To draw vertical lines, Equation 2.1 is rewritten to solve for x given a y-coordinate,

$$\begin{aligned} x &= \frac{y-b}{m} \\ &= \frac{1}{m}y - \frac{1}{m}b \\ &= [\frac{x_2-x_1}{y_2-y_1}]y - [\frac{x_2-x_1}{y_2-y_1}]([-\frac{y_2-y_1}{x_2-x_1}]x_1 + y_1) \\ &= [\frac{x_2-x_1}{y_2-y_1}]y - [\frac{x_2-x_1}{y_2-y_1}]y_1 + x_1. \end{aligned} \tag{2.4}$$

Equation 2.4 not only draws vertical lines, but all non-horizontal lines ($y_2 - y_1 \neq 0$). Therefore, it is just as useful as the standard line formula (Equation 2.1).

With Equations 2.1 and 2.4, a simple line drawing algorithm may be developed. However, instead of drawing all points on the line, the algorithm draws a line segment.

As discussed in Chapter 1, an image is a rectangular grid of pixels, and the goal for a line algorithm is to *fill in* the pixels (or cells) between a start pixel location and end pixel location, with color. As an example, to draw a yellow line from point (2,1) to point (6,8) – Figure 2.2 – the first step for a *simple line algorithm* is to determine if Equation 2.1 or Equation 2.4 should be used. The

choice is made by finding the longer dimension, x or y, and using the longer dimension to solve for the shorter dimension. For our example, the x length is $|2 - 6| = 4$ and the y length is $|1 - 8| = 7$. Since the y length is longer, Equation 2.4 is used to solve for x.

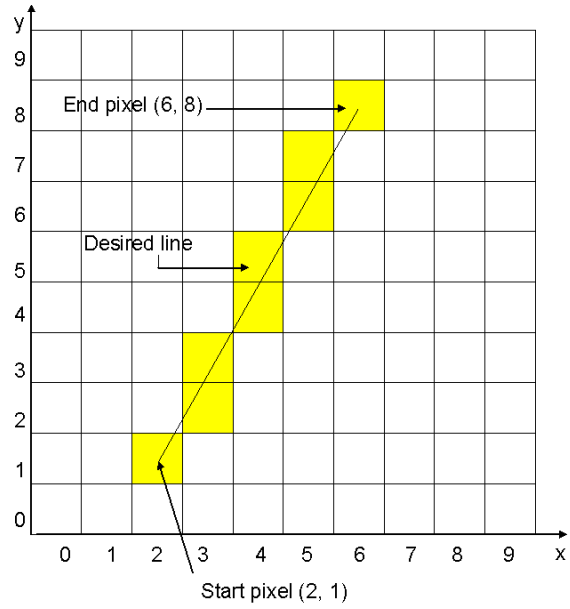


Figure 2.3. Example Line drawing with line pixels filled in.

Before solving for x, the y values from point (2,1) to point (6,8) are found: [1, 2, 3, 4, 5, 6, 7, 8]. For these y values, the x values using Equation 2.4 are [2, 2.57, 3.14, 3.71, 4.28, 4.85, 5.42, 6]. Because the line is drawn on a grid, only integer values between (2,1) and (6,8) are of importance; therefore, the x values are rounded, which gives us eight integer-points used to draw the line: (2,1), (3,2), (3,3), (4,4), (4,5), (5,6), (5,7), and (6,8) – Figure 2.3.

The line in Figure 2.3 is not straight because it is drawn on a grid with a limited number of locations. However, the actual size of a pixel displayed on a computer screen is very small; therefore, the line will *appear* straight. Also, there are graphical techniques beyond the scope of this book designed to make lines look straighter.

A simple **line algorithm** is outlined in the following steps for a given starting point (x_1, y_1) and ending point (x_2, y_2):

1. Find the x length $|x_1 - x_2|$ and the y length $|y_1 - y_2|$
2. If the x length is longer
   (a) Find all the integer values from x_1 to x_2: $[x_1...x_2]$

(b) Solve for the corresponding y values using Equation 2.1: $[y_1...y_2]$
(c) Round the y values to the nearest integer value

3. If the y length is longer
   (a) Find all the integer values from y_1 to y_2: $[y_1...y_2]$
   (b) Solve for the corresponding x values using Equation 2.4: $[x_1...x_2]$
   (c) Round the x values to the nearest integer value

Using the image-formation information in Chapter 1, an image is created of the line from the discovered pixel locations. The line pixels are set to one color and the non-line pixels to another color.

2.2 Drawing Circles

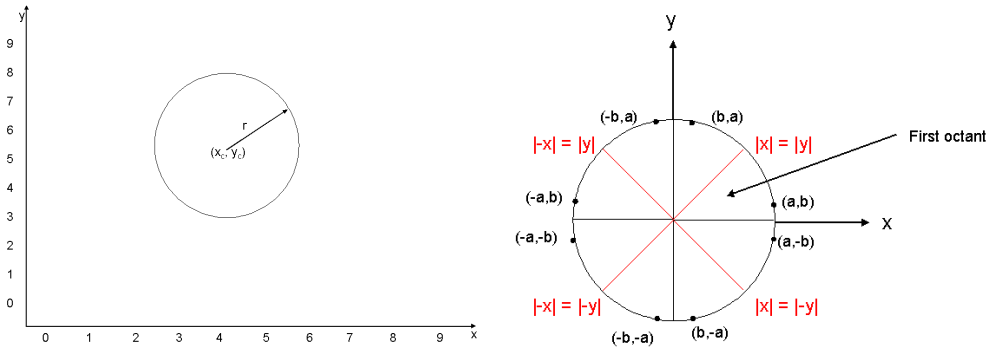
Circles are defined by a radius r and center point (x_c, y_c) – Figure 2.4(a). For a circle centered at $(0, 0)$, the equation is

$$x^2 + y^2 = r^2. \tag{2.5}$$

As with lines, we want to draw a circle on an image grid. Therefore, the points on the border of the circle are calculated. To find the border points, Equation 2.5 may be rearranged to solve for x,

$$x = \pm\sqrt{r^2 - y^2}, \tag{2.6}$$

for all the possible values of y. However, large gaps in the circle appear for values of y close to r. To avoid computing values of x for y values close to r, the eight-way symmetry that exists for circles is taken advantage of.



(a) (b)

Figure 2.4. (a) Example Circle (b) Circle with the eight octants shown. The boundaries of the octants are in red.

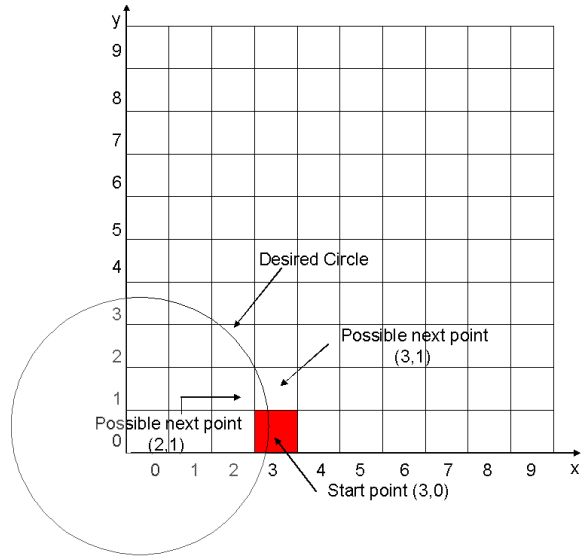


Figure 2.5. Circle centered at (0,0) with radius 3.

Figure 2.4(b) shows a circle centered at $(0, 0)$, and the eight octants of symmetry for a circle. Octants boundaries occur where the magnitude of the x and y coordinates are the same, $|x| = |y|$. Because circles have eight-way symmetry, only the circle points in the first octant need to be calculated. The points in the other seven octants are found by symmetry. Since only points in the first octant are computed with Equation 2.6, computing points with y values close to r is avoided.

With this in mind, let's develop a *circle drawing algorithm* using the example of a red circle centered at $(0, 0)$ with radius 3. The starting point of the circle in the first octant is $(3, 0)$, and the curvature of the circle in this octant dictates that the next y value is always $y + 1$, and the next x value is either x or $x - 1$: Figure 2.5. To choose between x or $x - 1$, the x-value (x_a) for the next y value $(y = 1)$ using Equation 2.6, $x_a = +\sqrt{3^2 - 1^2} = 2.83$, is calculated and rounded to the nearest integer. The choice for the next x-value is 3, since 2.83 rounded is 3. Therefore, the next point for the circle is (3,1).

Starting now from point (3,1), the next point is either (3,2) or (2,2). Solving for the x-value for $y = 2$, gives us $x_a = \sqrt{3^2 - 2^2} = 2.24$. Therefore, the next point on the grid is (2,2). Since the x location and y location are equal to each other, the next point will be in the second octant – Figure 2.6. At this juncture, we stop calculating points because the goal is to find points in the first octant and use symmetry to find the other points on the circle.

After using symmetry to find the other points, the circle has positive and

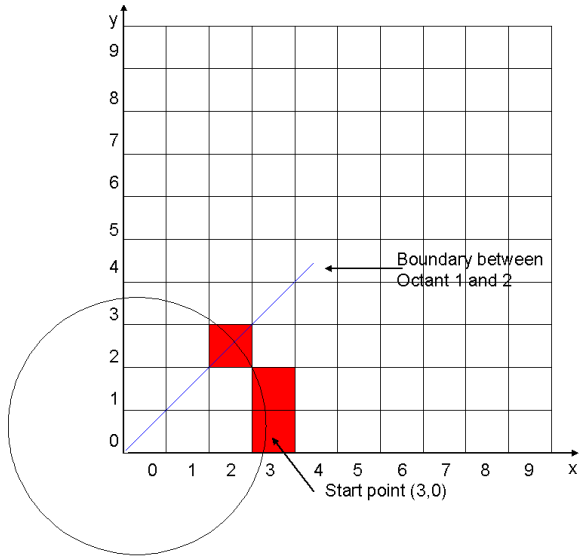


Figure 2.6. Circle centered at (0,0) with radius 3. Pixels of the circle in the first octant filled in.

negative x and y values. Whereas, an image grid has only positive x and y values. Therefore, the points in the first and second octants are only displayed. To draw a circle with all positive x and y values, the center of the circle needs to be located well within the positive quadrant of the XY coordinate system, so the border points have enough room to reside inside the image grid.

To draw a circle within the positive x-y quadrant, we first assumed that the circle is centered at (0,0). With this assumption, the same procedure as before is followed. However, after the points are computed, the center point (x_c, y_c) is added to all points; thereby moving the points to the proper location. For example, a circle centered at (4,4) with radius 3 starts off with points as a circle centered at (0,0): (3,0), (0,3), (0,-3), (-3,0), (3,1), (1,3), (1,-3), (3,-1), (-3,-1), (-1,-3), (-1,3), (-3,1), (2,2), (2,-2), (-2,-2), (-2,2). Next, the center point (4,4) is added to all points – Figure 2.7.

A simple **circle algorithm** is outlined in the following steps for a given center point (x_c, y_c) and radius r:

1. Initialize starting point to $(r, 0)$: $x = r$ and $y = 0$
2. Compute the next y location for the first octant: $y + 1$
3. Compute the corresponding x value (x_a) for $y + 1$ using Equation 2.6
4. Round x_a to the nearest integer value to find the next x-location
5. If x is greater than y, (Checking to see if points are still in first octant)
   (a) go back to step 2

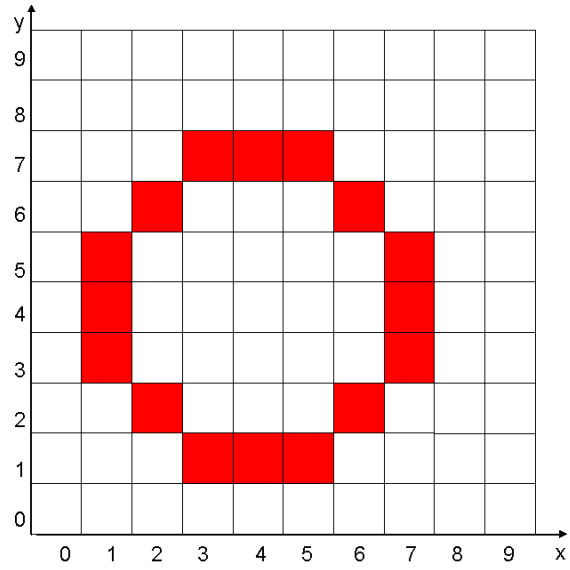


Figure 2.7. Circle centered at (4,4) with radius 3.

6. If x is less than or equal to y,
   (a) From the discovered points in the first octant, find the other points on the circle by symmetry as shown in Figure 2.6.
7. Add the center point (x_c, y_c) to all discovered points

Using the image-formation information in Chapter 1, an image is created of the circle from the discovered pixel locations. The circle pixels are set to one color and the non-circle pixels to another color.

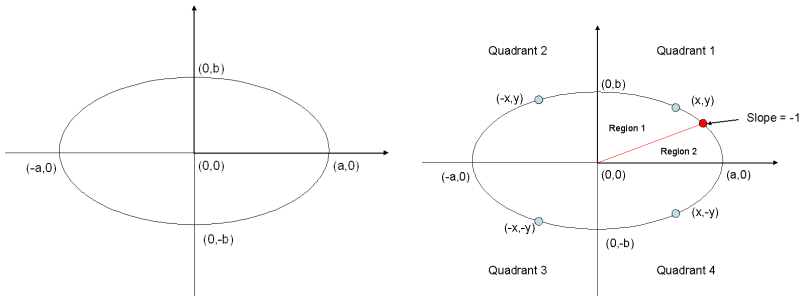
2.3 Drawing Ellipses

The formula for an ellipse centered at (0,0) is

$$b^2x^2 + a^2y^2 = a^2b^2, \tag{2.7}$$

where $2b$ is the minor axis and $2a$ is the major axis as shown in Figure 2.8(a). Similar to a circle, an ellipse drawing algorithm only computes some points and finds the others by symmetry. However, because of the curvature of the ellipse, the ellipse is divided into four quadrants instead of eight octants; providing for only four way symmetry – Figure 2.8(b).

If the points in the first quadrant are computed, the points in the other quadrants may be found by symmetry, but quadrant 1 is divided into two regions as shown in Figure 2.8(b). The separation point is where the slope of the curve



(a) (b)

Figure 2.8. (a) Ellipse example (b) Ellipse showing four way symmetry and the separation of quadrant 1 into region 1 and 2. The separation point between region 1 and 2 is where the slope of the curve is -1.

on the ellipse is -1. This separation points occurs where $a^2y = b^2x$, and this is where the curvature of the ellipse changes.

Because of the difference in curvature, computing the points on the ellipse is different for the two regions. In region 2, the x values of the ellipse are solved for by rewriting Equation 2.7,

$$x = \sqrt{a^2(1 - \frac{1}{b^2}y^2)}. \tag{2.8}$$

And in region 1, the y values of the ellipse are solved for by also rewriting Equation 2.7,

$$y = \sqrt{b^2(1 - \frac{1}{a^2}x^2)}. \tag{2.9}$$

Equations 2.8 and 2.9 are used to create an *ellipse drawing algorithm.*

Now let's develop an algorithm to find the border points for an ellipse on an image grid. A blue ellipse centered at (4,4) with $a = 4$ and $b = 2$ is our example. As with the circle algorithm, we start off by assuming the ellipse is centered at (0,0), and later we'll add the center point (4,4) to the result.

Beginning in region 2, the first point is (4,0), and because of the ellipse's curvature in region 2, the next y-value is $y + 1$ and the next x-value is either x or $x - 1$. Therefore, the next point is either (4,1) or (3,1) – Figure 2.9.

The first step is to check if the next point is in region 2 or region 1. If the next point is in region 2, the next x-value is computed for $y = 1$ using Equation 2.8,

$$x_a = \sqrt{4^2(1 - \frac{1}{2^2}1^2)} = 3.46.$$

The x-value 3.46 is rounded, and the next point is (3,1). However, point $(3, 1)$ is not in region 2, but it region 1. Remember, the transition point between region 2

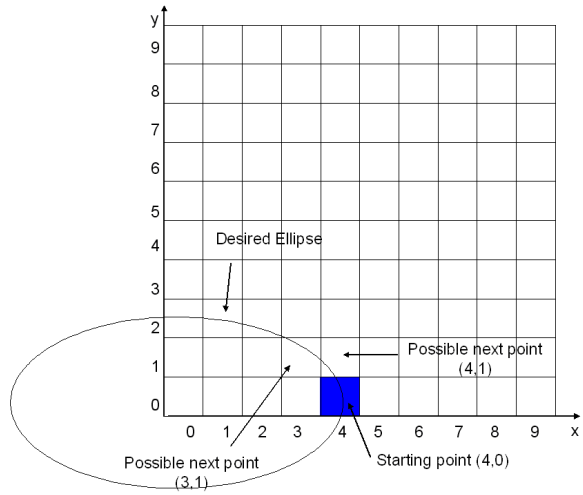


Figure 2.9. Ellipse centered at (0,0) with $a = 4$ and $b = 2$.

and region 1 is where where $a^2y = b^2x$, and any point that satisfies the condition $a^2y \geq b^2x$ is in region 1. For our ellipse with $a = 4$ and $b = 2$, this condition is satisfied for point $(3, 1)$: $16(1) \geq 4(3)$.

Without actually computing the next point, we can determine if the next point is region 2 or region 1 by first assuming that the next point is in region 2. Therefore, the next point is either $(4, 1)$ or $(3, 1)$. If the next point is in region 2, the ellipse will pass between points $(4, 1)$ and $(3, 1)$, so we can guess that the next point is approximately half way between the two points. As a way of predicting if the next point is in region 2 or region 1, we check to see if the *half-way* point $(3.5, 1)$ satisfies the condition $a^2y \geq b^2x$ to be in region 1. And for this case, $(3.5, 1)$ satisfies the condition – $16(1) \geq 4(3.5)$ – therefore, we know that the next point is in region 1.

Because the curvature is different in region 1, the next x value is $x-1$ and the next y-value needs to be founded. For our example, since the x-value is definitely equal to 3, Equation 2.9 is used to solve for y,

$$y_a = \sqrt{2^2(1 - \frac{1}{4^2}3^2)} = 1.32,$$

and the result is rounded to the nearest integer. Therefore, the next point is $(3, 1)$.

Starting with point $(3, 1)$, the next x-value is 2. Again, since the x-value is definitely equal to 2, use Equation 2.9 to solve for y: $y_a = \sqrt{2^2(1 - \frac{1}{4^2}2^2)} = 1.73$. Since the y-value is 1.73, the next point is (2,2).

For point (2,2) in region 1, the next x-value is 1, and the corresponding y-value is $y_a = \sqrt{2^2(1 - \frac{1}{4^2}1^2)} = 1.93$. After rounding 1.93, the next point on the ellipse is (1,2). Finally from (1,2), the next x-value is 0; therefore, the corresponding y-value for x = 0 is 2, so the last point in the first quadrant is (0,2) – Figure 2.10.

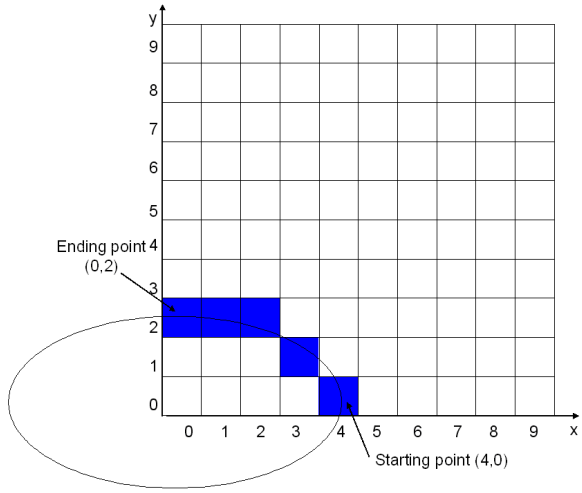


Figure 2.10. Ellipse centered at (0,0) with $a = 4$ and $b = 2$. Pixels in quadrant 1 fill in.

The points in the first quadrant are (4,0), (3,1), (2,2), (1,2), and (0,2). Using four-way symmetry as shown in Figure **??**, the other points on the ellipse are (-4,0), (-3,1), (-2,2), (-1,2), (-3,-1), (-2,-2), (-1,-2), (3,-1), (2,-2), (1,-2), and (0,-2). Last, the center point (4,4) is added to these point for the final image – Figure 2.11.

A simple **ellipse algorithm** is outlined in the following steps for a center point at (x_c, y_c), major axis of a, and minor axis of b:

1. Initialize starting point to $(a, 0)$: $x = a$ and $y = 0$
2. If $a^2(y + 1) < b^2(x - .5)$, (In region 2)
   (a) Compute the next y location for region 2: $y + 1$
   (b) Compute the x value (x_a) for $y + 1$ using Equation 2.8
   (c) Round x_a to the nearest integer
   (d) If $a^2(y + 1) < b^2(x - .5)$ (Still in region 2)
      i. go back to step 2(a)
   (e) If $a^2(y + 1) \geq b^2(x - .5)$, goto step 3
3. Now in region 1
   (a) Compute the next x location for region 1: $x - 1$
   (b) Compute the y location (y_a) for $x - 1$ using Equation 2.9

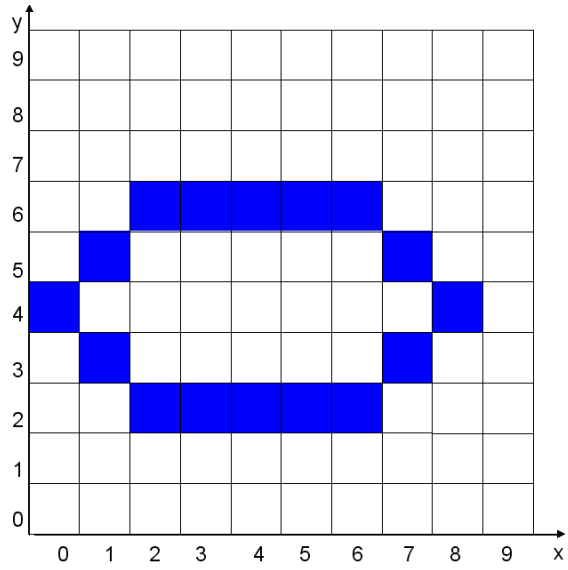


Figure 2.11. Ellipse centered at (4,4) with $a = 4$ and $b = 2$. All pixels fill in.

(c) Round y_a to the nearest integer

(d) If $x > 0$, goto step 3(a) (Still in region 1)

4. From the discovered points in the first quadrant, find the other points on the ellipse by symmetry as shown in Figure 2.8.
5. Add the center point (x_c, y_c) to all discovered points

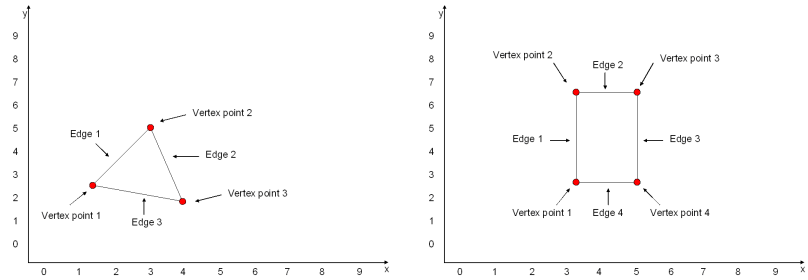
Using the image-formation information in Chapter 1, an image is created of the ellipse from the discovered pixel locations. The ellipse pixels are set to one color and the non-ellipse pixels to another color.

2.4 Drawing Polygons

In the previous sections, drawing the line, circle, and ellipse primitives on an image grid were discussed. The final 2D primitive presented is the polygon. A polygon is special primitive because it is constructed from the line primitive. It is a many sided object composed of line segments. The point where two line segments meet is called a *vertex point*, and the line segment between two vertex points is called an *edge* [1] .

Figure 2.12(a) is an example of a polygon. It has three vertex points and three edges. You may recognize this polygon because it is a triangle. A triangle is a polygon, and a rectangle is also a polygon with four edges and four vertex

[1] A polygon has at least 3 edges.



(a) (b)

Figure 2.12. Example polygon, the vertex points and edges are shown: (a)triangle (b) rectangle

points – Figure 2.12(b).

Beside triangles and rectangles, polygons can take on many shapes. In general, a polygon is either convex or concave. Given any two points inside a convex polygon, the points along the line segment between those points are inside the polygon; for a concave polygon this is not true. See Figure 2.13 for examples of each type.

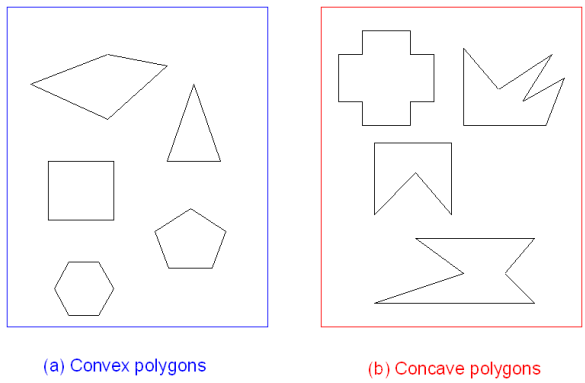


Figure 2.13. Example polygons: (a) convex polygons (b) concave polygons

A polygon drawing-algorithm is very simple because it is composed of line segments. For example, to draw the triangle in Figure 2.12(a), you start by drawing a line between point 1 and point 2, and next draw a line between point 2 and point 3, and finally draw a line between point 3 and point 1. The same line-drawing algorithm developed in section 2.1 can be used here. For an actual image example of a triangle see Figure 2.14. It is a green triangle with vertex points (1,1), (5,8), and (9,1).

A simple **polygon algorithm** is outlined in the following steps for n vertex points $[(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)]$, listed in the order to be connected:

1. Use the **line algorithm** in section 2.1 to draw a line between adjacent

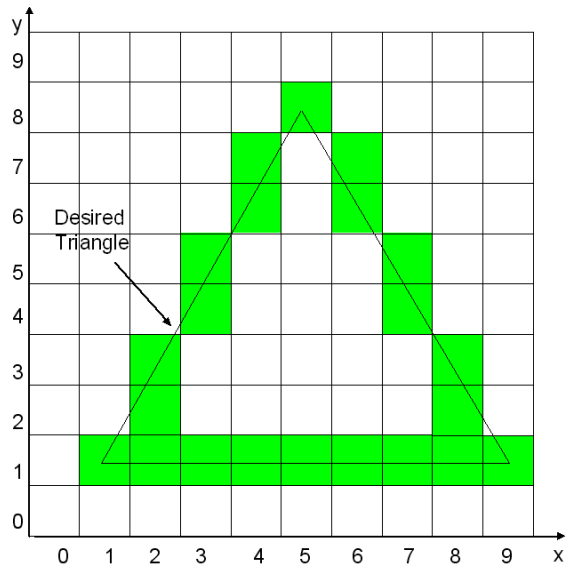


Figure 2.14. Triangular polygon. Desired triangle drawn on top of image.

points in the order listed

2. Use the **line algorithm** in section 2.1 to draw a line between the last point in the list and the first point

Using the image-formation information in Chapter 1, an image is created of the polygon from the discovered pixel locations. The polygon pixels are set to one color and the non-polygon pixels to another color.

2.5 Filling Primitives

The basic 2D primitives for 2D graphics have been presented. However, the algorithms in the previous sections are designed to draw only the boundary pixels of the primitive, and don't fill in the area enclosed by the primitive. [2] In this section, two fill algorithms are developed. The first algorithm fills in the area of the circle and ellipse, and the second algorithm fills in the area of a polygon. Two separate algorithms are needed because the circle and ellipse primitives don't collapse in on themselves; whereas, the polygon primitive may collapse in on itself (i.e.; concave polygons).

[2] The line primitive doesn't enclose an area, so a fill algorithm is not needed.

2.5.1 Filling Circles and Ellipses

To fill a circle or ellipse, first the boundary pixels for each row of the primitive is found. A row of a primitive is a set of pixels that have the same y-value. Figure 2.15 shows a row of a circle primitive.

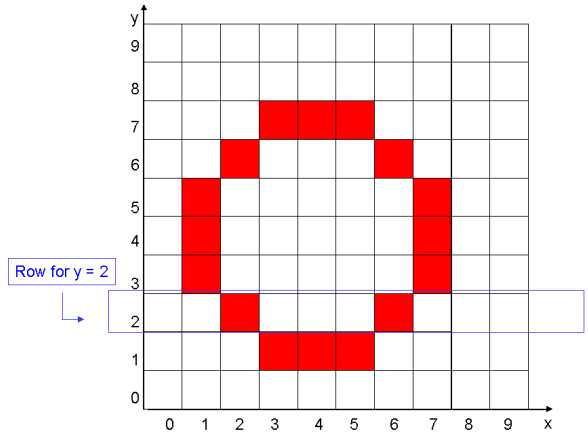


Figure 2.15. Circle showing an example of a row the primitive.

The circle in Figure 2.15 has 7 rows. Figure 2.16(a) shows the row for $y = 2$ and the two boundary pixels on that row. The goal is to fill in the pixels between the boundary pixels, as shown in Figure 2.16(a) – filled pixels are in green. Each row of a circle or ellipse has two boundary pixels: a start pixel and end pixel. The start pixel is the pixel that has the smallest x value and the end pixel is the pixel that has the largest x value. For the row of $y = 2$, the start pixel is $(2, 2)$, and the end pixel is $(6, 2)$. For each row of a circle or ellipse, the boundary pixels are found, and the pixels between the boundary pixels are filled in. Figure 2.16(b) shows the entire area of the circle filled in.

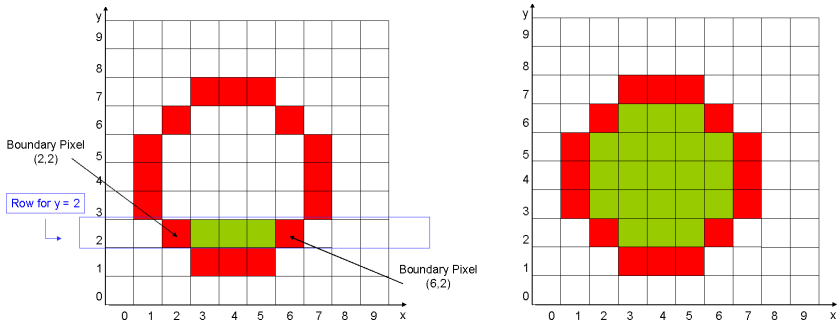


Figure 2.16. Circle primitive (a)Row for $y = 2$ filled in (b) Entire area filled in

A simple **fill algorithm** for the circle and ellipse primitive is outlined in the following steps:

1. For a Circle
   (a) Use the **circle algorithm** in section 2.2 to find the border points of the circle.
2. For an Ellipse
   (a) Use the **ellipse algorithm** in section 2.3 to find the border points of the ellipse.
3. For each row of the primitive, find the boundary pixels
4. For each row, fill in the pixels between boundary pixels

Using the image-formation information in Chapter 1, an image is created of the filled primitive from the boundary and internal pixel locations.

2.5.2 Filling Polygons

The polygon in Figure 2.17 will be used as an example to develop a fill algorithm for a polygon. The polygon is concave with 7 edges (line segments). The first step in the algorithm is to find the max and min y-values from among the vertex points of the polygon – as shown in Figure 2.17.

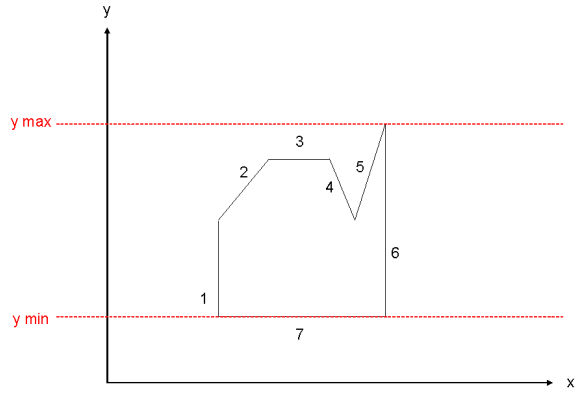


Figure 2.17. Polygon with the max and min y-values shown

The integer y-values from the min y-value (ymin) to the max y-value (ymax) are called *scan lines*. These scan lines are defined as $y = a$, where a is any integer value from ymin to ymax. The goal is to find where a scan line intersects a polygon and fill in the pixels between pairs of intersection points. For example, Figure 2.18 shows that the scan line $y = a_1$ intersects the polygon on edges 1 and 6.

To find where a scan line intersects a polygon, the intersection with the scan line and every edge in the polygon must be computed. Since the scan line has the form $y = a$, the y-value of the intersection is a and the x-values are solved for. To solve for x, line Equation 2.4 is used,

$$x = [\frac{x_2 - x_1}{y_2 - y_1}]y - [\frac{x_2 - x_1}{y_2 - y_1}]y_1 + x_1.$$

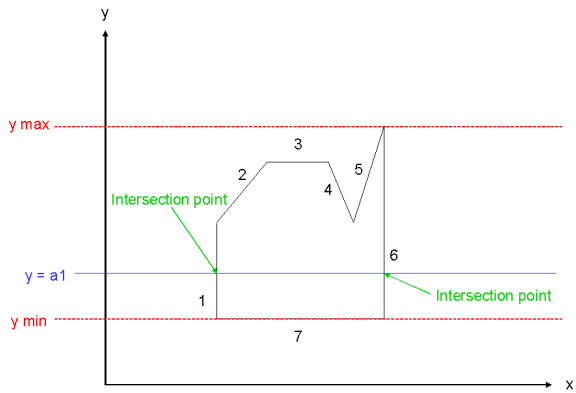


Figure 2.18. Polygon with the intersection points from a particular scan line ($y = a_1$) shown.

Each edge in the polygon has a starting vertex-point (x_1, y_1) and ending vertex-point (x_2, y_2). The vertex points are plugged into Equation 2.4 along with $y = a$ to solve for the corresponding x value. However, recall from section 2.1, Equation 2.4 doesn't work with horizontal lines. Therefore, intersections are not computed for horizontal lines. In our example, Figure 2.19 shows that edges 3 and 7 are horizontal. The points along the horizontal edge are filled in a secondary step.

Another, special case that needs to be accounted for is scan lines interesting vertex points. When a scan line intersects a vertex point, the intersection is with two edges because edges are connected at vertex points. For example, in Figure 2.20, scan line $y = a_2$ intersects edges 1 and 2 at a vertex point, and edges 4 and 5 at a vertex point. The procedure to handle vertex-point intersections is *do not count intersections with maximal vertex-points.* Every edge has two vertex points (x_1, y_1) and (x_2, y_2), and the maximal vertex-point is the vertex point with the greatest y value. For example if $y_2 > y_1$, then point (x_2, y_2) is the maximal vertex-point.

For scan line $y = a_2$ in Figure 2.20, the vertex point on edge 2 is counted and both vertex points on edges 4 and 5 are counted because scan line $y = a_2$

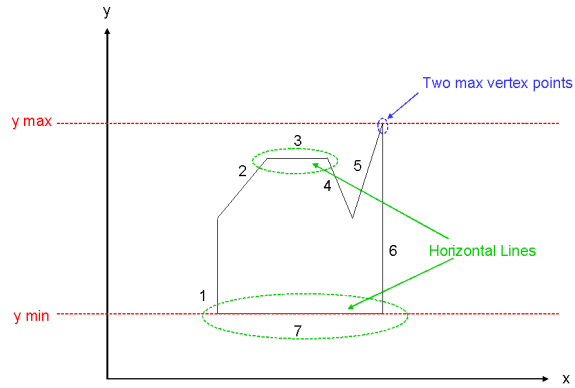


Figure 2.19. Horizontal lines and max vertex points are not counted in the fill algorithm.

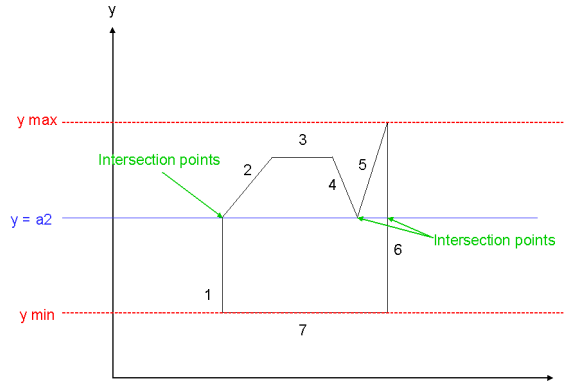


Figure 2.20. Polygon with the intersection points from a particular scan line ($y = a_2$) shown.

intersects edges 2, 4 and 5 at non-maximal vertex-points. The maximal vertex-point on edge 1 is not counted. Given this situation, scan line $y = a_2$ intersects the polygon at 4 locations – edge 2, edge 4, edge 5, and edge 6 . If the x-values of those intersections were 5, 10, 10, and 13, you would fill from 5 to 10 and 10 to 13.

A consequence of not counting the maximal vertex-points is shown in Figure 2.19. The scan line $y = ymax$ intersects edge 5 and edge 6 at maximal vertex-points. Therefore, nothing is counted for that scan line and that location of the polygon is left unfilled. However, just like horizontal edges, this situation is resolved in another step.

Finally, the scan line must intersect the edge between the vertex points of the edge. If the scan line intersection with the edge is not between the vertex points, then the scan line doesn't intersect the edge.

A simple **scan-line intersection algorithm** to compute the intersection

of a scan line $y = a$ and an edge – with vertex points (x_1, y_1) and (x_2, y_2) – is outlined in the following steps:

1. If $y_2 - y_1 = 0$,
   (a) This is a horizontal line, so there is not an intersection
2. If $y_2 - y_1 \neq 0$
   (a) If a is not in the range from y_1 to y_2
      i. The scan line is outside of the edge, so there is not an intersection
   (b) Find the y-value of the maximal vertex point
      i. if $y_1 \geq y_2$, y-maximal = y_1
      ii. if $y_1 < y_2$, y-maximal = y_2
   (c) If a = y-maximal
      i. The scan line intersects a maximal vertex-point, so there is not an intersection
   (d) If $a \neq$ y-maximal and is in the range from y_1 to y_2
      i. Find the x-value of the intersect for $y = a$ using Equation 2.4
      ii. Round the x-value to the nearest integer

For each scan line, the intersections with the edges of the polygon are computed with the **scan-line intersection algorithm**. For each scan line, the intersections are sorted from the minimal to maximal value based on the x-value of the intersection points. After sorting, pixels are filled between adjacent pairs of intersections. To compensate for not filling horizontal lines or points in the polygon where two maximal vertex-points meet, the border pixels of the polygon are filled in also – some pixels will be filled twice.

A simple **fill polygon algorithm** for a polygon with n vertex points $[(x_1, y_1), (x_2, y_2)$ $, ..., (x_n, y_n)]$, listed in the order to be connected, is outlined in the following steps:

1. Find the min y-value (ymin) and the max y-value (ymax)
2. For all the scan lines from ymin to ymax (integers values only)
   (a) For each edge
      i. Use the **scan-line intersection algorithm** to find intersections
      ii. If there are intersections
         A. Sort intersections from minimal to maximal value based on the x values of the intersection points
         B. Fill in pixels between adjacent pairs of intersection points
3. Use the **polygon algorithm** in section 2.4 to fill in the border pixels of the polygon

Using the image-formation information in Chapter 1, an image is created of the filled polygon from the boundary and internal pixel locations.

2.6 Algorithm Summary and Testing

In this Chapter, 2D primitives – lines, circles, ellipses, and polygons – were introduced and algorithms to draw them on an image grid were presented. In addition, a fill algorithm was developed that fills in the area enclosed by a 2D primitive.

The following is a list of algorithms developed in this chapter. In addition, there are input and output examples given for some of the algorithms to test your coding of the algorithms.

1. The **line algorithm** in section 2.1 draws a line between a start point and end point.
   - Input/Output Set 1
     - **Input**: Start Point = (2, 2), End Point = (6, 2)
     - **Output**: Line Points are (2, 2), (3, 2), (4, 2), (5, 2), (6, 2)
   - Input/Output Set 2
     - **Input**: Start Point = (2, 2), End Point = (2, 4)
     - **Output**: Line Points are (2, 2), (2, 3), (2, 4)
   - Input/Output Set 3
     - **Input**: Start Point = (1, 1), End Point = (4, 3)
     - **Output**: Line Points are (1, 1), (2, 2), (3, 2), (4, 3)
2. The **circle algorithm** in section 2.2 draws a circle for a particular center point and radius.
   - Input/Output Set 1
     - **Input**: Center Point = (0, 0), Radius = 3
     - **Output**: Circle Points in the *first octant* are $(3, 0)$, $(3, 1)$, and $(2, 2)$
   - Input/Output Set 2
     - **Input**: Center Point = (10, 5), Radius = 3
     - **Output**: Circle Points are $(13, 5)$, $(10, 8)$, $(10, 2)$, $(7, 5)$, $(13, 6)$, $(11, 8)$, $(11, 2)$, $(13, 4)$, $(7, 4)$, $(9, 2)$, $(9, 8)$, $(7, 6)$, $(12, 7)$, $(12, 3)$, $(8, 3)$, $(8, 7)$
3. The **ellipse algorithm** in section 2.3 draws an ellipse given a center point and the length of the major and minor axes.
   - Input/Output Set 1

- **Input**: Center Point = (0, 0), a = 4 (major axis), b = 2 (minor axis)
- **Output**: Ellipse Points in *first quadrant* are $(4, 0)$, $(3, 1)$, $(2, 2)$, $(1, 2)$, and $(0, 2)$

- Input/Output Set 2
  - **Input**: Center Point = (10, 5), a = 3 (major axis), b = 2 (minor axis)
  - **Output**: Ellipse Points are $(13, 5)$, $(7, 5)$, $(13, 6)$, $(7, 6)$, $(7, 4)$, $(13, 4)$, $(12, 6)$, $(8, 6)$, $(8, 4)$, $(12, 4)$, $(11, 7)$, $(9, 7)$, $(9, 3)$, $(11, 3)$, $(10, 7)$, $(10, 3)$
- Input/Output Set 3
  - **Input**: Center Point = (15, 10), a = 2 (major axis), b = 5 (minor axis)
  - **Output**: Ellipse Points are $(17, 10)$, $(13, 10)$, $(17, 11)$, $(13, 11)$, $(13, 9)$, $(17, 9)$, $(17, 12)$, $(13, 12)$, $(13, 8)$, $(17, 8)$, $(17, 13)$, $(13, 13)$, $(13, 7)$, $(17, 7)$, $(16, 14)$, $(14, 14)$, $(14, 6)$, $(16, 6)$, $(15, 15)$, $(15, 5)$

4. The **polygon algorithm** in section 2.4 draws a polygon given an order listed of vertex points.
5. The **fill algorithm** in section 2.5.1 fills in the area enclosed by a circle or ellipse.
6. The **scan-line intersection algorithm** in section 2.5.2 computes the intersection with a scan line and an edge of a polygon.
   - Input/Output Set 1
     - **Input**: Start Vertex-Point = $(10, 10)$, End Vertex-Point = $(20, 30)$, Scan line $y = 5$
     - **Output**: Does not intersect
   - Input/Output Set 2
     - **Input**: Start Vertex-Point = $(10, 10)$, End Vertex-Point = $(20, 30)$, Scan line $y = 15$
     - **Output**: Does intersect at Point $(13, 15)$
   - Input/Output Set 3
     - **Input**: Start Vertex-Point = $(10, 10)$, End Vertex-Point = $(20, 30)$, Scan line $y = 10$
     - **Output**: Does intersect at Point $(10, 10)$
   - Input/Output Set 4
     - **Input**: Start Vertex-Point = $(10, 10)$, End Vertex-Point = $(20, 30)$, Scan line $y = 30$

  - **Output**: Does not intersect
- Input/Output Set 5
  - **Input**: Start Vertex-Point = $(10, 10)$, End Vertex-Point = $(20, 30)$, Scan line $y = 35$
  - **Output**: Does not intersect
- Input/Output Set 6
  - **Input**: Start Vertex-Point = $(10, 10)$, End Vertex-Point = $(50, 10)$, Scan line $y = 1$
  - **Output**: Does not intersect

7. The **fill polygon algorithm** in section 2.5.2 fills in the area enclosed by a polygon.

2.7 Review Questions

1. Given two end points on a line (10,15) and (13,13), find the points between these two end points.

2. Given a circle with a center point of (0,0) and radius of 5, what are the points in the first octant?

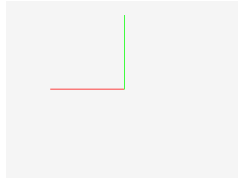
3. Given an ellipse with a center point of (0,0) and major axis of 5 and minor axis of 2, what are the points in the first quadrant?

4. What is the difference between a concave polygon and a convex polygon in terms of a line segment placed inside of the polygon?

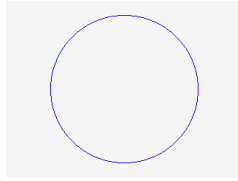
5. One edge of a polygon has a vertex points (100,200) and (500,700), what is the intersection point for a scan line of $y = 250$?

PROGRAMMING EXERCISES

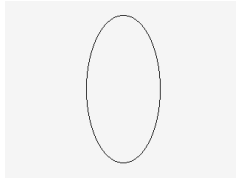
1. Create a PPM image of size 320x240, with maximal intensity of 255. The background pixels are to be light gray [245, 245, 245]. Using the **line algorithm** from section 2.1, on top of the background, draw a red [255, 0, 0] line from point (60, 120) to point (160, 120), and draw a green [0, 255, 0] line from point (160, 120) to point (160, 220). (*see example below*)



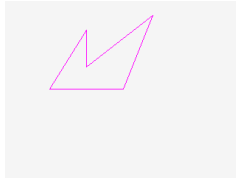
2. Create a PPM image of size 320x240, with maximal intensity of 255. The background pixels are to be light gray [245, 245, 245]. Using the **circle algorithm** from section 2.2, on top of the background, draw a blue [0, 0, 255] circle with center point at (160, 120) and radius of 100. (*see example below*)



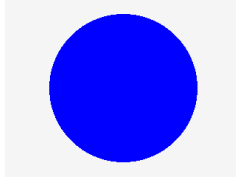
3. Create a PPM image of size 320x240, with maximal intensity of 255. The background pixels are to be light gray [245, 245, 245]. Using the **ellipse algorithm** from section 2.3, on top of the background, draw a black [0, 0, 0] ellipse with center point at (160, 120), major axis equal to 50 and minor axis equal to 100. (*see example below*)



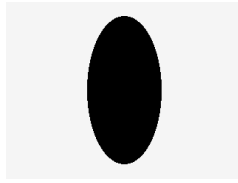
4. Create a PPM image of size 320x240, with maximal intensity of 255. The background pixels are to be light gray [245, 245, 245]. Using the **polygon algorithm** from section 2.4, on top of the background, draw a magenta [255, 0, 255] polygon with five vertex points. The vertex points (listed in order to be connected) are (60, 120), (110, 200), (110, 150), (200, 220), and (160, 120). (*see example below*)



5. Create a PPM image of size 320x240, with maximal intensity of 255. The background pixels are to be light gray [245, 245, 245]. Using the **fill algorithm** from section 2.5.1, fill in the blue circle created in Exercise 2. (*see example below*)

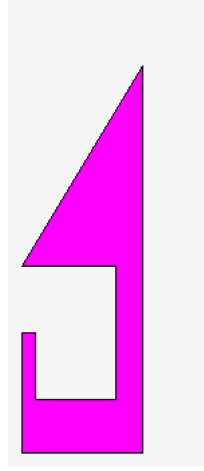


6. Create a PPM image of size 320x240, with maximal intensity of 255. The background pixels are to be light gray [245, 245, 245]. Using the **fill algorithm** from section 2.5.1, fill in the the black ellipse created in Exercise 3. (*see example below*)

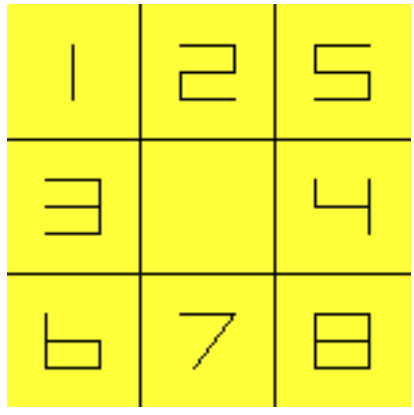


7. Create a PPM image of size 320x240, with maximal intensity of 255. The background pixels are to be light gray [245, 245, 245]. Using the **fill polygon algorithm** from section 2.5.2, fill in the the magenta polygon created in Exercise 4. (*see example below*)

8. Create a PPM image of size 150x350, with maximal intensity of 255. The background pixels are to be light gray [245, 245, 245]. Using the **fill polygon algorithm** from section 2.5.2 to a fill (with magenta [255, 0, 255]) a polygon with vertex points $(10, 10)$, $(100, 10)$, $(100, 300)$, $(10, 150)$, $(80, 150)$, $(80, 50)$, $(20, 50)$, $(20, 100)$, and $(10, 100)$. Make the border pixels black [0, 0, 0].



9. Create a PPM image of size 150 x 150 of an 8way puzzle board as shown below. The nine blocks of the board should be of size 50x50. Next create line drawings of numbers 1, 2, 3, 4, 5, 6, 7, 8 and arrange them on the board as shown. The background color of the board is yellowish [255,255,58] and the maximal intensity is 255.



III 2D LINE TRANSFORMATIONS

THE line primitive is an important primitive because polygons are formed from line segments, but the line primitive is also important because it can be used to create a world of lines. That is, an entire image can be created from lines; more specifically line segments. For example, Figure 3.1 is a simple line drawing made from 26 line segments.

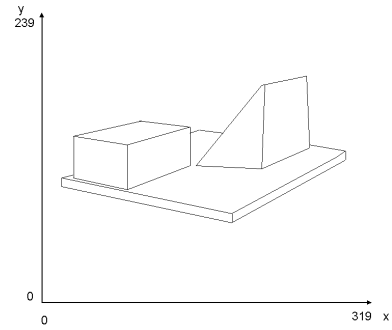


Figure 3.1. Example of a simple line drawing – two objects resting on a platform

This is a simple line drawing, but other more complex line drawings may be created. The only limitation is the imagination of the artist.

In this chapter, we present three general 2D transformations for line drawings: *translation, rotation*, and *scale*. The translation transformation moves the drawing to a different location in the XY coordinate system of the image; the rotation transformation rotates the drawing; and the scale transformation changes the size of the drawing. Figure 3.2 shows examples of each type of transformation.

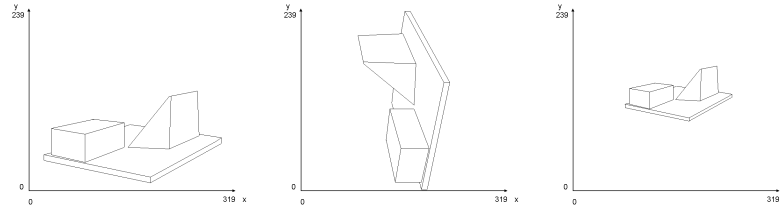


Figure 3.2. Transformation examples for Figure 3.1: (a)Translation (b) Rotation (c) Scale

3.1 Translation

Moving a line drawing is called translation. A line drawing is translated by adding scalar values to every start and end point of each line in the drawing. This process creates new (translated) start and end points for each line, and lines are drawn between the translated points using the line drawing algorithm in section 2.1.

Let's examine this process by using a simple line drawing. The line drawing in Figure 3.3(a) consists of only two lines: line 1 has a start point of (5, 5) and end point of (30, 5), and line 2 has a start point of (30, 5) and end point of (30,30). We want to translate the x values by x_t and the y values by y_t. The new x and y values (x_n, y_n) after translation are

$$x_n = x + x_t \tag{3.1}$$

$$y_n = y + y_t. \tag{3.2}$$

For example, if $x_t = 95$ and $y_t = 175$, the new points for line 1 are (100, 180) and (125, 180), and the new points for line 2 are (125, 180) and (125, 205). Figure 3.3(b) shows the results of this translation: it moves the drawing up and to the right. If we wanted to move the line drawing down and to the left, then x_t and y_t would have negative values: for example $x_t = -10$ and $y_t = -10$.

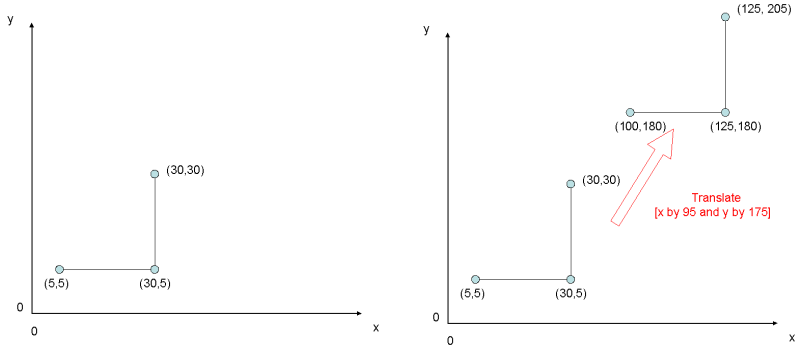


Figure 3.3. (a) Example line drawing with only 2 lines (b)Translation of line drawing: x by 95 and y by 175

A simple **translation algorithm** to translate a line drawing by x_t and y_t is outlined in the following steps:

1. For each line
   (a) Add x_t to the x-values of the start and end points as in Equation 3.1
   (b) Add y_t to the y-values of the start and end points as in Equation 3.2

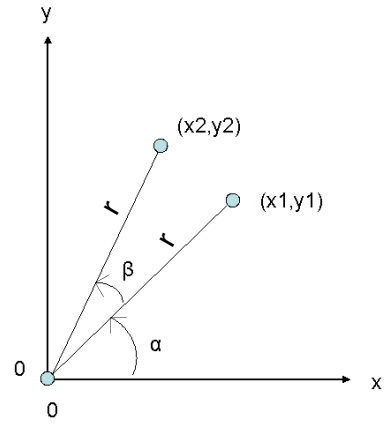


Figure 3.4. Rotating a line from (x_1, y_1) to (x_2, y_2)

2. Find points for each new line using the **line algorithm** in section 2.1

3.2 Rotation

The rotation-transformation rotates a line, and as before, the transformation is applied to the start and end points of each line. However, a line must be rotated about a point in the XY coordinate system. For example, a line with a start point of $(0, 0)$ and end point of (x_1, y_1), can be rotated β degrees about the origin of the x-y coordinate system – Figure 3.4. The origin point is $(0, 0)$; therefore, the rotated line has starting point $(0, 0)$ and ending point (x_2, y_2) as shown in Figure 3.4. Because the line in Figure 3.4 is rotated about the origin, only point (x_1, y_1) will move. Using this detail about the line, we can use this example to develop a rotation transformation.

To rotate the line (in the figure 3.4) β degrees, we need to rotate the point (x_1, y_1) to (x_2, y_2). The length of the line is r, therefore by trigonometry and right triangles

$$sin(\alpha + \beta) = \frac{y_2}{r} \tag{3.3}$$

$$cos(\alpha + \beta) = \frac{x_2}{r}, \tag{3.4}$$

and

$$sin(\alpha) = \frac{y_1}{r} \tag{3.5}$$

$$cos(\alpha) = \frac{x_1}{r}. \tag{3.6}$$

From the double angle formulas

$$\begin{aligned} sin(\alpha+\beta) &= sin(\alpha)cos(\beta)+cos(\alpha)sin(\beta) \\ cos(\alpha+\beta) &= cos(\alpha)cos(\beta)-sin(\alpha)sin(\beta). \end{aligned}$$

Therefore, by substituting Equations 3.3, 3.4, 3.5, and 3.6 into the double angle formulas we have

$$\begin{aligned} y_2 &= y_1 cos(\beta)+x_1 sin(\beta) \\ x_2 &= x_1 cos(\beta)-y_1 sin(\beta). \end{aligned}$$

Finally, the equations to rotate any point (x, y) by β degrees are

$$x_n = x \cdot cos(\beta) - y \cdot sin(\beta) \tag{3.7}$$

$$y_n = y \cdot cos(\beta) + x \cdot sin(\beta). \tag{3.8}$$

If the new start and end points from Equations 3.7 and 3.8 aren't integer values, than the points need to be rounded to nearest integer.

Equations 3.7 and 3.8 rotate any point on the line about the origin; however, we may want to rotate about other points. The process to rotate a line about an arbitrary point is composed of three steps. Let's use the 2-line drawing from Figure 3.3(a) as an example to demonstrate these steps.

To begin the process, a point to rotate about is chosen; for this example, let's pick point $(30, 5)$ – Figure 3.5. Point $(30, 5)$ is a point on both lines; however, the rotation point may be any point in the XY coordinate system.

The first step is to make the rotation point the origin, by translating the 2-line drawing by $x_t = -30$ and $y_t = -5$ – Figure 3.6(a). The translation is performed only on the start and end points of each line as in section 3.1. Second, the start and end points of each line are rotated -90 degrees using Equations 3.7 and 3.8 – Figure 3.6(b). Finally, the resulting lines are translated back by $x_t = 30$ and $y_t = 5$ – Figure 3.6(c).

A simple rotation algorithm to rotate a line drawing about a point (x_r, y_r) by β-degrees is outlined in the following steps:

1. For each line
   (a) Translate the end and start points by $x_t = -x_r$ and $y_t = -y_r$ using the **translation algorithm** in section 3.1
   (b) Rotate the translated x-values β-degrees using Equation 3.7
   (c) Round rotated x-values
   (d) Rotate the translated y-values β-degrees using Equation 3.8

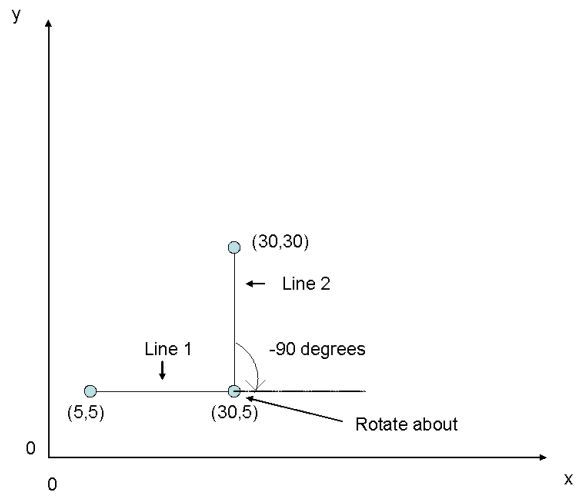


Figure 3.5. Rotate 2-Line drawing from Figure 3.3(a) by -90 degrees about point (30,5)

(e) Round rotated y-values

(f) Translate resulting points by $x_t = x_r$ and $y_t = y_r$ using the **translation algorithm** in section 3.1

2. Find the points for each new line using the **line algorithm** in section 2.1

3.3 Scale

The final 2D transformation for line drawings, which we are going to discuss, is scale. A scale transformation changes the size of a line drawing as shown in Figure 3.2(c). To scale a drawing, the start and end points of each line that compose the line drawing are scaled. The x and y values of the start and end points are scaled separately by an x scale-factor of S_x and y scale-factor of S_y. The equations to scale a point (x, y) on a line are

$$x_n = x \cdot S_x \tag{3.9}$$

$$y_n = y \cdot S_y. \tag{3.10}$$

Let's use the 1-line drawing in Figure 3.7 as an example. The line has start point (10,20) and end point (30,20), and we want to scale the x-dimension by .5 and the y-dimension by 2. Figure 3.7 shows the results of scaling each dimension with the appropriate value. The new start point is (5,40) and the new end point is (15,40). If the new start and end points aren't integer values, than the points need to be rounded to the nearest integer.

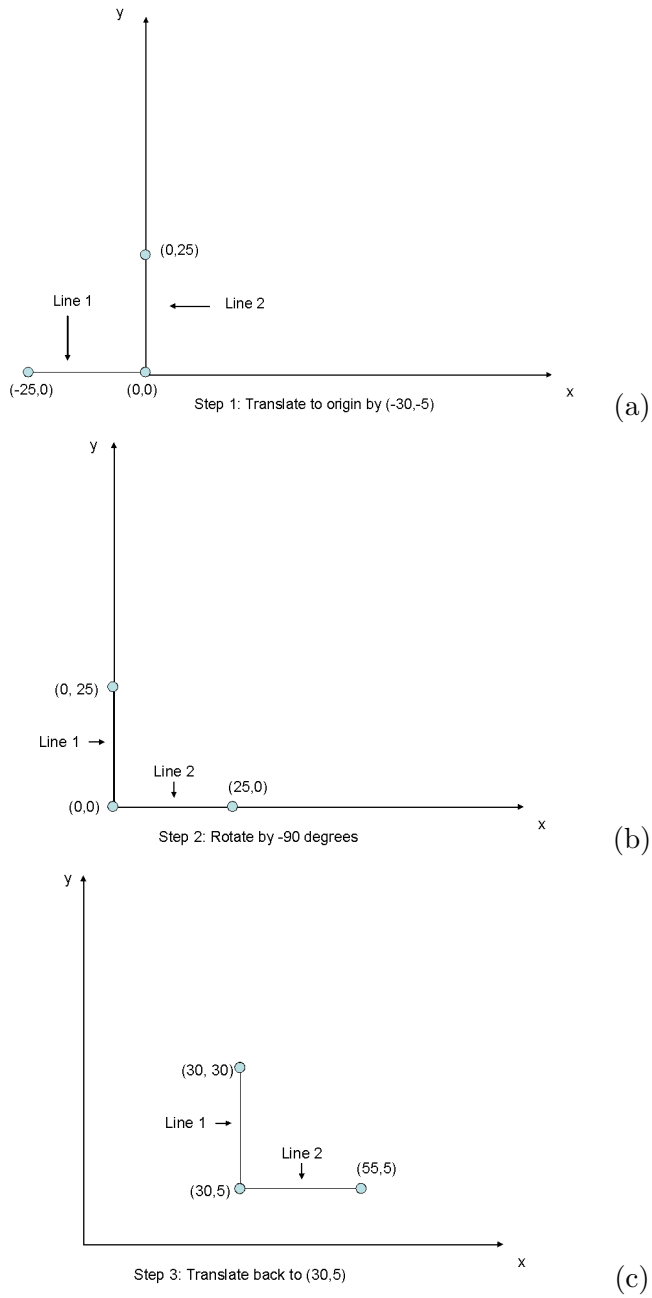


Figure 3.6. Rotation process for Figure 3.5: (a)Translate image to origin by (-30,-5) (b) Rotation by -90 degrees (c) Translate back to point (30,5)

The results of scaling this line, is to reduce the length of the line in half; however, S_y doesn't effect the length of this line because the line is horizontal. If the line was vertical, S_y would have increased the length of the line by 2.

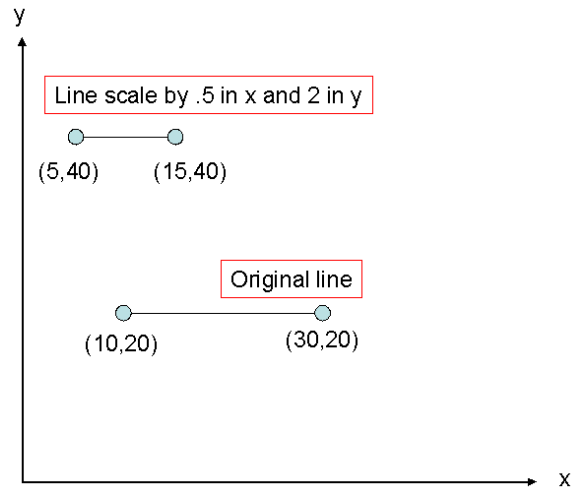


Figure 3.7. Scale a line by .5 in x-dimension and 2 in y-dimension

Although a scale transformation was applied to the line in Figure 3.7, the location of the line changed. Thus, the scale transformation has a side effect of moving the line. In order for a line drawing to remain in a fix location, a *fixed point* must be defined to scale the line by.

The process of scaling by a fixed point is the same process used in the previous section to rotate a line about an arbitrary point. Again let's use the 1-line drawing in Figure 3.7 as an example, and let's choose point (10,20) as our fixed point. The first step is to translate the line by $x_t = -10$ and $y_t = -20$ – Figure 3.8(a). Second, the x-dimension is scaled by .5 and the y-dimension is scaled by 2 using Equations 3.9 and 3.10 – Figure 3.8(b). Finally, the line is translated back by $x_t = 10$ and $y_t = 20$ – Figure 3.8(c).

A simple **scale algorithm** to scale a line drawing by S_x and S_y for a fixed point (x_f, y_f) is outlined in the following steps:

1. For each line
   (a) Translate the end and start points by $x_t = -x_f$ and $y_t = -y_f$ using the **translation algorithm** in section 3.1
   (b) Scale the translated x-values by S_x using Equation 3.9
   (c) Round scaled x-values
   (d) Scale the translated y-values by S_y using Equation 3.10
   (e) Round scaled y-values
   (f) Translate resulting points by $x_t = x_f$ and $y_t = y_f$ using the **translation algorithm** in section 3.1
2. Find the points of each new line using the **line algorithm** in section 2.1

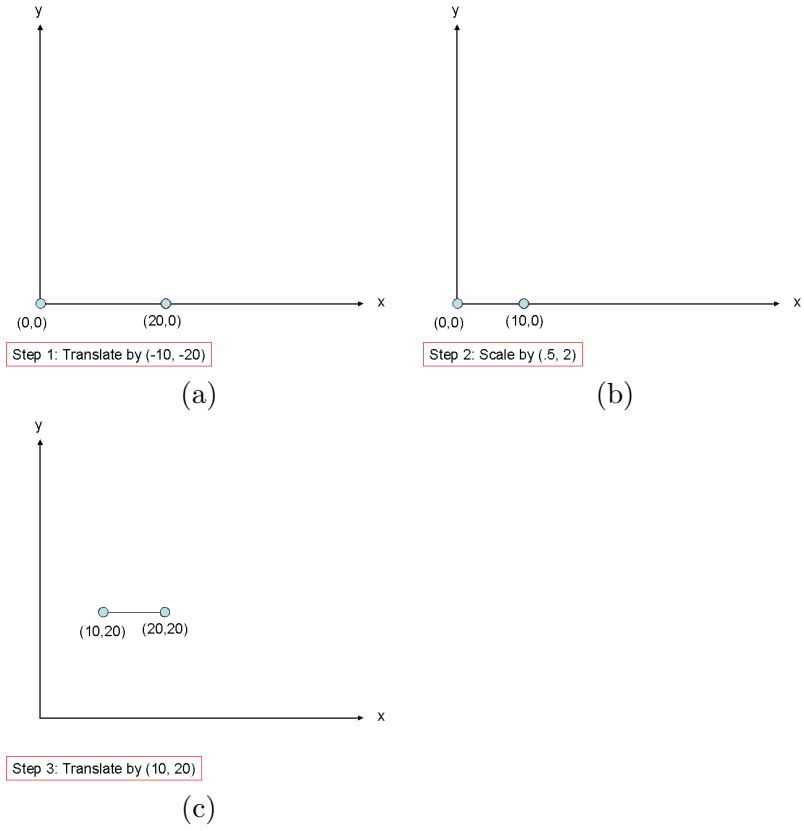


Figure 3.8. Scale process for Figure 3.7: (a)Translate image to origin by (-10,-20) (b) Scale by $S_x = .5$ and $S_y = 2$ (c) Translate back to point (10,20)

3.4 Composite Transformation

A composite transformation is a series of 2D transformations applied to a line drawing. It allows you to *play with* the location, orientation, and size of a previously created line drawing. For example, the 2-line drawing in Figure 3.3(a), may be transformed by rotating it, scaling it, and translating it. Let's see how this works by way of an example.

The goal, for this example, is to rotate the 2-line drawing by 45 degrees about point (30,5), then scale the x and y values by .5 by the fixed point (30,5), and finally translate it by (95, 175). The process for these sequence of transformation is

1. Use the **rotation algorithm** in section 3.2, with parameters $x_r = 30$, $x_y = 5$, and $\beta = 45$
2. Use the **scale algorithm** in section 3.3, with parameters $x_f = 30$, $y_f = 5$, $S_x = .5$, and $S_y = .5$

3. Use the **translation algorithm** in section 3.1, with parameters $x_t = 95$ and $y_t = 175$
4. Find the points of each new line using the **line algorithm** in section 2.1

Figure 3.9 shows the result of using this composite transformation. This composite example is only one example of an infinite set of possibilities. As stated before, the limitation is only your imagination.

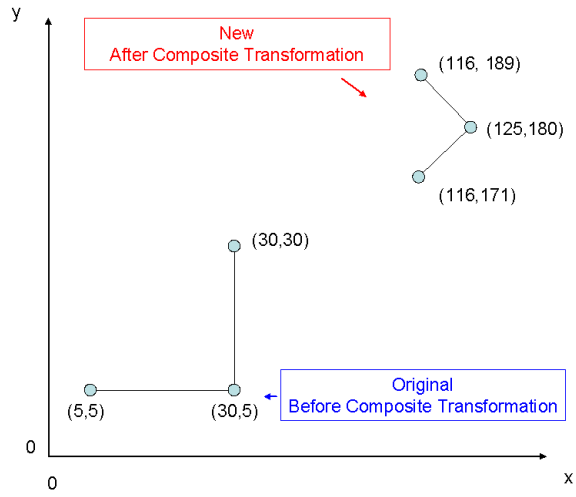


Figure 3.9. Composite Transformation example: Rotation → Scale → Translation

3.5 Algorithm Summary and Testing

In this chapter, three general 2D transformations – translation, rotation, and scale – that are applied to drawings composed of the line primitive, were discussed. Also, Section 3.4 gives an example of combining all the transformation together to perform a composite transformation.

The following is a list of algorithms developed in this chapter. In addition, there are input and output examples given for the algorithms to test your coding of the algorithms.

1. The **translation algorithm** in section 3.1 moves line drawing from one location in the image grid to another location on the image grid.
   - Input/Output Set
     - **Input**: 1-line drawing with Start Point = $(10, 20)$, End Point = $(40, 50)$, translate by $x_t = -10$ and $y_t = 20$
     - **Output**: Translated Start-Point = $(0, 40)$, Translated End-Point = $(30, 70)$

2. The **rotation algorithm** in section 3.2 rotates a line drawing about a fixed point by any specified amount of degrees.
   - Input/Output Set 1
     - **Input**: 1-line drawing with Start Point = $(10, 20)$, End Point = $(40, 50)$, rotate by 35 degrees about the fix point $(10, 20)$
     - **Output**: Rotated Start-Point = $(10, 20)$, Rotated End-Point = $(17, 62)$
   - Input/Output Set 2
     - **Input**: 1-line drawing with Start Point = $(10, 20)$, End Point = $(40, 50)$, rotate by -140 degrees about the fix point $(40, 50)$
     - **Output**: Rotated Start-Point = $(44, 92)$, Rotated End-Point = $(40, 50)$
   - Input/Output Set 3
     - **Input**: 1-line drawing with Start Point = $(10, 20)$, End Point = $(40, 50)$, rotate by 50 degrees about the fix point $(25, 15)$
     - **Output**: Rotated Start-Point = $(12, 7)$, Rotated End-Point = $(8, 49)$
3. The **scale algorithm** in section 3.3 scales a line drawing about a fixed point.
   - Input/Output Set 1
     - **Input**: 1-line drawing with Start Point = $(10, 20)$, End Point = $(40, 50)$, scale by $S_x = 5$ and $S_y = 12$ about the fix point $(10, 20)$
     - **Output**: Scaled Start-Point = $(10, 20)$, Scaled End-Point = $(160, 380)$
   - Input/Output Set 2
     - **Input**: 1-line drawing with Start Point = $(10, 20)$, End Point = $(40, 50)$, scale by $S_x = .25$ and $S_y = .5$ about the fix point $(40, 50)$
     - **Output**: Scaled Start-Point = $(32, 35)$, Scaled End-Point = $(40, 50)$
   - Input/Output Set 3
     - **Input**: 1-line drawing with Start Point = $(10, 20)$, End Point = $(40, 50)$, scale by $S_x = .1$ and $S_y = .2$ about the fix point $(100, 205)$
     - **Output**: Scaled Start-Point = $(91, 168)$, Scaled End-Point = $(94, 174)$
4. In section 3.4 a composite example of using all the transformations together was given.
   - Input/Output Set
     - **Input**: 2-line drawing; Line 1 – Start Point = $(5, 5)$, End Point = $(30, 5)$; Line 2 – Start Point = $(30, 5)$, End Point = $(30, 30)$.

First, rotate by 45 degrees about point $(30, 5)$. Second, scale by $S_x = .5$ and $S_y = .5$ about the fix point $(30, 5)$. Finally, translate the 2-line drawing by $(95, 175)$

- **Output**: Line 1 – Composite Start-Point = $(116, 171)$, Composite End-Point = $(125, 180)$; Line 2 – Composite Start-Point = $(125, 180)$, Composite End-Point = $(116, 189)$

3.6 Review Questions

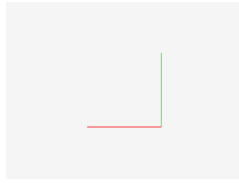
1. Rotate a line with a starting point of (25,30) and ending point of (60,75) by 45 degrees about a fix point located at (30,40). What are the start and end points after rotation?

2. Scale a line with a starting point of (25,30) and ending point of (60,75) in the x direction by 0.5 and in the y direction by 2.5 about the fix point (30,40). What are the start and end points after scaling?

3. Translate a line with a starting point of (25,30) and ending point of (60,75) in the x direction by -100 and in the y direction by 250. What are the start and end points after translation?

PROGRAMMING EXERCISES

1. Create a PPM image of size 320x240 (maximal intensity of 255). Background pixels are light gray [245, 245, 245]. From section 3.1 use the **translation algorithm** to translate a red [255, 0, 0] line, with start point (60, 120) and end point (160, 120), and a green [0, 255, 0] line, with start point (160, 120) and end point (160, 220), 50 in the x direction and -50 in the y direction. Draw the translated lines on top of the background. (*see example below*)

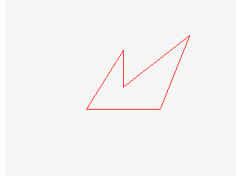


2. Create a PPM image of size 320x240 (maximal intensity of 255). Background pixels are light gray [245, 245, 245]. From section 3.2 use the **rotation algorithm** to rotate the original lines from Exercise 1, by 45 degrees about a fixed point at (160, 120). Draw the rotated lines on top of the background. (*see example below*)

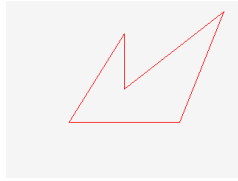
3. Create a PPM image of size 320x240 (maximal intensity of 255). Background pixels are light gray [245, 245, 245]. Using the **scale algorithm** in section 3.3, scale the original lines from Exercise 1, by .5 in x and .5 in y about a fixed point at (160, 120). Draw the rotated lines on top of the background. (*see example below*)

4. Create a PPM image of size 320x240 (maximal intensity of 255). Background pixels are light gray [245, 245, 245]. Using the composite transformation in section 3.4 as an example, rotated the original lines from Exercise 1, 45 degrees about a fixed point at (160, 120), scale the results by .5 in x and .5 in y about a fixed point at (160, 120), and then translate the results by 50 in the x direction and 50 in the y direction. Draw the composite lines on top of the background. (*see example below*)

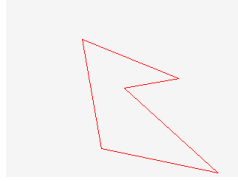
5. Create a PPM image of size 320x240 (maximal intensity of 255). Background pixels are light gray [245, 245, 245]. From section 3.1 use the **translation algorithm** to translate the lines of a red [255, 0, 0] polygon, with five vertex points, by 50 in the x directions and -30 in the y directions. The vertex points (listed in order to be connected) are (60, 120), (110, 200), (110, 150), (200, 220), and (160, 120). Draw the translated polygon on top of the background. (*see example below*)



6. Create a PPM image of size 320x240 (maximal intensity of 255). Background pixels are light gray [245, 245, 245]. Using the composite transformation in section 3.4 as an example, translate the lines of a red [255, 0, 0] polygon with five vertex points by 50 in the x directions and -30 in the y directions, and then scale the results by 1.5 in x and 1.5 in y about a fixed point at (160, 120). The vertex points (listed in order to be connected) are (60, 120), (110, 200), (110, 150), (200, 220), and (160, 120). Draw the composite polygon on top of the background. (*see example below*)

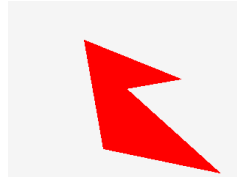


7. Create a PPM image of size 320x240 (maximal intensity of 255). Background pixels are light gray [245, 245, 245]. Using the composite transformation in section 3.4 as an example, translate the lines of a red [255, 0, 0] polygon with five vertex points by 50 in the x directions and -30 in the y directions, scale the results by 1.5 in x and 1.5 in y about a fixed point at (160, 120), and then rotate the results by -80 degrees about point (160, 120). The vertex points (listed in order to be connected) are (60, 120), (110, 200), (110, 150), (200, 220), and (160, 120). Draw the composite polygon on top of the background. (*see example below*)

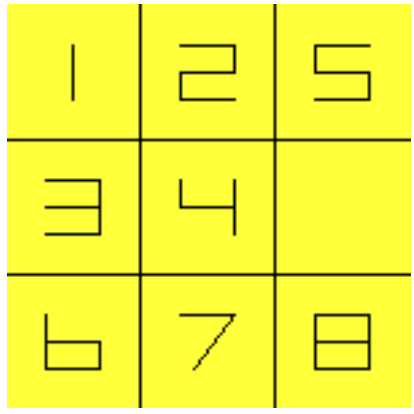


8. Create a PPM image of size 320x240 (maximal intensity of 255). Background pixels are light gray [245, 245, 245]. Using the composite transformation in section 3.4 as an example, translate the lines of a red [255, 0, 0] polygon with five vertex points by 50 in the x directions and -30 in the y directions, scale the results by 1.5 in x and 1.5 in y about a fixed point at (160, 120), and then rotate the results by -80 degrees about point (160, 120). The vertex points (listed in order to be connected) are (60, 120), (110, 200), (110, 150), (200, 220), and (160, 120). Use the **fill polygon algorithm** from section 2.5.2, to fill in the composite red

polygon. Draw the composite/filled polygon on top of the background. (*see example below*)



9. Using the same 8way puzzle board from Chapter 2 programming exercise 9, translate the number 4 to the square to its left (as shown below).



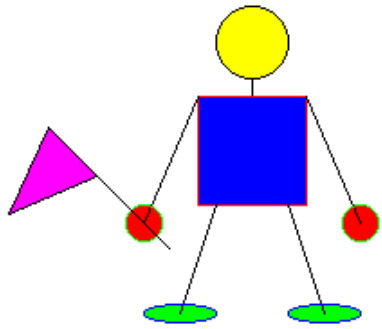
10. Create a 320 x 240 PPM image of a human-stick figure holding a flag (see example on next page). The background color is white [255,255,255], with a maximal intensity of 255. The human is constructed as such:

    - The head is made from a circle with center point of (160,120) and radius of 20.
    - The neck is made from a line with start and end points of (160,180) and (160,170).
    - The body is made from a polygon with vertex points of (130,170), (190,170), (190,110), and (130,110).
    - The right arm is a line with start and end points of (130,170) and (100,100)
    - The left arm is also a line with start and end points of (190,170) and (220,100).
    - The right hand is a circle with center point of (100,100) and radius of 10.

- The left hand is a circle with center point of (220,100) and radius of 10.
- The right leg is a line with start and end points of (140,110) and (120,50).
- The left leg is a line with start and end points of (180,110) and (200,50).
- The right foot is an ellipse with center point of (120,50), major axis is 20, and minor axis is 5.
- The left foot is also an ellipse with center point of (200,50), major axis is 20, and minor axis is 5.

The flag is constructed as such:

- The pole is a line with start and end points of (100,125) and (100,50).
- The flag is a polygon with vertex points of (100,125), (100,95), and (60,110)
- The pole and flag are translated by (0,35), in order to fit in the right hand of the stick figure.
- The pole and flag are rotated by 45 degrees for a fix point of (100,100), so the flag and pole are tilted in the right hand of the stick figure.
- The pole and flag are scaled by 1.25 in the x and y values for a fix point of (100,100), to be larger in the right hand of the stick figure.



IV VIEWING 3D ENVIRONMENTS

3D graphics is the process of viewing 3 dimensional (3D) environments on a 2D image-grid. To view a 3D environment on a 2D image-grid, the 3D world is converted to a 2D one. Therefore, transformations are needed to convert a 3D environment to a 2D one. The transformations to perform this task is the topic of this chapter.

4.1 3D Wire-frame Environment

A 3D environment is defined in an XYZ coordinate system. Within this coordinate system different types of 3D worlds may be constructed. For the purpose of learning to view a 3D environment on a 2D image-grid, simple 3D wire-frame environments are used as examples to demonstrate this process.

A 3D wire-frame environment is the 3D equivalence of a 2D line drawing. Figure 4.1 is an example of a 3D wire-frame environment in an XYZ coordinate system[1] . As this example shows, a wire-frame environment is composed of 3D line segments, and the connecting of these line segments together forms 3D objects, in this case a single *house*.

For our house example, there are 15 connecting line segments. Each line segment has a 3D start and end point. The connections between line segments are called vertex points, similar to 2D line-segment connections for 2D polygons (in section 2.4). With our house example, it should be easy to see why environments of this type are called wire-framed: just the frame of the objects are represented in this type of environment.

4.2 Perspective Projections

To view a 3D environment on a 2D image-grid, the 3D objects in the environment are first projected onto a 2D plane. In general, a projection converts points in one coordinate system into points of another coordinate system (of a lower number of dimensions). For our problem, we need to convert 3D points to 2D points: 3 dimensions to 2 dimensions. Because we are in a wire-frame environment,

[1]Example taken from Computer Graphics: Principles and Practice (C Edition) by Foley, van Dam, Feiner, and Hughes, Addison Wesley, 2nd Ed.,pg. 243

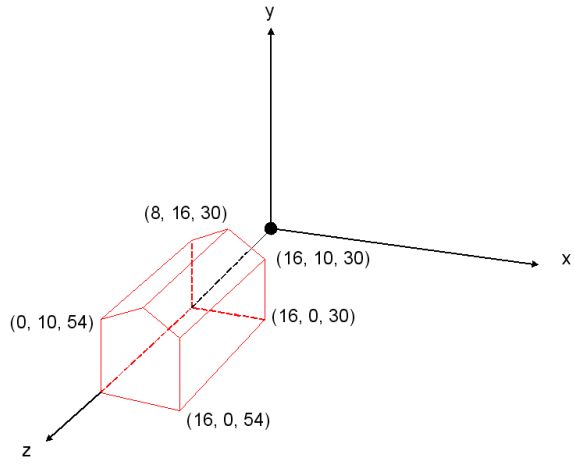


Figure 4.1. Wire-frame drawing of a 3D house in the XYZ coordinate system.

only the start and end points (vertex points) of each 3D line segment in the environment needs to be converted.

The projection of vertex points to a 2D plane (referred to as the *view plane*) is the intersection of projection rays with the view plane. Projection rays are lines which pass through the *center of projection (CoP)* and each vertex point in the environment. If the center of projection is not infinity, then the projection is called perspective. [2] For a center of projection at the origin (0,0,0), Figure 4.2(a) shows an example of projecting a 3D line segment onto a view plane that is perpendicular with the z-axis and located at $z = d$.

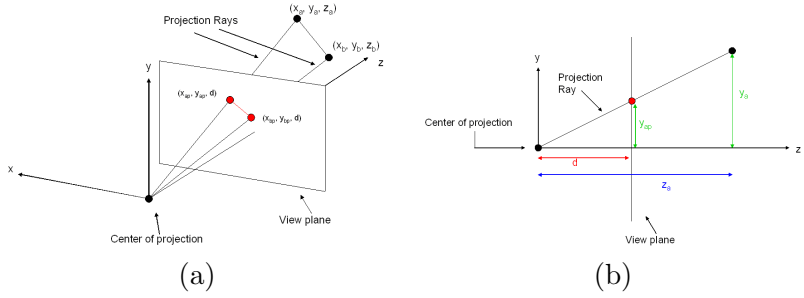


Figure 4.2. (a) Project of a 3D line segment onto View plane (b) Cross section view (along the x-axis) of a 3D point onto a View plane

In this example, vertex point (x_a, y_a, z_a) projects to point (x_{ap}, y_{ap}, d), and vertex point (x_b, y_b, z_b) projects to point (x_{bp}, y_{bp}, d). Because the view plane is located at $z = d$, all projected points have the same z value of d. By looking

[2] If the center of projection is at infinity, the projection is called parallel.

at the projection of one vertex point along the x-axis (Figure 4.2(b)), we can determine the projected y value (y_{ap}) on the view plane using similar triangles,

$$\frac{y_{ap}}{d} = \frac{y_a}{z_a}.$$

Solving for y_{ap} yields

$$y_{ap} = \frac{y_a}{z_a} \cdot d. \tag{4.1}$$

In a similar manner, by looking along the y-axis, the projected x value is

$$x_{ap} = \frac{x_a}{z_a} \cdot d. \tag{4.2}$$

Therefore, if the center of projection is the origin and the view plane is perpendicular with the z-axis, then the projected x and y values on the view plane can be found by Equations 4.2 and 4.1. Since the z values are the same for all points on the view plane, they can be disregarded. By disregarding the z-value, we have converted a 3D point to a 2D point,

$$(x_a, y_a, z_a) \Rightarrow (x_{ap}, y_{ap}).$$

A simple **projection algorithm** to project the vertex points of a 3D line segment onto a view plane located at $z = d$, for a center of projection at the origin (0,0,0), is outlined in the following steps:

1. Project the x-values of the start and end points to the view plane using Equation 4.2
2. Project the y-values of the start and end points to the view plane using Equation 4.1

4.3 Displaying Projected Lines

The 3D vertex points of a wire-frame environment are projected into a 2D XY-coordinate system using the **projection algorithm** from the previous section. However, the projected vertex-points need to be displayed as 2D lines on the image grid. As discussed in Chapter 1, an image grid has only positive x and y values; however, because of the perspective projection, some of the points in the XY coordinate system may have negative x and/or y values. Therefore, these 2D points may need to be translated to the positive quadrant of the XY coordinate system.

To translate, first locate the center (x_c, y_c) of the points by finding the max (x_{max}) and min (x_{min}) x values, and the max (y_{max}) and min (y_{min}) y values from among the points. Second, find the center of the points by

$$x_c = \frac{x_{max} + x_{min}}{2} \tag{4.3}$$

$$y_c = \frac{y_{max} + y_{min}}{2}. \tag{4.4}$$

Third, translate the center point to a location at (x_L, y_L). This translation moves all points so they lie within the positive quadrant. The 2D translation algorithm in section 3.1 is used to translate all points by

$$x_t = x_L - x_c \tag{4.5}$$

$$y_t = y_L - y_c. \tag{4.6}$$

Also, depending on the size of the 3D environment, the projected environment may be to small or to large for a particular image-grid. Therefore, the final points may need to be scaled. The 2D scale algorithm in section 3.3 is used for a fixed point of (x_L, y_L) and scale factors of $S_x = S_y = sf$. The x and y scale factor sf is the same to preserve the aspect ratio of the image.

Finally, after the vertex points are translated to the desired location and scaled accordingly, the line algorithm in section 2.1 is used to draw a line between start and end vertex-points. Remember, the vertex points that are projected to the 2D plane are start and end points for a line segment. Therefore, once the points are projected, the line between them can be drawn using our standard line drawing algorithm.

For a translation location at (x_L, y_L) and a scale factor of sf, a simple **display algorithm** to display points projected onto a view plane (as describe in section 4.2) as 2D lines is as follows:

1. Find the center of the 2D points using Equations 4.3 and 4.4.
2. Translate the start and end points of each line using the **translation algorithm** in section 3.1, where x_t and y_t are found from Equations 4.5 and 4.6
3. Scale the translated start and end points from Step 2 by $S_x = sf$ and $S_y = sf$ for a fix point of (x_L, y_L) using the **scale algorithm** in section 3.3.
4. Find the points between each start and end point using the **line algorithm** in section 2.1

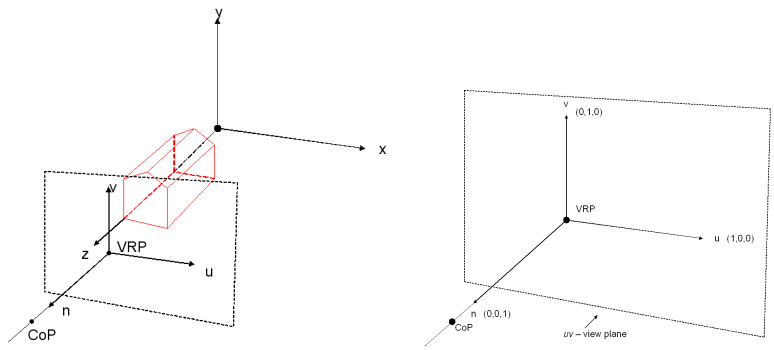
Using the image-formation information in Chapter 1, an image is created of the lines.

4.4 Arbitrary 3D View

Section 4.2 gives a method to view a 3D wire-frame environment by projecting the environment onto a 2D view-plane, but the view plane is limited to the z-axis. To view the environment from a different location and orientation in the environment, a different 3D View must be specified.

First, to view the environment from a different location, choose a point in the XYZ coordinate system from which to the view the environment – Figure 4.3(a). This point is called the view reference point (VRP). At the view reference point, a 2D view plane is created to be perpendicular to the z-axis – Figure 4.3(a). All vertex points in the wire-frame environment are projected to this view plane.

To define the view plane, a view reference coordinate-system located at the VRP is created, which makes the VRP the origin (0,0,0) for this coordinate system. The axes of the coordinate system are defined with 3 vectors $[\vec{u}, \vec{v}, \vec{n}]$, which originate from the VRP. The $\vec{u}$ vector (1,0,0) is parallel with the x-axis, the $\vec{v}$ vector (0,1,0) is parallel with the y-axis, and the $\vec{n}$ vector (0,0,1) is parallel with the z-axis – Figure 4.3(b).



(a) (b)

Figure 4.3. (a) Example of a 3D view in a wire-frame environment: with a view reference point (VRP), center of projection (CoP), and uv view-plane (b) 3D view in the view-reference coordinate-system

This new coordinate system is the UVN view-reference coordinate-system; whereas, the 3D environment and the VRP are defined in the XYZ coordinate system, referred to as the world coordinate system. It is important not to confuse the two coordinate systems.

The view plane in the view-reference coordinate-system is the uv-plane – Figure 4.3(b). Also in this coordinate system, a center of projection at $(0, 0, d_n)$ is defined along the n-axis – Figure 4.3(b). As before, the projection of vertex points onto this view plane is the intersection of projection rays with the view

plane. In the next section, we'll discuss how to perform this projection.

With what has been discussed, a 3D environment may be viewed from different locations; however, the orientation of the view is the same as in section 4.2. To change the orientation of the 3D view, the $\vec{u}$, $\vec{v}$, and $\vec{n}$ axes of the view reference coordinate systems are rotated about the VRP.

A *pan* rotation about the VRP is performed by rotating $\vec{u}$, $\vec{v}$, and $\vec{n}$ by α degrees about the v-axis – Figure 4.4.

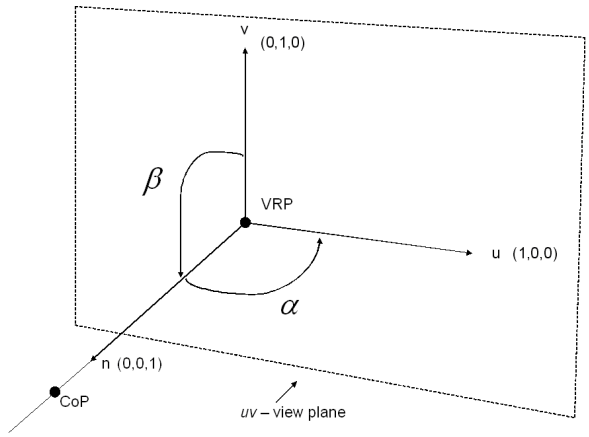


Figure 4.4. 3D view in the view-reference coordinate-system show the directions of possible rotation: a rotation of α about the v-axis (*pan*), and a rotation of β about the u-axis (*tilt*).

Positive α rotates the view plane from left to right, and negative α rotates the view plane from right to left. To perform the α rotation, the initial $\vec{u}$, $\vec{v}$, and $\vec{n}$ vectors,

$$\vec{u} = (u_1, u_2, u_3) \tag{4.7}$$
$$\vec{v} = (v_1, v_2, v_3) \tag{4.8}$$
$$\vec{n} = (n_1, n_2, n_3), \tag{4.9}$$

must be rotated to

$$\vec{u}' = (u'_1, u'_2, u'_3)$$
$$\vec{v}' = (v'_1, v'_2, v'_3)$$
$$\vec{n}' = (n'_1, n'_2, n'_3).$$

Using similar logic as in section 3.2, $\vec{u}'$ is

$$u'_1 = u_1 \cdot cos(\alpha) + u_3 \cdot sin(\alpha) \tag{4.10}$$
$$u'_2 = u_2 \tag{4.11}$$
$$u'_3 = u_3 \cdot cos(\alpha) - u_1 \cdot sin(\alpha), \tag{4.12}$$

$\vec{v}'$ is

$$v'_1 = v_1 \cdot cos(\alpha) + v_3 \cdot sin(\alpha) \quad (4.13)$$
$$v'_2 = v_2 \quad (4.14)$$
$$v'_3 = v_3 \cdot cos(\alpha) - v_1 \cdot sin(\alpha), \quad (4.15)$$

and $\vec{n}'$ is

$$n'_1 = n_1 \cdot cos(\alpha) + n_3 \cdot sin(\alpha) \quad (4.16)$$
$$n'_2 = n_2 \quad (4.17)$$
$$n'_3 = n_3 \cdot cos(\alpha) - n_1 \cdot sin(\alpha). \quad (4.18)$$

Notice that the u'_2, v'_2, and n'_2 components don't change value. This is because the α rotation is about the v-axis, and the u'_2, v'_2, and n'_2 components are on the v-axis. Figure 4.5 shows an example of a 3D view rotated by $\alpha = 45^o$ about the v-axis.

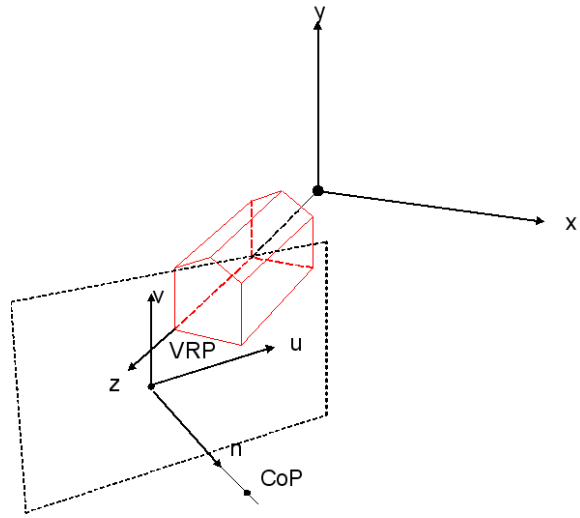


Figure 4.5. Example of a 3D view in a wire-frame environment: a rotation of $\alpha = 45^o$ about the v-axis (*pan*).

To *tilt* about the VRP, $\vec{u}$, $\vec{v}$, and $\vec{n}$ are rotated by β degrees about the u-axis – Figure 4.4. Positive β tilts the view plane backwards, and negative β tilts the view plane forwards. For the β rotation, the initial $\vec{u}$, $\vec{v}$, and $\vec{n}$ vectors (Equations 4.7, 4.9, and 4.8) must be rotated. Again, using similar logic as in

section 3.2, a β rotation for $\vec{u}'$ is

$$u_1' = u_1 \tag{4.19}$$
$$u_2' = u_2 \cdot cos(\beta) - u_3 \cdot sin(\beta) \tag{4.20}$$
$$u_3' = u_3 \cdot cos(\beta) + u_2 \cdot sin(\beta), \tag{4.21}$$

$\vec{v}'$ is

$$v_1' = v_1 \tag{4.22}$$
$$v_2' = v_2 \cdot cos(\beta) - v_3 \cdot sin(\beta) \tag{4.23}$$
$$v_3' = v_3 \cdot cos(\beta) + v_2 \cdot sin(\beta), \tag{4.24}$$

and $\vec{n}'$ is

$$n_1' = n_1 \tag{4.25}$$
$$n_2' = n_2 \cdot cos(\beta) - n_3 \cdot sin(\beta) \tag{4.26}$$
$$n_3' = n_3 \cdot cos(\beta) + n_2 \cdot sin(\beta), \tag{4.27}$$

With a β rotation, the u_1', v_1', and n_1' components don't change value. This is because the β rotation is about the u-axis, and the u_1', v_1', and n_1' components are on the u-axis. Figure 4.6 shows an example of a 3D view rotated by $\beta = 45^o$ about the u-axis.

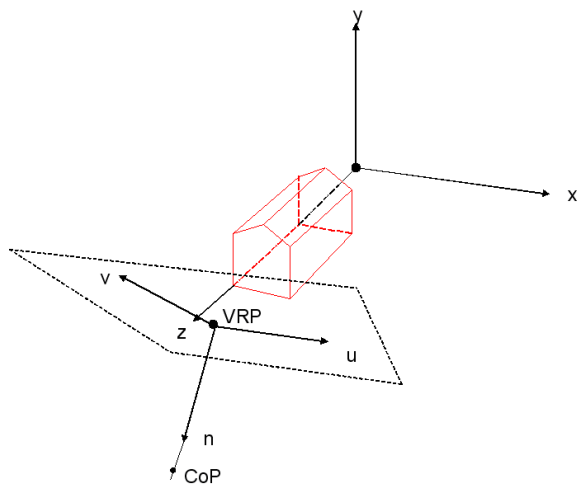


Figure 4.6. Example of a 3D view in a wire-frame environment: a rotation of $\beta = 45^o$ about the u-axis (*tilt*).

The α and β rotations may be applied separately, but usually you want to *pan* and *tilt*. This combination is preformed by applying the β rotation first, and then applying the α rotation on the results.

A simple **3D view algorithm** to define a view for a 3D environment given α and β is outlined in the following steps:

1. Initialize $\vec{u}$ to $(1, 0, 0)$, $\vec{v}$ to $(0, 1, 0)$, and $\vec{n}$ to $(0, 0, 1)$
2. Rotate $\vec{u}$ by β using Equations 4.19, 4.20, and 4.21
3. Rotate $\vec{v}$ by β using Equations 4.22, 4.23, and 4.24
4. Rotate $\vec{n}$ by β using Equations 4.25, 4.26, and 4.27
5. Rotate the result of step 2 by α using Equations 4.10, 4.11, and 4.12
6. Rotate the result of step 3 by α using Equations 4.13, 4.14, and 4.15
7. Rotate the result of step 4 by α using Equations 4.16, 4.17, and 4.18

4.5 Arbitrary 3D View Alignment

As in section 4.2, the vertex points in the 3D environment viewed from an arbitrary 3D view must be projected to the view plane. Projecting to an arbitrary view plane creates a potential problem: *all point on the view plane may not have the same z-value.* If the z-values are different, then they can't be disregarded to create 2D points. Therefore, to avoid this potential problem, the view reference coordinate system for any arbitrary 3D view is aligned with the XYZ coordinate system. When the view reference coordinate system is aligned, the view plane is the XY-plane, which is a 2D plane and all the z-values are the same.

The first step for this task is to translate the VRP $(x_{vrp}, y_{vrp}, z_{vrp})$ to the origin of the XYZ coordinate system, $(x_{vrp}, y_{vrp}, z_{vrp}) \Rightarrow (0, 0, 0)$. To translate the VRP to the origin, all vertex points (x, y, z) in the 3D environment must be translated as follows:

$$x_1 = x - x_{vrp} \tag{4.28}$$

$$y_1 = y - y_{vrp} \tag{4.29}$$

$$z_1 = z - z_{vrp}. \tag{4.30}$$

Figure 4.7 shows an example of this translation step, using Figure 4.5 as the beginning 3D view.

After the VRP is translated to the origin of the XYZ coordinate system, the $\vec{u}$, $\vec{v}$, and $\vec{n}$ axes of the view reference coordinate systems are aligned with the XYZ coordinate system: $\vec{u}$ is aligned with the x-axis, $\vec{v}$ is aligned with the y-axis, and $\vec{n}$ is aligned with the z-axis. To align these axes, each translated point

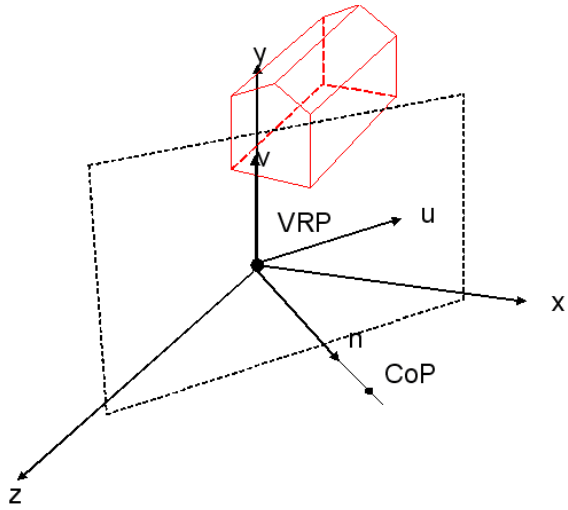


Figure 4.7. Translation step to move the view reference point to the origin on the XYZ coordinate system. The 3D view is from Figure 4.5

(x_1, y_1, z_1) in the environment is rotated as follows:

$$x_2 = x_1 \cdot u_1 + y_1 \cdot u_2 + z_1 \cdot u_3 \quad (4.31)$$

$$y_2 = x_1 \cdot v_1 + y_1 \cdot v_2 + z_1 \cdot v_3 \quad (4.32)$$

$$z_2 = x_1 \cdot n_1 + y_1 \cdot n_2 + z_1 \cdot n_3. \quad (4.33)$$

Figure 4.8(a) shows an example of this rotation step, after the translation step shown in Figure 4.7.

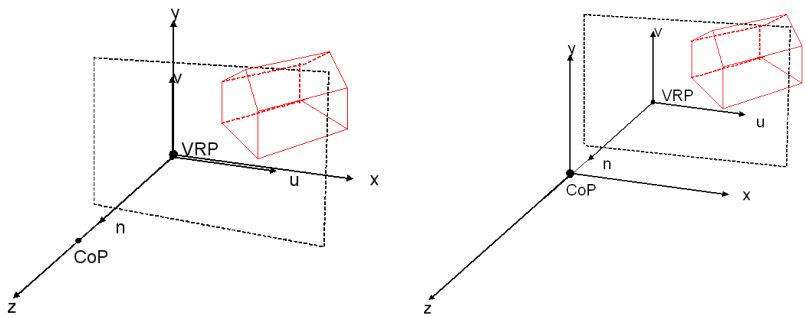


Figure 4.8. (a) Rotation step to align the $\vec{u}$, $\vec{v}$, and $\vec{n}$ axes with the XYZ axes. From view in Figure 4.7 (b) Translate the center of projection to origin of the XYZ coordinate system. From view in Figure 4.8(a)

We now have the view reference coordinate-system aligned with the world coordinate-system. Therefore, the two systems are now the same, and the view

plane is the XY-plane as desired.

In section 4.2, an algorithm was developed to project 3D vertex points to a view-plane perpendicular with the z-axis, for a center of projection at the origin. Currently the view plane is the XY-plane, which is perpendicular with the z-axis, but the center of projection is not the origin. To use the projection algorithm as developed, the center of projection $(0, 0, d_n)$ is translated to origin. This translation is applied to the rotated points (x_2, y_2, z_2) in the environment,

$$x_3 = x_2 - 0 \tag{4.34}$$

$$y_3 = y_2 - 0 \tag{4.35}$$

$$z_3 = z_2 - d_n. \tag{4.36}$$

Because the center of projection is on the z-axis, only the z values change. Figure 4.8(b) shows an example of this translation step, after the rotation step shown in Figure 4.8(a).

After the translation of the center of projection to the origin, the view plane is located at $z = -d_n$. The projection algorithm in section 4.2 can now be used for a view plane at $z = -d_n$.

A simple **3D view-alignment algorithm** to align the view reference coordinate system with the world coordinate-system, for a VRP $= (x_{vrp}, y_{vrp}, z_{vrp})$, CoP $= (0, 0, d_n)$, and view reference coordinate system $= [\vec{u}, \vec{v}, \vec{n}]$, is as follows:

1. For each vertex point
   (a) Translate the x-values by -x_{vrp} using Equation 4.28
   (b) Translate the y-values by -y_{vrp} using Equation 4.29
   (c) Translate the z-values by -z_{vrp} using Equation 4.30
   (d) Rotate the new x-values from step (a) by $\vec{u}$ using Equation 4.31
   (e) Rotate the new y-values from step (b) by $\vec{v}$ using Equation 4.32
   (f) Rotate the new z-values from step (c) by $\vec{n}$ using Equation 4.33
   (g) Translate the new z-values from step (f) by $-d_n$ using Equation 4.36

4.6 Displaying 3D Views

By combining the algorithms from the previous sections, to view a 3D wire-frame environment from a particular VRP $= (x_{vrp}, y_{vrp}, z_{vrp})$ with a CoP $= (0, 0, d_n)$ and a view plane orientated by α and β, a simple **3D view-display algorithm** is outlined in the following steps

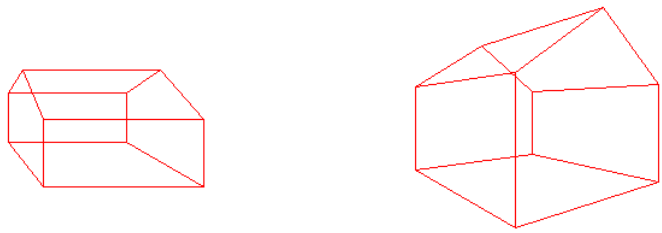
1. Find the view reference coordinate system $= [\vec{u}, \vec{v}, \vec{n}]$ for α and β using the **3D view algorithm** in section 4.4

2. Align the 3D environment to the standard view for the VRP, CoP, and $[\vec{u}, \vec{v}, \vec{n}]$ using the **3D view-alignment algorithm** in section 4.5.
3. Project the vertex points to the view plane at $z = -d_n$ using the **projection algorithm** in section 4.2.
4. Use the **display algorithm** from section 4.3 to display the projected vertex points as 2D lines.

Figures 4.9 and 4.10 are examples of 2D displays for the house in Figure 4.1. The images were generated from the 3D view-display algorithm.

(a) (b)

Figure 4.9. 2D display for house in Figure 4.1 (a) 3D-view Parameters: VRP = (36, 25, 74), α = 0, β = 0, CoP = $(0, 0, 25)$, sf = 10 (b)3D-view Parameters: VRP = (36, 25, 74), α = 45, β = -45, CoP = $(0, 0, 25)$, sf = 10



(a) (b)

Figure 4.10. 2D display for house in Figure 4.1 (a) 3D-view Parameters: VRP = (36, 25, 74), α = 90, β = 0, CoP = $(0, 0, 25)$, sf = 10 (b)3D-view Parameters: VRP = (8, 8, 54), α = -45, β = 0, CoP = $(0, 0, 25)$, sf = 10

4.7 Algorithm Summary and Testing

In this chapter, the process of viewing a 3 dimensional wire-frame environment on a 2D image grid was discussed. The following is a list of algorithms developed in this chapter. In addition, there are input and output examples given for the algorithms to test your coding of the algorithms.

1. The **projection algorithm** in section 4.2 projects vertex points of 3D lines (segments) in a wire-frame environment onto a view plane, which is perpendicular with the z-axis, and drops off the z-values of the projected points to form 2D points.
   - Input/Output Set 1
     - **Input**: Wire-frame environment with one 3D line, Start Point = $(35, 40, 70)$, End Point = $(20, 30, 50)$, project onto a view plane located at $d = 20$
     - **Output**: Projected Start-Point = $(10, 11.43)$, Projected End-Point = $(8, 12)$
   - Input/Output Set 2
     - **Input**: Wire-frame environment with one 3D line, Start Point = $(35, 40, 70)$, End Point = $(20, 30, 50)$, project onto a view plane located at $d = -20$
     - **Output**: Projected Start-Point = $(-10, -11.43)$, Projected End-Point = $(-8, -12)$
2. The **display algorithm** in section 4.3 displays projected vertex points as 2D lines on a image grid.
   - Input/Output Set 1
     - **Input**: Wire-frame environment with one 3D line, Start Point = $(35, 40, 70)$, End Point = $(20, 30, 50)$, projected onto a view plane located at $d = 20$ (using the **projection algorithm** in section 4.2) – translate to location $(160, 120)$, and scale by $sf = 10$
     - **Output**: Displayed Start-Point = $(170, 117)$, Displayed End-Point = $(150, 123)$
   - Input/Output Set 2
     - **Input**: Wire-frame environment with two 3D line, Line 1 – Start Point = $(35, 40, 70)$, and End Point = $(20, 30, 50)$; Line 2 – Start Point = $(55, 40, 20)$, and End Point = $(30, 50, 10)$; projected onto a view plane located at $d = 20$ (using the **projection algorithm** in section 4.2) – translate to location $(500, 500)$, and scale by $sf = 10$

  - **Output**: Line 1 –Displayed Start-Point = $(260, 57)$, and Displayed End-Point = $(240, 63)$; Line 2 – Displayed Start-Point = $(710, 343)$, and Displayed End-Point = $(760, 943)$

3. The **3D view algorithm** in section 4.4 changes the orientation of a view plane.
   - Input/Output Set 1
     - **Input**: Orientation of the view plane: $\alpha = 45$ and $\beta = 90$
     - **Output**: View plane axes: $\vec{u} = (0.7071, 0, -0.7071)$, $\vec{v} = (0.7071, 0, 0.7071)$, and $\vec{n} = (0, -1, 0)$
   - Input/Output Set 2
     - **Input**: Orientation of the view plane: $\alpha = -15$ and $\beta = 62$
     - **Output**: View plane axes: $\vec{u} = (0.9659, 0, 0.2588)$, $\vec{v} = (-0.2285, 0.4695, 0.8529)$, and $\vec{n} = (-0.1215, -0.8829, 0.4535)$
4. The **3D view-alignment algorithm** in section 4.5 aligns an arbitrary view plane, which is defined in a view reference coordinate system, with the world coordinate system at the origin.
   - Input/Output Set
     - **Input**: Wire-frame environment with one 3D line, Start Point = $(35, 40, 70)$, End Point = $(20, 30, 50)$. VRP = $(20, 20, 75)$, CoP = $(0, 0, 20)$, $\vec{u} = (0.7071, 0.7071, 0)$, $\vec{v} = (0, 0, 1)$, $\vec{n} = (0.7071, -0.7071, 0)$
     - **Output**: Aligned Start-Point = $(24.7487, -5, -23.5355)$, Aligned End-Point = $(7.0711, -25, -27.0711)$
5. The **3D view-display algorithm** in section 4.6 combines all the algorithms in this chapter to displays a 3D wire-frame environment – viewed from an arbitrary view plane – on a 2D image grid.
   - Input/Output Set 1
     - **Input**: Wire-frame environment with one 3D line, Start Point = $(35, 40, 70)$, End Point = $(20, 30, 50)$. VRP = $(20, 20, 75)$, CoP = $(0, 0, 20)$, $\alpha = 45$, and $\beta = 90$. Translate to location $(160, 120)$, and scale by $sf = 10$
     - **Output**: Displayed Start-Point = $(136, 197)$, Displayed End-Point = $(184, 43)$
   - Input/Output Set 2
     - **Input**: Wire-frame environment with one 3D line, Start Point = $(35, 40, 70)$, End Point = $(20, 30, 50)$. VRP = $(0, 0, 20)$, CoP = $(0, 0, -20)$, $\alpha = 0$, and $\beta = 0$. Translate to location $(160, 120)$, and scale by $sf = 10$

- **Output**: Displayed Start-Point = $(170, 117)$, Displayed End-Point = $(150, 123)$

4.8 Review Questions

1. What are the three view plane axes $(\vec{u}, \vec{v}, \vec{n})$, when the orientation of the view plane is α = -35 and β = 15?

2. Align a 3D line with starting point of (25,20,70) and ending point of (10,35,55) based the following viewing parameters: vrp = (10,15,135), cop = (0,0,50), $\vec{u}$ = (0.7071, 0.7071, 0), $\vec{v} = (0, 0, 1)$, $\vec{n} = (0.7071, -0.7071, 0)$. What is the start and end points after alignment?

3. For a view plane located at d = 25, what is the perspective projection of a 3D line with start point of (25,50,70) and end point of (45,78,30) onto the view plane. Drop off the z-values of the projected points and give your answer as 2D points.

4. Given projected points (10, 11.43) and (8, 12) from a 3D line onto a view plane, what is the location of these points of an image grid when the translation location is (60,20) and the points are scaled by 5?

PROGRAMMING EXERCISES

The house in Figure 4.1 will be used in programming exercise #2. The house is composed of 15 lines. Table 4.1 list the start and end points for each line.

Line	Start Point	End Point
1	(0, 0, 54)	(0, 10, 54)
2	(0, 0, 54)	(16, 0, 54)
3	(16, 0, 54)	(16, 10, 54)
4	(16, 10, 54)	(8, 16, 54)
5	(8, 16, 54)	(0, 10, 54)
6	(8, 16, 54)	(8, 16, 30)
7	(8, 16, 30)	(16, 10, 30)
8	(16, 10, 30)	(16, 0, 30)
9	(16, 0, 30)	(16, 0, 54)
10	(16, 10, 54)	(16, 10, 30)
11	(8, 16, 30)	(0, 10, 30)
12	(0, 10, 30)	(0, 10, 54)
13	(0, 0, 54)	(0, 0, 30)
14	(0, 0, 30)	(0, 10, 30)
15	(0, 0, 30)	(16, 0, 30)

Table 4.1. Start and end points for the 15 lines that are apart of the house example in Figure 4.1.

1. Create a PPM image of size 320x240 (maximal intensity of 255). Background pixels are light gray [245, 245, 245]. Use the **projection algorithm** in section 4.2 to project a wire-frame environment with one 3D line, Start Point $= (35, 40, 70)$ and End Point $= (20, 30, 50)$, onto a view plane located at $d = 20$. Use the **display algorithm** in section 4.3 to draw the line (in red [255, 0, 0]) on top of the background image for a translation location of $(160, 120)$ and scale of $sf = 80$. (*see example*)

2. Create a PPM image of size 320x240 (maximal intensity of 255). Background pixels are white [255, 255, 255].Display the house in Table 4.1 on top of the background using the **3D view-display algorithm** in section 4.6. Draw the lines of the house red $[255, 0, 0]$
   (a) Use the parameters settings for VRP, CoP, α, β, and sf in Figure 4.9(a), for a translation location of $(160, 120)$
   (b) Use the parameters settings for VRP, CoP, α, β, and sf in Figure 4.9(b), for a translation location of $(160, 120)$
   (c) Use the parameters settings for VRP, CoP, α, β, and sf in Figure 4.10(a), for a translation location of $(160, 120)$
   (d) Use the parameters settings for VRP, CoP, α, β, and sf in Figure 4.10(b), for a translation location of $(160, 120)$

V LIGHTING 3D ENVIRONMENTS

IN chapter 4, a 3D wire-frame environment was projected onto a 2D view plane, which allowed the environment to be viewed on a 2D image grid. Because the environment was wire-framed – Figure 4.1 – only the vertex points of the lines composing the environment were projected. Therefore, the wire-frame environment was *see-through* and *hollow*. In this chapter, we introduce a solid 3D environment, where instead of lines forming objects, objects are composed of 3D surfaces. These surfaces are created from 3D polygons or spheres.

Similar to wire-frame environments, solid environments also must be viewed on a 2D image-grid. Since we are dealing with surfaces (instead of lines), the color of a surface is projected onto the 2D image-grid, instead of vertex points. The color of a surface depends on the lighting model used. A lighting model is a set of rules of how light reflects off surfaces in the environment and how those rules affect the colors of the surfaces.

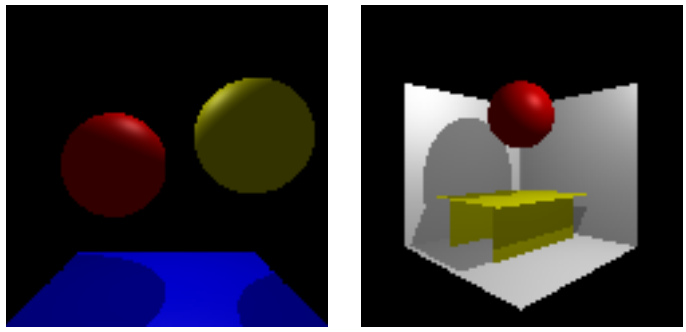
The color of a surface also varies based on whether or not another object is blocking the light source. An object blocking the light source causes a shadow on the object being blocked. Because viewing a solid 3D environment depends on the lighting model and objects blocking the light, the process of viewing a solid 3D environment is referred to as lighting the environment.

Figure 5.1 shows examples of lighting two different solid 3D environments. Figure 5.1(a) is an environment composed of 1 red sphere, 1 yellow sphere, and 1 blue polygon, and Figure 5.1(b) is an environment composed of 1 red sphere, 3 white polygons, and 3 yellow polygons. Each environment is demonstrating the effects (shadows) of objects blocking other objects from the light source. The following sections are dedicated to explaining the process of lighting a solid 3D environment composed of spheres and polygons.

5.1 3D Surfaces

Objects in a solid 3D environment may be composed of many different types of 3D surfaces. However, we narrow our coverage to only two surfaces: polygons [1] and spheres.

[1] Although, connections of polygons – known as polygon meshes – can create numerous types of objects. For example, a 3D box can be represent with 6 polygons.



(a) (b)

Figure 5.1. Example lighting of solid 3D environments (a) Environment contains 2 spheres and 1 polygon (b)Environment contains 1 sphere and 6 polygons

5.1.1 Spheres

The equation of a sphere is

$$(x-a)^2 + (y-b)^2 + (z-c)^2 = r^2. \tag{5.1}$$

The point (a, b, c) is the center of the sphere and r is the radius. Any point (x_0, y_0, z_0) in the XYZ coordinate system that satisfies the equation

$$(x_0-a)^2 + (y_0-b)^2 + (z_0-c)^2 - r^2 = 0$$

is on the surface of the sphere. Figure 5.2 shows an example of a sphere in the XYZ coordinate system with a center point of (a, b, c).

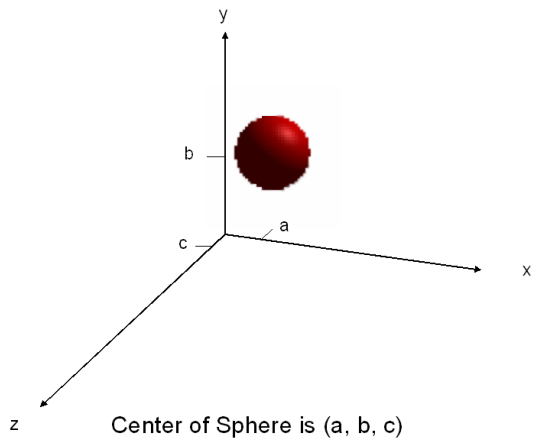
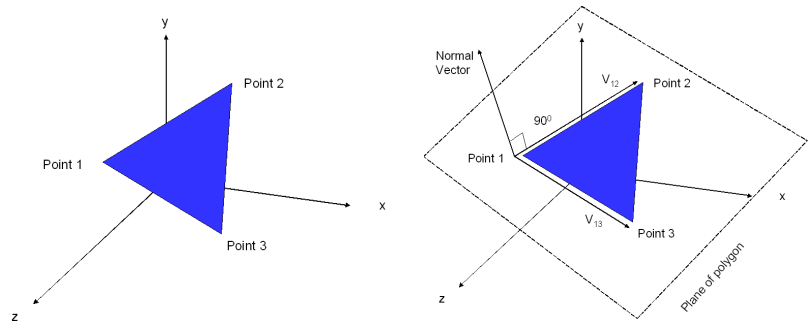


Figure 5.2. Example of a sphere with center point (a, b, c)

5.1.2 Polygons

A 3D polygon is defined the same as a 2D polygon in section 2.4. The difference is that 3D polygon have 3D vertex points (x, y, z) instead of 2D vertex points [2] . Figure 5.3(a) is an example of a convex polygon with three vertex points. *Objects created with polygons in this chapter are created from only convex polygons.*



(a) (b)

Figure 5.3. (a) Example of a 3D polygon with 3 vertex points (b)All points on the 3D polygon lie in a plane. Three vertex points of the polygon are used to find the normal vector of the plane.

A polygon in the XYZ coordinate system is a flat surface; therefore, all points on the surface of the polygon lie in a plane – Figure 5.3(b). To determine if a point (x_0, y_0, z_0) is on the surface of the polygon, the plane in which the polygon lie must first be defined. The equation of a plane is

$$A \cdot x + B \cdot y + C \cdot z + D = 0, \tag{5.2}$$

where $\vec{N} = (A, B, C)$ is the normal vector of the plane. Three vertex points of the polygon are used to find the normal vector – Figure 5.3(b). The normal vector is a vector that forms a 90 degree angle with the plane.

From three vertex points, to find the normal vector, we first compute the vector from point 1 to point 2

$$\vec{v}_{12} = P_2 - P_1 \tag{5.3}$$

and secondly, the vector from point 1 to point 3

$$\vec{v}_{13} = P_3 - P_1 \tag{5.4}$$

as shown in Figure 5.3(b). The cross product of the vectors $\vec{v}_{12}$ and $\vec{v}_{13}$ is the normal vector $\vec{N}$ of the plane

$$\vec{N} = \vec{v}_{13} \text{ x } \vec{v}_{12} = (A, B, C). \tag{5.5}$$

[2] A 3D or 2D polygon must have at least three vertex points.

Where

$$A = y_{v_{13}} \cdot z_{v_{12}} - z_{v_{13}} \cdot y_{v_{12}} \quad (5.6)$$

$$B = z_{v_{13}} \cdot x_{v_{12}} - x_{v_{13}} \cdot z_{v_{12}} \quad (5.7)$$

$$C = x_{v_{13}} \cdot y_{v_{12}} - y_{v_{13}} \cdot x_{v_{12}} \quad (5.8)$$

The parameter D of the plane Equation 5.2 is found by negating the dot product of the normal vector with any vertex point (*let's always use point 1 for simplicity*). The dot product between $\vec{N}$ and P_1 is

$$\vec{N} \bullet P_1 = A \cdot x_1 + B \cdot y_1 + C \cdot z_1,$$

therefore using the negative dot product

$$D = -(\vec{N} \bullet P_1). \quad (5.9)$$

A simple **polygon plane algorithm** to find the plane of a 3D polygon with n vertex points $[(x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3),$ ..., $(x_n, y_n, z_n)]$ – listed in the order to be connected – is outlined in the following steps:

1. Compute the parameters of the plane Equation 5.2
   (a) Compute the vector, $\vec{v}_{12}$, from point 1 (x_1, y_1, z_1) to point 2 (x_2, y_2, z_2) using Equation 5.3
   (b) Compute the vector, $\vec{v}_{13}$, from point 1 (x_1, y_1, z_1) to point 3 (x_3, y_3, z_3) using Equation 5.4
   (c) Calculate the normal vector $\vec{N} = (A, B, C)$ by taking the cross product of $\vec{v}_{13}$ and $\vec{v}_{12}$ using Equations 5.6, 5.7, and 5.8
   (d) Calculate the D parameter from the negative dot product of $\vec{N}$ with point 1 (x_1, y_1, z_1) using Equation 5.9

5.1.3 Polygon Surface

After the plane of the polygon is defined, to determine if a point (x_0, y_0, z_0) is on the surface of the polygon, the polygon is projected onto one of the three major planes – XY plane, XZ plane, YZ plane – of the XYZ coordinate system, see Figure 5.4(a). For the most accurate results, the plane whose coefficient in the polygon's plane Equation 5.2 – A for the YZ plane, B for the XZ plane, C for the XY plane – that has the highest absolute value is selected as the projection plane. The projection is obtained by dropping off the coordinate – x for A, y for B, or z for C – for the corresponding plane selected [3] . For example, if the absolute

[3] This is an example of a parallel projection. To project a 3D environment to a 2D environment, for a parallel projection, the 3D coordinates is simply dropped off.

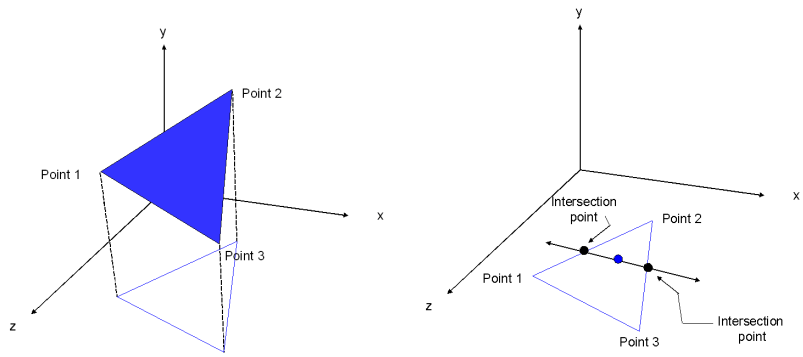


Figure 5.4. (a) 3D polygon projected onto one of the three major planes of the XYZ coordinate system (b) A point is inside a convex polygon if it intersects one edge to the left of the point and one edge to the right of the point

value of $|A|$ is greater than the absolute value of the other two coefficients, then the vertex points of the polygon and point (x_0, y_0, z_0) is projected as follows

$$(x, y, z) \Rightarrow (y, z); \tag{5.10}$$

if $|B|$ is higher,

$$(x, y, z) \Rightarrow (x, z); \tag{5.11}$$

if $|C|$ is higher,

$$(x, y, z) \Rightarrow (x, y). \tag{5.12}$$

By projecting the 3D polygon onto one of the three major planes, the 3D polygon becomes a 2D polygon. For a 2D polygon, to determine if a point lies inside the interior of the polygon, the intersections of the horizontal line that passes through the point is computed with the edges of the *now* 2D polygon – Figure 5.4(b). If the horizontal line intersects one edge to the left of the point and one edge to the right of the point, then the point lies within the interior of the polygon [4] .

Since, the vertex points were converted from 3D to 2D by either Equation 5.10, 5.11, or 5.12, the new vertex points exist in a 2D coordinate system. This new coordinate system can either be XY, XZ, or YZ depending on which of the three equations was used. If we use the variables i and j to represent points in either of the three coordinate systems, then a generic IJ coordinate system may be created – Figure 5.5(a).

[4] This only works for convex polygons. Concave polygon are treated differently, however, we are only focusing on convex polygons.

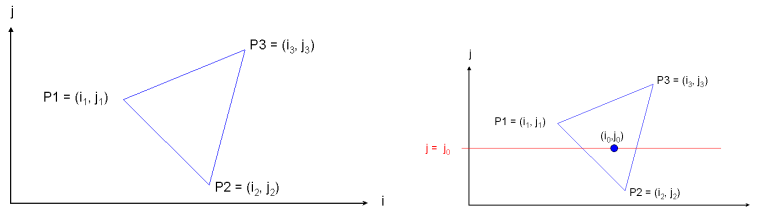


Figure 5.5. (a) Generic IJ coordinate system for projected 3D polygon (b) Determining if a point (i_0, j_0) is inside the boundary of the project 3D polygon. If the horizontal line $j = j_0$, which passes through point (i_0, j_0), intersects one edge to the left of (i_0, j_0) and one edge to the right of (i_0, j_0), then (i_0, j_0) lies within the interior of the polygon.

Given this generic 2D coordinate system, an edge of the polygon is computed between vertex points. As stated before, an edge is a line segment between two points. However, instead of using the standard line equation, Equation 2.1, to define the line segment between *any two* vertex points, $P_1 = (i_1, j_1)$ and $P_2 = (i_2, j_2)$, the parametric equations of a line are used

$$i = i_1 + t \cdot (i_2 - i_1) \tag{5.13}$$

$$j = j_1 + t \cdot (j_2 - j_1) \tag{5.14}$$

If $t = 0$, the point on the edge is (i_1, j_1). For $t = 1$, the point on the edge is (i_2, j_2). These points are the start and end points of the edge. Values of t between 0 and 1 generates points of the edge between the start and end points. The edge between all adjacent pairs of vertex points of the 2D polygon are defined by Equations 5.13 and 5.14.

After the polygon is projected to one of the 2D planes, project the point (x_0, y_0, z_0) to the same plane; the resulting projection point is (i_0, j_0). From the projected point, the horizontal line [5] that passes through the point in the IJ coordinate system is $j = j_0$, see Figure 5.5(b). To compute its intersection with an edge of the polygon, t is solved for in Equation 5.14 for $j = j_0$

$$t = \frac{j_0 - j_1}{j_2 - j_1} \tag{5.15}$$

If the value of t is greater than 1 or less than 0, then the horizontal line doesn't intersect the edge. However, if t is in the range of 0 to 1, then the i and j coordinates of the intersection point is computed by plugging the value of t into Equations 5.13 and 5.14.

After the intersection with every edge is computed, if there is at least one intersection on ether side of the point, then the point is inside the polygon. We

[5] This is the same as a scan line, section 2.5.2

say at least one because if the horizontal line intersects a vertex point, then that point is counted as an intersection with two edges. Also, if the point being test is a vertex point, then it is considered inside the polygon.

The final concern that needs to be addressed is horizontal edges. A horizontal edge generates the value of 0 for $j_2 - j_1$. If the edge is horizontal, then the intersection has to be computed differently. For a horizontal edge, you have to check to see if the point actually lies on the edge. First, for the point to be on the horizontal edge, j_0 must equal j_1 and j_2 (which are the same value). If j_0 is equal to j_1 and j_2, t is computed using Equation 5.13 for $i = i_0$

$$t = \frac{i_0 - i_1}{i_2 - i_1}. \tag{5.16}$$

If the value of t is *less than or equal to* 1 and *greater than or equal to* 0, then the point lies on the horizontal edge, and the original point (x_0, y_0, z_0) is on the surface of the 3D polygon.

A simple **on surface polygon algorithm** to compute if a point (x_0, y_0, z_0) is on the surface of a 3D polygon with n vertex points $[(x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3), ..., (x_n, y_n, z_n)]$ – listed in the order to be connected – is outlined in the following steps:

1. Using the **polygon plane algorithm** in section 5.1.2, find the plane of the 3D polygon – $(Ax + By + Cz + D = 0)$
2. Project the vertex points of the polygon and point (x_0, y_0, z_0) onto one of the 3 major 2D planes
   (a) If $|A|$ is greater than $|B|$ and $|C|$
      i. Convert polygon vertex-points and point (x_0, y_0, z_0) to 2D points using Equation 5.10
   (b) If $|B|$ is greater than $|A|$ and $|C|$
      i. Convert polygon vertex-points and point (x_0, y_0, z_0) to 2D points using Equation 5.11
   (c) If $|C|$ is greater than $|B|$ and $|A|$
      i. Convert polygon vertex-points and point (x_0, y_0, z_0) to 2D points using Equation 5.12
3. For each non-horizontal edge $(j_1 \neq j_2)$
   (a) From the projected point $(x_0, y_0, z_0) \Rightarrow (i_0, j_0)$, find the horizontal line that passes through point (i_0, j_0): $j = j_0$
   (b) Compute t for $j = j_0$ with the edge using Equation 5.15
   (c) If t is from 0 to 1, compute the i intersection point by inserting t into Equation 5.13

   (d) If t greater than 1 or less than 0, no intersection

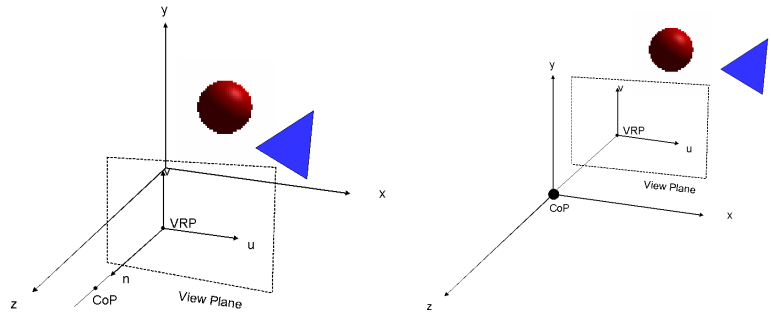
4. From the non-horizontal edges, determine the number of intersections on ether side of (i_0, j_0)
   (a) Count the number of intersection points that have i-values **less than *or* equal to** i_0
   (b) Count the number of intersection points that have i-values **greater than *or* equal to** i_0
   (c) If the number of intersections is greater than 0 from step 3(a) and if the number of intersections is greater than 0 from step 3(b), then the point (x_0, y_0, z_0) is on the surface of the polygon
   (d) If the number of intersections is not greater than 0 from step 3(a) and if the number of intersections is not greater than 0 from step 3(b), then the point (x_0, y_0, z_0) is not on the surface of the polygon
5. For each horizontal edge $(j_1 = j_2)$
   (a) From the projected point $(x_0, y_0, z_0) \Rightarrow (i_0, j_0)$, if j_0 not equal to j_1, point (i_0, j_0) is not on the edge
   (b) If $j_0 = j_1$
      i. Compute t for $i = i_0$ using Equation 5.16
      ii. If t is from 0 to 1, point (x_0, y_0, z_0) is on the surface of the polygon
      iii. If t greater than 1 or less than 0, point (i_0, j_0) is not on the edge
6. If (x_0, y_0, z_0) is a polygon vertex-point, than the point is on the surface of the polygon

5.2 View Point Specification and Alignment

As with wire-frame environments, an arbitrary view point can be specified for a solid environment – Figure 5.6(a). The process of specifying the view is the same as in section 4.4. A view reference point (VRP), a center of projection (CoP), and orientation of the view plane given by α and β are defined.

The **3D view algorithm** from section 4.4 is used to adjust the orientation of the view plane based on α and β. The output of the algorithm are the vectors $\vec{u}, \vec{v}$, and $\vec{n}$, which define the view plane.

The **3D view-alignment algorithm** from section 4.5 is used to align the view reference coordinate system with the world coordinate system – Figure 5.6(b) – after the view plane is defined. The algorithm is used on the vertex points of the 3D polygons in the environment, and the center point of the spheres in the environment.



(a) (b)

Figure 5.6. (a) 3D view specified for an environment, with a view reference coordinate system (b) View reference coordinate system aligned with the world coordinate system

Also, to illuminate a solid 3D environment a light source is needed. The light source is modeled as a point light source, where rays of light emanate from the light source and impact objects in the environment – Figure 5.12. A light source is defined by a located at (x_{ls}, y_{ls}, z_{ls}) in the environment. The light source is discussed in great detail in section 5.8; however, what is important in this section to know is that the light source has an actual location in the environment. Therefore, the light source location needs to be adjusted just like the vertex points of the polygons and the center points of the spheres in the environment.

A simple **align environment algorithm**, using the algorithms from Chapter 4, for a 3D view – defined by the orientations α and β, the view reference point, and the center of projection – is outlined in the following steps:

1. Given α and β, find the vectors $\vec{u}, \vec{v}$, and $\vec{n}$ that defines the 3D view using the **3D view algorithm** from section 4.4
2. For every sphere in the environment, adjust its center point using the **3D view-alignment algorithm** from section 4.5
3. For every polygon in the environment, adjust its vertex points using the **3D view-alignment algorithm** from section 4.5
4. For the light source in the environment, adjust its location using the **3D view-alignment algorithm** from section 4.5

5.3 View Window

Once the 3D view is specified and aligned, the colors of the 3D objects are projected onto the view plane. Before the colors are projected, a *view window* is defined on the view plane – Figure 5.7(a). The view window is a rectangular

grid of pixels defined by a point in the top left corner of the grid (x_{min}, y_{max}, z), and a point in the bottom right corner of the grid (x_{max}, y_{min}, z); the z value is the location of the view plane. The first pixel on the window is (x_{min}, y_{max}, z), and the last pixel on the window is (x_{max}, y_{min}, z). The pixel locations on the grid are defined in the XY world coordinate system.

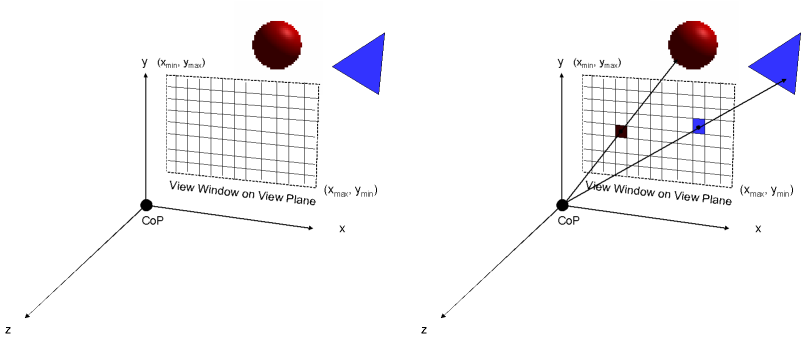


Figure 5.7. (a) View window on view plane (b) Ray tracing example. The color of a pixel on view window is determined by which objects its ray intersects.

5.4 Ray Tracing

The color of an object's surface is projected on a view window. This projection is accomplished by ray tracing. Ray-tracing traces imaginary rays of light from the center of projection through the view window on the view plane to objects in the environment – Figure 5.7(b). Whatever object the ray intersects, the color of the intersection point on the object is the color of the pixel, which the ray passed through on the view window.

Rays are defined as 3D lines that pass through the center of projection (x_{cp}, y_{cp}, z_{cp}) and pixels (x_w, y_w, z_w) on the view window. The ray equations are

$$x = x_{cp} + t \cdot (x_w - x_{cp}) \quad (5.17)$$

$$y = y_{cp} + t \cdot (y_w - y_{cp}) \quad (5.18)$$

$$z = z_{cp} + t \cdot (z_w - z_{cp}). \quad (5.19)$$

Note that after the **align environment algorithm** *is used for a center of projection at* $(0, 0, d_n)$*, the center of projection after alignment is* $(0, 0, 0)$*, and the* z_w *value of the view plane is* $-d_n$*.*

5.4.1 Sphere Intersection

From the ray Equations 5.17, 5.18, and 5.19, the intersection with a ray and a sphere can be calculated. The goal is to find the value of t for the intersection point – *if there is an intersection* – of the ray with the sphere.

The value of t is found by substituting ray Equations 5.17, 5.18, and 5.19 into the sphere Equation 5.1 for x, y, and z,

$$\begin{aligned}(x_{cp} + t \cdot (x_w - x_{cp}) - a)^2 + (y_{cp} + t \cdot (y_w - y_{cp}) - b)^2 + \\ (z_{cp} + t \cdot (z_w - z_{cp}) - c)^2 = r^2,\end{aligned} \tag{5.20}$$

and solving for t. Equation 5.20 is a quadratic equation, therefore, the quadratic formula is used to solve for t. The quadratic formula for t is

$$t = \frac{-B \pm \sqrt{B^2 - 4 \cdot A \cdot C}}{2 \cdot A}, \tag{5.21}$$

where

$$A = (x_w - x_{cp})^2 + (y_w - y_{cp})^2 + (z_w - z_{cp})^2 \tag{5.22}$$

$$\begin{aligned}B &= 2[(x_w - x_{cp})(x_{cp} - a) + (y_w - y_{cp})(y_{cp} - b) \\ &+ (z_w - z_{cp})(z_{cp} - c)]\end{aligned} \tag{5.23}$$

$$C = (x_{cp} - a)^2 + (y_{cp} - b)^2 + (z_{cp} - c)^2 - r^2. \tag{5.24}$$

The quadratic equation 5.20 gives two solutions for t. Also, the solutions can be real or imaginary. An imaginary solution occurs when the value of $B^2 - 4 \cdot A \cdot C$ from Equation 5.21 is less than 0.

If both solutions are imaginary, then the ray and the sphere do not intersect – Figure 5.8(a). If there is only one real solution, then the ray only grazes the sphere and there is not an intersection – Figure 5.8(b). For an intersection to occur, Equation 5.20 must produce two different real solutions, and the solution with the smallest value of t is the intersection point – Figure 5.8(c). The larger solution is where the ray exits the sphere.

If there is an intersection, the intersection point is found by plugging the value of t, where the intersection occurred, into Equations 5.17, 5.18, and 5.19. These equations gives the (x, y, z) location of the intersection point on the sphere. *Note that the value of t should be greater than zero. If t is negative, the intersection is behind the center of projection and should not be counted as an intersection.*

A simple **sphere intersection algorithm** to determine if (and where) a ray intersects a sphere is outlined in the following steps:

1. Insert ray Equations 5.17, 5.18, and 5.19 into the sphere Equation 5.1

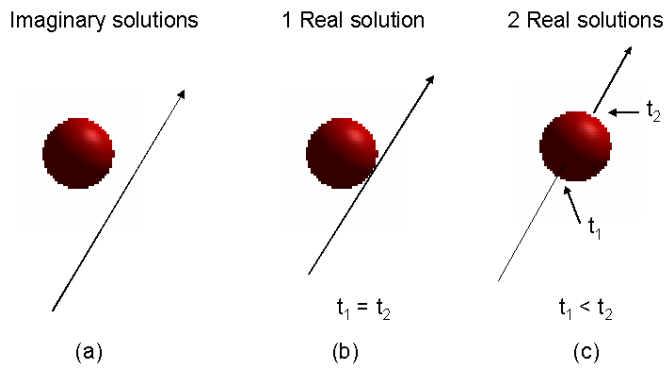


Figure 5.8. (a) No intersection between ray and Sphere (b) Only grazes the sphere, so no intersection (c) Ray intersects sphere

2. Solve for t using Equations 5.21, 5.22, 5.23, and 5.24
3. If both solutions of t are imaginary, then there is not an intersection with the ray and the sphere
4. If both solutions are real and if *any* solution is less than zero, then there is not an intersection
5. If both solutions of t are real and are equal, then there is not an intersection
6. If both solutions of t are real, different, and greater than 0, then there is an intersection
   (a) Find the intersection point by plugging the smaller solution of t into Equations 5.17, 5.18, and 5.19

5.4.2 Polygon Intersection

Determining if a ray intersects a polygon is a two step process. The first step is to determine if the ray intersects the plane,

$$A \cdot x + B \cdot y + C \cdot z + D = 0,$$

which the polygon lies in. From the vertex points of the polygon, the A, B, C, and D parameters of the plane can be found using the **polygon plane algorithm** in section 5.1.2.

From the ray definition in Equations 5.17, 5.18, and 5.19, the intersection of a ray and the plane of the polygon is calculated by substituting Equations 5.17, 5.18, and 5.19 into the plane Equation 5.2 for x, y, and z,

$$\begin{aligned} A(x_{cp} + t \cdot (x_w - x_{cp})) + B(y_{cp} + t \cdot (y_w - y_{cp})) + \\ C(z_{cp} + t \cdot (z_w - z_{cp})) + D = 0. \end{aligned} \tag{5.25}$$

Next the value of t is solved for

$$t = -\frac{A \cdot x_{cp} + B \cdot y_{cp} + C \cdot z_{cp} + D}{A(x_w - x_{cp}) + B(y_w - y_{cp}) + C(z_w - z_{cp})}. \quad (5.26)$$

If the denominator, $A(x_w - x_{cp}) + B(y_w - y_{cp}) + C(z_w - z_{cp})$, of Equation 5.26 is 0, the ray and the plane of the polygon do not intersect [6] . Therefore, the ray does not intersect the polygon.

However, if the denominator is not 0 and the value of t is *greater than zero*, then the intersection point of the ray and the plane is found by inserting the value of t from Equation 5.26 into Equations 5.17, 5.18, and 5.19. Given the intersection point, the second step is to determine if the intersection point on the plane is within the boundary of the polygon. This is accomplished by using the **on surface polygon algorithm** from section 5.1.3, which computes if a point is within the boundary of a polygon.

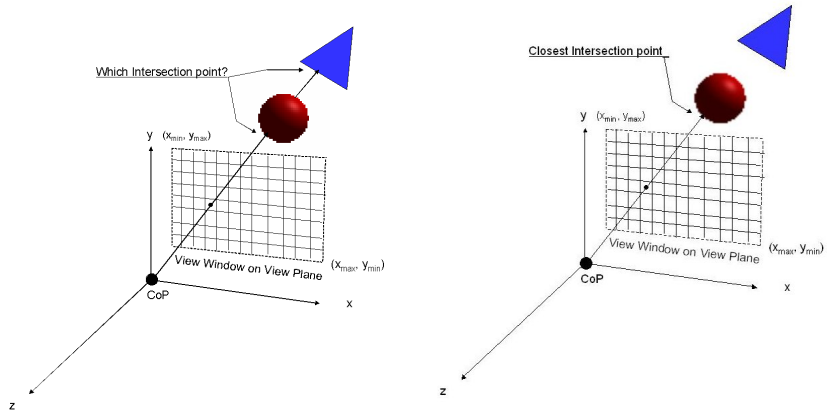
A simple **polygon intersection algorithm** to determine if (and where) a ray intersects a polygon is outlined in the following steps:

1. Find the plane of the polygon, from the vertex points of the polygon using the **polygon plane algorithm** in section 5.1.2
2. Insert ray Equations 5.17, 5.18, and 5.19 into the plane equation as in Equation 5.25
3. Solve for t using Equation 5.26
4. If the denominator, $A(x_w - x_{cp}) + B(y_w - y_{cp}) + C(z_w - z_{cp})$, of Equation 5.26 is 0, the ray and the plane of the polygon do not intersect
5. If the value of t is less than zero, then the ray and the plane of the polygon do not intersect
6. If the denominator is not 0 and t is greater than zero
   (a) Find the intersection point of the ray and the plane by inserting the value of t into ray Equations 5.17, 5.18, and 5.19
   (b) Determine if the intersection point is within the boundary of the polygon, by using the **on surface polygon** algorithm from section 5.1.3
      i. If intersection point is within the boundary of the polygon, then the ray and the polygon do intersect
      ii. If intersection point is not within the boundary of the polygon, then the ray and the polygon do not intersect

[6] This situation only occurs when the ray and the plane are parallel to each other.

5.4.3 Visible Surface Determination

The final step in ray tracing is to determine which surface in the environment is visible to the ray being casted. From the point of view of the ray, it doesn't know that multiple objects are in the environment. If multiple surfaces are in the direct path of the ray, as Figure 5.9(a) shows, a ray will intersect multiple surfaces.



(a) (b)

Figure 5.9. (a) A ray can intersection two surfaces (b) The surface of intersection is determined by which intersection point is closest.

With multiple intersection points, only one surface can be displayed on the view window. The surface that is displayed should be the one with the closest intersection point – Figure 5.9(b). Therefore, distances between the center of projection (x_{cp}, y_{cp}, z_{cp}) and the intersection points (x_i, y_i, z_i) for $i = 1..n$ (n is the number of intersections) are computed

$$d_i = \sqrt{(x_i - x_{cp})^2 + (y_i - y_{cp})^2 + (z_i - z_{cp})^2}. \tag{5.27}$$

Equation 5.27 is the distance between two points. The intersection point (x_i, y_i, z_i) with the closest distance d_i is the visible surface for the ray and that surface is displayed on the view window.

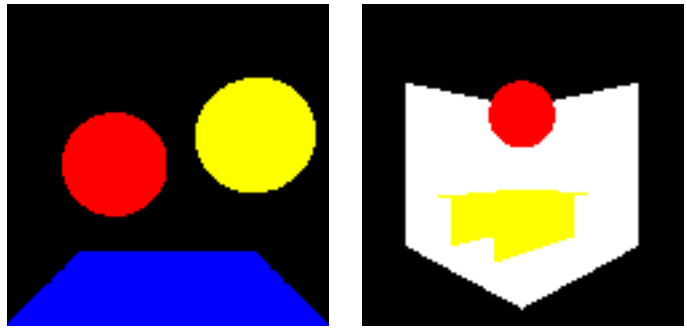
5.4.4 Ray Tracing Algorithm

The following is a **ray tracing algorithm** to determine which surface (*if any*) a ray – from the center of projection to a pixel on the view window – intersects in a solid 3D environment composed of spheres and polygons:

1. Compute the ray from the center of projection to a pixel on the view window using Equations 5.17, 5.18, and 5.19
2. For each sphere in the environment
   (a) Use the **sphere intersection algorithm** from section 5.4.1 to determine if (and where) the ray intersect the sphere
   (b) If there is an intersection, compute the distance between the intersection point and the center of projection using Equation 5.27
3. For each polygon in the environment
   (a) Use the **polygon intersection algorithm** from section 5.4.2 to determine if (and where) the ray intersect a polygon
   (b) If there is an intersection, compute the distance between the intersection point and the center of projection using Equation 5.27
4. If there is an intersection, the intersection point is the one with the closest distance

The **ray tracing algorithm** is used for every ray – remember rays are defined for the center of projection and *every* point on the view window. For example, if the pixel in the top left corner of the view window is $(x_{min} = -30, y_{max} = 30)$ and the pixel in the bottom right corner is $(x_{max} = 30, y_{min} = -30)$, then the x length of the view window is $x_{max} - x_{min} + 1 = 61$ and the y length is $y_{max} - y_{min} + 1 = 61$. Therefore, the view window is 61x61, so there are a total of 3,721 rays.

Figure 5.10 shows results of using the **ray tracing algorithm** on the solid 3D environments from Figure 5.1. Notice that without the lighting models or shadow model, which we will discuss shortly, the images appear flat. The lighting and shadow models add *substance* to the images.

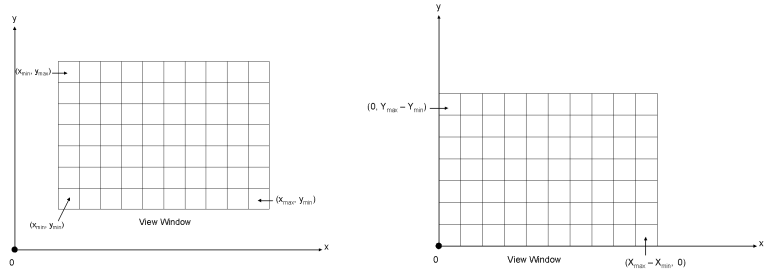


(a) (b)

Figure 5.10. Example lighting of solid 3D environments with only ray tracing (no lighting models used) (a) Environment from Figure 5.1(a) (b) Environment from Figure 5.1(b)

5.5 Displaying View Window

The view window is a 2D grid of pixels – Figure 5.7 – but the grid is located on the view plane which is in the XYZ coordinate system. We need to convert the view window to a 2D image grid in the XY coordinate system. This is accomplished by dropping the z value from the coordinates of the view window – Figure 5.11(a). Now that the view window is in the XY coordinate system, it is translated to the origin by translating each pixel by $(-x_{min}, -y_{min})$ – Figure 5.11(b). The translation values $(-x_{min}, -y_{min})$ are used because we want to move the bottom left corner (x_{min}, y_{min}) of the view window to the origin of the XY coordinate system.



(a) (b)

Figure 5.11. (a) View Window without the z value (b) View Window translated to the origin

Using the image-formation information in Chapter 1, the color contents of the view window are displayed as an image. The x length of the image is

$$x_L = x_{max} - x_{min} + 1, \tag{5.28}$$

and the y length of the image is

$$y_L = y_{max} - y_{min} + 1. \tag{5.29}$$

Therefore, the image size is x_Lxy_L.

A simple **display window algorithm** to display the color contents of a view window whose size is defined by corners (x_{min}, y_{max}, z) and (x_{max}, y_{min}, z) is outlined in the following steps:

1. Drop off the z values of the view window's coordinates
2. Translate each coordinate in the view window by $(-x_{min}, -y_{min})$
3. Using the image-formation information in Chapter 1 to display the color contents of the view window

5.6 Scaling Environment

The size of the image created depends on the size of the view window, whose size is defined based on the amount of the 3D environment to be viewed. If the size of the view window is increased or decreased, then the image increases or decreases, respectively. However, the contents of the view window changes. More of the environment is projected onto the window if the view window size is increased. On the other hand, if the view window's size is decreased, then less of the environment is seen.

If the goal is to only make the image bigger or smaller, then changing the size of the view window is not our best option. Instead, we should scale the entire environment. If we desire an image twice as big, then the environment is scaled by 2, and so on. In general, whatever scale (sf) increase or decrease of the image that is desired, the environment is scaled by sf.

Before the environment is scaled, the view reference coordinate system is aligned with the world coordinate system as describe in section 5.2. After alignment, the transformed environment is scaled by scaling every vertex point (x_p, y_p, z_p) of a polygon by sf

$$x'_p = x_p \cdot sf \tag{5.30}$$
$$y'_p = y_p \cdot sf \tag{5.31}$$
$$z'_p = z_p \cdot sf. \tag{5.32}$$

For a sphere, the center point (x_s, y_s, z_s) is scaled by sf

$$x'_s = x_s \cdot sf \tag{5.33}$$
$$y'_s = y_s \cdot sf \tag{5.34}$$
$$z'_s = z_s \cdot sf. \tag{5.35}$$

Also, the radius r is scale by sf

$$r' = r \cdot sf. \tag{5.36}$$

In addition to scaling the environment, the corners of the view window (x_{min}, y_{max}, z) and (x_{max}, y_{min}, z) are scale by sf

$$x'_{min} = x_{min} \cdot sf \tag{5.37}$$
$$y'_{max} = y_{max} \cdot sf \tag{5.38}$$
$$z' = z \cdot sf \tag{5.39}$$
$$x'_{max} = x_{max} \cdot sf \tag{5.40}$$
$$y'_{min} = y_{min}. \cdot sf \tag{5.41}$$

The view window needs to be scaled because it defines what part of the environment is seen. The scaling operation changes not only the size of objects in the environment, but the their location also. The view window is scaled, so that the correct part of the environment will be projected onto the view window after scaling. Finally, the light source location (x_{ls}, y_{ls}, z_{ls}) needs to be scaled by sf

$$x'_{ls} = x_{ls} \cdot sf \quad (5.42)$$

$$y'_{ls} = y_{ls} \cdot sf \quad (5.43)$$

$$z'_{ls} = z_{ls} \cdot sf. \quad (5.44)$$

The light source location is scaled in order to maintain the relative location of the light source with respect to the objects in the environment.

A simple **scale environment algorithm** to scale an environment, view window, and light source location by sf (*after section 5.2's alignment operations*) is outlined in the following steps:

1. For each polygon
   (a) Scale each vertex point sf using Equations 5.30, 5.31, and 5.32
2. For each sphere
   (a) Scale each center point by sf using Equations 5.33, 5.34, and 5.35
   (b) Scale each radius by sf using Equation 5.36
3. For the view window
   (a) Scale top left corner (x_{min}, y_{max}, z) by sf using Equations 5.37, 5.38, and 5.39
   (b) Scale bottom right corner (x_{max}, y_{min}, z) by sf using Equations 5.40, 5.41, and 5.39
   (c) Round all corner points to integer values
4. For the light source
   (a) Scale the location by sf using Equations 5.42, 5.43, and 5.44

Once the environment, view window, and the light source's location are scaled, rays are constructed from the center of projection to pixels on the *scaled* view-window as in section 5.4, and the *scaled* environment is projected onto the *scaled* view-window. From section 5.5, the **display window algorithm** is used next to create the image of the contents of the view window. The size of this *scaled* image is $(sf \cdot x_L - 1)$ x $(sf \cdot y_L - 1)$.

5.7 Partial Lighting Process

In this section we pause and create a *partial* lighting algorithm that displays an environment composed of spheres and polygons using the algorithms from

the previous sections. This is a *partial* algorithm because the lighting models in section 5.8 and shadow model in section 5.9 are not apart of this algorithm. As stated before, the lighting and shadow models add *substance* to images. However, for this algorithm, the objects are illuminated based only on the color of the object. The outline of the **partial lighting algorithm** is as follows

1. Define a 3D view by a view reference point (VRP), a center of projection (CoP), and orientation of the view plane given by α and β
2. Use the **align environment algorithm** in section 5.2 to adjust the location of the 3D objects in the environment
3. Defined the view window on the view plane as in section 5.3
4. Scale the aligned-environment and view window by sf using the **scale environment algorithm** from section 5.6
5. For each ray from the center of projection to the view window
   (a) Determine which surface (*if any*) a ray intersects in the environment using the **ray tracing algorithm** from section 5.4.4
   (b) If there is not an intersection point, set the color of the pixel on the view window to black [R = 0, G = 0, B = 0]
   (c) If there is an intersection, (x_i, y_i, z_i), set the color the of pixel to the color of the object intersected
6. Display the color contents of the view window using the **display window algorithm** from section 5.5

Figure 5.10 shows the results of using this algorithm on the solid 3D environments from Figure 5.1.

5.8 Lighting Models

Light is simulated to illuminate a solid 3D environment, and is modeled as a point light source, where rays of light emanate from the light source and impact object surfaces in the environment – Figure 5.12. When a light ray impacts a surface, the point of intersection with the surface is lit. Therefore, a light source located at (x_{ls}, y_{ls}, z_{ls}) in the environment lights the environment.

The color of an object is based on the amount of light absorbed by and reflected from the *material* of the surface. From section 1.2, we learned that the human eye can only perceive colors within the visible light spectrum, and mixtures of these colors produce the different colors we see. The light rays from a light source contains all colors in the visible light spectrum giving a light ray a white color. However, when the light ray intersects the surface of an object, the material of the object will absorb and reflect colors in the light ray. The

mix of colors that are reflected hits our eye, and whatever mix of colors we see is the perceived color of the material. Therefore, the color of an object depends on what mix of colors is reflected to our eye.

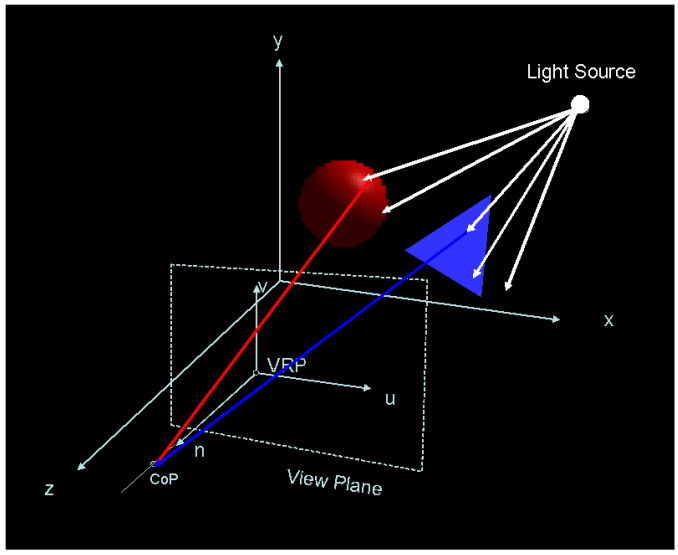


Figure 5.12. Example of a light source lighting an environment. Light rays (white) reflect off objects and the reflected color travels to the eye. The eye is the center of projection (CoP)

Figure 5.12 shows an example of light rays, which are white, reflecting off a sphere and a polygon. Because the *material* of the sphere absorbs all colors except red, red light rays are reflected to the eye – the eye is the center of projection. The *material* of the polygon absorbs all colors except blue, therefore, blue light rays are reflected to the center of projection.

A lighting model is a set of rules of how light-rays reflecting off surfaces in the environment effect the colors of those surfaces. The general method to determine the effect is to assume the surface absorbs the appropriate colors from the light ray to reflect the desired color of the object $[R, G, B]$.

With this assumption, the question that is addressed is how does the intensity (*i.e.; brightness*) of the surface changes based on the position of the light source with respect to the surface. Different positions of the light source with respect to an object effect how light reflects off the surface of that object, and the reflection determines the intensity I of the color of the object.

The intensity I is a number from 0 to 1. $I = 0$ means that the color is not seen, but $I = 1$ implies that the color is seen at full intensity. The value of I is computed from a lighting model, and is multiplied with the desired color of an

object $[R, G, B]$,

$$R' = R \cdot I \quad (5.45)$$
$$G' = G \cdot I \quad (5.46)$$
$$B' = B \cdot I, \quad (5.47)$$

to produce the *actual* color $[R', G', B']$ that is displayed on the view window.

Figure 5.1 is an example of lighting an environment with a lighting model, and Figure 5.10 shows the effects of not using a lighting model. As shown, lightings models give *depth* to images. In the following sections, three different lighting models – *Ambient Light, Diffuse Reflection*, and *Specular Reflection* – are discussed.

5.8.1 Ambient Light

Ambient light is a lighting model that models the effect of light scattering off all surfaces in the environment to light an object. Ambient light is passive and is applied to all surfaces equally. The value of the intensity ρ_a for ambient light is

$$\rho_a = I_a \cdot k_a, \quad (5.48)$$

where I_a is the intensity of the ambient light and k_a is the reflectivity of ambient light off the surface of the object. I_a and k_a are values from 0 to 1. If $I_a = 0$, there isn't any ambient light; if $k_a = 0$, the surface doesn't reflect ambient light. The value of I_a is the same for all objects in an environment, but k_a depends on the object.

If ambient light is the only source of light, the intensity I defined by the ambient light model for an object is

$$I = \rho_a. \quad (5.49)$$

The value of I is plugged into Equations 5.45, 5.46, 5.47 to get the *actual* color of the surface. Figure 5.13 shows examples of lighting the environment from Figure 5.1(a) using the ambient light model for $I_a = 1$, and $k_a = .2, .5, 1$ for all objects. As the value of k_a increases the brightness of the objects increase.

A simple **ambient light algorithm** to determine the light intensity, ρ_a, of a surface point based on the reflectivity coefficient k_a and the intensity of ambient light I_a is outlined in the following step:

1. Compute ρ_a from k_a and I_a using Equation 5.48

(a) (b) (c)

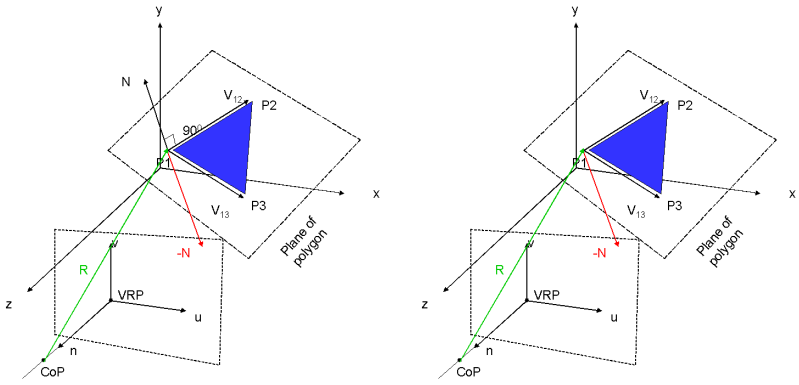
Figure 5.13. Ambient Lighting of Figure 5.1.a $I_a = 1$ (a) $k_a = .2$ for all objects (b) $k_a = .5$ for all objects (c) $k_a = 1$ for all objects

5.8.2 Diffuse Reflection

Diffuse reflection models the reflectivity of light off a surface based on the angle between the light ray and the normal of the surface. The surface normal, $\vec{N}_p$, for a polygon is the normal vector of the polygon's plane as computed from Equation 5.5 in section 5.1.2. However, the direction of the normal should be facing the center of projection – Figure 5.14(a). To determine if the normal vector is facing the center of projection, first the ray vector $\vec{R}$ from the center of projection (CoP) to the appropriate point on the view window (P_v) is computed

$$\vec{R} = P_v - CoP. \tag{5.50}$$

Second, the dot product between the ray vector and surface normal is computed, $\vec{R} \bullet \vec{N}_p$. If $\vec{R} \bullet \vec{N}_p > 0$, then the surface normal is not facing the center of projection; therefore, $-\vec{N}_p$ is used as the surface normal – Figure 5.14(b).



(a) (b)

Figure 5.14. (a) Must chose between the normal vectors $\vec{N}$ and $-\vec{N}$ (b) Because the normal vector is not facing the view point in this example, the normal vector $-\vec{N}$ is chosen.

The surface normal $\vec{N}_s$ for a sphere is simply the vector in the direction from the center point P_c of the sphere to a point P_s on the surface of the sphere

$$\vec{N}_s = P_s - P_c. \tag{5.51}$$

Now that surface normals for a polygon and sphere have been determined; the next step is to find the light ray vector $\vec{L}$ from a point of intersection at P_i on a surface to the light source at P_{ls},

$$\vec{L} = P_{ls} - P_i. \tag{5.52}$$

Using the light ray vector $\vec{L}$ and surface normal vector $\vec{N}$ – where $\vec{N} = \vec{N}_s$ or $\vec{N}_p$ – at the point of intersection, the intensity of the diffuse reflection is measured by the cosine of the angle ϕ between the two vectors – Figure 5.15.

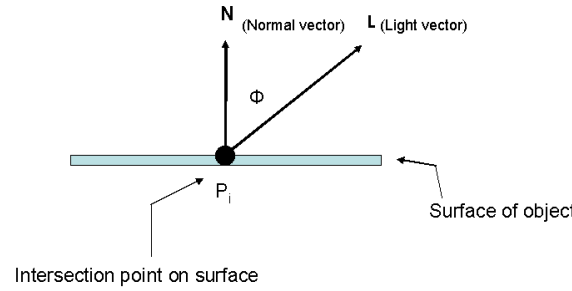


Figure 5.15. Diffuse reflection lighting model depends on the angle between the light ray and the normal of the surface

The cosine of the angle ϕ is computed by taking the dot product between $\vec{L}$ and $\vec{N}$ and dividing by the magnitude of each vector,

$$cos\phi = \frac{\vec{L} \bullet \vec{N}}{|\vec{L}||\vec{N}|}. \tag{5.53}$$

The magnitude of a vector $\vec{v} = (v_x, v_y, v_z)$ is the length of the vector and is computed as follows

$$|\vec{V}| = \sqrt{v_x^2 + v_y^2 + v_z^2}. \tag{5.54}$$

For the light source to illuminate the intersection point on the surface of an object, the angle ϕ must be in the range from 0 to 90 degrees. Angles bigger than 90 degrees means that the light source is behind the surface. To avoid this situation, the maximal value between Equation 5.53 and 0 is used in the following equation to find the intensity ρ_d of the diffuse reflection

$$\rho_d \quad = \quad I_{ls} \cdot k_d \cdot max(0, \frac{\vec{L} \bullet \vec{N}}{|\vec{L}||\vec{N}|}), \tag{5.55}$$

where I_{ls} is the intensity of the light source and k_d is the reflectivity of light off the surface of the object. I_{ls} and k_d are values from 0 to 1. If $I_{ls} = 0$, the light source is turned off; if $k_d = 0$, the surface doesn't reflect light from the light source. The value of I_{ls} is the same for all objects in an environment, but k_d depends on the object.

If diffuse reflection is the only source of illumination, the intensity I defined by the diffuse reflection model for an object is

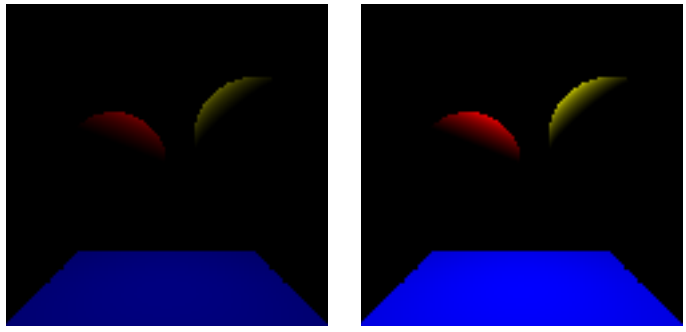
$$I = \rho_d. \tag{5.56}$$

The value of I is plugged into Equations 5.45, 5.46, 5.47 to get the *actual* color of the surface. Figure 5.16 shows examples of lighting the environment from Figure 5.1(a) using the diffuse reflection model for $I_{ls} = 1$, and $k_d = .5$, for all objects. As the value of k_d increases from .5 to 1, the brightness of the objects increase. However, notice that because the light source is behind the two spheres, the back side of the spheres are unilluminated.

Adding the ambient light and diffuse reflection models together changes the appearance of an image. The intensity I for the combined models using Equations 5.48 and 5.55 is

$$I = \rho_a + \rho_d. \tag{5.57}$$

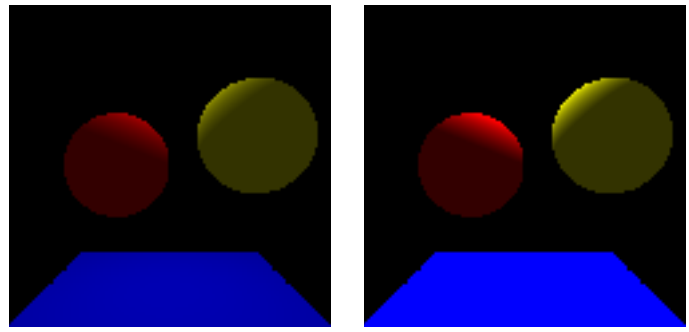
Figure 5.17 shows examples of adding ambient light to Figure 5.16 for $k_a = .2$. The images are brighter because ambient light adds additional illumination to the environment.



(a) (b)

Figure 5.16. Diffuse Reflection Lighting Model of Figure 5.1.a $I_{ls} = 1$ (a) $k_d = .5$ for all objects (b) $k_d = 1$ for all objects

A simple **diffuse reflection algorithm** to determine the light intensity ρ_d of a surface point based on the reflectivity coefficient k_d, the intensity of the



(a) (b)

Figure 5.17. Ambient Light and Diffuse Reflection Lighting Models of Figure 5.1.a $I_{ls} = 1$, $I_a = 1$, $k_a = .2$ (a) $k_d = .5$ for all objects (b) $k_d = 1$ for all objects

light source I_{ls}, the light ray vector $\vec{L}$, and the normal vector $\vec{N}$ of the surface is outlined in the following steps:

1. If the surface point is from a polygon
   (a) Compute the dot product between the ray vector $\vec{R}$ and the normal vector $\vec{N} = (A, B, C)$
   (b) If the dot product is greater than zero, then negate the normal vector: $\vec{N} = (-A, -B, -C)$
2. For the reflectivity constant k_d, the intensity of the light source I_{ls}, the light ray vector $\vec{L}$, and the normal vector $\vec{N}$, compute the intensity ρ_d at the surface point using Equation 5.55

5.8.3 Specular Reflection

Unlike diffuse reflection and ambient light models that scale the color values of an object, specular reflection ρ_s adds color to the color of an object $[R, G, B]$,

$$R' = R + \rho_s \tag{5.58}$$
$$G' = G + \rho_s \tag{5.59}$$
$$B' = B + \rho_s, \tag{5.60}$$

to produce the *actual* color $[R', G', B']$ that is displayed on the view window.

Specular reflection can be viewed on any shiny surface. For example, when an apple is illuminated with light, at some locations on the surface of the apple, you may notice a bright spot that looks almost white. This effect is caused by specular reflections. Shiny materials don't absorb light, but reflect it. Therefore, your eye sees all the colors of the visible light spectrum, which gives the material

a white color. Figure 5.18 is an example of a specular reflection on a red sphere – notice the white spot on the sphere.

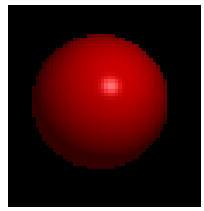


Figure 5.18. Specular reflection example

Although, shiny materials reflect light, they reflect light unequally in different directions. For a perfectly shiny surface, depending on your point of view of the object, you may or may not see a specular reflection. This is because the angle θ between the reflected light vector $\vec{RL}$ (from a point on the object) and your view-point vector $\vec{V}$ (to the point on the object) needs to be 0 in order to see the specular reflection – Figure 5.19. For a non-perfectly shiny surface the amount of specular reflection decrease as the angle θ increases.

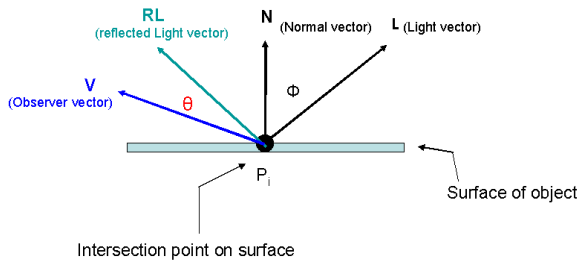


Figure 5.19. Specular reflection lighting

Researcher Phong Bui-Tuong[7] developed a specular reflection model to capture the effects of specular reflection as the angle θ changes. If the light source is white, the specular reflection added to each color channel is some percentage of 255. Phong's model gives the amount of 255 to be

$$255 \cdot I_{ls} \cdot k_s \cdot (cos\theta)^n, \tag{5.61}$$

where I_{ls} is the intensity of the light source (from 0 to 1), k_s is the reflectivity of specular reflection from the surface (from 0 to 1), and n is an exponent that controls how *fast* the amount of specular reflection decreases as θ gets bigger. The value of n varies from 1 to infinite. $n = 1$ creates a broad specular reflection,

[7]Bui-Tuong, Phong, "Illumination for Computer Generated Pictures," CACM, 18(6), June 1975, 311-317.

whereas, for large values of n, the specular reflection is very sharp. The value of n is 30 for Figure 5.18. For a perfect reflecting surface n is infinite.

The $cos\theta$ is the dot product between the light reflection vector $\vec{RL}$ and the viewer vector $\vec{V}$. However, instead of calculating the reflection vector $\vec{RL}$, using geometry the $cos\theta$ in terms of surface normal vector $\vec{N}$, light ray vector $\vec{L}$, and viewer vector $\vec{V}$ is

$$cos\theta = [2\frac{\vec{N}}{|\vec{N}|}(\frac{\vec{N}}{|\vec{N}|} \bullet \frac{\vec{L}}{|\vec{L}|}) - \frac{\vec{L}}{|\vec{L}|}] \bullet \frac{\vec{V}}{|\vec{V}|}. \tag{5.62}$$

The viewer vector $\vec{V}$ is the ray vector $\vec{R}$ from the center of the projection CoP to the appropriate point on the view window P_v from Equation 5.50, but in the opposite direction. Therefore, $\vec{V} = -\vec{R}$ and Equation 5.62 becomes

$$cos\theta = [2\frac{\vec{N}}{|\vec{N}|}(\frac{\vec{N}}{|\vec{N}|} \bullet \frac{\vec{L}}{|\vec{L}|}) - \frac{\vec{L}}{|\vec{L}|}] \bullet \frac{-\vec{R}}{|-\vec{R}|}. \tag{5.63}$$

The final consideration for the specular reflection model is to limit the angle θ to be in the range from 0 to 90 degrees for the same reason the angle ϕ was limited to be in the range from 0 to 90 degrees in section 5.8.2. As in section 5.8.2, this is accomplished be taking the maximal value between 0 and $cos\theta$. With this, the specular reflection model is

$$\rho_s = 255 \cdot I_{ls} \cdot k_s \cdot [max(0, cos\theta)]^n, \tag{5.64}$$

where $cos\theta$ is determined from Equation 5.63. Figure 5.20 gives examples of the specular reflection model for $I_{ls} = 1$, $k_s = .5$, and n is 3, 15, and 30. As the value of n increases the specular reflection becomes more focused.

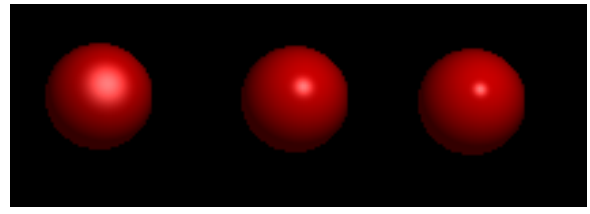


Figure 5.20. Specular reflection examples for $k_s = .5$ and $I_{ls} = 1$. From left to right n is 3, 15, and 30.

A simple **specular reflection algorithm** to determine how much specular reflection, ρ_s, to added to a surface point based on the reflectivity coefficient k_s, the intensity of the light source I_{ls}, the exponent n, the light ray vector $\vec{L}$, the normal vector $\vec{N}$, and the ray vector $\vec{R}$ is outlined in the following steps:

1. If the surface point is from a polygon
   (a) Compute the dot product between the ray vector $\vec{R}$ and the normal vector $\vec{N} = (A, B, C)$
   (b) If the dot product is greater than zero, then negate the normal vector: $\vec{N} = (-A, -B, -C)$ (as in section 5.8.2)
2. Using the light reflection vector $\vec{L}$, the normal vector $\vec{N}$, and the ray vector $\vec{R}$, compute the $cos\theta$ using Equation 5.63
3. For the reflectivity constant k_s, the intensity of the light source I_{ls}, the $cos\theta$, and the exponent n, compute the amount of specular reflection ρ_s at the surface point using Equation 5.64

5.8.4 Combine Models

Combining the ambient light model (Equation 5.48), diffuse reflection model (Equation 5.55), and specular reflection model (Equation 5.64), the **Lighting Model** for a 3D solid environment to compute the *actual* color $[R', G', B']$ for a surface of color $[R, G, B]$ is

$$R' = R \cdot (\rho_a + \rho_d) + \rho_s \quad (5.65)$$

$$G' = G \cdot (\rho_a + \rho_d) + \rho_s \quad (5.66)$$

$$B' = B \cdot (\rho_a + \rho_d) + \rho_s. \quad (5.67)$$

Figure 5.21 is an example of using the Lighting Model to light the environment from Figure 5.1(b).

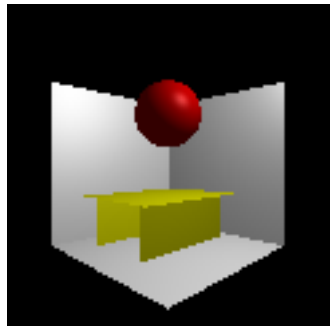


Figure 5.21. Environment from Figure 5.1(b) is illuminated by the Lighting Model (Equations 5.65, 5.66, and 5.67) for $k_a = .2$, $k_d = .6$, $k_s = .25$ for all objects. $n = 5$ for polygons and $n = 10$ for sphere.

5.9 Shadows

Figure 5.21 is illuminated using the Lighting Model from section 5.8.4; however, shadows should appear in the image as it does in Figure 5.1(b) because some objects are blocked from the light source by other objects. When an object is blocked, the object is in a shadow.

5.9.1 Finding Shadows

To determine if an object is in a shadow, the light ray from the intersection point (x_i, y_i, z_i) on the object to the light source at (x_{ls}, y_{ls}, z_{ls}) is first computed,

$$x = x_i + t \cdot (x_{ls} - x_i) \tag{5.68}$$

$$y = y_i + t \cdot (y_{ls} - y_i) \tag{5.69}$$

$$z = z_i + t \cdot (z_{ls} - z_i). \tag{5.70}$$

The second step is to determine if the light ray intersects another surface in the environment. Because the light ray has the same form as the equations of the ray from the center of projection to the view window (Equations 5.17, 5.18, and 5.19), a simple algorithm similar to the **ray tracing algorithm** from section 5.4.4 may be developed to compute if there is an intersection with the light ray and other objects in the environment. If there is an intersection, the value of t of the intersection point must be from 0 to 1 for the light ray to be block.

A simple **shadow ray-tracing algorithm** to determine if the light ray – from an intersection point to the light source – is blocked by another object in the environment is outlined as follows:

1. For each sphere in the environment (*not including the object the intersection point belongs to*)
   (a) Use the **sphere intersection algorithm** from section 5.4.1 to determine if the light ray intersects the sphere – (*Use Equations 5.68, 5.69, and 5.70 in the algorithm instead of Equations 5.17, 5.18, and 5.19.*)
   (b) If there is an intersection
      i. Find the value of t of the intersection point
      ii. If the value of t is > 0 and ≤ 1, the light ray is blocked
      iii. If the value of t is ≤ 0 or > 1, the light ray is not blocked
2. For each polygon in the environment (*not including the object the intersection point belongs to*)

(a) Use the **polygon intersection algorithm** from section 5.4.2 to determine if (and where) the light ray intersects a polygon – (*Use Equations 5.68, 5.69, and 5.70 in the algorithm instead of Equations 5.17, 5.18, and 5.19.*)

(b) If there is an intersection

i. Find the value of t of the intersection point

ii. If the value of t is > 0 and ≤ 1, the light ray is blocked

iii. If the value of t is ≤ 0 or > 1, the light ray is not blocked

3. If step 1 or 2 results in an intersection and the value of t of the intersection is from 0 to 1, then the light ray is blocked

5.9.2 Shadow Model

If the light ray is blocked by another surface, then the point (x_i, y_i, z_i) is in a shadow. Therefore, the light source intensity I_{ls} is 0 and the specular reflection $\rho_s = 0$. However, to create softer shadows the light source intensity can just be lowered to a value of I_{shadow}. The **Shadow Lighting Model** is

$$R' = R \cdot (\rho_a + \rho_d) + sh \cdot \rho_s \quad (5.71)$$

$$G' = G \cdot (\rho_a + \rho_d) + sh \cdot \rho_s \quad (5.72)$$

$$B' = B \cdot (\rho_a + \rho_d) + sh \cdot \rho_s, \quad (5.73)$$

where sh is 0 if there is a shadow and 1 if there isn't. Also $I_{ls} = I_{shadow}$. Figure 5.1 is example of using the shadow lighting model for $I_{shadow} = .5$, $I_a = 1$, $k_a = .2$, $k_d = .6$, and $k_s = .25$. Specifically for Figure 5.1(a), $n = 10$ for the red sphere and blue polygon, $n = 5$ for the yellow sphere. For Figure 5.1(b), $n = 10$ for sphere and $n = 5$ for all polygons.

5.10 Complete Lighting Process

This section combines the lighting techniques from section 5.8 and the shadow technique from section 5.9 to create an algorithm to compute the *actual* color display on the view window. This algorithm determines the color of a surface based on the lighting techniques and whether the surface is blocked by another surface. In addition, a final algorithm to light an environment is presented. This algorithm merges all the algorithms in this chapter to light and display a 3D environment composed of spheres and polygons.

5.10.1 Actual Color

Given a light source located at P_{ls}; an intersection point of (x_i, y_i, z_i) on an object; the RGB color of the intersection point; a point on the view window P_v that the intersection point maps to; and a determination[8] if intersection point is in shadow or not, a simple **actual color algorithm** to determine the color of the surface is outlined in the the following steps:

1. Compute the ray vector $\vec{R}$ using Equation 5.50 for the Center of Projection at CoP and the point P_v on the view window
2. Find the normal vector $\vec{N}$ for the intersection point
   (a) If the intersection point is from a polygon
      i. The normal vector, $\vec{N}$, for a polygon is the normal vector from the equation of the plane the polygon lies in (as computed from Equation 5.5 in section 5.1.2, $\vec{N} = (A, B, C)$)
   (b) If the intersection point is from a sphere
      i. Find the normal vector $\vec{N}$ using Equation 5.51 (where $P_s = (x_i, y_i, z_i)$)
3. Compute the light ray vector $\vec{L}$ from the intersection point to the light source using Equation 5.52
4. If the intersection point is in a shadow, then $sh = 0$ and $I_{ls} = I_{shadow}$
5. If the intersection point is not in a shadow, then $sh = 1$ and $I_{ls} = 1$
6. Set the ambient light constant I_a to 1
7. Compute the Ambient Light intensity ρ_a using the **ambient light algorithm** from section 5.8.1
8. Compute the Diffuse Reflection intensity ρ_d using the **diffuse reflection algorithm** from section 5.8.2
9. Compute the amount of Specular Reflection ρ_s using the **specular reflection algorithm** from section 5.8.3
10. Using sh, ρ_a, ρ_d, and ρ_s, determine the *actual* color $[R', G', B']$ of the intersection point, (x_i, y_i, z_i), using Equations 5.71, 5.72, and 5.73
11. Round R', G', and B' to the nearest integer value
12. *The max value of R', G', or B' is 255, so clip all value to a max of 255.*

[8] This determination is made using the **shadow ray-tracing algorithm** from section 5.9.1.

5.10.2 Lighting Environment

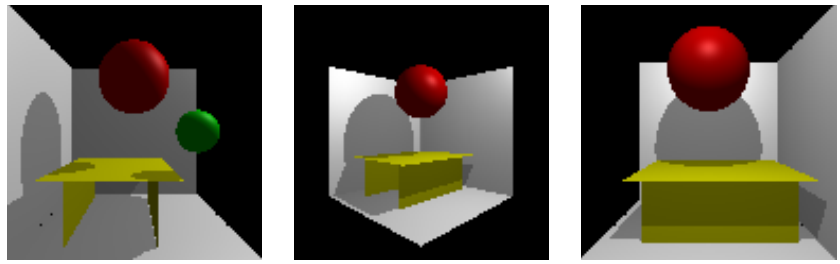
Combining the necessary algorithms in this chapter, a simple **lighting algorithm** to light and display a 3D environment composed of spheres and polygons is outlined as follows:

1. Define a 3D view by a view reference point (VRP), a center of projection (CoP), and orientation of the view plane given by α and β
2. Use the **align environment algorithm** in section 5.2 to adjust the location of the 3D objects in the environment and the light source
3. Defined the view window on the view plane as in section 5.3
4. Scale the aligned environment and light source along with the view window by sf using the **scale environment algorithm** from section 5.6
5. For each ray from the center of projection to a point on the view window
   (a) Determine which surface (*if any*) a ray intersects in the environment using the **ray tracing algorithm** from section 5.4.4
   (b) If there is not an intersection point, set the color of the pixel on the view window to black [R = 0, G = 0, B = 0]
   (c) If there is an intersection, (x_i, y_i, z_i)
      i. Compute the light ray from the intersection point to the light source using Equations 5.68, 5.69, and 5.70
      ii. Determine if the intersection point at (x_i, y_i, z_i) is or isn't in a shadow, by checking if the light ray is blocked by another object in the environment using the **shadow ray-tracing algorithm** from section 5.9.1
      iii. Compute the *actual* color – $[R', G', B']$ – of the intersection point using the **actual color algorithm** in section 5.10.1
      iv. Set the color of the pixel on the view window for the ray to $[R', G', B']$
6. Display the color contents of the view window using the **display window algorithm** from section 5.5

Figure 5.22 is an example of using this lighting algorithm to view the environment in Figure 5.1(b) from three different view points.

5.11 Algorithm Summary and Testing

In this chapter, the process of viewing a three dimensional environment, composed of spheres and polygons, on a 2D image grid was discussed. The following is a list of algorithms developed in this chapter. In addition, there are input



(a) (b) (c)

Figure 5.22. Examples of view the environment in Figure 5.1(b) from three different view points

and output examples given for the algorithms to test your coding of some of the algorithms.

1. The **polygon plane algorithm** in section 5.1.2 finds the plane of a 3D polygon.
   - Input/Output Set 1
     - **Input**: Polygon vertex points: $P_1 = (50, 50, 50)$, $P_2 = (60, 70, 50)$, $P_3 = (80, 50, 50)$
     - **Output**: Plane equation: N = (A, B, C) = $(0, 0, 600)$, $D = -30000$
   - Input/Output Set 2
     - **Input**: Polygon vertex points: $P_1 = (50, 50, 70)$, $P_2 = (30, 40, 50)$, $P_3 = (25, 70, 30)$
     - **Output**: Plane equation: N = (A, B, C) = $(-800, 300, 650)$, $D = -20500$
2. The **on surface polygon algorithm** in section 5.1.3 computes if a 3D point is on the surface of a 3D polygon.
   - Input/Output Set 1
     - **Input**: Polygon vertex points: $P_1 = (50, 70, 50)$, $P_2 = (30, 70, 40)$, $P_3 = (25, 70, 70)$; Test Point = $(40, 70, 50)$
     - **Output**: Test Point on surface of polygon
   - Input/Output Set 2
     - **Input**: Polygon vertex points: $P_1 = (50, 50, 70)$, $P_2 = (30, 40, 70)$, $P_3 = (25, 70, 70)$; Test Point = $(50, 60, 70)$
     - **Output**: Test Point not on surface of polygon
   - Input/Output Set 3
     - **Input**: Polygon vertex points: $P_1 = (50, 70, 50)$, $P_2 = (30, 70, 40)$, $P_3 = (25, 70, 70)$; Test Point = $(120, 70, 100)$

  - **Output**: Test Point not on surface of polygon
- Input/Output Set 4
  - **Input**: Polygon vertex points: $P_1 = (50, 50, 70)$, $P_2 = (30, 40, 70)$, $P_3 = (25, 70, 70)$; Test Point = $(30, 40, 70)$
  - **Output**: Test Point on surface of polygon
- Input/Output Set 5
  - **Input**: Polygon vertex points: $P_1 = (10, 10, 70)$, $P_2 = (50, 10, 70)$, $P_3 = (30, 30, 60)$; Test Point = $(30, 10, 70)$
  - **Output**: Test Point on surface of polygon

3. The **align environment algorithm** in section 5.2 adjusts the locations of the 3D objects and the light source based on the 3D view desired.
   - Input/Output Set 1
     - **Input**: Polygon vertex points: $P_1 = (50, 50, 70)$, $P_2 = (30, 40, 50)$, $P_3 = (25, 70, 30)$; VRP = $(50, 50, 100)$, CoP = $(0, 0, 25)$, $\alpha = -25$, $\beta = 79$
     - **Output**: Adjusted Polygon vertex points: $P_1 = (-12.6785, -26.6897, -30.1880)$, $P_2 = (-39.2571, -38.0938, -22.2175)$, $P_3 = (-52.2410, -48.0884, -54.7218)$
   - Input/Output Set 2
     - **Input**: Sphere Center = $(100, 40, 32)$, VRP = $(50, 50, 100)$, CoP = $(0, 0, 25)$, $\alpha = -25$, $\beta = 79$
     - **Output**: Adjusted Sphere Center is $(16.5773, -83.1474, -30.9751)$
   - Input/Output Set 3
     - **Input**: 3D Environment: 1 Polygon – Polygon vertex points: $P_1 = (50, 50, 40)$, $P_2 = (60, 70, 40)$, $P_3 = (85, 50, 40)$; 1 Sphere – Center point = $(50, 50, 70)$, Radius = 20; Light source – $(50, 50, 100)$; VRP = $(50, 50, 150)$, CoP = $(0, 0, 25)$, $\alpha = 0$, $\beta = 0$
     - **Output**: Adjusted Polygon vertex points: $P_1 = (0, 0, -135)$, $P_2 = (10, 20, -135)$, $P_3 = (35, 0, -135)$; Adjusted Sphere Center is $(0, 0, -105)$; Adjusted Light Source is $(0, 0, -75)$

4. The **sphere intersection algorithm** in section 5.4.1 determines if (and where) a ray intersects a sphere.
   - Input/Output Set 1
     - **Input**: Center point = $(50, 50, 70)$, Radius = 20; For a center of projection at $(50, 50, 200)$ and pixel on view window at $(50, 50, 150)$
     - **Output**: The value of t of the intersection is 2.2, and the intersection point is $(50, 50, 90)$
   - Input/Output Set 2
     - **Input**: Center point = $(50, 50, 70)$, Radius = 20; For a center of projection at $(50, 0, 200)$ and pixel on view window at $(50, 50, 150)$
     - **Output**: There is not an intersection
   - Input/Output Set 3
     - **Input**: Center point = $(50, 50, 70)$, Radius = 20; For a center of projection at $(50, 35, 200)$ and pixel on view window at $(50, 35, 150)$
     - **Output**: The value of t of the intersection is 2.3354, and the intersection point is $(50, 35, 83.2288)$
5. The **polygon intersection algorithm** in section 5.4.2 determines if (and where) a ray intersects a polygon.
   - Input/Output Set 1
     - **Input**: Polygon vertex points: $P_1 = (50, 50, 50)$, $P_2 = (60, 70, 50)$, $P_3 = (85, 50, 50)$; For a center of projection at $(50, 50, 200)$ and pixel on view window at $(50, 50, 150)$
     - **Output**: The value of t of the intersection is 3, and the intersection point is $(50, 50, 50)$
   - Input/Output Set 2
     - **Input**: Polygon vertex points: $P_1 = (50, 50, 50)$, $P_2 = (60, 70, 50)$, $P_3 = (85, 50, 50)$; For a center of projection at $(50, 0, 200)$ and pixel on view window at $(50, 50, 150)$
     - **Output**: There is not an intersection
   - Input/Output Set 3
     - **Input**: Polygon vertex points: $P_1 = (50, 50, 50)$, $P_2 = (60, 70, 50)$, $P_3 = (85, 50, 50)$; For a center of projection at $(70, 55, 70)$ and pixel on view window at $(70, 55, 60)$
     - **Output**: The value of t of the intersection is 2, and the intersection point is $(70, 55, 50)$

6. The **ray tracing algorithm** in section 5.4.4 determines which surface (*if any*) a ray intersects in a 3D environment composed of spheres and polygons.
   - Input/Output Set 1
     - **Input**: 3D Environment: 1 Polygon – Polygon vertex points: $P_1 = (50, 50, 40)$, $P_2 = (60, 70, 40)$, $P_3 = (85, 50, 40)$; 1 Sphere – Center point = $(50, 50, 70)$, Radius = 20; For a center of projection at $(70, 55, 200)$ and pixel on view window at $(70, 55, 150)$
     - **Output**: The ray doesn't intersect the sphere, but it does intersect polygon for $t = 3.2$, and the intersection point is $(70, 55, 40)$
   - Input/Output Set 2
     - **Input**: 3D Environment: 1 Polygon – Polygon vertex points: $P_1 = (50, 50, 40)$, $P_2 = (60, 70, 40)$, $P_3 = (85, 50, 40)$; 1 Sphere – Center point = $(50, 50, 70)$, Radius = 20; For a center of projection at $(50, 50, 200)$ and pixel on view window at $(50, 50, 150)$
     - **Output**: The ray intersects the sphere and polygon, but the sphere intersection is closer. Therefore, the intersection with the sphere is at point $(50, 50, 90)$ for $t = 2.2$
7. The **display window algorithm** in section 5.5 displays the color contents of a view window on an image grid.
8. The **scale environment algorithm** in section 5.6 scales an environment and view window by a specified scale factor.
   - Input/Output Set
     - **Input**: 3D Environment: 1 Polygon – Polygon vertex points: $P_1 = (50, 50, 40)$, $P_2 = (60, 70, 40)$, $P_3 = (85, 50, 40)$; 1 Sphere – Center point = $(50, 50, 70)$, Radius = 20; Light source – $(50, 50, 100)$; VRP = $(50, 50, 150)$, CoP = $(0, 0, 25)$, $\alpha = 0$, $\beta = 0$; view-window – top left corner: $(-30, 30, z_w)$, Bottom right corner: $(30, -30, z_w)$.
     - After using the **align environment algorithm** in section 5.2, the adjusted locations of the 3D objects and the light source are
     - Adjusted Polygon vertex points: $P_1 = (0, 0, -135)$, $P_2 = (10, 20, -135)$, $P_3 = (35, 0, -135)$; Adjusted Sphere Center = $(0, 0, -105)$; Adjusted Light Source $(0, 0, -75)$; the z_w location of the view window is now -25, *which was the value of the z location of the CoP.* Therefore, adjusted view-window – top left corner: $(-30, 30, -25)$, Bottom right corner: $(30, -30, -25)$.

- Now used the **scale environment algorithm** to scale the polygon, sphere, light source, and view window by $sf = 10$
- **Output**: Scaled Light source – $(0, 0, -750)$; Scaled view-window – top left corner: $(-300, 300, -250)$, Bottom right corner: $(300, -300, -250)$; Scaled Polygon vertex points: $P_1 = (0, 0, -1350)$, $P_2 = (100, 200, -1350)$, $P_3 = (350, 0, -1350)$; Scaled Sphere Center located at $(0, 0, -1050)$ and Radius = 200;

9. The **ambient light algorithm** in section 5.8.1 determines the intensity of ambient light at surface point.
   - Input/Output Set
     - **Input**: $I_a = 1$, $k_a = 0.2$
     - **Output**: $\rho_a = 0.2$
10. The **diffuse reflection algorithm** in section 5.8.2 determine the light intensity due to diffuse reflection at a surface point.
    - Input/Output Set 1
      - **Input**: $I_{ls} = 1$, $k_d = 0.6$, Light ray vector = $[25, 10, 30]$, Normal vector = $[40, 28, 75]$, Ray vector = $[0, 0, -50]$, the surface point is from a polygon
      - **Output**: $\rho_d = 0.5871$
    - Input/Output Set 2
      - **Input**: $I_{ls} = 1$, $k_d = 0.6$, Light ray vector = $[25, 10, 30]$, Normal vector = $[40, 28, 75]$, Ray vector = $[0, 0, 50]$, the surface point is from a polygon
      - **Output**: $\rho_d = 0$
11. The **specular reflection algorithm** in section 5.8.3 determine amount of specular reflection to added to a surface point.
    - Input/Output Set 1
      - **Input**: $I_{ls} = 1$, $k_s = 0.25$, $n = 10$, Light ray vector = $[0, 50, 35]$, Normal vector = $[0, 0, 20]$, Ray vector = $[0, 0, -50]$, the surface point is from a polygon
      - **Output**: $\rho_s = 0.2452$
    - Input/Output Set 2
      - **Input**: $I_{ls} = 1$, $k_s = 0.25$, $n = 10$, Light ray vector = $[0, 50, 35]$, Normal vector = $[0, 0, 20]$, Ray vector = $[0, 0, 50]$, the surface point is from a polygon
      - **Output**: $\rho_s = 0$

12. The **shadow ray-tracing algorithm** in section 5.9.1 determines if a light ray is blocked.
    - Input/Output Set
      - **Input**: 3D Environment: 1 Polygon – Polygon vertex points: $P_1 = (50, 50, 40)$, $P_2 = (60, 70, 40)$, $P_3 = (85, 50, 40)$; 1 Sphere – Center point = $(50, 50, 70)$, Radius = 20; For a light source located at $(50, 50, 200)$ and an intersection point on the polygon at $(50, 50, 40)$
      - **Output**: The light ray is blocked by the Sphere; therefore, the intersection point $(50, 50, 40)$ is in shadow.
13. The **actual color algorithm** in section 5.10.1 determines the color of the surface, given a light source model and if the object is in shadow.
    - Input/Output Set 1
      - **Input**: 3D Environment: 1 Polygon – Polygon vertex points: $P_1 = (50, 50, 40)$, $P_2 = (60, 70, 40)$, $P_3 = (85, 50, 40)$; color of polygon is [255, 100, 50]; $k_a = .2$, $k_d = .6$, $k_s = .25$, $n = 10$, $I_a = 1$, $I_{ls} = 1$, and $I_{shadow} = .5$
      - Light source at $(50, 100, 125)$; center of projection at $(50, 50, 200)$; and pixel on view window at $(50, 50, 150)$,
      - The intersection point on the polygon is $(50, 50, 40)$, and the intersection point is NOT in a shadow
      - **Output**: The *actual* color is $[197, 86, 50]$
    - Input/Output Set 2
      - **Input**: 3D Environment: 1 Polygon – Polygon vertex points: $P_1 = (50, 50, 40)$, $P_2 = (60, 70, 40)$, $P_3 = (85, 50, 40)$; color of polygon is [255, 100, 50]; $k_a = .2$, $k_d = .6$, $k_s = .25$, $n = 10$, $I_a = 1$, $I_{ls} = 1$, and $I_{shadow} = .5$
      - Light source at $(50, 100, 125)$; center of projection at $(50, 50, 200)$; and pixel on view window at $(50, 50, 150)$,
      - The intersection point on the polygon is $(50, 50, 40)$, and the intersection point Is in a shadow
      - **Output**: The *actual* color is $[117, 46, 23]$
    - Input/Output Set 3
      - **Input**: 3D Environment: 1 Sphere – Center point = $(50, 50, 70)$, Radius = 20; color of sphere is [255, 100, 50]; $k_a = .2$, $k_d = .6$, $k_s = .25$, $n = 10$, $I_a = 1$, $I_{ls} = 1$, and $I_{shadow} = .5$
      - Light source at $(50, 100, 125)$; center of projection at $(50, 50, 200)$;

and pixel on view window at $(50, 50, 150)$,
  - The intersection point on the sphere is $(50, 50, 90)$, and the intersection point is NOT in a shadow
  - **Output**: The *actual* color is $[139, 55, 27]$
- Input/Output Set 4
  - **Input**: 3D Environment: 1 Sphere – Center point = $(50, 50, 70)$, Radius = 20; color of sphere is [255, 100, 50]; $k_a = .2$, $k_d = .6$, $k_s = .25$, $n = 10$, $I_a = 1$, $I_{ls} = 1$, and $I_{shadow} = .5$
  - Light source at $(50, 100, 125)$; center of projection at $(50, 50, 200)$; and pixel on view window at $(50, 50, 150)$,
  - The intersection point on the sphere is $(50, 50, 90)$, and the intersection point Is in a shadow
  - **Output**: The *actual* color is $[95, 37, 19]$

14. The **lighting algorithm** in section 5.10.2 lights and displays a 3D environment based on all the algorithms presented in this chapter.

5.12 Review Questions

1. A ray from a center of project at (165, 150, 600) to a point on the view window at (155, 160, 300) intersects a sphere and polygon in a 3D environment.

   (a) What is the intersection point of the ray with a polygon with vertex points p_1 = (100, 50, 40), p_2 = (160, 270, 40), and p_3 = (285, 50, 40)?
   (b) What is the intersection point of the ray with a sphere with a center point located at (150,150,70) and radius of 20?
   (c) Which of the two intersection points is closer?

2. A 3D environment contains
   - a sphere with center point of (120,25,80) and radius of 40
   - a light source located at point (45,10,90)

   The viewing parameters of the environment are
   - top left corner of the view window is located at (-30, 30, z_w) and the bottom right corner is located at (30,-30, z_w)
   - vrp = [70, 30, 200]
   - cop = [0, 0, 20]

- $\alpha = 0$ and $\beta = 0$

(a) What is the new locations of the sphere and light source after aligning the environment using the viewing parameters?
(b) What is the new location of the cop (center of projection)?
(c) What is the new location of the vrp (view reference point)?
(d) Next after alignment, the environment is scaled by 15. What are the values of the sphere, light source, and view window after scaling the environment?

3. Given a point on a polygon with vertex points p_1 = (100, 50, 45), p_2 = (160, 270, 45), and p_3 = (285, 50, 45).

- the Ray Vector to that point is (-30,-10,-300)
- the Light Vector is (-94.5,-81.5,55)
- the point is not in a shadow
- the ambient light intensity $I_a = 0.5$
- the light source intensity $I_l s = 1$
- ka = 0.2, kd = 0.6, ks = 0.25, and n = 10
- the color of the polygon is R = 125, G = 28, and B = 95

(a) What is the normal vector at this point on the polygon?
(b) What is the ambient light value at this point?
(c) What is the diffuse reflection value at this point?
(d) What is the specular reflection value at this point?
(e) What is the actual color of this point?

PROGRAMMING EXERCISES

The 3D environment described in Table 5.1 is used for the programming exercises below. The environment is composed of 6 polygons and 2 spheres.

	Polygon #1
Point 1	(0, 0, 0)
Point 2	(100, 0, 0)
Point 3	(100, 0, 100)
Point 4	(0, 0, 100)
k_a	0.2
k_d	0.6
k_s	0.25
n	5
Color	[255, 255, 255]

	Polygon #2
Point 1	(0, 0, 0)
Point 2	(0, 100, 0)
Point 3	(0, 100, 100)
Point 4	(0, 0, 100)
k_a	0.2
k_d	0.6
k_s	0.25
n	5
Color	[255, 255, 255]

	Polygon #3
Point 1	(0, 0, 0)
Point 2	(0, 100, 0)
Point 3	(100, 100, 0)
Point 4	(100, 0, 0)
k_a	0.2
k_d	0.6
k_s	0.25
n	5
Color	[255, 255, 255]

	Polygon #4
Point 1	(10, 30, 85)
Point 2	(70, 30, 85)
Point 3	(70, 30, 5)
Point 4	(10, 30, 5)
k_a	0.2
k_d	0.6
k_s	0.25
n	5
Color	[255, 255, 0]

	Polygon #5
Point 1	(60, 0, 80)
Point 2	(60, 30, 80)
Point 3	(60, 30, 10)
Point 4	(60, 0, 10)
k_a	0.2
k_d	0.6
k_s	0.25
n	5
Color	[255, 255, 0]

	Polygon #6
Point 1	(20, 0, 80)
Point 2	(20, 30, 80)
Point 3	(20, 30, 10)
Point 4	(20, 0, 10)
k_a	0.2
k_d	0.6
k_s	0.25
n	5
Color	[255, 255, 0]

	Sphere #1
Center	(50, 80, 50)
Radius	20
k_a	0.2
k_d	0.6
k_s	0.25
n	10
Color	[255, 0, 0]

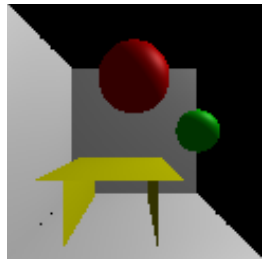
	Sphere #2
Center	(80, 50, 70)
Radius	10
k_a	0.2
k_d	0.6
k_s	0.25
n	10
Color	[0, 255, 0]

Table 5.1. Description of objects in a 3D environment. The environment is composed of 6 polygons and 2 spheres.

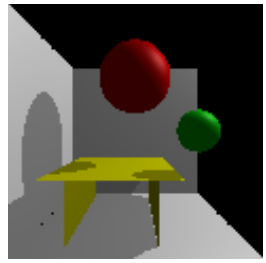
1. Parameters: light source located at $(200, 200, 50)$, VRP $= (50, 50, 120)$, CoP $= (0, 0, 50)$, $\alpha = 0$, $\beta = 0$, view-window – top left corner: $(-30, 30, z_w)$, bottom right corner: $(30, -30, z_w)$, and a scale factor of $sf = 2$. From

section 5.7, use the **partial lighting algorithm** with the parameters stated above to create a PPM image of the 3D environment described in Table 5.1. (*see example below*)

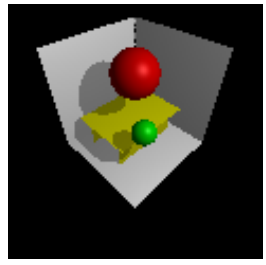
2. Parameters: light source located at $(200, 200, 50)$, VRP $= (50, 50, 120)$, CoP $= (0, 0, 50)$, $\alpha = 0$, $\beta = 0$, view-window – top left corner: $(-30, 30, z_w)$, bottom right corner: $(30, -30, z_w)$, and a scale factor of $sf = 2$. Using the **lighting algorithm** in section 5.10.2 and the parameters stated above, create a PPM image of the 3D environment described in Table 5.1. Don't implement the **shadow ray-tracing algorithm**. (*see example below*)



3. Parameters: light source located at $(200, 200, 50)$, VRP $= (50, 50, 120)$, CoP $= (0, 0, 50)$, $\alpha = 0$, $\beta = 0$, view-window – top left corner: $(-30, 30, z_w)$, bottom right corner: $(30, -30, z_w)$, the scale factor is $sf = 2$, and $I_{shadow} = .5$. Using the **lighting algorithm** in section 5.10 and the parameters stated above, create a PPM image of the 3D environment described in Table 5.1. (*see example below*)



4. Parameters: light source located at $(200, 200, 50)$, VRP $= (130, 130, 130)$, CoP $= (0, 0, 50)$, $\alpha = 45$, $\beta = -45$, view-window – top left corner: $(-30, 30, z_w)$, bottom right corner: $(30, -30, z_w)$, scale factor is $sf = 2$, $I_{shadow} = .5$. Using the **lighting algorithm** in section 5.10 and the parameters stated above, create a PPM image of the 3D environment described in Table 5.1. (*see example below*)



VI 3D OBJECT TRANSFORMATIONS

IN this chapter 3D object transformations are discussed. Recall from Chapter 5, 3D object are composed of spheres and polygons. Therefore, our discussion centers around *translating*, *scaling*, and *rotating* these two primitives. Similar to the 2D transformations in Chapter 3, the translation transformation moves an object to a different location in the XYZ coordinate system; the scale transformation changes the size of the object, and the rotation transformation rotates the object about a particular axis. Figure 6.1 shows examples of each type of transformation for an image made of 1 sphere and 4 polygons.

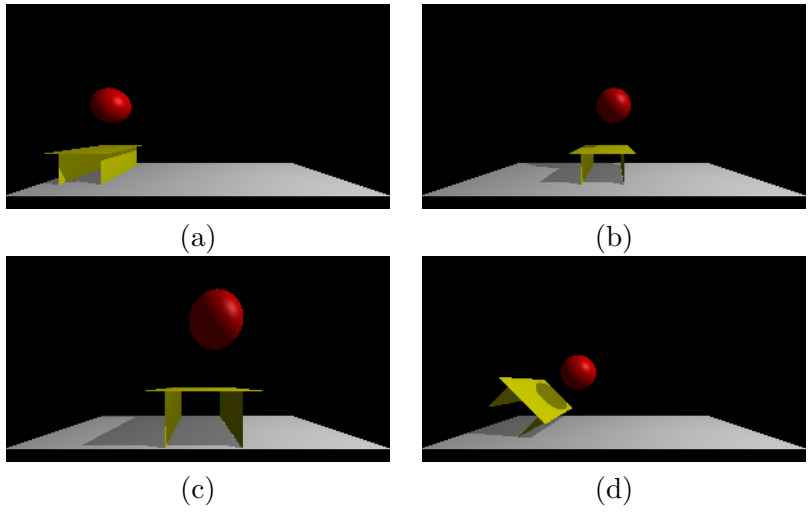


Figure 6.1. Transformation examples: (a) Original Image (b) Translated Image (c) Scaled Image (d) Rotated Image

6.1 Translation

Translating a sphere or polygon from one location to another location involves adding scalar values (x_t, y_t, z_t) to the x, y, and z values of the vertex points of the polygon and the center point of a sphere. For example, to translate a sphere centered at (x, y, z) by (x_t, y_t, z_t), the resulting new location of the sphere's center

is (x', y', z'), where

$$x' = x + x_t \tag{6.1}$$
$$y' = y + y_t \tag{6.2}$$
$$z' = z + z_t. \tag{6.3}$$

Because this is a translation operation, the radius of the sphere doesn't change.

In Figure 6.1(a), the red sphere's center is located at $(50, 80, 50)$. To translate the sphere to its new location in Figure 6.1(b), the center point is translated by $(100, 0, 0)$. The new sphere's location in Figure 6.1(b), using Equations 6.1, 6.2, and 6.3, is

$$\begin{aligned} 150 &= 50 + 100 \\ 80 &= 80 + 0 \\ 50 &= 50 + 0, \end{aligned}$$

For a new center point at $(150, 80, 50)$. This particular translation simply moves the sphere horizontally in the x direction. If you wanted to move the sphere vertically, the y_t value of the translation would be non-zero. For example, to move the sphere down by 100 points, the translation values are $x_t = 0$, $y_t = -100$, and $z_t = 0$. For this translation, the sphere's center point's location, using Equations 6.1, 6.2, and 6.3, is

$$\begin{aligned} 15 &= 50 + 0 \\ -20 &= 80 - 100 \\ 50 &= 50 + 0, \end{aligned}$$

for a center point of $(15, -20, 50)$.

A polygon also uses Equations 6.1, 6.2, and 6.3, but these equations are applied to all the vertex points of the polygon. The yellow polygons in Figure 6.1(a), which form a table, had the same translation of $(100, 0, 0)$ applied to them. Therefore, in Figure 6.1(b), both the yellow polygons and the red sphere are translated horizontally.

A simple **3D-translation algorithm** to translate a sphere or polygon by x_t, y_t, and z_t is outlined in the following steps:

1. If the object is a *Sphere*
   (a) Add x_t to the x-value of the sphere's center point as in Equation 6.1
   (b) Add y_t to the y-value of the sphere's center point as in Equation 6.2
   (c) Add z_t to the z-value of the sphere's center point as in Equation 6.3

2. If the object is a *Polygon*
   (a) Add x_t to the x-values of the polygon's vertex points as in Equation 6.1
   (b) Add y_t to the y-values of the polygon's vertex points as in Equation 6.2
   (c) Add z_t to the z-values of the polygon's vertex points as in Equation 6.3

6.2 Scale

A scale transformation changes the size of a sphere or polygon as shown in Figure 6.1(c) – the yellow polygons and red sphere were scaled by 1.5. To scale a polygon by a scale factor of sf, the vertex points of the polygon, $(x_1, y_1, z_2)...(x_n, y_n, z_n)$, are scaled

$$x' = x \cdot sf \tag{6.4}$$
$$y' = y \cdot sf \tag{6.5}$$
$$z' = z \cdot sf. \tag{6.6}$$

A sphere is scaled by scaling the radius,

$$radius' = radius \cdot sf. \tag{6.7}$$

However, from section 3.3, we learned that scaling a point actually moves the object being scaled. Therefore, an object needs to be scaled by a fixed point (x_f, y_f, z_f). The process is the same as in section 3.3, except instead of 2D points as in section 3.3, here we have 3D points.

For example, for a polygon with 4 vertex points – $(10, 30, 85)$, $(70, 30, 85)$, $(70, 30, 5)$, and $(10, 30, 5)$ – let's scale it by 2 about point $(10, 30, 85)$. The first step is to translate the vertex points of the polygon by $(-10, -30, -85)$, for a result of $(0, 0, 0)$, $(60, 0, 0)$, $(60, 0, -80)$, and $(0, 0, -80)$. The second step is to scale the translated vertex points by 2 as in Equations 6.4, 6.5, and 6.6, for a result of $(0, 0, 0)$, $(120, 0, 0)$, $(120, 0, -160)$, and $(0, 0, -160)$. The final step is to translate the scaled vertex points by $(10, 30, 85)$, for the final result of $(10, 30, 85)$, $(130, 30, 85)$, $(130, 30, -75)$, and $(10, 30, -75)$.

If a sphere is scaled with a polygon as in Figure 6.1(c), the sphere also needs to be scaled by a fixed point. This is accomplished by applying in the same steps as in the previous paragraph to the *center point* and *radius*. Of course, since the radius is not a point, the translation steps are not applied to it.

For example, for a sphere with center point of $(50, 80, 50)$ and radius = 20, let's scale it by 2 about point $(10, 30, 85)$ as we did with the polygon. First, translate the center point by $(-10, -30, -85)$, for a new center point of $(40, 50, -35)$. Second, scale the center point by 2, for a result of $(80, 100, -70)$. Also, scale the radius by 2 as in Equation 6.7, for a new radius of 40. Finally, translate the scaled center point by $(10, 30, 85)$. The result of this process is a sphere centered at $(90, 130, 15)$ with a radius of 40.

A simple **3D-scale algorithm** to scale a sphere or polygon by sf for a fixed point of (x_f, y_f, z_f) is outlined in the following steps:

1. If the object is a *Sphere*
   (a) Translate the center point by $(-x_f, -y_f, -z_f)$ using the **3D-translation algorithm** in section 6.1
   (b) Scale the translated x-value of the center point by sf using Equation 6.4
   (c) Scale the translated y-value of the center point by sf using Equation 6.5
   (d) Scale the translated z-value of the center point by sf using Equation 6.6
   (e) Scale the radius by sf using Equation 6.7
   (f) Translate the scaled center point by (x_f, y_f, z_f) using the **3D-translation algorithm** in section 6.1
2. If the object is a *Polygon*
   (a) Translate the polygon's vertex points by $(-x_f, -y_f, -z_f)$ using the **3D-translation algorithm** in section 6.1
   (b) Scale the translated x-values of the polygon's vertex points by sf using Equation 6.4
   (c) Scale the translated y-values of the polygon's vertex points by sf using Equation 6.5
   (d) Scale the translated z-values of the polygon's vertex points by sf using Equation 6.6
   (e) Translate the polygon's scaled vertex points by (x_f, y_f, z_f) using the **3D-translation algorithm** in section 6.1

6.3 Rotation

The rotation transformation is the most involved transformation of the three transformation discussed in this chapter. To rotate a 3D object, first a rotation axis must be defined for the object to rotate around. Figure 6.2(a) shows an

example of defining a rotation axis for 3 polygons (which form a yellow table) and 1 sphere. For this example, the rotation axis is defined by two points, $p_1 = (x_1, y_1, z_1)$ and $p_2 = (x_2, y_2, z_2)$. These two points come from the bottom vertex points of the polygon that forms the right leg of the table. Second, after the rotation axis is defined, the 3D objects are rotated about that axis. Figure 6.2(b) shows the result of rotating the polygons and sphere about the rotation axis by 45 degrees. Notice how the polygons and sphere all rotate together.

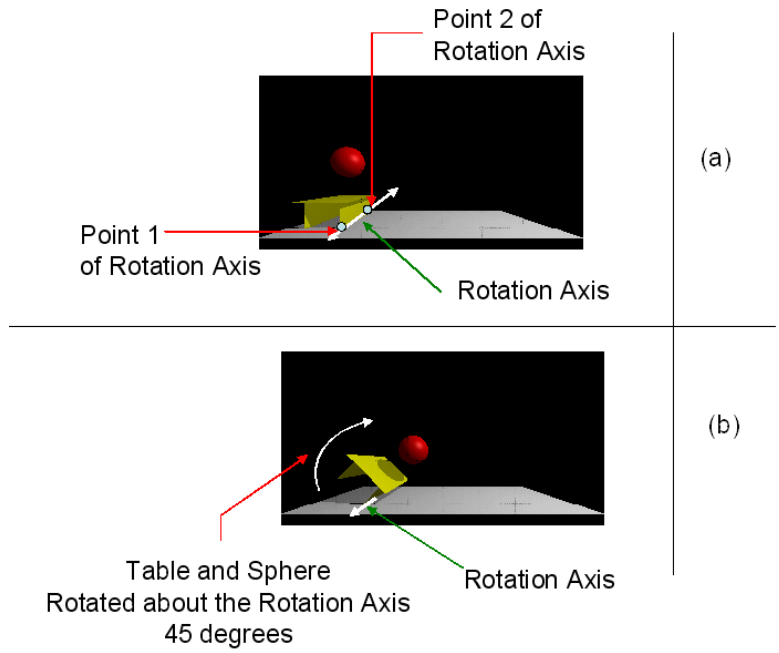


Figure 6.2. (a) Table and Sphere before rotation, outlining the rotation axis (b) Table and Sphere after rotation

In the following subsections, first the process to rotate 3D objects about one of the majors 3 axes (x-axis, y-axis, and z-axis) is discussed. Second, the process to rotate 3D objects about an arbitrary axis (which is not one of the major 3 axes) is explain. Rotating about an arbitrary axis is the example given in Figure 6.2, which was briefly discussed above.

6.3.1 Major Axes Rotations

The major axes of an XYZ-coordinate system are the x-axis, y-axis, and z-axis, as we have seen before. To rotate a 3D object about one of these axes, the vertex points of a polygon or the center point of a sphere must be rotated about one of the axes. Figure 6.3 shows the direction of positive rotations for the x, y, and z

axes.

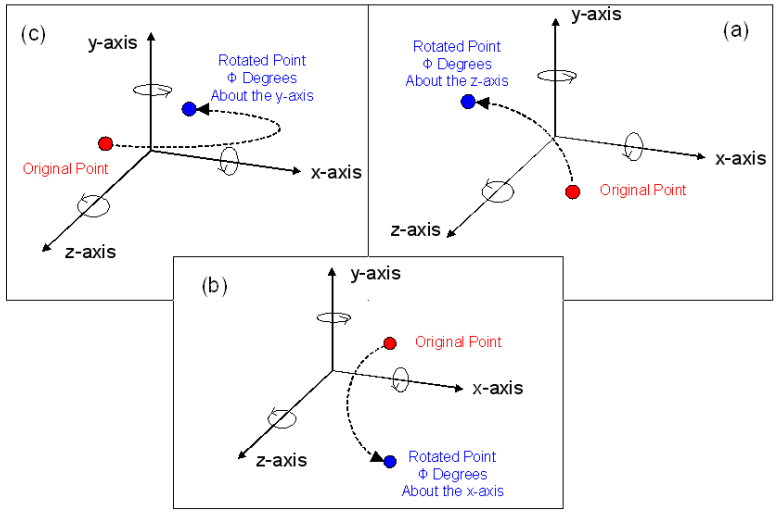


Figure 6.3. Rotation of a 3D point (a) about the z-axis (b) about the x-axis (c) about the y-axis

6.3.1.1 z-axis Rotation

Rotating a 3D point about the z-axis as shown in Figure 6.3(a) is the same as rotating a 2D point in the XY-coordinate system described in section 3.2. Therefore, the equations to rotate the x and y values of a 3D point are Equations 3.7 and 3.8, which are the same equations to rotate the x and y values of a 2D point in the XY-coordinate system. Since the 3D point is rotated about the z-axis, the z value of the 3D point doesn't change. Therefore, using the same Equations 3.7 and 3.8 and the fact that the z value doesn't change, the equations to rotate a 3D point (x, y, z) about the z-axis Φ degrees are

$$x' = x \cdot cos(\Phi) - y \cdot sin(\Phi) \quad (6.8)$$

$$y' = y \cdot cos(\Phi) + x \cdot sin(\Phi) \quad (6.9)$$

$$z' = z. \quad (6.10)$$

A simple **z-axis rotation-algorithm** to rotate a 3D point (x, y, z) about the z-axis Φ degrees is outlined in the following steps:

1. Rotate x by Φ using Equation 6.8
2. Rotate y by Φ using Equation 6.9
3. The z value remains the same

6.3.1.2 *x-axis Rotation*

In section 4.4, the UVN-coordinate system was introduced. However, the UVN-coordinate system has similar axes to that of the XYZ-coordinate system. The u-axis is equivalent to the x-axis; the v-axis is equivalent to the y-axis; and the n-axis is equivalent to the z-axis. From section 4.4, the Equations 4.19, 4.20, and 4.21 were developed to rotate about the u-axis.

Because the u and x axes are equivalent, these same equations may be used to rotate a 3D point about the x-axis Φ degrees as shown in Figure 6.3(b). Using Equations 4.19, 4.20, and 4.21, for the XYZ-coordinate, the equations to rotate a 3D point (x, y, z) about the x-axis Φ degrees are

$$x' = x \tag{6.11}$$
$$y' = y \cdot cos(\Phi) - z \cdot sin(\Phi) \tag{6.12}$$
$$z' = z \cdot cos(\Phi) + y \cdot sin(\Phi). \tag{6.13}$$

Again, here the x value doesn't change because the rotation is about the x-axis. A simple **x-axis rotation-algorithm** to rotate a 3D point (x, y, z) about the x-axis Φ degrees is outlined in the following steps:

1. The x value remains the same
2. Rotate y by Φ using Equation 6.12
3. Rotate z by Φ using Equation 6.13

6.3.1.3 *y-axis Rotation*

Last, to rotate a 3D point about the y-axis as shown in Figure 6.3(c), the Equations 4.13, 4.14, and 4.15, developed to rotate about the v-axis in the UVN-coordinate system, are used because the y-axis is equivalent to the v-axis. Therefore, the equations to rotate a 3D point (x, y, z) about the y-axis Φ degrees are

$$x' = x \cdot cos(\Phi) + z \cdot sin(\Phi) \tag{6.14}$$
$$y' = y \tag{6.15}$$
$$z' = z \cdot cos(\Phi) - x \cdot sin(\Phi), \tag{6.16}$$

Here again the y value doesn't change because the rotation is about the y-axis. A simple **y-axis rotation-algorithm** to rotate a 3D point (x, y, z) about the y-axis Φ degrees is outlined in the following steps:

1. Rotate x by Φ using Equation 6.14
2. The y value remains the same
3. Rotate z by Φ using Equation 6.16

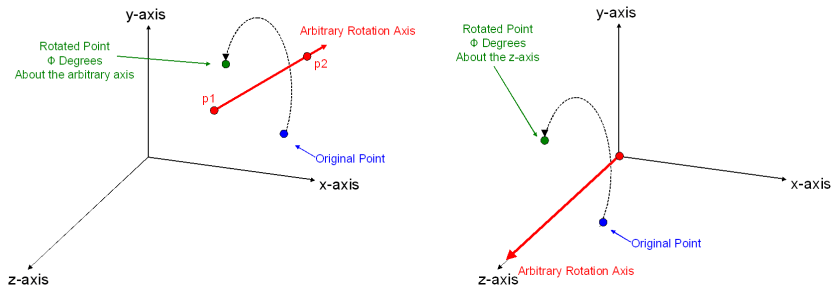


Figure 6.4. (a) Rotation of a 3D point about an arbitrary axis (b) Alignment of the arbitrary axis with the z-axis to rotate the 3D point about the z-axis

6.3.2 Arbitrary Axis Rotation

Unlike rotating a 3D point about one of the major axes, an arbitrary axis first must be defined by two points: $p_1 = (x_1, y_1, z_1)$ and $p_2 = (x_2, y_2, z_2)$ – Figure 6.4(a). Once an axis is defined, the goal is the same - *rotate a 3D point about the axis*. The method to accomplish this is to first align the arbitrary axis with the z-axis – Figure 6.4(b) – and then preform the rotation about the z-axis. After rotating about the z-axis, all the steps needed to align the arbitrary axis with z-axis are applied in reverse to get the proper location of the rotated 3D point. The focus of this section is to develop these alignment steps. [1]

Figure 6.4(a) is used as our example to develop the step necessary to rotate a 3D point Φ degrees about an arbitrary axis defined by points p_1 and p_2. The first step, Figure 6.5(a), is to compute the vector $\vec{U}$ from p_1 to p_2, where $\vec{U} = (x_u, y_u, z_u)$ and

$$x_u = x_2 - x_1 \tag{6.17}$$

$$y_u = y_2 - y_1 \tag{6.18}$$

$$z_u = z_2 - z_1. \tag{6.19}$$

The second step is to translate point p_1 to the origin, so the arbitrary axis is connected to the origin – Figure 6.5(b). Translating point p_1 to the origin implies that the 3D point to be rotated is translated by $(-x_1, -y_1, -z_1)$.

Our third step is rotate the arbitrary axis to the yz plane – Figure 6.5(c). This is accomplished by finding the angle between the projection of the arbitrary axis on the xz plane and the z-axis. As shown in Figure 6.5(b) that angle is β.

[1] These alignment steps apply only to non-vertical axes, and will create a division by zero error for vertical axes. Other alignment steps are needed to rotate about a vertical axes; however, they are not discussed here.

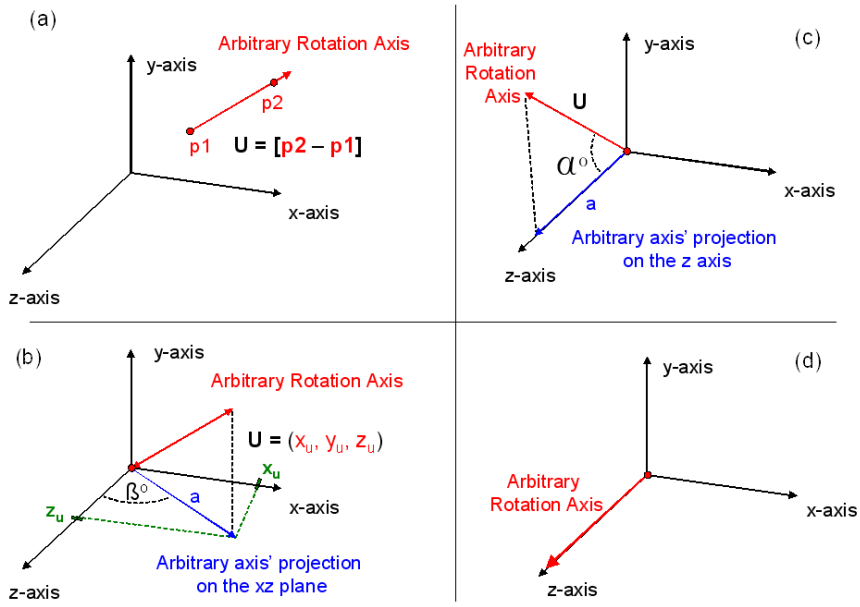


Figure 6.5. Rotation of a 3D point about an arbitrary axis: (a) Compute vector of axis (b) Translate vector to origin, (c) Rotate vector to the yz plane about the x-axis, and (d) Rotate vector to the xz plane about the y-axis.

To compute the angle β, first the magnitude[2] of the projection is computed as such

$$\mathbf{a} = \sqrt{x_u^2 + z_u^2}. \tag{6.20}$$

Next, using trigonometry and right triangles, the magnitude and the z length of the projection, the angle is

$$\beta = \cos^{-1}(\frac{z_u}{\mathbf{a}}). \tag{6.21}$$

Now, the arbitrary axis is rotated about the y-axis by $-\beta$ degrees, which is accomplished by rotating the 3D point about the y-axis by $-\beta$ degrees.

Note that we rotate by negative β because we are rotating about the y-axis from the x-axis to the z-axis. The direction of positive rotation about the y-axis, as shown in Figure 6.3(c), is from the z-axis to the x-axis. Since the direction of rotation is opposite the positive direction, the rotation must be negative.

After rotating the arbitrary axis about the y-axis, the axis now lies in the yz plane – Figure 6.5(c). The fourth step is to rotate the arbitrary axis so that it lies in the xz plane. This rotation will align the arbitrary axis with the z-axis –

[2]The length of the projection.

Figure 6.5(d). By rotating the arbitrary axis about the x-axis by α degrees as shown in Figure 6.5(c), the arbitrary axis is aligned with the z-axis. To compute the angle α, first the magnitude of $\vec{U}$ is calculated as such

$$|\vec{U}| = \sqrt{x_u^2 + y_u^2 + z_u^2}. \tag{6.22}$$

Again using trigonometry and right triangles,

$$\alpha = \cos^{-1}(\frac{\mathbf{a}}{|\vec{U}|}). \tag{6.23}$$

The arbitrary axis is now rotated about the x-axis α degrees. As before, to rotate the arbitrary axis, the 3D point to be rotated is rotated about the x-axis by α degrees.

After these series of rotations, the arbitrary axis is aligned with the z-axis – Figure 6.5(d). Now the fifth step is to rotate the 3D point about the z-axis Φ degrees as shown in Figure 6.4(b).

To actually put the rotated 3D point in the correct location in the XYZ-coordinate system, the steps are reversed. So, the sixth step is to rotate the 3D point about the x-axis by $-\alpha$ degrees. Seventh, rotate the 3D point about the y-axis by β degrees. The final step is to translate the 3D point by (x_1, y_1, z_1).

A simple **3D-rotation algorithm** to rotate a sphere or polygon about an arbitrary axis (defined by $p_1 = (x_1, y_1, z_1)$ and $p_2 = (x_2, y_2, z_2)$) by Φ degrees is outlined in the following steps:

1. Compute the vector, $\vec{U}$, of the arbitrary axis from p_1 to p_2 as in Equations 6.17, 6.18, and 6.19
2. Compute the magnitude of $\vec{U}$ using Equation 6.22
3. Compute the magnitude, $\mathbf{a}$, of the projection of the axis on the xz plane using Equation 6.20
4. Compute the angle β using Equation 6.21
5. Compute the angle α using Equation 6.23
6. If the object is a *Sphere*
   (a) Translate the center point of the sphere by $(-x_1, -y_1, -z_1)$ using the **3D-translation algorithm** in section 6.1
   (b) Rotate the translated center point about the y-axis by $-\beta$ degrees using the **y-axis rotation-algorithm** from section 6.3.1.3
   (c) Rotate the rotated center point about the x-axis by α degrees using the **x-axis rotation-algorithm** from section 6.3.1.2
   (d) Rotate the rotated center point about the z-axis by Φ degrees using the **z-axis rotation-algorithm** from section 6.3.1.1

(e) Rotate the rotated center point about the x-axis by $-\alpha$ degrees using the **x-axis rotation-algorithm** from section 6.3.1.2
(f) Rotate the rotated center point about the y-axis by β degrees using the **y-axis rotation-algorithm** from section 6.3.1.3
(g) Translate the rotated center point of the sphere by (x_1, y_1, z_1) using the **3D-translation algorithm** in section 6.1

7. If the object is a *Polygon*
   (a) Translate the vertex points of the polygon by $(-x_1, -y_1, -z_1)$ using the **3D-translation algorithm** in section 6.1
   (b) Rotate the translated vertex-points about the y-axis by $-\beta$ degrees using the **y-axis rotation-algorithm** from section 6.3.1.3
   (c) Rotate the rotated vertex-points about the x-axis by α degrees using the **x-axis rotation-algorithm** from section 6.3.1.2
   (d) Rotate the rotated vertex-points about the z-axis by Φ degrees using the **z-axis rotation-algorithm** from section 6.3.1.1
   (e) Rotate the rotated vertex-points about the x-axis by $-\alpha$ degrees using the **x-axis rotation-algorithm** from section 6.3.1.2
   (f) Rotate the rotated vertex-points about the y-axis by β degrees using the **y-axis rotation-algorithm** from section 6.3.1.3
   (g) Translate the rotated vertex-points of the polygon by (x_1, y_1, z_1) using the **3D-translation algorithm** in section 6.1

6.4 Algorithm Summary and Testing

In this chapter, three general 3D transformations – translation, rotation, and scale – were discussed. These transformations are applied to 3D objects – *spheres* and *polygons*. The following is a list of algorithms developed in this chapter. In addition, there are input and output examples given for the algorithms to test your coding of the algorithms.

1. The **3D-translation algorithm** in section 6.1 translates a sphere or polygon from one location to another in an 3D environment.
   - Input/Output Set
     - **Input**: 1-sphere with center point of $(50, 80, 50)$ and radius of 20; 1-polygon with 4 vertex points $(10, 30, 85)$, $(70, 30, 85)$, $(70, 30, 5)$, & $(10, 30, 5)$; Translate the sphere and polygon by $(100, -20, -15)$
     - **Output**: Translated sphere: center = $(150, 60, 35)$, radius = 20; Translated polygon: vertex points are $(110, 10, 70)$, $(170, 10, 70)$, $(170, 10, -10)$, & $(110, 10, -10)$

2. The **3D-scale algorithm** in section 6.2 scales a polygon or sphere about a fixed point.
   - Input/Output Set 1
     - **Input**: 1-sphere with center point of $(50, 80, 50)$ and radius of 20; 1-polygon with 4 vertex points $(10, 30, 85)$, $(70, 30, 85)$, $(70, 30, 5)$, & $(10, 30, 5)$; Scale the sphere and polygon by 1.75, for a fix point of $(10, 30, 85)$
     - **Output**: Scaled sphere: center = $(80, 117.5, 23.75)$, radius = 35; Scaled polygon: vertex points are $(10, 30, 85)$, $(115, 30, 85)$, $(115, 30, -55)$, & $(10, 30, -55)$
   - Input/Output Set 2
     - **Input**: 1-sphere with center point of $(25, 45, 100)$ and radius of 20; 1-polygon with 4 vertex points $(60, 0, 80)$, $(60, 30, 80)$, $(60, 30, 10)$, & $(60, 0, 10)$; Scale the sphere and polygon by 0.5, for a fix point of $(20, 30, 40)$
     - **Output**: Scaled sphere: center = $(22.5, 37.5, 70)$, radius = 10; Scaled polygon: vertex points are $(40, 15, 60)$, $(40, 30, 60)$, $(40, 30, 25)$, & $(40, 15, 25)$
3. The **z-axis rotation-algorithm** in section 6.3.1.1 rotates a 3D point about the z-axis for a specified amount of degrees.
   - Input/Output Set 1
     - **Input**: $point = (25, 105, 30)$, rotate by 25 degrees
     - **Output**: $point = (-21.71, 105.73, 30)$
   - Input/Output Set 2
     - **Input**: $point = (25, 105, 30)$, rotate by -34 degrees
     - **Output**: $point = (79.44, 73.07, 30)$
4. The **x-axis rotation-algorithm** in section 6.3.1.2 rotates a 3D point about the x-axis for a specified amount of degrees.
   - Input/Output Set 1
     - **Input**: $point = (25, 105, 30)$, rotate by 25 degrees
     - **Output**: $point = (25, 82.48, 71.56)$
   - Input/Output Set 2
     - **Input**: $point = (25, 105, 30)$, rotate by -34 degrees
     - **Output**: $point = (25, 103.82, -33.84)$
5. The **y-axis rotation-algorithm** in section 6.3.1.3 rotates a 3D point about the y-axis for a specified amount of degrees.
   - Input/Output Set 1

  - **Input**: *point* = $(25, 105, 30)$, rotate by 25 degrees
  - **Output**: *point* = $(35.34, 105, 16.62)$
- Input/Output Set 2
  - **Input**: *point* = $(25, 105, 30)$, rotate by -34 degrees
  - **Output**: *point* = $(3.95, 105, 38.85)$

6. The **3D-rotation algorithm** in section 6.3.2 rotates a sphere or polygon about an arbitrary axis for a specified amount of degrees.

- Input/Output Set 1
  - **Input**: 1-sphere with center point of $(50, 80, 50)$ and radius of 20; 1-polygon with 4 vertex points $(10, 30, 85)$, $(70, 30, 85)$, $(70, 30, 5)$, & $(10, 30, 5)$; Rotate the sphere and polygon about an arbitrary axis with $p_1 = (10, 30, 85)$ and $p_2 = (70, 30, 85)$ by 45 degrees
  - **Output**: Rotated sphere: center = $(50, 90.10, 95.61)$, radius = 20; Rotated polygon: vertex points are $(10, 30, 85)$, $(70, 30, 85)$, $(70, 86.57, 28.43)$, & $(10, 86.57, 28.43)$
- Input/Output Set 2
  - **Input**: 1-sphere with center point of $(50, 80, 50)$ and radius of 20; 1-polygon with 4 vertex points $(10, 30, 85)$, $(70, 30, 85)$, $(70, 30, 5)$, & $(10, 30, 5)$; Rotate the sphere and polygon about an arbitrary axis with $p_1 = (70, 30, 5)$ and $p_2 = (10, 30, 5)$ by -90 degrees
  - **Output**: Rotated sphere: center = $(50, 75, -45)$, radius = 20; Rotated polygon: vertex points are $(10, 110, 5)$, $(70, 110, 5)$, $(70, 30, 5)$, & $(10, 30, 5)$

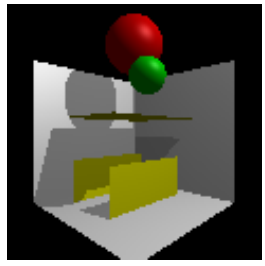
6.5 Review Questions

1. Rotate a sphere with a center point of (150,180,150) and radius 15 degrees about an arbitrary axis. The arbitrary axis is defined by points: p_1 = (25,35,75) and p_2 = (60,35,80). What is the location of the center point after the rotation?

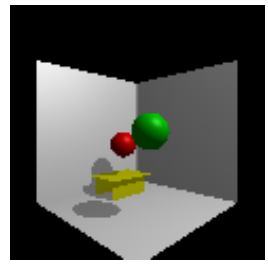
PROGRAMMING EXERCISES

The 3D environment described in Table 5.1 from Chapter 5 is used for the programming exercises below. The environment is composed of 6 polygons and 2 spheres.

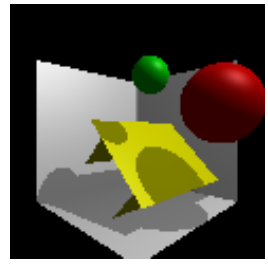
1. **Parameters**: light source located at $(200, 200, 50)$, VRP $= (120, 50, 120)$, CoP $= (0, 0, 50)$, $\alpha = 45$, $\beta = 0$, view-window – top left corner: $(-30, 30, z_w)$, bottom right corner: $(30, -30, z_w)$, and a scale factor of $sf = 2$.
   - Using the **3D-translation algorithm** from section 6.1, translate sphere #1, sphere #2, and polygon #4 by $(0, 30, 0)$.
   - Next, use the **lighting algorithm** from section 5.10 and the parameters stated above to create a PPM image of the 3D environment described in Table 5.1. *See example below.*



2. **Parameters**: light source located at $(200, 200, 50)$, VRP $= (120, 50, 120)$, CoP $= (0, 0, 50)$, $\alpha = 45$, $\beta = 0$, view-window – top left corner: $(-30, 30, z_w)$, bottom right corner: $(30, -30, z_w)$, and a scale factor of $sf = 2$.
   - Using the **3D-scale algorithm** from section 6.2, scale sphere #1, polygon #4, polygon #5, and polygon #6 by 0.5 for a fixed point of $(10, 0, 40)$.
   - Next, use the **lighting algorithm** from section 5.10 and the parameters stated above to create a PPM image of the 3D environment described in Table 5.1. *See example.*



3. **Parameters**: light source located at $(200, 200, 50)$, VRP $= (120, 50, 120)$, CoP $= (0, 0, 50)$, $\alpha = 45$, $\beta = 0$, view-window – top left corner: $(-30, 30, z_w)$, bottom right corner: $(30, -30, z_w)$, and a scale factor of $sf = 2$.
   - Using the **3D-translation algorithm** from section 6.1, translate sphere #2 by $(0, 30, 0)$.
   - Using the **3D-rotation algorithm** from section 6.3.2, rotate sphere #1, polygon #4, polygon #5, and polygon #6 by 45 degrees for the axis between points $(60, 0, 80)$ and $(60, 0, 10)$
   - Finally, use the **lighting algorithm** from section 5.10 and the parameters stated above to create a PPM image of the 3D environment described in Table 5.1. *See example below.*



VII COMPUTER ANIMATION

ANIMATION is defined as the act of infusing life and when dealing with graphical images, computer animation can be defined as the process to bring static images to life. The differing factor between computer graphics and computer animation is that computer graphics deals with the creation of one image, whereas computer animation deals with creating a sequence of images.

A sequence of images when displayed one after the other in quick succession conveys a feeling of motion. Each image in the sequence contributes to the feeling of motion by being slightly different than the pervious image. For example, Figure 7.1 shows a sequence of images depicting a stick figure holding a flag. In each image, the position of the flag is in a different location. When these images are displayed one after the other in quick succession, the flag appears to move and the feeling of the flag spinning is conveyed.

In this final chapter, a few simple techniques to create 2D and 3D animations based on the computer graphics algorithms in the proceeding chapters are presented. However, we will begin our discussing of animation with Frames.

7.1 Frames

Each image in a sequence of images is referred to as a frame in the animation. The image sequence in Figure 7.1 has 5 frames. When dealing with frames two important factors are import: the number of frames and the duration of each frame. When creating animations, the more frames you use to convey the desired motion, the smoother the motion looks to the human eye. If the number of frames is small, the motion can look choppy. For example, each frame in Figure 7.1 has the flag displaced by 45 degrees from the pervious frame. To make this motion look smoother, 10 more frames could be added and each of the (now) 15 frames could displace the flag by 15 degrees. The extra 10 frames spreads out the motion to included more transitional steps of the flag spinning.

The second factor that affects the animation is the amount of time each frame is displayed before the next frame is displayed. This duration is call the Frame Rate, and is measured in frames per second. If a frame is displayed for a long amount to time, the motion will move to slow, and if a frame is displayed for to short of a time, the motion will move to quickly.

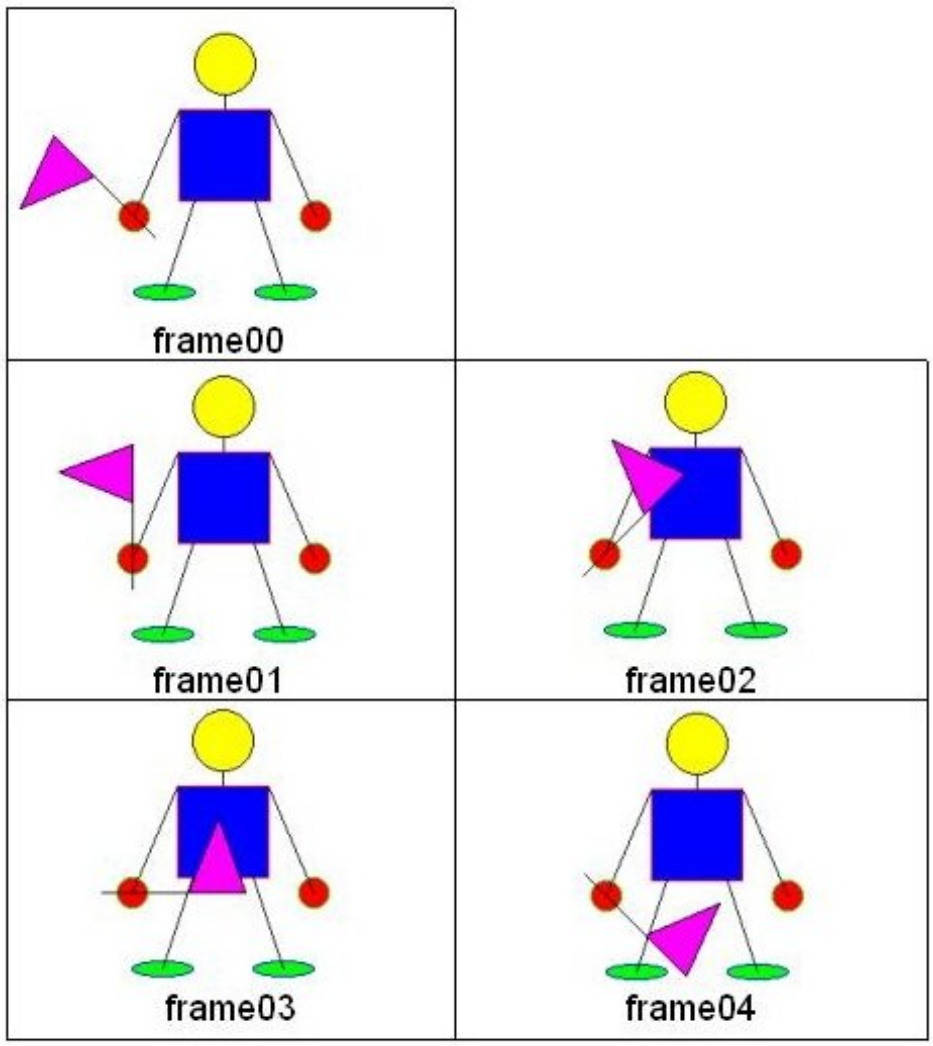


Figure 7.1. 2D Animation Example: A stick figure spinning a flag

Figure 7.2. Motion Blur example of an amusement park ride.

When filming live action with a video camera, a human can perceive natural motion at about 24 frames per second; so using our example in Figure 7.1 each frame should be display for 1/24 of a second before the next frame is displayed. However, computer generated motion is not quite the same as natural motion filmed with a video camera.

When a camera creates an image, that image does not represent a single instant of time. This is because video cameras capture live action based on the shutter speed of the video camera. The shutter speed is the amount of time required for a video camera to record one frame. During this time the object being recorded is moving and all position of the object will be integrated into one frame. This integration causes motion blur. Figure 7.2 is an example of motion blur. It is a live action image captured with a video camera using a long shutter speed, which causes the circular motion to blur the image.[1]

Therefore, one frame recorded with a video camera contains integrated motion, so natural motion filmed at 24 frames per second appears continuous. However, in a computer animation, each frame in the animation is a perfect instance in time with zero motion blur. Without motion blur, a higher frame rate is needed.

Since this chapter is only an introduction to computer animation, in the next sections we will only discuss creating 2D and 3D animations. The correct number of frames to create the animation or the correct frame rate to make the animation smooth will not be explored.

[1]Image in Figure 7.2 is used by permission from the author, Gphoto, under the GUN Free Documentations License.

7.2 Creating 2D Animations

The most common form of 2D computer animation is sprite animation. A sprite is a 2D object that is overlaid on a background. The configuration of a sprite is changed between frames by blending or morphing, which produces the illusion of motion. For example, the animation in Figure 7.1 is a sequence of 2D images composed of 2 sprites. The stick figure is one sprite and the flag is the other. The configuration of the flag is changed between frames to create the illusion of the flag spinning about the hand of a stick figure. Rotating the 2D primitives that compose the flag by various degrees creates the illusion of motion.

The 2D primitives of the flag are a line and triangular polygon. The line represents the flagpole, and the polygon represents the actual flag. The location of the flag and other objects in Frame00 are stated in Programming Exercises 10 in Chapter 3. To start the illusion of motion in frame01, the line and polygon of the flag are rotated by -45 degrees about the fix point (100,100). The fix point is the stick figure's hand center; therefore, the rotation moves about the hand and creates the sensation that the stick figure is causing the spinning.

Rotating the flag by -90 degrees about the fix point (100,100) creates frame02. The rotation is performed based on the parameters of the flag from frame00. Using the same fixed point of (100,100); frame03 and frame04 are rotated by -135 degrees and -180 degrees, respectively. These rotations produce a 45-degree offset of the flag between frames and thus creates the illusion of motion.

Mostly all 2D animations are based on this simple concept. The complexity occurs in defining the transformations of the sprites. In our example, we used simple rotations. However, other motions could be based on more complex mathematical equation, such as fractals.[2]

7.3 Creating 3D Animations

3D animation involves constructing a virtual world in which characters and objects move and interact. The animator must model, animate, and render the 3D scene. Briefly stated, modeling involves describing the elements of a scene and placing them appropriately. Animating specifies how the objects should move in the 3D world. Rendering converts the description of the objects and their motion into images. Modeling and rendering are, for the most part, independent of their

[2] A fractal is a rough or fragmented geometric shape that can be subdivided in parts, each of which is (at least approximately) a reduced-size copy of the whole.

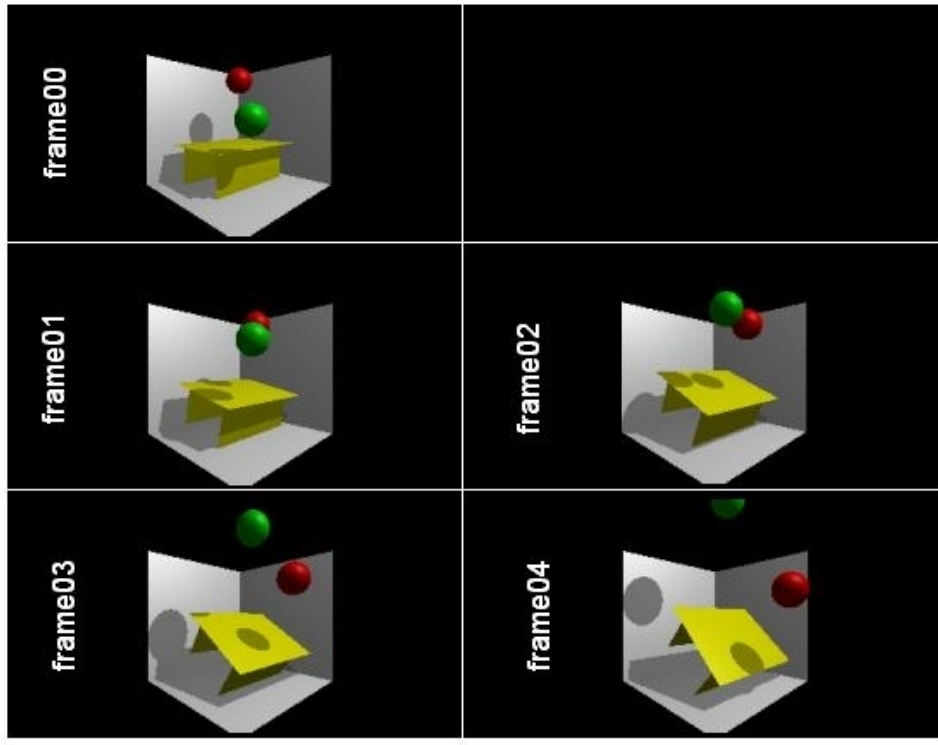


Figure 7.3. 3D Animation Example: A table and sphere moving.

role in the animation process.[3]

Figure 7.3 is an example of a 3D animation. Three objects - 1 table and 2 spheres - are animated in this sequence. The 3D environment was created based on the parameters from Table 5.1 from Chapter 5. However, the radius of sphere #1 was changed from 20 to 10, so the two spheres would not collide with each other as they moved.

Frame00 represents the original configuration of the 3D scene, and displacing the table and sphere objects derived the subsequent frames. In Frame01, the table and the red sphere were rotated by 11 degrees about the axis between points (60,0,80) and (60,0,10) - this axis is the base of the right table leg. By using this rotation axis, the table and sphere appears to tilt. In addition, the animation includes translating the green sphere by (0,17,0).

Frame02 is created by rotating the table and red sphere by 22 degrees about the axis between points (60,0,80) and (60,0,10) and translating the green sphere

[3]Information taken from Hodgins, J. K., O'Brien, J. F., Bodenheimer, R. E. *Computer Animation.* In the Wiley Encyclopedia of Electrical and Electronics Engineering, John G. Webster, ed., v. 3, pp. 686-690, 1999.

by (0,34,0). These 3D transformations are based on the original positions of the 3D objects in Frame00.

Continuing, frame03 is created by rotating the table and red sphere by 33 degrees about the axis between points (60,0,80) and (60,0,10) and translating the green sphere by (0,51,0). Finally, frame04 is created by rotating the table and red sphere by 44 degrees about the axis between points (60,0,80) and (60,0,10) and translating the green sphere by (0,68,0).

Displaying these frames one after the other in quick succession creates the sensation that the table and red sphere are tilting over and the green sphere is floating away. By using graphics concepts from previous chapters, a 3D animation was created.

As with 2D animations, mostly all 3D animations are based on simple concepts. With 3D animation those concepts are model, animate, and render. The complexity occurs in the complexity of the models created and difficulty of the animation. In our example, an animation was created using simple rotations and translations, and the scene was modeled using a simple connection of polygons and placement of spheres. More complex animations could be based on scenes containing characters created from polygon meshes and motions generated by the laws of physics, or an approximation to those laws.

APPENDIX A

PROGRAMMING AND PYTHON

A program (also referred to as **code**) is a means of communicating a sequences of instructions to a computer. These instructions are written in a language that the computer understands, so the computer can interpret and preform the instructions. It is very similar to writing a recipe in English for a meal you want to eat, and then giving that recipe to a chef to cook for you. The chef will read the recipe and follow the steps exactly as you wrote them.

The programs you are going to write in this book will be in the **Python** language. This is a language that the computer understands, and if written correctly the computer will do exactly what you want. *But what is meant by written correctly*? If you write the program with language errors (known as syntax errors), than the computer will not be able to understand what you want it to do. For example, if you want your chef to *Bake a cake with lemon*, but if you write *Bake a cake with lmn*, your chef will not understand what you want. In this example, there is a language error, instead of writing *lemon*, you wrote *lmn*, which is not part of the English language. This example holds true for the Python language also. If you create a program with incorrect language, the computer will not preform your instructions because it will not understand you.

With that said, this brief section is not designed to teach you the Python language, but instead the goal is to only introduce to. To learn more about the Python language, read through the first few chapters of "**How to Think Like a Computer Scientist**." It is a free internet book located at

http://www.ibiblio.org/obp/thinkCSpy/

and will teach you the basics of programming in Python.

A.1 Installing Python

These installation instructions are for Windows. If you are using a non Windows PC, please follow instructions at **http://www.python.org/download/** for your operating systems. If the download address given below is not an active link, goto **http://www.python.org/download** to get the latest version

of python for windows.

1. Download version 2.4.1 at
   http://www.python.org/ftp/python/2.4.1/python-2.4.1.msi
   Note: File is about 10 MB

2. Save the file python-2.4.1.msi to your hard drive.

3. Double click on python-2.4.1.msi to install python.
   Note: Don't uncheck any optional components. Some of these components are useful for you, especially the IDLE (the GUI interface to run python).

A.2 Running Python

1. Click on the "**Start**" button to bring up the menu.

2. From that menu, click on "**Programs**" to bring up another menu.

3. From that menu, click on "**Python 2.4**" to bring up another menu.

4. On this menu, click on "**IDLE (Python GUI)**."
   Note: This starts up the "Python Shell" window.

A.3 Typing Commands in the Python Shell Window

The "Python Shell" window allows you to directly type instructions for the computer to preform. For example, in the "Python Shell" window type:

>>> print "Hello World"

and press the "Enter" key on the keyboard. The result is:

```
>>> print "Hello World"
Hello World
>>>
```

Congratulations you have written your first program in the Python language. This programs instructs the computer to print out the expression "Hello World"

to the display screen. By the way, the >>> is called the "prompt." It tells you where to start typing.

A.4 Typing Commands in a File

Typing instructions directly into the Python Shell window is fine, but for more complicated programs, such as the ones in this book, you need to create a file to store your instructions. To create a file:

1. Click on the "**File**" button at the top of the "Python Shell" window.

2. This bring up a menu, and on this menu click the "**New Window**" button.
   Note: This new window is similar to the Python Shell window, but the name of this window is "Untitled."

Within this "Untitled" window, type the following instruction:

print "Hello World"

And then:

1. Click on the "**File**" button at the top of the "Untitled" window.

2. This brings up a menu, and on this menu click the "**Save As**" button.

3. A "Save As" window pops up, and save this file as "hello.py" to your hard drive.
   Note: The *.py* extension tells the computer that this file is written in the python language.

After saving the file, the "Untitled" window is now called "hello.py." To preform the instructions in this file:

1. Click on the "**Run**" button at the top of the "hello.py" window.

2. This brings up a menu, and on this menu click the "**Run Module**" button.

This runs the instructions in the "hello.py" file, and the results appear in the "Python Shell" window as:

```
>>>
Hello World
>>>
```

Now close the window "hello.py" by doing the following:

1. Click on the "**File**" button at the top of the "Untitled" window.

2. This brings up a menu, and on this menu click the "**Exit**" button.

All the programs in this book are to be written to a file as described above, and ran as described.

A.5 Loading Saved Files

If you have saved a file and want to open it again to make changes to it or run it again. The way to do this is:

1. In the "Python Shell" window, click on the "**File**" button at the top of the window.

2. This brings up a menu, and on this menu click the "**Open**" button.

3. A window pops up named "Open," and from here find and open your file.

APPENDIX B

PYTHON CODE FOR IMAGE BASICS

IN this appendix the programming code for the graphics concepts in Chapter 1 is given. The first step is to create a directory on your computer to store the files you will create. These files will contain the code for the algorithms. For this chapter, 4 files are created

1. one_color.py
2. write_ppm.py
3. index_translation.py
4. add_to_image.py

Each file will contain code to do a specified thing that corresponds to information in Chapter 1 and each file needs to be created in the order presented.

B.1 One Color

The first file to create is one_color.py, which creates an image of just one color. From the *Python Shell* window, create a file named one_color.py and save it to the code directory your created. In this file, type the following code:

```
———- [Start] ———-
# This function creates an image with
# just one color given by RGB
# and size of xd and yd

import copy

# allows any variable to be a flexible structure
class struc:
    pass

def main(xd, yd, r, g, b):
    # xd = width of image
    # yd = height of image
    # r = red color
    # g = green color
```

```
    # b = blue color

    L = xd*yd # Number of pixel in image

    image_size = copy.deepcopy(struc())
    pixel = copy.deepcopy(struc())

    # Set width and height

    image_size.x = xd
    image_size.y = yd
    image = [image_size]

    # Set pixel color

    pixel.r = r
    pixel.g = g
    pixel.b = b

    # Set all pixels to the same color

    for i in range(L):
        image.append(copy.deepcopy(pixel))
    # end for loop on i

    return image
———- [Stop] ———-
```

A few notes about typing in this code. First, python is very picky about indentions, so make sure your indentions are correct. Second, the # sign represents comments in the code. The information after the # sign is not read by the computer: it is only there to help explain how the code works to a person who is reading it. Also, don't type in ———- [Start] ———- or ———- [Stop] ———-, this just tells you when to start typing and when to stop typing.

B.2 Write PPM

The second file to create is write_ppm.py, which writes the image data to a .ppm file. From the *Python Shell* window, create a file named write_ppm.py and save it to your code directory. In this file, type the following code:

```
———- [Start] ———-
# This file takes an RGB image and creates a .ppm file.
# The input to this file is the image and a name for the image

def main(image, name):      # finds the number of pixels in the image
    L = image[0].x*image[0].y
    # Adds .ppm to file name
    name = name + '.ppm'
    s = '  '
    n = '\n'
    f = open(name,'w')

    # Write header information to PPM file

    f.write('P3\n')
    f.write('# Created by Amos Johnson Jr.\n')
    x = str(image[0].x)
    y = str(image[0].y)
    temp = x + s + y + n
    f.write(temp)
    f.write('255\n')

    # Write image data to PPM file
    c = 0
    for i in range(1,L+1):
        temp = str(image[i].r) + s + str(image[i].g) + s
        temp = temp + str(image[i].b) + '\t'
        f.write(temp)
        c = c + 1
        if c == 4: # Puts only 4 pixels per line
            c = 0
            f.write(n)
        #end of if statement
```

```
    # end of for loop
    f.close()
———— [Stop] ————
```

TIME TO TEST THIS OUT!

With files one_color.py and write_ppm.py you can now create an image that contains one color. To test these files, create a file named temp.py, and in this file type the following code:

```
———— [Start] ————
import one_color
import write_ppm

image = one_color.main(320,240,255,0,0)
write_ppm.main(image,'test')
———— [Stop] ————
```

Run temp.py, and the output is a *ppm* image named test.ppm. The size of the image is 320 by 240, and the image is all red. Use IrfanView to display the image. This image can be found in the same directory as your code.

HOW DID THAT WORK?

- First you must add one_color.py and write_ppm.py to temp.py –

  ```
  import one_color
  ```

  ```
  import write_ppm
  ```

- Next, the size of the image, 320 by 240 (where the x dimension is 320 and the y dimension is 240), and the color Red is entered into the function one_color.main –

  ```
  image = one_color.main(320,240,255,0,0)
  ```

- Finally, the name of the image *test* is entered into the function write_ppm.main

–

```
write_ppm.main(image,'test')
```

Note: Quotation marks are placed around the image name: 'test'

Use this same method to create images of different sizes and colors, and names.

B.3 Index Translation

In this section, we create a file called index_translation.py, which implements Equation 1.1 from Chapter 1. This file and the next are needed later in the book. With that said, create a file named index_translation.py and type in the following code:

```
———- [Start] ———-
# This file translates an
# x-y location to a pixel number

# xd : x-dimension
# yd : y-dimension

def main(x,y,xd,yd):
    i = x + xd*(yd-y-1)+1
    return i
———- [Stop] ———-
```

B.4 Add to Image

The last file to create in this chapter is add_to_image.py, which takes xy locations and sets those locations in the image to a particular RGB color. In your code directory, create a file named add_to_image.py, and in the file type the following code:

```
———- [Start] ———-
# This file sets the color of
# a location in the image
```

```
import index_translation
import copy

def main(image,points,r,g,b):
    L = len(points)

    for j in range(L):
        # translate x and y location to a pixel number
        x = points[j].x
        y = points[j].y
        xd = image[0].x
        yd = image[0].y

        i = index_translation.main(x,y,xd,yd)

        # set the color

        image[i].r = r
        image[i].g = g
        image[i].b = b
    #end for loop
    return image
———— [Stop] ————
```

TIME TO TEST THIS OUT!

We are now going to test files index_translation.py and add_to_image.py; however, the file add_to_image.py uses the file index_translation.py, so we only need to test file add_to_image.py. To test this file, lets do programming exercise number 3 from Chapter 1; therefore, rewrite temp.py with the following code:

```
———— [Start] ————
# Creates a PPM image of size 100 x 100. Each 10 x 10
# block of pixels alternates between blue and red
# to form a checker broad pattern

import one_color
import write_ppm
```

```
import add_to_image
import copy

def checker(boardsize_x,boardsize_y,blocksize_x,blocksize_y):
    print 'Starting'

    xwt = 0
    ywt = 0
    temp = one_color.struc()
    bluepoints = [copy.deepcopy(one_color.struc)]
    redpoints = [copy.deepcopy(one_color.struc)]

    for y in range(boardsize_y):
        t2 = y%blocksize_y
        if t2 == 0:
            ywt = ywt + 1
        # end if on t2

        ywt2 = ywt%2

        if ywt2 == 1:
            n = 1
        elif ywt2 == 0:
            n = 2
        # end if on ywt2

        for x in range(boardsize_x):
            temp.x = x
            temp.y = y
            t = x%blocksize_x
            if t == 0:
                xwt = xwt + 1
            # end if on t

            xwt2 = xwt%2

            if n == 1:
                if xwt2 == 1:
```

```
                bluepoints.append(copy.deepcopy(temp))
            elif xwt2 == 0:
                redpoints.append(copy.deepcopy(temp))
            # end if on xwt2
        elif n == 2:
            if xwt2 == 1:
                redpoints.append(copy.deepcopy(temp))
            elif xwt2 == 0:
                bluepoints.append(copy.deepcopy(temp))
            # end if on xwt2
        # end if on n
    # end for loop on x
# end for loop on y

del redpoints[0] # remove unused location
del bluepoints[0] # remove unused location

image = one_color.main(boardsize_x,boardsize_y,255,255,255)
image = add_to_image.main(image,redpoints,255,0,0)
image = add_to_image.main(image,bluepoints,0,0,255)

write_ppm.main(image,'test')
print 'Finish'
# end checker function

# Run checker function
checker(100,100,10,10)
```

——— [Stop] ———

Run temp.py and your result is a *ppm* image named test.ppm. This image should look exactly like the image from programming exercise number 3.

HOW DID THAT WORK?

- First, temp.py uses one_color.py to create an initial board that is all white –

```
image = one_color.main(boardsize_x,boardsize_y,255,255,255)
```

- Second, it uses add_to_image.py to set the pixels that should be red to red –

```
image = add_to_image.main(image,redpoints,255,0,0)
```

- Third, it uses add_to_image.py again to set the pixels that should be blue to blue –

```
image = add_to_image.main(image,bluepoints,0,0,255)
```

- Finally, it used write_ppm.py to create the final image –

```
write_ppm.main(image,'test')
```

In this Appendix you created 4 files

1. one_color.py
2. write_ppm.py
3. index_translation.py
4. add_to_image.py

You also created the file temp.py to test your code. Using these files as a foundation complete the programming exercises in the other chapters.

APPENDIX C

TRANSFORMATION MATRICES

In mathematics, a matrix (plural matrices) is a rectangular table of numbers (Figure C.1) or, more generally, a table consisting of abstract quantities that can be added and multiplied. Matrices are used to describe linear equations, keep track of the coefficients of linear transformations and to record data that depend on two parameters.

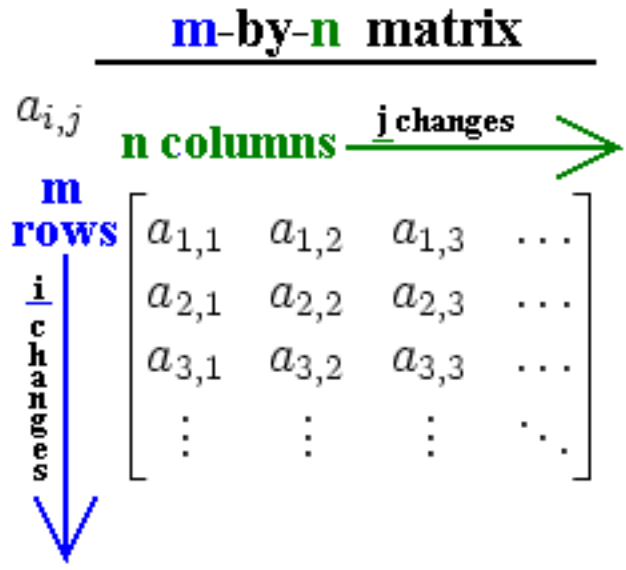


Figure C.1. Matrix example

The horizontal lines in a matrix are called rows and the vertical lines are called columns. A matrix with m rows and n columns is called an m-by-n matrix (written m x n) and m and n are called its dimensions. The dimensions of a matrix are always given with the number of rows first, then the number of columns. It is commonly said that a matrix m x n has an order of m x n (order meaning size).

The entry of a matrix A that lies in the i-th row and the j-th column is called the i, j entry or (i, j)-th entry of A. This is written as $A_{i,j}$ or $A[i, j]$. The row is always noted first, then the column.

A matrix where one of the dimensions equals one is often called a vector, and interpreted as an element of a real coordinate space. A 1 x n matrix (one row and n columns) is called a row vector, and an m x 1 matrix (one column and m rows) is called a column vector.[1]

Because matrices can be added, multiplied (see Figure C.2), and decomposed in various ways, they are key concepts in linear algebra, matrix theory, and computer graphics. Computer graphics uses matrices to perform 2D and 3D transformation.

$$\begin{bmatrix} 1 & 0 & 2 \\ -1 & 3 & 1 \end{bmatrix} \times \begin{bmatrix} 3 & 1 \\ 2 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} (1\times 3+0\times 2+2\times 1) & (1\times 1+0\times 1+2\times 0) \\ (-1\times 3+3\times 2+1\times 1) & (-1\times 1+3\times 1+1\times 0) \end{bmatrix}$$
$$= \begin{bmatrix} 5 & 1 \\ 4 & 2 \end{bmatrix}$$

Figure C.2. Matrix multiplication example.

Throughout this book, 2D and 3D transformations were computed using linear equations. However, these linear equations can be represented as a matrix, allowing 2D and 3D transformations by matrix multiplication.

C.1 2D Transformation Matrices

When using 2D transformation matrices, a 2D points is represented as a 3 x 1 column vector,

$$\mathbf{P} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

The extra coordinate (1) converts the 2D point into Homogeneous coordinates. This extra coordinate is simply used as a pad, so all the 2D transformations – *translation, scale*, and *rotation* – may be perform using matrix multiplication. A 2D transformation is represented as a 3 x 3 matrix,

[1] Information taken from Wikipedia - The Free Encyclopedia, permission to used granted under the GNU Free Documentation License.

$$\mathbf{T} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix},$$

and is multiplied with the point vector,

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix},$$

to produce the new transformed point (x', y'). The padding allows the matrix multiplication to occur because two matrices must have the same internal dimensions when multiplying. In this class, the number of columns (3) of the transformation matrix must match the number of rows (3) of the point vector.

In the following subsections, the matrices used to compute 2D transformations (*translation, scale*, and *rotation*) are discussed.

C.1.1 Translation

The translation transformation matrix is

$$\mathbf{T}(x_t, y_t) = \begin{bmatrix} 0 & 0 & x_t \\ 0 & 0 & y_t \\ 0 & 0 & 1 \end{bmatrix},$$

where (x_t, y_t) is the amount of translation to apply to a point.

C.1.2 Scale

The scale transformation matrix is

$$\mathbf{S}(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

where s_x and s_y are the scale factors for the x and y coordinates (respectively) of a point. However, as discussed in section 3.3, a point must be scaled by a fixed point (x_r, y_r).

The scaling process begins by first translating a point by $\mathbf{T}(-x_r, -y_r)$. Second, scale the result by $\mathbf{S}(s_x, s_y)$, and third, translate by $\mathbf{T}(x_r, y_r)$. To achieve a correct scale, these three steps are combined into one matrix using matrix multiplication,

$$\begin{bmatrix} 0 & 0 & x_r \\ 0 & 0 & y_r \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 0 & 0 & -x_r \\ 0 & 0 & -y_r \\ 0 & 0 & 1 \end{bmatrix},$$

to create a new transformation matrix that will scale a point about a fixed point. Notice how the matrix are range in reverse order. Also, multiply from left to right to create the new transformation matrix.

This is a great example of the power of using matrices instead of a set of linear equations. Because these transformations are represented by matrices, individual transformations such as translation and scale can be combine into one "super" transformation-matrix. Once the super transformation-matrix is created, the individual transformation are no longer needed. Removing the individual transformations and using the super transformation-matrix reduces the number of steps to transform 2D points.

C.1.3 Rotation

The Rotation transformation matrix is

$$\mathbf{R}(\phi) = \begin{bmatrix} cos(\phi) & -sin(\phi) & 0 \\ sin(\phi) & cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

where ϕ is the number of degrees to rotate a point. However section 3.2 explains that a correct rotation requires a fixed point (x_r, y_r) to rotate by.

The rotating process begins by first translating a point by $\mathbf{T}(-x_r, -y_r)$. Second, rotate the result by $\mathbf{R}(\phi)$, and third, translate back by $\mathbf{T}(x_r, y_r)$. To achieve a correct rotation, these three steps are combined into one matrix using matrix multiplication,

$$\begin{bmatrix} 0 & 0 & x_r \\ 0 & 0 & y_r \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} cos(\phi) & -sin(\phi) & 0 \\ sin(\phi) & cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 0 & 0 & -x_r \\ 0 & 0 & -y_r \\ 0 & 0 & 1 \end{bmatrix},$$

to create a new transformation matrix that will rotate a point about a fixed point.

C.2 3D Transformation Matrices

In 3 dimensions, a 3D points is represented as a 4 x 1 column vector,

$$\mathbf{P} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix},$$

and the 3D transformation matrices are of size 4 x 4,

$$\mathbf{T} = \begin{bmatrix} a & b & c & j \\ d & e & f & k \\ g & h & i & l \\ m & n & o & p \end{bmatrix},$$

which is multiplied by the point vector,

$$\begin{bmatrix} a & b & c & j \\ d & e & f & k \\ g & h & i & l \\ m & n & o & p \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$$

to produce the new transformed point (x', y', z'). In the following subsections, the matrices used to compute 3D transformations (*translation, scale*, and *rotation*) are discussed.

C.2.1 Translation

The translation transformation matrix is

$$\mathbf{T}(x_t, y_t, z_t) = \begin{bmatrix} 0 & 0 & 0 & x_t \\ 0 & 0 & 0 & y_t \\ 0 & 0 & 0 & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where (x_t, y_t, z_t) is the amount of translation to apply to a point.

C.2.2 Scale

The scale transformation matrix is

$$\mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where s_x, s_y, s_z are the scale factors for the x, y, z coordinates (respectively) of a point. However, as discussed in section 6.2, a point must be scaled by a fixed point (x_r, y_r, z_r).

The scaling process begins by first translating a point by $\mathbf{T}(-x_r, -y_r, -z_r)$, secondly scaling the result by $\mathbf{S}(s_x, s_y, s_z)$, and thirdly translating back by $\mathbf{T}(x_r, y_r, z_r)$. To achieve a correct scale, these three steps are combined into one matrix using matrix multiplication,

$$\begin{bmatrix} 0 & 0 & 0 & x_r \\ 0 & 0 & 0 & y_r \\ 0 & 0 & 0 & z_r \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 0 & -x_r \\ 0 & 0 & 0 & -y_r \\ 0 & 0 & 0 & -z_r \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

to create a new transformation matrix that will scale a point about a fixed point.

C.2.3 Rotation

The rotation transformation is defined by rotating about an axis - see section 6.3. Rotation can be done about the x-axis, y-axis, z-axis, or an arbitrary axis.

The transformation matrix to rotate about the x-axis is

$$\mathbf{R_x}(\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\phi) & -sin(\phi) & 0 \\ 0 & sin(\phi) & cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where ϕ is the number of degrees to rotate about the x-axis. The transformation matrix to rotate about the y-axis is

$$\mathbf{R_y}(\phi) = \begin{bmatrix} cos(\phi) & 0 & sin(\phi) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\phi) & 0 & cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where ϕ is the number of degrees to rotate about the y-axis. The transformation matrix to rotate about the z-axis is

$$\mathbf{R_z}(\phi) = \begin{bmatrix} cos(\phi) & -sin(\phi) & 0 & 0 \\ sin(\phi) & cos(\phi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where ϕ is the number of degrees to rotate about the z-axis.

Rotating a point about an arbitrary axis requires using the translation, x-axis rotation, y-axis rotation, and z-axis rotation transformation-matrices. The rotation process is explained using the example from section 6.3.2.

To rotate a point Φ degrees about an arbitrary axis defined between points $p_1 = (x_1, y_1, z_1)$ and $p_2 = (x_2, y_2, z_2)$ (see Figure 6.4(a))

1. Translate the point by $\mathbf{T}(-x_1, -y_1, -z_1)$
2. Rotate the result about the y-axis by $\mathbf{R_y}(-\beta)$
3. Rotate the result about the x-axis by $\mathbf{R_x}(\alpha)$
4. Rotate the result about the z-axis by $\mathbf{R_z}(\Phi)$
5. Rotate the result about the x-axis by $\mathbf{R_x}(-\alpha)$
6. Rotate the result about the y-axis by $\mathbf{R_y}(\beta)$
7. Translate the result by $\mathbf{T}(x_1, y_1, z_1)$

To achieve a correct rotation, these seven steps are combined into one matrix using matrix multiplication,

$$\mathbf{T}(x_1, y_1, z_1)\mathbf{R_y}(\beta)\mathbf{R_x}(-\alpha)\mathbf{R_z}(\Phi)\mathbf{R_x}(\alpha)\mathbf{R_y}(-\beta)\mathbf{T}(-x_1, -y_1, -z_1),$$

to create a new transformation matrix that will rotate a point Φ degrees about an arbitrary axis.

APPENDIX D

ANSWERS TO REVIEW QUESTIONS

Chapter 1 Answers

1. 54106

2. P3
   4 3
   255

   255 255 255 0 0 0 255 0 0 0 255 0
   0 0 255 255 255 255 0 0 0 255 0 0
   0 255 0 0 0 255 255 255 255 0 0 0

Chapter 2 Answers

1. (10,15), (11,14), (12,14), (13,13)

2. (5,0), (5,1), (5,2), (4,3)

3. (5,0), (4,1), (3,2), (2,2), (1,2), (0,2)

4. All points along the line segment are inside the polygon if it is convex, and this is not true if the polygon is concave.

5. (140,250)

Chapter 3 Answers

1. (34,29), (26,86)

2. (27,15), (45,128)

3. (-75,280), (-40,325)

Chapter 4 Answers

1. $\vec{u} = (0.8192, 0, 0.5736)$
   $\vec{v} = (-0.1485, 0.9659, 0.21201)$
   $\vec{n} = (-0.5540, -0.2588, 0.7912)$

2. (14.142,-65,-42.929)
   (14.142,-80,-64.142)

3. (8.93,17.86), (37.5,65)

4. (65,19), (55,21)

Chapter 5 Answers

1. Question 1

   (a) (146.33, 168.67, 40)
   (b) (147.66, 167.34, 79.68)
   (c) The sphere's at (147.66, 167.34, 79.68)

2. Question 2

   (a) Sphere = (50, -5, -140), Light source = (-25,-20,-130)
   (b) cop = (0,0,0)
   (c) vrp = (0,0,-20)

(d) sphere: Center = (750,-75,-2100), radius = 600,
Light source: (-375,-300,-1950)
View Window: TLC = (-450,450,-300), BRC = (450,-450,-300)

3. Question 3

(a) (0,0,40700)
(b) 0.1
(c) 0.24198
(d) 0.05067
(e) R = 43, G = 10, B = 32

Chapter 6 Answers

---

1. (144.97, 160.42, 185.24)