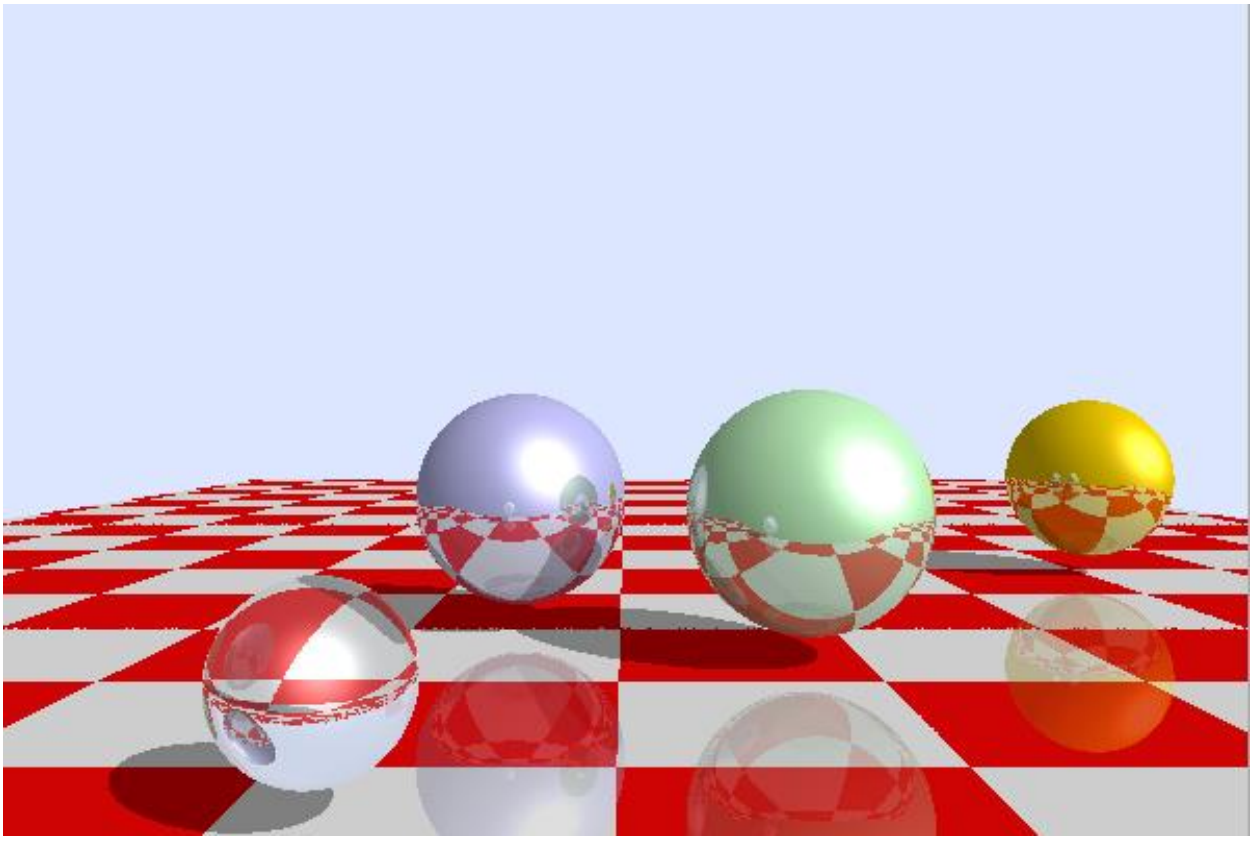
Computer Science 470

Computer Graphics

Lecture Notes

Dr. Mike O'Neal

(Winter 2020-2021)



Sample of Student Work: Assignment 5 (Programmed by Hunter Anderson)

Table of Contents	Page
PART 1: Fundamentals of Wire Frame Graphics	

Table of Contents	Page
PART 2: Making Objects Appear Solid	
11. Background: Linear Algebra	35
11.1 Points and Vectors	35
11.2 Matrix Representation	35
11.3 Multiplying a Scalar Quantity by a Vector	36
11.4 Vectors Add and Subtract Component-wise	36
11.5 Associative, Commutative, and Distributive Properties of Vectors	36
11.6 Computing the Magnitude of a Vector	37
11.7 Unit Vectors	38
11.8 Basic Properties of Vector Magnitudes	38
11.9 Dot Product	39
11.9.1 Computing the Dot Product of Two Vectors	39
11.9.2 The Importance of the Sign of the Dot Product	39
11.10 Cross Product	40
12. Back face Culling	42
12.1 Establishing a Polygon's Orientation in 3-D Space	43
12.2 Establishing a Polygon's Position in 3-D Space	45
12.3 Determining if an Arbitrary Point is Above, In, or Below the Plane of a Polygon	46
12.4 The Back Face Culling Algorithm	48
13. Polygon Filling	49
13.1 A Description of the Polygon Fill Process	50
13.2 The Polygon Fill Algorithm	54
14. Using a Z-Buffer to Determine Occlusion	56
14.1 An Overview of Z-Buffers	56
14.2 Computing the Z values	57
14.3 The Z-Buffer Algorithm	61
PART 3: Lighting and Shading	
15. Light and Color	63
15.1 Light and its Properties	63
15.2 The Perception of Color	65
15.3 Color Display Devices	67
15.4 RGB versus CMY Color Systems	70
16. Illumination and Reflection	71
16.1 Developing an Illumination Model for Computer Graphics	71
16.2 Ambient Diffuse Reflection	73
16.3 Emitted Diffuse Reflection of Point Light Sources	74
16.4 Specular Reflection	79
16.4.1 The Phong Specular Reflection Equations	82
16.4.2 Computing the Reflection Vector, R	84
16.5 The Complete Illumination Model: 20 Spheres Example	86
17. Polygon Shading	92
17.1 Flat Shading	93
17.2 Gouraud Shading	96
17.3 Phong Shading	102

1. Display Devices

Ultimately, to be seen, computer graphics end up producing images on display devices, such as monitors. This is true even of VR which consists to separate display devices, one for each eye.

Display devices include: **LCD**'s, **OLED**s, Plasma displays, etc.

An older type of display device (that ruled for over ½ a century) was the **CRT** or Cathode Ray Tube.

Early CRTs were **Vector** devices. Later CRTs were **Raster** devices.

The quality of a displayed image is dependent on its resolution.

Display resolutions:

	Horizontal resolution	X	Vertical resolution	X	Bytes per pixel	
	1024	X	768	X	3	= 2.304 MB
	1024	X	768	X	4	= 3.072 MB
HD	1920	X	1080	X	3	= 6.075 MB
	1920	X	1080	X	4	= 8.100 MB
4K	3840	X	2160	X	4	= 33.178 MB

Common (modern) display resolutions tend to use a 16:9 aspect ratio, reflecting monitor shape. Older broadcast TV resolution used a 4:3 aspect ratio, reflecting the fact that CRT-based TV's were closer to square in shape.

Interlaced versus progressive: Sometimes you will see an (I) or an (P) after the resolution numbers, to indicate interlaced or progressive updates

For comparison **Standard TV broadcast resolution** was 640 by 480, interlaced.

Pixel color: In addition to the number of pixels, image quality depends on the available colors that each pixel can take on. The standard is 3 bytes per pixel, with each byte (8 bits) providing 256 levels of Red, Green, and Blue. All together this gives 16.8 million colors. This hasn't changed in several decades, since this is pretty much at the limit of the sensitivity to color of the human eye. (Sometimes a 4 byte is added to add transparency or Z-buffer [we will talk about this later].)

Refresh rate: For moving images, the quality of animation depends not only or resolution, but also on the refresh rate – the rate as which images can be updated. Movies (based on actual film) use 24 frames per second, with each frame shown twice (to reduce flicker) so 48 individual images per second. We generally need at least 30 frames per second to keep flicker from becoming noticeable, Games tend to be unhappy with less than 60 frames per second.

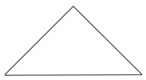
Pixel rate: Computer graphics require extremely fast hardware.

HD graphics 1920 X 1080 X 24 bits (3 bytes) = 49,766,400 bits must be produced per image.

Given the need for 60 frames per second we need: 19,766,400 X 60 = 2,985,984,000 or 3B bits or 3 GHZ

2. Polygons

Throughout the vast majority of this class, and throughout most of modern computer graphics, 3-D objects are composed of (generally very large numbers of very small) polygons. A polygon is a flat, multisided figure – like a triangle, square, or rectangle. Generally, we require that edges not cross one another.



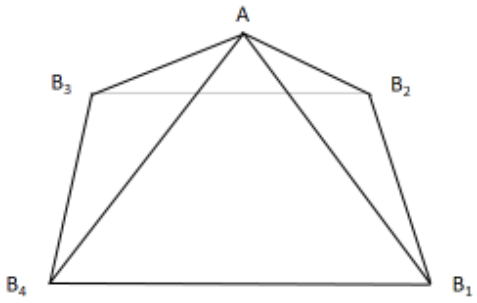
Polygon – Triangle

Polygon – Rectangle

Polygon? (not in this class)

For simplicity, most modern computer graphics system limit themselves to triangles. In this class we will generally use triangles, but may occasionally use rectangles. Note that any arbitrary polygon can be constructed from multiple triangles. For example, a rectangle can be composed of two triangles.

3-D objects can be constructed from multiple planar (flat) polygons. Consider this 3-D pyramid. It consists of five polygons: 4 triangles (**front**, **back**, **left**, **right**) and a rectangular [actually a square] **base**. These names are arbitrary, as which side is "front" and which side is "back" depends completely on the position of the viewer. [Note: if the polygon were solid you would not be able to see the "back face".]



The equations we will study in this class are *linear* equations. Linear, in this case, simply means that straight lines are converted to straight lines. This property will allow us to focus solely on the vertices making up our polygons.

The five polygons (front, back, left, right, and base) are defined using only five total vertices: A (for "Apex"), and B1, B2, B3, and B4 (where the B stands for "base").

We can think of our pyramid as an object defined in a hierarchical manner: The "pyramid" object consists of a collection of five polygonal objects ("Front", "Back", "Left", "Right", "Base") where each of these objects consists of a list of vertices, specified in clockwise order, *when viewed from the outside*.

Pyramid: Front, Back, Left, Right, Base
Front: A, B1, B4 Back: A, B3, B2 Left: A, B4, B3 Right: A, B2, B1 Base: B1, B2, B3, B4
Vertices: A, B1, B2, B3, B4

An object may be rendered (drawn) by drawing each of its polygons. A polygon can be drawn by tracing lines from its first vertex to its second, from its second to its third, and so on; ending with a line from its final vertex back to its first vertex. Note that each edge is drawn twice for a closed, convex object.

3. Perspective Projection

Though this class focuses on 3-D worlds, ultimately to see "into" that world, we will need to generate a 2-D (flat) image. This is true even for VR in which two 2-D images are generated; one for each eye, drawn from slightly different perspectives.

To generate a projection in a LHS (Left Handed System) we imagine the (human) viewer being at some distance, d, in front of the view screen, where that screen is coincident with the XY plane at $Z = 0$. All of the objects we are modeling will be in the +Z region of space, the viewer exists in the -Z region of space. The viewer is also assumed to be looking directly at the center of the screen / window, which is taken to be the origin (0,0,0). The viewer is located at (0,0,-d). This position is called the **view point**.

A Note for Later: *If we make a simplifying assumption, which we often do, of thinking of the distance, d, as being very large (essentially infinite), then the vector $\langle 0,0,-d \rangle$ will always point towards the viewer's location. This simplifies polygon visibility calculations, as the view vector is always the same, no matter where you are in space (as long as you are in the +Z region).*

Let's say we want to project the 3-D point (X,Y,Z) on to the XY Plane (where $Z = 0$). Here are the three key equations for perspective projection:

$$\boldsymbol{X_{ps} = d \cdot \frac{X}{d+Z}} \qquad \boldsymbol{Y_{ps} = d \cdot \frac{Y}{d+Z}} \qquad \boldsymbol{Z_{ps} = d \cdot \frac{Z}{d+Z}}$$

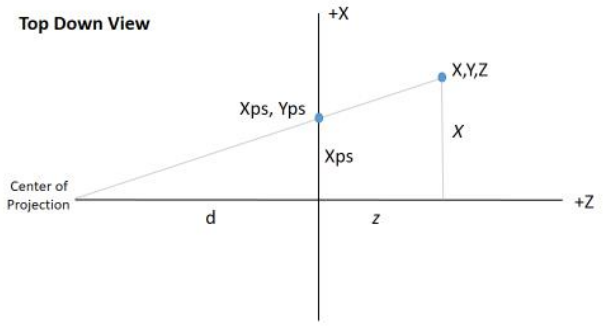
Where: $(X, Y, Z)$is the point to be projected.

d is the distance from the origin on XY plane to the center of projection.

(X_{ps}, Y_{ps}) is the projected point on the XY plane / screen

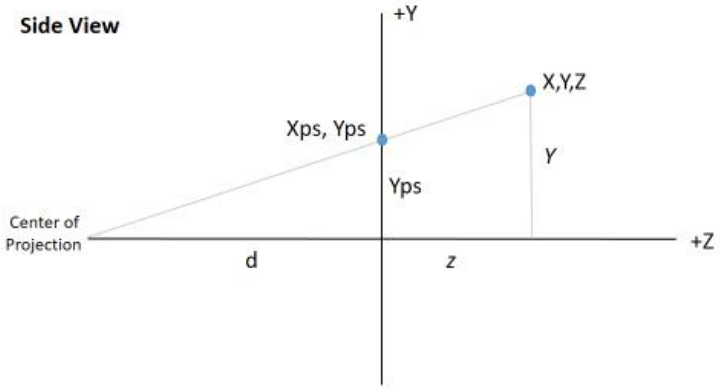
Another Note for Later: *What is Z_{ps}? Since we are converting to 2-D, keeping a Z value around doesn't seem to make much sense. Well, actually it does. Z_{ps} is a measure of the relative distance to the point (X,Y,Z) from the viewpoint (0,0,-d). We keep this "relative measure of distance" around for our projected points as it comes in handy for determining when one point is in front of another, when viewed from the viewpoint.*

Here is a **top down view** of the geometry of prospective projection, **looking from +Y toward the origin**.



Given this image, it is easy to derive the equation for computing X_{ps}. You might recall from trigonometry that the tangent is defined as $TAN = \frac{OPP}{ADJ}$. Thus, by similar triangles, $\frac{X_{ps}}{d} = \frac{X}{d+Z}$ and so $X_{ps} = d \cdot \frac{X}{d+Z}$

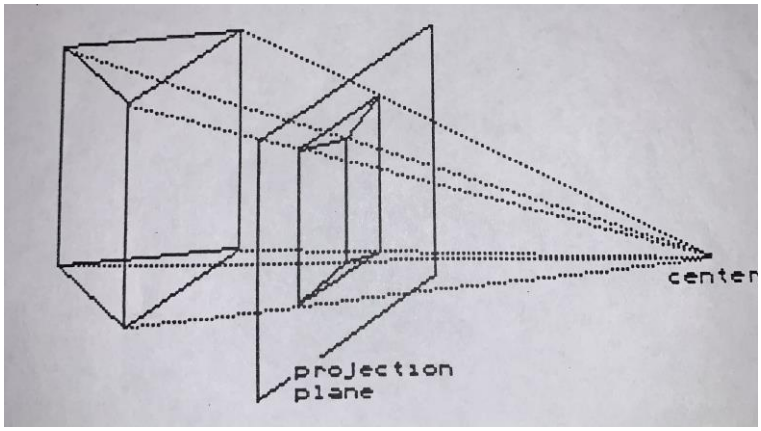
Here is a **side view** of the geometry of prospective projection, **looking from +X toward the origin**.



Again, given $TAN = \frac{OPP}{ADJ}$ by similar triangles, $\frac{Y_{ps}}{d} = \frac{Y}{d+Z}$ and so $Y_{ps} = d \cdot \frac{Y}{d+Z}$

These perspective equations are linear, which means that straight lines in 3-D space will be mapped to straight lines in 2-D space.

We can use this knowledge to save a tremendous amount of time and effort when drawing polygons. Essentially, we only need to the project the 3-D vertices of our polygons when rendering them on the screen and then draw lines in 2-D space between those vertices. Drawing 2-D lines is often much, much faster (orders of magnitude faster) than trying to project each point along the edges of the polygon in 3-D space.

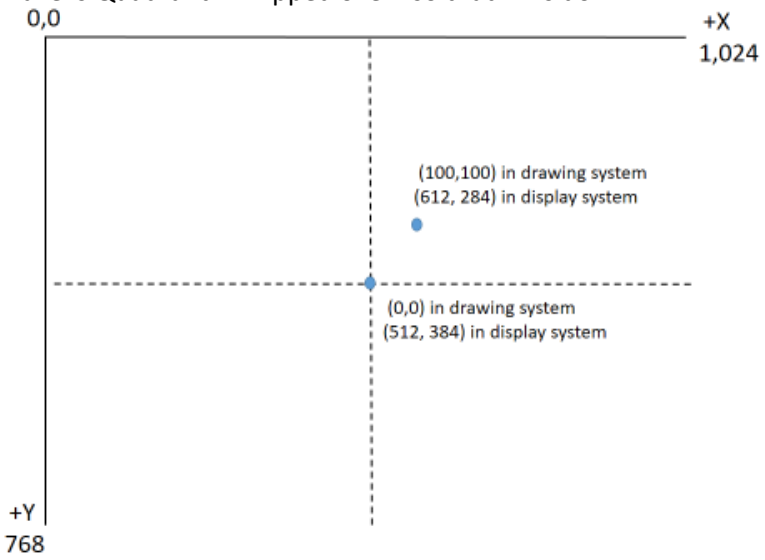


4. Conversion to Display Coordinates

The perspective projection equations are organized with the notion that the origin of the coordinate system is positioned at the center of the display window – with the viewer located somewhere in front of the window (in -Z in our LHS) looking directly at the center of the display window (towards the origin).

However, many popular graphical display systems are organized differently. (0,0) is often located in the upper left hand corner of the display window. To complicate things even more, +Y is DOWN, not up.

Essentially what you have is Quadrant 1 "flipped over" so that +Y is down.



Layout of a Typical 1024 by 768 Graphics Display Window
[The origin (0,0) of the Drawing System is at (512, 384) in this 1,024 by 768 Display System]

The way we reconcile these differences is that immediately before we draw a projected point, we convert from drawing coordinates to display coordinates.

$$X_{display} = \frac{DisplayWindowWidth}{2} + X_{ps}$$

$$Y_{display} = \frac{DisplayWindowHeight}{2} - Y_{ps}$$

Example:

Given a 1,024 by 768 display window, organized as shown above:

$(X_{ps}, Y_{ps}) = (100, 100)$ translates to $(X_{display}, Y_{display}) = (612, 284)$

$$X_{display} = \frac{1024}{2} + 100 = 512 + 100 = 612$$

$$Y_{display} = \frac{768}{2} - 100 = 384 - 100 = 284$$

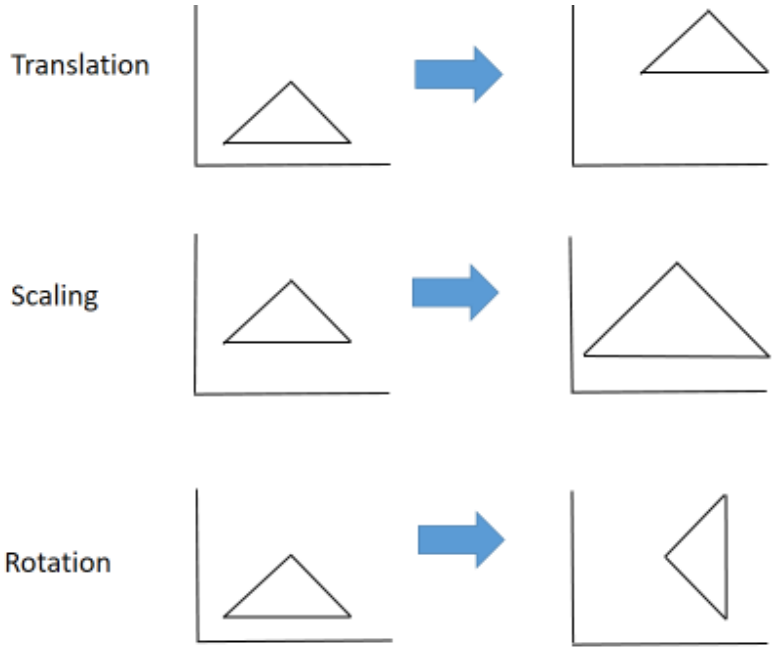
5. 2-D Transformations

Transformations are a way of manipulating graphical objects (points, lines, polygons). There are three basic kinds of transformations: Translation, Scaling, and Rotation; and these three kinds of transformations come in both 2-D and 3-D version, where the 3-D versions are simple extensions of the basic 2-D versions. Thus, we begin with 2-D transformations.

Translation : Moves an object by an arbitrary amount in an arbitrary direction.

Scaling : changes the size of an object. Can be used to "squeeze" or "stretch" objects.

Rotation : Rotates an object around the origin, or around some specified point.



Our transformation equations for translation, scaling, and rotation are *linear* in nature. This means that they transform straight lines to straight lines. This is very important, as we will apply the transformation to individual points that generally represent the vertices of polygons.

The translation equations are position **independent**.

The Scaling and Rotation equations are position **dependent**, and are thus defined to take place about the origin.

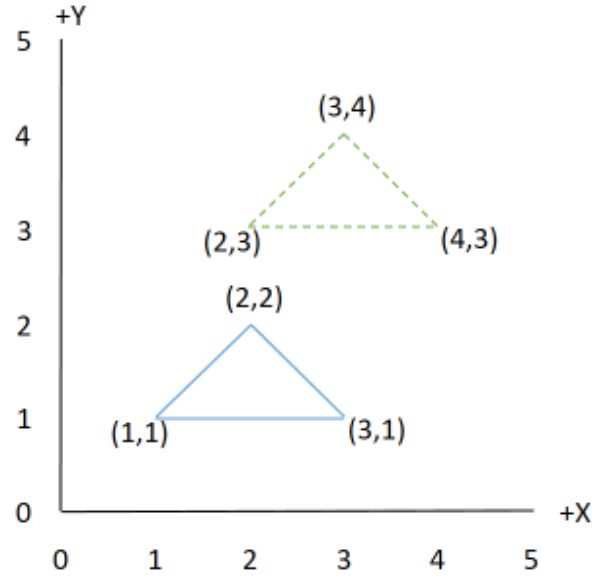
5.1 2-D Translation

The purpose of translation is to move the location of an object by two independent factors: T_X and T_Y. Translation is "position independent" in that the size and shape of the translated object does not depend on the starting position of the original object.

$\boldsymbol{X_T = X + T_X}$ Where: T_X is distance that the X coordinate will be shifted.
Positive T_X shifts to the right. Negative T_X shifts to the left.

$\boldsymbol{Y_T = Y + T_Y}$ Where: T_Y is distance that the Y coordinate will be shifted.
Positive T_Y shifts up. Negative T_Y shifts down.

Example:



$T_X = 1$ and $T_Y = 2$

$Vertex_1 = (X_1, Y_1) = (1,1) \quad X_{T_1} = 1 + 1 = 2 \quad Y_{T_1} = 1 + 2 = 3$
$Vertex_{T_1} = (X_{T_1}, Y_{T_1}) = (2,3)$

$Vertex_2 = (X_2, Y_2) = (2,2) \quad X_{T_2} = 2 + 1 = 3 \quad Y_{T_2} = 2 + 2 = 4$
$Vertex_{T_2} = (X_{T_2}, Y_{T_2}) = (3,4)$

$Vertex_3 = (X_3, Y_3) = (3,1) \quad X_{T_3} = 3 + 1 = 4 \quad Y_{T_3} = 1 + 2 = 3$
$Vertex_{T_3} = (X_{T_3}, Y_{T_3}) = (4,3)$

5.2 2-D Scaling

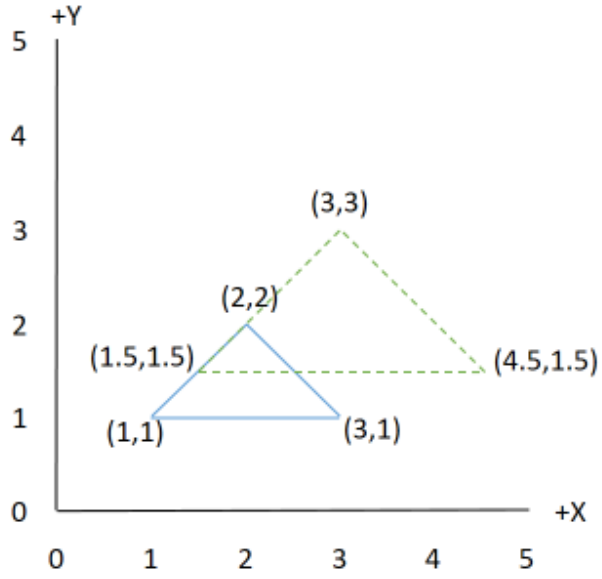
The purpose of scaling is to change the size of an object. Normally, we think about scaling by a single factor in both the X and Y dimensions. Doing so is called "uniform scaling". However, the equations are such that you can scale by different amounts in different directions. You can even shrink an object in one dimension, while simultaneously expanding it in another dimension.

Scaling is position dependent. If your object is not centered at the origin, it will move in space while changing size. We will learn how to address this issue in subsequent lectures.

$\boldsymbol{X_S = X \cdot S_X}$ Where: S_X is the scale factor along the horizontal, X, dimension.
An $S_X > 1$ increases the size of the object in X, while $S_X < 1$ decreases the size of the object in X.

$\boldsymbol{Y_S = Y \cdot S_Y}$ Where: S_Y is the scale factor along the vertical, Y, dimension.
An $S_Y > 1$ increases the size of the object in Y, while $S_Y < 1$ decreases the size of the object in Y.

Example 1 (Uniform scaling of an object not centered on the origin)



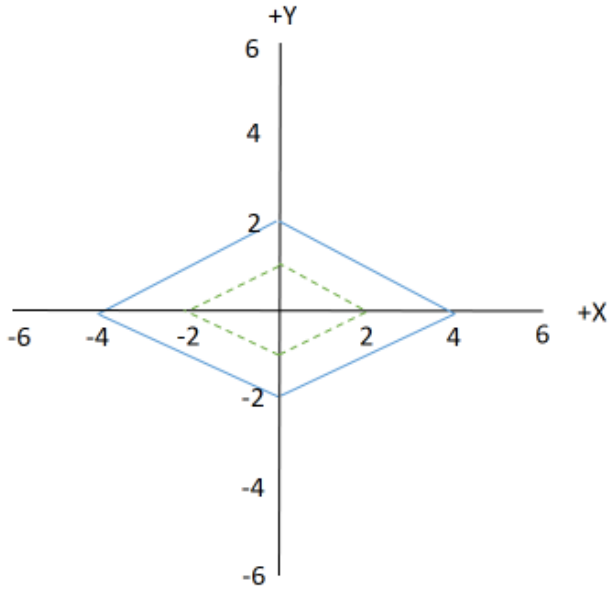
$S_X = S_Y = 1.5$

$Vertex_1 = (X_1, Y_1) = (1,1) \qquad X_{S_1} = 1 \cdot 1.5 = 1.5 \quad Y_{S_1} = 1 \cdot 1.5 = 1.5$
$Vertex_{S_1} = (X_{S_1}, Y_{S_1}) = (1.5,1.5)$

$Vertex_2 = (X_2, Y_2) = (2,2) \qquad X_{S_2} = 2 \cdot 1.5 = 3 \qquad Y_{S_2} = 2 \cdot 1.5 = 3$
$Vertex_{S_2} = (X_{S_2}, Y_{S_2}) = (3,3)$

$Vertex_3 = (X_3, Y_3) = (3,1) \qquad X_{S_3} = 3 \cdot 1.5 = 4.5 \quad Y_{S_3} = 1 \cdot 1.5 = 1.5$
$Vertex_{S_3} = (X_{S_3}, Y_{S_3}) = (4.5,1.5)$

Example 2 (Uniform scaling of an object that is centered on the origin)



$S_X = S_Y = 0.5$

$Vertex_1 = (X_1, Y_1) = (0,2) \quad X_{S_1} = 0 \cdot 0.5 = 0 \quad Y_{S_1} = 2 \cdot 0.5 = 1$
$Vertex_{S_1} = (X_{S_1}, Y_{S_1}) = (0,1)$

$Vertex_2 = (X_2, Y_2) = (4,0) \quad X_{S_2} = 4 \cdot 0.5 = 2 \quad Y_{S_2} = 0 \cdot 0.5 = 0$
$Vertex_{S_2} = (X_{S_2}, Y_{S_2}) = (2,0)$

$Vertex_3 = (X_3, Y_3) = (0,-2) \quad X_{S_3} = 0 \cdot 0.5 = 0 \quad Y_{S_3} = -2 \cdot 0.5 = -1$
$Vertex_{S_3} = (X_{S_3}, Y_{S_3}) = (0,-1)$

$Vertex_4 = (X_4, Y_4) = (-4,0) \quad X_{S_3} = -4 \cdot 0.5 = -2 \quad Y_{S_3} = 0 \cdot 0.5 = 0$
$Vertex_{S_4} = (X_{S_4}, Y_{S_4}) = (-2,0)$

Some interesting facts about scaling:

- Scaling factors are generally assumed to be positive.
- When $S_X = S_Y = 0$ scaling reduces the object to a point at the origin; all vertices become zero.
- When $S_X = S_Y = 1$ scaling leaves the object unchanged.
- Scaling factors > 1 increase the size of the object. (e.g., $S_X = S_Y = 2$ doubles the object size)
- Scaling factors < 1 decrease the size of the object. (e.g., $S_X = S_Y = 0.5$ shrinks the object in half)
- Scaling factors may be independent. S_X and S_Y are not required to be equal.
- And always remember, if your object is not centered on the origin, the equations for scaling have the side effect of translating the object, as well as scaling it.

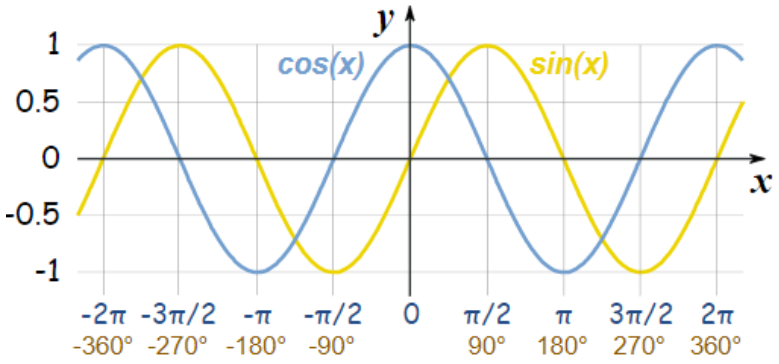
5.3 2-D Rotation

The purpose of the rotation equations is to rotate an object around the origin (0,0) through some angle ϕ (phi) in the counter clockwise direction – from +X to +Y.

Rotation is position dependent. If the "visual center" of your object is located at the origin (0,0), the object will appear to rotate in-place about that visual center. If the visual center of the object is not on the origin (0,0) the object will still rotate about the origin – it will appear to rotate through an arc.

$$X_R = X \cdot \cos(\phi) - Y \cdot \sin(\phi)$$

$$Y_R = X \cdot \sin(\phi) + Y \cdot \cos(\phi)$$



Basic Definition of Sin and Cos (Thank you MathIsFun.com!)

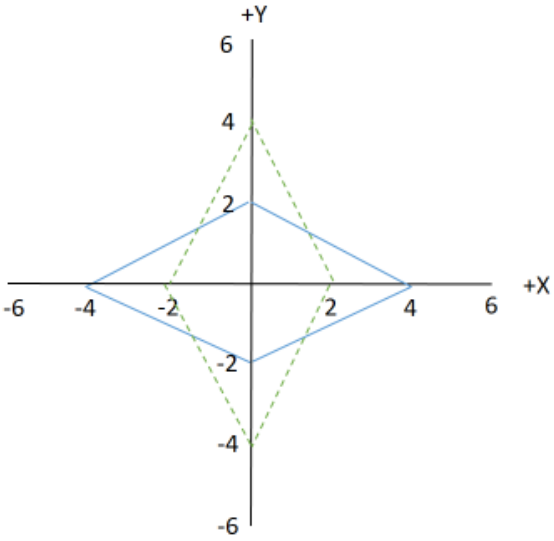
Example 1: Counter Clockwise Rotation 90° from +X to +Y

Let $\phi = 90°$ $\sin(90°) = 1$ $\cos(90°) = 0$

$X_R = X \cdot \cos(\phi) - Y \cdot \sin(\phi)$
$X_R = X \cdot \cos(90°) - Y \cdot \sin(90°)$
$X_R = X \cdot 0 - Y \cdot 1$
$X_R = -Y$

$Y_R = X \cdot \sin(\phi) + Y \cdot \cos(\phi)$
$Y_R = X \cdot \sin(90°) + Y \cdot \cos(90°)$
$Y_R = X \cdot 1 + Y \cdot 0$
$Y_R = X$

Points	Rotated Points
(0,2)	(-2,0)
(4,0)	(0,4)
(0.-2)	(2,0)
(-4,0)	(0,-4)



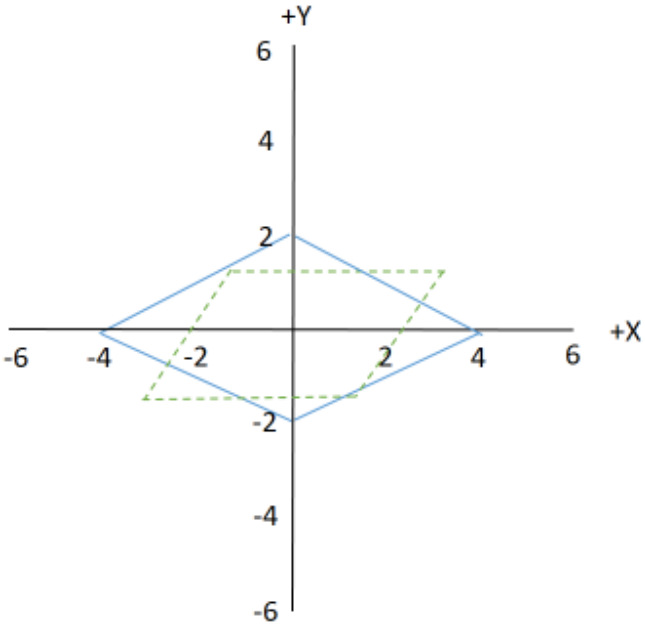
Example 2: Counter Clockwise Rotation 45° from +X to +Y

Let $\phi = 45°$ $\quad \sin(45°) = \frac{\sqrt{2}}{2} = 0.707$ $\quad \cos(45°) = \frac{\sqrt{2}}{2} = 0.707$

$X_R = X \cdot \ cos\ (\phi)\ - Y\ \cdot sin\ (\phi)$
$X_R = X \cdot\ cos\ (45°) - Y\ \cdot sin\ (45°)$
$X_R = X \cdot\ 0.707\ \ \ - Y\ \cdot 0.707$
$X_R = (X - Y) \cdot 0.707$

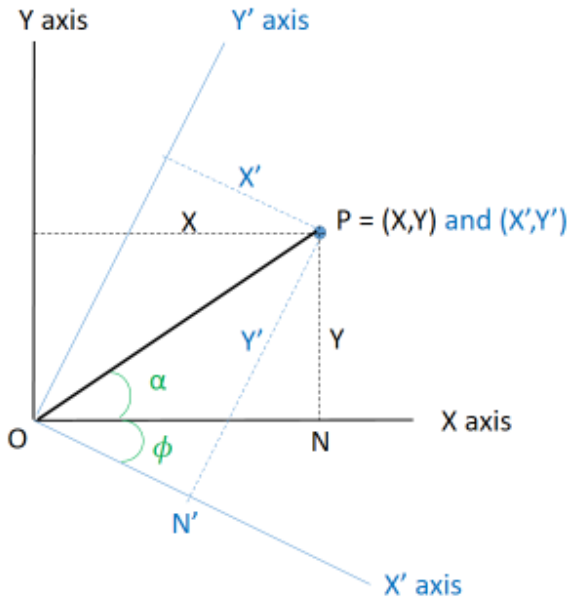
$Y_R = X \cdot\ \ sin\ (\phi)\ \ + Y\ \cdot\ cos\ (\phi)$
$Y_R = X \cdot\ sin\ (45°) + Y\ \cdot cos\ (45°)$
$Y_R = X \cdot\ \ 0.707\ \ + Y\ \cdot\ \ 0.707$
$Y_R = (X + Y) \cdot 0.707$

Points	Rotated Points
(0,2)	(-1.4, 1.4)
(4,0)	(2.8, 2.8)
(0.-2)	(1.4, -1.4)
(-4,0)	(-2.8,-2.8)



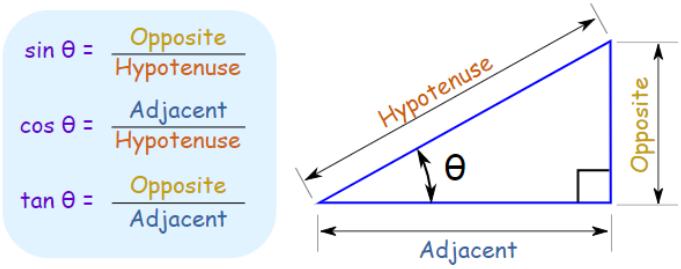
5.4 Derivation of Rotation Formulas

The equations for rotation can be derived from basic trigonometry. Rotation of a point counter clockwise about the origin through the angle ϕ is equivalent to rotating the X-Y axes in a clockwise direction by the angle ϕ. Viewing the problem in this way simplifies deriving the equations.



Viewing a CCW rotation through the angle ϕ as a CW rotation of the coordinate system through ϕ

To begin to understand how the above figure can be used to derive the rotation equations, let's begin by reviewing the definition of sine, cosine, and tangent in terms of the ratios of the sides of a right triangle.

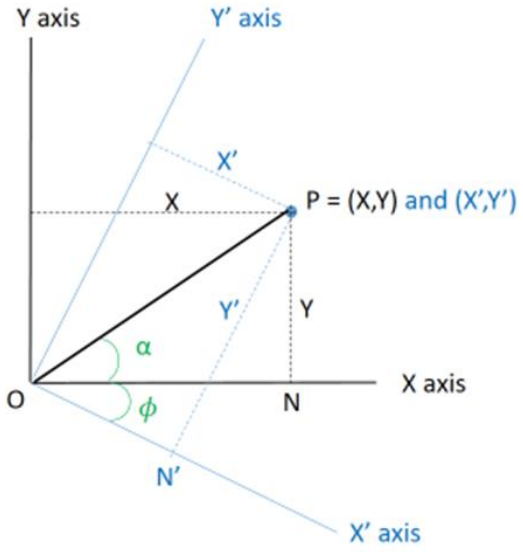


Definition of Sine, Cosine, and Tangent using the ratio of sides of a right triangle (MathIsFun.com)

Also, though we've all long ago forgotten, we once learned in trigonometry that the following are true:

$$(1) \quad \cos(\alpha + \phi) = \cos(\alpha) \cdot \cos(\phi) - \sin(\alpha) \cdot \sin(\phi)$$

$$(2) \quad \sin(\alpha + \phi) = \sin(\alpha) \cdot \cos(\phi) + \cos(\alpha) \cdot \sin(\phi)$$



Equation	Reason
(1) $\cos(\alpha+\phi)=\cos(\alpha)\cdot\cos(\phi)-\sin(\alpha)\cdot\sin(\phi)$	Trigonometry
(2) $\sin(\alpha+\phi)=\sin(\alpha)\cdot\cos(\phi)+\cos(\alpha)\cdot\sin(\phi)$	Trigonometry
$X=ON$	See diagram
$X=ON=OP\cdot\frac{ON}{OP}$	$\frac{OP}{OP}$ = 1 and you can always multiply by 1
(3) $X=ON=OP\cdot\frac{ON}{OP}=OP\cdot cos(\alpha)$	$\cos(\alpha)=\frac{adjacent}{hypotenuse}=\frac{ON}{OP}$
$Y=NP$	See diagram
$Y=NP=OP\cdot\frac{NP}{OP}$	$\frac{OP}{OP}$ = 1 and you can always multiply by 1
(4) $Y=NP=OP\cdot\frac{NP}{OP}=OP\cdot sin(\alpha)$	$\sin(\alpha)=\frac{opposite}{hypotenuse}=\frac{NP}{OP}$
$X'=ON'$	See diagram
$X'=ON'=OP\cdot\frac{ON'}{OP}$	$\frac{OP}{OP}$ = 1 and you can always multiply by 1
$X'=ON'=OP\cdot\frac{ON'}{OP}=OP\cdot cos(\alpha+\phi)$	$\cos(\alpha+\phi)=\frac{adjacent}{hypotenuse}=\frac{ON'}{OP}$
$X'=OP\cdot(\cos(\alpha)\cdot\cos(\phi)-\sin(\alpha)\cdot\sin(\phi))$	Equation (1)
$X'=OP\cdot\cos(\alpha)\cdot\cos(\phi)-OP\cdot\sin(\alpha)\cdot\sin(\phi)$	Algebra
$\boldsymbol{X'=X\cdot\cos(\phi)-Y\cdot\sin(\phi)}$	Equations (3) and (4)
$Y'=N'P$	See diagram
$Y'=N'P=OP\cdot\frac{N'P}{OP}$	$\frac{OP}{OP}$ = 1 and you can always multiply by 1
$Y'=N'P=OP\cdot\frac{N'P}{OP}=OP\cdot sin(\alpha+\phi)$	$\sin(\alpha+\phi)=\frac{opposite}{hypotenuse}=\frac{N'P}{OP}$
$Y'=OP\cdot(\sin(\alpha)\cdot\cos(\phi)+\cos(\alpha)\cdot\sin(\phi))$	Equation (2)
$Y'=OP\cdot\sin(\alpha)\cdot\cos(\phi)+OP\cdot\cos(\alpha)\cdot\sin(\phi)$	Algebra
$\boldsymbol{Y'=Y\cdot\cos(\phi)+X\cdot\sin(\phi)}$	Equations (4) and (3)

6. 3-D Transformations

The 3-D versions of translation, scaling, and rotation are straightforward extensions of their 2-D forms.

6.1 3-D Translation

$\boldsymbol{X_T = X + T_X}$ Where: T_X is distance that the X coordinate will be shifted.
Positive T_X shifts to the right. Negative T_X shifts to the left.

$\boldsymbol{Y_T = Y + T_Y}$ Where: T_Y is distance that the Y coordinate will be shifted.
Positive T_Y shifts up. Negative T_Y shifts down.

$\boldsymbol{Z_T = Z + T_Z}$ Where: T_Z is distance that the Z coordinate will be shifted.
In LHS, positive T_Z shifts away from viewer, negative T_Z shifts towards viewer.

6.2 3-D Scaling

$\boldsymbol{X_S = X \cdot S_X}$ Where: S_X is the scale factor along the horizontal, X, dimension.
An $S_X > 1$ increases the size of the object in X, while $S_X < 1$ decreases the size of the object in X.

$\boldsymbol{Y_S = Y \cdot S_Y}$ Where: S_Y is the scale factor along the vertical, Y, dimension.
An $S_Y > 1$ increases the size of the object in Y, while $S_Y < 1$ decreases the size of the object in Y.

$\boldsymbol{Z_S = Z \cdot S_Z}$ Where: S_Z is the scale factor along the distance, Z, dimension.
An $S_Z > 1$ increases the distance to the object in Z, while $S_Z < 1$ decreases the distance to the object.

6.3 3-D Rotation

There are three sets of 3-D rotation equations, one for each axis.

Z-axis rotation – LHS, rotation CCW when looking "forward" from (0,0,–Z) to origin; from +X to +Y

$$\boldsymbol{X_{R_Z} = X \cdot cos(\phi) - Y \cdot sin(\phi)}$$
$$\boldsymbol{Y_{R_Z} = X \cdot sin(\phi) + Y \cdot cos(\phi)}$$
$$\boldsymbol{Z_{R_Z} = Z}$$ **/*Rotation about Z leaves Z values unchanged */**

Y-axis rotation – LHS, rotation CCW when looking "up" from (0,-Y,0) to origin; from +Z to +X

$$\boldsymbol{X_{R_Y} = X \cdot cos(\phi) + Z \cdot sin(\phi)}$$
$$\boldsymbol{Y_{R_Y} = Y}$$ **/*Rotation about Y leaves Y values unchanged */**
$$\boldsymbol{Z_{R_Y} = -X \cdot sin(\phi) + Z \cdot cos(\phi)}$$

X-axis rotation – LHS, rotation CCW when looking "sideways" from (-X,0,0) to origin; from +Y to +Z

$$\boldsymbol{X_{R_X} = X}$$ **/*Rotation about X leaves X values unchanged */**
$$\boldsymbol{Y_{R_X} = Y \cdot cos(\phi) - Z \cdot sin(\phi)}$$
$$\boldsymbol{Z_{R_X} = Y \cdot sin(\phi) + Z \cdot cos(\phi)}$$

7. In-Place Rotation and Scaling

7.1 In-Place 2-D Rotation

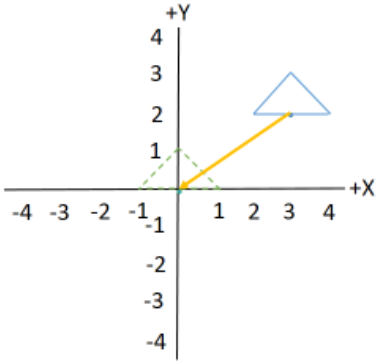
Given a triangle defined by the three vertices (2,2), (3,3), (4, 2). Let's say we want to rotate it counter clockwise 90° about the point (3,2). Note that this point, (3,2), is not the exact visual center of the triangle [that would be (3, 2.5)] but it is the center point of the base of the triangle.

This "in-place" rotation can be accomplished in three steps.

(1) Translate the polygon to align its reference point with the origin, using $T_X = -3 \quad T_Y = -2$.

$X_a = X - 3 \qquad Y_a = Y - 2$

Points	Translated Points
(2, 2)	(-1, 0)
(3, 3)	(0, 1)
(4, 2)	(1, 0)

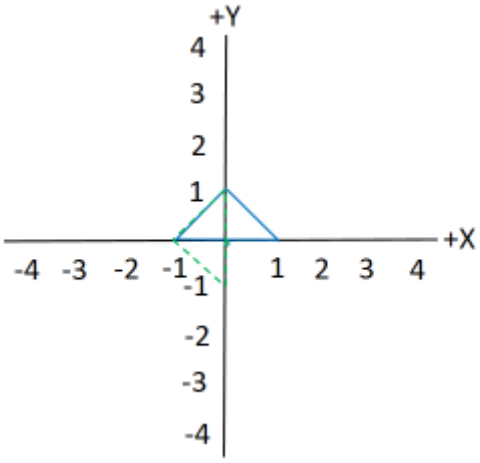


(2) Rotate the polygon about the origin by 90° counter clockwise. [$\cos(90°) = 0 \quad \sin(90°) = 1$]

$X_b = X_a \cdot \cos(90°) - Y_a \cdot \sin(90°)$
$X_b = X_a \cdot \quad 0 \quad - Y_a \cdot \quad 1$
$X_b = -Y_a$

$Y_b = X_a \cdot \sin(90°) + Y_a \cdot \cos(90°)$
$Y_b = X_a \cdot \quad 1 \quad + Y_a \cdot \quad 0$
$Y_b = X_a$

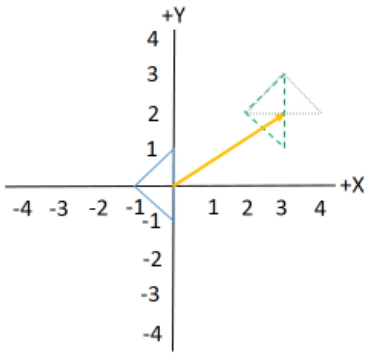
Points	Rotated Points
(-1, 0)	(0, -1)
(0, 1)	(-1, 0)
(1, 0)	(0, 1)



(3) Translate the rotated polygon back to the original reference point, using $T_X = 3 \quad T_Y = 2$.

$X_c = X_b + 3 \qquad Y_c = Y_b + 2$

Points	Translated Points
(0, -1)	(3, 1)
(-1, 0)	(2, 2)
(0, 1)	(3, 3)



7.2 In-Place 2-D Scaling

In-Place Scaling about some arbitrary reference point is similar to in-place rotation, in that it requires a three step process.

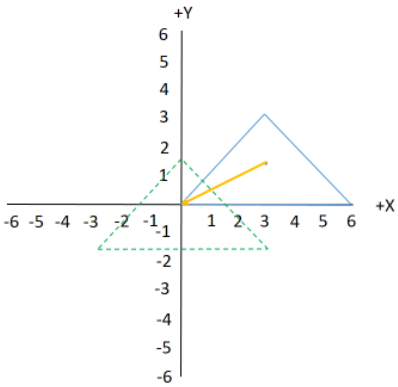
Given a triangle defined by the three vertices (0,0), (3,3), (6, 0). Let's say we want to scale it in-place by 0.5. The visual center of this polygon is (3, 1.5), so that become our reference point.

This "in-place" rotation can be accomplished in three steps.

(1) Translate the polygon to align its reference point with the origin, using $T_X = -3 \quad T_Y = -1.5$.

$X_a = X - 3 \qquad Y_a = Y - 1.5$

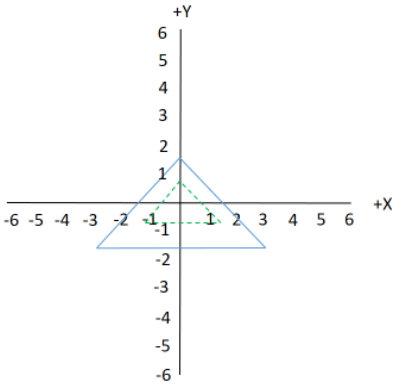
Points	Translated Points
(0, 0)	(-3, -1.5)
(3, 3)	(0, 1.5)
(6, 0)	(3, -1.5)



(2) Scale the polygon about the origin by 50%, using $S_X = 0.5 \quad S_Y = 0.5$.

$X_b = X_a \cdot 0.5 \qquad Y_b = Y_a \cdot 0.5$

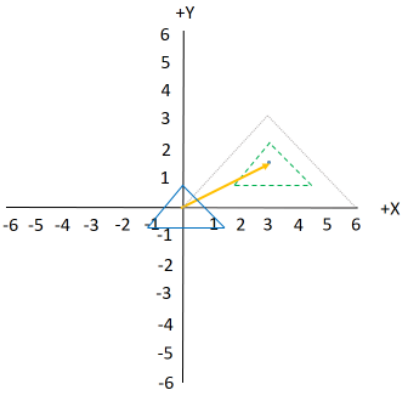
Points	Scaled Points
(-3, -1.5)	(-1.5, -0.75)
(0, 1.5)	(0, 0.75)
(3, -1.5)	(1.5, -0.75)



(3) Translate the scaled polygon back to the original reference point, using $T_X = 3 \quad T_Y = 1.5$

$X_c = X_b + 3 \qquad Y_c = Y_b + 1.5$

Points	Translated Points
(-1.5, -0.75)	(1.5, 0.75)
(0, 0.75)	(3, 2.25)
(1.5, -0.75)	(4.5, 0.75)



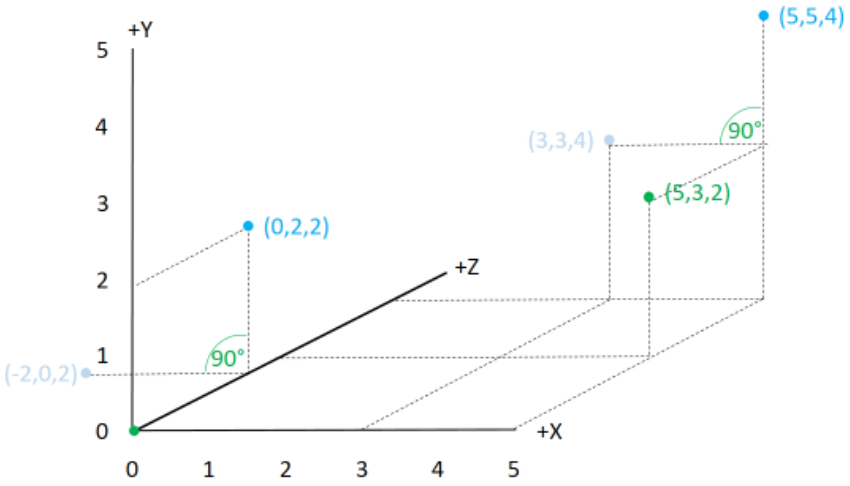
7.3 In-Place 3-D rotation and Scaling

In-Place rotation and scaling in 3-D work similarly to their 2-D counterparts. As before, the three step process is:

(1) Translate the polygon to align its reference point with the origin.
(2) Rotate (or scale) the polygon
(3) Translate the rotated (or scaled) polygon back to its original reference point

The only difference is that you will use the 3-D transformation equations from Section 6, as opposed to the 2-D transformations of Section 5.

Example: Perform a 90° CCW in-place Z-axis rotation of the point $P = (5,5,4)$ about the reference point $Reference = (5,3,2)$



Step 1:

$$X_a = P_X - Reference_X \qquad Y_a = P_Y - Reference_Y \qquad Z_a = P_Z - Reference_Z$$

$$X_a = 5 - 5 = 0 \qquad Y_a = 5 - 3 = 2 \qquad Z_a = 4 - 2 = 2$$

Step 2:

$$X_b = X_a \cdot cos\,(90°) - Y_a \cdot sin\,(90°)$$
$$X_b = X_a \cdot \quad 0 \quad - Y_a \cdot \quad 1$$
$$X_b = -Y_a = -2$$

$$Y_b = X_a \cdot sin\,(90°) + Y_a \cdot cos\,(90°)$$
$$Y_b = X_a \cdot \quad 1 \quad + Y_a \cdot \quad 0$$
$$Y_b = X_a = 0$$

$$Z_b = Z_a = 2$$

Step 3:

$$X_c = X_b + Reference_X \qquad Y_c = Y_b + Reference_Y \qquad Z_c = Z_b + Reference_Z$$

$$X_c = -2 + 5 = 3 \qquad X_c = 0 + 3 = 3 \qquad Z_a = 2 + 2 = 4$$

7.4 Computing the Reference Point / Estimating the "Visual Center" of an Object

In the examples presented so far, a reference point was supplied about which the scaling and rotation operations could take place. Often, we are asked to rotate or scale an object "in-place" but no reference point, representing the "visual center" of the object is explicitly defined.

A technique that generally produces "natural looking" results is to compute the center point of a "bounding box" around an object and use that box's center point as the reference point. In 3-D space an object's "bounding box" is simply the smallest rectangular box that can contain that object. In 2-D space the "bounding box" simplifies to a "bounding rectangle".

Given the vertices of an object, computing the center point of the bounding box is quite easy.

First, examine all of the object's vertices noting the smallest X value (X_{min}), the largest X value (X_{max}), the smallest Y value (Y_{min}), the largest Y value (Y_{max}), the smallest Z value (Z_{min}), and the largest Z value (Z_{max}). The center point of the bounding box can be computed as:

$$Center_X = X_{min} + \frac{X_{max} - X_{min}}{2} \qquad Center_Y = Y_{min} + \frac{Y_{max} - Y_{min}}{2} \qquad Center_Z = Z_{min} + \frac{Z_{max} - Z_{min}}{2}$$

which simplifies algebraically to:

$$Center_X = \frac{X_{min} + X_{max}}{2} \qquad Center_Y = \frac{Y_{min} + Y_{max}}{2} \qquad Center_Z = \frac{Z_{min} + Z_{max}}{2}$$

Example: Compute the "visual center" reference point of a pyramid defined by the following 5 vertices:

$$A = (0, 50, 100) \quad B1 = (50, -50, 50) \quad B2 = (50, -50, 150) \quad B3 = (-50, -50, 150) \quad B4 = (-50, -50, 50)$$

$$X_{min} = -50 \quad X_{max} = 50 \quad Y_{min} = -50 \quad Y_{max} = 50 \quad Z_{min} = 50 \quad Z_{max} = 150$$

$$Center_X = \frac{(-50) + 50}{2} \qquad Center_Y = \frac{(-50) + 50}{2} \qquad Center_Z = \frac{50 + 150}{2}$$

$$Reference = (Center_X, Center_Y, Center_Z) = (0,0,100)$$

8. Representing the Transformation Equations in Matrix Notation

8.1 Homogeneous Coordinates

In homogeneous coordinates we include a "1" as a third "coordinate" in the "ordered pair" for specifying a 2-D point. So, (X,Y) becomes (X,Y,1). Similarly, in homogeneous coordinates the point (X,Y,Z) become (X,Y,Z,1).

I know it seems odd to do this at first, but by making this representational tweak, we will be able to represent our 2-D transformation equations using 3 by3 matrix multiplication; and our 3-D transformations using 4 by 4 matrix multiplications.

8.2 2-D Transformations in Matrix Notation

Here we present the 2-D transformations (translation, scaling, and rotation) using matrix notation. Note that the result of the matrix multiplications are completely equivalent to the equations given previously.

8.2.1 2-D Translation

Translation of $(X, Y, 1)$ by (T_X, T_Y) giving $(X_T, Y_T, 1)$

$$(X_T, Y_T, 1) = (X, Y, 1) \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_X & T_Y & 1 \end{bmatrix}$$

$$\begin{aligned} X_T &= X \cdot 1 + Y \cdot 0 + 1 \cdot T_X = X + T_X \\ Y_T &= X \cdot 0 + Y \cdot 1 + 1 \cdot T_Y = Y + T_Y \\ 1 &= X \cdot 0 + Y \cdot 0 + 1 \cdot 1 = 1 \end{aligned}$$

8.2.2 2-D Scaling

Scaling $(X, Y, 1)$ by (S_X, S_Y) giving $(X_S, Y_S, 1)$

$$(X_S, Y_S, 1) = (X, Y, 1) \cdot \begin{bmatrix} S_X & 0 & 0 \\ 0 & S_Y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{aligned} X_S &= X \cdot S_X + Y \cdot 0 + 1 \cdot 0 = X \cdot S_X \\ Y_S &= X \cdot 0 + Y \cdot S_Y + 1 \cdot 0 = Y \cdot S_Y \\ 1 &= X \cdot 0 + Y \cdot 0 + 1 \cdot 1 = 1 \end{aligned}$$

8.2.3 2-D Rotation

Rotation of $(X, Y, 1)$ by ϕ giving $(X_R, Y_R, 1)$

$$(X_R, Y_R, 1) = (X, Y, 1) \cdot \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{aligned} X_R &= X \cdot \cos\phi - Y \cdot \sin\phi + 1 \cdot 0 = X \cdot \cos\phi - Y \cdot \sin\phi \\ Y_R &= X \cdot \sin\phi + Y \cdot \cos\phi + 1 \cdot 0 = X \cdot \sin\phi + Y \cdot \cos\phi \\ 1 &= X \cdot 0 + Y \cdot 0 + 1 \cdot 1 = 1 \end{aligned}$$

8.3 3-D Transformations in Matrix Notation

In this section we present the matrix multiplication versions of the 3-D transformation equations. The most significant different between 2-D and 3-D transformations is that there are three basic rotations – one for each axis (X, Y, and Z).

8.3.1 3-D Translation

Translation of $(X, Y, Z, 1)$ by (T_X, T_Y, T_Z) giving $(X_T, Y_T. Z_T, 1)$

$$(X_T, Y_T, Z_T, 1) = (X, Y, Z, 1) \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_X & T_Y & T_Z & 1 \end{bmatrix}$$

$$\begin{aligned} X_T &= X \cdot 1 + Y \cdot 0 + Z \cdot 0 + 1 \cdot T_X = X + T_X \\ Y_T &= X \cdot 0 + Y \cdot 1 + Z \cdot 0 + 1 \cdot T_Y = Y + T_Y \\ Z_T &= X \cdot 0 + Y \cdot 0 + Z \cdot 1 + 1 \cdot T_Z = Z + T_Z \\ 1 &= X \cdot 0 + Y \cdot 0 + Z \cdot 0 + 1 \cdot 1 = 1 \end{aligned}$$

8.3.2 3-D Scaling

Scaling of $(X, Y, Z, 1)$ by (S_X, S_Y, S_Z) giving $(X_S, Y_S, Z_S, 1)$

$$(X_S, Y_S, Z_S, 1) = (X, Y, Z, 1) \cdot \begin{bmatrix} S_X & 0 & 0 & 0 \\ 0 & S_Y & 0 & 0 \\ 0 & 0 & S_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{aligned} X_S &= X \cdot S_X + Y \cdot 0 + Z \cdot 0 + 1 \cdot 0 = X \cdot S_X \\ Y_S &= X \cdot 0 + Y \cdot Y_S + Z \cdot 0 + 1 \cdot 0 = Y \cdot S_Y \\ Z_S &= X \cdot 0 + Y \cdot 0 + Z \cdot S_Z + 1 \cdot 0 = Z \cdot S_Z \\ 1 &= X \cdot 0 + Y \cdot 0 + Z \cdot 0 + 1 \cdot 1 = 1 \end{aligned}$$

8.3.3 Z-axis Rotation in 3-D Space

The 2-D rotation we have already covered can be thought of as Z-axis rotation in 3-D space. We use a **Left-Handed system** (where +Z points away from the viewer) in these equations to produce a counter clockwise rotation (from +X to +Y) when looking along the Z axis, toward the origin, from point (0, 0, –Z).

Rotation of $(X, Y, Z, 1)$ by ϕ about the Z axis CCW looking "forward" from (0,0,-Z) to origin; +X to +Y

$$\left(X_{R_Z}, Y_{R_Z}, Z_{R_Z}, 1\right) = (X, Y, Z, 1) \cdot \begin{bmatrix} \cos\phi & \sin\phi & 0 & 0 \\ -\sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{aligned} X_{R_Z} &= X \cdot \cos\phi - Y \cdot \sin\phi + Z \cdot 0 + 1 \cdot 0 = X \cdot \cos\phi - Y \cdot \sin\phi \\ Y_{R_Z} &= X \cdot \sin\phi + Y \cdot \cos\phi + Z \cdot 0 + 1 \cdot 0 = X \cdot \sin\phi + Y \cdot \cos\phi \\ Z_{R_Z} &= X \cdot 0 + Y \cdot 0 + Z \cdot 1 + 1 \cdot 0 = Z \quad \textbf{/*Rotation about Z leaves Z values unchanged */} \\ 1 &= X \cdot 0 + Y \cdot 0 + Z \cdot 0 + 1 \cdot 1 = 1 \end{aligned}$$

8.3.4 Y-axis Rotation in 3-D Space

These equations assume a **Left-Handed system** (where +Z points away from the viewer). They produce a counter clockwise rotation (from +Z to +X) when looking "up" along the Y axis from a point (0, -Y, 0) toward the origin. [Clockwise when viewed from (0,Y,0).]

Rotation of $(X, Y, Z, 1)$ by ϕ about the Y axis CCW looking "up" from (0,-Y,0) to origin; from +Z to +X

$$\left(X_{R_Y}, Y_{R_Y}, Z_{R_Y}, 1\right) = (X, Y, Z, 1) \cdot \begin{bmatrix} \cos\phi & 0 & -\sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{aligned} X_{R_Y} &= X \cdot \cos\phi + Y \cdot 0 + Z \cdot \sin\phi + 1 \cdot 0 = X \cdot \cos\phi + Z \cdot \sin\phi \\ Y_{R_Y} &= X \cdot 0 + Y \cdot 1 + Z \cdot 0 + 1 \cdot 0 = Y \quad \textbf{/*Rotation about Y leaves Y values unchanged*/} \\ Z_{R_Y} &= -X \cdot \sin\phi + Y \cdot 0 + Z \cdot \cos\phi + 1 \cdot 0 = -X \cdot \sin\phi + Z \cdot \cos\phi \\ 1 &= X \cdot 0 + Y \cdot 0 + Z \cdot 0 + 1 \cdot 1 = 1 \end{aligned}$$

Example: Rotation of (0,0,1) by 90° about the Y axis (from +Z to +X) produces (1,0,0)

$$\sin(90°) = 1 \qquad \cos(90°) = 0$$

$$\begin{aligned} X_{R_Y} &= X \cdot \cos\phi + Y \cdot 0 + Z \cdot \sin\phi + 1 \cdot 0 = X \cdot \cos\phi + Z \cdot \sin\phi = 0 \cdot 0 + 1 \cdot 1 = 1 \\ Y_{R_Y} &= X \cdot 0 + Y \cdot 1 + Z \cdot 0 + 1 \cdot 0 = Y = 0 \\ Z_{R_Y} &= -X \cdot \sin\phi + Y \cdot 0 + Z \cdot \cos\phi + 1 \cdot 0 = -X \cdot \sin\phi + Z \cdot \cos\phi = -0 \cdot 1 + 1 \cdot 0 = 0 \\ 1 &= X \cdot 0 + Y \cdot 0 + Z \cdot 0 + 1 \cdot 1 = 1 \end{aligned}$$

8.3.5 X-axis Rotation in 3-D Space

These equations assume a **Left-Handed system** (where +Z points away from the viewer). They produce a counter clockwise rotation (from +Y to +Z) when looking "sideways" along the X axis from a point at (-X, 0, 0) toward the origin. [Clockwise when viewed from (X,0,0).]

Rotation of $(X, Y, Z, 1)$ by ϕ about the X axis CCW looking "sideways" from (-X,0,0) to origin; +Y to +Z

$$\left(X_{R_X}, Y_{R_X}, Z_{R_X}, 1\right) = (X, Y, Z, 1) \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & \sin\phi & 0 \\ 0 & -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{aligned} X_{R_X} &= X \cdot 1 + Y \cdot 0 + Z \cdot 0 + 1 \cdot 0 = X \quad \textbf{/*Rotation about X leaves X values unchanged*/} \\ Y_{R_X} &= X \cdot 0 + Y \cdot \cos\phi - Z \cdot \sin\phi + 1 \cdot 0 = Y \cdot \cos\phi - Z \cdot \sin\phi \\ Z_{R_X} &= X \cdot 0 + Y \cdot \sin\phi + Z \cdot \cos\phi + 1 \cdot 0 = Y \cdot \sin\phi + Z \cdot \cos\phi \\ 1 &= X \cdot 0 + Y \cdot 0 + Z \cdot 0 + 1 \cdot 1 = 1 \end{aligned}$$

Example: Rotation of (0,1,0) by 90° about the X axis (from +Y to +Z) produces (0,0,1)

$$\sin(90°) = 1 \qquad \cos(90°) = 0$$

$$\begin{aligned} X_{R_X} &= X \cdot 1 + Y \cdot 0 + Z \cdot 0 + 1 \cdot 0 = X = 0 \\ Y_{R_X} &= X \cdot 0 + Y \cdot \cos\phi - Z \cdot \sin\phi + 1 \cdot 0 = Y \cdot \cos\phi - Z \cdot \sin\phi = 1 \cdot 0 - 0 \cdot 1 = 0 \\ Z_{R_X} &= X \cdot 0 + Y \cdot \sin\phi + Z \cdot \cos\phi + 1 \cdot 0 = Y \cdot \sin\phi + Z \cdot \cos\phi = 1 \cdot 1 + 0 \cdot 0 = 1 \\ 1 &= X \cdot 0 + Y \cdot 0 + Z \cdot 0 + 1 \cdot 1 = 1 \end{aligned}$$

9. Mathematical Conventions

9.1 Left Handed versus Right Handed Coordinate Systems

I prefer the Left Handed system because I like the idea of objects being drawn in +Z and the viewpoint existing in –Z. DirectX / Direct3D uses a Left Hand system by default, while OpenGL uses a Right Hand system. The choice of reference system and direction of rotation is arbitrary, as long as you are consistent.

We've defined our rotations consistently in the Left Hand coordinate system as occurring counter clockwise when looking towards the origin from a point in the *negative* region of the axis about which the rotation occurs. This is equivalent to clockwise rotation in the same Left Hand coordinate system when viewed from the *positive* region of the rotational axis looking to the origin, which is how Direct3D defines rotations. [https://docs.microsoft.com/en-us/windows/win32/direct3d9/transforms#rotate]

You may run across rotation matrices in the literature with the signs (+ and -) swapped on the *sin* functions. One reason could be the authors are rotating clockwise in a Right Hand system when looking towards the origin from a point in the positive region of the axis about which the rotation occurs.

Additionally, other authors may use the same rotation matrices but modify the direction of rotation, or the handedness of the coordinate system, or the position along the rotational axis from which the direction of rotation is defined.

For example:

$$\left(X_{R_Z}, Y_{R_Z}, Z_{R_Z}, 1\right) = (X, Y, Z, 1) \cdot \begin{bmatrix} \cos\phi & \sin\phi & 0 & 0 \\ -\sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Defines:

(1) A CCW rotation in a LHS system when looking from (0,0,-Z) to the origin; +X to +Y
(2) A CW rotation in a LHS system when looking from (0,0,Z) to the origin; +X to +Y
(3) A CCW rotation in a RHS system when looking from (0,0,Z) to the origin; +X to +Y
(4) A CW rotation in a RHS system when looking from (0,0,-Z) to the origin; +X to +Y

While:

$$\left(X_{R_Z}, Y_{R_Z}, Z_{R_Z}, 1\right) = (X, Y, Z, 1) \cdot \begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Defines:

(1) A CCW rotation in a RHS system when looking from (0,0,-Z) to the origin; +Y to +X
(2) A CW rotation in a RHS system when looking from (0,0,Z) to the origin; +Y to +X
(3) A CCW rotation in a LHS system when looking from (0,0,Z) to the origin; +Y to +X
(4) A CW rotation in a LHS system when looking from (0,0,-Z) to the origin; +Y to +X

9.2 "Point times Matrix" versus "Matrix times Point" Notation

Just as one can set up a 3-D coordinate system that uses either a Left or Right hand convention, and define rotations as taking place either clockwise or counter clockwise; either a "point times matrix" or "matrix times point" convention can be used to express 2-D and 3-D transformations. Up to this point we've used "point times matrix". In this section we show the corresponding "matrix times point" notation.

9.2.1 3-D Translation

$$(X_T, Y_T, Z_T, 1) = (X, Y, Z, 1) \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_X & T_Y & T_Z & 1 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} X_T \\ Y_T \\ Z_T \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_X \\ 0 & 1 & 0 & T_Y \\ 0 & 0 & 1 & T_Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

9.2.2 3-D Scaling (matrices are identical)

$$(X_S, Y_S, Z_S, 1) = (X, Y, Z, 1) \cdot \begin{bmatrix} S_X & 0 & 0 & 0 \\ 0 & S_Y & 0 & 0 \\ 0 & 0 & S_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} X_S \\ Y_S \\ Z_S \\ 1 \end{bmatrix} = \begin{bmatrix} S_X & 0 & 0 & 0 \\ 0 & S_Y & 0 & 0 \\ 0 & 0 & S_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

9.2.3 Z-axis: Rotation of $(X, Y, Z, 1)$ by ϕ CCW looking "forward" from (0,0,-Z) to origin; +X to +Y

$$(X_{R_Z}, Y_{R_Z}, Z_{R_Z}, 1) = (X, Y, Z, 1) \cdot \begin{bmatrix} \cos\phi & \sin\phi & 0 & 0 \\ -\sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} X_{R_Z} \\ Y_{R_Z} \\ Z_{R_Z} \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

9.2.4 Y-axis: Rotation of $(X, Y, Z, 1)$ by ϕ CCW looking "up" from (0,-Y,0) to origin; +Z to +X

$$(X_{R_Y}, Y_{R_Y}, Z_{R_Y}, 1) = (X, Y, Z, 1) \cdot \begin{bmatrix} \cos\phi & 0 & -\sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} X_{R_Y} \\ Y_{R_Y} \\ Z_{R_Y} \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

9.2.5 X-axis: Rotation of $(X, Y, Z, 1)$ by ϕ CCW looking "sideways" from (-X,0,0) to origin; +Y to +Z

$$(X_{R_X}, Y_{R_X}, Z_{R_X}, 1) = (X, Y, Z, 1) \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & \sin\phi & 0 \\ 0 & -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} X_{R_X} \\ Y_{R_X} \\ Z_{R_X} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

The resulting equations are identical either way. Let's show this is true by examining the two forms of the Z axis rotation matrix multiplications as an example:

$$X_{R_Z} = X \cdot \cos\phi - Y \cdot \sin\phi + Z \cdot 0 + 1 \cdot 0 = \cos\phi \cdot X - \sin\phi \cdot Y + 0 \cdot Z + 0 \cdot 1 = X \cdot \cos\phi - Y \cdot \sin\phi$$
$$Y_{R_Z} = X \cdot \sin\phi + Y \cdot \cos\phi + Z \cdot 0 + 1 \cdot 0 = \sin\phi \cdot X + \cos\phi \cdot Y + 0 \cdot Z + 0 \cdot 1 = X \cdot \sin\phi + Y \cdot \cos\phi$$
$$Z_{R_Z} = X \cdot 0 + Y \cdot 0 + Z \cdot 1 + 1 \cdot 0 = 0 \cdot X + 0 \cdot Y + 1 \cdot Z + 0 \cdot 1 = Z$$
$$1 = X \cdot 0 + Y \cdot 0 + Z \cdot 0 + 1 \cdot 1 = 0 \cdot X + 0 \cdot Y + 1 \cdot Z + 0 \cdot 1 = 1$$

10. Why Matrix Notation?

As you learned while programming Assignment 1, implementing the transformations (translation, scaling, and rotation) in no way requires expressing the equations in matrix notation, nor implementing matrix multiplication operations. In fact, many students never bother to implement matrix notation in their code throughout this entire course.

Given these facts, why spend time covering matrix notation?

Expressing the transformations in matrix notation and implementing those transformations in code as matrices have two very important implications:

- Matrix notation allows for parallelism
- Composite matrices can be pre-computed which combine multiple transformations into a single matrix

We will cover each of these idea in this section. But, the critical idea is that BOTH concepts have the ability to dramatically speed up computations, allowing more complex 3-D models (more polygons) and animations (frames per second), to become practical.

Finally, a third reason for being familiar with matrix notations is that:

- Graphics libraries, such as Direct3D or OpenGL, use matrix notation to express transformations. You won't be able to effectively use these libraries without an understanding of matrix notation.

10.1 Matrix Notation Allows for Parallelism

Expressing all five 3-D transformations (translation, scaling, X-axis rotation, Y-axis rotation, and Z-axis rotation) as 4 by 4 matrices, allows special purpose hardware accelerators to be constructed (GPUs and TPUs) that perform matrix multiplications *in parallel*.

Without getting into hardware details, let's think about the different between implementing the rotation of a collection of vertices of an object using sequential code versus as matrices.

With sequential code, each vertex would be computed sequentially, one after another, using a loop that iterated over the points. Furthermore the rotation for each point would involve sequentially computing the rotated X value from the original X, and then the rotated Y values from the original Y, and finally the rotated Z values from the original Z. So, with 1,000 vertices and 3 equations per vertex (for X, Y, and Z) we are looking at 3,000 sequential equation computations.

Nothing however, except the sequential nature of von Neumann machines, requires that these operations take place sequentially. The rotation of one vertex is completely independent of the rotation of any other vertex. Thus, conceptually, ALL of the vertices can be rotated in parallel. Furthermore, when considering an individual point, the computations applied to X, Y, and Z are also completely independent – there is no reason not to carry these operations out in parallel.

Expressing all transformations in a fixed notation, a 4 by 4 array, allows the construction of blindingly fast special purpose processors that are optimized for throughput of matrix multiplication operations. If we limit our 3-D models to being composed of triangles (only) that too simplifies (standardizes) the processing of polygons.

10.2 Composite Matrices

Transformations can be concatenated by multiplying their matrices together in the order that the operations are to be performed.

Highly Efficient.

Consider an in-place rotation a 2-D object with 10,000 vertices around an arbitrary point.

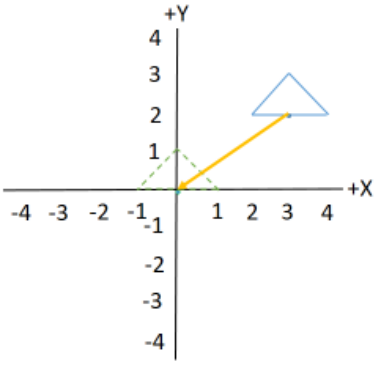
Without a composite matrix, a total of 20,000 translations and 10,000 rotations must be performed –

- 10,000 translations to move all the vertices of the object so that the "visual center" of that object is centered on the origin;
- 10,000 rotations, one for each vertex;
- 10,000 translation to move all of the rotated vertices of the object to return its "visual center" to its original location.

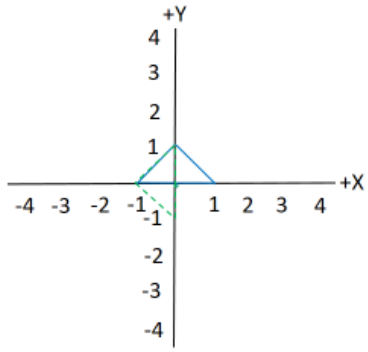
Instead, we can perform two matrix multiplications to construct a composite matrix; and then multiply each of the 10,000 vertices by that composite matrix. In other words, in this case we can accomplish the same goal (in-place 2-D rotation) with about 1/3 of the total effort that would be expended otherwise.

10.2.1 Composite Matrix Example: In-Place 2-D Rotation

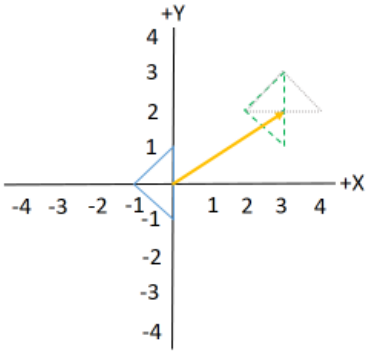
Earlier, we introduced in-place rotation using a triangle defined by three vertices (2,2), (3,3), 4,2) that we rotated around the point (3,2).



Points	Translated Points
(2, 2)	(-1, 0)
(3, 3)	(0, 1)
(4, 2)	(1, 0)



Points	Rotated Points
(-1, 0)	(0, -1)
(0, 1)	(-1, 0)
(1, 0)	(0, 1)



Points	Translated Points
(0, -1)	(3, 1)
(-1, 0)	(2, 2)
(0, 1)	(3, 3)

$X_a = X - 3 \quad Y_a = Y - 2$

$X_b = X_a \cdot cos\,(90°) - Y_a \cdot sin\,(90°) \quad X_c = X_b + 3 \quad Y_c = Y_b + 2$

$X_b = X_a \cdot \quad 0 \quad - Y_a \cdot \quad 1$

$X_b = \ -Y_a$

$Y_b = X_a \cdot \ sin\,(90°) + Y_a \ \cdot cos\,(90°)$

$Y_b = X_a \cdot \quad 1 \quad + Y_a \ \cdot \quad 0$

$Y_b = X_a$

$$T_{-Center} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & -2 & 1 \end{bmatrix} \qquad R = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad T_{Center} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 2 & 1 \end{bmatrix}$$

$$X_a = X - 3 \qquad X_b = -Y_a \qquad X_c = X_b + 3$$
$$Y_a = Y - 2 \qquad Y_b = X_a \qquad Y_c = Y_b + 2$$

Step 1 : Compute Composite Matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & -2 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 2 & -3 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 5 & -1 & 1 \end{bmatrix}$$

$$[T_{-Center}] \cdot [R] = [T_{-Center} \cdot R] \cdot [T_{Center}] = [T_{Center} \cdot R \cdot T_{Center}]$$

Step 2 : Multiply Vertices by Composite Matrix

$$(2,2,1) \cdot \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 5 & -1 & 1 \end{bmatrix} = (3,1,1)$$

$$(3,3,1) \cdot \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 5 & -1 & 1 \end{bmatrix} = (2,2,1)$$

$$(4,2,1) \cdot \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 5 & -1 & 1 \end{bmatrix} = (3,3,1)$$

Points	Translated Points
(2, 2, 1)	(3, 1, 1)
(3, 3, 1)	(2, 2, 1)
(4, 2, 1)	(3, 3, 1)

10.2.2 Composite Matrix Example: Rotation about an Arbitrary Line in 3-D Space

Given a line segment in 3-D space defined by two points $P_1 = (X_1, Y_1, Z_1)$ and $P_2 = (X_2, Y_2, Z_2)$, we will define a composite matrix capable of rotating an object (a collection of vertices) CCW through some specified angle ϕ with respect to the reference line when viewed from P_1 looking towards P_2.

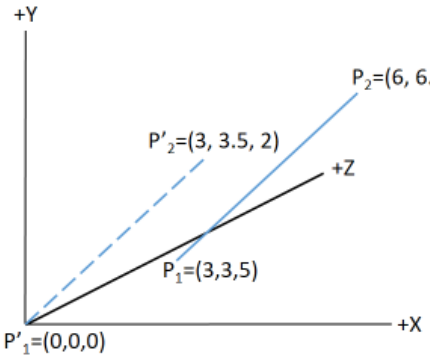
Rotation about an arbitrary line in 3-D space is a seven step process:

1. Translate the reference line segment so that P_1 will be at the origin.
2. Rotate the reference line into the X-Z plane. This can be accomplished by performing an X-axis rotation.
3. Rotate the reference line within the X-Z plane until it lines up with the Z-axis. This can be accomplished by performing a Y-axis rotation.
4. Rotate the object's vertices CCW through ϕ [when viewed from $(0,0,-Z)$ in our Left Hand coordinate system].
5. Reverse the effect of Step 3 by performing the inverse Y-axis rotation.
6. Reverse the effect of Step 2 by performing the inverse X-axis rotation.
7. Reverse the effect of Step 1 by performing the inverse translation.

In reality, of course, we will combine all of these operations into a single composite matrix to perform all of these operations simultaneously.

As we develop matrices to carry out the seven individual steps (before multiplying them to create a composite matrix), we will verify the accuracy of the matrices with an example reference line. Our example reference line will be defined by $P_1 = (3, 3, 5)$ and $P_2 = (6, 6.4641, 7)$. As you will see, the odd Y value for P_2 is necessary to make (most of) the math work out easily – if that value is bothering you, think of it as 6.5; which we will often use as a shorthand for the actual value.

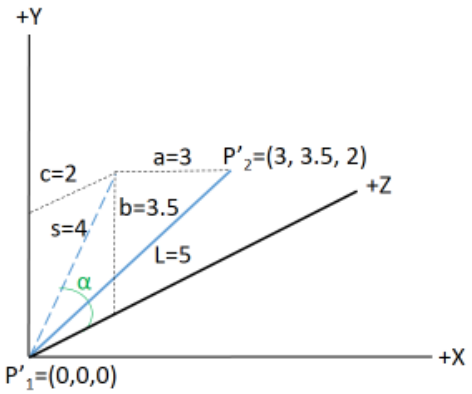
Step 1: Translate P_1 to origin

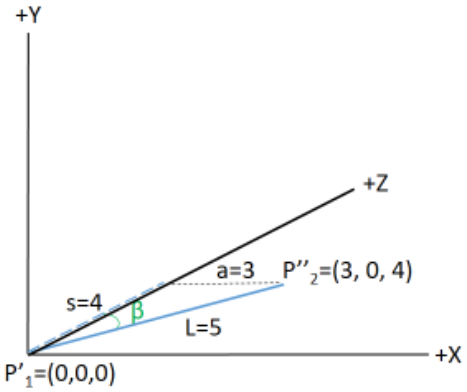


$$T_{-P_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -P_{1_X} & -P_{1_Y} & -P_{1_Z} & 1 \end{bmatrix} \quad T_{-P_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & -3 & -5 & 1 \end{bmatrix}$$

Points	Translated Points
(3, 3, 5)	(0, 0, 0)
(6, 6.5, 7)	(3, 3.5, 2)

Step 2: Rotate Reference Line into the X-Z plane





Let S be the "shadow" of the line segment from P'_1 to P'_2 on the Y-Z plane. If we rotate S about the X-axis through the angle α until S lines up with the Z-axis, then the P'_1 to P'_2 line segment (now P'_1 to P''_2) will be in the X-Z plane.

$a = P'_{2_X} = (P_{2_X} - P_{1_X}) = 6 - 3 = 3$
$b = P'_{2_Y} = (P_{2_Y} - P_{1_Y}) = 6.5 - 3 = 3.5$ (or, more accurately, 6.4641 – 3 = 3.4641)
$c = P'_{2_Z} = (P_{2_Z} - P_{1_Z}) = 7 - 5 = 2$

Since $c^2 + b^2 = s^2$, $s = \sqrt{c^2 + b^2}$ and thus $s = \sqrt{2^2 + 3.4641^2} = \sqrt{4 + 12} = \sqrt{16} = 4$

Likewise $a^2 + s^2 = L^2$, $L = \sqrt{a^2 + s^2}$ and thus $L = \sqrt{3^2 + 4^2} = \sqrt{9 + 16} = \sqrt{25} = 5$

We don't need to know the actual angle α, only its sine and cosine.

$$\sin\alpha = \frac{opp}{hyp} = \frac{b}{s} = \frac{3.4641}{4} = 0.866 \qquad \cos\alpha = \frac{adj}{hyp} = \frac{c}{s} = \frac{2}{4} = 0.5$$

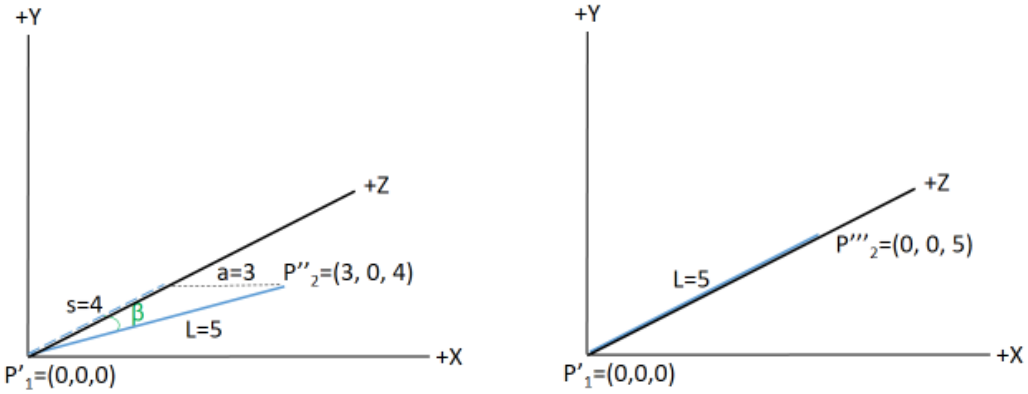
$$R_X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{c}{s} & \frac{b}{s} & 0 \\ 0 & -\frac{b}{s} & \frac{c}{s} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R_X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.5 & 0.866 & 0 \\ 0 & -0.866 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Points	Transformed Points
(0, 0, 0)	(0, 0, 0)
(3, 3.5, 2)	(3, 0, 4)

Here are the details of the P'_2 to P''_2 computation:

$$\begin{aligned} (P''_{2X}, P''_{2Y}, P''_{2Z}) &= \left((P'_{2X}), (P'_{2Y} \cdot 0.5 - P'_{2Z} \cdot 0.866), (P'_{2Y} \cdot 0.866 + P'_{2Z} \cdot 0.5)\right) \\ &= \left(3, (3.4641 \cdot 0.5 - 2 \cdot 0.866), (3.4641 \cdot 0.866 + 2 \cdot 0.5)\right) \\ &= \left(3, (1.732 - 1.732), (3 + 1)\right) = (3,0,4) \end{aligned}$$

Step 3: Rotate Reference Line within the X-Z Plane so that it coincides with the Z-axis



The reference line now lies in the X-Z plane. Next, it must be rotated through some angle β in order to coincide with the Z-axis. As before, we don't need to know that actual angle β, just its sine and cosine.

$$\sin\beta = \frac{opp}{hyp} = \frac{a}{L} = \frac{3}{5} = 0.6 \qquad \cos\beta = \frac{adj}{hyp} = \frac{S}{L} = \frac{4}{5} = 0.8$$

$$R_Y = \begin{bmatrix} \frac{a}{L} & 0 & \frac{S}{L} & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{S}{L} & 0 & \frac{a}{L} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R_Y = \begin{bmatrix} 0.8 & 0 & 0.6 & 0 \\ 0 & 1 & 0 & 0 \\ -0.6 & 0 & 0.8 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Points	Transformed Points
(0, 0, 0)	(0, 0, 0)
(3, 0, 4)	(0, 0, 5)

Note that compared to our standard Y-axis rotation matrix, I swapped the signs of the sine functions. This is necessary as we are rotating from +X to +Z while our standard Y-axis rotation matrix assumes a rotation from +Z to +X.

Here are the details of the P''_2 to P'''_2 computation:

$$\begin{aligned} (P'''_{2X}, P'''_{2Y}, P'''_{2Z}) &= \left((P''_{2X} \cdot 0.8 - P''_{2Z} \cdot 0.6), (P''_{2Y}), (P''_{2X} \cdot 0.6 + P''_{2Z} \cdot 0.8)\right) \\ &= \left((3 \cdot 0.8 - 4 \cdot 0.6), 0, (3 \cdot 0.6 + 4 \cdot 0.8)\right) \\ &= \left((2.4 - 2.4), 0, (1.8 + 3.2)\right) = (0,0,5) \end{aligned}$$

Step 4: Rotate the object's vertices CCW through ϕ

Perform the Z-axis rotations of our Object's vertices through the angle ϕ using our standard Z-axis rotation matrix.

$$R_Z = \begin{bmatrix} \cos\phi & \sin\phi & 0 & 0 \\ -\sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Step 5: Reverse the Effect of Step 3 by Performing the Inverse Y-axis Rotation

Reverse the effect of Step 3. This can be accomplished by moving the minus sign from the lower left sine operation to the upper right sine operation.

$$R_{-Y} = \begin{bmatrix} \frac{a}{L} & 0 & -\frac{S}{L} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{S}{L} & 0 & \frac{a}{L} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R_{-Y} = \begin{bmatrix} 0.8 & 0 & -0.6 & 0 \\ 0 & 1 & 0 & 0 \\ 0.6 & 0 & 0.8 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Points	Transformed Points
(0, 0, 0)	(0, 0, 0)
(0, 0, 5)	(3, 0, 4)

Details of the P'''_2 to P''_2 computation:

$$\begin{aligned} (P''_{2X}, P''_{2Y}, P''_{2Z}) &= \left((P'''_{2X} \cdot 0.8 + P'''_{2Z} \cdot 0.6), (P'''_{2Y}), (P'''_{2X} \cdot -0.6 + P'''_{2Z} \cdot 0.8)\right) \\ &= \left((0 \cdot 0.8 + 5 \cdot 0.6), 0, (0 \cdot -0.6 + 5 \cdot 0.8)\right) \\ &= \left((0 + 3), 0, (0 + 4.0)\right) = (3,0,4) \end{aligned}$$

Step 6: Reverse the Effect of Step 2 by Performing the Inverse X-axis Rotation

Similarly, we can reverse the effects of the X-axis rotation of Step by once again swapping the signs of the sine functions.

$$R_{-X} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{c}{s} & -\frac{b}{s} & 0 \\ 0 & \frac{b}{s} & \frac{c}{s} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R_{-X} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.5 & -0.866 & 0 \\ 0 & 0.866 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Points	Transformed Points
(0, 0, 0)	(0, 0, 0)
(3, 0, 4)	(3, 3.5, 2)

Here are the details of the P''_2 to P'_2 computation:

$$\begin{aligned} (P'_{2X}, P'_{2Y}, P'_{2Z}) &= \left((P''_{2X}), (P''_{2Y} \cdot 0.5 + P''_{2Z} \cdot 0.866), (P''_{2Y} \cdot -0.866 + P''_{2Z} \cdot 0.5)\right) \\ &= \left(3, (0 \cdot 0.5 + 4 \cdot 0.866), (0 \cdot -0.866 + 4 \cdot 0.5)\right) \\ &= \left(3, (0 + 3.464), (0 + 2)\right) = (3, 3.464, 2) \end{aligned}$$

Step 7: Reverse the Effect of Step 1 by Performing the Inverse Translation

Finally, we can translate the reference line back to its original position in space.

$$T_{+P_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ P_{1X} & P_{1Y} & P_{1Z} & 1 \end{bmatrix} \qquad T_{+P_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 3 & 3 & 5 & 1 \end{bmatrix}$$

Points	Translated Points
(0, 0, 0)	(3, 3, 5)
(3, 3.5, 2)	(6, 6.5, 7)

Now that we have defined all seven matrices we can multiply them together, in the order they would have been applied, to construct a single composite matrix.

$$R_{Composite} = T_{-P_1} \cdot R_X \cdot R_Y \cdot R_Z \cdot R_{-Y} \cdot R_{-X} \cdot T_{+P_1}$$

The vertices of our object would then be multiplied by this single composite matrix to perform an in-place rotation around the reference line.

Using Composite matrices can dramatically increase the efficiency of performing complex operations like rotations around arbitrary lines. This is true even without a GPU.

For example, given a 1,000 vertex object to be rotated about an arbitrary reference line, after computing the composite matrix, we need perform only 1,000 point * matrix calculations, instead of 7,000 point * matrix calculations – a seven fold improvement in efficiency.

With a GPU we can do much better – given enough hardware, all 1,000 point * matrix calculations could take place in parallel in a single step. Of course, this assumes we have already computed the composite matrix, which is a seven step process, yes? Well, actually, using the associative property and performing matrix multiplications in parallel, we could shorten the process of creating the composite matrix from 7 to 3 steps.

Step 1: Three parallel matrix multiplications: $R_{Composite} = \left(T_{-P_1} \cdot R_X\right) \cdot \left(R_Y \cdot R_Z\right) \cdot \left(R_{-Y} \cdot R_{-X}\right) \cdot T_{+P_1}$

Step 2: Two parallel matrix multiplications: $R_{Composite} = \left(\left(T_{-P_1} \cdot R_X\right) \cdot \left(R_Y \cdot R_Z\right)\right) \cdot \left(\left(R_{-Y} \cdot R_{-X}\right) \cdot T_{+P_1}\right)$

Step 3: A single matrix multiplication: $R_{Composite} = \left(\left(T_{-P_1} \cdot R_X\right) \cdot \left(R_Y \cdot R_Z\right)\right) \cdot \left(\left(R_{-Y} \cdot R_{-X}\right) \cdot T_{+P_1}\right)$

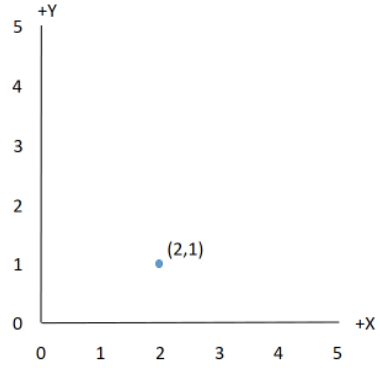
Thus, with enough hardware, we could **reduce this problem from requiring 7,000 time steps to only 4 time steps** -- three time steps to compute the composite matrix and one additional time step to compute the point * matrix calculations.

This is why GPUs / TPUs have revolutionized computer graphics and related fields like machine learning where the underlying operations can be expressed in matrix notation.

11. Background: Linear Algebra

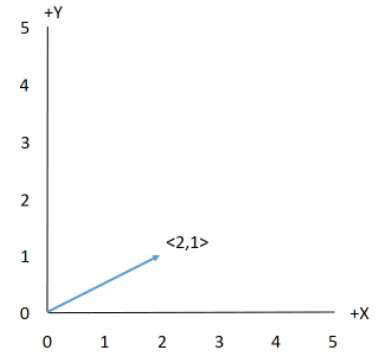
11.1 Points and Vectors

We are familiar with representing points in 2-D space as an ordered pair (X,Y) or (P_X, P_Y) and in 3-D space using an ordered triple (X,Y,Z) or (P_X, P_Y, P_Z). Here is the point (2,1).



A 2-D point $P = (P_X, P_Y) = (2,1)$

Vectors are n-tuples of real numbers that can be used to represent spatial directions. They are defined by specifying an endpoint, with the starting point assumed to be the origin. While vectors may exist in N dimensions, where N > 1, we will focus on 2-D vectors which can be represented by <X,Y> or $\langle V_X, V_Y \rangle$ and 3-D vectors which can be represented by <X,Y,Z> or $\langle V_X, V_Y, V_Z \rangle$. To distinguish vectors from points, and vice versa, we often use angle brackets <X,Y> for vectors and parentheses (X,Y) for points. Vectors have both length and direction.



A 2-D Vector $V = \langle V_X, V_Y \rangle = \langle 2,1 \rangle$

When viewed in this way, the specified coordinates provide the endpoint of a line segment where the origin of the coordinate system specifies the other (starting) point.

11.2 Matrix Representation

Vectors may be represented by n by 1 matrices in addition to n-tuples

$$V = \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix}$$

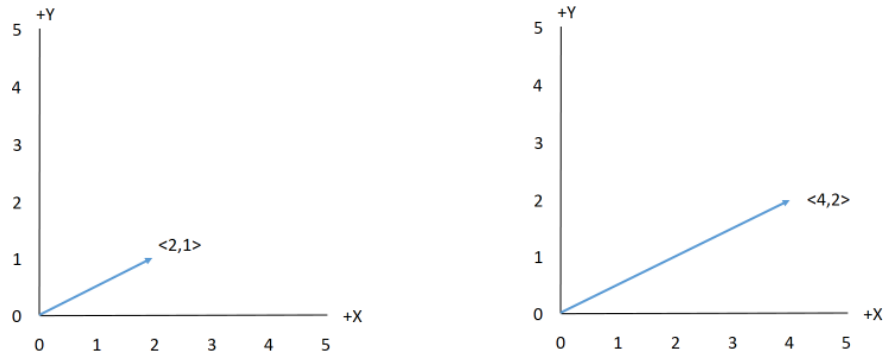
It is often useful to be able to also represent the vector as a 1 by n matrix, instead of an n by 1 matrix. In this case we talk about the "transpose" of a vector, V, written as V^T.

$$V^T = \begin{bmatrix} V_X & V_Y & V_Z \end{bmatrix}$$

11.3 Multiplying a Scalar Quantity by a Vector

A scalar quantity, a, can be multiplied by a vector, V. Doing so changes the magnitude (length) of the vector, but not its direction.

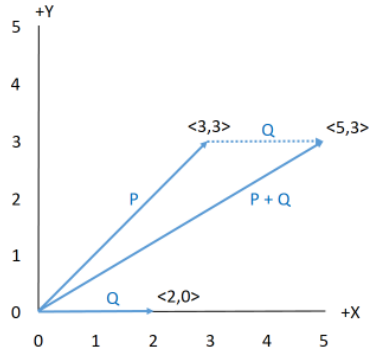
$$aV = < aV_1, aV_2, aV_3, \dots, aV_n >$$



Multiplication of a Scalar Times a Vector: $2 \cdot \langle 2,1 \rangle = \langle 4,2 \rangle$

11.4 Vectors Add and Subtract Component-wise

$$P + Q = < (P_1 + Q_1), (P_2 + Q_2), (P_3 + Q_3), \dots, (P_n + Q_n) >$$



Example: $P = < 3,3 >$ $Q = < 2,0 >$ $P + Q = < 5,3 >$

11.5 Associative, Commutative, and Distributive Properties of Vectors

Below, **a** and **b** are scalars; **P**, **Q**, and **R** are vectors**.**

(a) $P + Q = Q + P$ [Commutative Property of Vector Addition]
(b) $(P + Q) + R = P + (Q + R)$ [Associative Property of Vector Addition]
(c) $(a\ b)P = a\ (b\ P)$ [Associative Property of Scalar Vector Multiplication]
(d) $a\ (P + Q) = aP + aQ$ [Distributive Property: Scalar times Vector Addition]
(e) $(a + b)P = aP + bP$ [Distributive Property: Scalar Addition times Vector]

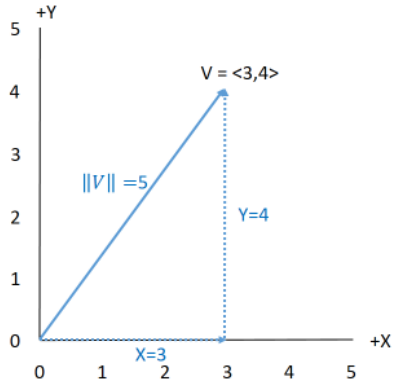
11.6 Computing the Magnitude of a Vector

The magnitude of an n-dimensional vector, V, is denoted by the scalar $\|V\|$, where $\|V\| = \sqrt{\sum_{i=0}^{n} V_i^2}$

The magnitude of a vector is often referred to as its length.

For 2-D vectors $\|V\| = \sqrt{V_X^2 + V_Y^2}$

This equation is based on the Pythagorean Theorem you learned back in middle / high school. For a right triangle $X^2 + Y^2 = r^2$ and basic algebra gives $r = \sqrt{X^2 + Y^2}$

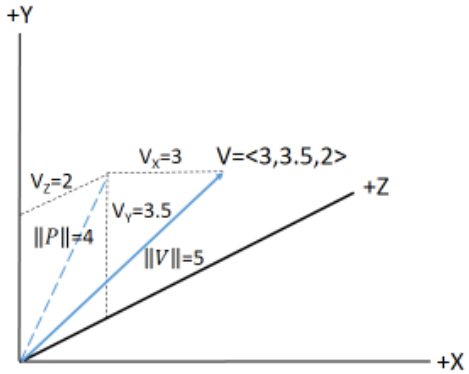


So, given vector V = <3, 4> its length is: $\|V\| = \sqrt{3^2 + 4^2} = \sqrt{9 + 16} = \sqrt{25} = 5$

For 3-D vectors $\|V\| = \sqrt{V_X^2 + V_Y^2 + V_Z^2}$

The 3-D version of the magnitude computation employs two separate right triangles.

The first right triangle utilizes the V_Y and V_Z components of the vector to compute the length of an orthographic projection, P, of the vector, V, onto the Y-Z plane. $\|P\| = \sqrt{V_Y^2 + V_Z^2}$



The second right triangle uses V_X and $\|P\|$ as its height and base; with V being the hypotenuse. Thus,

$$\|V\| = \sqrt{V_X^2 + (\|P\|)^2} = \sqrt{V_X^2 + \left(\sqrt{V_Y^2 + V_Z^2}\right)^2} = \sqrt{V_X^2 + V_Y^2 + V_Z^2}$$

So, given vector V = <3, 3.46410162, 2> its length is:

$$\|V\| = \sqrt{3^2 + 3.46410162^2 + 2^2} = \sqrt{9 + 12 + 4} = \sqrt{25} = 5$$

11.7 Unit Vectors

A Unit Vector is a vector with magnitude equal to 1.

Vectors can be normalized into unit vectors by dividing each component of the vector by the vector's magnitude.

For 2-D Vectors:

$$V = < V_X, V_Y > \quad V_{norm} = < \frac{V_X}{\|V\|}, \frac{V_Y}{\|V\|} > \quad V_{norm} = < \frac{V_X}{\sqrt{V_X^2+V_Y^2}}, \frac{V_Y}{\sqrt{V_X^2+V_Y^2}} >$$

Example: 2-D Vector: $V = < 3, 4 > \quad V_{norm} = < 0.6, 0.8 >$

$$V_{norm} = < \frac{3}{\sqrt{3^2+4^2}}, \frac{4}{\sqrt{3^2+4^2}} > = < \frac{3}{\sqrt{9+16}}, \frac{4}{\sqrt{9+16}} > = < \frac{3}{\sqrt{25}}, \frac{4}{\sqrt{25}} > = < \frac{3}{5}, \frac{4}{5} > = < 0.6, 0.8 >$$

For 3-D Vectors:

$$V = < V_X, V_Y, V_Z > \quad V_{norm} = < \frac{V_X}{\|V\|}, \frac{V_Y}{\|V\|}, \frac{V_Z}{\|V\|} > \quad V_{norm} = < \frac{V_X}{\sqrt{V_X^2+V_Y^2+V_Z^2}}, \frac{V_Y}{\sqrt{V_X^2+V_Y^2+V_Z^2}}, \frac{V_Z}{\sqrt{V_X^2+V_Y^2+V_Z^2}} >$$

Example: 3-D Vector: $V = < 3, 3.46410162, 2 > \quad V_{norm} = < 0.6, 0.69282032, 0.4 >$

$$\|V\| = \sqrt{3^2 + 3.46410162^2 + 2^2} = \sqrt{9+12+4} = \sqrt{25} = 5$$

$$V_{norm} = \langle \frac{3}{5}, \frac{3.46410162}{5}, \frac{2}{5} \rangle = \langle 0.6, 0.69282032, 0.4 \rangle$$

11.8 Basic Properties of Vector Magnitudes

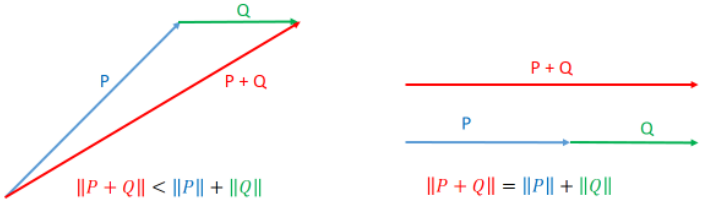
(a) $\|P\| \geq 0$ All vectors have a positive (or zero) length.

(b) $\|P\| = 0$ *if and only if* $P = < 0, 0, 0, \ldots, 0 >$ The start and end points of the vector are the origin, thus the vector has no (zero) length.

(c) $\|a\, P\| = |a|\|P\|$ If you multiple a scalar by a vector and then take the magnitude of the resulting vector, you will get the same value that you would have gotten by multiplying the absolute value of the scalar times the magnitude of the original vector. For example:

$$\|2 \cdot < 3,4 >\| = \|< 6,8 >\| = \sqrt{36+64} = \sqrt{100} = 10$$
$$|2| \cdot \|< 3,4 >\| = 2 \cdot \sqrt{9+16} = 2 \cdot \sqrt{25} = 2 \cdot 5 = 10$$

(d) $\|P + Q\| \leq \|P\| + \|Q\|$ The length of one side of a triangle cannot be longer than the sum of the lengths of the other two sides. (Shortest distance between two points is a straight line.)



11.9 Dot Product

The dot product of two vectors produces a **scalar** value that provides a measure of the difference between the directions in which the two vectors point.

11.9.1 Computing the Dot Product of Two vectors

Dot products are computed as follows:

$$P \cdot Q = \sum_{i=1}^{n} P_i Q_i \quad \text{or} \quad P \cdot Q = P^T Q$$

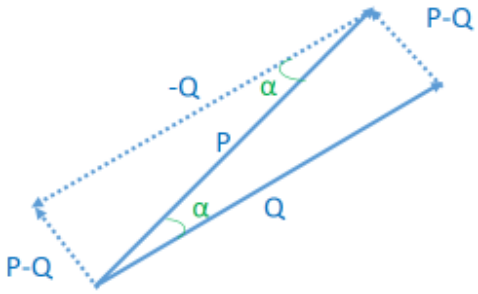
If P & Q are 2-D $P \cdot Q = P_X \cdot Q_X + P_Y \cdot Q_Y$ or in matrix form $[P_X \quad P_Y] \cdot \begin{bmatrix} Q_X \\ Q_Y \end{bmatrix}$

If P & Q are 3-D $P \cdot Q = P_X \cdot Q_X + P_Y \cdot Q_Y + P_Z \cdot Q_Z$ or in matrix form $[P_X \quad P_Y \quad P_Z] \cdot \begin{bmatrix} Q_X \\ Q_Y \\ Q_Z \end{bmatrix}$

The Dot Product of two vectors may also be defined in terms of the lengths of the vectors and the cosine of the angle between them. Thus, given vectors P and Q the following equation holds true:

$$P \cdot Q = \|P\|\|Q\| \cos \alpha$$

Where $\boldsymbol{\alpha}$ is the planar angle between vectors.



This result can be proven using the Law of Cosines. [If you would like to reacquaint yourself with the proof, consult online sources such as https://proofwiki.org/wiki/Cosine_Formula_for_Dot_Product].

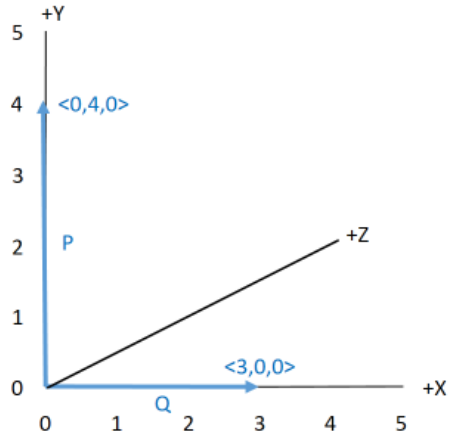
11.9.2 The Importance of the Sign of the Dot Product

Vectors P and Q are normal to each other (i.e., separated by 90°) if and only if $P \cdot Q = 0$

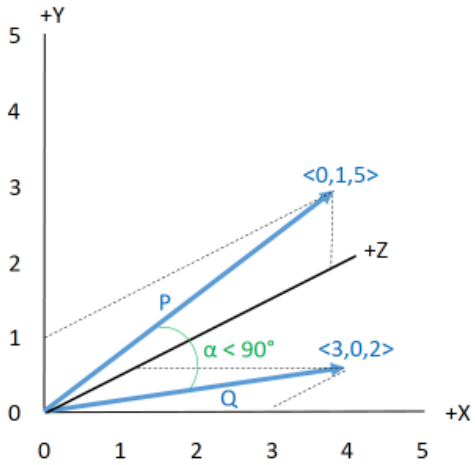
Example:

$P = \langle 0,4,0 \rangle \quad Q = \langle 3,0,0 \rangle$
$P \cdot Q = P_X \cdot Q_X + P_Y \cdot Q_Y + P_Z \cdot Q_Z = 0 \cdot 3 + 4 \cdot 0 + 0 \cdot 0 = 0$

In this case we have two vectors that align with the vertices of our coordinate system. $P = \langle 0,4,0 \rangle$ aligns with the Y-axis and $Q = \langle 3,0,0 \rangle$ aligns with the X-axis. Thus, these vectors must be at right angles to one another (i.e., separated by 90°) and $P \cdot Q = 0$



Vectors P and Q are separated by < 90° if and only if $P \cdot Q > 0$



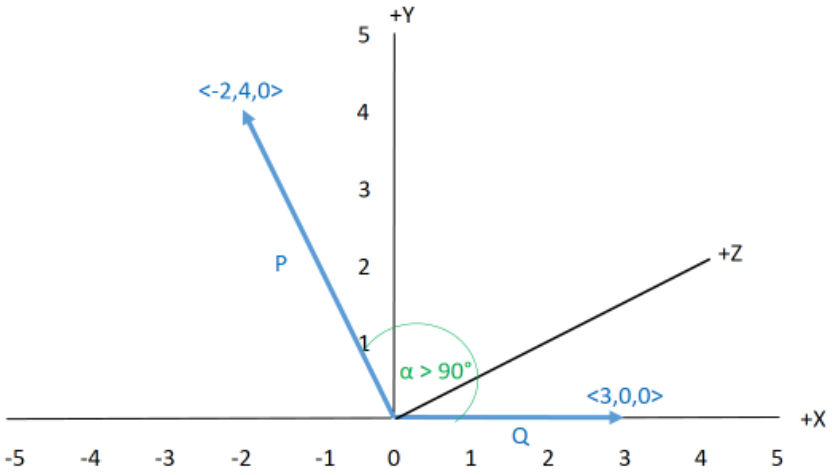
Example:

$P = \langle 0,1,5 \rangle \;\; Q = \langle 3,0,2 \rangle$
$P \cdot Q = P_X \cdot Q_X + P_Y \cdot Q_Y + P_Z \cdot Q_Z = 0 \cdot 3 + 1 \cdot 0 + 5 \cdot 2 = 10$

Vectors P and Q are separated by > 90° if and only if $P \cdot Q < 0$

Example:

$P = \langle -2,4,0 \rangle \;\; Q = \langle 3,0,0 \rangle$
$P \cdot Q = P_X \cdot Q_X + P_Y \cdot Q_Y + P_Z \cdot Q_Z = -2 \cdot 3 + 4 \cdot 0 + 0 \cdot 0 = -6$



11.10 Cross Product

The cross product of two 3-D vectors returns a new **vector** that is **perpendicular to the original vectors**.

$$P \times Q = \langle P_Y \cdot Q_Z - P_Z \cdot Q_Y, \quad P_Z \cdot Q_X - P_X \cdot Q_Z, \quad P_X \cdot Q_Y - P_Y \cdot Q_X \rangle$$

or

$$P \times Q = \begin{vmatrix} [1,0,0] & [0,1,0] & [0,0,1] \\ P_X & P_Y & P_Z \\ Q_X & Q_Y & Q_Z \end{vmatrix}$$

$$P \times Q = \langle +1 \cdot (P_Y \cdot Q_Z - P_Z \cdot Q_Y), -1 \cdot (P_X \cdot Q_Z - P_Z \cdot Q_X), +1 \cdot (P_X \cdot Q_Y - P_Y \cdot Q_X) \rangle$$

$$P \times Q = \langle P_Y \cdot Q_Z - P_Z \cdot Q_Y, \quad P_Z \cdot Q_X - P_X \cdot Q_Z, \quad P_X \cdot Q_Y - P_Y \cdot Q_X \rangle$$

Example:

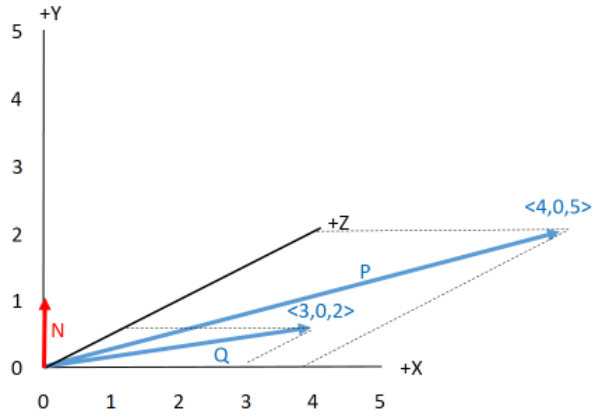
$P = \langle 4, 0, 5\rangle \quad Q = \langle 3, 0, 2\rangle$
$P \times Q = \langle P_Y \cdot Q_Z - P_Z \cdot Q_Y, \quad P_Z \cdot Q_X - P_X \cdot Q_Z, \quad P_X \cdot Q_Y - P_Y \cdot Q_X\rangle$
$P \times Q = \langle 0 \cdot 2 - 5 \cdot 0, \quad 5 \cdot 3 - 4 \cdot 2, \quad 4 \cdot 0 - 0 \cdot 3\rangle = \langle 0,7,0\rangle$

Since these two vectors lie in the X-Z plane (because $Y = 0$ on both vectors), the normal vector to the plane containing these vectors is parallel to the Y-axis.

We can normalize this vector as follows:

$$\|P \times Q\| = \sqrt{0^2 + 7^2 + 0^2} = 7$$

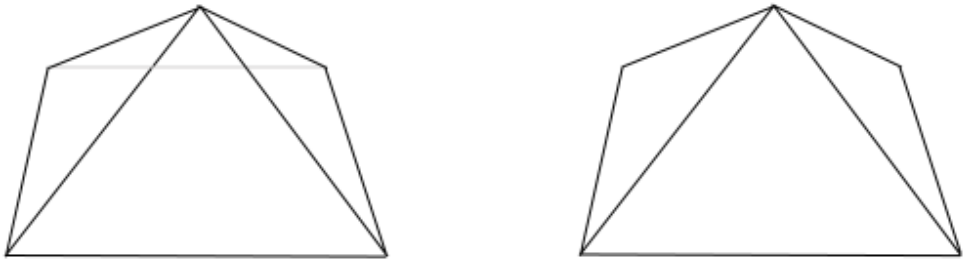
$$(P \times Q)_{norm} = \langle \tfrac{0}{7}, \tfrac{7}{7}, \tfrac{0}{7}\rangle = \langle 0,\ 1,\ 0\rangle$$



12. Back face Culling

Up until this point, the course has focused on so-called “wireframe graphics”. Objects are drawn by rendering the edges of the individual polygons making up those objects. All polygons are drawn, whether or not they represent the front, sides, or back of an object – as defined from the viewer’s orientation and location in 3-D space.

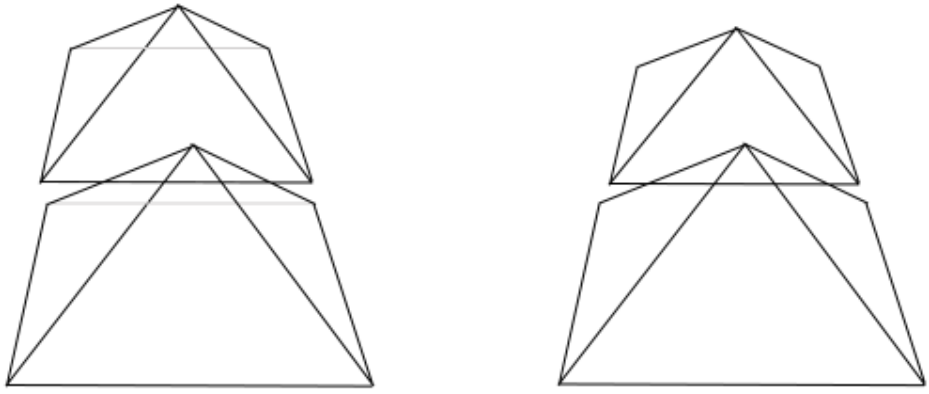
In this section we will examine a technique for determining whether a polygon faces the viewer or faces away from the viewer. Polygons that face away from the viewer will not be drawn. For a single convex object, back face removal will make the object appear solid.



Without Back Face Culling With Back Face Culling

Looking toward and slightly down at a Pyramid-Shaped Object

However, when multiple objects are present, or the objects are (partially) concave, back face removal is not a general solution to the hidden surface removal problem.



Without Back Face Culling With Back Face Culling

Looking toward and slightly down at two Pyramid-Shaped Objects

Despite its limitations, back face removal (or “culling” as it is often called) is frequently the “first step” in more general hidden surface removal algorithms.

In order to implement back face culling, we begin by defining the orientation and position of the polygons making up our object(s).

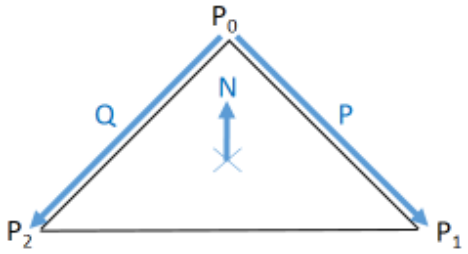
12.1 Establishing a Polygon's Orientation in 3-D Space

As we saw in Section 11.10, given two vectors, the Cross Product of those two vectors will define a "normal" (a vector at 90°) to the orientation of the 2-D plane containing those vectors. This will always be true as long as the two vectors do not point in exactly the same (or exactly the opposite) direction.

So, our first question is "where do these vectors come from?"

In general polygons are defined in 3-D space using a sequence of points (vertices), not vectors. Objects, in turn, can be defined by collections of polygons (often just triangles). Such a collection of polygons is referred to as a "polygon mesh".

Given three non-collinear points on a polygon (the vertices of a triangle will do nicely), we can compute two in-plane vectors, P and Q.



P is the vector from P_0 to P_1 which is computed as $P_1 - P_0 = \langle P_{1_X} - P_{0_x},\ P_{1_Y} - P_{0_Y},\ P_{1_Z} - P_{0_Z} \rangle$
Q is the vector from P_0 to P_2 which is computed as $P_2 - P_0 = \langle P_{2_X} - P_{0_x},\ P_{2_Y} - P_{0_Y},\ P_{2_Z} - P_{0_Z} \rangle$

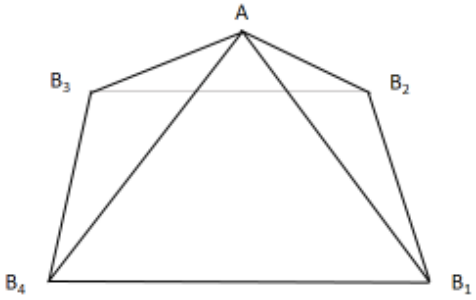
N is the surface normal of the polygon, which can be computed from the Cross Product $N = P \times Q$.

$$N = P \times Q = \langle P_Y \cdot Q_Z - P_Z \cdot Q_Y,\quad P_Z \cdot Q_X - P_X \cdot Q_Z,\quad P_X \cdot Q_Y - P_Y \cdot Q_X \rangle = \langle N_X,\ N_Y,\ N_Z \rangle$$

The surface normal, N, is generally normalized to a unit vector as follows: $N_{norm} = \frac{P \times Q}{\|P \times Q\|}$

$$N_{norm} = \langle \frac{N_X}{\sqrt{N_X^2 + N_Y^2 + N_Z^2}}, \frac{N_Y}{\sqrt{N_X^2 + N_Y^2 + N_Z^2}}, \frac{N_Z}{\sqrt{N_X^2 + N_Y^2 + N_Z^2}} \rangle$$

To see how all this works, let's turn our attention to a version of the pyramid we defined for the initial programming assignment.



The pyramid object consists of a mesh of five polygons: Front, Back, Left, Right, and Base.

Front: A, B1, B4 Back: A, B3, B2 Left: A, B4, B3 Right: A, B2, B1 Base: B1, B2, B3, B4

$A = (0, -100, 100)$ $B1 = (50, -150,\ 50)$ $B2 = (50, -150, 150)$ $B3 = (-50, -150, 150)$ $B4 = (-50, -150,\ 50)$

This version of the pyramid is shorter than the original, and it has been translated downward, so that we are looking at it from a position somewhat above.

I'll do the calculations to compute the surface normal for the Front polygon by hand, and then present a summary of results for the other four polygons.

$$A = (\mathbf{0}, -\mathbf{100}, \ \mathbf{100}) \qquad B_1 = (\mathbf{50}, \ -\mathbf{150}, \ \mathbf{50}) \qquad B_4 = (-\mathbf{50}, \ -\mathbf{150}, \ \mathbf{50})$$

$$P = B_1 - A = \langle B_{1_X} - A_X, B_{1_Y} - A_Y, B_{1_Z} - A_Z \rangle$$
$$= \langle 50 - 0, \ (-150) - (-100), \ 50 - 100 \rangle = \langle \mathbf{50}, -\mathbf{50}, -\mathbf{50} \rangle$$

$$Q = B_4 - A = \langle B_{4_X} - A_X, B_{4_Y} - A_Y, B_{4_Z} - A_Z \rangle$$
$$= \langle (-50) - \ 0, \ (-150) - (-100), \ 50 - 100 \rangle = \langle -\mathbf{50}, -\mathbf{50}, -\mathbf{50} \rangle$$

$$N = P \times Q = \langle P_Y \cdot Q_Z - P_Z \cdot Q_Y, \quad P_Z \cdot Q_X - P_X \cdot Q_Z, \quad P_X \cdot Q_Y - P_Y \cdot Q_X \rangle$$
$$= \langle (-50) \cdot (-50) - (-50) \cdot (-50), \quad (-50) \cdot (-50) - 50 \cdot (-50), \quad 50 \cdot (-50) - (-50) \cdot (-50) \rangle$$
$$= \langle 2500 - 2500, \ \ 2500 - (-2500), \ \ (-2500) - 2500 \rangle = \langle \mathbf{0}, \ \ \mathbf{5000}, -\mathbf{5000} \rangle$$

$$N_{norm} = \langle \frac{N_X}{\sqrt{N_X^2 + N_Y^2 + N_Z^2}}, \frac{N_Y}{\sqrt{N_X^2 + N_Y^2 + N_Z^2}}, \frac{N_Z}{\sqrt{N_X^2 + N_Y^2 + N_Z^2}} \rangle$$

$$N_{norm} = \langle \frac{0}{\sqrt{0^2 + 5000^2 + (-5000^2)}}, \frac{5000}{\sqrt{0^2 + 5000^2 + (-5000^2)}}, \frac{-5000}{\sqrt{0^2 + 5000^2 + (-5000^2)}} \rangle$$

$$N_{norm} = \langle \frac{0}{\sqrt{25{,}000{,}000 + 25{,}000{,}000}}, \frac{5000}{\sqrt{25{,}000{,}000 + 25{,}000{,}000}}, \frac{-5000}{\sqrt{25{,}000{,}000 + 25{,}000{,}000}} \rangle$$

$$N_{norm} = \langle \frac{0}{\sqrt{50{,}000{,}000}}, \frac{5000}{\sqrt{50{,}000{,}000}}, \frac{-5000}{\sqrt{50{,}000{,}000}} \rangle = \langle \frac{0}{7{,}071}, \frac{5000}{7{,}071}, \frac{-5000}{7{,}071} \rangle$$

$$N_{norm} = \langle \mathbf{0}, \ \ \mathbf{0.7071}, \ \ -\mathbf{0.7071} \rangle$$

Polygon Surface Normal Calculations							
Polygon	P_0	P_1	P_2	P	Q	N	N_{norm}
Front	(0,-100,100)	(50,-150,50)	(-50,-150,50)	(50,-50,-50)	(-50,-50,-50)	(0,5000, -5000)	(0, 0.7071, -0.7071)
Back	(0,-100,100)	(-50,-150,150)	(50,-150,150)	(-50,-50,50)	(50,-50,50)	(0,5000,5000)	(0, 0.7071, 0.7071)
Left	(0,-100,100)	(-50,-150,50)	(-50,-150,150)	(-50,-50,-50)	(-50,-50,50)	(-5000,5000,0)	(-0.7071, 0.7071, 0)
Right	(0,-100,100)	(50,-150,150)	(50,-150,150)	(50,-50,50)	(50,-50,-50)	(5000, 5000, 0)	(0.7071, 0.7071, 0)
Base	(50,-150,50)	(50,-150,150)	(-50,-150,150)	(0, 0, 100)	(-100, 0, 100)	(0, -10000, 0)	(0, -1, 0)

Notice that the sign of the Z component of the Front polygon's normal is negative. In a Left Handed coordinate system, this polygon is facing toward the $Z = 0$ projection plane. The sign of the Z component of the Back polygon's normal is positive. In a Left Handed coordinate system, that polygon is facing away from the $Z = 0$ projection plane. The Z components of the Left, Right, and Base polygon normals are 0 meaning they point parallel to the $Z = 0$ projection plane.

Frequently, a polygon with a surface normal containing a negative Z value will be visible in a Left Hand coordinate system, but not always. The Front polygon is visible.

Likewise, a polygon with a surface normal containing a positive Z value often will **not** be visible in a Left Hand coordinate system, but not always. The Back polygon is not visible.

Some polygons with zero Z values will be visible (e.g., Left and Right) and other's not visible (e.g., Base).

We clearly need some additional information to implement a back face removal algorithm.

This is not surprising as **the normal vector of a polygon indicates the polygon's orientation in space, but not its position**. Remember, vectors specify a magnitude (length) and direction. When normalized to unit vectors of length 1, vectors specify only direction.

Before we move on to the problem of specifying the position of a polygon, let's pause for a moment to remind ourselves about the importance of vertex ordering when defining polygons.

In the above example, notice that the vertices of all of the polygons were labeled clockwise when looking toward the object from the outside. **IF** all polygons (triangles) in the object mesh are defined using a consistent order – either always clockwise or always counter clockwise when looking toward the object from the outside – we will be able to construct outwardly pointing surface normal, as we did above.

Consistency is critical here, if you define some "outward" facing sides clockwise and others counter-clockwise, some of your surface normals will point "inward" and others "outward". The unfortunate result will be that some polygons that should be visible may be culled, and some polygons that should be back facing will be drawn.

The convention that we will use in this class is to define all polygons by listing their vertices in **clockwise** order when viewed from the outside.

12.2 Establishing a Polygon's Position in 3-D Space

You will often see a plane defined in the mathematical literature as:

$$Ax + By + Cz = D \quad where\ D = Aa + Bb + Cc$$

A, B, C, and D are scalar constants defining the plane, and (x, y, z) is any point in the plane.

This equation can easily be turned into a test to determine the relative position of any point in 3-D space relative to the plane.

If $Ax + By + Cz - D = 0 \quad where\ D = Aa + Bb + Cc$ Then (x, y, z) is in the plane.

If $Ax + By + Cz - D > 0 \quad where\ D = Aa + Bb + Cc$ Then (x, y, z) is above the plane.

If $Ax + By + Cz - D < 0 \quad where\ D = Aa + Bb + Cc$ Then (x, y, z) is below the plane.

You may be asking yourself, "Ok, but where do these six constants (A, B, C, a, b, c) come from?"

It turns out we can also define the plane offset, D, as the dot product of the plane's surface normal and any point in the plane.

In the previous section we learned how to compute the normal vector of the plane containing a polygon from the (first) three vertices of that polygon. The computation of the normal vector, N, established the orientation in 3-D space of the plane containing the polygon, but not the position of the plane – there are an infinite number of different planes that have the same surface normal, and thus orientation. To lock down the position of the unique plane that contains our polygon, we simply need a point that lies within that plane – we often use the first vertex of the polygon, P_0, as that point.

While not strictly necessary, by convention we normalize the normal vector, N, to the unit vector N_{norm}. Although this may seem like a wasteful, unnecessary step, there are other calculations involving the surface normal that require the normalize version of that vector to work properly, so it just makes sense to go ahead normalize the normal vector.

Here is the calculation of the plane offset, D, in dot product notation.

$$[N_{norm_X} \quad N_{norm_Y} \quad N_{norm_Z}] \cdot \begin{bmatrix} P_{0_X} \\ P_{0_Y} \\ P_{o_Z} \end{bmatrix} = D$$

Or equivalently, $$N_{norm_X} \cdot P_{0_X} + N_{norm_Y} \cdot P_{0_Y} + N_{norm_Z} \cdot P_{0_Z} = D$$

And we see our original plane equation can be re-written:

$$N_{norm_X} \cdot x + N_{norm_Y} \cdot y + N_{norm_Z} \cdot z = D \quad where\ D = N_{norm_X} \cdot P_{0_X} + N_{norm_Y} \cdot P_{0_Y} + N_{norm_Z} \cdot P_{0_Z}$$

Note that all we did was rewrite the standard plane equation:

$$Ax + By + Cz = D \quad where\ D = Aa + Bb + Cc$$

Substituting $N_{norm} = \left(N_{norm_X}, N_{norm_Y}, N_{norm_Z}\right)$ for (A, B, C) and $P_0 = \left(P_{0_X}, P_{0_Y}, P_{0_Z}\right)$ for (a, b, c).

12.3 Determining if an Arbitrary Point is Above, In, or Below the Plane of a Polygon

We now have all the information we need to determine whether any point in 3-D space (x, y, z) is above, in, or below the plane of a given polygon. Furthermore if point (x, y, z) represents our viewpoint in space and that point is above the plane of the polygon, then the polygon will be visible from that point, unless it is occluded by some other, closer, polygon.

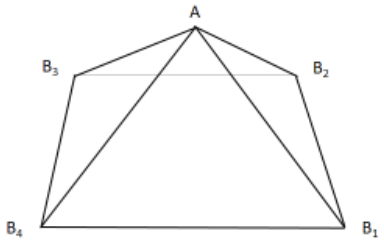
Given $D = N_{norm_X} \cdot P_{0_X} + N_{norm_Y} \cdot P_{0_Y} + N_{norm_Z} \cdot P_{0_Z}$

If $N_{norm_X} \cdot x + N_{norm_Y} \cdot y + N_{norm_Z} \cdot z - D > 0$ Then (x, y, z) is above the plane of the polygon. The polygon is facing (x, y, z) and if not occluded by other polygons will be visible from that point.

If $N_{norm_X} \cdot x + N_{norm_Y} \cdot y + N_{norm_Z} \cdot z - D = 0$ Then (x, y, z) is in the plane of the polygon. The polygon is "edge on" to (x, y, z) and will not be visible from that point.

If $N_{norm_X} \cdot x + N_{norm_Y} \cdot y + N_{norm_Z} \cdot z = D < 0$ Then (x, y, z) is below the plane of the polygon. The polygon is facing away from (x,y,z) and will not be visible from that point.

Returning to our pyramid…



Front: A, B1, B4 Back: A, B3, B2 Left: A, B4, B3 Right: A, B2, B1 Base: B1, B2, B3, B4

$A = (0, -100, 100)$ $B1 = (50, -150, 50)$ $B2 = (50, -150, 150)$ $B3 = (-50, -150, 150)$ $B4 = (-50, -150, 50)$

Polygon Surface Normal Calculations							
Polygon	P_0	P_1	P_2	P	Q	N	N_{norm}
Front	(0,-100,100)	(50,-150,50)	(-50,-150,50)	(50,-50,-50)	(-50,-50,-50)	(0,5000, -5000)	(0, 0.7071, -0.7071)
Back	(0,-100,100)	(-50,-150,150)	(50,-150,150)	(-50,-50,50)	(50,-50,50)	(0,5000,5000)	(0, 0.7071, 0.7071)
Left	(0,-100,100)	(-50,-150,50)	(-50,-150,150)	(-50,-50,-50)	(-50,-50,50)	(-5000,5000,0)	(-0.7071, 0.7071, 0)
Right	(0,-100,100)	(50,-150,150)	(50,-150,150)	(50,-50,50)	(50,-50,-50)	(5000, 5000, 0)	(0.7071, 0.7071, 0)
Base	(50,-150,50)	(50,-150,150)	(-50,-150,150)	(0, 0, 100)	(-100, 0, 100)	(0, -10000, 0)	(0, -1, 0)

Visibility Computations for the Front polygon, in dot product notation:

$$[N_{norm_X} \quad N_{norm_Y} \quad N_{norm_Z}] \cdot \begin{bmatrix} P_{0_X} \\ P_{0_Y} \\ P_{0_Z} \end{bmatrix} = D \qquad [N_{norm_X} \quad N_{norm_Y} \quad N_{norm_Z}] \cdot \begin{bmatrix} V_X \\ V_y \\ V_Z \end{bmatrix} - D \begin{matrix} > \\ = \\ < \end{matrix} 0$$

$$[0 \quad 0.7071 \quad -0.7071] \cdot \begin{bmatrix} 0 \\ -100 \\ 100 \end{bmatrix} = -141.42 \qquad [0 \quad 0.7071 \quad -0.7071] \cdot \begin{bmatrix} 0 \\ 0 \\ -500 \end{bmatrix} - (-141.42) = 494.97 > 0$$

Visibility Computations for the Front polygon, in equation notation:

$$N_{norm_X} \cdot P_{0_X} + N_{norm_Y} \cdot P_{0_Y} + N_{norm_Z} \cdot P_{0_Z} = D \qquad N_{norm_X} \cdot x + N_{norm_Y} \cdot y + N_{norm_Z} \cdot z - D \begin{matrix} > \\ = \\ < \end{matrix} 0$$

$$0 \cdot 0 + 0.7071 \cdot (-100) + (-0.7071) \cdot 100 = -141.42$$
$$0 \cdot 0 + 0.7071 \cdot 0 + (-0.7071) \cdot (-500) - (-141.42) = 494.97$$

Here are the results of the polygon visibility calculations for all five of the polygons in our pyramid object when viewed from (0,0,-500). As you can see, these results match the image given above with the Front, Left, and Right polygons being drawn, and the Back and Base polygons not drawn. Thus, the light grey edge would not be drawn.

Polygon Visibility Calculations						
Polygon	P_0	N_{norm}	D	View Point	Result	Conclusion
Front	(0,-100,100)	(0, 0.7071, -0.7071)	-141.42	(0,0,-500)	494.97	Visible
Back	(0,-100,100)	(0, 0.7071, 0.7071)	0	(0,0,-500)	-353.55	Not Visible
Left	(0,-100,100)	(-0.7071, 0.7071, 0)	-70.71	(0,0,-500)	70.71	Visible
Right	(0,-100,100)	(0.7071, 0.7071, 0)	-70.71	(0,0,-500)	70.71	Visible
Base	(50,-150,50)	(0, -1, 0)	150	(0,0,-500)	-150	Not Visible

12.4 The Back Face Culling Algorithm

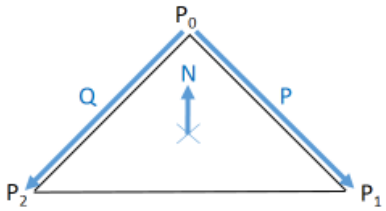
The Back Face Culling algorithm can be called as the first step of "Polygon Draw" method. If the culling algorithm returns "visible" then proceed to draw the polygon as you would normally do. If the culling algorithm returns "not visible" then immediately terminate the "Polygon Draw" method as the polygon cannot be seen from the view point.

This algorithm is written assuming a Left Handed coordinate system.

Inputs: V, The View Point [Often $(0,0,-d)$ where d is the distance from the $Z = 0$ projection plane.]
P_0, P_1, P_2: The first three vertices of the polygon, listed clockwise when viewed from the outside

Output: Visible / Not Visible [This could be implemented as a simple Boolean.]

Step 1: Compute the normal vector for the polygon and then normalize that vector to a unit vector.



This step establishes the orientation of the plane containing the polygon.

P is the vector from P_0 to P_1 which is computed as $P_1 - P_0 = \langle P_{1_X} - P_{0_x},\ P_{1_Y} - P_{0_Y},\ P_{1_Z} - P_{0_Z} \rangle$
Q is the vector from P_0 to P_2 which is computed as $P_2 - P_0 = \langle P_{2_X} - P_{0_x},\ P_{2_Y} - P_{0_Y},\ P_{2_Z} - P_{0_Z} \rangle$

N is the surface normal of the polygon, which can be computed from the Cross Product $N = P \times Q$.

$$N = P \times Q = \langle P_Y \cdot Q_Z - P_Z \cdot Q_Y,\quad P_Z \cdot Q_X - P_X \cdot Q_Z,\quad P_X \cdot Q_Y - P_Y \cdot Q_X \rangle = \langle N_X,\ N_Y,\ N_Z \rangle$$

The surface normal vector, N, can be normalized to a unit vector as follows:

$$N_{norm} = \langle \frac{N_X}{\sqrt{N_X^2 + N_Y^2 + N_Z^2}}, \frac{N_Y}{\sqrt{N_X^2 + N_Y^2 + N_Z^2}}, \frac{N_Z}{\sqrt{N_X^2 + N_Y^2 + N_Z^2}} \rangle$$

Step 2: Compute the Plane Offset

This step establishes the position of the plane containing the polygon.

$$D = N_{norm_X} \cdot P_{0_X} + N_{norm_Y} \cdot P_{0_Y} + N_{norm_Z} \cdot P_{0_Z}$$

Step 3: Determine Visibility of the Polygon from Viewpoint, V

If $N_{norm_X} \cdot V_X + N_{norm_Y} \cdot V_Y + N_{norm_Z} \cdot V_Z - D > 0$ Then Return Visible (True)
If $N_{norm_X} \cdot V_X + N_{norm_Y} \cdot V_Y + N_{norm_Z} \cdot V_Z - D = 0$ Then Return Not Visible (False)
If $N_{norm_X} \cdot V_X + N_{norm_Y} \cdot V_Y + N_{norm_Z} \cdot V_Z - D < 0$ Then Return Not Visible (False)

13. Polygon Filling

In the previous section (12. Back face Culling), we learned how to avoid drawing polygons that face away from the viewer. That is the first step in making 3-D polygonal objects appear "solid". However, in that section we were still rendering polygon by drawing only their edges.

In this section we will look at "painting" or "filling" polygons. This is the next step in making our objects appear more solid.

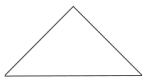
Polygon filling takes place in **displace coordinate space.**

Unfortunately, polygon filling is a very expensive process, in that it operates at the level of individual pixels, rather than at the "line draw" level. To offset this cost, filling is generally implemented in hardware within graphics cards.

In this class, however, we will code the polygon filling algorithm directly in Python. Since Python's TKinter graphics library doesn't provide access to individual pixels, we will draw tiny one pixel width lines. So, to draw a "pixel" at (x, y) we will use a method such as `w.create_line(x,y,x+1,y)`.

Our polygon fill algorithm will come with a few assumptions.

One important assumption is that **our polygons are defined such that their edges never cross.** Mathematicians refer to these kinds of polygons as "simple polygons". A triangle is a simple polygon since it is impossible for a triangle's three edges to cross one another. Once we have four or more edges it is possible that the polygon's edges could cross. Mathematicians refer to these edge crossing polygons as "complex polygons".

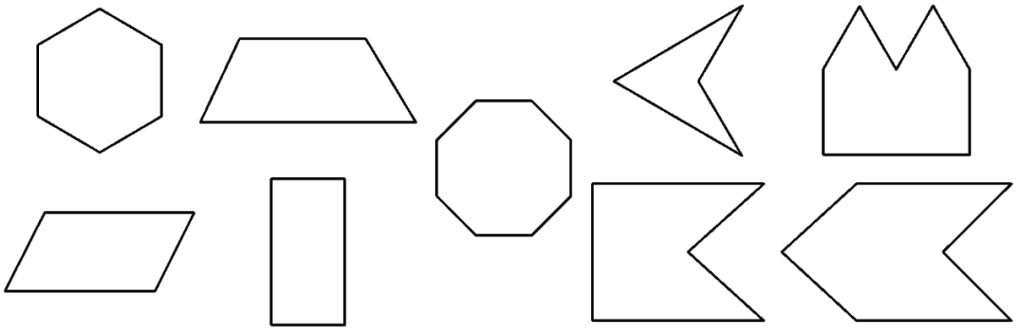


Polygon – Triangle

Polygon – Rectangle

Polygon? (not in this class)

Another underlying assumption of our polygon fill algorithm is that **our polygons are convex**. Simple polygons are "convex" if all of the points on any straight line drawn between two edges lie within the interior of the polygon. The opposite of a convex polygon is a "concave" polygon. Note that the shape of any concave polygon can be constructed from multiple simple polygons.



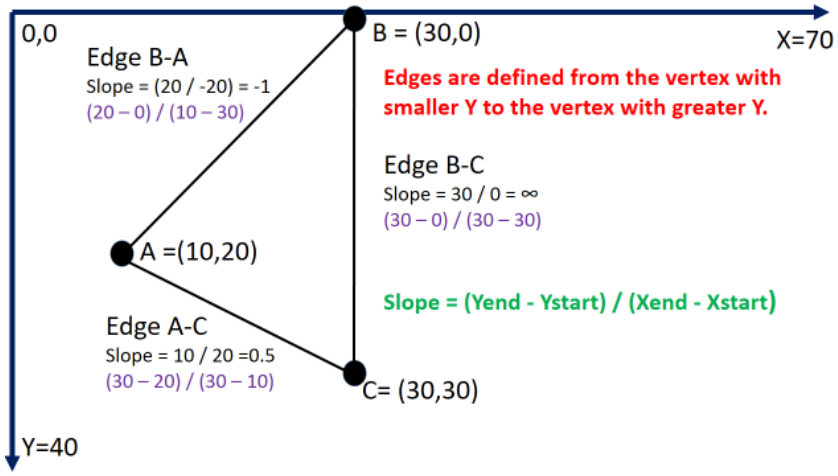
From www.morioh.com: https://morioh.com/p/9ea6ee3536b8

13.1 A Description of the Polygon Fill Process

Now that we have established that the polygon filling (or painting) will be limited to simple convex polygons, like triangles and rectangles, we can begin to think about the details of the process.

A polygon is generally painted from top to bottom, horizontal line by horizontal line. These horizontal lines are filled pixel by pixel working from left to right across the line. By "top to bottom", we mean the polygon will be filled from the point closest to the top of the display window down towards the bottom of the display window.

Since our display coordinate system locates 0,0 in the upper left hand corner of the display window, with increasing X to the right and increasing Y going down (NOT up), the "top most" vertex of a polygon is the vertex with the smallest Y value (NOT the largest).



The image above contains a polygon, in this case a triangle, with three vertices: $A = (10,20)$ $B = (30,0)$ $C = (30,30)$. Of these three vertices, the one with the smallest Y value is $B = (30,0)$. So B is the "top most" vertex, our starting point for filling the polygon.

Our polygon fill algorithm requires that we have a consistent way of defining polygon edges. Individual edges are defined starting from their vertex with the smaller Y (start point), to their vertex with the larger Y (end point). Thus the above polygon consists of three edges, defined: from B to C (written B-C), from B to A (written B-A), and from A to C (written A-C).

What about perfectly horizontal edges, where both vertices have the same Y value? As we will see below, such edges can be safely ignored.

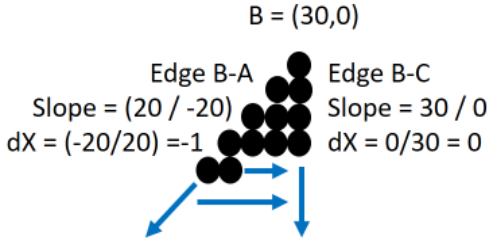
In addition to a start point and an end point, the edges of a polygon will each have a slope. Slope is defined as rise over run; or $\frac{\Delta Y}{\Delta X} = \frac{Y_{end} - Y_{start}}{X_{end} - X_{start}}$.

So, B-A (the edge starting at B and ending at A) has slope $\frac{\Delta Y}{\Delta X} = \frac{Y_{end} - Y_{start}}{X_{end} - X_{start}} = \frac{20-0}{10-30} = \frac{20}{-20} = -1.$

A-C (the edge starting at A and ending at C) has slope $\frac{\Delta Y}{\Delta X} = \frac{Y_{end} - Y_{start}}{X_{end} - X_{start}} = \frac{30-20}{30-10} = \frac{10}{20} = 0.5.$

B-C (the edge starting at B and ending at C) has ∞ slope $\frac{\Delta Y}{\Delta X} = \frac{Y_{end} - Y_{start}}{X_{end} - X_{start}} = \frac{30-0}{30-30} = \frac{30}{0} = \infty.$

So, we will paint or fill our polygons from top to bottom, horizontal fill line by horizontal fill line, with each horizontal fill line painted pixel by pixel working from left to right. Thus, we need an efficient way of determining which horizontal fill lines (rows of pixels) to fill; and where we should start and stop filling along each of those horizontal lines.



Determining which horizontal lines to fill is straightforward. Look at all the vertices of the polygon. Find the smallest Y value. This value represents the first horizontal line to be filled. Similarly, the largest Y value represents the last horizontal line to be filled.

Now that we know which horizontal lines will be filled, we need to figure out where we will start and where we will stop the filling process on each of these horizontal fill lines. A horizontal line of pixels is filled starting at one edge (the Left edge) and ending at the other edge (the Right edge). But how do we find the X values associated with the two edges we wish to fill between?

As mentioned earlier, each edge has a slope that specifies the amount of change in Y (vertical change) over the amount of change in X (horizontal change). In polygon filling however it is the change in Y that is fixed (at 1) as we move from one horizontal fill line (one row of pixels) down to the next horizontal fill line (the next row of pixels).

Given that the change in Y is fixed at 1 as we move from one horizontal fill line to the next horizontal fill line, the inverse slope of the edge $\frac{X_{end}-X_{start}}{Y_{end}-Y_{start}} = \frac{\Delta X}{\Delta Y} = \frac{\Delta X}{1} = \Delta X$ (often written "dX") provides the corresponding change in X along that edge from horizontal fill line to horizontal fill line.

On any particular fill line there are two edges (Left and Right) that we fill between and the slope of these lines may be different, so we need two different ΔX values: one for the Left edge $Edge_{Left}$ and one for the Right edge $Edge_{Right}$. Note that we will be able to tell the Left edge from the Right Edge as the Left edge will have smaller X values than the Right edge – except, perhaps, at the very beginning of the process if both edges share a polygon vertex. So…

The **change in X for the Left edge as we move from horizontal fill line to horizontal fill line** is given by:

$$\Delta X_{Edge_{Left}} = \frac{Xend_{Edge_{Left}} - Xstart_{Edge_{Left}}}{Yend_{Edge_{Left}} - Ystart_{Edge_{Left}}}$$

The **X value for the Left edge on the first horizontal line** of pixels is given by:

$$X_{Edge_{Left}\, FillLine_0} = Xstart_{Edge_{Left}}$$

The **X values for the Left edge on the subsequent horizontal lines** of pixels is given by:

$$X_{Edge_{Left}\, FillLine_{i+1}} = X_{Edge_{Left}\, FillLine_i} + \Delta X_{Edge_{Left}}$$

The **change in X for the Right edge as we move from horizontal fill line to horizontal fill line** is given by:

$$\Delta X_{Edge_{Right}} = \frac{Xend_{Edge_{Right}} - Xstart_{Edge_{Right}}}{Yend_{Edge_{Right}} - Ystart_{Edge_{Right}}}$$

The **X value for the Right edge on the first horizontal line** of pixels is given by:

$$X_{Edge_{Right}\, FillLine_0} = Xstart_{Edge_{Right}}$$

The **X values for the Right edge on the subsequent horizontal lines** of pixels is given by:

$$X_{Edge_{Right}\, FillLine_{i+1}} = X_{Edge_{Right}\, FillLine_i} + \Delta X_{Edge_{Right}}$$

Now that we can compute the starting and ending X values on any particular fill line, all that remains is computing the X values across that fill line. This is very simple as we start at the Left edge and simply add one to the current X value to get the next X value, stopping when we reach the Right edge.

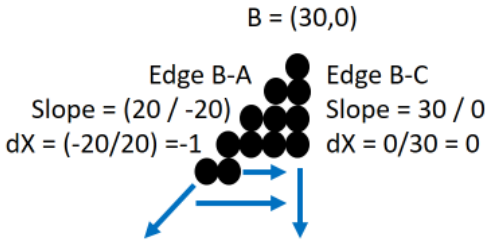
The **X value of the first pixel to be painted on the current fill line** is given by:

$$X_{FillLine_i Pixel_0} = X_{Edge_{Left}\, FillLine_i}$$

The **subsequent X values for the pixels to be painted on the current fill line** is given by:

$$X_{FillLine_i Pixel_{j+1}} = X_{FillLine_i Pixel_j} + 1$$

Returning to the example introduced on the previous page:



The first two edges we will fill between are Edge B-A and Edge B-C, where B-A is the Left edge and B-C is the Right edge. The dX for Edge B-A is $\frac{-20}{20} = -1$ and the dX for Edge B-C is $\frac{0}{30} = 0$. Since both of these edges share a vertex $(30,0)$ we will both start and end our fill at $X = 30$. In other words, we will set the pixel at $(30,0)$.

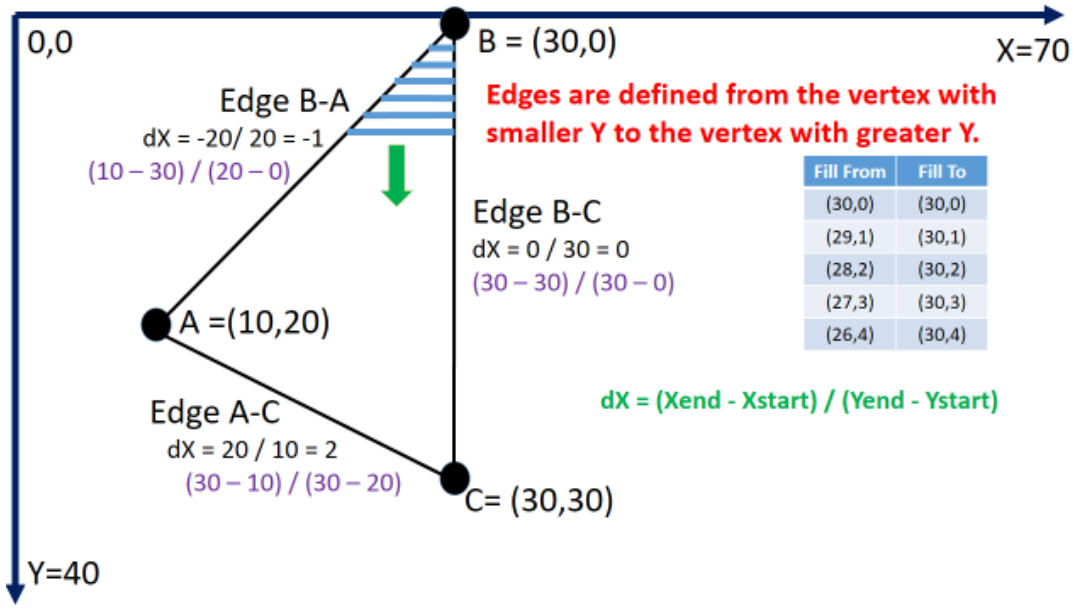
We then move down to the next row of pixels, row 1. The X value for edge (B-A), the Left edge, on row 1 will be $30 + (-1) = 29$; and the X value for edge (B-C), the Right edge, on row 1 will be $30 + 0 = 30$. So two pixels $(29,1)$ and $(30,1)$ are set on row 1.

Moving down one more row to row 2, we once again subtract one from the X value of the Left edge $(29 - 1 = 28)$ and add zero to the X value of the Right edge $30 + 0 = 30$. Thus we begin filling at $(28,2)$ and end filling at (30,2), so three pixels will be set on this row $(28,2)$, $(29,2)$, and $(30,2)$.

These calculations will proceed, adding -1 to the X value of the Left edge and adding 0 to X value of the Right edge, each time we move down a row. This pattern continues until we reach the YMax value of

either of the edges we are filling between; at which point we swap out the edge we are passing and replace it with another edge – until we run out of edges (which occurs when we reach the largest Y value of all the vertices).

Here is a final look at the pyramid we been using to illustrate the fill process. The X values for the Left and Right edges at each of the first five horizontal fill lines are included.



Fill From	Fill To
(30,0)	(30,0)
(29,1)	(30,1)
(28,2)	(30,2)
(27,3)	(30,3)
(26,4)	(30,4)

The information needed to perform the polygon fill procedure can be pre-computed, before we begin painting (filling) the polygon, and stored in a Polygon Fill Table, such as this.

Edge	Xstart	Ystart	Yend	Inverse Slope (dX)
B-A	30	0	20	-20 / 20 = -1
B-C	30	0	30	0 / 30 = 0
A-C	10	20	30	20 / 10 = 2

Polygon Fill Tables have the following characteristics:

- Edges are defined from a start point with the smaller Y, to an end point with the larger Y. Thus the A/B edge is defined from B to A (B-A); the B/C edge from B to C (B-C); and the A/C edge from A to C (A-C)
- The starting point of an edge will be referred to as $(Xstart, Ystart)$
- The edges are sorted by increasing $Ystart$ values. So B-A and B-C appear first, with A-C appearing next. The order of edges that share a starting vertex is arbitrary.
- Fill lines are always drawn between two active edges, but edges are not ordered by leftness and rightness in the table. **Before drawing each fill line you will need to check which of the active edges is the left edge and which is the right edge.**
- $Yend$ is the Y value of the end point (the edge's vertex with the larger Y value). Since B-A goes from B (30,0) to A (10,20), $Yend$ for edge B-A is 20.

- The change in an edge's X value from horizontal fill line to horizontal fill line, dX, is given by the inverse slope of the edge $\frac{Xend-Xstart}{Yend-Ystart}$.
- Horizontal edges are not entered in the table, since they can be neither a left nor a right edge.
- Triangular polygons may have either two or three edges in the table.

Two Edges in Fill Table Three Edges in Fill Table

- Rectangular polygons may have either two or four edges.

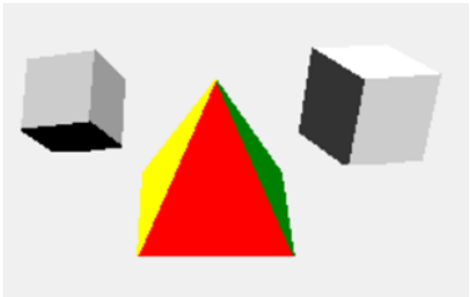
Two Edges in Fill Table Four Edges in Fill Table

- Very "short and thin" or very "tall and skinny" polygons (viewed almost edge on) where their height or thickness is less than a pixel, may generate an empty fill table, even though those polygons may technically (and mathematically) be forward facing.
- It is generally best to ROUND the X and Y coordinates stored in the fill table (do NOT round the dX value), but keep them as real numbers (do NOT convert them to integers). The reason rounding is that ultimately pixel addresses are X-Y (or row-column) indices.

13.2 The Polygon Fill Algorithm

The Polygon Fill Algorithm is presented in pseudo-code form on the following page. The algorithm isn't lengthy or convoluted, but it is "delicate". There are lots of moving parts, so study it carefully. Your third assignment (in part) asks you to implement polygon fill. Please do not blindly "transliterate" this pseudo-code into Python in order to try to solve your programming assignment. Instead, THINK CAREFULLY about what the algorithm is doing and how / why it works. This is important as debugging this program can be time consuming, and without a deep understanding of the algorithm the task will be pretty much hopeless.

The following image was generate by my implementation of the algorithm. I've tested it over a variety of different polygon types: triangles and rectangles; over many different orientations. The basic algorithm should be quite solid (excepting any errant typos, which are always a possibility).



```
PolygonFill(poly, color)

   # Backfacing polygons are not drawn
   If not Visible(poly)Then Return

   # Convert the polygon vertices to projected display coordinates
   # Round them but do not convert to integers
   displayPoly = ProjectandConvertToDisplay(poly)

   # Pre-compute the edge constants: XStart, Ystart, Yend, dX
   EdgeTable = ComputeEdgeTable(displayPoly)

   # Very short polygons (less than a pixel high) are not drawn
   If EdgeTable is empty Then Return

   # Painting proceeds line by line from the first to the last fill line
   FirstFillLine = Edge(0).YStart   # smallest Ystart in EdgeTable
   LastFillLine = Max(Edge(*).Yend) # largest Yend in EdgeTable

   # Indices for the first (I=0), second (J=1), and next (next=2) edges
   I = 0;  J = 1;  Next = 2

   # Paint a fill line by setting all pixels between the first two edges
   EdgeIX = Edge(I).Xstart
   EdgeJX = Edge(J).Xstart

   # Paint one fill line at a time
   For Y from FirstFillLine to LastFillLine

      # Determine which edge is Left and which is Right
      If EdgeIX < EdgeJX Then LeftX = EdgeIX; RightX = EdgeJX
      Else LeftX = EdgeJX; RightX = EdgeIX

      # Paint across a fill line
      For X from LeftX to RightX
         w.create_line(X,Y,X+1,Y,fill=color) /* Set a Pixel */

      # Update the X values of edges I and J for the next fill line
      EdgeIX = EdgeIX + Edge(I).dX
      EdgeJX = EdgeJX + Edge(J).dx

      # Upon reaching the bottom of an edge switch out with next edge
      If (Y >= Edge(I).YEnd) and (Y < LastFillLine)
         Then I=Next; EdgeIX = Edge(I).Xstart; Next++
      If (Y >= Edge(J).YEnd) and (Y < LastFillLine)
         Then J=Next; EdgeJX = Edge(J).Xstart; Next++
```

Notes:

- Remember this algorithm operates in **projected display coordinates**. You will want to round() your coordinates (but NOT dx) to the nearest pixel, but you don't want to use integer math.

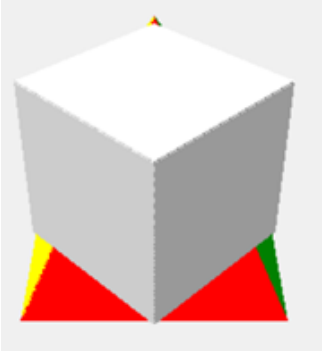
- This algorithm tends to run VERY slowly. On my old laptop it takes about one second per screen refresh, given a scene with two cubes and a pyramid.

14. Using a Z-Buffer to Determine Occlusion

14.1 An Overview of Z-Buffers

The Z-Buffer Algorithm is a general hidden surface removal algorithm. It works with multiple objects, each consisting of multiple polygons, to render a scene in such a way that polygons that are closer occlude (hide) polygons that are farther way. This algorithm is generally integrated into the polygon fill procedure.

We can illustrate the effect of the Z-Buffer Algorithm given two objects: a pyramid and a cube. On the right the pyramid is in front of the cube. On the left the cube is in front of the pyramid. The Z-Buffer Algorithm renders the scene properly, regardless of the order in which the objects were drawn on the screen. The visual result is that both objects appear solid.



The Z-Buffer Algorithm can even handle situations in which polygonal objects intersect one another, as shown below.

While powerful, the Z-Buffer Algorithm does assume that polygons are completely opaque – they are not transparent nor translucent.

The main idea behind the Z-Buffer algorithm is easy to understand. Essentially, we will implement an array – a Z-buffer – with the same dimensions and resolution as our drawing canvas. The Z-buffer will store the depth, or Z information, associated with each pixel we draw on the canvas. When rendering (filling) a polygon, immediately before setting a pixel at some location (X, Y) we check the Z-Buffer at the corresponding location to see if this pixel has already been set by an object that is closer to the view point than the one we are currently rendering. If so, we simply skip setting the pixel because another, closer, object has already set this pixel. Otherwise we set the pixel and update the Z-Buffer to record the depth associated with the object at the point corresponding to the pixel.

The Z-Buffer is initialized with the Z information of each pixel set to a number greater than any practical Z value (call this *MaxDistance*). For example, if we have a 1,920 by 1,080 drawing canvas, we could also have a 1,920 by 1,080 Z-Buffer array, with each location in the array initialized to $MaxDistance$. Immediately before setting pixel (X,Y), we check the Z value at (X,Y). If the Z value of our pixel is less than the Z stored in the Z-buffer for location (X,Y), we set the pixel and update the Z-Buffer at (X,Y) to our Pixel's Z value. If the stored Z value at (X,Y) is less than (or equal to) the depth at (X,Y) of the object we are currently rendering, we skip setting the pixel.

You might be wondering about the difference between a drawing canvas's (X,Y) coordinates and the row, column nature of an array. If one wanted the Z-Buffer's "orientation" to be the same as the image on the drawing canvas, one could reverse the X and Y values in the Z-Buffer. So, for a 1,920 by 1,080 drawing canvas, we could define a 1,080 row by 1,920 column Z-buffer array, and when setting point (X,Y) consult Z-Buffer position (Y,X). None of this swapping is really necessary as the Z-Buffer will work properly if you define it as 1,920 rows by 1,080 columns and consult Z-Buffer position (X,Y) when setting pixel (X,Y). Just be aware that the Z data is logically "flipped" from the orientation of the drawn image.

14.1 Computing the Z values

While the general idea behind the Z-Buffer Algorithm probably makes sense to you, there are a lot of questions we've not yet addressed. The most important question is: Where are all of these Z values supposed to come from? I mean any implementation of Z-buffer requires that we know the Z value for EVERY point we paint on the screen. Surely, we don't have a Z for EVERY point, right?"

It is true that we will need a Z for every pixel we want to paint on a polygon, and we are given only the Z values for the polygon vertices. However, since polygons are planar (flat), we will be able to compute the Z value associated with every point (every pixel) we paint.

Another concern that some students express is that, since the Z-Buffer Algorithm is usually incorporated into the polygon fill procedure, the Z values of the polygon's vertices will no longer be easily accessible when needed. Although polygon fill is done in the display coordinate system of the drawing canvas – a flat 2-D space – we defined THREE perspective projection equations: one for X, one for Y, and one for Z.

The perspective projection equations defined in Section 3 and the conversion to display coordinates equations presented in Section 4 are reproduced immediately below.

$$\boldsymbol{X_{ps} = d \cdot \frac{X}{d+Z}} \qquad \boldsymbol{Y_{ps} = d \cdot \frac{Y}{d+Z}} \qquad \boldsymbol{Z_{ps} = d \cdot \frac{Z}{d+Z}}$$

Where: (X, Y, Z) is the polygon vertex to be projected
d is the distance from the origin on X-Y plane to the center of projection
(X_{ps}, Y_{ps}) is the projected polygon vertex on the X-Y plane
Z_{ps} is the relative depth of that projected point

$$X_{display} = \frac{DisplayWindowWidth}{2} + X_{ps} \qquad Y_{display} = \frac{DisplayWindowHeight}{2} - Y_{ps}$$

The original vertex (X, Y, Z) becomes $(X_{display}, Y_{display})$ with a relative Z value of Z_{ps}.

Below, we compute the display coordinates and final Z values for our pyramid example. We will use $d = 500$; $DisplayWindowWidth = 400$; and $DisplayWindowHeight = 400$, as these are the same values employed in our programming assignments.

Before we can begin filling polygons, the Z-buffer must be initialized. Given our standard viewing system, what should we use for *MaxDistance*?

Let's look at our projection equation for Z_{ps} once more and use $d = 500$:

$$\boldsymbol{Z_{ps} = d \cdot \frac{Z}{d + Z}}$$

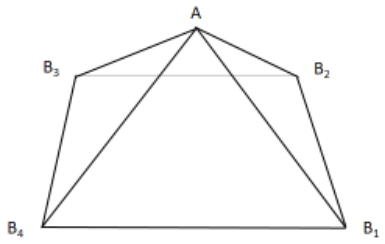
Projecting a point with $Z = 0$ gives: $Z_{ps} = d \cdot \frac{Z}{d+Z} = 500 \cdot \frac{0}{500+0} = 0$

Negative Z's produce negative results. $Z = -100: Z_{ps} = d \cdot \frac{Z}{d+Z} = 500 \cdot \frac{-100}{500+(-100)} = 500 \cdot \frac{-100}{400} = -125$

Large positive Z's generate results approaching, but never reaching 500. $Z = 1{,}000: Z_{ps} = 333$ and $Z = 10{,}000: Z_{ps} = d \cdot \frac{Z}{d+Z} = 500 \cdot \frac{10{,}000}{500+10{,}00} \approx 500 \cdot 95\% = 475$

Thus, we can safely use $MaxDistance = 500$. More generally, $MaxDistance = d$ is always a good choice when projecting Z as we do above.

We can now turn our attention to computing the projected display coordinates for our pyramid.



Front: A, B1, B4 $A = (0, -100, 100)$ Back: A, B3, B2 $B1 = (50, -150, 50)$ Left: A, B4, B3 $B2 = (50, -150, 150)$ Right: A, B2, B1 $B3 = (-50, -150, 150)$ Base: B1, B2, B3, B4 $B4 = (-50, -150, 50)$

Vertex	Original Vertices	Projected Vertices	Display Vertices
A	(0, -100, 100)	(0, -83.33, 83.33)	(200, 283, 83.33)
B1	(50, -150, 50)	(45.45, -136.36, 45.45)	(245, 336, 45.45)
B2	(50, -150, 150)	(38.46, -115.38, 115.38)	(238, 315, 115.38)
B3	(-50, -150, 150)	(-38.46, -115.38, 115.38)	(162, 315, 115.38)
B4	(-50, -150, 50)	(-45.45, -136.36, 45.45)	(155, 336, 45.45)

Now that we have the display coordinates for each pyramid vertex and their corresponding Z_{ps} values, let's focus on painting one of the polygons, Front, and computing the Z values for each pixel as we paint the polygon.

This process will be VERY similar to the way we computed the X values in the Polygon Fill Algorithm. Just as we computed a dX value (inverse slope) to specify how much an edge's X value should change each time we moved down a fill line, we now define a similar equation that specifies how much an edge's Z value will change each time we move down a fill line.

$$\Delta Z = \frac{Z_{end} - Z_{Start}}{Y_{end} - Y_{start}}$$

We will use a "bi-linear interpolation" technique to reproduce the missing Z values. First, we will compute the Z values for the two current edges on the active fill line. When considering an edge, we use the Z value at the starting vertex as the initial Z value for this edge and then each time we move down a fill line we add the dZ value for the edge to the Z value of that edge on the previous fill line. This gives us the edge's Z value on the next fill line. We then use the Z values of those edges as starting and ending Z values for the fill line, and generate Z values for each pixel across the fill line.

The **change in Z for the Left edge as we move from horizontal fill line to horizontal fill line** is given by:

$$\Delta Z_{Edge_{Left}} = \frac{Zend_{Edge_{Left}} - Zstart_{Edge_{Left}}}{Yend_{Edge_{Left}} - Ystart_{Edge_{Left}}}$$

The **Z value for the Left edge on the first horizontal fill line** of pixels is given by:

$$Z_{Edge_{Left}\, FillLine_0} = Zstart_{Edge_{Left}}$$

The **Z values for the Left edge on the subsequent horizontal fill lines** of pixels is given by:

$$Z_{Edge_{Left}\, FillLine_{i+1}} = Z_{Edge_{Left}\, FillLine_i} + \Delta Z_{Edge_{Left}}$$

The **change in Z for the Right edge as we move from horizontal fill line to horizontal fill line** is given by:

$$\Delta Z_{Edge_{Right}} = \frac{Zend_{Edge_{Right}} - Zstart_{Edge_{Right}}}{Yend_{Edge_{Right}} - Ystart_{Edge_{Right}}}$$

The **Z value for the Right edge on the first horizontal fill line** of pixels is given by:

$$Z_{Edge_{Right}\, FillLine_0} = Zstart_{Edge_{Right}}$$

The **Z values for the Right edge on the subsequent horizontal fill lines** of pixels is given by:

$$Z_{Edge_{Right}\, FillLine_{i+1}} = Z_{Edge_{Right}\, FillLine_i} + \Delta Z_{Edge_{Right}}$$

These equations for computing the Z values along the active edges are identical to polygon fill equations for computing the X values along the active edges with the sole exception being that we have replaced every reference of X with a reference to Z.

The process of computing Z Values across a horizontal fill line is different than computing X values across the horizontal fill line in the polygon fill algorithm. The reason is that X values across a fill line increase simply by 1 as we move from pixel to pixel. Z values, on the other hand, need to increase smoothly between the Z value of the Left edge on the current fill line to the Z value of the Right Edge on the current fill line.

The **change in Z as we move across the horizontal fill line from pixel to pixel** is given by:

$$\Delta Z_{FillLine_i} = \frac{Z_{Edge_{Right}FillLine_i} - Z_{Edge_{Left}FillLine_i}}{X_{Edge_{Right}FillLine_i} - X_{Edge_{Left}FillLine_i}}$$

Notice that this equation requires the X values of the active edges on the current fill line, which is one reason it makes sense to integrate the Z-Buffer Algorithm into the polygon fill procedure. We need these X values because the difference between them provides us with the "number of steps" over which the difference in Z between the two active edges on the current fill line must be spread.

The **Z value for the first pixel on the current horizontal fill line** is given by:

$$Z_{FillLine_i\ Pixel_0} = Z_{Edge_{Left}FillLine_i}$$

The **subsequent Z values for the pixels on the current horizontal fill line** is given by:

$$Z_{FillLine_i\ Pixel_{j+1}} = Z_{FillLine_i\ Pixel_j} + \Delta Z_{FillLine_i}$$

These equations provide us with the Z values for every pixel on the polygon. To implement general hidden surface removal, before painting a pixel compare that pixel's Z value to the value held in the Z-buffer at the pixel's location. If the pixel's Z is smaller than the recorded Z, paint the pixel and update the Z-buffer with the pixel's Z value; otherwise this pixel location has already been set while painting some other object that is closer, so do nothing.

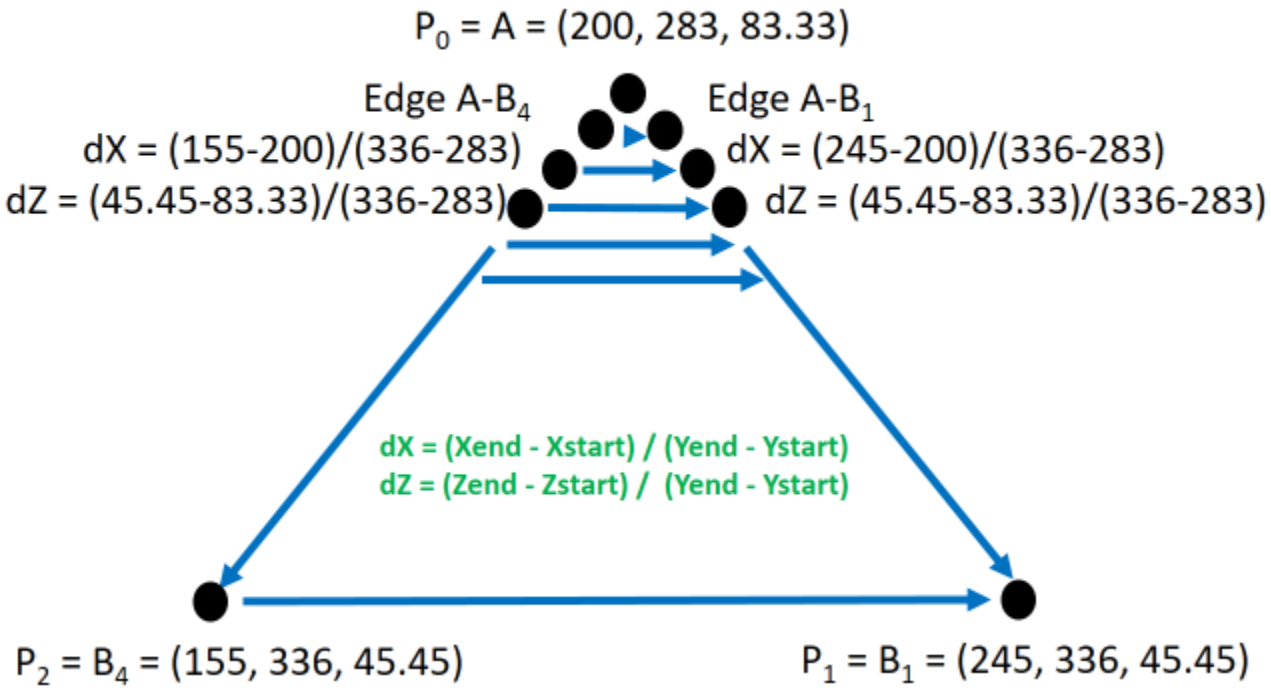
Let's return to the Front polygon of our pyramid object to see how the addition of a Z-buffer affects the Polygon Fill Algorithm.

Here is a Polygon Fill Table for our pyramid's Front polygon with columns added for Zstart and dZ. The dZ values stored in the table are edge dZ's (ΔZ_{edge}). The dZ for the fill lines ($\Delta Z_{FillLine}$) cannot be pre-computed and stored in the table as their computation requires X and Z values from both active edges that are specific to individual fill lines.

Edge	Xstart	Ystart	Yend	dX	Zstart	dZ
A-B1	200	283	336	0.84906	83.33	-0.71472
A-B4	200	283	336	-0.84906	83.33	-0.71472
B4-B1	155	336	336	∞	45.45	∞

An interesting feature of this polygon fill table is that the final edge in the table (edge B_4-B_1) is shown in grey to indicate that it would be omitted from the table. This edge is horizontal and is thus redundant; the final fill line between the two active edges will automatically draw the edge. Horizontal edges, with infinite dX and dZ, and which can never serve as either a Left edge or a Right edge, are always omitted from the table.

Here is a graphical representation of the Front polygon during the filling process showing the first few pixels of the two active edges and the computation of ΔX and ΔZ on both active edges.



14.3 The Z-Buffer Algorithm

The Z-Buffer Algorithm is generally integrated into the Polygon Fill algorithm, since it naturally fits into the polygon filling process. The Polygon Fill Algorithm of Section 13.2 is modified below to include the Z-buffer procedure. All new Z-buffer related code is shown in **green**.

I have implemented this algorithm and tested it rather thoroughly. It should be relatively solid. As was the case with the original Polygon Fill Algorithm. When implementing Z-Buffering in your programming assignment, I ask that you don't just "transliterate" the code, as you will need a deep understanding to be able to debug your program. Diagnostic prints can be your friend in this process. Additionally, limiting the number of on screen polygons while in the debug process can reduce the amount of data you must look through to track down problems.

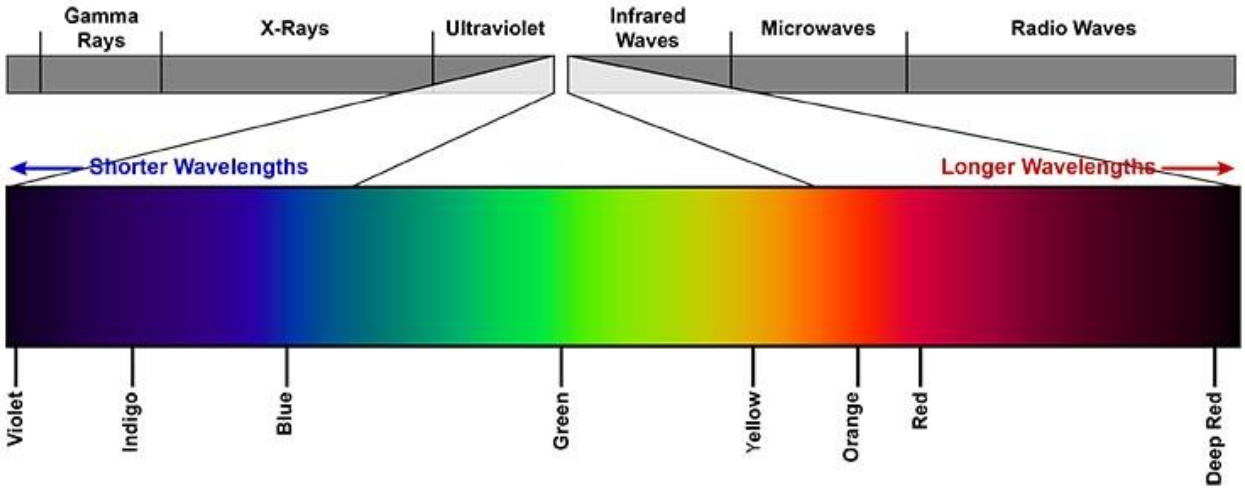
The code is presented on the following page.

```
PolygonFill(poly, color, zBuffer)

   # Backfacing polygons are not drawn
   If not Visible(poly)Then Return

   # Convert the polygon vertices to projected display coordinates
   # Round them but do not convert to integers
   displayPoly = ProjectandConvertToDisplay(poly)

   # Pre-compute the edge constants: XStart, Ystart, Yend, dX, Zstart, dZ
   EdgeTable = ComputeEdgeTable(displayPoly)

   # Very short polygons (less than a pixel high) are not drawn
   If EdgeTable is empty Then Return

   # Painting proceeds line by line from the first to the last fill line
   FirstFillLine = Edge(0).YStart   # smallest Ystart in EdgeTable
   LastFillLine = Max(Edge(*).Yend) # largest Yend in EdgeTable

   # Indices for the first (I=0), second (J=1), and next (next=2) edges
   I = 0;  J = 1;  Next = 2

   # Paint a fill line by setting all pixels between the first two edges
   EdgeIX = Edge(I).Xstart;  EdgeJX = Edge(J).Xstart
   EdgeIZ = Edge(I).Zstart;  EdgeJZ = Edge(J).Zstart

   # Paint one fill line at a time
   For Y from FirstFillLine to LastFillLine

      # Determine which edge is Left and which is Right
      If EdgeIX < EdgeJX
         Then LeftX=EdgeIX; LeftZ=EdgeIZ; RightX=EdgeJX; RightZ=EdgeJZ
         Else LeftX=EdgeJX; LeftZ=EdgeJZ; RightX=EdgeIX; RightZ=EdgeIZ

      # The initial Z for the current fill line
      Z = LeftZ

      # Compute dZ for the fill line. Can be 0 if line is 1 pixel long
      If RightZ-LeftZ != 0 Then dZFillLine=(RightZ-LeftZ)/(RightX-LeftX)
      Else dZFillLine = 0

      # Paint across a fill line
      For X from LeftX to RightX
         If Z < zBuffer[X][Y]
            Then w.create_line(X,Y,X+1,Y,fill=color); zBuffer[X][Y] = Z
         Z = Z + dZFillLine

      # Update the X and Z values of edges I and J for the next fill line
      EdgeIX = EdgeIX + Edge(I).dX; EdgeJX = EdgeJX + Edge(J).dx
      EdgeIZ = EdgeIZ + Edge(I).dZ; EdgeJZ = EdgeJZ + Edge(J).dZ

      # Upon reaching the bottom of an edge switch out with next edge
      If (Y >= Edge(I).YEnd) and (Y < LastFillLine)
         Then I=Next; EdgeIX=Edge(I).Xstart; EdgeIZ=Edge(I).Zstart; Next++
      If (Y >= Edge(J).YEnd) and (Y < LastFillLine)
         Then J=Next; EdgeJX=Edge(J).Xstart; EdgeIZ=Edge(J).Zstart; Next++
```

15. Light and Color

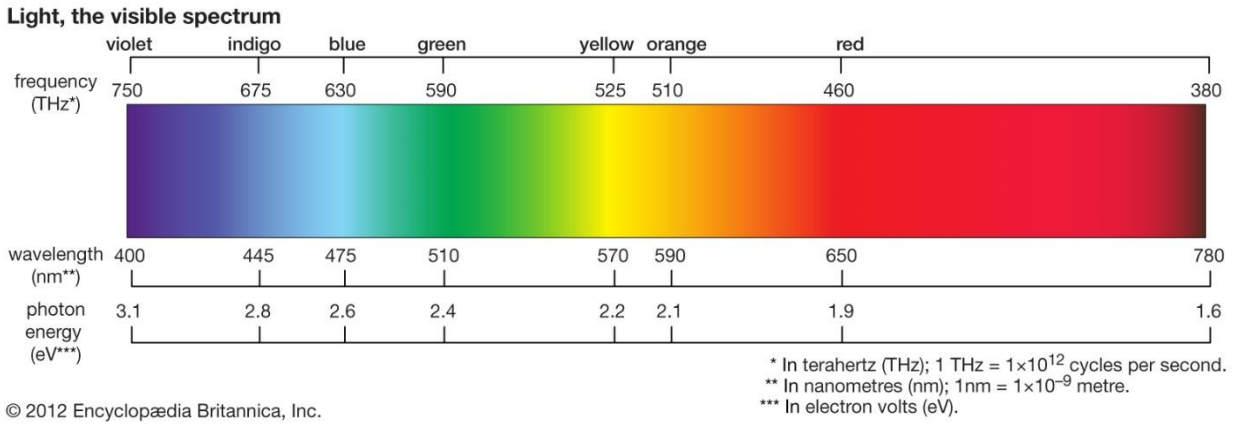
15.1 Light and its Properties

Light is a part of the electromagnetic (EM) spectrum that our eyes are sensitive to. It falls between the ultraviolet (which we humans cannot directly sense) and the infrared part of the spectrum (which humans sense as heat).



[https://www.macuhealth.com/learning-center/visual-performance/]

Light is often characterized by its wavelength, expressed in nanometers (nm); or sometimes by its frequency, measured in terahertz (THz, trillions of cycles per second). Humans have attached names to many of the wavelengths of light; such as “red” for light at a wavelength of 650 nm, “green” for light near 550 nm, and “yellow” at 580 nm.



[https://www.britannica.com/science/light]

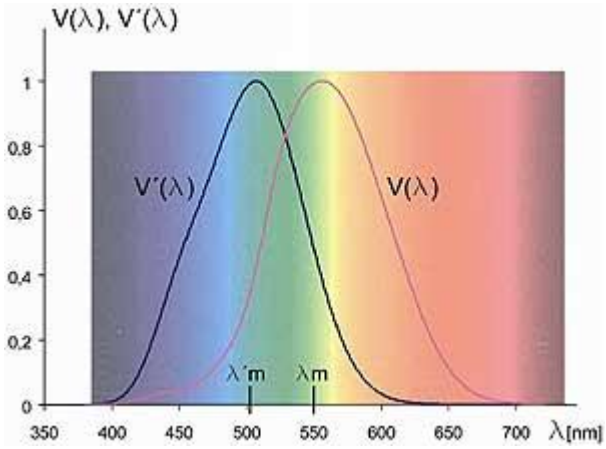
Three common terms used to describe the quality of light, other than its wavelength, are: ”radiance”, “luminance”, and “brightness”.

Radiance is the total amount of energy that flows from an EM source (light, heat, X-Rays, etc.) measured in watts. Old fashion incandescant lights are often categorized by their radiance: a 60 watt light bulb, or

a 100 watt light bulb. Generally we think of incandescant lights with higher wattage as “brighter”, though “brightness” depends on many factors.

Our eyes are not equally sensitive to all visible wavelengths. The apparent “brightness” of 1 watt of EM energy is dependent on its wavelength. This shouldn’t be surprising as we are blind to wavelengths that are too short (ultraviolet or below) or too long (infrared or above) regardless of the amount of radiated energy. Thus, as we vary the wavelength of a one watt EM energy source across the visible spectrum, the apparent brightness of that source will change.

During daylight (photopic) conditions, the human eye is most sensitive to light near 555 nm (“bright green”); under nighttime (scotopic) conditions, peak sensitivity shifts to around 507 nm (“dark green”). [Wikipedia: Spectral sensitivity]



Spectral luminous efficiency functions V(λ) for photopic vision and V'(λ) for scotopic vision
[https://light-measurement.com/spectral-sensitivity-of-eye/]

The above graphic illustrates that under daylight conditions 1 watt of 490 nm light is only about 20% as “bright” as 1 watt of 555 nm light. Stated another way, under normal daylight conditions, 1 watt of 555 nm light is FIVE times “brighter” than 1 watt of 490 nm light. To make the 490 nm light appear as “bright” as the 1 watt 555 nm light, you must quintuple its power output to 5 watts.

Luminance, which is measured in candela per square meter, is the total light emitted or reflected from a surface in a given direction. **Luminous flux**, which is measured in lumens, is the total perceived power emitted in all directions by a light source.

Whereas radience is concerned with the entire EM spectrum, luminance / luminous flux are concerned only with that portion of the EM spectrum we refer to as “light”. Modern LED light bulbs are categorized by lumens, to express the total perceived light they emit in all directions. This makes more sense than using watts, since much of the energy consumed by an old fashioned incandescent bulb is radiated as waste heat. Incandescant bulbs generate about 15 lumens/watt. LED bulbs, while they get hot, are much cooler than incandescant bulbs – far more of their energy goes to generating light than heat. LED bulbs generate about 60 lumens/watt and are thus approximately four times more efficient at generating light than incandescant bulbs.

In the above discussion, I've been (rather annoyingly) placing quotes around the words "bright" and "brightness" as I hadn't yet formally defined the term. **Brightness** is a subjective measure of the perceived intensity of light. Because brightness is subjective, and dependent on many factors, brightness is hard to precisely measure (quantify).

Take a burning candle for example. The candle's radiance can be measured. Much of the EM energy given off by the candle is heat, some part of its radiated energy is light – a yellowish-orangish light with a wavelength in the 580-600 nm range. The luminosity of the candle can be measured, of course, but its brightness is subjective. When the lit candle is in a very dark room, it can appear quite bright. Yet, the same burning candle, when viewed outdoors on a bright sunshiny day, will appear far les bright – though its radiance and luminosity has not changed.

In this course, we will model light sources as either **"point" light sources** or **"diffuse" light sources**, where a point light source is modeled as originating from a point in space, and a diffuse light source is modeled as originating from a surface. In the real world, there is no such thing as a true point light source. Light is always emitted or reflected from a surface – even a tiny LED light, such as the one on the back of your phone, has a surface. However, light bulbs can often be effectively modeled as point sources, while light reflecting off a wall can be modeled as a diffuse source.

15.2 The Perception of Color

Color is a characteristic of human visual perception. *There is no unique unambiguous real-world phenomenon that corresponds to what we humans call "color".*

Wait!!! What??? Didn't we JUST defined color as a particular wavelength of light, and say something to the effect that "green" is light with a wavelength of 550 nm, and "red" is light at 650 nm.

Well, actually I said:

> ***Humans have attached names to many of the wavelengths of light****; such as "red" for light at a wavelength of 650 nm, "green" for light near 550 nm, and "yellow" at 580 nm.*

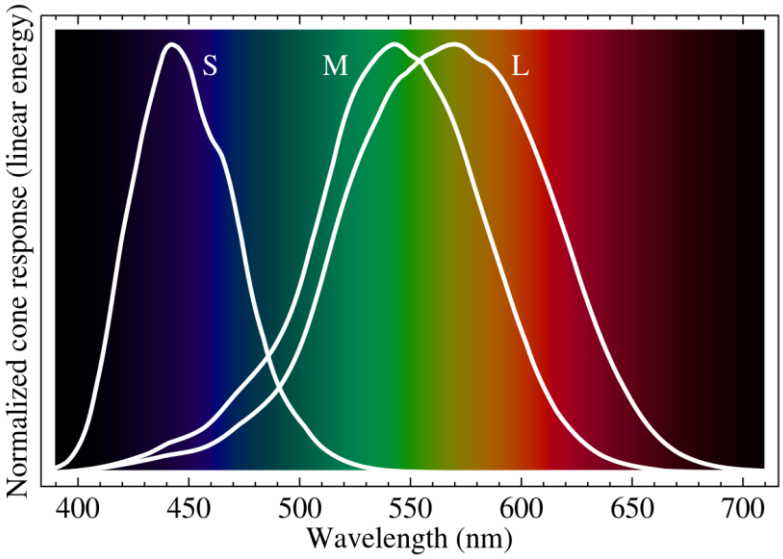
That is not quite the same as saying: "the color yellow is defined as EM radiation at a wavelength of 580 nm".

To understand the point I am trying to make and why color doesn't really exist as a unique, unambiguous real world phenomenon, we will need to learn a bit about how the human visual system perceives color.

The human eye contains three types of "color" receptor cells at the back of the eye in the retina, called **cones**. These cells are called "cones" because their physical shape is cone-like.

The three types of cones are called "S" cones, "M" cones, and "L" cones; based on their sensitivity to particular wavelengths of light. The "S" cones are most sensitive to wavelengths of light on the shorter end of the human visual spectrum (peaking around 450 nm). The "M" cones are most sensitive to wavelengths of light in the middle region of the human visual spectrum (peaking around 540 nm). The "L" cones are most sensitive to wavelengths of light on the longer end of the human visual spectrum (peaking around 570 nm). "S" cones are sometimes called "blue" cones, since we humans have attached the name "blue" to light at a wavelength of 450 nm. Likewise, "M" cones are often called

"green" cones, and "L" cones "red" cones. To be clear, when viewed under a microscope, "green" cones do **not** appear green. The relative sensitivity of the three cone types to the wavelengths of light that make up the visible spectrum is given below.



The human construct of color is ultimately derived from the inputs of these three types of cones. Humans do NOT have a "wavelength" sensing organ that directly feeds precise wavelength measurements into our brains. Instead, we have these three independent "detectors", each of which is most responsive to a particular wavelength.

Consider light at 450 nm, which we call "blue" light. How do we know the light is "blue"? Well, the "S" cone will be generating a large response to the income light; "M" and "L" will be generating a small response, with "M" somewhat higher than "L". That is how our eyes detect "blue".

"Red" light can be detected by the "L" cones generating a large response, a weak response from "M" cones, and essentially no response from "S" cones.

Ok, Mike, humans can not directly sense a particular wavelength, but the wavelength can be inferred, so why do you say that color doesn't exist as a unique, unambiguous characteristic of the real world?

For two reasons: (1) humans can see "colors", such as yellow, when their corresponding wavelengths aren't actually present; and (2) humans can see colors that don't correspond to any single wavelength of light.

Take "yellow" light, defined as light with a wavelength of 580 nm. Although the human visual system does not have "yellow" cones maximally responsive to 580 nm wavelengths of light, our visual system can "infer" the presence of that wavelength of light when the "L" cones respond at near 100%, while "M" cones respond at about 75%, and the "S" cones do not respond at all.

Our visual system will also respond in exactly the same way (i.e., detect the presence of the color "yellow") when 650 nm wavelength light (light we normally call "red") and 550 nm wavelength light (which we normally call "green") are both present with the proper intensities at the same time. A human will see "yellow" although there is no 580 nm wavelength light present. The subjective

experience is identical. In fact, when you see “yellow” you have no way of knowing whether 580 nm wavelength light is present, or a combination “red” and “green” are present.

So, we can see “colors that aren’t there”.

We can also see colors for which there is no corresponding wavelength of light.

“S” cones respond best to shorter wavelengths, such as “blue”. “L” cones respond best to longer wavelengths, such as “red”. Obviously, there is no wavelength of light that will stimulate both the “S” cones and the “L” cones at the same time. We can, however, shine both 450 nm wavelength light and 650 nm wavelength light into a human eye at the same time. The human will perceive magenta, a reddish purple color. This color has no corresponding wavelength and it does not appear in the spectrum – you won’t find magenta in a rainbow.

15.3 Color Display Devices

The fact that humans do not directly sense the wavelengths of light and that instead our perception of color is based solely on the output of three types of cone cells in the retina, has enabled us to create display devices that appear to be “full color” while actually producing only three colors: red, green, and blue.

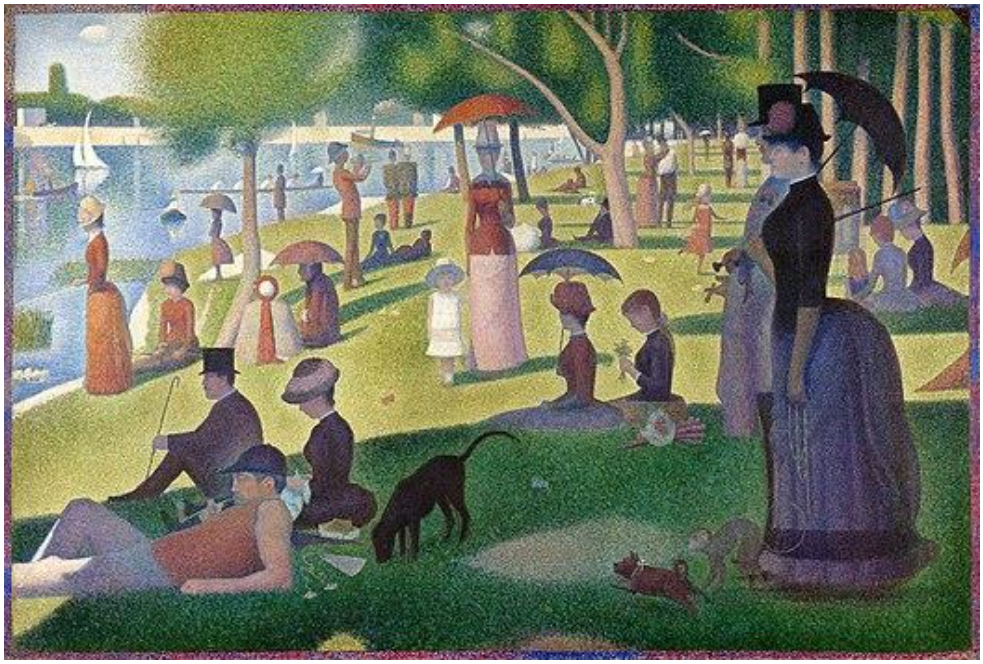
In an LED (Light Emitting Diode) display, each pixel consists of three tiny LEDs: a red LED, a green LED, and a blue LED. The LEDs are so tiny and placed so close together than our eyes cannot make out the individual LEDs, and the colors blend together.

Typically a pixel will have 256 levels of red, 256 levels of green, and 256 levels of blue. Essentially a pixel can reproduce any “color” a human can experience by varying the red, green, and blue levels appropriately. The individual color values of the LEDs are often expressed as two digit hexadecimal numbers 00x to FFx (0 – 255). Pure red is FF0000x. Pure green is 00FF00x. And pure blue is 0000FFx. White is simply red, green, and blue; all at full intensity: FFFFFFx. A LED display approximates a black pixel by turning off all three of the sub-pixel LEDs: 000000x. Levels of grey are produced by having all three sub-pixel LEDs set to the same intensity level: CCCCCCx represents a light grey, while 333333x is a dark grey.

The hexadecimal code for bright yellow is FFFF00x. To be clear, an LED monitor cannot produce light at 580 nm wavelength (which is how we originally defined “yellow”), but by lighting up the red and green LED sub-pixels equally and keeping the blue LED sub-pixel off, a human will experience “yellow” in exactly the same way as if 580 nm wavelength light had been produced.

While LED-based digital displays did not become common until the 21st century, the idea of creating images from large numbers of closely spaced dots, together with the idea that the human eye could “blend” a small number of colors to create the perception of a far wider pallet of colors was explored by the 19th century French post-impressionist artist Georges Seurat (1859-1891).

Seurat is credited with developing the painting technique of pointillism (creating an image from tiny dots) and chromoluminarism (limiting the dots to a small number of colors, which the eye blends together). Seurat’s most famous painting is called “A Sunday Afternoon on the Island of La Grande Jatte” which he painted from 1884 to 1886.



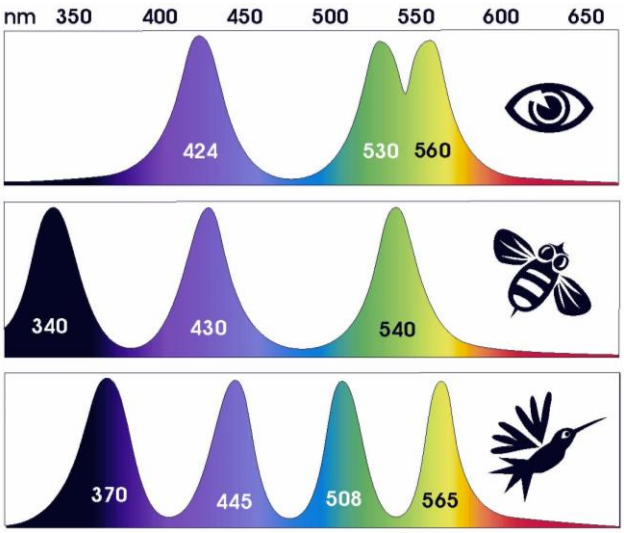
[“A Sunday Afternoon on the Island of La Grande Jatte” by George Seurat, painted from 1884 to 1886]

While Seurat did not use red, green, and blue (or cyan, yellow, magenta for those of you who may have had an art class at some point), his work – 70 years before the advent of the digital computer, and well over a century before color digital display devices became common – is truly amazing. Seurat could not have imagined that the ideas he was exploring in his art would someday form the foundation of the devices that display most of the text and imagery we consume today.

Before leaving the topic of “full color” display devices, I’d like to drive home the fact that really there is nothing “full color” about our RGB devices. These devices simply “trick” the human brain into thinking they can generate the entire visible color spectrum, by manipulating levels of red, green, and blue at the sub-pixel level. This becomes apparent when we think about how different species of animals sense the visible (to us humans) and near UV regions of the EM spectrum.

If the human visual system were closer to that of bees we could see into the UV (ultraviolet) part of the spectrum, but our perception of longer wavelengths wouldn’t be as good -- we probably wouldn’t be able to see “red” as well. Intelligent descendants of bees would therefore construct very different display devices than us humans. Instead of an R-G-B color system, they’d need to have something like a UV-B-G system to convincingly reproduce the illusion of “color” for them. Additionally, the fact that Bees have compound eyes might greatly increase the difficulty of building realistic displays.

Birds have much better color vision that either humans or bees – they can see “red” as well as we can, while also being able to see into the UV part of the spectrum almost as far as bees. Birds have four types of cones (instead of the three us humans have), and the wavelengths at which their peak sensitivities occur are more evenly spread out. To birds, humans are rather severely color blind. Intelligent birds would need to build display devices with four color emitters per pixel. Each pixel would probably consist of four sub-pixels: a UV light emitter, a purpleish emitter, a blue-greenish emitter, and a red emitter.



[http://coldcreek.ca/wp-content/uploads/2013/08/spectrum2-1024x852.jpg]

We can't really know how birds internally experience "color". We also cannot know how our limited tri-color human display devices might appear to them. We can be sure that from a bird's point of view many "colors" are missing. The colors that are present probably look somewhat "off" as the LEDs we humans construct have obviously been tuned to emit wavelengths of light at near the "peak" wavelengths our cones are sensitive to.

Perhaps the best analogy to understanding how unrealistic our color displays must appear to a bird is to look at color systems used in early color movies from the 1920's, such as the two-color Technicolor system based on red and green (but not blue). Here is a still image from the movie "Ben Hur" (released in 1925) that used this two color system. Clearly the image is not black and white, but it also isn't "full" color either.

[https://publicdomainmovies.info/ben-hur-a-tale-of-the-christ-1925-film/]

15.4 RGB versus CMY Color Systems

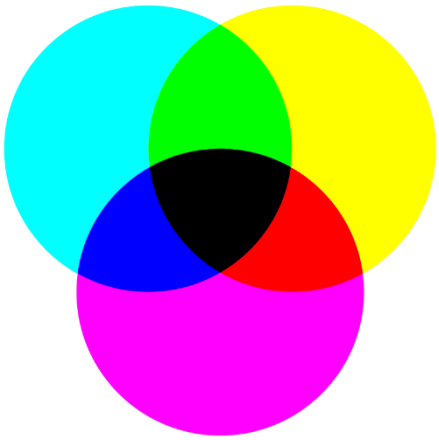
Throughout this section of the notes we have been focused on the fundamental role of “red”, “green”, and “blue” light; how human eyes have evolved three types of cone cells, each most responsive to a particular wavelength; and how the biology of the human eye has driven the design of color display devices. These colors, “red”, “green”, and “blue” are called **primary colors**, because any color a human can experience, can be crafted from a combination of red, green, and blue light.

The RGB (Red, Green, Blue) color system we’ve been discussing is an “additive” system. So named because adding different amounts of red, green, and blue light together can generate the experience of any particular color. We’ve illustrated this with the example of yellow light that can be “simulated” from red and green light.

If you have ever taken an art class, or even replaced the color ink cartridges in a laser printer, you may have seen “cyan”, “magenta”, and “yellow” referred to as the primary colors.

What gives? Are the primary colors red, green, and blue? Or are they cyan, magenta, and yellow?

Well, the answer is that **red, green, and blue are the primary colors of light**; while **cyan, magenta, and yellow are the primary colors of pigment**. Cyan is a bluish-green color. Magenta is a reddish-purple color. Mixing cyan pigment and yellow pigment will generate green pigment. Mixing cyan pigment and magenta pigment will produce blue pigment. Combining magenta and yellow pigment gives red pigment. Mixing all three primaries together gives black. [Note that laser printers often include black as a separate color, rather than “wasting” cyan, magenta, and yellow pigment just to generate black ink. When black is treated separately, the system is referred to as a CMYK system.]



[https://en.wikipedia.org/wiki/CMY_color_model]

Whereas the RGB system for light is additive, the CMY system for pigments is subtractive. Cyan pigment appears bluish-green because it absorbs (subtracts) red light, and thus reflects blue and green light. We can indicate this by the notation (-R,+G,+B). Magenta is a reddish-purple color because it absorbs only green light, and reflects red and blue light, indicated by (+R,-G,+B). When you mix cyan and magenta pigment together you create a pigment that subtracts both red and green, and reflects blue (-R,-G,+B). The result is a pigment, or ink, that appears blue. Yellow pigment absorbs only blue light (+R,+G,-B). So, mixing yellow and cyan pigment produces green pigment (-R,+G,-B).

16. Illumination and Reflection

An **illumination mode**l is a mathematical model for the interaction of light with a surface. In computer graphics, these models are empirical —they are developed by observation and experimentation. The illumination model we explore in this section produces reasonable results within a reasonable computational budget. The model is not a precise simulation of physics, since modeling physics would require computational resources far beyond what is generally available.

16.1 Developing an Illumination Model for Computer Graphics

Illumination models generally consider two types of light sources: light emitting sources and light reflecting sources.

Light emitting sources give off their own light. These sources are often modeled as **point light sources**, which is light originating from a single point in space; and **distributed light sources**, light originating from a surface. An LED light bulb would probably be modeled as a point light source, while a florescent light would probably be modeled as a distributed light source.

Light reflecting sources are non-emitters, most everything in the environment, except the light emitting sources, can be considered light reflecting sources. Walls, floors, books, desks, people, etc. – all are considered light reflecting sources. It's important to realize that even if you can't see a particular light reflecting source, perhaps because it is behind you, or because it is occluded by another object, its reflectivity still adds to the overall illumination of a scene.

Ambient light is the light coming from all of the reflecting surfaces in the environment. Ambient light is sometimes called "background light". Because ambient light is coming from all directions at once, it is considered as directionless in most illumination models.

Objects can, of course, be illuminated by both ambient light and light emitting sources at the same time. In fact, in reality, this is almost always the case. Thus, in order to model illumination of a scene realistically, the model should account for the effects of both ambient light and emitted light, from all sources, both point and distributed. However, the "standard" illumination model we will study limits itself to ambient and point sources. While it quickly gets expensive, distributed light sources could be modeled by multiple closely spaced point sources.

Just as there are two types of light emitters: point light sources and distributed light sources; there are two kinds of reflection: diffuse reflection and specular reflection.

With **diffuse reflection** incoming light is reflected in all directions equally. We will model two types of diffuse reflection: ambient diffuse reflection and emitted diffuse reflection.

Ambient diffuse reflection models background ambient light that arrives at a surface equally from all directions and is reflected equally in all directions. The term "ambient diffuse reflection" is often just shortened to "*ambient*".

Emitted diffuse reflection of point light sources models light that arrives at a surface from a particular direction, but is reflected equally in all directions. The term "emitted diffuse reflection of point light sources" is often shortened to "*diffuse*". Emitted diffuse reflection (diffuse) captures how light interacts with dull non-reflective surfaces, like paper or cloth.

Specular reflection models light that arrives at a surface from a particular direction and is reflected preferentially in a particular direction. Specular reflection captures the way light interacts with shiny surfaces.

The most common illumination-reflection model used in computer graphics calculates the intensity of reflected light at a point on a surface (often, but not always, a polygon) using an equation of the following general form, where "n" is the number of point light sources illuminating the scene, and "i" is a particular light source.

$$Intensity = ambient + \sum_{i=1}^{n} (diffuse_i + specular_i)$$

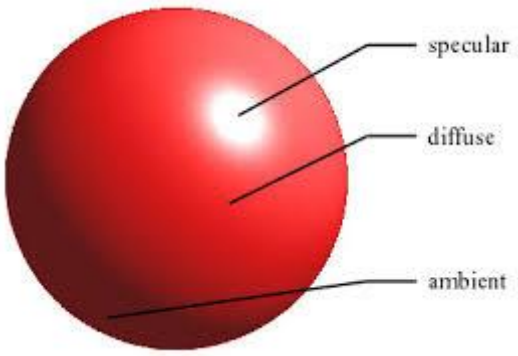
The equation generates the overall intensity of light by summing the ambient diffuse reflection (*ambient*) together with the emitted diffuse reflection of point light sources (*diffuse*) and specular reflection (*specular*) generated by each of the point light sources illuminating the scene. In order to generate a color image, the model is repeated for each of the three primary colors: red, green, and blue giving:

$$Intensity_{red} = ambient_{red} + \sum_{i=1}^{n} (diffuse_{red_i} + specular_{red_i})$$

$$Intensity_{green} = ambient_{green} + \sum_{i=1}^{n} (diffuse_{green_i} + specular_{green_i})$$

$$Intensity_{blue} = ambient_{blue} + \sum_{i=1}^{n} (diffuse_{blue_i} + specular_{blue_i})$$

Here we have an illustration of a sphere rendered using a monochrome (single color, red in this case) illumination model, lit by a single point light source. The effects of the ambient, diffuse and specular components of the model are highlighted.

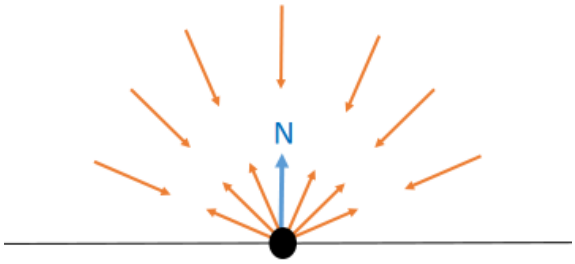


[http://www.csc.villanova.edu/~mdamian/Past/graphicsfa10/notes/LightingShading.pdf]

We now turn our attention to discussing how we compute each of the three terms in the illumination equations, beginning with the ambient term.

16.2 Ambient Diffuse Reflection

Ambient diffuse reflectivity captures the level of background lighting in an environment. This is lighting that appears to be coming equally from all directions and is reflected equally in all directions. Thus, the perceived intensity of ambient light reflecting off an object will be constant, regardless of the orientation of the object or the angle from which it is viewed.



You may be thinking: There is no such thing as ambient lighting and reflection in the real world – all illumination must originate from one of more sources. And you'd be technically correct. However, remember my comment earlier about our illumination model being empirical, and not rigorously based on physics. Some environments are brightly lit and have lots of background illumination; other environments are dimly lit and have very little background illumination. We model this with the ambient term in our illumination equation.

When you look at an object, what determines the brightness (and color) of that object? A number of things: the intensity of the light reaching the object, the color of that light, the intrinsic color of the object being illuminated, and the properties of the object's surface (e.g., shiny or dull).

In the monochrome illumination model, we represent the intensity of incoming ambient light as I_a and the constant of diffuse reflectivity for an object as K_d. The constant of diffuse reflectivity expresses how intrinsically light or dark the object is. I_a represents the level of ambient light in the environment. K_d is a property of the object being rendered. An object's actual appearance depends on both I_a and K_d.

Consider a bright white billiard ball. In a completely dark room the white ball will appear black – you won't be able to see it as there is no light to illuminate anything. Likewise, if the room is very dimly lit, you will see the ball but it won't be "bright white" – it will look more like a dark grey ball. The greater the intensity of the room lighting the more brightly lit the billiard ball will be and the closer to "bright white" it will appear (up to a point where we reach saturation). We capture this interaction between the illumination level and the properties of the object by multiplying the two terms together to produce the apparent shade of the object from ambient lighting.

$$Ambient = I_a \cdot K_d$$

Now let's consider illuminating objects in a "full color" (actually tri-color) RGB model. We will need three equations, each identical to the above, one for the red component of ambient reflection, one for the green component of ambient refection, and one for the blue component.

$$Ambient_{red} = I_{a_{red}} \cdot K_{d_{red}}$$

$$Ambient_{green} = I_{a_{green}} \cdot K_{d_{green}}$$

$$Ambient_{blue} = I_{a_{blue}} \cdot K_{d_{blue}}$$

In the tri-color RGB illumination model there are three separate components of the intensity of incoming ambient light: $I_{a_{red}}$, $I_{a_{green}}$, and $I_{a_{blue}}$. Similarly, the intrinsic color of the object to be illuminated is expressed as three independent constants of diffuse reflectivity, one for each of the three primary colors: $K_{d_{red}}$, $K_{d_{green}}$, and $K_{d_{blue}}$. As above we multiply $I_a \cdot K_d$ to produce the ambient component of the object's apparent brightness and color of the object in the environment.

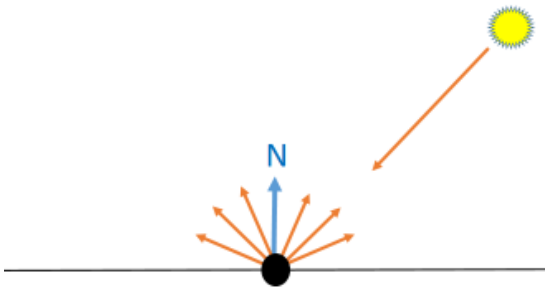
When computing illumination, it is standard practice to scale inputs between 0 and 1, inclusive. Thus, even though color values in most display systems range from 0 to 255; both I_a and K_d will range from 0 to 1.

In the monochrome illumination model, if we have a room with a moderate amount of ambient lighting ($I_a = 0.25$); and an object with a K_d of 0.5 (medium grey), the intensity of reflected ambient light generated by this object would be: $0.25 \cdot 0.5 = 0.125$. Before painting the pixel we'd need to multiply the result by 255 giving 31.875, which rounds to 32 or 20x in hexadecimal.

In the RBG illumination model, given a scene containing a moderate amount of white ambient lighting ($I_{a_{red}} = 0.25$, $I_{a_{green}} = 0.25$, and $I_{a_{blue}} = 0.25$) together with a yellow object of medium intensity ($K_{d_{red}} = 0.5$, $K_{d_{green}} = 0.5$, and $K_{d_{blue}} = 0$), the intensity of the reflected ambient light generated by this object would be: (0.125, 0.125, 0). After multiplying each component by 255 and converting to hexadecimal we have (20x, 20x, 0x).

16.3 Emitted Diffuse Reflection of Point Light Sources

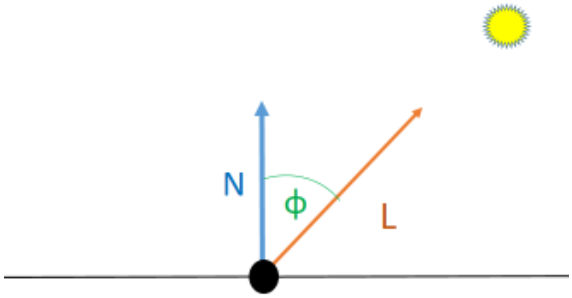
Emitted diffuse reflectivity captures the level of lighting reaching the surface of an object from a point light source that is then reflected equally in all directions. Light from a point source will strike the surface of the object from some particular direction. The amount of light from the source reaching that surface will depend on the orientation of the object's surface to the light source. Since we are discussing the emitted diffuse component of reflection, that light will be reflected equally in all directions. Thus, the perceived intensity of emitted diffuse reflection of an object is independent of the angle from which the object is viewed, but dependent on the angle between the surface normal and the direction of lighting.



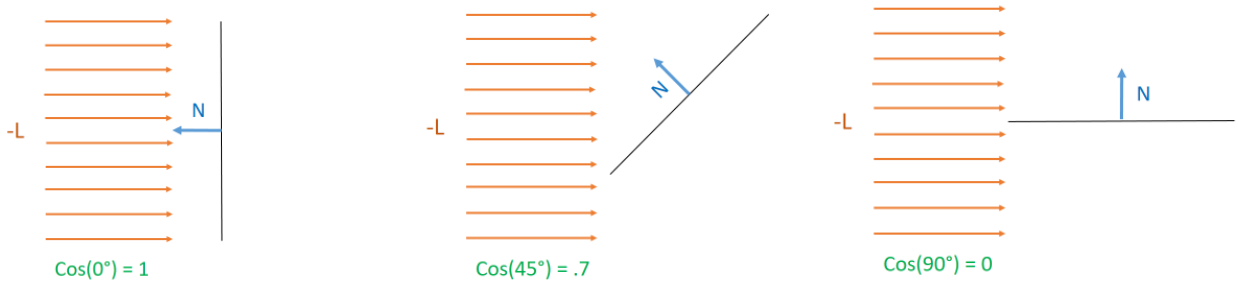
Obviously, an environment can be illuminated by multiple light sources, and each would need to be handled separately, but in the discussion below we limit ourselves to a single point light source.

When a surface is illuminate by light coming from a particular direction, the illumination of that surface is governed by Lambert's Cosine Law (named after Johann Heinrich Lambert, a Swiss mathematician / physicist / astronomer / and cartographer who lived in the mid 1700's).

Specifically, a surface with a normal, N, illuminated by light arriving at the surface from direction, L, is directly proportional to the cosine of ϕ, the angle between N and L.



We can intuitively grasp why this is true if we think of equally spaced parallel rays of light striking a surface with some fixed width. When the light is directly overhead and the angle between the lighting vector and the surface normal is 0, 100% of the available light will strike the surface [$\cos(0°) = 1$]. When the angle between the lighting vector and the surface normal is 45°only 70% of the available light will strike the surface [$\cos(45°) = 0.7$]. (To see that this is true, count the little light rays in the illustration below where the angle is 45°, only seven of the ten rays actually reach the surface.) When the angle between the lighting vector and the surface normal is 90° none of the light rays strike the surface because they are all parallel to that surface [$\cos(90°) = 0$].

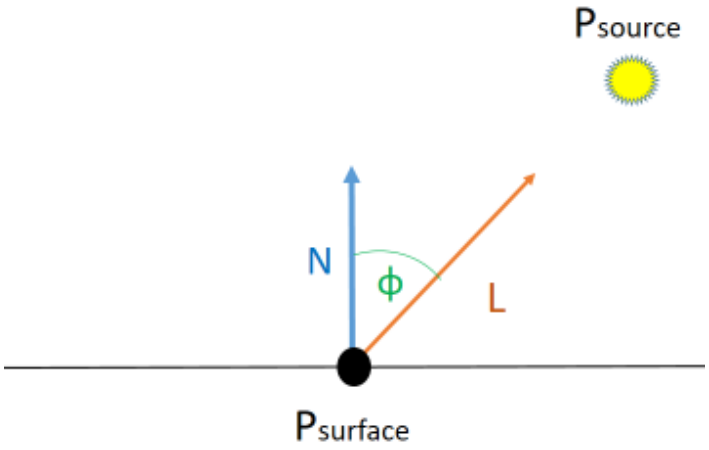


It is important to keep in mind that the lighting vector, L, points in the direct the light is coming from. As is the case with the surface normal vector, N, by convention L is normalized – both N and L are unit vectors of length 1. (The reason for this will become clear momentarily.)

Computing L is straightforward:

$$L = \frac{P_{source} - P_{surface}}{\left|P_{source} - P_{surface}\right|}$$

Of course, since P_{source} and $P_{surface}$ are points in 3-D space, with an X, Y, and Z component; and L is a vector in 3-D space with X, Y, and Z components; this equation breaks out into three separate equations – one for X, one for Y, and one for Z.



$$L_X = \frac{P_{source_X} - P_{surface_X}}{\sqrt{\left(P_{source_X} - P_{surface_X}\right)^2 + \left(P_{source_Y} - P_{surface_Y}\right)^2 + \left(P_{source_Z} - P_{surface_Z}\right)^2}}$$

$$L_Y = \frac{P_{source_Y} - P_{surface_Y}}{\sqrt{\left(P_{source_X} - P_{surface_X}\right)^2 + \left(P_{source_Y} - P_{surface_Y}\right)^2 + \left(P_{source_Z} - P_{surface_Z}\right)^2}}$$

$$L_Z = \frac{P_{source_Z} - P_{surface_Z}}{\sqrt{\left(P_{source_X} - P_{surface_X}\right)^2 + \left(P_{source_Y} - P_{surface_Y}\right)^2 + \left(P_{source_Z} - P_{surface_Z}\right)^2}}$$

Each point on the surface of the polygon will have a slightly different L vector, so technically we should re-compute L for every point on the surface of every polygon to be rendered.

This can get expensive, and in many cases the light source is far enough away from the objects in the scene that it can essentially be taken as being "at infinity". In such cases, we often directly specify the lighting vector, L, that defines the direction to the light source for all objects in the scene. All light rays from the source are assumed parallel in this case.

For example, light entering the scene from a 45° angle over the viewer's right shoulder could be defined by the vector $(1,1,-1)$ in our left-handed viewing system, which normalizes to $L = (0.577, 0.577, -0.577)$. This vector, L, would then be used throughout the scene, regardless of which polygon you were rendering or for which point on the polygon you were computing the illumination.

In addition to the lighting angle, the distance of the surface from the light source can also affect illumination. The further an object is from a source the more dimly it will be lit. From basic physics you may recall that this relationship is governed by an inverse square law.

$$I_{surface} = \frac{I_{source}}{d^2}$$

The intensity of light reaching a surface equals the intensity of light at the source divided by the distance between the source and surface squared. Though this equation faithfully expresses the way light attenuates with distance in the "real world", we tend not to use this for formula as the results are often deemed too "aggressive" in the way in which the light reaching a surface dims with distance. Often, either distance, d, will be ignored (set to 1 in the following equations) or instead of dividing by d^2we divide by d. If this bothers you, remember the model we are developing is empirical (the model is organized the way it is because it gives reasonable looking results with a reasonable expenditure of computing resources).

Monochrome single source diffuse reflection of point light is modeled as follows:

$$diffuse = I_P \cdot K_d \cdot \cos(\phi) \,/\, d$$

Where:

I_P : intensity of the light at the point source

ϕ : angle between the lighting vector, L, and the surface normal, N

K_d : constant of diffuse reflectivity for the object being illuminated

d : distance between the point light source and the surface (set to 1 to ignore distance)

Monochrome multiple source diffuse reflection of point light is modeled as follows:

$$diffuse = \sum_{i=1}^{n} \left(I_{P_i} \cdot K_d \cdot \cos(\phi_i) \,/\, d_i \right)$$

Or equivalently:

$$diffuse = K_d \cdot \sum_{i=1}^{n} \left(I_{P_i} \cdot \cos(\phi_i) \,/\, d_i \right)$$

In Section 11.9.1 we established that the dot product of two vectors, P and Q, was equal to the length of P times the length of Q times the cosine of the angle between P and Q.

$$P \cdot Q = \|P\| \|Q\| \cos \alpha$$

When P and Q are both unit vectors (vectors of length 1):

$$P \cdot Q = 1 \cdot 1 \cdot \cos \alpha$$

So, when P and Q are unit vectors (vectors of length 1):

$$\cos \alpha = P \cdot Q$$

From the above, when N and L are unit vectors (vectors of length 1):

$$\cos(\phi) = N \cdot L \qquad \text{or} \qquad \cos(\phi) = N_X \cdot L_X + N_Y \cdot L_Y + N_Z \cdot L_Z$$

Monochrome single source diffuse reflection of point light *rewritten in dot product form* is modeled as:

$$diffuse = I_P \cdot K_d \cdot (N \cdot L) \,/\, d$$

or

$$diffuse = I_P \cdot K_d \cdot (N_X \cdot L_X + N_Y \cdot L_Y + N_Z \cdot L_Z) \,/\, d$$

Monochrome multiple source diffuse reflection of point light *rewritten in dot product form* is modeled as:

$$diffuse = K_d \cdot \sum_{i=1}^{n} \left(I_{P_i} \cdot (N \cdot L_i)/d_i \right)$$

or

$$diffuse = K_d \cdot \sum_{i=1}^{n} \left(I_{P_i} \cdot \left(N_X \cdot L_{X_i} + N_Y \cdot L_{Y_i} + N_Z \cdot L_{Z_i} \right) /d_i \right)$$

Example: Monochrome single source diffuse reflection of point light

Given: $N = (0,5,0)$; $L = (1,1,0)$; $I_p = 0.5$; $K_d = 0.5$; $d = 1$

The surface normal, N, is parallel to the Y axis point towards $+Y$. The lighting vector, L, models light entering from the upper right at a 45° angle with the rays parallel to the $Z = 0$ plane. Neither N nor L are normalized to unit vectors. The light source is of medium intensity (0.5) as is the object's constant of diffuse reflectivity (0.5). Distance from the light source will be ignored ($d = 1$).

The first thing we must do is normalize N and L (convert them to unit vectors).

$$|N| = \sqrt{0^2 + 5^2 + 0^2} = \sqrt{0 + 25 + 0} = \sqrt{25} = 5$$

$$N_{norm} = \left(\frac{0}{5}, \frac{5}{5}, \frac{0}{5}\right) = (0,1,0)$$

$$|L| = \sqrt{1^2 + 1^2 + 0^2} = \sqrt{1 + 1 + 0} = \sqrt{2} = 1.4142$$

$$L_{norm} = \left(\frac{1}{1.4142}, \frac{1}{1.4142}, \frac{0}{1.4142}\right) = (0.707, 0.707, 0)$$

$$diffuse = I_P \cdot K_d \cdot (N_X \cdot L_X + N_Y \cdot L_Y + N_Z \cdot L_Z) \,/\, d$$

$$diffuse = 0.5 \cdot 0.5 \cdot (0 \cdot 0.707 + 1 \cdot 0.707 + 0 \cdot 0) \,/\, 1$$

$$diffuse = 0.5 \cdot 0.5 \cdot 0.707 = 0.17675$$

Scaling this result to $0 - 255$ gives $0.17675 * 255 = 45 = 2\text{Dx}$.

The monochrome ambient diffuse and single point source diffuse models can be combined to form a combined monochrome reflection model that includes both types of diffuse reflection.

Monochrome single emitter point source + ambient model:

$$Intensity_{ambient+diffuse} = I_a \cdot K_d + I_p \cdot K_d \cdot (N_X \cdot L_X + N_Y \cdot L_Y + N_Z \cdot L_Z)/d$$

Monochrome multiple emitter point source + ambient model:

$$Intensity_{ambient+diffuse} = I_a \cdot K_d + K_d \cdot \sum_{i=1}^{n} \left(I_{P_i} \cdot \left(N_X \cdot L_{X_i} + N_Y \cdot L_{Y_i} + N_Z \cdot L_{Z_i}\right)/d_i\right)$$

This combined ambient diffuse + point diffuse reflection model can be extended to support tri-color RGB.

RGB multiple emitter point source + ambient model:

$$Intensity_{a+d_{red}} = I_{a_{red}} \cdot K_{d_{red}} + K_{d_{red}} \cdot \sum_{i=1}^{n} \left(I_{P_{i_{red}}} \cdot \left(N_X \cdot L_{X_i} + N_Y \cdot L_{Y_i} + N_Z \cdot L_{Z_i}\right)/d_i\right)$$

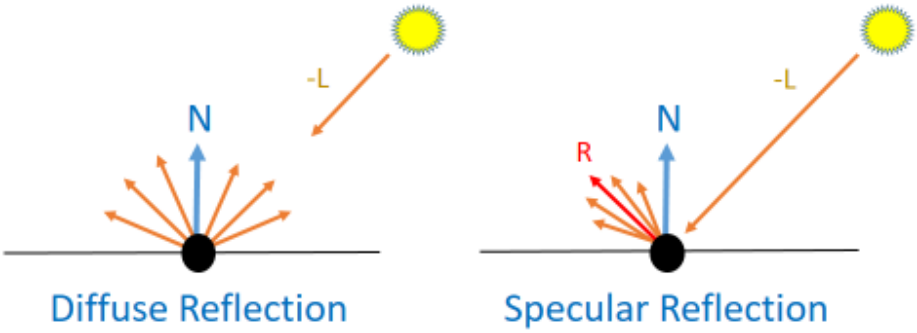
$$Intensity_{a+d_{green}} = I_{a_{green}} \cdot K_{d_{green}} + K_{d_{green}} \cdot \sum_{i=1}^{n} \left(I_{P_{i_{green}}} \cdot \left(N_X \cdot L_{X_i} + N_Y \cdot L_{Y_i} + N_Z \cdot L_{Z_i}\right)/d_i\right)$$

$$Intensity_{a+d_{blue}} = I_{a_{blue}} \cdot K_{d_{blue}} + K_{d_{blue}} \cdot \sum_{i=1}^{n} \left(I_{P_{i_{blue}}} \cdot \left(N_X \cdot L_{X_i} + N_Y \cdot L_{Y_i} + N_Z \cdot L_{Z_i}\right)/d_i\right)$$

All of the equations presented in this Section 16.3 that utilize the cosine of the angle ϕ (or the dot product $N \cdot L$) are valid only in situations were $\cos(\phi) \geq 0$ (or $N \cdot L \geq 0$). When $\cos(\phi) < 0$ (or $N \cdot L < 0$) the surface is being lit from below (as opposed to from above). In such circumstances the point light source does not illuminate the side of the object being viewed, and has no effect on overall illumination. In order to model this situation properly, **when $\cos(\phi) < 0$ (or $N \cdot L < 0$), set $Diffuse = 0$**.

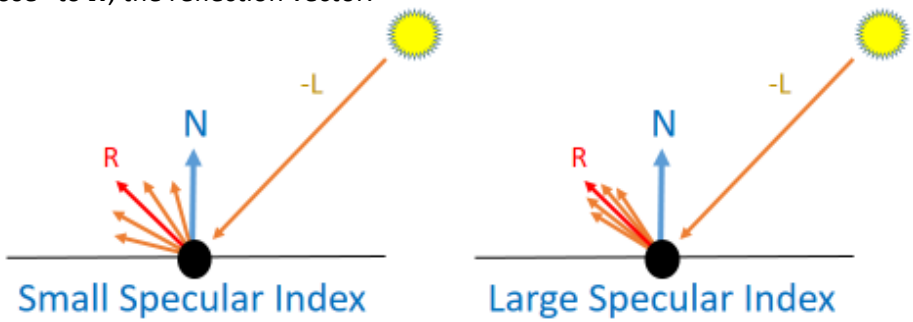
16.4 Specular Reflection

Most surfaces are not perfect diffusers of light. They have some degree of glossiness. In other words, some of the point light reaching a surface from a particular direction will be reflected along the "mirror direction". This reflection along the mirror direction is known as "specular reflection". The model of specular reflection presented in this section was originally developed by Vietnamese-born computer graphics researcher Bui Tuong Phong (1942-1975) and is known as the Phong Reflection Model.



The above figure illustrates the difference between diffuse reflection, in which light is reflected equally in all directions, and specular reflection. Point diffuse reflection is shown on the left, but ambient reflection (reflection of background lighting which arrives at the surface from all directions) is also diffuse in nature. On the right specular reflection is shown. R is the mirror, or reflection, vector. If the surface being illuminated were a "perfect reflector", as is the case with mirrors, the incoming light rays would be reflected precisely along the direction of the R vector.

Most surfaces are neither perfect diffusers of light, nor perfect reflectors – instead they exhibit some degree of "glossiness". For such surfaces, the specular reflection is not perfect. Light will be reflected in directions "close" to R, the reflection vector.



The amount of spread of the light rays around the mirror direction is controlled by the specular index. The **specular index** is a term in the Phong Illumination Model that controls the spread of reflected light around the mirror direction. It often appears as the term n in the Phong equations. However, since n is such a common variable name, used for so many different things, we will refer to the specular index as *specIndex* in our equations and code.

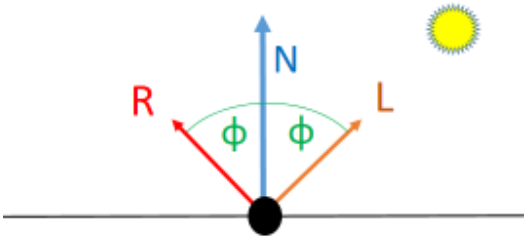
A small specular index, illustrated on the left side of the above figure, is used to represent a surface that is not very glossy. Thus, the reflected light will spread quite a bit from the reflection direction – and produce a large (but dim) highlight. As the specular index approaches 0, the surface becomes less and less glossy – we can see the highlight over a wider and wider area, but it appears dimmer and dimmer. Eventually, at $specIndex = 0$ the surface becomes a perfect diffuser of light and the specular contribution degenerates into another ambient term.

Large specular indices work in the opposite direction. A large specular index, illustrated on the right side of the above figure, represents a highly glossy surface. The reflected light will spread very little and produce a small (but bright) highlight. As the specular index approaches ∞, the specular highlight will become smaller and smaller, eventually reaching a point. One might think that when this occurs the surface would be a mirror and support true reflections. However, that result is beyond the capabilities of the Phong Illumination Model. A different model, called *ray tracing*, described in the final section of

these notes, is often used to model reflections. When the specular index is very large Phong will simply render the highlight as a bright point.

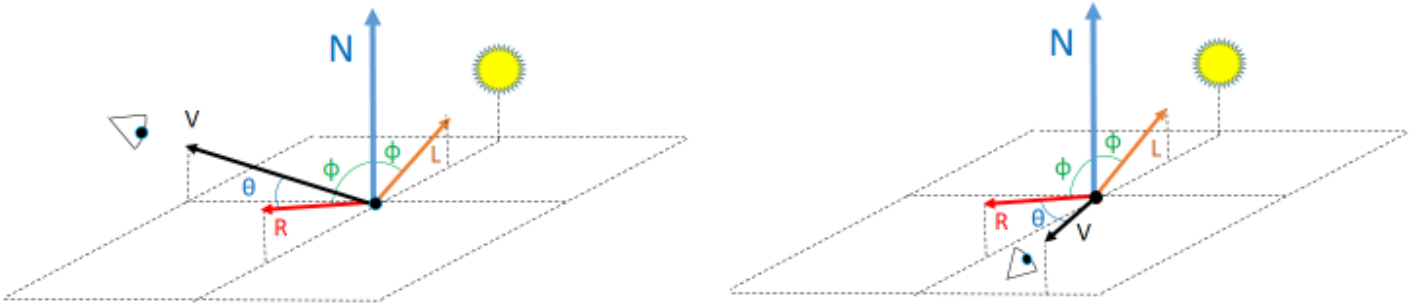
The calculation of Phong specular highlights depend on the interactions between four vectors and two angles.

We are familiar with the first two vectors, N, the surface normal of the object at the point being illuminated; and L, the lighting vector that points towards the light source. We are also familiar with phi, ϕ , the angle between N and L; the cosine of which can be computed as $N \cdot L$, if N and L are unit vectors.



The Reflection Vector: R

We have introduced R, the refection vector defining the mirror direction, in our discussion of Phong specular illumination, but we have not yet described how to compute R. We discuss the computation of R in detail in Section 16.4.2 below. However, for now just know that R can be computed from N and L.



The View Vector, V, in relation to L, N, and R

The final pieces of information we need to compute Phong specular reflection is V, the view vector; and the cosine of the angle theta, θ , between V and R. The cosine of θ can be computed as $V \cdot R$, if V and R are first normalized to unit vectors.

The view vector points towards the location in 3-D space from which the user is viewing the scene. In our programs and examples we've fixed the view vector at $< 0, 0, -1 >$ in our left handed viewing system, but in general the view point can reside anywhere in 3-D space. (If the center of projection and view vectors are not the same, you'd have to go through the various transformations to line the two up prior to rendering the scene, a process very similar to the one we illustrated in Section 10.2.2 for rotation about an arbitrary line in space.)

It is important to note that L, N,and R are all in the same plane. V, in general, is not in the same plane, as should be clear from the above figure.

16.4.1 The Phong Specular Reflection Equations

This section presents the various versions of the Phong specular equations for both monochrome and RGB environments considering both single and multiple point light sources. We begin with the monochrome single source case and analyze the behavior of the specular reflection equation in some detail.

Monochrome single source specular reflection of point light is modeled as:

$$Specular = I_P \cdot K_s \cdot (\cos\theta)^{specIndex}$$

$$Specular = I_P \cdot K_s \cdot (R \cdot V)^{specIndex}$$

$$Specular = I_P \cdot K_s \cdot (R_X \cdot V_X + R_Y \cdot V_Y + R_Z \cdot V_Z)^{specIndex}$$

Above are various versions of the monochrome single source Phong specular calculation. The first directly references the $cos\ (\theta)$, the second expression replaces $cos\ (\theta)$ with $R \cdot V$ which is equivalent when R and V are normalized to unit vectors. The third version of the equation simply expands out $R \cdot V$ into its underlying calculations.

Our first observation is that when R and V are identical, that is when the angle between them, θ , is 0 and thus $\cos(\theta) = 1$, *specIndex* is ignored since 1 raised to any power is 1. The upshot is that the point will be illuminated with the full intensity of $I_P \cdot K_s$.

Next, we observed that when $specIndex = 0$ the value of $(\cos\theta)^{specIndex}$ or $(R \cdot V)^{specIndex}$ is simply 1, as anything raised to the zero power is 1. In this case, the specular computation again simplifies to $Specular = I_P \cdot K_s$ which acts as a secondary diffuse component (similar to ambient, as Lambert's cosine law isn't applied to the angle of incoming light) raising the overall illumination level of the object.

As discussed above, when $specIndex > 0$ incoming light is reflected at angles close to the refection vector, R. The greater the *specIndex* the more tightly focused the reflected light is around R. This means that for a fixed lighting angle, L, and view vector, V, as *specIndex* increases, the amount of specularly reflected light decreases – except when V and R are identical, as discussed above.

We can see this behavior in the equations by fixing the angle between V and R at $\theta = 45^\circ$ (in which case $\cos(45^\circ) \approx 0.7$); and observing the resulting percentage of $I_P \cdot K_s$ illuminating the point of interest as *specIndex* increases.

$specIndex = 0$	$(\cos(45^\circ))^0 = 1.0$	100% of specular light visible from the view point
$specIndex = 1$	$(\cos(45^\circ))^1 \approx 0.7$	70% of specular light visible from the view point
$specIndex = 2$	$(\cos(45^\circ))^2 \approx 0.49$	49% of specular light visible from the view point
$specIndex = 3$	$(\cos(45^\circ))^3 \approx 0.343$	34% of specular light visible from the view point

The monochrome single source specular reflection model can easily be extended to an RGB model by modeling each of the primary colors: red, green, and blue separately.

RGB single source specular reflection of point light (using dot product notation) is modeled as:

$$Specular_{red} = I_{P_{red}} \cdot K_{s_{red}} \cdot (R \cdot V)^{specIndex}$$

$$Specular_{green} = I_{P_{green}} \cdot K_{s_{green}} \cdot (R \cdot V)^{specIndex}$$

$$Specular_{blue} = I_{P_{blue}} \cdot K_{s_{blue}} \cdot (R \cdot V)^{specIndex}$$

The single source models can be extended to multiple sources by computing the effects of each source separately and summing them all together.

Monochrome multiple source specular reflection of point light is modeled as:

$$Specular = K_s \cdot \sum_{i=1}^{n} \left(I_{P_i} \cdot (\cos \theta_i)^{specIndex} \right)$$

$$Specular = K_s \cdot \sum_{i=1}^{n} \left(I_{P_i} \cdot (R_i \cdot V)^{specIndex} \right)$$

$$Specular = K_s \cdot \sum_{i=1}^{n} \left(I_{P_i} \cdot \left(R_{X_i} \cdot V_X + R_{Y_i} \cdot V_Y + R_{Z_i} \cdot V_Z \right)^{specIndex} \right)$$

RGB multiple source specular reflection of point light (using dot product notation) is modeled as:

$$Specular_{red} = K_{s_{red}} \cdot \sum_{i=1}^{n} \left(I_{P_{i_{red}}} \cdot (R_i \cdot V)^{specIndex} \right)$$

$$Specular_{green} = K_{s_{green}} \cdot \sum_{i=1}^{n} \left(I_{P_{i_{green}}} \cdot (R_i \cdot V)^{specIndex} \right)$$

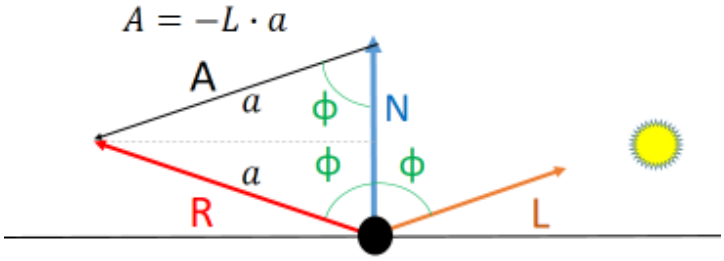
$$Specular_{blue} = K_{s_{blue}} \cdot \sum_{i=1}^{n} \left(I_{P_{i_{blue}}} \cdot (R_i \cdot V)^{specIndex} \right)$$

Before we can make use of the equations for specular reflection, we are still missing the details on one important piece of the puzzle. We've not yet covered the computation of R, the reflection vector. Once we knock that out of the way – in the next section, Section 16.4.2 – we will be in the position to put all the pieces of the Phong illumination equation together, and present a complete example.

16.4.2 Computing the Reflection Vector, R

Given a surface normal, N, and a lighting vector, L, we wish to find the reflection vector, R. The angle between N and L is ϕ, as is the angle between R and N and the angle between $-N$ and A. We can see that A, N, R forms an isosceles triangle.

The situation is illustrated as follows for the case where **ϕ is less than 90°** and the surface is lit from above.



From vector addition, it is clear that in this case we have:

$$R = N + A$$

You might at first think that A is simply $-L$, allowing us to compute $R = N + -L$. Unfortunately, $A \neq -L$. While it is true that that the direction of A is $-L$, vectors possess both magnitude and direction, and there is no guarantee that the magnitude of A and L will be equal. So, the above equation for R can be rewritten:

$$R = N + (-L) \cdot \|A\|$$

How do we compute, a, the magnitude of A? $\|A\| = a$

Since triangle A, N, R is isosceles, dropping a bisector from its apex to its base will create two similar right triangles each with a base of length ½ $\|N\|$ and a hypotenuse of length a.

$$\cos(\phi) = \frac{adjacent}{hypotenuse} = \frac{\frac{1}{2}\|N\|}{a}$$

Furthermore since we normalized our L and N vectors $\|N\| = 1$

$$\cos(\phi) = \frac{\frac{1}{2}\|N\|}{a} = \frac{\frac{1}{2}}{a} = \frac{1}{2 \cdot a}$$

Thus the magnitude of Vector A, a, is:

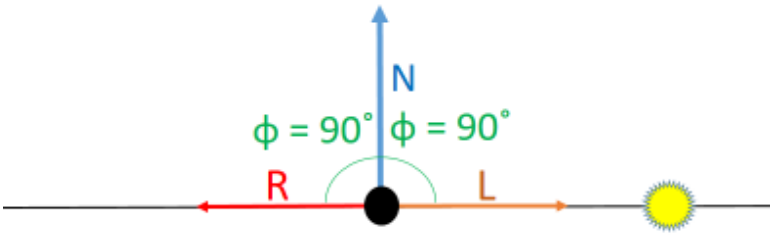
$$a = \frac{1}{2 \cdot \cos(\phi)} = \frac{1}{2 \cdot (N \cdot L)} = \frac{1}{2 \cdot (N_X \cdot L_X + N_Y \cdot L_Y + N_Z \cdot L_Z)}$$

And we can rewrite our equation for R as:

$$R = N + (-L) \cdot \frac{1}{2 \cdot \cos(\phi)}$$

$$\boldsymbol{R = N - \frac{L}{2 \cdot (N_X \cdot L_X + N_Y \cdot L_Y + N_Z \cdot L_Z)}}$$

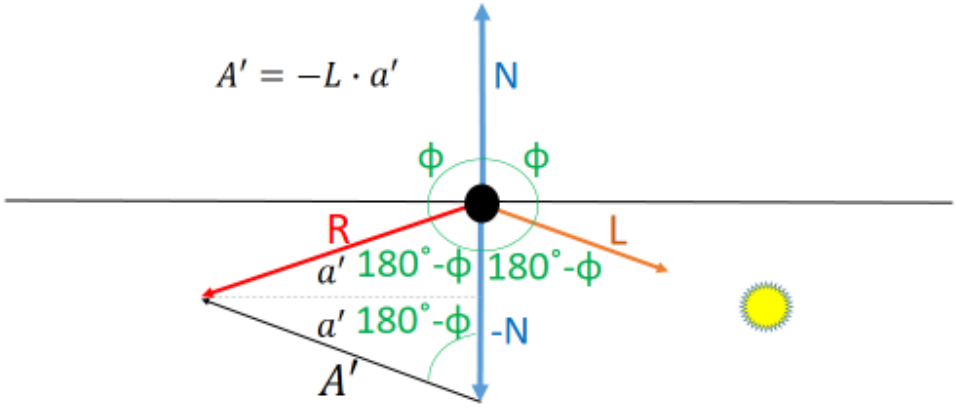
When **ϕ equals 90°** and the light rays are parallel to the surface. This situation can be illustrated as follows:



While the light rays never intersect the surface, we can define the "reflection vector" as:

$$\boldsymbol{R = -L}$$

When **ϕ is greater than 90°** the surface is lit from below.



and the reflection vector is computed as:

$$R = -N + A'$$

The direction of A' is – L. so we have:

$$R = -N + (-L) \cdot \|A'\|$$

The magnitude of Vector A', a', is:

$$a' = \frac{1}{2 \cdot \cos(180^\circ - \phi)} = \frac{1}{2 \cdot (-N \cdot L)} = \frac{1}{2 \cdot (-N_X \cdot L_X + -N_Y \cdot L_Y + -N_Z \cdot L_Z)}$$

And we can rewrite our equation for R as:

$$R = -N + (-L) \cdot \frac{1}{2 \cdot \cos(180^\circ - \phi)}$$

$$R = -N - \frac{L}{2 \cdot (-N \cdot L)} = -N - \frac{L}{2 \cdot (-N_X \cdot L_X + -N_Y \cdot L_Y + -N_Z \cdot L_Z)}$$

$$\boldsymbol{R = -N + \frac{L}{2 \cdot (N_X \cdot L_X + N_Y \cdot L_Y + N_Z \cdot L_Z)}}$$

In the following figure, I have implemented the reflection calculation as a Python method.

```python
import math

# Calculate a 3-D reflection vector, R, given surface normal, N, and lighting vector, L
def reflect(N, L):
    R = []
    N = normalize(N)
    L = normalize(L)
    twoCosPhi = 2 * (N[0]*L[0] + N[1]*L[1] + N[2]*L[2])
    if twoCosPhi > 0:
        for i in range(3):
            R.append(N[i] - (L[i] / twoCosPhi))
    elif twoCosPhi == 0:
        for i in range(3):
            R.append( - L[i])
    else:# twoCosPhi < 0
        for i in range(3):
            R.append( - N[i] + (L[i] / twoCosPhi))
    return normalize(R)

# Convert an N dimensional vector into a unit vector (i.e., normalize the vector)
def normalize(vector):
    sumOfSquares = 0
    for i in range(len(vector)):
        sumOfSquares += vector[i]**2
    magnitude = math.sqrt(sumOfSquares)
    vect = []
    for i in range(len(vector)):
        vect.append(vector[i]/magnitude)
    return vect
```

16.5 The Complete Illumination Model: 20 Spheres Example

In this section we put all the pieces of the Phong illumination model together and illustrate the illumination process with an end-to-end example, implemented in Python.

Monochrome single source Phong Illumination Model can be computed as:

$$Intensity = I_a \cdot K_d + I_p \cdot K_d \cdot \frac{(N \cdot L)}{d} + I_P \cdot K_s \cdot (R \cdot V)^{specIndex}$$

In the above, the ambient reflection component is shown in green, the diffuse reflection component in blue, and the specular reflection component in red.

RGB single source Phong Illumination Model can be computed as:

$$Intensity_{red} = I_{a_{red}} \cdot K_{d_{red}} + I_{p_{red}} \cdot K_{d_{red}} \cdot \frac{(N \cdot L)}{d} + I_{P_{red}} \cdot K_{s_{red}} \cdot (R \cdot V)^{specIndex}$$

$$Intensity_{green} = I_{a_{green}} \cdot K_{d_{green}} + I_{p_{green}} \cdot K_{d_{green}} \cdot \frac{(N \cdot L)}{d} + I_{P_{green}} \cdot K_{s_{green}} \cdot (R \cdot V)^{specIndex}$$

$$Intensity_{blue} = I_{a_{blue}} \cdot K_{d_{blue}} + I_{p_{blue}} \cdot K_{d_{blue}} \cdot \frac{(N \cdot L)}{d} + I_{P_{blue}} \cdot K_{s_{blue}} \cdot (R \cdot V)^{specIndex}$$

Monochrome multiple source Phong Illumination Model can be computed as:

$$Intensity = I_a \cdot K_d + \sum_{i=1}^{n} \left(I_{P_i} \cdot K_d \cdot \frac{(N \cdot L_i)}{d_i} + I_{P_i} \cdot K_S \cdot (R_i \cdot V)^{specIndex} \right)$$

Once again to clearly illustrate each of the three parts of the model, the ambient reflection component is shown in green, the diffuse reflection component in blue, and the specular reflection component in red.

RGB multiple source Phong Illumination Model can be computed as:

$$Intensity_{red} = I_{a_{red}} \cdot K_{d_{red}} + \sum_{i=1}^{n} \left(I_{P_{i_{red}}} \cdot K_{d_{red}} \cdot \frac{(N \cdot L_i)}{d_i} + I_{P_{i_{red}}} \cdot K_{S_{red}} \cdot (R_i \cdot V)^{specIndex} \right)$$

$$Intensity_{red} = I_{a_{red}} \cdot K_{d_{red}} + \sum_{i=1}^{n} \left(I_{P_{i_{red}}} \cdot K_{d_{red}} \cdot \frac{(N \cdot L_i)}{d_i} + I_{P_{i_{red}}} \cdot K_{S_{red}} \cdot (R_i \cdot V)^{specIndex} \right)$$

$$Intensity_{red} = I_{a_{red}} \cdot K_{d_{red}} + \sum_{i=1}^{n} \left(I_{P_{i_{red}}} \cdot K_{d_{red}} \cdot \frac{(N \cdot L_i)}{d_i} + I_{P_{i_{red}}} \cdot K_{S_{red}} \cdot (R_i \cdot V)^{specIndex} \right)$$

Before moving on to an end-to-end example, it’s important to take a moment to point out some of the limitations of the Phong illumination model.

First, as mentioned earlier, **the Phong model cannot calculate true reflections**, if one were to faithfully model a mirror with Phong reflection, instead of a reflection all you would get is a white light.

Second, **the model does not account for shadowing**. In other words, if there are objects between the light source and the object of interest, it will be rendered as if the other objects are not present. If the object of interest has a complex shape, like a valley between two mountains, the valley will be illuminated as if the mountains do not exist.

While Phong can be made to work well for most real-world materials by carefully selecting K_S and *specIndex*, **there are some real world substances like glass, and water, and even gold, for which Phong is not well suited**.

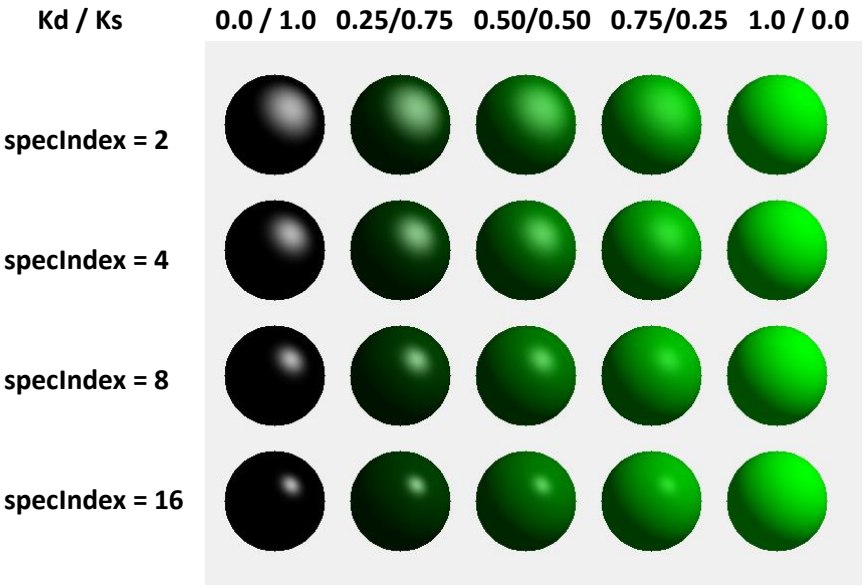
The root problem is that K_S is a simplification that ignores the fact that the reflectivity of certain materials is based, in part, on the incident angle. In other words, for some materials the K_S term is not a constant; it is a function that varies specular reflectivity with the angle of the incoming light rays.

This is obvious if you think about the water. When the angle between the surface normal of a still pond and the lighting vector is 0, the light is directly overhead and passes through the water with *no reflection* being generated. However, when light rays hit the water at a very shallow angle the water become mirror-like, reflecting almost all of the light hitting its surface. This is why even when the sun is directly overhead, if the water in a pond is not still, light will glint off the waves causing the water to sparkle. Gold “glisters” for the same reason, its specular reflectivity is a function of incident angle.

Even saying that specular reflectivity depends on the incident angle is a simplification, as in reality specular reflectivity depends on both the incident angle and the wavelength of the incoming light. This means that in addition to angle, different colors of light will produce different specular reflections.

Despite these limitations, Phong illumination produces good looking results in most instances and unlike illumination models based more closely on the underlying physics of light, it is vastly more affordable.

We are now ready to move on to our end-to-end example. Here is an example of twenty spheres illustrating the effect of various *specIndex* values, and K_d / K_S ratios.



Twenty Spheres Illustration (K_d / K_S ratios with varying specular indices)

The *render20Spheres* method shown on the following page renders twenty spheres, each with a radius of 50 pixels using the Phong illumination model. The code generates four rows of spheres, each with a particular specular index. On row one the specular index is 2, on row two the specular index is 4, on row three 8, and on row four 16. Five spheres are rendered on each row. Each column of the illustration feature a different Kd/Ks ratio (constant of diffuse reflectivity / constant of specular reflectivity.

The environmental characteristic in which each sphere is rendered are fixed. The intensity of ambient light is 30% of the overall lighting ($I_a = 0.3$). The intensity of the single point light source illuminating each sphere is 70% ($I_p = 0.7$). Light from the point source is striking each sphere at a 45˚ angle – the point source is located behind the viewer's right shoulder ($L =< 1,1,1 >$). The viewer is looking directly at the center of each sphere from a point in +Z ($V =< 0,0,1 >$). The surface normal for the spheres (N) will vary smoothly across their surfaces and will be computed in the code.

Note that the viewing system is assumed to be right handed – which is different from the left hand system we have been consistently using to date. The viewing system used in this example can be switched to the more familiar left handed system by flipping the sign of the Z-axis components in the L, V, and N vectors. The right handed system was used in this particular example as it was deemed more intuitive. We are not abandoning our left hand system, and will continue to use it elsewhere, unless specifically noted in the text.

```
import math
from tkinter import *

CanvasWidth = 800
CanvasHeight = 600

Ia = .3  # intensity of the ambient light in the scene
Ip = .7  # intensity of the point light source in the scene
L = [1,1,1] # Lighting vector, 45 degree angle, light is behind viewer's right shoulder [Right Hand Viewing System]
V = [0,0,1] # View vector, points towards viewer / center of projection [Right Hand Viewing System]

def render20Spheres(Ia, Ip, L, V):
    # Sphere Characteristics
    xCenter = 150  # xCenter of first sphere on each row
    yCenter = 112  # yCenter of first sphere on FIRST row
    radius = 50
    SpecIndex = [2, 4, 8, 16]
    Kd = 0   # Kd of first sphere on each row
    Ks = 1   # Ks of first sphere on each row
    # There will be four rows of spheres
    for row in range(4):
        # There will be five spheres per row
        for col in range(5):
            # Paint a sphere
            renderSphere(xCenter, yCenter, radius, Kd, Ks, SpecIndex[row], Ia, Ip, L, V)
            # Prep for next sphere on row
            Kd += 0.25
            Ks -= 0.25
            xCenter += 125
        # Prep for next row of spheres
        xCenter = 150   # reset xCenter
        yCenter += 125  # increment yCenter for next row
        Kd = 0          # reset Kd
        Ks = 1          # reset Ks
```

The *render20Spheres* method

```
def renderSphere(xCenter, yCenter, radius, Kd, Ks, specIndex, Ia, Ip, L,V):
    # insure vectors passed to renderSphere are normalized unit vectors
    L = normalize(L)
    V = normalize(V)
    # ambient diffuse component of illumination model
    ambient = Ia * Kd
    # render a sphere
    rSquare = radius**2
    for y in range (-radius, radius+1):
        ySquare = y**2
        for x in range(-radius, radius+1):
            xSquare = x**2
            if (xSquare + ySquare) <= rSquare:
                # x**2 + y**2 + z**2 = r**2
                z = round(math.sqrt(rSquare - xSquare - ySquare))
                N = normalize([x,y,z])
                NdotL = N[0]*L[0] + N[1]*L[1] + N[2]* L[2]
                if NdotL < 0: NdotL = 0
                diffuse = Ip * Kd * NdotL
                R = reflect(N,L) # return vector is normalized in "reflect"
                RdotV = R[0]*V[0] + R[1]*V[1] + R[2]*V[2]
                if RdotV < 0: RdotV = 0
                specular = Ip *Ks * RdotV**specIndex
                color = triColorHexCode(ambient, diffuse, specular)
                # -y instead of +y since y is upside down in Tkinter display coordinates
                w.create_line(xCenter+x, yCenter-y, xCenter+x+1, yCenter-y, fill=color)
```

The *renderSphere* method

The *renderShpere* method shown on the previous page is the core code for rendering a single sphere. The radius of the sphere is 50 units (pixels). The code assumes the sphere is centered at the origin $(0,0,0)$, in a right handed system, with positive Z towards the viewer. The surface normal for each pixel on the surface of the sphere is calculated. Using a pixel's surface normal, N, together with the view vector, V, the lighting vector L, and the reflection vector, R; the single source monochrome Phong illumination model is used to compute reflected intensity values for the ambient, point diffuse, and point specular reflection components.

This code utilizes the *reflect* method presented in Section 16.4.2. It also depends on the *triColorHexCode* method we discuss below.

In addition to showing a complete example of Phong illumination, this code illustrates a new way of thinking about object definitions. Throughout this course, we have focused on objects defined as collections of planar polygons. In this program we ***compute*** the surface of an object, a sphere, from the underlying equation of a sphere.

A sphere is all points that are radius distance from the sphere's center point. The definition of a 3-D sphere is essentially the 2-D definition of a circle extended to three dimensions. Thus, the equation $r^2 = x^2 + y^2 + z^2$ will always hold for a sphere whose center is the origin. For any point (x, y) we can determine whether (x, y) is a point on the surface of the sphere by testing to see if $x^2 + y^2 \leq r^2$. If so, we can find the z value of the surface normal on the sphere at the (x, y) point from $z = \sqrt{r^2 - x^2 - y^2}$.

You may be wondering why we don't use this approach for defining objects more often in computer graphics. The answer is that we while we do use equations for planes and spheres and cubes and other objects with easy to define formulas, most objects in the real world are highly complex. There is no "easy to define formula" to describe a human face, let alone a particular human face. We are thus forced to approximate complex shapes using many small polygons. In the next chapter we will explore ways of making our "approximations" of complex shapes look better with a technique called **shading**, that uses color to more realistically render a polygonal object.

The last major piece of the 20 sphere's example is presented on the next page: the *triColorHexCode* method and the *colorHexCode* method. These methods are specific to our Python / TKinter solution. Since TKinter does not provide a way to directly access individual pixels, we have been forced to emulate a pixel set command with `create_line()` which takes in an x, y starting point; an x, y ending point; and a color string `fill = color`. The color string can be a named color (e.g. "red") or a hexadecimal number formatted as a string, such as "`#ffffff`". The first two hex characters (after the #, represent the intensity of red (0-255); the next two hex characters represent the intensity of blue (0-255); and the last two hex characters specify the intensity of red (0-255).

The *colorHexCode* method takes as input an $intensity$, which should be in the range 0-1, and returns a $trimmedHexString$ exactly two hex characters long.

The *triColorHexCode* method takes in the *ambient*, *diffuse*, and *specular* components of the model (which should all add to 1) and, with the help of *colorHexCode*, constructs the six digit hex character string needed by `create_line()`. All three illumination components are summed together and placed in the green channel. Only the specular component is placed in the red and blue channels. Thus, the specular highlight is rendered as white light, while the diffuse components are rendered in green.

```
# generate a color hex code string from the illumination components
def triColorHexCode(ambient, diffuse, specular):
    combinedColorCode = colorHexCode(ambient + diffuse + specular)
    specularColorCode = colorHexCode(specular)
    colorString = "#" + specularColorCode + combinedColorCode + specularColorCode
    return colorString

def colorHexCode(intensity):
    hexString = str(hex(round(255 * intensity)))
    if hexString[0] == "-":  # illumination intensity should not be negative
        print("illumination intensity is Negative.  Setting to 00. Did you check for negative NdotL?")
        trimmedHexString = "00"
    else:
        trimmedHexString = hexString[2:] # get rid of "0x" at beginning of hex strings
        # convert single digit hex strings to two digit hex strings
        if len(trimmedHexString) == 1: trimmedHexString = "0"+ trimmedHexString
        # we will use the green color component to display our monochrome illumination results
    return trimmedHexString
```

The methods used to generate the color hex code string from the individual illumination components

The final bit of code is needed to define the drawing canvas in TKinter, call render20Spheres, and kick off the event loop in order to respond to interface objects (e.g., buttons etc.), of which there are none in this simple program. Even without an interface to support, the mainloop is necessary to keep the program from immediately terminating as soon as the 20 spheres are rendered, which would prevent us from appreciating the image resulting from all of our hard work.

```
# Define a drawing canvas and render the 20 spheres on it
root = Tk()
outerframe = Frame(root)
outerframe.pack()
w = Canvas(outerframe, width=CanvasWidth, height=CanvasHeight)
render20Spheres(Ia, Ip, L, V)
w.pack()
root.mainloop()
```

Python TKinter housekeeping code required to render the 20 spheres

17. Polygon Shading

So far we have talked about "point" illumination / reflection models and have applied such a model (Phong) to a sphere. A sphere is a smooth non-planar object, but since the equations underlying it are simple it can be easily rendered directly from its underlying definition.

Most objects in the real world have shapes that are far from "simple". No equations exist (at least not simple, easy to compute equations) that allow the shape of the object to be computed. Instead we represent objects as collections of planar polygons. These polygons approximate the object's shape.

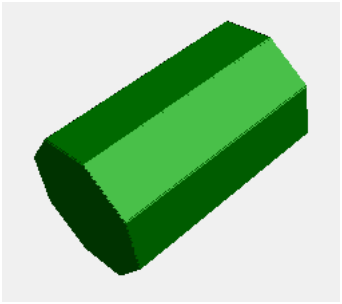
Typically, the larger the number of polygon we have to work with and the smaller the size of the individual polygons, the closer our polygonal approximation will be to the shape of the underlying object we are modeling. Regardless of the size of the polygons used, as long as each polygon is larger than the size of our pixels, the fact that our object is an approximation will be apparent to the viewer.

There is a natural tension between our desire for realism (which pushes us to use more and smaller polygons) and the limited computational budgets we have available (which pushes us to use fewer and larger polygons). This has driven the search for methods of "visually hiding" the underlying polygonal structure of graphical objects without moving to unaffordably detailed polygonal meshes.

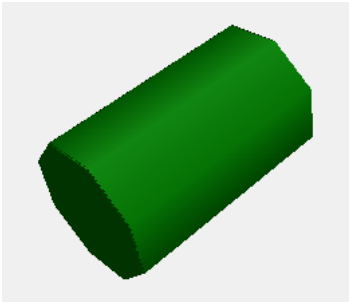
Shading is a graphical technique that consists of applying a point illumination / reflection model, such as the Phong illumination model, to a polygonal mesh in such a way as to render the approximated object realistically. A good shading technique aims to achieve three goals:

1. giving the polygonal mesh a solid 3-D appearance,
2. decreasing the visibility of the polygonal mesh, and
3. realistically rendering specular highlights

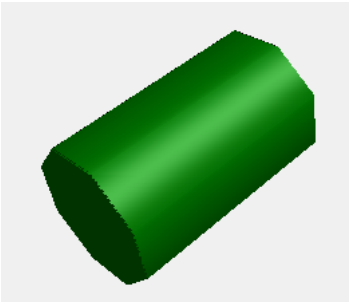
We will cover three shading techniques, each progressively more realistic and expensive.



Flat Shading



Gouraud Shading



Phong Shading

The first shading technique we will examine is flat shading. Flat shading is inexpensive and achieves the first goal of making the polygonal mesh appear solid.

The second shading technique we will look at is Gouraud shading. Gouraud is moderately expensive and achieves the first and second goals: the object will appear solid and smooth.

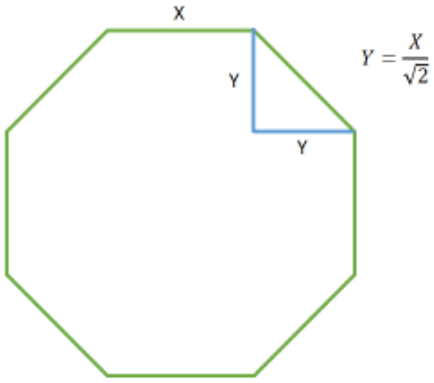
The third and final shading technique we explore is called Phong shading. Phong shading is the most expensive of the three, but it is also the best looking; able to achieve all three goals, including realistically rendering specular highlights on the polygonal mesh.

17.1 Flat Shading

Flat shading is simple and easy to implement. **Flat shading** computes the color of ONE point on a surface of a polygon, typically using the Phong illumination model, and then paints the entire polygon using that single color. This shading technique is sometimes called "faceted" shading, since curved surfaces are approximated with many small flat surfaces, like the facets of a diamond. Another name it occasionally goes by is "Lambert" shading, since the illumination model used to compute the color of each polygon usually incorporates Lambert's Cosine Law (as in the point diffuse component of the Phong illumination model) to light the polygon's appropriately.

Before we can dig into the details of flat shading (or any other type of shading) we need a polygonal approximation of a smoothly curved object. Let's define a polygonal approximation of the cylindrical object illustrated on the previous page.

The lengthwise dimension of the cylinder is approximated by eight rectangles, which we will number from 0 to 7. We can generated these eight rectangular polygons by defining an octagon and "extruding" it into 3-D space.



If we let $X = 100$ then $Y = 70.7107$. Centering the octagon at $0,0$ in X-Y with a Z of 50 for the near end of the cylinder and 450 for the far end of the cylinder gives:

Cylinder Vertices

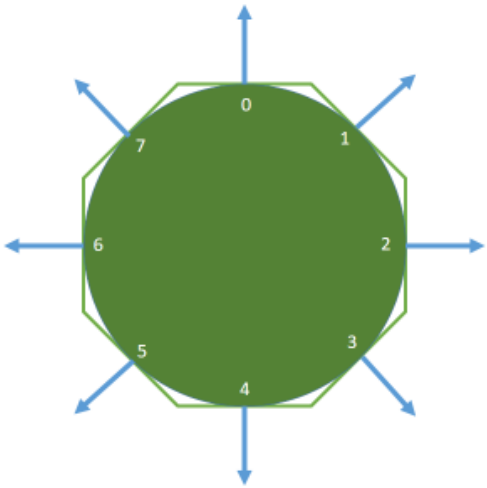
	X	Y	Z
Front 0	-50.0000	120.7107	50.0000
Front 1	50.0000	120.7107	50.0000
Front 2	120.7107	50.0000	50.0000
Front 3	120.7107	-50.0000	50.0000
Front 4	50.0000	-120.7107	50.0000
Front 5	-50.0000	-120.7107	50.0000
Front 6	-120.7107	-50.0000	50.0000
Front 7	-120.7107	50.0000	50.0000
Back 0	-50.0000	120.7107	450.0000
Back 1	50.0000	120.7107	450.0000
Back 2	120.7107	50.0000	450.0000
Back 3	120.7107	-50.0000	450.0000
Back 4	50.0000	-120.7107	450.0000
Back 5	-50.0000	-120.7107	450.0000
Back 6	-120.7107	-50.0000	450.0000
Back 7	-120.7107	50.0000	450.0000

Given this table of vertices, we define our eight rectangles starting with the top rectangle, polygon 0, with its vertices in clockwise order: Front 0, Back 0, Back 1, Front 1. Each polygon is in turn defined clockwise: top 0, top-right 1, right 2, bottom-right 3, bottom 4, bottom-left 5, left 6, top-left 7. From the first three points of each polygon two vectors, P and Q, can be generated and from the cross product of P and Q the surface normals can be computed.

	Surface Normals (unit)		
	X	Y	Z
Polygon 0	0	1	0
Polygon 1	0.707107	0.707107	0
Polygon 2	1	0	0
Polygon 3	0.707107	-0.70711	0
Polygon 4	0	-1	0
Polygon 5	-0.70711	-0.70711	0
Polygon 6	-1	0	0
Polygon 7	-0.70711	0.707107	0
Polygon 8	0	0	-1
Polygon 9	0	0	1

Note that the above table has a total of ten polygons, rather than eight. Polygon 8 is the Front "end cap" of the cylinder and Polygon 9 is the Back "end cap" of the cylinder.

Flat shading requires the computation of one surface normal per polygon and one illumination computation per polygon.



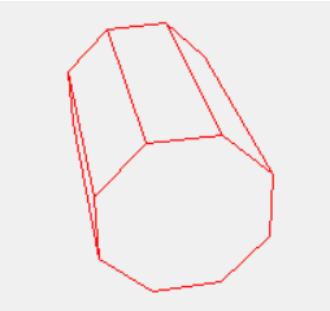
The above figure illustrates, in cross section, both our polygonal representation of the cylinder and the original cylinder we are approximating. The blue arrows represent the surface normals of each polygon. As can be seen from the figure, the surface normal of a polygon will be identical to the surface normal of the actual cylinder at the center of each polygon.

Thus, at the center point of each polygon the illumination intensity calculated from the polygon surface normal and lighting vector matches what would be returned by the underlying cylindrical object. As we move away from the center point of a polygon, that polygon's surface normal is farther and farther from the underlying object's actual surface normal, and thus the flat shaded color of the polygon is less and less accurate the farther a point is from the polygon's center.

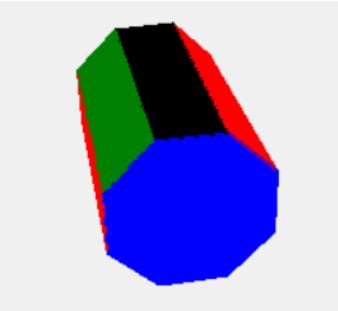
If the lighting vector is constant – in other words, if all rays from the light source are considered to be parallel – you don't have to worry about selecting an x, y location on the polygon for the illumination computation, as all points on the polygon will return the same result (since they have the same surface normal). If the rays from the point source are not assumed to be parallel, then illumination should be computed at the center of the polygon using a computed lighting vector from that center point back to the light source.

Essentially, flat shading computes the underlying object's illumination intensity at one point per polygon (where the polygon's surface normal matches the underlying object's surface normal) and then using this single sample "assumes" that all other points on that polygon share the same intensity.

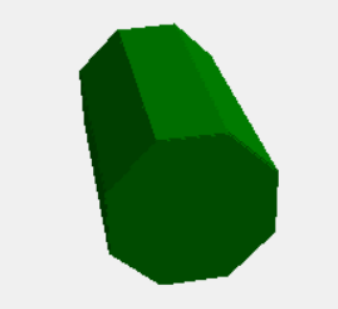
Flat shading is much more realistic than rendering a polygonal object as a wireframe, since it gives the polygonal object a "solid" appearance. Flat shading is also generally more realistic than painting each polygon with a separate color chosen by the software developer – as doing so gives the object a 3-D look but it also a "cartoon like" appearance, disconnected from any environmental lighting.



WireFrame



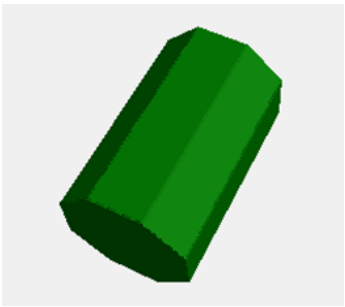
Basic Polygon Fill



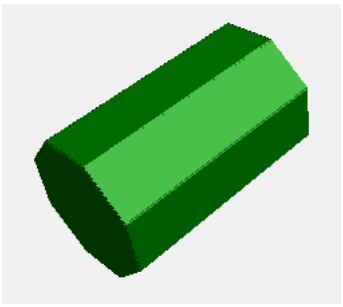
Flat Shading

The biggest disadvantage of flat shading is that it makes the underlying polygon mesh highly visible – the opposite of what an optimal shading algorithm should do.

Flat shading also does a poor job rendering specular highlights. Essentially, either the highlight is totally missed, or the entire polygon is painted with the highlight (illustrated below). In fact, when animating a flat shaded object– such as an animation of an object rotating around the various axes – the specular component of the illumination model is often turned off ($K_S = 0$ and/or $specIndex = 0$) to prevent annoying "flashes" as various polygon surface normals happen to land in the specular highlight.



Specular Highlight Totally Missed



Specular Highlight Spread Across Entire Polygon

17. 2 Gouraud Shading

Gouraud shading, named after French computer scientist Henri Gouraud (1944-), was the first serious attempt to overcome the disadvantages of flat polygon shading. The technique was first described in 1971.

The general idea behind **Gouraud shading** is that given a polygonal representation of a curved surface:

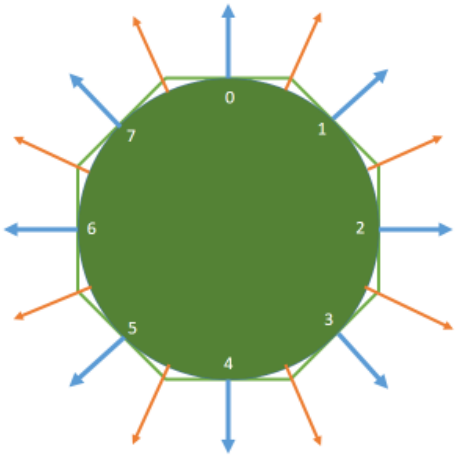
(1) First compute the surface normals at each vertex of each polygon making up the object,

(2) Apply an illumination model at each vertex utilizing that vertex's normal, and then

(3) Perform a bi-linear interpolation of the vertex illumination intensity values in order to paint the polygons with an intensity that varies smoothly across the surface of each polygon.

In order to implement Gouraud shading, the first thing we have to get a handle on is determining how we compute the vertex normals. We begin by computing the surface normal for each polygon – exactly as we did for flat shading. Given the surface normals we can compute the vertex normals.

For a smoothly curved surface, like the lengthwise dimension of a cylinder, which we have approximated by eight rectangular polygons, we ADD together the surface normals of the two rectangular polygons that adjoin one another.



For example, using the cylindrical object definition described in Section 17.1 above, polygon 0 would have four vertices: [-50,120.7107,50], [-50,120.7107,450], [50,120.7107,450], [50,120.7107,50]. The normals for the first two vertices can be computed by adding together the polygon 7 and polygon 0 surface normals (and then converting the result to a unit vector). The vertex normals for the last two vertices of polygon 0 are obtained by adding together the surface normals for polygon 0 and polygon 1 (and then converting to a unit vector). The vertex normals (as unit vectors) for the four vertices of polygon 0 are presented in the following table. The vertex normals for the other rectangles are computed similarly.

Polygon 0 Vertices Normals

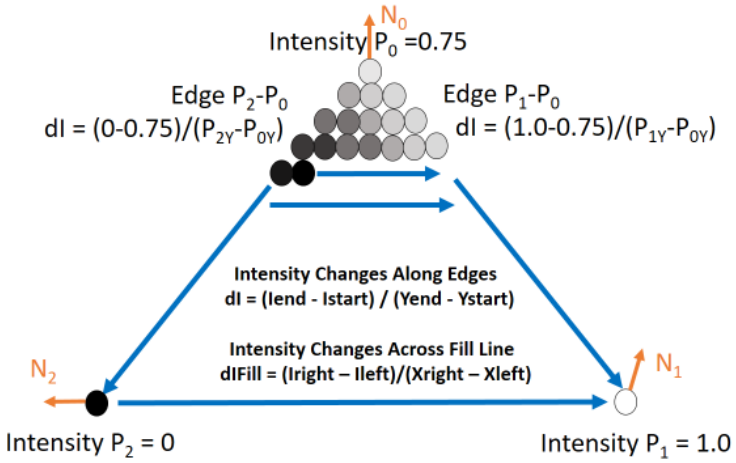
	X	Y	Z
Front 0	-0.3827	0.9239	0.0000
Back 0	-0.3827	0.9239	0.0000
Back 1	0.3827	0.9239	0.0000
Front 1	0.3827	0.9239	0.0000

It's important to note that the vertex normals are associated with individual polygons. A vertex may have different normals associated with it depending on the polygon being considered. For example, the front and back "end caps" of the cylinder share the vertices used to define polygons 0-7, but when considering polygon 8 (front end cap) or polygon 9 (back end cap), we do not want to compute "blended" vertex normals. Instead we want all of the end cap vertex normals to equal the polygon's surface normal. Thus, when our approximation of the cylinder is in the orientation shown above all vertex normals for the front end cap should be $< 0,0,-1 >$ and for the back end cap $< 0,0,1 >$.

Now that we know how to compute the necessary vertex normals, the next step is to simply apply the Phong illumination model at each of the vertices of a polygon (using the vertex normals). This produces an illumination intensity at each vertex.

The final step of Gouraud shading interpolates these intensity values smoothly across the face of the polygon.

The equations for Gouraud shading intensity interpolations are very similar to the equations for computing Z buffer values. In fact, the equations for Gouraud shading and Z-buffer interpolation are IDENTICAL if you swap out "Intensity" for "Z".



Interpolating Illumination Intensity values under Gouraud shading

As with Z-buffer computations, the Gouraud shading computations will take place in the polygon fill method. Just as we computed a *ΔZ* value to specify how much an edge's Z value should change each time we moved down a fill line, the Gouraud equations utilize a *ΔIntensity* that captures how much an edge's illuminated intensity will change from pixel to pixel each time we move down a fill line.

$$\Delta Intensity = \frac{Intensity_{end} - Intensity_{Start}}{Y_{end} - Y_{start}}$$

We will use a "bi-linear interpolation" technique to generate pixel illumination intensity values. First, we compute the illumination intensity values of the starting pixel and ending pixel on the current fill line – the pixels on the two active edges where they cross the current fill line.

When considering an edge, we begin with the illumination intensity value for the pixel at the starting vertex of that edge. Each time we move down a fill line we add the *ΔIntensity* value for the edge to the

illumination intensity value of the edge pixel on the previous fill line. This gives us the illumination intensity value for this edge's pixel on the next fill line. We perform these calculations for both active edges. We then use the pixel illumination intensity values derived from the two active edges as the starting and ending pixel illumination intensity values for the current fill line.

Once we have the pixel illumination intensity values for the starting and ending pixel on the current fill line, we can then generate illumination intensity values for each pixel across the fill line.

The **change in pixel illumination intensity for the Left edge as we move from horizontal fill line to horizontal fill line** is given by:

$$\Delta Intensity_{Edge_{Left}} = \frac{IntensityEnd_{Edge_{Left}} - IntensityStart_{Edge_{Left}}}{Yend_{Edge_{Left}} - Ystart_{Edge_{Left}}}$$

The **pixel illumination intensity value for the Left edge on the first horizontal fill line**:

$$Intensity_{Edge_{Left}\, FillLine_0} = IntensityStart_{Edge_{Left}}$$

The **pixel illumination intensity values for the Left edge on the subsequent horizontal fill lines**:

$$Intensity_{Edge_{Left}\, FillLine_{i+1}} = Intensity_{Edge_{Left}\, FillLine_i} + \Delta Intensity_{Edge_{Left}}$$

The **change in pixel illumination intensity for the Right edge as we move from horizontal fill line to horizontal fill line** is given by:

$$\Delta Intensity_{Edge_{Right}} = \frac{IntensityEnd_{Edge_{Right}} - IntensityStart_{Edge_{Right}}}{Yend_{Edge_{Right}} - Ystart_{Edge_{Right}}}$$

The **pixel illumination intensity value for the Right edge on the first horizontal fill line**:

$$Intensity_{Edge_{Right}\, FillLine_0} = InstensityStart_{Edge_{Right}}$$

The **pixel illumination intensity values for the Right edge on the subsequent horizontal fill lines**:

$$Intensity_{Edge_{Right}\, FillLine_{i+1}} = Intensity_{Edge_{Right}\, FillLine_i} + \Delta Intensity_{Edge_{Right}}$$

The above equations provide the illumination intensity values for the starting pixel (left edge) and ending pixel (right edge) on the current fill line. As with Z-buffer computations, both the *IntensityStart* and *ΔIntensity* for each edge can be pre-computed and stored in the Polygon Fill Table.

The next set of equations compute the pixel illumination intensity values across the current horizontal fill line.

The **change in pixel illumination intensity as we move across the horizontal fill line from pixel to pixel**:

$$\Delta Intensity_{FillLine_i} = \frac{Intensity_{Edge_{Right}FillLine_i} - Intensity_{Edge_{Left}FillLine_i}}{X_{Edge_{Right}FillLine_i} - X_{Edge_{Left}FillLine_i}}$$

This equation requires the pixel illumination intensity values of the active edges on the current fill line. It also utilizes the X values of the left and right edges on the current fill line to compute the "number of steps" over which the difference in illumination intensity values between the two active edges on the current fill line must be spread. For these reasons, this equation cannot be pre-computed and stored in the Polygon Fill table – its inputs and the result it produces changes from horizontal fill line to horizontal fill line.

The **pixel illumination intensity value for the first pixel on the current horizontal fill line** is given by:

$$Intensity_{FillLine_i\ Pixel_0} = Intensity_{Edge_{Left}FillLine_i}$$

The **subsequent illumination intensity values for each pixel on the current horizontal fill line** are:

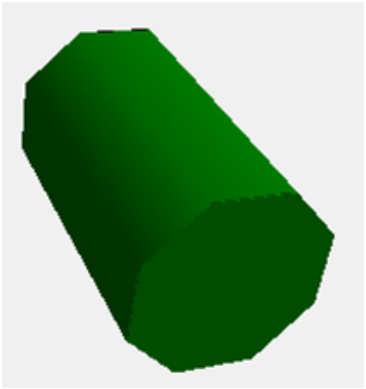
$$Intensity_{FillLine_i\ Pixel_{j+1}} = Intensity_{FillLine_i\ Pixel_j} + \Delta Intensity_{FillLine_i}$$

These equations provide us with the pixel illumination intensity values for every pixel on the polygon.

It should be noted that the above equations treat *Intensity* as a single value. In our 20 spheres example, provided in Section 16.5, pixel illumination intensity was actually held as three separate values *ambient*, *diffuse*, and *specular* intensity – which were combined to produce an RGB hex code color string immediately before painting the pixel. When implementing Gouraud shading, you may find it beneficial to continue to represent intensity as these three separate components. In which case all of the above nine equations would be repeated twice, once for diffuse intensity, and once for specular intensity – you could also repeat these nine equations for ambient intensity, but that is not really necessary as every pixel on every polygon of an object should generate the same value – all of the Δ's will be zero.

Another "enhancement" to the above procedure for computing Gouraud shading would be to repeat all nine of the equations for each of the three primary colors: red, green, and blue. This is necessary in order to support "full color" RGB Gouraud shading.

The end result of Gouraud shading looks rather good. Here is our approximation of the cylinder rendered with monochrome Gouraud shading, with intensities mapped to the green color channel.



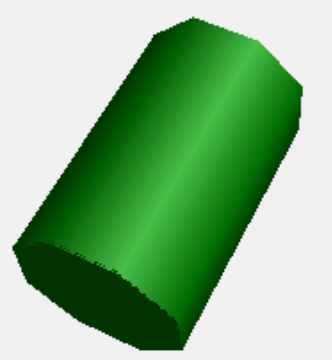
Gouraud Shading

Gouraud gives our low polygon count approximation of a cylinder a solid appearance and along the body of the cylinder provides a smooth surface. If it were not for the front and rear edges, our polygonal approximation would be near-identical to an actual cylinder.

Gouraud works so well because it captures the true illumination intensities of the actual cylinder at each vertex and then smoothly "blends" these intensities across the face of the polygon. Gouraud shading is also relatively inexpensive. It requires computing a surface normal for each polygon, and from those surface normals computing vertex normals for each vertex of each polygon. Our octangular polygonal mesh consists of ten polygons (eight rectangles and two octagons), requiring the computation of a surface normal for each (10 surface normals). From these ten surface normals we must compute vertex normals for the eight rectangles each with four vertices (32 vertex normals) plus two octagons, each with eight vertices (16 vertex normals). All together, we must compute 58 normals for our polygon mesh. At each of the 48 vertex normals we compute illumination intensity (which is a rather expensive computation). We then perform linearly interpolations of these illumination intensities across the polygons, which is relatively inexpensive.s are performed only at the vertices of each polygon.

Gouraud shading does suffer from some limitations, (1) including: poor treatment of specular highlights, (2) the Mach band effect, and (3) lack of shadowing.

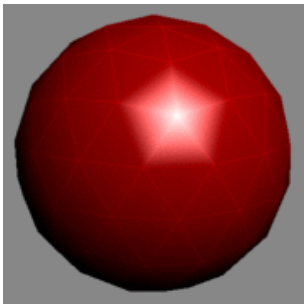
Specular highlights: As with flat shading, Gouraud shading either entirely misses specular highlights or spreads them over adjacent highlights in an irregular manner. For our cylinder approximation the results don't look too bad in a still image, such as the one below.



Gouraud Shading rendering a specular highlight incorrectly

The main anomaly with our cylinder approximation is that the specular highlight is spread far wider than it should be. The problem would be more noticeable if the cylinder (or light source) were animated as you'd get a flickering effect as vertices "randomly" catch a specular highlight.

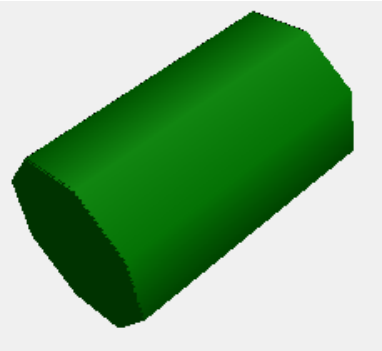
We can get a better appreciation of how poorly Gouraud shading handles specular reflections in general by looking at a polygonal approximation of a sphere.



Specular highlight rendered irregularly by Gouraud shading
[https://en.wikipedia.org/wiki/Gouraud_shading]

The specular highlight ends up with a pentagonal appearance, as opposed to being round, which highlights the polygonal mesh. Additionally if we were to animate the sphere so that it rotates, these weirdly shaped highlights would flash on and off as the polygon vertices happen to catch the highlight.

The Mach band effect: The human visual system is very good as noticing changes in rates of change. This helps our visual system detect edges – which certainly has evolutionary advantages. Unfortunately, the effect, called the Mach band effect after the German physicist Ernst Mach (1838-1916) who discovered this optical illusion, can make the polygonal mesh underlying our objects more apparent than they would otherwise be.



Gouraud shaded polygonal approximation of a cylinder showing Mach band effect

We can clearly see the "edges" of the polygons in the above rendering where the rate of change in illumination intensities occurs.

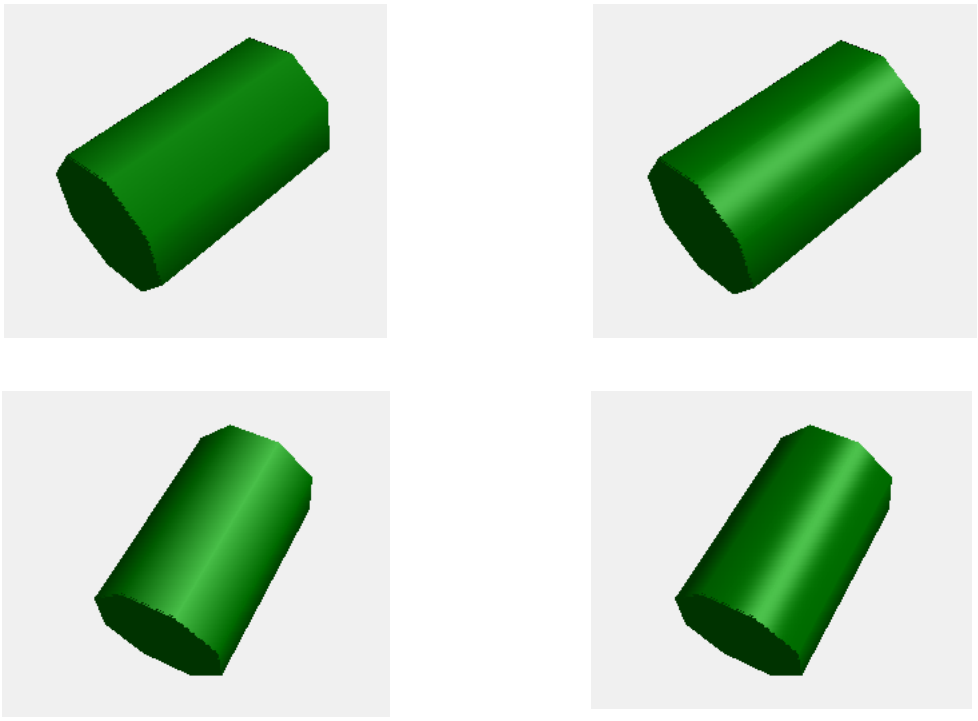
Lack of shadowing: A final problem with Gouraud shading we'll mention is the complete lack of support for shadowing in Gouraud shading. This problem is more accurately assigned to the Phong illumination / reflection model that we are using to compute our intensities than Gouraud shading itself – though if our illumination model did handle shadowing Gouraud would "bork" their rendering in the same way that it mishandles specular highlights. (The shadowing would be irregularly applied to the polygon mesh.)

The reason for the lack of shadowing is that when we are given the lighting vector, L, we simply plug it in to the Lambert cosine law to determine the amount of point light hitting the surface. The model doesn't attempt to determine whether some other object exists between the object we are rendering and the light source, even though in the real world if there were an object between the object we are rendering and the light source, none of the (direct) light from the source would reach the surface of our object – it would be in shadow.

In complicated multi-object scenes the lack of shadowing greatly decreases realism.

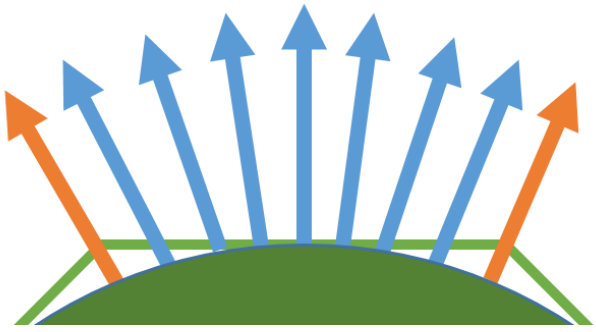
17. 3 Phong Shading

The third shading technique we'll explore is Phong shading. Phong shading (which is distinct from Phong illumination) improves on Gouraud shading to realistically render specular highlights. It also overcomes the Mach band effect. Standard Phong shading (which uses the Phong illumination model) does not, however, solve the shadowing problem, though if we used an illumination model that detected occlusion between the surface we are rendering and the light source, Phong shading could be upgraded to overcome the shadowing limitation.



Gourard Shading versus **Phong Shading**

The reason Phong shading can render what looks like a curved surface so realistically, even with a low polygon count polygonal mesh is that, though the individual polygons are still flat, the color (intensity) of each pixel is computed using a surface normal at that pixel that is identical / near identical to what the surface normal would be on the underlying object.



Interpolating surface normals of the underlying object from the vertex normals

The computations for Phong shading are conceptually quite similar to those required to implement Gouraud shading, except that instead of interpolating intensity values, Phong shading interpolates surface normals.

Phong shading works in the following way. Given a polygonal representation of a curved surface:

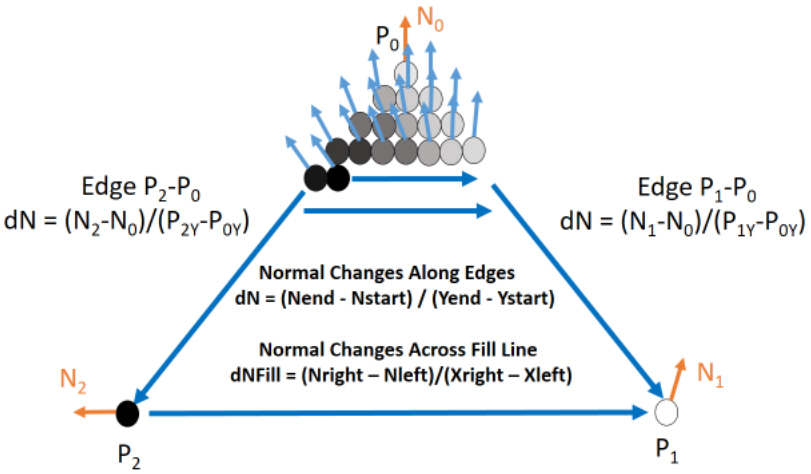
(1) First compute the surface normals at each vertex of each polygon making up the object,

(2) For each pixel to be painted on each polygon:

(2.1) Perform a bi-linear interpolation of the vertex normals, in order to derive a surface normal for the pixel.

(2.2) Apply an illumination model to compute an illumination intensity for the pixel utilizing the pixel's surface normal

(2.3) Paint the pixel with the illumination intensity value



Interpolating surface normals under Phong shading

Though the algorithm for Phong shading is similar to the algorithm for Gouraud shading; Phong shading is much more expensive due to the separate illumination intensity computation performed for each visible pixel (using the pixel's interpolated surface normal) on every visible surface of the polygon mesh to be rendered. Remember though, you eliminate the Mach band effect and realistically render specular highlights with Phong shading, so the visual end result, especially for "metallic" objects looks more realistic with Phong shading.

We can now work our way through the equations for computing Phong shading. The first step is to determine how to compute the surface normals along the active edges. Just as we did with Z value interpolations for the Z-buffer algorithm, and illumination intensities for Gouraud shading, we can pre-compute the amount that the surface normal will change along a polygon's edge from fill line to fill line. The general formula is:

$$\Delta N = \frac{N_{end} - N_{start}}{Y_{end} - Y_{start}}$$

Since N is a vector with X, Y, and Z components, the above equation actually breaks down into three separate equations, one for the X component of N, one for the Y component of N, and one for the Z component of N.

$$\Delta N_X = \frac{N_{X_{end}} - N_{X_{start}}}{Y_{end} - Y_{start}} \qquad \Delta N_Y = \frac{N_{Y_{end}} - N_{Y_{start}}}{Y_{end} - Y_{start}} \qquad \Delta N_Z = \frac{N_{Z_{end}} - N_{Z_{start}}}{Y_{end} - Y_{start}}$$

The full equation set for computing the surface normals along both the left and right edges are given below.

First the equations for the left edge:

The **Change in surface normals for the Left edge as we move from horizontal fill line to horizontal fill line** is given by:

$$\Delta N_{X\,Edge_{Left}} = \frac{NEnd_{X\,Edge_{Left}} - NStart_{X\,Edge_{Left}}}{Yend_{Edge_{Left}} - Ystart_{Edge_{Left}}}$$

$$\Delta N_{Y\,Edge_{Left}} = \frac{NEnd_{Y\,Edge_{Left}} - NStart_{Y\,Edge_{Left}}}{Yend_{Edge_{Left}} - Ystart_{Edge_{Left}}}$$

$$\Delta N_{Z\,Edge_{Left}} = \frac{NEnd_{Z\,Edge_{Left}} - NStart_{Z\,Edge_{Left}}}{Yend_{Edge_{Left}} - Ystart_{Edge_{Left}}}$$

The **initial surface normal for the Left edge on the first horizontal fill line** can be computed as:

$$N_{X\,Edge_{Left}\,FillLine_0} = NStart_{X\,Edge_{Left}}$$

$$N_{Y\,Edge_{Left}\,FillLine_0} = NStart_{Y\,Edge_{Left}}$$

$$N_{Z\,Edge_{Left}\,FillLine_0} = NStart_{Z\,Edge_{Left}}$$

The **surface normal for the Left edge on the subsequent horizontal fill lines** follows from:

$$N_{X\,Edge_{Left}\,FillLine_{i+1}} = N_{X\,Edge_{Left}\,FillLine_i} + \Delta N_{X\,Edge_{Left}}$$

$$N_{Y\,Edge_{Left}\,FillLine_{i+1}} = N_{Y\,Edge_{Left}\,FillLine_i} + \Delta N_{Y\,Edge_{Left}}$$

$$N_{Z\,Edge_{Left}\,FillLine_{i+1}} = N_{Z\,Edge_{Left}\,FillLine_i} + \Delta N_{Z\,Edge_{Left}}$$

Next, the equations for the right edge.

The **Change in surface normals for the Right edge as we move from horizontal fill line to horizontal fill line** is given by:

$$\Delta N_{X\,Edge_{Right}} = \frac{NEnd_{X\,Edge_{Right}} - NStart_{X\,Edge_{Right}}}{Yend_{Edge_{Right}} - Ystart_{Edge_{Right}}}$$

$$\Delta N_{Y\,Edge_{Right}} = \frac{NEnd_{Y\,Edge_{Right}} - NStart_{Y\,Edge_{Right}}}{Yend_{Edge_{Right}} - Ystart_{Edge_{Right}}}$$

$$\Delta N_{Z\,Edge_{Right}} = \frac{NEnd_{Z\,Edge_{Right}} - NStart_{Z\,Edge_{Right}}}{Yend_{Edge_{Right}} - Ystart_{Edge_{Right}}}$$

The **initial surface normal for the Right edge on the first horizontal fill line** can be computed by:

$$N_{X\,Edge_{Right}\,FillLine_0} = NStart_{X\,Edge_{Right}}$$

$$N_{Y\,Edge_{Right}\,FillLine_0} = NStart_{Y\,Edge_{Right}}$$

$$N_{Z\,Edge_{Right}\,FillLine_0} = NStart_{Z\,Edge_{Right}}$$

The **surface normal for the Right edge on the subsequent horizontal fill lines** follows from:

$$N_{X\,Edge_{Right}\,FillLine_{i+1}} = N_{X\,Edge_{Right}\,FillLine_i} + \Delta N_{X\,Edge_{Right}}$$

$$N_{Y\,Edge_{Right}\,FillLine_{i+1}} = N_{Y\,Edge_{Right}\,FillLine_i} + \Delta N_{Y\,Edge_{Right}}$$

$$N_{Z\,Edge_{Right}\,FillLine_{i+1}} = N_{Z\,Edge_{Right}\,FillLine_i} + \Delta N_{Z\,Edge_{Right}}$$

The above equations provide the surface normals for the starting pixel (left edge) and ending pixel (right edge) on the current fill line. As with Z-buffer and Gouraud shading computations, both the *NStart* ($NStart_X$, $NStart_Y$, $NStart_Z$) and *ΔN* (ΔN_X, ΔN_Y, ΔN_Z) for each edge can be pre-computed and stored in the Polygon Fill Table.

The next set of equations compute the surface normals across the current horizontal fill line.

The **change in surface normals as we move across the horizontal fill line from pixel to pixel** is given by:

$$\Delta N_{X\,FillLine_i} = \frac{N_{X\,Edge_{Right}FillLine_i} - N_{X\,Edge_{Left}FillLine_i}}{X_{Edge_{Right}FillLine_i} - X_{Edge_{Left}FillLine_i}}$$

$$\Delta N_{Y\,FillLine_i} = \frac{N_{Y\,Edge_{Right}FillLine_i} - N_{Y\,Edge_{Left}FillLine_i}}{X_{Edge_{Right}FillLine_i} - X_{Edge_{Left}FillLine_i}}$$

$$\Delta N_{Z\,FillLine_i} = \frac{N_{Z\,Edge_{Right}FillLine_i} - N_{Z\,Edge_{Left}FillLine_i}}{X_{Edge_{Right}FillLine_i} - X_{Edge_{Left}FillLine_i}}$$

The **surface normals for the first pixel on the current horizontal fill line** is given by:

$$N_{X\,FillLine_i\,Pixel_0} = N_{X\,Edge_{Left}FillLine_i}$$

$$N_{Y\,FillLine_i\,Pixel_0} = N_{Y\,Edge_{Left}FillLine_i}$$

$$N_{Z\,FillLine_i\,Pixel_0} = N_{Z\,Edge_{Left}FillLine_i}$$

The **subsequent surface normals for each pixel on the current horizontal fill line** are:

$$N_{X\,FillLine_i\,Pixel_{j+1}} = N_{X\,FillLine_i\,Pixel_j} + \Delta N_{X\,FillLine_i}$$

$$N_{Y\,FillLine_i\,Pixel_{j+1}} = N_{Y\,FillLine_i\,Pixel_j} + \Delta N_{Y\,FillLine_i}$$

$$N_{Z\,FillLine_i\,Pixel_{j+1}} = N_{Z\,FillLine_i\,Pixel_j} + \Delta N_{Z\,FillLine_i}$$

These equations provide us with the surface normals for every pixel on the polygon.

The code implementing these equations is generally integrated into the polygon fill and polygon fill table methods. Immediately before painting a pixel (with the TKinter fill_line method) the illumination intensity of the pixel is computed (using the Phong illumination model). The intensity value(s) returned by Phong illumination are then converted to a 24 bit hexadecimal RGB string by the *triColorHexCode* method provided in Section 16.5

The "Utah Teapot", a classic object, rendered using Phong Shading and Phong Illumination
[https://www.cs.ubc.ca/~tmm/courses/314-16/slides/lighting.pdf]

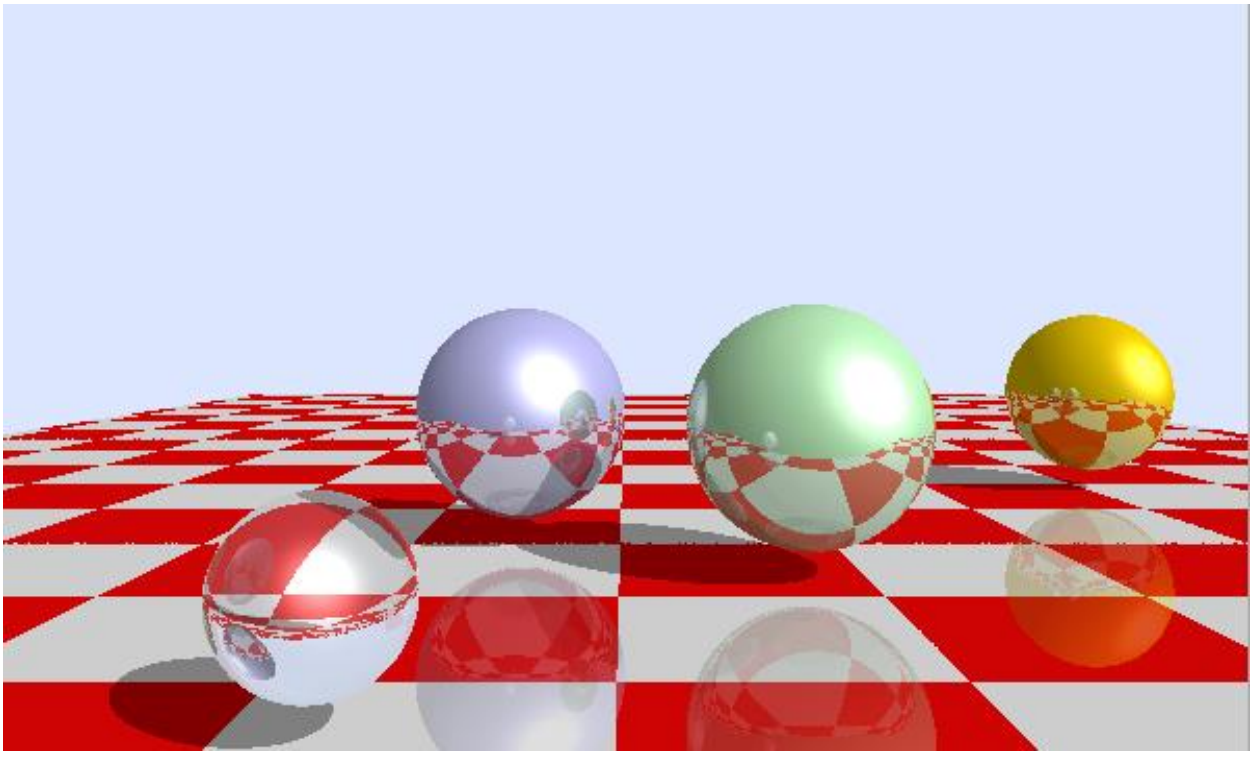
18. Ray Tracing

In this chapter we will look at a technique called “ray tracing” that is one of the most comprehensive simulations of illumination and reflection in computer graphics.

18.1 An Introduction to Ray Tracing

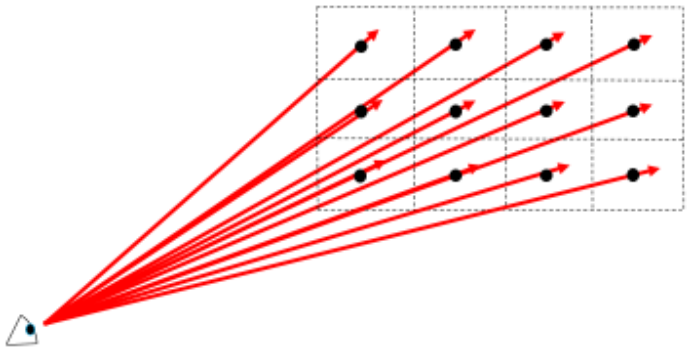
One can think of most other illumination-reflection models, such as the Phong Illumination-reflection model as a simplification of ray tracing. In ray tracing, light is simulated as straight lines traveling through a homogeneous media. The major difference between ray tracing and Phong illumination is the “depth” to which interactions between light rays and objects in the scene are examined. Ray tracing is capable of realistically rendering optical effects such as reflection, refraction, and shadows.

The image reproduced below, which was generated by a student taking this class several years ago, consists of four sphere’s resting on a checkerboard patterned “floor”. The three spheres on the right are highly reflective “metallic” spheres: one with a blue tint, one with a green tint, and one with a yellow tint. The fourth sphere, the one on the left, is a translucent “glass” sphere. The red and white “checkerboard” on which the spheres rest is itself somewhat shiny in appearance. Note that you can “see through” the glass sphere on the left, while the metallic spheres on the right, all sport reflections of the checkerboard and the other spheres. Also note that all the spheres cast realistic shadows.



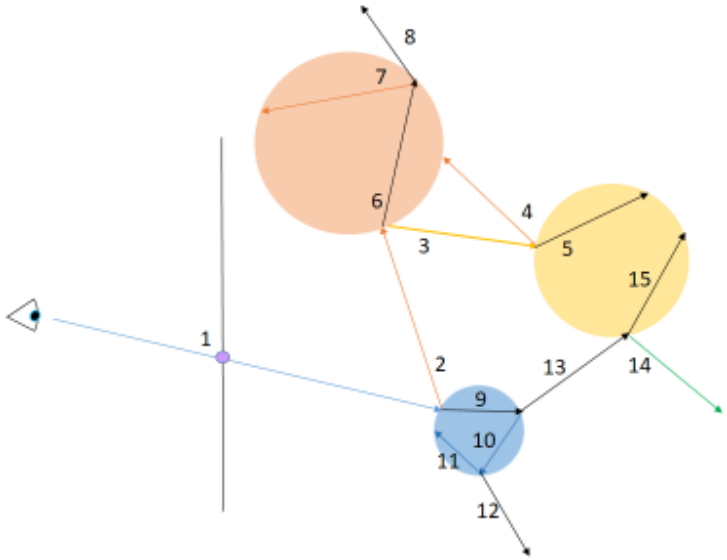
Assignment 5 (with enhancements) [Programmed by Hunter Anderson]

Ray tracing is a recursive technique that traces rays BACKWARDS from the viewpoint (where the viewer’s eye is assumed to be) through the screen pixels into the environment. The reason for tracing rays backwards is that the only light rays you can actually see are those that enter your eye, so these are the rays we focus on.



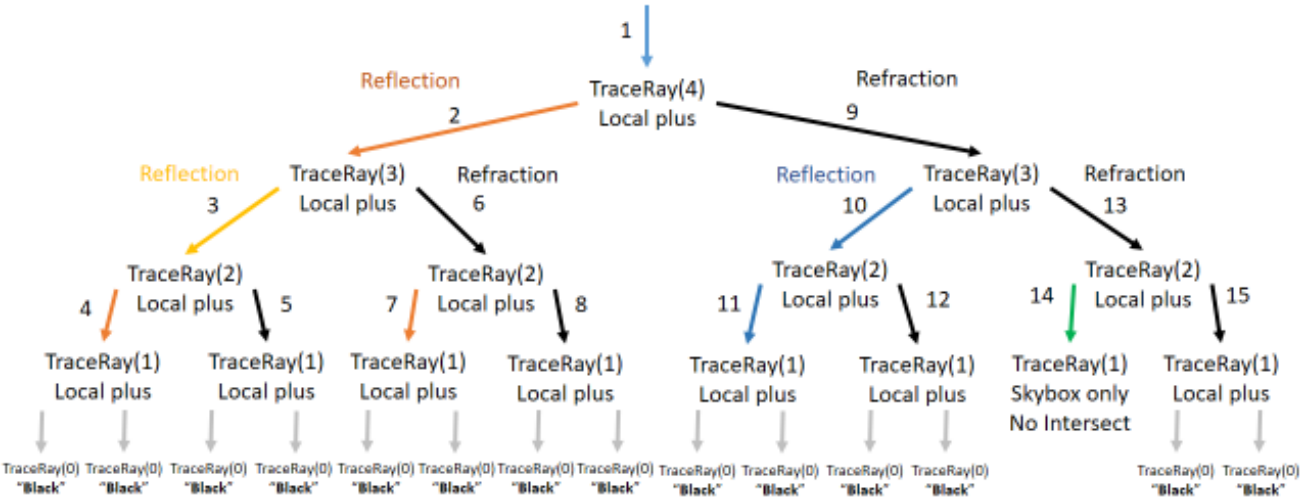
Tracing rays backwards from the viewpoint through the screen pixels into the environment

When tracing a ray, if the ray doesn't hit anything the color of the sky dome is returned. When a ray intersects one or more objects, the closest intersection, that is not behind the ray's starting point, is determined and the algorithm computes: (1) a local color for the intersected object (perhaps using a simple constant representing ambient lighting ($I_a \cdot K_d$), but more generally from a complete Phong illumination calculation); (2) the reflection vector and traces the reflected ray recursively, until some maximum depth is reached, returning a reflected color; (3) the refraction vector and traces the refracted ray recursively, until some maximum depth is reached, returning a refracted color. Finally, the algorithm combines the local, reflected, and refracted colors; and returns this combined color.



The recursive nature of ray tracing: tracing reflected and refracted rays 4 levels deep

The above figure illustrates the recursive nature of the ray tracing technique. Both reflected and refracted rays are shown. The reflected rays take on the color of the object they intersect. If a reflected ray doesn't intersect with any object it is shown in green. Refracted rays are drawn in black. Each ray is numbered to reflect the order in which that ray is processed by the recursion. In the above example, rays are traced four levels deep, resulting in 15 rays (including reflections and refractions) being traced in order to set the color of a single pixel. This process would be repeated for each pixel in the image.



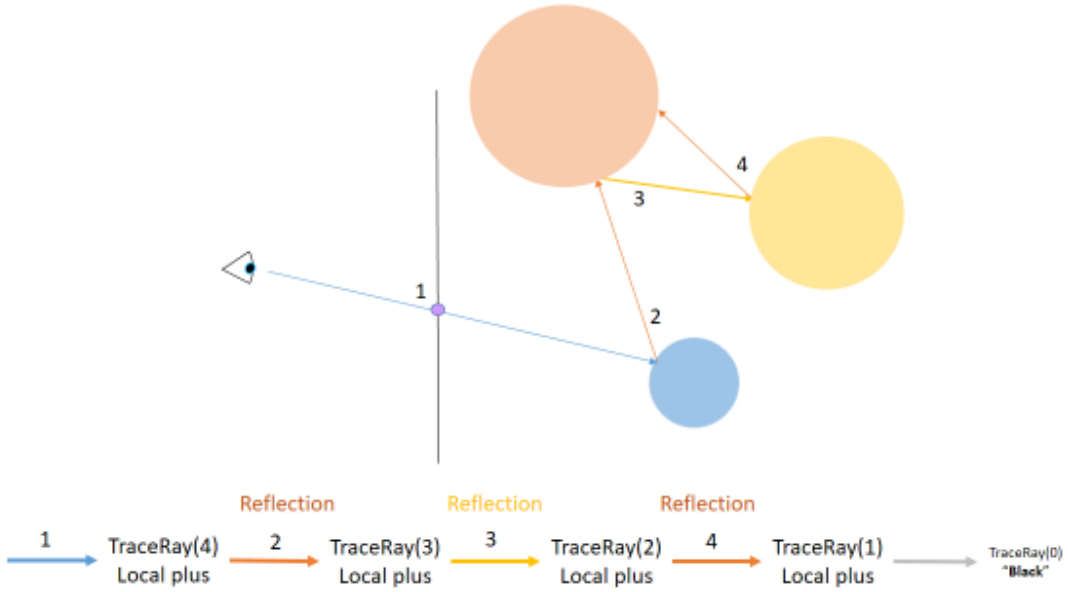
A pseudo-code representation of the *TraceRay()* method is given below. Above, the recursive calls to *TraceRay()* generated by the example on the previous page are illustrated as a binary tree.

```
# High-Level pseudo code for recursive ray trace method
# Inputs are startPoint and ray (which are both vectors) and
#            depth (an integer) that specifies the depth of recursion
# The method returns the color of the pixel

TraceRay(startPoint, ray, depth)
      # Return "black" when you reach the bottom of the recursive calls
      If depth == 0 Then Return color = "black"

      # Intersect ray with all objects and find intersection point
      # (if any) that is closest to startPoint of ray
      intersection = findClosestIntersection(startPoint, ray)

      # If the ray doesn't hit anything, let color equal "sky box" color
      If intersection == [] Then Return color = skyBoxColor

      # Compute local color (from Phong Illumination model)
      localColor = computeLocalColor(/*ARGS*/)

      # Calculate direction of reflected ray
      reflectedDirection = computeReflection(/*ARGS*/)

      # Compute reflected color
      reflectedColor = TraceRay(intersection, reflectedDirection, depth-1)

      # Calculate direction of the refracted ray
      refractedDirection = computeRefraction(/*ARGS*/)

      # Compute refracted color
      refractedColor = TraceRay(intersection, refractedDirection, depth-1)

      # Combine the local, reflected, and refracted colors
      color = combineColors(localColor, localWeight, reflectedColor,
                            reflectedWeight, refractedColor, refractedWeight)
      Return color
```

Pseudo-code for the core ray tracing method

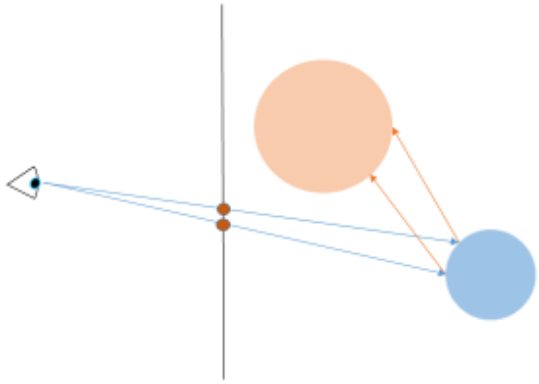
In an effort to limit the computational complexity of ray tracing, sometimes we eliminate refracted rays and trace only reflected rays. This works fine for scenes that may contain highly reflective objects but lack translucent / transparent objects (e.g., scenes without glass or water). By eliminating refraction, we can reduce the complexity of the ray tracing procedure from exponential in the trace depth to linear in the trace depth.

For example, as illustrated above, to trace both reflected and refracted rays four levels deep requires the computation of up to 15 intersections ($2^4 - 1$). Eliminating refracted rays from consideration reduces the complexity of tracing four levels deep to 4 intersections. This situation is illustrated below.



Tracing reflected rays 4 levels deep (refracted rays are not traced)

Before moving on to the computational details of intersections, reflections, and refractions; I wanted to bring to your attention something you may have overlooked. In ray traced images, the back faces of objects may become visible via their reflections in other objects. This means that the back face culling is not compatible with ray tracing. In the figure below, the back face of the pink sphere would be visible as a reflection in the blue sphere, assuming that sphere were highly reflective.



In ray tracing, a back face of one object may be visible as a reflection in another object

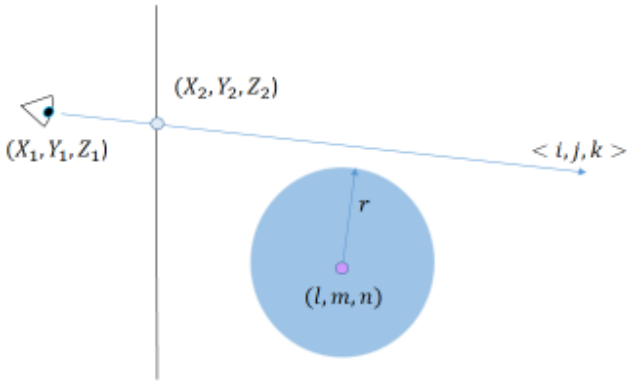
18.2 Computational Details of Intersections, Reflections, and Refractions

In order to implement ray tracing and render an image, we will need to: (1) compute the intersection points (if any) between each traced ray and all objects in our scene; (2) compute the direction vector of reflected light at the appropriate intersection points; (3) compute the direction vector of refracted light at the intersection points. Of these three types of computations, the only one we've covered so far is computation of a reflection vector. Computing the reflection vector was presented in Section 16.4.2 when discussing the mirror direction for light rays hitting a surface, which we needed in order to compute the specular component of the Phong illumination model.

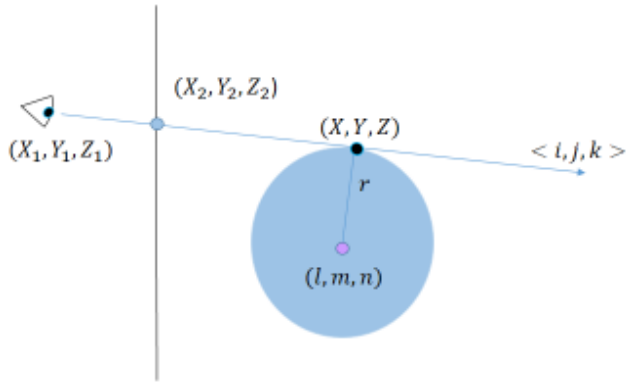
The scene we will render in Programming Assignment 5 consists of a number of colored spheres and a plane (overlaid with a checker board pattern). Thus, we will look at computing ray intersections with both spheres and planes. We also briefly review reflection vector computations, and present the equations for computing refraction vectors.

18.2.1 Computing the Intersection of a Traced Ray with a Sphere

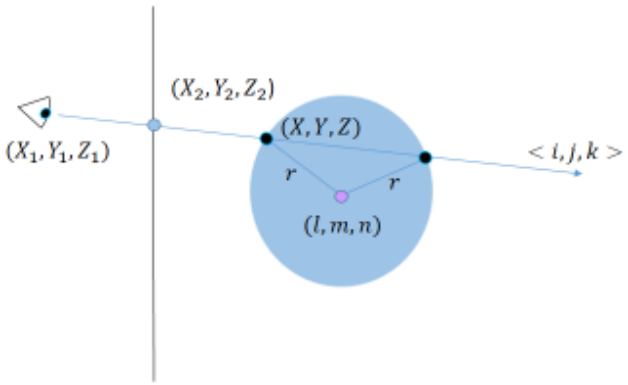
The questions we will answer in this section are: "Does a ray intersect a sphere?" and if so, "What is the nearest intersection point of the ray with the surface of the sphere?" There are three possible cases: (1) The traced ray may not intersect the sphere at all; (2) the traced ray may "graze" the sphere, intersecting it at one point; or (3) the traced ray may intersect the sphere at two points – in which case the nearer point would be the actual "real world" intersection point. Each of these three cases is illustrated below.



A traced ray may fail to intersect a sphere



A traced ray may intersect a sphere at one point



A traced ray may intersect a sphere at two points

The starting point of the traced ray (X_1, Y_1, Z_1) is illustrated as the center of projection. The second point on the traced ray (X_2, Y_2, Z_2) can be thought of as the screen pixel the ray passes through. Given these two points, a vector $< i, j, k >$ representing the traced ray can be computed by subtracting the first point from the second:

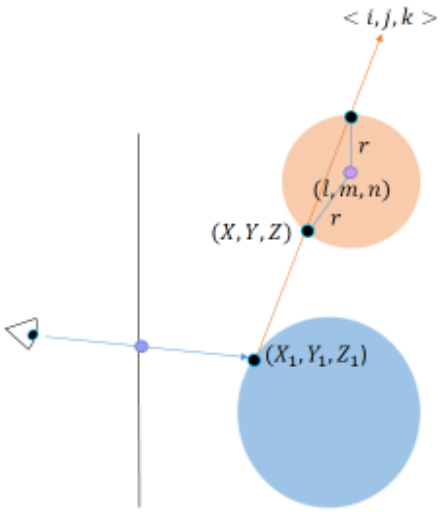
$$i = X_2 - X_1 \qquad j = Y_2 - Y_1 \qquad k = Z_2 - Z_1$$

Any point (X, Y, Z) along the traced ray anchored at the starting point (X_1, Y_1, Z_1) with direction vector $< i, j, k >$ can be computed as:

$$X = X_1 + i \cdot t \qquad Y = Y_1 + j \cdot t \qquad Z = Z_1 + k \cdot t$$

Where t represents the distance along the vector to travel in order to reach point (X, Y, Z). As long as t is positive, the point (X, Y, Z) will be in front of the starting point (X_1, Y_1, Z_1).

Note that for reflected and refracted rays, the starting point will be the previous ray's intersection point, rather than the center of projection. Additionally, the $< i, j, k >$ vector will be the reflected or refracted ray, so an (X_2, Y_2, Z_2) isn't required for the computations. The following figure illustrates the path of a reflected ray and its intersection with a second sphere.



A reflected ray intersects a second sphere

The center point of the sphere is denoted by (l, m, n) and the radius of the sphere by r. **Given any point (X, Y, Z) on the surface of the sphere, we have:**

$$r = \sqrt{(X-l)^2 + (Y-m)^2 + (Z-n)^2}$$

Squaring both sides gives:

$$r^2 = (X-l)^2 + (Y-m)^2 + (Z-n)^2$$

Or

$$(X-l)^2 + (Y-m)^2 + (Z-n)^2 - r^2 = 0$$

Next, substitute X, Y, Z from the traced ray equation for X, Y, Z in the sphere equation, and solve for t.

$$(X_1 + i \cdot t - l)^2 + (Y_1 + j \cdot t - m)^2 + (Z_1 + k \cdot t - n)^2 - r^2 = 0$$

If the above equation has any real roots, those roots will represent the intersection point(s) between the traced ray and the sphere.

In order to simplify this equation we need to begin by driving the squares inwards:

$$(X_1 + i \cdot t - l) \cdot (X_1 + i \cdot t - l) + (Y_1 + j \cdot t - m) \cdot (Y_1 + j \cdot t - m) + (Z_1 + k \cdot t - n) \cdot (Z_1 + k \cdot t - n) - r^2 = 0$$

$$\left({X_1}^2 + X_1 \cdot i \cdot t - X_1 \cdot l + X_1 \cdot i \cdot t + i^2 \cdot t^2 - i \cdot t \cdot l - l \cdot X_1 - l \cdot i \cdot t + l^2\right) +$$

$$\left({Y_1}^2 + Y_1 \cdot j \cdot t - Y_1 \cdot m + Y_1 \cdot j \cdot t + j^2 \cdot t^2 - j \cdot t \cdot m - m \cdot Y_1 - m \cdot j \cdot t + m^2\right) +$$

$$\left({Z_1}^2 + Z_1 \cdot k \cdot t - Z_1 \cdot n + Z_1 \cdot k \cdot t + k^2 \cdot t^2 - k \cdot t \cdot n - n \cdot Z_1 - n \cdot k \cdot t + n^2\right) - r^2 = 0$$

The ray-sphere intersection equation reduces to the following quadratic:

$$\boldsymbol{a \cdot t^2 + b \cdot t + c = 0}$$

Where:

$$\boldsymbol{a = i^2 + j^2 + k^2}$$

$$\boldsymbol{b = 2 \cdot i \cdot (X_1 - l) + 2 \cdot j \cdot (Y_1 - m) + 2 \cdot k \cdot (Z_1 - n)}$$

$$\boldsymbol{c = l^2 + m^2 + n^2 + {X_1}^2 + {Y_1}^2 + {Z_1}^2 + 2 \cdot (-l \cdot X_1 - m \cdot Y_1 - n \cdot Z_1) - r^2}$$

The roots of this equation gives the intersection points between the ray and the sphere. No real roots indicates that the ray doesn't intersect the sphere. Two real roots indicate that the ray intersects the surface of the sphere at two points. In this case the smaller value for t gives the real world (nearer) intersection point of the ray and sphere.

$$\boldsymbol{t = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}}$$

If the discriminant $(b^2 - 4 \cdot a \cdot c) < 0$ then there are no real roots (no intersection).
If the discriminant $(b^2 - 4 \cdot a \cdot c) = 0$ then there is one real root (one intersection).
If the discriminant $(b^2 - 4 \cdot a \cdot c) > 0$ then there are two real roots (two intersections, select nearest).

Here is a complete example illustrating the computation of a ray-sphere intersection point.

Given (1) a sphere with radius $r = 2$ and a center point $(l, m, n) = (5, 4, 10)$; and (2) a traced ray beginning at start point $(0,0,0)$ with a second point at $(5,4,12)$; determine if the ray intersects the sphere, and, if so, return the nearest intersection point.

Since we were given two point along the ray, $(0,0,0)$ and $(5,4,12)$ compute the ray's direction vector:

$$i = X_2 - X_1 \qquad j = Y_2 - Y_1 \qquad k = Z_2 - Z_1$$
$$i = 5 - 0 = 5 \qquad j = 4 - 0 = 4 \qquad k = 12 - 0 = 12$$

Plug the ray's starting point (0,0,0) and direction vector $(5,4,12)$ into the equation for computing all points (X, Y, Z) along the ray.

$$X = X_1 + i \cdot t \qquad Y = Y_1 + j \cdot t \qquad Z = Z_1 + k \cdot t$$
$$X = 0 + 5 \cdot t = 5 \cdot t \qquad Y = 0 + 4 \cdot t = 4 \cdot t \qquad Z = 0 + 12 \cdot t = 12 \cdot t$$

Now, compute the ray-sphere intersection equation.

$$a \cdot t^2 + b \cdot t + c = 0$$

$$a = i^2 + j^2 + k^2$$
$$a = 5^2 + 4^2 + 12^2 = 25 + 16 + 144 = 185$$

$$b = 2 \cdot i \cdot (X_1 - l) + 2 \cdot j \cdot (Y_1 - m) + 2 \cdot k \cdot (Z_1 - n)$$
$$b = 2 \cdot 5 \cdot (0 - 5) + 2 \cdot 4 \cdot (0 - 4) + 2 \cdot 12 \cdot (0 - 10) = -50 - 32 - 240 = -322$$

$$c = l^2 + m^2 + n^2 + {X_1}^2 + {Y_1}^2 + {Z_1}^2 + 2 \cdot (-l \cdot X_1 - m \cdot Y_1 - n \cdot Z_1) - r^2$$
$$c = 5^2 + 4^2 + 10^2 + 0^2 + 0^2 + 0^2 + 2 \cdot (-5 \cdot 0 - 4 \cdot 0 - 10 \cdot 0) - 2^2 = 25 + 16 + 100 - 4 = 137$$

Thus, for this particular sphere and traced ray we have:

$$185 \cdot t^2 - 322 \cdot t + 137 = 0$$

$$t = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$$

$$t = \frac{322 \pm \sqrt{(-322)^2 - 4 \cdot 185 \cdot 137}}{2 \cdot 185} = \frac{322 \pm \sqrt{2304}}{370} = \frac{322 \pm 48}{370}$$

$$t_1 = \frac{322 + 48}{370} = \frac{370}{370} = 1 \qquad t_2 = \frac{322 - 48}{370} = \frac{274}{370} \approx 0.74$$

Choosing the smaller t value and inserting back into the original equations gives the intersection point:

$$X = 0 + 5 \cdot t = 5 \cdot t \qquad Y = 0 + 4 \cdot t = 4 \cdot t \qquad Z = 0 + 12 \cdot t = 12 \cdot t$$

$$X = 0 + 5 \cdot 0.74 = 3.7 \qquad Y = 0 + 4 \cdot 0.74 = 2.96 \qquad Z = 0 + 12 \cdot 0.74 = 8.88$$

Thus, the nearest intersection point between this ray and sphere is $(X, Y, Z) = (3.7, 2.96, 8.88)$.

18.2.2 Computing the Intersection of a Traced Ray with a Plane

In Chapter 12.2 we noted that **all points (x, y, z) where:**

$$Ax + By + Cz - D = 0 \quad \text{and} \quad D = Aa + Bb + Cc$$

reside in a plane. (A, B, C) is the plane's surface normal, often written (N_X, N_Y, N_z), and (a, b, c) is a point on the plane.

Also, as discussed in the previous section, Section 18.2.1, the direction vector $< i, j, k >$ for our traced ray can be computed from two points along that ray as:

$$i = X_2 - X_1 \qquad j = Y_2 - Y_1 \qquad k = Z_2 - Z_1$$

And any point (X, Y, Z) along the traced ray anchored at the starting point (X_1, Y_1, Z_1) with direction vector $< i, j, k >$ can be computed as:

$$X = X_1 + i \cdot t \qquad Y = Y_1 + j \cdot t \qquad Z = Z_1 + k \cdot t$$

The intersection point between the traced ray and a plane (if such an intersection point exists) can be computed by substituting the (X, Y, Z) from the traced ray equation for (x, y, z) in the plane equation and then solving for t.

$$Ax + By + Cz - D = 0 \quad \text{and} \quad D = Aa + Bb + Cc$$

$$A \cdot (X_1 + i \cdot t) + B \cdot (Y_1 + j \cdot t) + C \cdot (Z_1 + k \cdot t) - D = 0$$

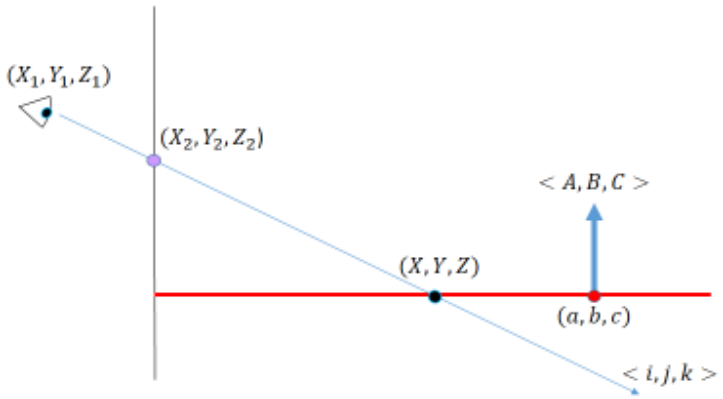
$$A \cdot X_1 + A \cdot i \cdot t + B \cdot Y_1 + B \cdot j \cdot t + C \cdot Z_1 + C \cdot k \cdot t - D = 0$$

$$A \cdot i \cdot t + B \cdot j \cdot t + C \cdot k \cdot t = -(A \cdot X_1 + B \cdot Y_1 + C \cdot Z_1 - D)$$

$$(A \cdot i + B \cdot j + C \cdot k) \cdot t = -(A \cdot X_1 + B \cdot Y_1 + C \cdot Z_1 - D)$$

$$t = \frac{-(A \cdot X_1 + B \cdot Y_1 + C \cdot Z_1 - D)}{A \cdot i + B \cdot j + C \cdot k}$$

If the denominator $(A \cdot i + B \cdot j + C \cdot k) = 0$ the traced ray is parallel to the plane (no intersection).



A traced ray may intersect a plane at one point

Here is a complete example illustrating the computation of a ray-plane intersection point.

Given (1) a plane with surface normal $< A, B, C > = < 0,0,1 >$ and anchor point $(a, b, c) = (0,0,10)$; and (2) a traced ray beginning at start point $(0,0,-10)$ with a second point at $(10,5,12)$; determine if the ray intersects the plane, and, if so, return the intersection point.

Given two points along the traced ray, $(0,0,-10)$ and $(10,5,12)$ compute the ray's direction vector:

$$i = X_2 - X_1 \qquad j = Y_2 - Y_1 \qquad k = Z_2 - Z_1$$
$$i = 10 - 0 = 10 \qquad j = 5 - 0 = 5 \qquad k = 12 - (-10) = 22$$

Plug the ray's starting point (0,0,-10) and direction vector $(10,5,22)$ into the equation for computing all points (X, Y, Z) along the ray.

$$X = X_1 + i \cdot t \qquad Y = Y_1 + j \cdot t \qquad Z = Z_1 + k \cdot t$$
$$X = 0 + 10 \cdot t = 10 \cdot t \qquad Y = 0 + 5 \cdot t = 5 \cdot t \qquad Z = -10 + 22 \cdot t$$

Now, compute the ray-plane intersection equation, and solve for t.

$$t = \frac{-(A \cdot X_1 + B \cdot Y_1 + C \cdot Z_1 - D)}{A \cdot i + B \cdot j + C \cdot k}$$

$$D = Aa + Bb + Cc \qquad \text{so} \qquad D = 0 \cdot 0 + 0 \cdot 0 + 1 \cdot 10 = 10$$

$$t = \frac{-(0 \cdot 0 + 0 \cdot 0 + 1 \cdot (-10) - 10)}{0 \cdot 10 + 0 \cdot 5 + 1 \cdot 22} = \frac{-(-20)}{22} = \frac{20}{22} \approx 0.91$$

The traced ray intersects the plane, since the denominator is not zero, and the intersection point is:

$$X = 0 + 10 \cdot t = 10 \cdot t \qquad Y = 0 + 5 \cdot t = 5 \cdot t \qquad Z = -10 + 22 \cdot t$$

$$X = 0 + 10 \cdot 0.91 = 9.1 \qquad Y = 0 + 5 \cdot 0.91 = 4.55 \qquad Z = -10 + 22 \cdot 0.91 = -10 + 20.02 = 10.02$$

$$(X, Y, Z) = (9.1, 4.55, 10.02)$$

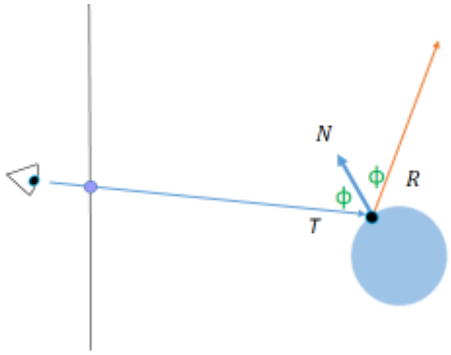
18.2.3 Computing the Reflection Vector

In Section 16.4.2 we discussed how to compute a reflection vector given a surface normal, N, and a lighting vector, L. We can use those equations again to compute the reflection vector of a traced ray with one tiny tweak. When a traced ray, T, intersects the surface of an object, T points towards the object. However, our original reflection equations assume the lighting vector, L, points away from the object towards the light source. Thus, we will need to update the equations to replace L with $-T$.

The **original equation for reflecting a light ray, $\boldsymbol{L}$**, for incident angles $\phi < 90^{\circ}$ is:

$$R = N + (-L) \cdot \frac{1}{2 \cdot \cos(\phi)}$$

$$R = N - \frac{L}{2 \cdot (N_X \cdot L_X + N_Y \cdot L_Y + N_Z \cdot L_Z)}$$



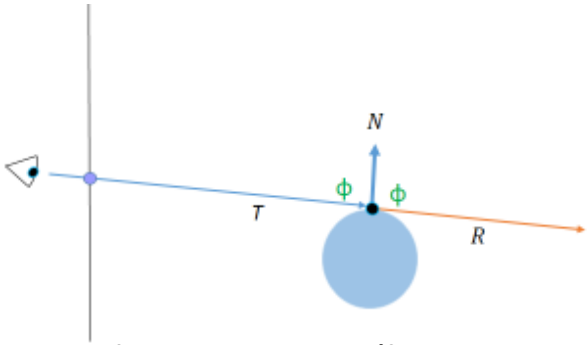
A traced ray, T, is reflected in direction R (for incident angles $\phi < 90^{\circ}$)

The modified equation for reflecting a traced ray, T, for incident angles $\boldsymbol{\phi < 90^{\circ}}$ is:

$$R = N + T \cdot \frac{1}{2 \cdot \cos(\phi)}$$

Or, when N and T are unit vectors:

$$\boldsymbol{R = N + \frac{T}{2 \cdot (N_X \cdot -T_X + N_Y \cdot -T_Y + N_Z \cdot -T_Z)}}$$

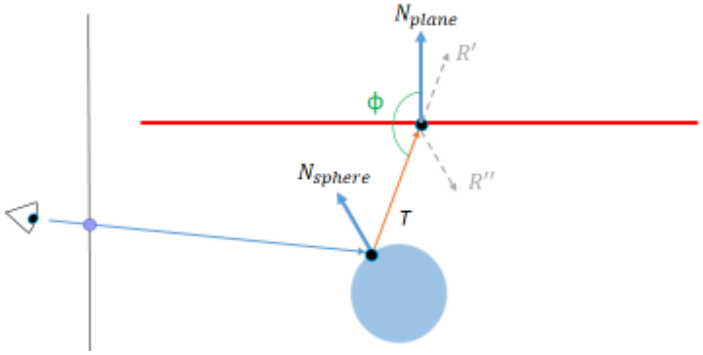


A traced ray, T, is reflected in direction R (for incident angles $\phi = 90^{\circ}$)

When $\boldsymbol{\phi = 90^{\circ}}$ the traced ray grazes the surface and thus,

$$\boldsymbol{R = T}$$

The original reflection equations include the case where the incident angle $\phi > 90^{\circ}$. Can such a case occur when computing the reflection of a traced ray?



A traced ray, T, intersects the back side of a plane (for incident angles $\phi > 90^{\circ}$)

When tracing rays, intersections with spheres will always be at incident angles $\phi \leq 90^{\circ}$. However, when considering planes, it is possible for a traced ray to intersect the back side of a plane.

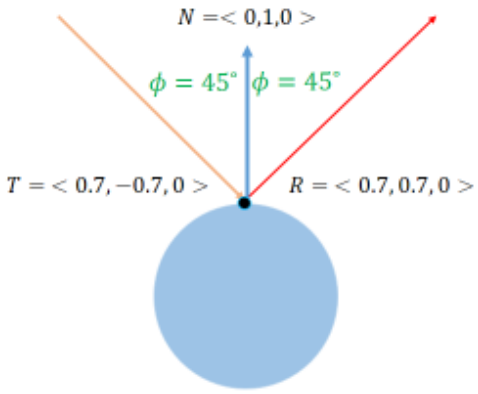
We can either ignore such intersections, allowing the ray to pass through the plane, if we decide that only the front sides of a plane should be visible, giving $\boldsymbol{R'} = \boldsymbol{T}$ as shown in the above illustration.

On the other hand, we can decide that both sides of the plane should be visible, which would generate **reflection vector $\boldsymbol{R''}$ for incident angles $\boldsymbol{\phi > 90^{\circ}}$** as shown in the above illustration.

$$\boldsymbol{R''} = -\boldsymbol{N} - \frac{\boldsymbol{T}}{\boldsymbol{2 \cdot (N_X \cdot -T_X + N_Y \cdot -T_Y + N_Z \cdot -T_Z)}}$$

The above equation requires that N and T be unit vectors.

Here is an example illustrating the computation of the reflection vector of a traced ray (angles $\boldsymbol{\phi < 90^{\circ}}$)



Computing the reflection vector of a traced ray

Object surface normal at intersection point: $N = < 0,1,0 >$

Traced ray (points towards surface of object): $T = < 0.7, -0.7, \ 0 >$

The reflection equation for traced rays with incident angles $\phi < 90^{\circ}$:

$$\boldsymbol{R} = \boldsymbol{N} + \frac{\boldsymbol{T}}{\boldsymbol{2 \cdot (N_X \cdot -T_X + N_Y \cdot -T_Y + N_Z \cdot -T_Z)}}$$

Remember than N and T are vectors with X, Y, and Z components, so this equation becomes:

$$\boldsymbol{R_X} = \boldsymbol{N_X} + \frac{\boldsymbol{T_X}}{\boldsymbol{2 \cdot (N_X \cdot -T_X + N_Y \cdot -T_Y + N_Z \cdot -T_Z)}}$$

$$\boldsymbol{R_Y} = \boldsymbol{N_Y} + \frac{\boldsymbol{T_Y}}{\boldsymbol{2 \cdot (N_X \cdot -T_X + N_Y \cdot -T_Y + N_Z \cdot -T_Z)}}$$

$$\boldsymbol{R_Z} = \boldsymbol{N_Z} + \frac{\boldsymbol{T_Z}}{\boldsymbol{2 \cdot (N_X \cdot -T_X + N_Y \cdot -T_Y + N_Z \cdot -T_Z)}}$$

$$R_X = 0 + \frac{0.7}{2 \cdot (0 \cdot -0.7 + 1 \cdot -(-0.7) + 0 \cdot -0)} = 0 + \frac{0.7}{2 \cdot (0.7)} = \frac{0.7}{1.4} = 0.5$$

$$R_Y = 1 + \frac{-0.7}{2 \cdot (0 \cdot -0.7 + 1 \cdot -(-0.7) + 0 \cdot -0)} = 1 + \frac{-0.7}{2 \cdot (0.7)} = 1 + \frac{-0.7}{1.4} = 1 - 0.5 = 0.5$$

$$R_Z = 0 + \frac{0}{2 \cdot (0 \cdot -0.7 + 1 \cdot -(-0.7) + 0 \cdot -0)} = 0 + \frac{0}{2 \cdot (0.7)} = \frac{0}{1.4} = 0$$

So, $R = < 0.5, 0.5, 0 >$. Once we normalize the refection vector we get $R = < 0.7, 0.7, 0 >$

$$Magnitude = \sqrt{0.5^2 + 0.5^2 + 0^2} = \sqrt{0.25 + 0.25 + 0} = \sqrt{0.5} = 0.707$$

$$R_{norm} = \langle \frac{0.5}{0.707}, \frac{0.5}{0.707} . \frac{0}{0.707} \rangle \approx \langle 0.7, 0.7, 0 \rangle$$

18.2.4 Computing the Refraction Vector

When a traced light ray, T, travelling through a medium with density, d_1, encounters a partially transparent (translucent) object, with density d_2, that light ray will be refracted at an angle, ϕ_2, related to (1) the angle of incidence, ϕ_1, between the traced ray and the object's surface normal N at the intersection point and (2) the change in density, d, between the two mediums. We will refer to the vector for this refracted ray as $Trans$, which stands for "transmitted". (We are already using T for the traced ray and R for the reflection vector, so $Trans$ is about the best I can do.)

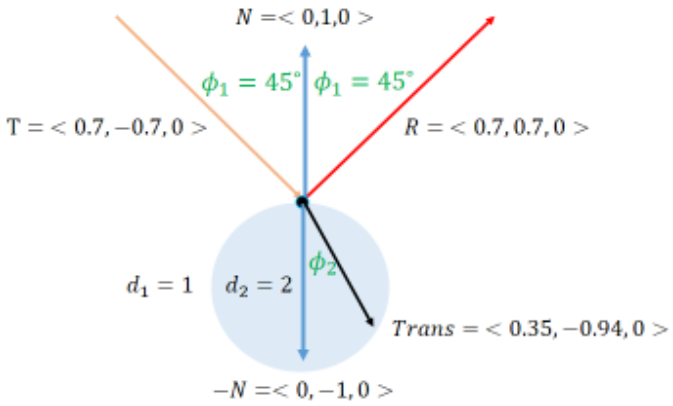


Illustration of a refracted, or transmitted, ray: $Trans$

The equations for computing the refraction vector are:

$$Trans = \frac{1}{d} \cdot T - \left(\cos \phi_2 - \frac{1}{d} \cos \phi_1 \right) \cdot N$$

$$d = \frac{d_2}{d_1}$$

$$\cos \phi_2 = \sqrt{1 - \frac{1}{d^2} \cdot (1 - (\cos \phi_1)^2)}$$

We will not attempt to derive these equations (as we have done with most other equations in the course) but simply accept them as given. A separate equation for $\cos\phi_1$ was not included above as we learned previously that $\cos\phi_1$ can be computed as: $\cos\phi_1 = -T \cdot N$

Assuming vectors T and N are unit vectors, this collection of equations for computing the transmitted refraction vector, $Trans$, can be rewritten as a single equation using dot products as:

$$Trans = \frac{1}{d} \cdot T - \left(\sqrt{1 - \frac{1}{d^2} \cdot (1 - (-T \cdot N)^2)} - \frac{1}{d}(-T \cdot N) \right) \cdot N$$

Of course, since $Trans$ is a vector, it has X, Y, and Z components. Thus, our final **equations for computing the transmitted refraction vector** are:

$$\boldsymbol{Trans_X = \frac{1}{d} \cdot T_X - \left(\sqrt{1 - \frac{1}{d^2} \cdot (1 - (-T_X \cdot N_X - T_Y \cdot N_Y - T_Z \cdot N_Z)^2)} - \frac{1}{d}(-T_X \cdot N_X - T_Y \cdot N_Y - T_Z \cdot N_Z) \right) \cdot N_X}$$

$$\boldsymbol{Trans_Y = \frac{1}{d} \cdot T_Y - \left(\sqrt{1 - \frac{1}{d^2} \cdot (1 - (-T_X \cdot N_X - T_Y \cdot N_Y - T_Z \cdot N_Z)^2)} - \frac{1}{d}(-T_X \cdot N_X - T_Y \cdot N_Y - T_Z \cdot N_Z) \right) \cdot N_Y}$$

$$\boldsymbol{Trans_Z = \frac{1}{d} \cdot T_Z - \left(\sqrt{1 - \frac{1}{d^2} \cdot (1 - (-T_X \cdot N_X - T_Y \cdot N_Y - T_Z \cdot N_Z)^2)} - \frac{1}{d}(-T_X \cdot N_X - T_Y \cdot N_Y - T_Z \cdot N_Z) \right) \cdot N_Z}$$

Here is an example illustrating the computation of the refraction vector of a traced ray.

Given a traced ray $T = < 0.7, -0.7, \ 0 >$, which points towards the surface of an object; the surface normal of the object at the intersection point $N = < 0,1,0 >$; and the densities $d_1 = 1$ and $d_2 = 2$, giving $d = \frac{d_2}{d_1} = \frac{2}{1} = 2$

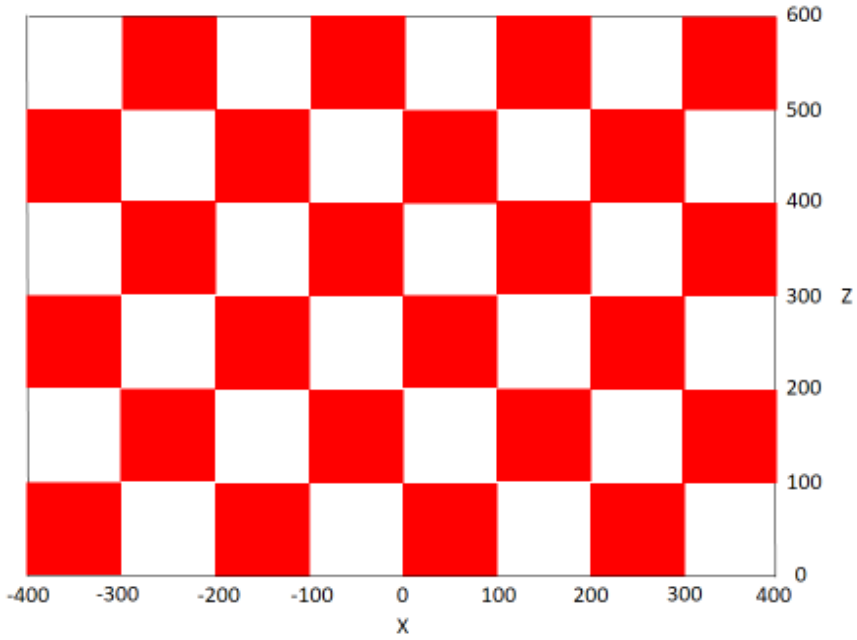
$$Trans_X = \frac{1}{2} \cdot 0.7 - \left(\sqrt{1 - \frac{1}{2^2} \cdot (1 - (0.7)^2)} - \frac{1}{2}(0.7) \right) \cdot 0 = 0.35$$

$$\begin{aligned} Trans_Y &= \frac{1}{2} \cdot (-0.7) - \left(\sqrt{1 - \frac{1}{2^2} \cdot (1 - (0.7)^2)} - \frac{1}{2}(0.7) \right) \cdot 1 \\ &= (-0.35) - \left(\sqrt{1 - 0.25 \cdot (1 - 0.49)} - 0.35 \right) = (-0.35) - \left(\sqrt{1 - 0.25 \cdot 0.51} - 0.35 \right) \\ &= (-0.35) - \left(\sqrt{1 - 0.1275} - 0.35 \right) = (-0.35) - \left(\sqrt{0.875} - 0.35 \right) \\ &= (-0.35) - (0.94 - 0.35) = -0.94 \end{aligned}$$

$$Trans_Z = \frac{1}{2} \cdot 0 - \left(\sqrt{1 - \frac{1}{2^2} \cdot (1 - (0.7)^2)} - \frac{1}{2}(0.7) \right) \cdot 0 = 0$$

So, $Trans = < 0.35, -0.94, 0 >$ with $magnitude = \sqrt{0.35^2 + (-0.94)^2 + 0^2} \approx 1$

18.3 Overlaying a Checker Board Pattern on a Plane



A checker board pattern overlaid on plane parallel to X-Z

Assignment 5 asks you to implement a ray traced scene consisting of several highly reflective spheres floating over a plane painted in a checker board pattern. In order to implement this assignment you will need to generate a checker board pattern on an X-Z plane ($N = < 0,1,0 >$) shifted downward. In other words, the checker board plane will be located at $Y = -PlaneConstant$ where $PlaneConstant$ is the number of units the plane has been shifted downwards (e.g., $Y = -200$).

We know from the equations presented in Section 18.2.2 how to calculate the intersection point between a traced ray and a plane, but how will be know the color of the square at that intersection point?

In the above illustration our checker board contains red and white squares that are each 100 by 100 units. Given any point on the plane (X, Y, Z) where $Z = -PlaneConstant$, the color of that point on the checker board can be computed by the following pseudo code:

```
If X ≥ 0 Then ColorFlag = 1; Else ColorFlag = 0;
If ||X|| mod 200 > 100 Then ColorFlag = !ColorFlag;
If ||Z|| mod 200 > 100 Then ColorFlag = !ColorFlag;
If ColorFlag Then Color = "red"; Else Color = "white";
```

In the code, ColorFlag is a bit used to indicate "red" (1) or "white" (0). This code works by first considering X and then Z. On the first row of squares, the square immediately to the right of 0 is red, while the square immediately to the left of 0 is white. Thus, if X is positive, the first square is red (so ColorFlag is set to 1); otherwise X is negative and the first square is white (so ColorFlag is set to 0).

Next, we note that the red-white pattern for positive X values (or white-red pattern for negative X values) repeats every 200 units. The mod operator enables us to skip over some number of pattern repetitions (possibly zero such repetitions) moving rightward for positive X and leftward for negative X.

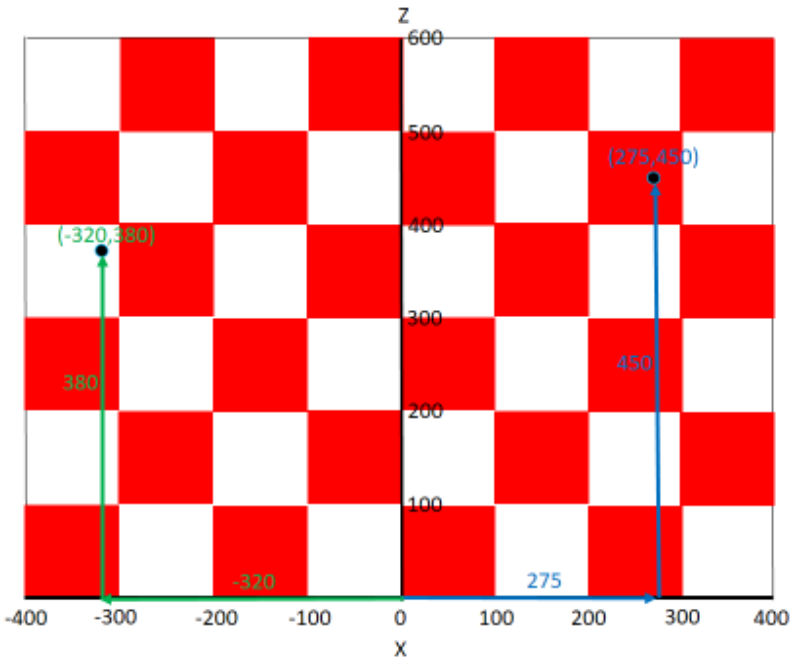
The remainder returned by $||X \ mod \ 200||$ indicates how far we are into the current red-white (or white-red) pattern. If this remainder is greater than 100 the ColorFlag must be flipped.

For example, say we have an X value of 275, X is positive so ColorFlag starts at 1 ("red"). 275 mod 200 = 75. We moved rightward, skipping over one repetition of the pattern and end up 75 units into the "current" pattern. Since 75 is not greater than 100, ColorFlag is not flipped and continues to be 1, indicating a red square.

In addition to the ColorFlag flipping along X, it can also flip along Z. You can think of the first two lines of pseudo code as moving rightward or leftward to the appropriate color square at the beginning of a column. The third line of pseudo code get us to the appropriate square along that column. As before the mod operator allow us to skip over as many repetitions of the pattern as necessary, and the remainder tells us whether to flip the ColorFlag or not (if > 100, flip).

Continuing our example of X = 275, let's say Z=450. 450 mod 200 = 50. We forward in Z over two red-white patterns and ended up 50 units into the third occurrence of the pattern. Since 50 is not greater than 100, ColorFlag is not flipped. It continues to be 1 ("red"). Thus the color of the pixel at $(275, 450)$ is red.

Finally, let's look at one more example. Say we are given an X value of -320 and a Z value of 380. As X is negative we begin with ColorFlag = 0 ("white"). This time we will move leftward, skipping over one repetition of the white-red pattern. As the absolute value of $-320 \ mod \ 200$ is 120 we end up 120 units into the "current" pattern. Since $120 \ > \ 100$ the ColorFlag flips from 0 to 1, indicating a red square. The Z is 380 and the absolute value of $380 \ mod \ 200$ is 180. Since 180 is greater than 100, the ColorFlag flips again, this time from 1 to 0, indicating we landed on a white square. Thus, the color of the pixel at $(-320, 380)$ is white.



Determining whether a pixel should be red or white

18.4 Improving the Appearance of Ray Traced Images

Ray tracing can be understood as a hybrid approach, in the sense that some rays are traced and others are not.

Untraced rays are spread empirically. These untraced rays come from the local illumination components that are generally computed using an illumination model, such as Phong illumination. As you will recall, the Phong illumination model incorporates three types of illumination: (1) ambient (which captures general background lighting (Ia) and its interaction with an object (Kd)); (2) diffuse reflection of point light (which uses Lambert's Cosine Law to determine the amount of point light that reaches and is reflected from an object); and (3) specular reflection of point light (which computes the spread of a specular highlight around a reflection vector, R). The paths of individual light rays in the ambient, diffuse, and specular components of Phong illumination are not traced, but are instead treated as percentages of overall illumination – they are spread empirically in a manner to achieve as realistic an outcome as possible within an acceptable computational budget.

Traced rays are not spread. Individual rays are reflected and refracted. Reflected rays bouncing off a surface are computed as if the surface were a perfect mirror. Refracted rays act as if the refracting objects were perfect transmitters of light. In the real world, there is no such things as a "perfect" mirror or a "perfect" refractor. All objects contain minor imperfections – think of microscopic scratches in a mirror or tiny imperfections in a crystal ball. Because these imperfections are missing from our reflection and refraction equations, scenes generated via ray tracing can take on a "hyper-perfect" look, often referred to as a "ray traced signature", consisting of sharp reflections, refractions, and shadows. Some of these visual effects are lessened to a degree because reflected objects are often smaller in size and rendered with lower illumination intensities. Nevertheless, these effects can be annoying and lead to "aliasing artifacts".

To understand why these effects happen, we can visualize ray tracing as a "sampling" technique where a single ray is transmitted through each pixel on the screen and the sum of the interactions of that ray with the objects in the scene, the ray's reflections and refractions, are summarized in the color of the pixel. Though pixels are small, they are not infinitesimal points, so the path of a single ray passing through a pixel, especially a ray that perfectly reflects and perfectly refracts, does a poor job capturing the complexities of the (essentially) infinite number of light rays bouncing around our real world, reflecting off, and refracting through, the highly imperfect objects with which that world is populated.

We next look at several different techniques that can be used to improve the look of ray traced images.

18.4.1 Tracing Multiple Rays per Pixel

One obvious method of improving the look of ray traced images is to generate multiple rays per pixel and average the results. In 4X oversampling, four rays are transmitted through a pixel and the results are averaged. Of course, if we send four rays through the center of a pixel and make no other changes, all four rays will produce exactly the same result.

There are two ways around this problem. The first approach is to send each of the four rays through different portions of the pixel, say the corners of the pixel, rather than all rays passing through the exact center of the pixel. Since each of the four rays traveling through the pixel will be at slightly different

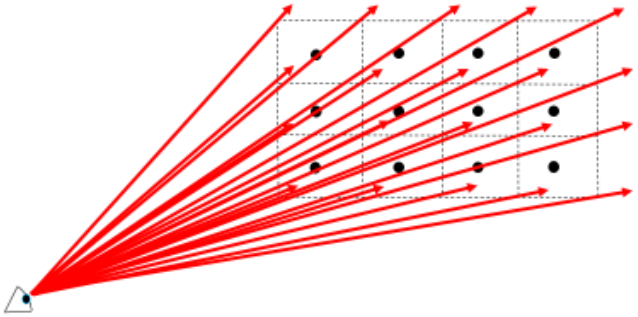
angles, they will trace out somewhat different paths through the environment. The returned color information from each ray will be averaged to give the pixel's final color.

The second approach is to send all four rays through the pixel center, but at every intersection point "jiggle" the reflection and refraction vectors with a small amount of random noise. This will cause the rays to "spread" in slightly different direction, simulating the minor imperfections that exist in all real world objects. As before, the returned color from each ray is averaged to give the pixel it's color.

While it should be obvious 4X oversampling would produce an image with less of a "ray traced signature", at first glance this suggestion might seem completely impractical, given the huge computational expense of ray tracing to begin with – even though the paths of individual rays through the environment can be computed in parallel given the appropriate hardware.

For example, if we want to ray trace a 3840 by 2160 (4K) image we must trace all of the reflection and refraction interactions of each of the 8,294,400 rays (each of which can generate an exponential number of intersection calculations to the specified tracing depth). And, if we are creating an animation, we must repeat this process for every image, 30 to 60 frames a second if we want a reasonable animation displayed in real time (or 144 frames a second that some hard core gaming fans now seem to demand).[1] Thus, it seems foolhardy to even suggest using four rays per pixel.

Fortunately, oversampling, if properly implemented, doesn't have to be terribly expensive. Instead of sending one ray through the center of each pixel or through random region of the pixel, imagine each pixel as a little rectangle. If we project our rays through the corners of the pixels we can capture the effect of 4X oversampling each pixel without quadrupling the workload. In fact, for an N by M image, we need trace only $N + M + 1$ additional rays. Thus, for a 3840 by 2160 image, we need only trace 6,001 additional rays, meaning we get the benefit of 4X oversampling with only 0.07% more effort.



Ray tracing with efficient 4X pixel oversampling

Oversampling will produce blurry reflections (from reflected rays) and translucency (from refracted rays). If we implement shadow feeler rays (Section 18.4.3), oversampling shadow feeler rays can generate penumbra – the "fuzziness" around the edge of shadows.

Finally, there is a more than a bit of irony that ray tracing goes through so much effort to enable reflections and refractions (and shadows) and then we propose spending even more effort to "degrade" the quality reflections and refractions, but capturing "real world" imperfections result in greater realism.

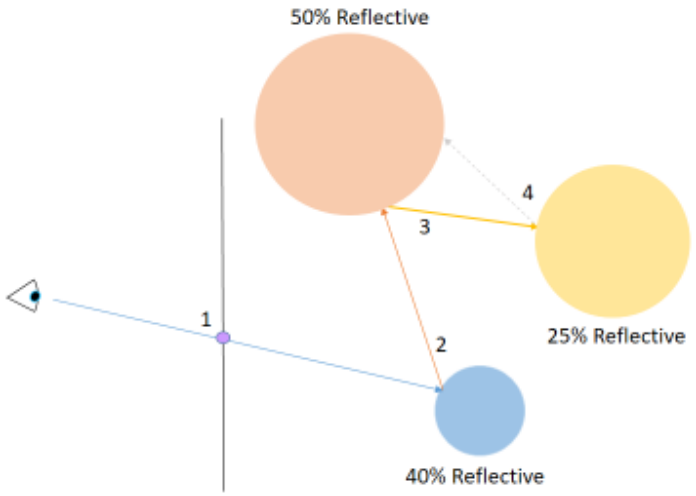
[1] This is why we can't have nice things. Or, more specifically, why ray tracing is only now (2021) starting to be incorporated into top-of-the-line video games. It is just so damn expensive, even with modern GPU hardware.

18.4.2 Adaptive Depth Control

Ray tracing can be incredibly expensive. Adaptive depth control is a technique that attempts to reduce computational expense without degrading the overall quality of the image. Viewed another way, adaptive depth control can often generate higher quality images within a fixed computational budget, when compared to standard ray tracing.

As presented in Section 18.1, rays are traced to some fixed depth. When you think about it, this "one size fits all" approach to computing pixel color isn't very efficient. The reason is that some objects in an environment may be highly reflective, while other objects may not be very reflective at all. As a result, it makes sense to trace some areas of the screen deeper than others.

Adaptive depth control expends the most effort where that effort will be most visible. While there still needs to be a maximum depth of recursion[2], adaptive depth control can terminate a recursive call early when the contribution of a reflected or refracted ray to the final intensity / color of a pixel drops below some fixed threshold and is deemed not worth additional effort.



Adaptive Depth Control: Given a 10% cutoff, Ray 4 would not be computed

For example, as illustrated above, let's say the original traced ray (Ray 1) intersects an object (e.g., a blue sphere) that is 40% reflective. Thus, the reflected ray (Ray 2) will generate 40% of the overall intensity / color of the rendered pixel. Furthermore, let's say the surface this reflected ray (Ray 2) intersects (e.g., a pink sphere) is 50% reflective. This means that the ray reflected from this surface (Ray 3), will contribute 20% (40% * 50%) to the final pixel's intensity / color. Following this reflected ray (Ray 3), let's say the surface it lands on (e.g., a yellow sphere) is 25% reflective. That means that if we trace the ray reflected from this intersection point (Ray 4), the impact of that ray on the intensity / color of the pixel would be only 5% (40% * 50% * 25%). Is calculating Ray 4 worth doing when that ray will only affect the intensity / color of the pixel by 5%? Given (an arbitrary) cutoff of 10% the answer would be "no" and we would terminate the recursion.

[2] A "hall of mirrors" could lead to infinite recursion between two (or more) 100% reflective surfaces without a fixed maximum depth of recursion.

18.4.3 Shadow Feeler Rays

Shadows are caused by one or more objects being located between a surface and the light source(s) illuminating that surface.

Shadows can be detected by casting a separate "shadow feeler ray" from the surface point being illuminated to each light source illuminating the scene. If a shadow feeler ray from the surface point to a light source intersects an opaque (non-translucent) object, then that surface point is in shadow and thus: (1) the diffuse and (2) specular components of the Phong illumination model are set zero for that light source, leaving only ambient lighting, when computing the local illumination of the surface point.

Since a shadow feeler ray is just as expensive as a normal traced ray, shadow computations can become very expensive for multiple light sources. Because of this expense, shadow feeler rays are not usually refracted or reflected – just attenuated. So, for example, if an object that is 25% translucent (i.e., 75% of its color derives from local and reflected components) is positioned between a surface point and a light source, light from that source reaching this surface point will be reduced by 75%. Said another way, the diffuse and specular components of that light's contribution to the Phong illumination model would be multiplied by 0.25 when determining the local color of that surface point. This is a simplification; when computing shadows all objects are considered to be perfect transmitters of light.

Even with these simplifications shadows are frequently "faked" in modern video games. In a forest, for example, a "random" pattern of light and dark areas may be applied to the ground to capture the overall look of sunlight passing through the overhead trees. This is MUCH cheaper than actually computing how the individual branches and trees block the sunlight, and is often nearly as visually compelling.

18.5 A Complete Example

In this section I present a complete (well, almost complete) example of ray tracing. The program code is quite old, dating back to the mid 1990's – yes before most of you were born. It is written in C using the SunView environment. Believe it or not, SunView won't look too strange to most of you, as the TKinter interface model is quite similar to SunView. (I have omitted some of the code that draws buttons and specifies call backs and what not as that code isn't really relevant to implementing ray tracing.

Besides being not at all object-oriented, this code has a number of limitations compared to what you are asked to do in Assignment 5.

- First, this codes does not implement the full Phong illumination model. When computing the local component of an object's color, It omits diffuse and specular and essentially limits itself to the ambient term only. This should be corrected in your assignment 5.
- Second, the checker board is non-reflective in this code: reflected rays are not computed. This should be corrected in your assignment 5.
- Third, shadow feeler rays are not implement in this code, so objects do not cast shadows. For extra credit on assignment 5 you should implement shadows.
- Finally, the program given below includes code to implement color table reduction. Way back when this program was written even high-end workstations had limited ability to display color. While the Sun workstation this code was originally written for could display 24 bit color (around 24 million different colors) it could only display 256 of those 24 million colors on the screen at once. Consider this code "dead code" and just ignore it.

```
    canvas_sw = window_create(frame, CANVAS,
                              WIN_BELOW,          panel_sw,
                              CANVAS_AUTO_SHRINK, FALSE,
                              CANVAS_WIDTH,       1200,
                              CANVAS_HEIGHT,      1200,
                              WIN_VERTICAL_SCROLLBAR,   bar1,
                              WIN_HORIZONTAL_SCROLLBAR, bar2,
                              0);

    window_main_loop(frame);
    exit(0);
}


static void
render_proc( /* ARGS UNUSED */)
{
    static void setup_drawing_canvas();
    static void trace_ray();
    static void put_pixel();

    int pixel_x, pixel_y, screen_x, screen_y;
    double  xs, ys, zs;       /* center of projection */
    int ray_i, ray_j, ray_k;  /* vector for light entering eye */
    int ir, ig, ib;           /* intensity of red, green, and blue primaries */
    int depth;                /* maximum ray depth for reflecting objects */

    depth = 5;                /* initialize maximum ray depth */

    setup_drawing_canvas();

    /* center of projection */
    xs = 0;  ys = 0;  zs = -800;

    for ( pixel_x = 1; pixel_x <= 1200; pixel_x++)
    {
       pw_batch_on(pw);    /* batch a vertical swipe */
       screen_x = pixel_x - 600;
       for ( pixel_y = 1; pixel_y <= 1200; pixel_y++)
       {
          screen_y = 600 - pixel_y;

          /* compute vector for ray from center of projection through pixel */
          ray_i = screen_x - xs;
          ray_j = screen_y - ys;
          ray_k = 0 - zs;

          /* trace the ray through the environment to obtain the pixel color */
          trace_ray ( 0, depth, xs, ys, zs,
                    (double) ray_i, (double) ray_j, (double) ray_k,
                    &ir, &ig, &ib);

          put_pixel(pixel_x, pixel_y, ir, ig, ib);
       }
       pw_batch_off(pw);  /* end of vertical swipe */
    }

}

static void
trace_ray(flag, level, xs, ys, zs, ray_i, ray_j, ray_k, ir, ig, ib)
int flag, level;
double xs, ys,zs, ray_i, ray_j, ray_k;
int *ir, *ig, *ib;
{
```

```
static int sphere1_intersection();
static int sphere2_intersection();
static int checkerboard_intersection();
static void checkerboard_point_intensity();
static void sphere1_point_intensity();
static void sphere2_point_intensity();

double t;                   /* distance of closest object */
double intersect_x;
double intersect_y;         /* intersection point of ray and object */
double intersect_z;
double obj_normal_x;
double obj_normal_y;        /* normal of closest object at intersect point */
double obj_normal_z;
int object_code;            /* integer code for the intersected object */
int object_r, object_g, object_b;  /* RGB values of an object */


if (level == 0)
{
   /* maximum depth exceeded -- return black */
   *ir = 0;  *ig = 0;  *ib = 0;
}
else
{
   /* check for intersection of ray with objects */
   /* and set rgb values corresponding to objects  */

   /* set distance of closest object initially to a very large number */
   t = 100000;

   /* initially no object has been intersected by the ray */
   object_code = -1;

   if ( checkerboard_intersection(
                xs,  ys,  zs,
                ray_i, ray_j, ray_k,
                &t,
                &intersect_x, &intersect_y, &intersect_z) )
       {
       object_code = 0;
       if (flag) printf(" checkerboard ");
       }

    if ( sphere1_intersection(
                xs,  ys,  zs,
                ray_i, ray_j, ray_k,
                &t,
                &intersect_x, &intersect_y, &intersect_z,
                &obj_normal_x, &obj_normal_y, &obj_normal_z) )
        {
        object_code = 1;
        if (flag) printf(" green_sphere ");
        }

     if ( sphere2_intersection(
                xs,  ys,  zs,
                ray_i, ray_j, ray_k,
                &t,
                &intersect_x, &intersect_y, &intersect_z,
                &obj_normal_x, &obj_normal_y, &obj_normal_z) )
        {
        object_code = 2;
        if (flag) printf(" reflective_sphere ");
        }
```

```
      switch(object_code)
      {
         case 0:
            checkerboard_point_intensity(
                    intersect_x, intersect_y, intersect_z,
                    ir, ig, ib);
            break;
         case 1:
            sphere1_point_intensity( level,
                    ray_i, ray_j, ray_k,
                    intersect_x, intersect_y, intersect_z,
                    obj_normal_x, obj_normal_y, obj_normal_z,
                    ir, ig, ib);
            break;
         case 2:
            sphere2_point_intensity( level,
                    ray_i, ray_j, ray_k,
                    intersect_x, intersect_y, intersect_z,
                    obj_normal_x, obj_normal_y, obj_normal_z,
                    ir, ig, ib);
            break;
         default:
            /* set pixel color to background color (light blue) */
            *ir = 150;  *ig = 150;  *ib = 255;
      }
   }
}



static int
checkerboard_intersection (xs, ys, zs, ray_x, ray_y, ray_z,
                           t, intersect_x, intersect_y, intersect_z)
double xs, ys, zs, ray_x, ray_y, ray_z;
double *t, *intersect_x, *intersect_y, *intersect_z;
{

  double a, b, c, x1, y1, z1;
  double denom, t_object, d, x, y, z;
  int    color_flag;

   /* normal of plane */
   a = 0;  b = 1;  c = 0;

   /* point on plane */
   x1 = 0;  y1 = -500;  z1 = 0;

   /* compute intersection of ray with plane */
   denom = a * ray_x + b * ray_y + c * ray_z;

   if (fabs(denom) <= 0.001)  return (0);  /* ray parallel to plane */
   else
   {
       d = a * x1 + b * y1 + c * z1;
       t_object = -( a * xs + b * ys + c * zs - d) / denom;
       x = xs + ray_x * t_object;
       y = ys + ray_y * t_object;
       z = zs + ray_z * t_object;
       if ( ( z < 0.0 ) || (z > 8000) || ( t_object < 0.0 ) )
          return (0);  /* no visible intersection */
       else if ( *t < t_object)  return (0);  /* another object is nearer */
       else
       {
          *t = t_object;
          *intersect_x = x;
          *intersect_y = y;
```

```
        *intersect_z = z;
        return(1);
      }
    }
}


static void
checkerboard_point_intensity(x, y, z, ir, ig, ib)
double x, y, z;
int *ir, *ig, *ib;
{
  int color_flag;

  /* a red and white checkered plane */

  /* compute color at intersection point */
  if (x >= 0.0 ) color_flag = 1; else color_flag = 0;
  if ( (fabs(fmod(x, 400.0))) > 200.0 ) color_flag = ! color_flag;
  if ( (fabs(fmod(z, 400.0))) > 200.0 ) color_flag = ! color_flag;
  if (color_flag)
  {
      /* red */
      *ir = 255;
      *ig = 0;
      *ib = 0;
  }
  else
  {
      /* white */
      *ir = 255;
      *ig = 255;
      *ib = 255;
  }
}


static int
sphere1_intersection (xs, ys, zs, ray_x, ray_y, ray_z,
                      t, intersect_x, intersect_y, intersect_z,
                      obj_normal_x, obj_normal_y, obj_normal_z)
double xs, ys, zs, ray_x, ray_y, ray_z;
double *t, *intersect_x, *intersect_y, *intersect_z;
double *obj_normal_x, *obj_normal_y, *obj_normal_z;
{
   double disc, ts1, ts2;
   double l, m, n, r, asphere, bsphere, csphere, tsphere;

   /* center of sphere */
   l = 0;  m = -400;  n = 600;

   /* radius of sphere */
   r = 100;

   /* compute intersection of ray with sphere */
   asphere = ray_x * ray_x + ray_y * ray_y + ray_z * ray_z;

   bsphere = 2 * ray_x * (xs-l) + 2 * ray_y * (ys-m) + 2 * ray_z * (zs-n);

   csphere = l * l + m * m + n * n + xs * xs + ys * ys + zs * zs
                    + 2 * ( -l * xs - m * ys - n * zs) - r * r;

   disc = bsphere * bsphere - 4 * asphere *csphere;

   if ( disc < 0 ) return (0);
   else
```

```
  {
     ts1 = (-bsphere + sqrt(disc)) / (2 * asphere);
     ts2 = (-bsphere - sqrt(disc)) / (2 * asphere);
     if (ts1 >= ts2) tsphere = ts2; else tsphere = ts1;
     if (*t < tsphere) return(0); /* another object is closer */
     else if (tsphere < 0.0) return(0); /* no visible intersection */
     else
     {
       *t = tsphere;
       *intersect_x = xs + ray_x * tsphere;
       *intersect_y = ys + ray_y * tsphere;
       *intersect_z = zs + ray_z * tsphere;
       *obj_normal_x = *intersect_x - l;
       *obj_normal_y = *intersect_y - m;
       *obj_normal_z = *intersect_z - n;
       return(1);
     }
   }
}



static void
sphere1_point_intensity(level, ray_x, ray_y, ray_z, x, y, z,
                                        nx, ny, nz, ir, ig, ib)
int level;
double ray_x, ray_y, ray_z, x, y, z, nx, ny, nz;
int *ir, *ig, *ib;
{
   double magnitude;
   double ray_x_norm, ray_y_norm, ray_z_norm;  /* normalized incoming ray */
   double nx_norm, ny_norm, nz_norm;           /* normalized surface vector */
   double cosine_phi;                          /* cosine of reflection angle */
   double rx, ry, rz;                          /* reflection vector */


   /*  normalize the incoming ray vector and the surface normal vector  */
   magnitude = sqrt( ray_x * ray_x + ray_y * ray_y + ray_z * ray_z );
   ray_x_norm = ray_x / magnitude;
   ray_y_norm = ray_y / magnitude;
   ray_z_norm = ray_z / magnitude;

   magnitude = sqrt ( nx * nx + ny * ny + nz * nz);
   nx_norm = nx / magnitude;
   ny_norm = ny / magnitude;
   nz_norm = nz / magnitude;

   /*  calculate reflection vector  */


   cosine_phi = (-ray_x_norm) * nx_norm +
                (-ray_y_norm) * ny_norm +
                (-ray_z_norm) * nz_norm ;

   if (cosine_phi > 0)
   {
      rx = nx_norm - (-ray_x_norm) / (2 * cosine_phi);
      ry = ny_norm - (-ray_y_norm) / (2 * cosine_phi);
      rz = nz_norm - (-ray_z_norm) / (2 * cosine_phi);
   }
   if (cosine_phi == 0)
   {
      rx = ray_x_norm;
      ry = ray_y_norm;
      rz = ray_z_norm;
    }
```

```
    if (cosine_phi < 0)
    {
      rx = -nx_norm + (-ray_x_norm) / (2 * cosine_phi);
      ry = -ny_norm + (-ray_y_norm) / (2 * cosine_phi);
      rz = -nz_norm + (-ray_z_norm) / (2 * cosine_phi);
    }

   /* trace the reflection ray */
  trace_ray(0, level - 1, x, y, z, rx, ry, rz, ir, ig, ib);

  /* add effect of local color */
  *ir = .7 * *ir + .3 * 120;
  *ig = .7 * *ig + .3 * 180;
  *ib = .7 * *ib + .3 * 0;
}



static int
sphere2_intersection (xs, ys, zs, ray_x, ray_y, ray_z,
                      t, intersect_x, intersect_y, intersect_z,
                      obj_normal_x, obj_normal_y, obj_normal_z)
double xs, ys, zs, ray_x, ray_y, ray_z;
double *t, *intersect_x, *intersect_y, *intersect_z;
double *obj_normal_x, *obj_normal_y, *obj_normal_z;
{
    double disc, ts1, ts2;
    double l, m, n, r, asphere, bsphere, csphere, tsphere;

    /* center of sphere */
    l = -200;  m = -300;  n = 1000;

    /* radius of sphere */
    r = 250;

    /* compute intersection of ray with sphere */
    asphere = ray_x * ray_x + ray_y * ray_y + ray_z * ray_z;

    bsphere = 2 * ray_x * (xs-l) + 2 * ray_y * (ys-m) + 2 * ray_z * (zs-n);

    csphere = l * l + m * m + n * n + xs * xs + ys * ys + zs * zs
                    + 2 * ( -l * xs - m * ys - n * zs) - r * r;

    disc = bsphere * bsphere - 4 * asphere *csphere;

    if ( disc < 0 ) return (0);
    else
    {
       ts1 = (-bsphere + sqrt(disc)) / (2 * asphere);
       ts2 = (-bsphere - sqrt(disc)) / (2 * asphere);
       if (ts1 >= ts2) tsphere = ts2; else tsphere = ts1;
       if (*t < tsphere) return(0); /* another object is closer */
       else if (tsphere < 0.0) return(0); /* no visible intersection */
       else
       {
         *t = tsphere;
         *intersect_x = xs + ray_x * tsphere;
         *intersect_y = ys + ray_y * tsphere;
         *intersect_z = zs + ray_z * tsphere;
         *obj_normal_x = *intersect_x - l;
         *obj_normal_y = *intersect_y - m;
         *obj_normal_z = *intersect_z - n;
         return(1);
       }
    }
```

```
}


static void
sphere2_point_intensity(level, ray_x, ray_y, ray_z, x, y, z,
                                    nx, ny, nz, ir, ig, ib)

int level;
double ray_x, ray_y, ray_z, x, y, z, nx, ny, nz;
int *ir, *ig, *ib;
{
   double magnitude;
   double ray_x_norm, ray_y_norm, ray_z_norm;  /* normalized incoming ray */
   double nx_norm, ny_norm, nz_norm;           /* normalized surface vector */
   double cosine_phi;                          /* cosine of reflection angle */
   double rx, ry, rz;                          /* reflection vector */


   /*    *ir = 200;  *ig = 0;  *ib = 250;    */  /* old color of sphere */

   /*  normalize the incoming ray vector and the surface normal vector  */
   magnitude = sqrt( ray_x * ray_x + ray_y * ray_y + ray_z * ray_z );
   ray_x_norm = ray_x / magnitude;
   ray_y_norm = ray_y / magnitude;
   ray_z_norm = ray_z / magnitude;

   magnitude = sqrt ( nx * nx + ny * ny + nz * nz);
   nx_norm = nx / magnitude;
   ny_norm = ny / magnitude;
   nz_norm = nz / magnitude;

   /*  calculate reflection vector  */
   /* algorthim from P & G page 525 */

   cosine_phi = (-ray_x_norm) * nx_norm +
                (-ray_y_norm) * ny_norm +
                (-ray_z_norm) * nz_norm   ;

   if (cosine_phi > 0)
   {
      rx = nx_norm - (-ray_x_norm) / (2 * cosine_phi);
      ry = ny_norm - (-ray_y_norm) / (2 * cosine_phi);
      rz = nz_norm - (-ray_z_norm) / (2 * cosine_phi);
   }
   if (cosine_phi == 0)
   {
      rx = ray_x_norm;
      ry = ray_y_norm;
      rz = ray_z_norm;
    }
    if (cosine_phi < 0)
    {
      rx = -nx_norm + (-ray_x_norm) / (2 * cosine_phi);
      ry = -ny_norm + (-ray_y_norm) / (2 * cosine_phi);
      rz = -nz_norm + (-ray_z_norm) / (2 * cosine_phi);
    }

    /* trace the reflection ray */
   trace_ray(0, level - 1, x, y, z, rx, ry, rz, ir, ig, ib);

   /* add effect of local color */
   *ir = .7 * *ir + .3 * 200;
   *ig = .7 * *ig + .3 * 100;
   *ib = .7 * *ib + .3 * 100;
}
```

```
static void
setup_drawing_canvas(/*ARGS UNUSED*/)
{
  int width =  (int)window_get(canvas_sw, CANVAS_WIDTH);
  int height = (int) window_get(canvas_sw, CANVAS_HEIGHT);
  static void setup_colormap();


  /* get the canvas_pixwin to draw into */
  pw = canvas_pixwin(canvas_sw);

  /* setup the color map */
  setup_colormap();

  /* erase the canvas */
  pw_writebackground(pw, 0, 0, width, height, PIX_SRC);

  /* center the window over the canvas                    */
  /*              The window is 600 by 600.               */
  /*              The canvas is 1200 by 1200              */
  /* place upper left hand corner of window at 300, 300 */

  /* center the window over the canvas */
  scrollbar_scroll_to(bar1, 300); /* vertically */
  scrollbar_scroll_to(bar2, 300); /* horizontally */

}



static void
setup_colormap(/*ARGS UNUSED*/)
{
   u_char       red[256],
                green[256],
                blue[256];

   int  i, red_i, green_i, blue_i;

   /* initialize the color vectors */

   red  [0] = 255;
   green[0] = 255;
   blue [0] = 255;

   i = 1;

   for (red_i = 80; red_i <= 244; red_i = red_i * 1.25)
   {
      for (green_i = 80; green_i <= 244; green_i = green_i * 1.25)
      {
         for (blue_i = 80; blue_i <= 244; blue_i = blue_i * 1.25)
         {
            red[i] = red_i;
            green[i] = green_i;
            blue[i] = blue_i;
            i++;
         }
      }
   }

   while (i <= 255)
   {
       red[i] = 0;
```

```
        green[i] = 0;
        blue[i] = 0;
        i++;
    }

    /* give a name to the color map we are creating for this pixwin */
    pw_setcmsname(pw,"temp");

    /* make our color map the active color map for this pixwin */
    pw_putcolormap(pw, 0, 256, red, green, blue);
}



static void
put_pixel (pixel_x, pixel_y, ir, ig, ib)
int pixel_x, pixel_y, ir, ig, ib;
{
    int color;

    /* clip out of range color intensities */
    if (ir > 244) ir = 244;
    if (ig > 244) ig = 244;
    if (ib > 244) ib = 244;
    if (ir < 80 ) ir = 80;
    if (ig < 80 ) ig = 80;
    if (ib < 80 ) ib = 80;

    /* compute color table index value */
    color = anint( (log10((double)ir)/log10(1.25)) - 19.6 ) * 6 * 6 +
            anint( (log10((double)ig)/log10(1.25)) - 19.6 ) * 6     +
            anint( (log10((double)ib)/log10(1.25)) - 19.6 )         + 1;

    /* plot pixel on screen */
    pw_put(pw, pixel_x, pixel_y, color);
}



static void
quit_proc(/* ARGS UNUSED */)
{
    window_destroy(frame);
}
```