Computer Graphics Course Notes

(Osama Hosam Eldeen)

(BOOK TITLE GOES HERE)

ISBN (XXXXXXXXXXXXX)

Printed in USA by 48HrBooks (www.48HrBooks.com)

Dedication

Replace this type with own wording, saying who you are dedicating this book to, and why. Replace this type with own wording, saying who you are dedicating this book to, and why. If you don't have a dedication, simply delete this entire page.

If you have further questions, contact 48HrBooks. Our regular business hours are Mon-Thurs. 8:30 am – 8 pm EST, and Friday 8:30 am – 5 pm. During these hours, you can reach us by phone, email or on-line chat. Outside of these hours, either call and leave a message or email us.

Phone: 800-231-0521

Email: info@48HrBooks.com

On-Line Chat: go to our website, **www.48HrBooks.com.**

Table of Contents

Contents

Foreword

Replace this wording with your own foreword.

Phone: 800-231-0521

Email: info@48HrBooks.com

Expected Chapters

1. Introduction, (include rendering pipeline from chapter. 4)
2. Display Systems
3. Geometric Transf.(rotate, translate and scale)
4. Geometric Primitives (lines, points, meshes, etc.) + Textures
5. Lights and Materials
6. 2D camera Transformation and Clipping
7. 3D Camera Transformation
8. Projection (Ortho, Perspective)
9. Animation, collision detection, bounding box collision, etc.

Syllabus

CS 451– Computer Graphics

Course Number	Course Title	Credit-Hours	Lecture Hours	Lab Hours	Prerequisite(s)
CS 451	**Computer Graphics**	4	3	2	CS 211

Course Objectives:

Having successfully completed this course, the student will be able to:

- Develop an understanding of the principles and methods used in creating, manipulating, storing and displaying of 2D and 3D graphics.
- Manipulate computer graphics using suitable tools.
- Distinguish between different graphics formats.
- Expertise concepts and techniques behind computer-generated animation.

Course Description:

The course begins with a discussion of formats, storage and display technologies before moving onto graphics rendering pipelines that demonstrate the process of generating 2D and 3D images. Thereafter, coverage of the techniques used in the creation and manipulation of 2D and 3D graphics (geometric, viewing and projection transformations), illumination, shading, and texture mapping techniques are provided. Following a discussion of graphical user interfaces, the creation and manipulation of computer-generated animations is introduced.

Topics Covered:

Lectures:

- WPF-based Computer Graphics vs. Human Visual Perception: frame buffer, Monitor and other hardware
- Color modelling, Image creation and display using both Raster and vector graphics
- Rendering Pipeline
- Curves and Surfaces Representations
- 2D & 3D Object Modeling using Meshes
- 2D & 3D Viewing and Modelling transformations
- Matter-Lighting interaction models
- Scan conversion algorithms: line, circle drawing, etc.
- Textures & Texture Mapping techniques & algorithms
- HSR, Shading, Blending and image processing techniques
- Interaction & animation techniques

Laboratories:

- Computer Graphics development platform
- Graphics Program structure (by examples)
- Graphic’s OpenGL-based programs
- . Project definition
- . Application analysis and studies:
  - Case Study 1: Pipeline, Modeling & Transformations
  - Case Study 2: Lighting, interaction & animation
  - Case Study 3: Textures & image processing
- Project follow up, guidance and assessment
- Project report exam, student presentation

Textbooks:

J. Foley, A. van Dam, S. Feiner, and J. Hughes, "Computer Graphics: Principles and Practice",3rd ed., Addison-Wesley, .2012 (or later ed.)

Chapter 1
Introduction

This is the course website http://osama-hosam.blogspot.com/p/cs451.html

Prerequisites: You will be writing programs

- Non-trivial data structures, pointers
- C++ with object oriented programming

Refer to the requirements and the needed textbooks in (Wolfe [13])

An ability to learn a programming library on your own

- OpenGL
- Comfortable with matrix algebra and calculus

Basic linear algebra used

Graphics Pipeline

– Core graphics pipeline: *Modeling transformation, viewing transformation, hidden surface removal, illumination / shading / textures, scan conversion / clipping*

- OpenGL
- Morphing, curves and surfaces, animation
- Not a course about graphic design, using graphics tools like PhotoShop or Maya

.

The concept of pipelining is illustrated in Figure 1.37 for a simple arithmetic calculation. In our pipeline, there is an adder and a multiplier. If we use this configuration to compute a + (b ∗ c), the calculation takes one multiplication and one addition—the same amount of work required if we use a single processor to carry out both operations. However, suppose that we have to carry out the same computation with many values of a, b, and c. Now, the multiplier can pass on the results of its calculation to the adder and can start its next multiplication while the adder carries out the second step of the calculation on the first set of data. Hence, whereas it takes the same amount of time to calculate the results for any one set of data, when we are working on two sets of data at one time, our total time for calculation is shortened markedly. Here the rate at which data flows through the system, the throughput of the system, has been doubled. Note that as we add more boxes to a pipeline, it takes more time for a single datum to pass through the system. This time is called the latency of the system; we must balance it against increased throughput in evaluating the performance of a pipeline.

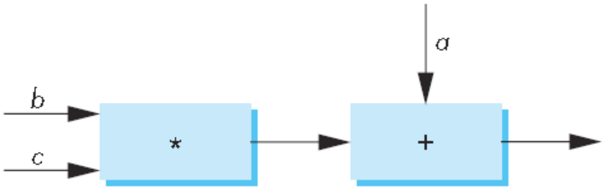


FIGURE 1.37 Arithmetic pipeline.

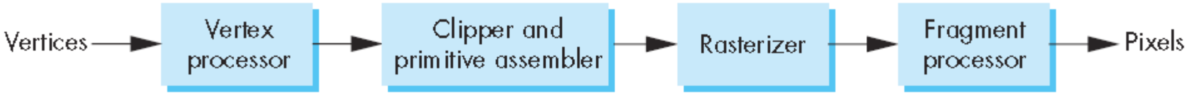


FIGURE 1.38 Geometric pipeline.

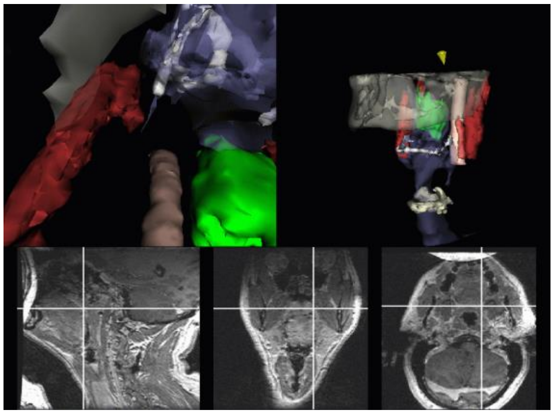
We start with a set of objects. Each object comprises a set of graphical primitives. Each primitive comprises a set of vertices. We can think of the collection of primitive types and vertices as defining the geometry of the scene. In a complex scene, there may be thousands—even millions—of vertices that define the objects. We must process all these vertices in a similar manner to form an image in the frame buffer. If we think in terms of processing the geometry of our objects to obtain an image, we can employ the block diagram in Figure 1.38, which shows the four major steps in the imaging process:

1. Vertex processing
2. Clipping and primitive assembly
3. Rasterization
4. Fragment processing

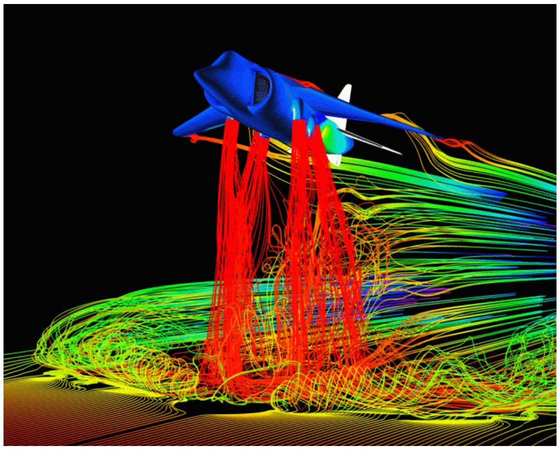
Applications

- Entertainment
- Medical Applications
- Scientific Visualization
- CAD
- Education
- Games

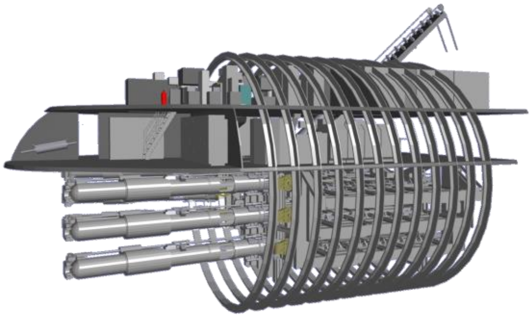
Entertainment



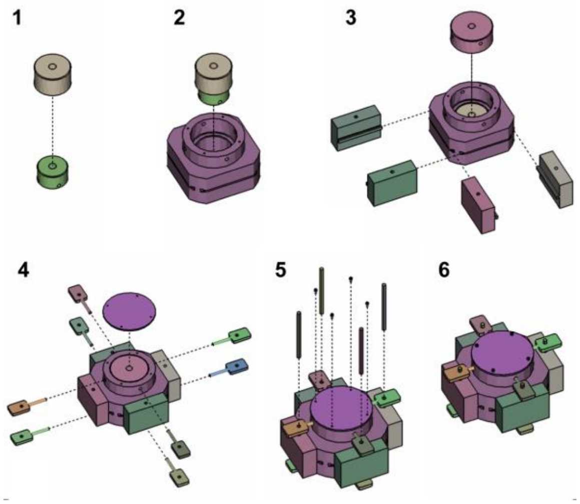
Medical applications



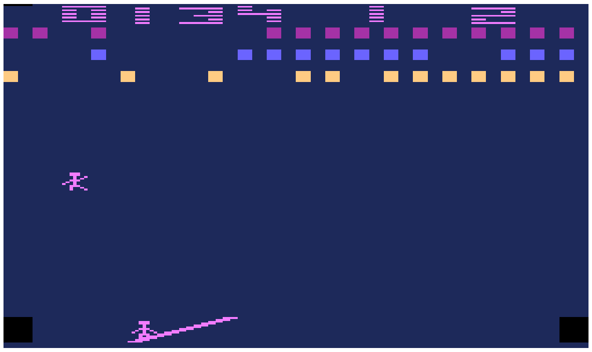
Scientific Visualization



Computer Aided Design (CAD)



Education: Designing Effective Step-By-Step Assembly Instructions



Games

Everyday Use

- Microsoft’s Whistler OS will use graphics seriously
- Graphics visualizations and debuggers
- Visualize complex software systems

Computer graphics Main task

generating 2D images of a 3D world represented in a computer.

- modeling: (shape) creating and representing the geometry of objects in the 3D world
- rendering: (light, perspective) generating 2D images of the objects
- animation: (movement) describing how objects change in time

Can we learn from history?

There are some who look at the things produced by nature through glass, or other surfaces or transparent veils. They trace outlines on the surface of the transparent medium… But such an invention is to be condemned in those who do not know how to portray things without it, no how to reason about nature with their minds… They are always poor and mean in every invention and in the composition of narratives, which is the final aim of this science

3.2. Depth perception of 2D image

Depth perception is based on 10 cues. These cues contain information which, when added to the 2D image projected onto the retina, allow us to relate the objects of the image to 3D space, so we can easily define which object is near and which is far away.

There are four physiological and six psychological cues [28].

Retinal Image Size, the larger an object image the closer it appears, it is applied only on the same objects of the same physical volume, if they are different in volume and be put in different distances from the eye, the observer may be confused, Figure 3.11.

Linear Perspective: the gradual reduction of image size as distance from the object increases, Figure 3.12;

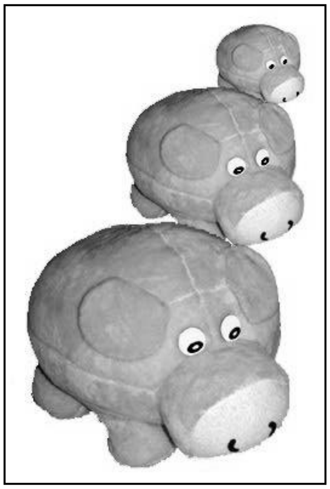
Aerial Perspective, the haziness of distant objects, Figure 3.13;

Overlapping(Occlusion), the effect where continuous outlines appear closer to the observer, Ancient Egyptians (Pharaoh's) painted their history in flat images containing overlapped objects to indicate the third dimension, Figure 3.14.

Shade and Shadows, the impression of convexity or concavity based on the fact that most illumination is from above, eye can entirely discriminate the object using only its Shade and Shadows, Figure 3.15. In computer systems the light patterns and shadows are used to infer the third dimension. For more information about shape from shading will be given in section 3.4.

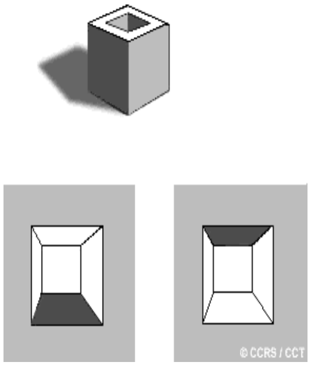
Texture Gradient, a kind of linear perspective describing levels of roughness of a uniform material as it recedes into the distance, Figure 3.16. Computer systems use the texture of an object to infer its shape. More information about shape from texture will be given in section 3.5.

Psychological cues are learned cues; therefore, they are assisted by experience. When combined, these cues enhance depth perception greatly. Stereo viewing of images usually combines binocular disparity and shade and shadow cues for effective depth perception.

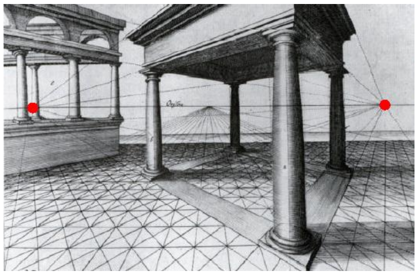


Retinal Image Size

Texture Gradient



Shade and Shadows



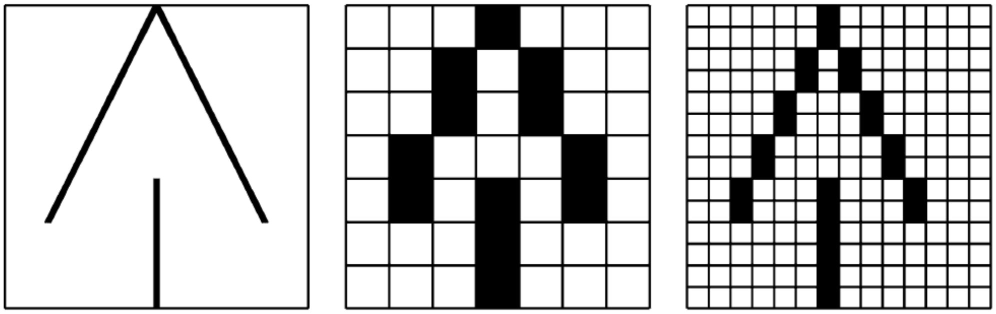
Linear Perspective

Aerial Perspective

Overlapping(Occlusion)

Chapter 2
Display Color Systems

7.1 Resolution



The tip of an arrow drawn as raster graphics in two different resolutions [11]

Image resolution is typically described in PPI, which refers to how many pixels are displayed per inch of an image.

Higher resolutions mean that there more pixels per inch (PPI), resulting in more pixel information and creating a high-quality, crisp image.

Images with lower resolutions have fewer pixels, and if those few pixels are too large (usually when an image is stretched), they can become visible like the image above (middle arrow)

7.2 Depth Buffering: Depth buffering is one way to stop things in the background from being drawn over the top

of things in the foreground. With depth buffering enabled, every pixel that is drawn knows how far away from the camera it is. It stores this distance as a number in the "depth buffer." When you draw a pixel over an existing pixel, OpenGL will look at the depth buffer to determine which pixel is closer to the camera. If the new pixel being drawn is closer to the camera, it will overwrite the existing pixel. If the preexisting pixel is closer to the camera, then the new pixel being drawn is discarded. That is, a preexisting pixel only gets overwritten if the new pixel is closer to the camera. This is called "depth testing." [1]

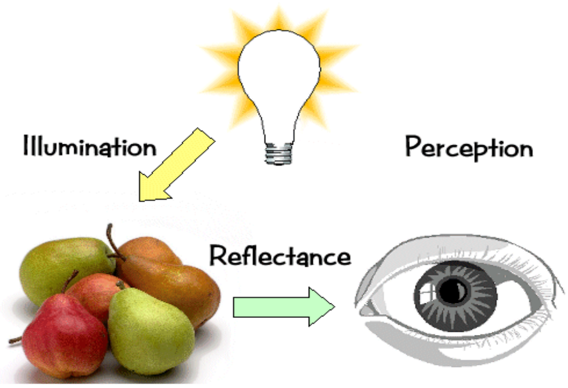
7.3 Color Systems

To understand how to make realistic images, we need a basic understanding of the physics and physiology of vision. Here we step away from the code and math for a bit to talk about basic principles

*Physics***:**

- Illumination
  - The light must hit the object

- Reflection
  - The light will be reflected to the human eyes, according to the object material and brightness.
- Perception
  - The human eyes differentiate between colors using the RGB light components.



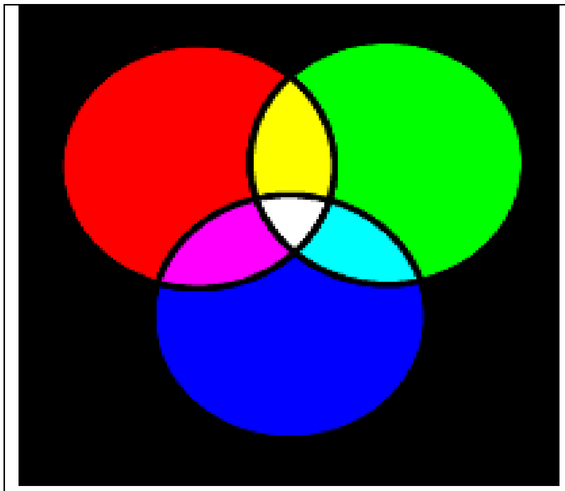
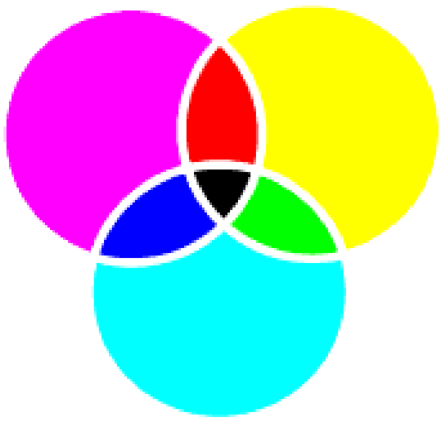
Specifying Color

Color perception usually involves three quantities:

- Hue: Distinguishes between colors like red, green, blue, etc
- Saturation: How far the color is from a gray of equal intensity
- Lightness: The perceived intensity of a reflecting object

Sometimes lightness is called brightness if the object is emitting light instead of reflecting it.

In order to use color precisely in computer graphics, we need to be able to specify and measure colors.

	
Additive (RGB)	Subtractive (CMYK)

Shining colored lights on a white ball	Mixing paint colors and illuminating with white light

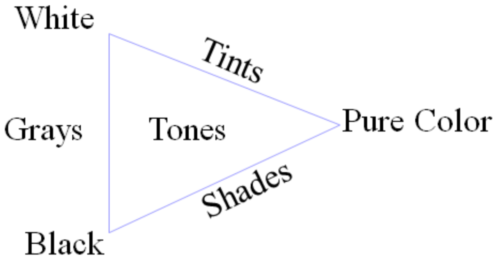
In RGB, all colors mean WHITE color, The absence of all colors is called BLACK color, suitable for screens and displays.

In CMYK, all colors means black, and the absence of all colors means white. Suitable for printers.

<u>How Do Artists Do It?</u>

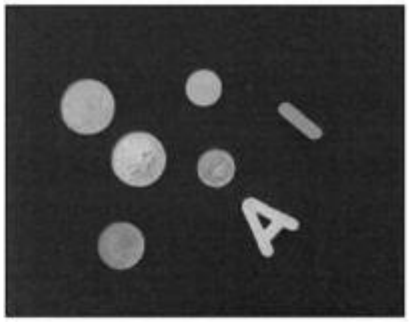
Artists often specify color as tints, shades, and tones of saturated (pure) pigments

- Tint: Obtained by adding white to a pure pigment, decreasing saturation
- Shade: Obtained by adding black to a pure pigment, decreasing lightness
- Tone: Obtained by adding white and black to a pure pigment



7.3.1 Grayscale Color

Take this image for example, a gray scale image, here you can see the intensity of the image gradually varies across the brightness spectrum. The intensity at any point is a whole number in range of [0,1] thus it can pick any value from the [0,1] range.

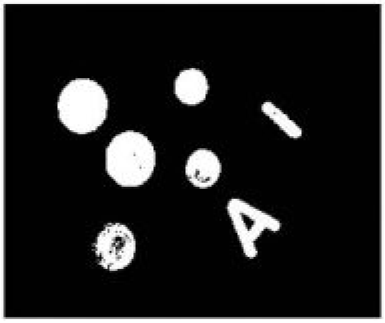


7.3.2 Binary Color

Now just imagine if were to set a **threshold** on the image i.e. if the intensity value is greater than 0.5, change it to 1 and if the intensity value is less than 0.5 change it to 0; then the image changes to a binary one.

The image shown below is a typical example of a binary image here the pixel can take only two values, either 0 or 1 [high or low].

The **threshold** value is used to decide the level of intensity value to be converted to white and black. Different values would give different results



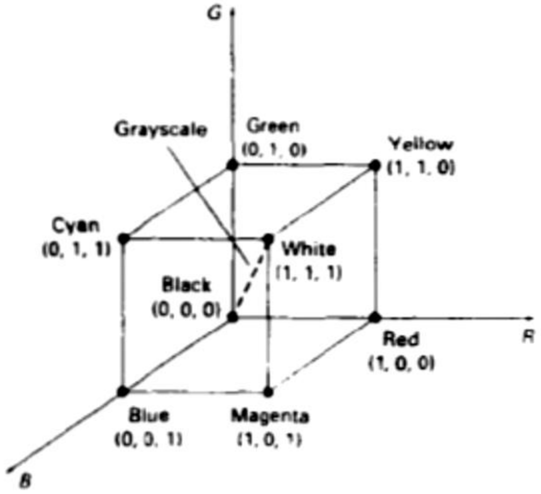
Example: convert the following matrix to binary, given the threshold= 0.5

0.024381	0.782003	0.756109	0.518727
0.988835	0.814833	0.087744	0.387661
0.332572	0.940600	0.365909	0.732100
0.755613	0.062953	0.405805	0.360955

7.3.3 RGB Color Space

Based on the **tristimulus theory** of vision, our eyes perceive color through the stimulation of three visual pigments in the cones of the eyes. These visual pigments have a peak sensitivity wavelengths of about 630 nm (**red**), 530 nm **(green),**and 450 nm (**blue**). By comparing intensities in a light source, we perceive the color of the light. This theory of vision is the basis for displaying color output on a video monitor using the three color primaries, red, green, and blue, referred to as the RGB color space.

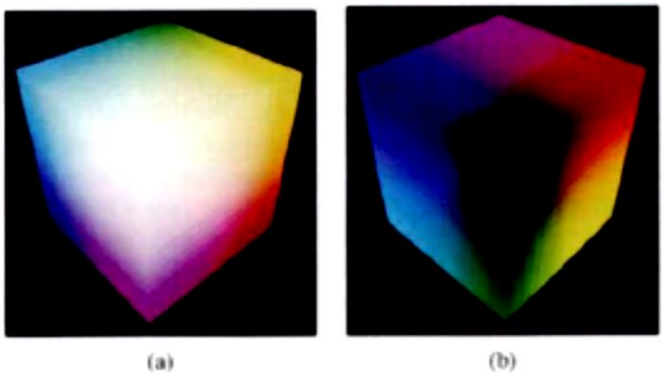
We can represent this model with the unit cube defined on R, G, and B axes, as shown in Fig. 15-11. The origin represents black, and the vertex with coordinates (1,1,l) is white. Vertices of the **cube** on the **axes** represent the primary colors, and the remaining vertices represent the complementary color for each of the primary colors.



The **RGB** color scheme is an additive model. Intensities of the primary colors are added to produce other colors. Each color paint within the bounds of the **cube** can be represented as the triple ***(R,*** G, B), where values for R, G, and ***B*** are assigned in the range from 0 to 1. Thus, a color **C,** is expressed in **RGB** components as

$$C_\lambda = R\mathbf{R} + G\mathbf{G} + B\mathbf{B}$$

The magenta vertex is obtained by adding red and blue to produce the triple (1, 0,l). and white at (1,l. 1) is the sum of the red, green, and blue vertices. Shades of gray are represented along the main diagonal of the cube from the origin (black) to the white vertex. Each point along this diagonal has an equal contribution from each primary color, so that a gray shade halfway between black and white is represented as **(0.5, 0.5, 0.5).** The color graduations along the front and top planes of the RGB **cube** are illustrated in Fig. 15-12.



(a) (b)

FIGURE 12-12 Two views of the RGB color cube. View (a) is along the gray-scale diagonal from white to black, and view (b) is along the gray-scale diagonal from black to white.

You can convert RGB color to grayscale color by using the following formula

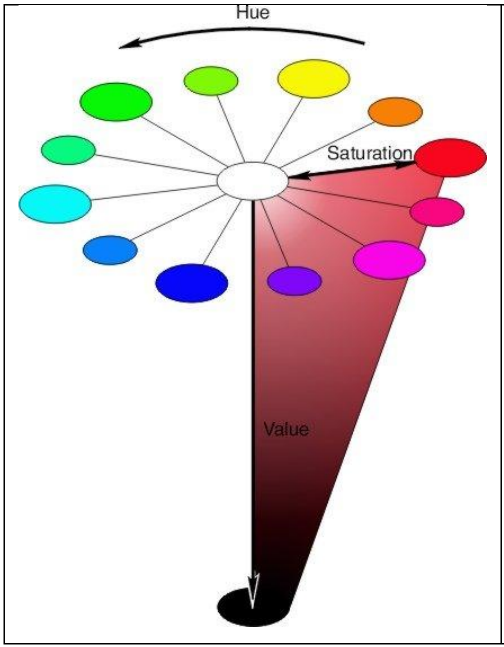
Grayscale_color = 0.2126 * R + 0.7152 * G + 0.0722 * B

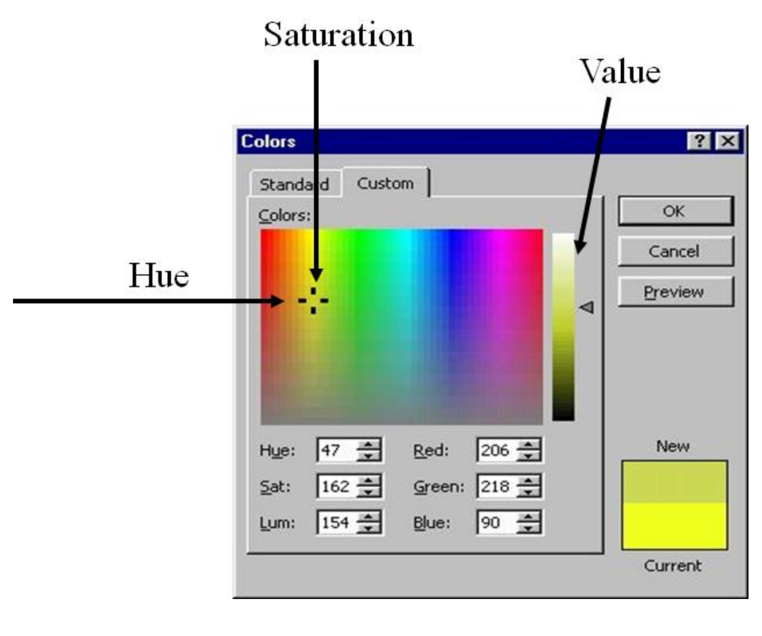
7.3.4 HSV Color Space

Computer scientists frequently use an intuitive color space that corresponds to tint, shade, and tone:

- *Hue* - The color we see (red, green, purple)
- *Saturation* - How far is the color from gray (pink is less saturated than red, sky blue is less saturated than royal blue)
- *Brightness* (Luminance) - How bright is the color (how bright are the lights illuminating the object?)

- Hue (H) is the angle around the vertical axis
- Saturation (S) is a value from 0 to 1 indicating how far from the vertical axis the color lies
- Value (V) is the height of the hexcone





Hue: 47
Sat: 162
Lum: 154
Red: 206
Green: 218
Blue: 90

7.3.5 The CMY Color Model

Cyan, magenta, and yellow are the complements of red, green, and blue

- We can use them as filters to subtract from white
- The space is the same as RGB except the origin is white instead of black

This is useful for hardcopy devices like laser printers

- If you put cyan ink on the page, no red light is reflected

- Add black as option (CMYK) to match equal parts CMY

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

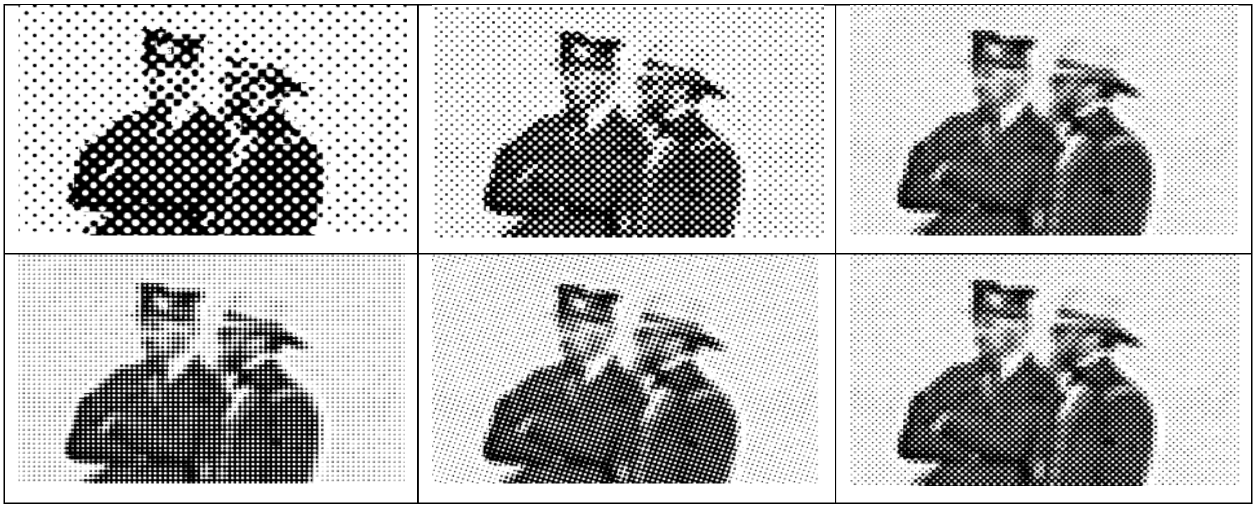
7.3.6 Halftoning and Dithering

A technique used in newspaper printing

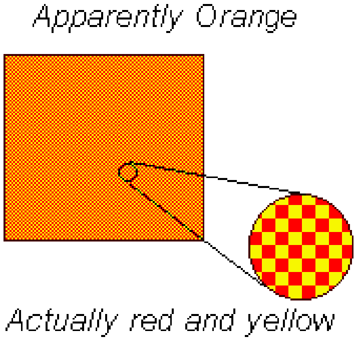
Only two intensities are possible, blob of ink and no blob of ink

But, the size of the blob can be varied

Also, the dither patterns of small dots can be used



Dithering is halftoning for color images, example of dithering is shown in the figure



Chapter 3
Geometric Transformations

Introduction

Specify transformations for objects

- – Allows definitions of objects in own coordinate systems
- – Allows use of object definition multiple times in a scene
    - Remember how OpenGL provides a transformation stack because they are so frequently reused

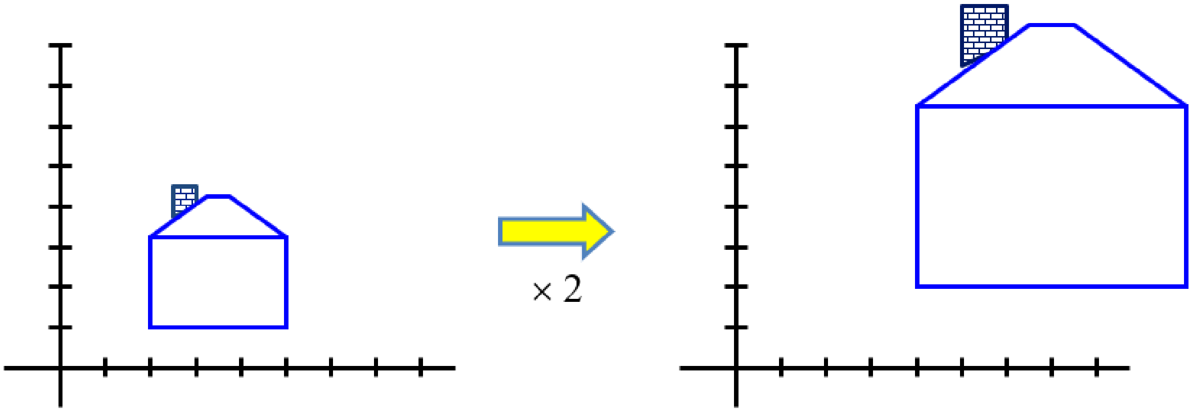
Contents

- 2D Transformations
    - – Basic 2D transformations
    - – Matrix representation
    - – Matrix composition
- 3D Transformations
    - – Basic 3D transformations
    - – Same as 2D

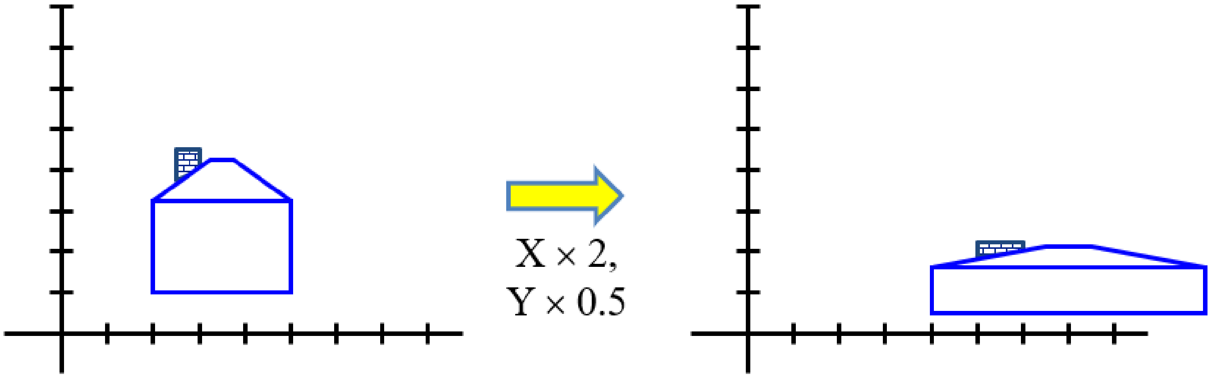
Scaling

- Scaling a coordinate means multiplying each of its components by a scalar

Uniform scaling means this scalar is the same for all components



- Non-uniform scaling: different scalars per component:



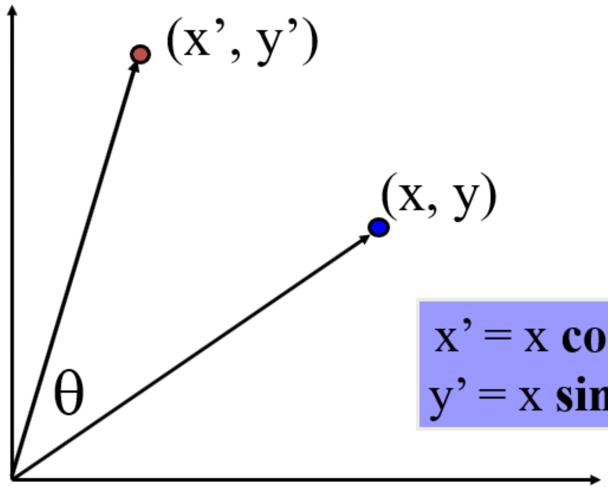
- Scaling operation:

$$\boxed{\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} ax \\ by \end{bmatrix}}$$

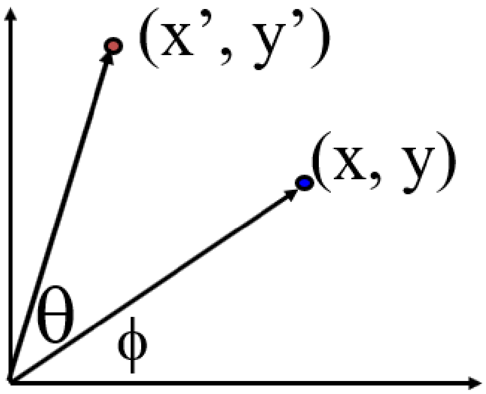
- Or, in matrix form:

$$\boxed{\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}}$$

2-D Rotation



$x' = x\,\mathbf{cos}(\theta) - y\,\mathbf{sin}(\theta)$
$y' = x\,\mathbf{sin}(\theta) + y\,\mathbf{cos}(\theta)$



$x = r\cos(\phi)$
$y = r\sin(\phi)$
$x' = r\cos(\phi + \theta)$
$y' = r\sin(\phi + \theta)$

Trig Identity…

$x' = r\cos(\phi)\cos(\theta) - r\sin(\phi)\sin(\theta)$
$y' = r\cos(\phi)\sin(\theta) + r\sin(\phi)\cos(\theta)$

Substitute…

$x' = x\,\mathbf{cos}(\theta) - y\,\mathbf{sin}(\theta)$
$y' = x\,\mathbf{sin}(\theta) + y\,\mathbf{cos}(\theta)$

- This is easy to capture in matrix form:

$$\boxed{\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}}$$

- Even though sin(θ) and cos(θ) are nonlinear functions of θ,
  - ***x' is a linear combination of x and y***
  - ***y' is a linear combination of x and y***

Basic 2D Transformations

- Translation:
  - x’ = x + t_x
  - y’ = y + t_y
- Scale:
  - x’ = x * s_x
  - y’ = y * s_y
- Rotation:
  - x’ = x*cos θ - y*sin θ
  - y’ = x*sin θ + y*cos θ

Matrix Representation

- Represent 2D transformation by a matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

- Multiply matrix by column vector ⇔ apply transformation to point

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad \Leftrightarrow \quad \begin{aligned} x' &= ax + by \\ y' &= cx + dy \end{aligned}$$

- Transformations combined by multiplication

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} \begin{bmatrix} i & j \\ k & l \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Matrices are a convenient and efficient way to represent a sequence of transformations!

- What types of transformations can be represented with a 2x2 matrix?

2D Identity?

$$x' = x \qquad y' = y \quad \Rightarrow \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

2D Scale around (0,0)?

$$\boldsymbol{x}' = \boldsymbol{s}_x * \boldsymbol{x} \qquad \boldsymbol{y}' = \boldsymbol{s}_y * \boldsymbol{y} \quad \Rightarrow \quad \begin{bmatrix} \boldsymbol{x}' \\ \boldsymbol{y}' \end{bmatrix} = \begin{bmatrix} \boldsymbol{s}_x & 0 \\ 0 & \boldsymbol{s}_y \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{bmatrix}$$

2D Rotate around (0,0)?

$$x' = \cos\Theta * x - \sin\Theta * y \qquad y' = \sin\Theta * x + \cos\Theta * y \quad \Rightarrow \quad \begin{bmatrix} \boldsymbol{x}' \\ \boldsymbol{y}' \end{bmatrix} = \begin{bmatrix} \cos\Theta & -\sin\Theta \\ \sin\Theta & \cos\Theta \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{bmatrix}$$

2D Shear?

$$\boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{sh}_x * \boldsymbol{y} \qquad \boldsymbol{y}' = \boldsymbol{sh}_y * \boldsymbol{x} + \boldsymbol{y} \quad \Rightarrow \quad \begin{bmatrix} \boldsymbol{x}' \\ \boldsymbol{y}' \end{bmatrix} = \begin{bmatrix} 1 & \boldsymbol{sh}_x \\ \boldsymbol{sh}_y & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{bmatrix}$$

2D Mirror about Y axis?

$$x' = -x \qquad y' = y \quad \Rightarrow \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

2D Mirror over (0,0)?

$$x' = -x \qquad y' = -y \quad \Rightarrow \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

2D Translation?

$$\boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{t}_x \qquad \boldsymbol{y}' = \boldsymbol{y} + \boldsymbol{t}_y \quad \Rightarrow$$

NO!

Only linear 2D transformations can be represented with a 2x2 matrix

Linear Transformations

- Linear transformations are combinations of …
  - Scale,
  - Rotation, $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$
  - Shear, and
  - Mirror
- Properties of linear transformations:
  - Satisfies:
  - Origin maps to origin
  - Lines map to lines
  - Parallel lines remain parallel
  - Ratios are preserved
  - Closed under composition

$$T(s_1\mathbf{p}_1 + s_2\mathbf{p}_2) = s_1T(\mathbf{p}_1) + s_2T(\mathbf{p}_2)$$

- Q: How can we represent translation as a 3x3 matrix?

$$\boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{t}_x$$

$$\boldsymbol{y}' = \boldsymbol{y} + \boldsymbol{t}_y$$

Homogeneous coordinates

- represent coordinates in 2 dimensions with a 3-vector

$$\begin{bmatrix} x \\ y \end{bmatrix} \xrightarrow{\text{homogeneous coords}} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

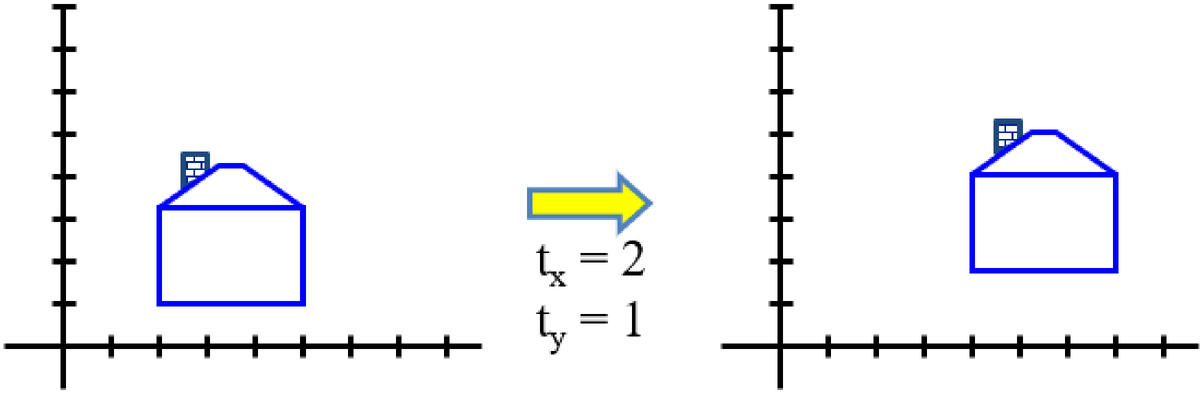
Homogeneous coordinates seem unintuitive, but they make graphics operations much easier

- Q: How can we represent translation as a 3x3 matrix?
- A: Using the rightmost column:

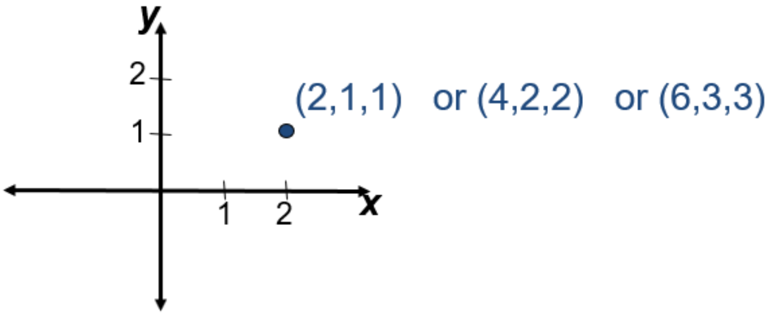
$$\textbf{\textit{Translation}} = \begin{bmatrix} 1 & 0 & \boldsymbol{t_x} \\ 0 & 1 & \boldsymbol{t_y} \\ 0 & 0 & 1 \end{bmatrix}$$

Translation

$$\begin{bmatrix} \boldsymbol{x'} \\ \boldsymbol{y'} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \boldsymbol{t_x} \\ 0 & 1 & \boldsymbol{t_y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{x} + \boldsymbol{t_x} \\ \boldsymbol{y} + \boldsymbol{t_y} \\ 1 \end{bmatrix}$$



- Add a 3rd coordinate to every 2D point
  - (x, y, w) represents a point at location (x/w, y/w)
  - (x, y, 0) represents a point at infinity
  - (0, 0, 0) is not allowed



Convenient coordinate system to represent many useful transformations

- Basic 2D transformations as 3x3 matrices

$\begin{bmatrix} \boldsymbol{x}' \\ \boldsymbol{y}' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \boldsymbol{t}_x \\ 0 & 1 & \boldsymbol{t}_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ 1 \end{bmatrix}$ Translate	$\begin{bmatrix} \boldsymbol{x}' \\ \boldsymbol{y}' \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{s}_x & 0 & 0 \\ 0 & \boldsymbol{s}_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ 1 \end{bmatrix}$ Scale
$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\Theta & -\sin\Theta & 0 \\ \sin\Theta & \cos\Theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$ Rotate	$\begin{bmatrix} \boldsymbol{x}' \\ \boldsymbol{y}' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & \boldsymbol{sh}_x & 0 \\ \boldsymbol{sh}_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ 1 \end{bmatrix}$ Shear

- Affine transformations are combinations of …
  - Linear transformations, and
  - Translations

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

- Properties of affine transformations:
  - Origin does not necessarily map to origin
  - Lines map to lines
  - Parallel lines remain parallel
  - Ratios are preserved
  - Closed under composition

Matrix Composition

- Transformations can be combined by matrix multiplication

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \left(\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\Theta & -\sin\Theta & 0 \\ \sin\Theta & \cos\Theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \right) \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

$$\mathbf{p}' \quad = \quad T(t_x,t_y) \qquad R(\Theta) \qquad S(s_x,s_y) \qquad \mathbf{p}$$

- Matrices are a convenient and efficient way to represent a sequence of transformations
  - General purpose representation

- Hardware matrix multiply

  p’ = (T * (R * (S***p**)))

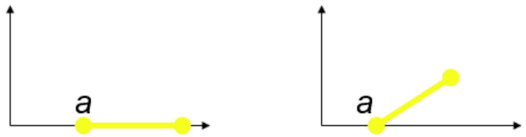
  p’ = (T*R*S) * **p**

- Be aware: order of transformations matters
  - Matrix multiplication is not commutative

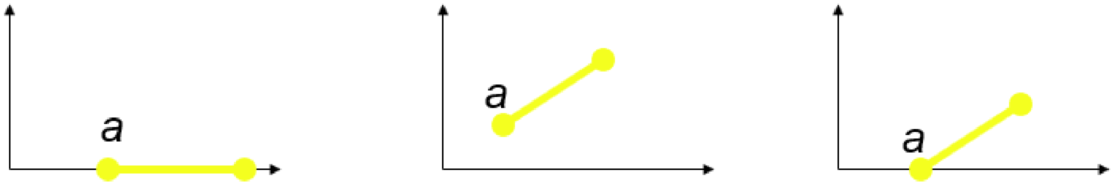
p’ = T * R * S * **p**

“Global” “Local”

- What if we want to rotate and translate?
  - Ex: Rotate line segment by 45 degrees about endpoint *a* *and lengthen*



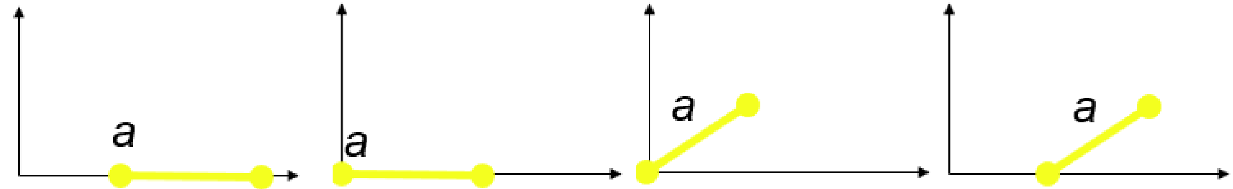
- Our line is defined by two endpoints
  - Applying a rotation of 45 degrees, R(45), affects both points
  - We could try to translate both endpoints to return endpoint *a* to its original position, but by how much?



Wrong R(45)

Correct T(-3) R(45) T(3)

- Isolate endpoint a from rotation effects
  - First translate line so *a* is at origin: T (-3)
  - Then rotate line 45 degrees: R(45)
  - Then translate back so *a* is where it was: T(3)



Will this sequence of operations work?

$$\begin{bmatrix} 1 & 0 & -3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} \cos(45) & -\sin(45) & 0 \\ \sin(45) & \cos(45) & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} a_x \\ a_y \\ 1 \end{bmatrix} = \begin{bmatrix} a'_x \\ a'_y \\ 1 \end{bmatrix}$$

- After correctly ordering the matrices
- Multiply matrices together
- What results is one matrix – store it (on stack)!
- Multiply this matrix by the vector of each vertex
- All vertices easily transformed with one matrix multiply

3D Transformations

- Same idea as 2D transformations
  - Homogeneous coordinates: (x,y,z,w)
  - 4x4 transformation matrices

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$ Identity	$$\begin{bmatrix} \mathbf{x'} \\ \mathbf{y'} \\ \mathbf{z'} \\ \mathbf{w} \end{bmatrix} = \begin{bmatrix} \mathbf{s}_x & 0 & 0 & 0 \\ 0 & \mathbf{s}_y & 0 & 0 \\ 0 & 0 & \mathbf{s}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{z} \\ \mathbf{w} \end{bmatrix}$$ Scale

$\begin{bmatrix} \boldsymbol{x'} \\ \boldsymbol{y'} \\ \boldsymbol{z'} \\ \boldsymbol{w} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \boldsymbol{t_x} \\ 0 & 1 & 0 & \boldsymbol{t_y} \\ 0 & 0 & 1 & \boldsymbol{t_z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ \boldsymbol{z} \\ \boldsymbol{w} \end{bmatrix}$ Translation	$\begin{bmatrix} x' \\ y' \\ z' \\ w \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$ Mirror about Y/Z plane
Rotate around Z axis:	$\begin{bmatrix} x' \\ y' \\ z' \\ w \end{bmatrix} = \begin{bmatrix} \cos\Theta & -\sin\Theta & 0 & 0 \\ \sin\Theta & \cos\Theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$
Rotate around Y axis:	$\begin{bmatrix} \boldsymbol{x'} \\ \boldsymbol{y'} \\ \boldsymbol{z'} \\ \boldsymbol{w} \end{bmatrix} = \begin{bmatrix} \cos\Theta & 0 & \sin\Theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\Theta & 0 & \cos\Theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ \boldsymbol{z} \\ \boldsymbol{w} \end{bmatrix}$
Rotate around X axis:	$\begin{bmatrix} x' \\ y' \\ z' \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\Theta & -\sin\Theta & 0 \\ 0 & \sin\Theta & \cos\Theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$

Reverse Rotations

- Q: How do you undo a rotation of θ, R(θ)?
- A: Apply the inverse of the rotation… $R^{-1}(\theta) = R(-\theta)$
- How to construct R-1(θ) = R(-θ)
  - Inside the rotation matrix: cos(θ) = cos(-θ)
    - The cosine elements of the inverse rotation matrix are unchanged
  - The sign of the sine elements will flip
- Therefore… $R^{-1}(\theta) = R(-\theta) = R^T(\theta)$

Summery

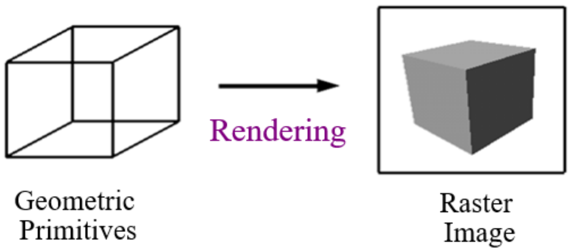
- Coordinate systems
  - World vs. modeling coordinates
- 2-D and 3-D transformations
  - Trigonometry and geometry
  - Matrix representations

    - Linear vs. affine transformations
- Matrix operations
    - Matrix composition

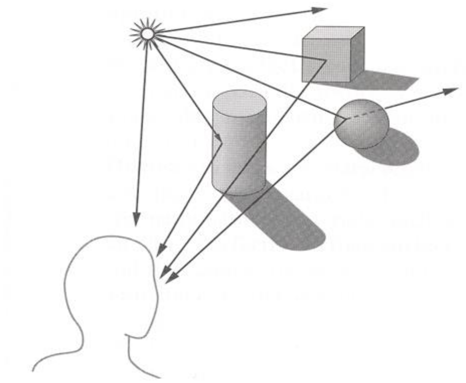
Chapter 4
Rendering Geometric Primitives

3.1 Rendering primitives

- Describe objects with points, lines, and surfaces
  - Compact mathematical notation
  - Operators to apply to those representations
- Render the objects: Generate an image from geometric primitives



Rendering



Issues involved in rendering (Lights, shades, etc.)

What issues must be addressed by a 3D rendering system?

- 3D scene representation
- 3D viewer representation (Object transformations, lighting simulation, Viewing Transform, Clipping, Projection, Rasterize)

3D Scene Representation

- Scene is usually approximated by 3D primitives
  - Point, Line segment, Polygon, Polyhedron, Curved surface, Solid object, etc.

3.2 Scene Representation with Geometric Primitives

Geometric primitives form the basic building blocks used to describe three-dimensional shapes. In this section, we introduce points, lines, and planes

3.2.1 Point

Specifies a location, Represented by three coordinates and Infinitely small

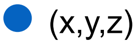
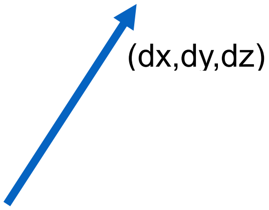
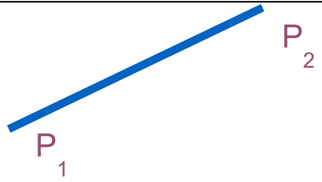
Example: the distance between the two points (8,2,6) and (3,5,7) is:

$= \sqrt{[\,(8-3)^2 + (2-5)^2 + (6-7)^2\,]}$

$= \sqrt{[\,5^2 + (-3)^2 + (-1)^2\,]}$

$= \sqrt{(\,25 + 9 + 1\,)}$

$= \sqrt{35} = 5.9$

		
3D point	3D Vector	Lines

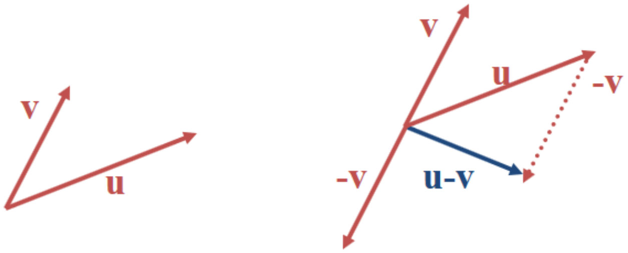
3.2.2 Vector

Specifies a direction and a magnitude

- Represented by three coordinates
- Magnitude ||V|| = sqrt(dx dx + dy dy + dz dz)
- Has no location

Vector Addition/Subtraction: operation u + v, with:

- Identity 0 : v + 0 = v
- Inverse - : v + (-v) = 0
- Addition uses the "parallelogram rule":



Vector Space: Vectors define a vector space

- They support vector addition
  - Commutative x + y = y + x
  - and associative (x + y) + z = x + (y + z)
  - Possess identity and inverse
- They support scalar multiplication
  - Associative, distributive
  - Possess identity

Affine Spaces: Vector spaces lack position and distance

- They have magnitude and direction but no location
- Combine the point and vector primitives
  - Permits describing vectors relative to a common location
- A point and three vectors define a 3-D coordinate system
- Point-point subtraction yields a vector

Points + Vectors: Points support these operations

- Point-point subtraction: *Q* - *P* = **v,** Result is a vector pointing from P to Q
- Vector-point addition: *P* + **v** = *Q*, Result is a new point
- Note that the addition of two points is not defined

Dot Product: Permits the computation of distance and angles

- The dot product or, more generally, inner product of two vectors is a scalar:

$$\mathbf{v}_1 \bullet \mathbf{v}_2 = \mathbf{x}_1\mathbf{x}_2 + \mathbf{y}_1\mathbf{y}_2 + \mathbf{z}_1\mathbf{z}_2 \quad \text{(in 3D)}$$

- Useful for many purposes
  - Computing the length (Euclidean Norm) of a vector: length(**v**) = ||**v**|| = sqrt(**v • v)**
  - *Normalizing* a vector, making it unit-length: **v = v / ||v||**
  - Computing the angle between two vectors: **u • v** = |**u**| |**v**| cos(θ)
  - Checking two vectors for orthogonality, **u • v = 0.0**

Example. Find the angle between two vectors u = (3; 4; 0) and v = (4; 4; 2).

Solution: calculate dot product of vectors:

u·v = 3 · 4 + 4 · 4 + 0 · 2 = 12 + 16 + 0 = 28.

Calculate vectors magnitude:

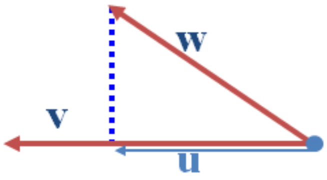
$|u| = \surd[3^2 + 4^2 + 0^2]= \surd 9 + 16 = \surd 25 = 5$

$|v| = \surd[4^2 + 4^2 + 2^2]= \surd 16 + 16 + 4 = \surd 36 = 6$

Calculate the angle between vectors:

$$\cos \alpha = \frac{u \cdot v}{|u| \cdot |v|} = \frac{28}{5 \cdot 6} = \frac{14}{15}$$

- Projecting one vector onto another
  - If **v** is a unit vector and we have another vector, **w**
  - We can project **w** perpendicularly onto **v**
  - And the result, **u**, has length **w • v**

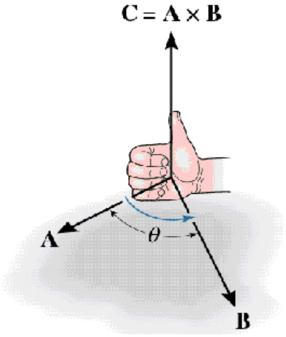


$$\begin{aligned}\|u\| &= \|w\|\cos(\theta)\\ &= \|w\|\left(\frac{v \bullet w}{\|v\|\|w\|}\right)\\ &= v \bullet w\end{aligned}$$

  - Is commutative, u • v = v • u
  - Is distributive with respect to addition, u • (v + w) = u • v + u • w

Cross Product: The cross product or vector product of two vectors is a vector:

$$\mathbf{v}_1 \times \mathbf{v}_2 = \begin{vmatrix} u_x & u_y & u_z \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix}_{\text{determinant}} = \begin{bmatrix} y_1 z_2 - y_2 z_1 \\ -(x_1 z_2 - x_2 z_1) \\ x_1 y_2 - x_2 y_1 \end{bmatrix}$$



- The cross product of two vectors is orthogonal to both
- Right-hand rule dictates direction of cross product

Example: show how to find normal vector of plane with defined three points

Cross Product Right Hand Rule: Orient your right hand such that your palm is at the beginning of A and your fingers point in the direction of A

- Twist your hand about the A-axis such that B extends perpendicularly from your palm

- As you curl your fingers to make a fist, your thumb will point in the direction of the cross product

Other helpful formulas: Length = sqrt $(x2 - x1)^2 + (y2 - y1)^2$

- Midpoint, p2, between p1 and p3, p2 = ((x1 + x3) / 2, (y1 + y3) / 2))

- Two lines are perpendicular if:
  - M1 = -1/M2
  - cosine of the angle between them is 0
  - Dot product = 0

3.2.3 Line Segment

Use a linear combination of two points, Parametric representation:

P = P1 + t (P2 - P1), $(0 \leq t \leq 1)$

3.2.4 Ray

Line segment with one endpoint at infinity

- Parametric representation: $P = P_1 + t V$, $(0 <= t < \infty)$

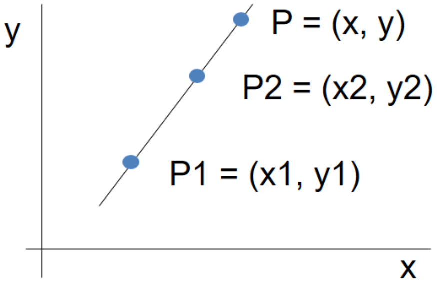
3.2.5 Line

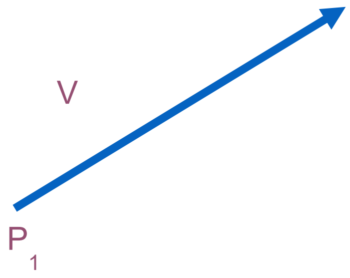
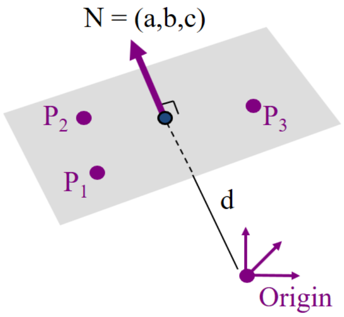
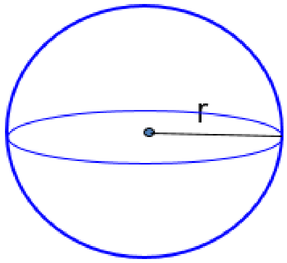
Line segment with both endpoints at infinity

- Parametric representation: $P = P_1 + t V$, $(-\infty < t < \infty)$

3D Line – Slope Intercept: Slope m = rise / run= (y - y1) / (x - x1) = (y2 - y1) / (x2 - x1)

- Solve for y: y = [(y2 - y1)/(x2 - x1)]x + [-(y2-y1)/(x2 - x1)]x1 + y1
- or: y = mx + b



		
3D ray	3D plane	3D sphere

3.2.6 Plane

A linear combination of three points， Implicit representation:

- $ax + by + cz + d = 0$, or $P\cdot N + d = 0$
- N is the plane “normal”, Unit-length vector and Perpendicular to plane

3.2.7 Sphere

All points at distance “r” from point “(c_x, c_y, c_z)”

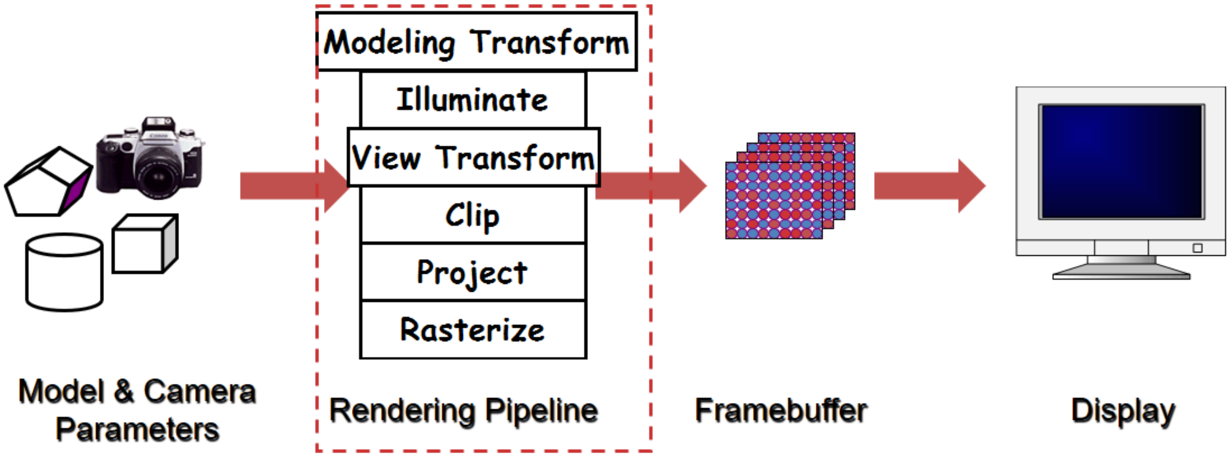
- Implicit representation: $(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2$
- Parametric representation:

  $x = r\cos(\phi)\cos(\Theta) + c_x$, $y = r\cos(\phi)\sin(\Theta) + c_y$, $z = r\sin(\phi) + c_z$

3.3 3D Viewer Representation

We’ve learned about transformations, but they are used in three ways:

- Modeling transforms
- Viewing transforms (Move the camera)
- Projection transforms (Change the type of camera)



3.3.1 Modeling transforms

- Size, place, scale, and rotate objects and parts of the model w.r.t. each other
- Object coordinates -> world coordinates

3.3.2 Illumination transform

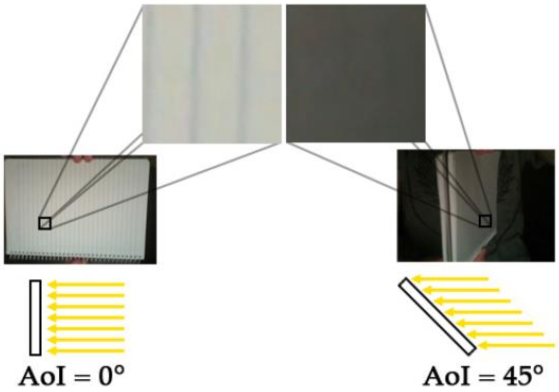
As the notepad rotates away from its brightest position, the surface gets darker. The angle at which the rays of light hit the surface is called the *angle of incidence* (AoI). The angle of incidence affects the brightness of the surface.

Maximum brightness occurs when the surface is *perpendicular* to the light rays (AoI = 0°). Complete darkness occurs when the surface is *parallel* to the light rays (AoI = 90°). If the AoI is greater than 90°, then the ray is hitting the *back* of the surface. If light is hitting the back, then it's not hitting the front, so the pixel should also be completely dark.

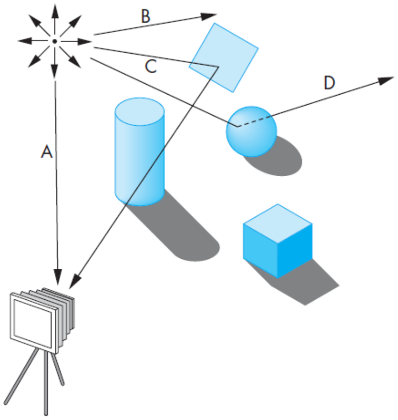
If we represent the brightness as a single number, where 0.0 is completely dark and 1.0 is maximum brightness, then it's easy to calculate based on the cosine of the AoI. The formula is brightness = cos(AoI) . Let's have a look at the cosine of some angles, just to prove that it works:

```
cos( 0°) = 1.00 (100% of maximum brightness)
cos( 5°) = 0.98 ( 98% of maximum brightness)
cos( 45°) = 0.71 ( 71% of maximum brightness)
cos( 85°) = 0.09 ( 9% of maximum brightness)
cos( 90°) = 0.00 (Completely dark)
cos(100°) = -0.17 (Completely dark. Negative value means light is hitting the back
side)
```

The angle on incident (AoI) can be simply calculated using the dot product of the light to surface vector and the surface normal vector

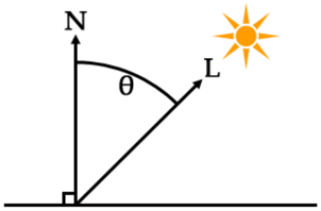


The brightness of the surface w.r.t viewer is proportional to the light's angle of incident (AoI).



Ray interactions. Ray A enters camera directly. Ray B goes off to infinity. Ray C is reflected by a mirror. Ray D goes through a transparent sphere.

Surface Normals: In order to calculate the AoI, we first need to know the direction that each surface is facing. The direction that a surface is facing is called the normal of that surface. Normals are unit vectors that are perpendicular (at right angle, 90°) to a surface. The angle of incidence is defined as the angle between the surface normal, and the direction to the light source.



N = the surface normal vector
L = a vector from the surface to the light source
θ = the angle of incidence

Surface Normals are calculated with the cross product of any two lines on the object surface. The vector from the surface to the light source, L, can be calculated with vector subtraction, like so:

L=LightPosition – SurfacePosition

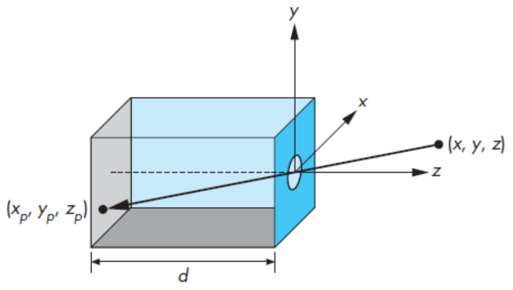
Ray Tracing: We include the viewer in the figure because we are interested in the light that reaches her eye. For example, if the source is visible from the camera, some of the rays go directly from the source through the lens of the camera and strike the film plane. Most rays, however, go off to infinity, neither entering the camera directly nor striking any of the objects. These rays contribute nothing to the image, although they may be seen by some the viewer. The remaining rays strike and illuminate objects. These rays can interact with the objects' surfaces in a variety of ways. For example, if the surface is a mirror, a reflected ray might—depending on the orientation of the surface—enter the lens of the camera and contribute to the image. Other

surfaces scatter light in all directions. If the surface is transparent, the light ray from the source can pass through it and may interact with other objects, enter the camera, or travel to infinity without striking another surface. Figure 1.18 shows some
of the possibilities.

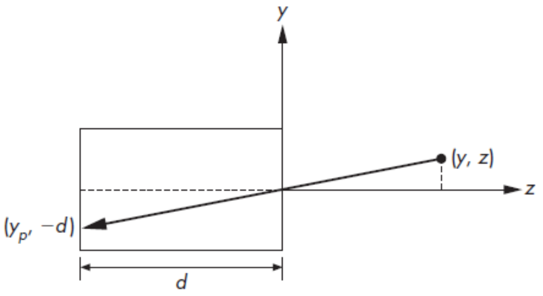
3.3.3 Viewing transform

First, we describe different camera models to understand camera parameters, accordingly we present camera transformations.

1) **Pinhole Camera**: The most common model is pin-hole camera A *pinhole camera* is a box with a small hole in the center of one side of the box; the film is placed inside the box on the side opposite the pinhole. Suppose that we orient our camera along the *z*-axis, with the pinhole at the origin of our coordinate system. We assume that the hole is so small that only a single ray of light, emanating from a point, can enter it. The film plane is located a distance *d* from the pinhole.



Pin-Hole camera Model



Side view of the pin-hole camera model

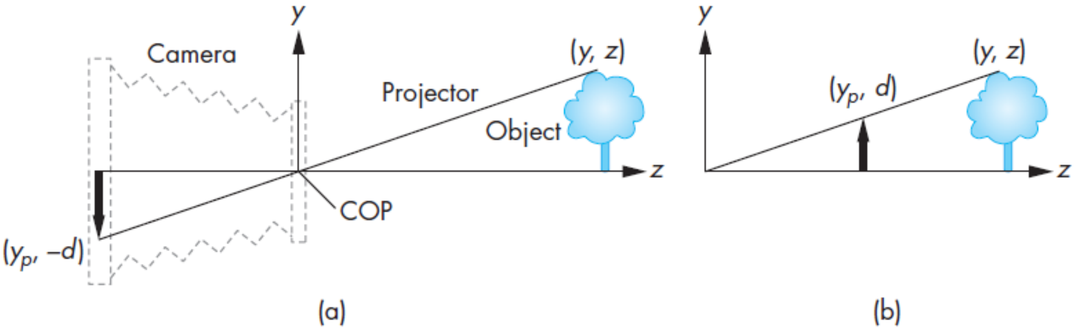
The point *(xp, yp, −d)* is called the <u>*projection*</u> of the point *(x, y, z)*. The <u>*field of view*</u> of our camera is the angle made by the largest object that our camera can image on its film plane. By replacing the pinhole with a lens, we solve the two problems of the pinhole camera. First, the lens gathers more light than can pass through the pinhole. The larger the aperture of the lens, the more light the lens can collect.

Second, by picking a lens with the proper focal length—a selection equivalent to choosing *d* for the pinhole camera—we can achieve any desired angle of view (up to 180 degrees).

2) **Synthetic-camera model:** The image is formed on the film plane at the back of the camera, Note that the image of the object is flipped relative to the object. Whereas with a real camera we would simply flip the film to regain the original orientation of the object, with our synthetic camera we can avoid the flipping by a simple trick. We draw another plane in front of the lens (Figure 1.24(b)) and work in three dimensions, as shown in Figure 1.25.

We find the image of a point on the object on the virtual image plane by drawing a line, called a **projector**, from the point to the center of the lens, or the **center of projection**

(**COP**). Note that all projectors are rays emanating from the center of projection. In our synthetic camera, the virtual image plane that we have moved in front of the lens is called the **projection plane**.



Equivalent views of image formation. (a) Image formed on the back of the camera. (b) Image plane moved in front of the camera.

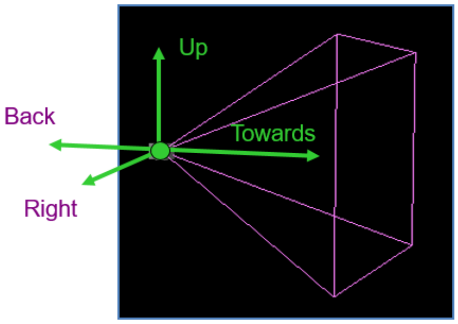
Viewing Transformations: Rotate & translate the world to lie directly in front of the camera

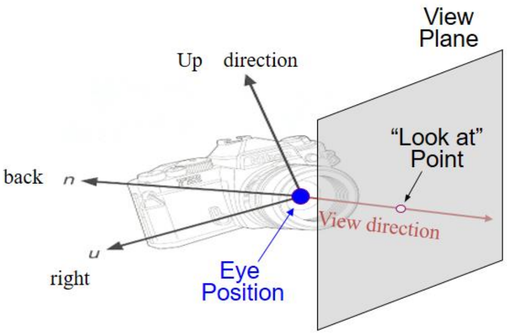
- Typically place camera at origin
- Typically looking down -Z axis
  - World coordinates -> view coordinates

Camera Parameters: Position

  - Eye position (px, py, pz)
  - Orientation (View direction (dx, dy, dz), Up direction (ux, uy, uz))

- Aperture
  - Field of view (xfov, yfov)
- Film plane
  - “Look at” point, View plane normal





View Frustum

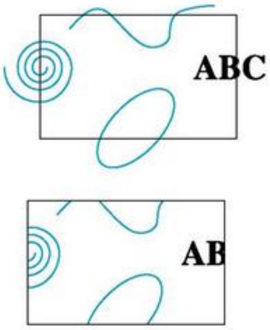
Camera parameters

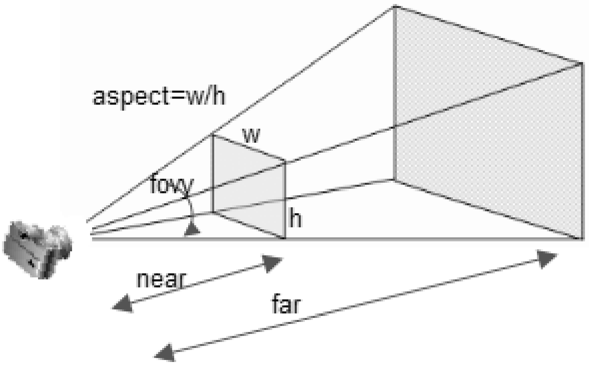
3.3.4 Clipping Transform

Anything outside the viewing frustum will be clipped. The frustum has the following parameters:

- The first argument "field of view" argument. This is an angle, in radians, that specifies how wide the camera's vision is. Instead of specifying radians directly, A large field of view means the camera can see a lot of the scene, so the camera appears to be zoomed out. A small field of view means the camera can only see a small portion of the scene, so it appears to be zoomed in.
- The second argument is the "aspect" argument, which specifies the aspect ratio of the view. This is almost always set to width/height of the window.
- The second last argument is the "near plane." The near plane is the front of the clip volume, and the value 0.1 says that the near plane is 0.1 units away from the camera. Anything that is closer to the camera than 0.1 it will not be visible. The near plane must be a value greater than zero.
- The last argument is the "far plane", which is the back of the clip volume. For example if far plane = 10, the value 10.0 says that the camera will display everything that is within 10 units of the camera. Anything further than 10 units away will not be visible.

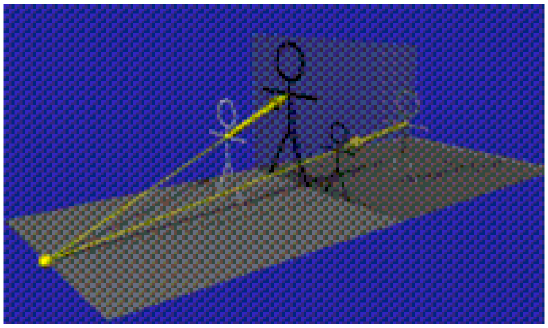
A frustum is like a pyramid with the top cut off. The flat base of the pyramid is the far plane. The flat part where the top has been cut off is the near plane. The field of view is the skinniness or fatness of the frustum. Anything inside the frustum will be displayed on the screen, and anything outside will be hidden.



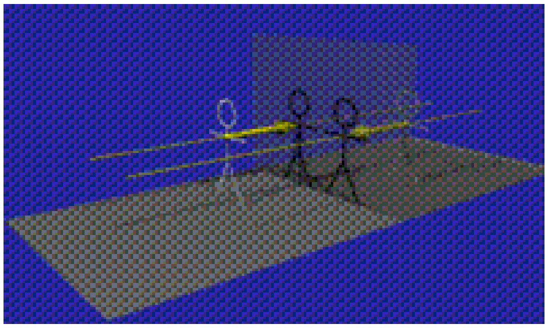


3.3.5 Projection Transform

- Apply perspective foreshortening
  - Distant = small: the pinhole camera model
- View coordinates a screen coordinates



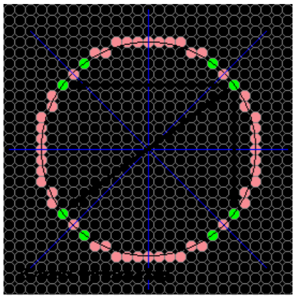
Perspective Camera

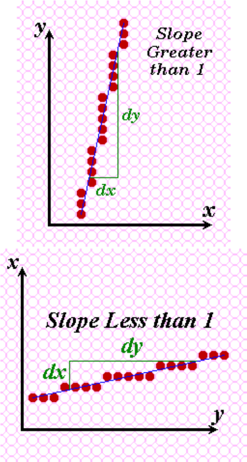


Orthographic Camera

3.3.6 Rasterize

Convert screen coordinates to pixel colors

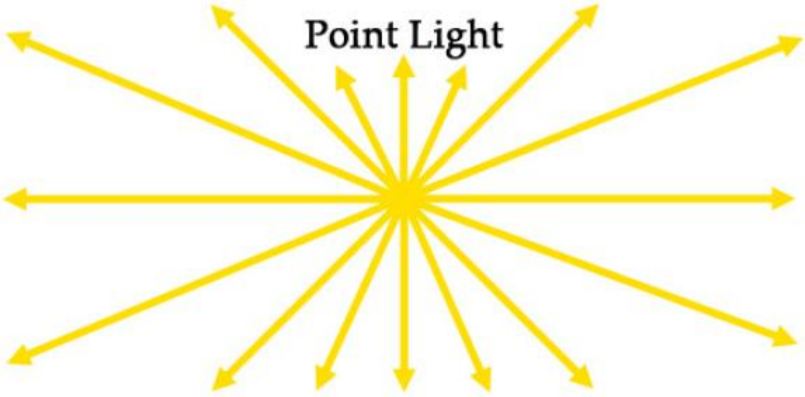




Chapter 5
Matter-Lighting Interaction

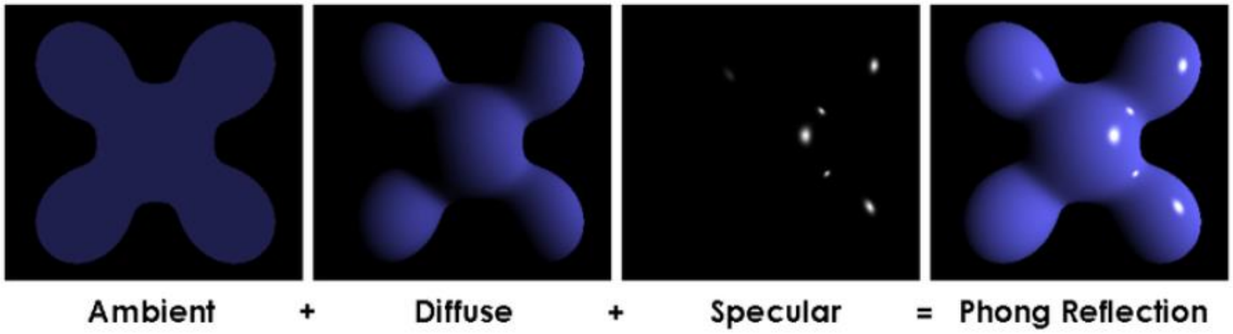
5.1 Point Lights

Point lights radiate light outwards in all directions from a single point, much like a candle. There are other common types of lights, such as directional lights and spotlights



Phong Reflection Model: The Phong reflection model provides a method of calculating the color of a pixel, based on the parameters of a light source and a surface. The light source parameters include the

- position/direction of the light, and the
- color/intensity of the light.



The surface parameters include

- the color of the surface, the
- direction the surface is facing (a.k.a the normal)
- , and the "shininess" of the surface.

The Phong reflection model has three components: ambient, diffuse, and specular.

- The **diffuse** component is the most important one, as you can see from the image above.
- The **ambient** component is used to stop the unlit, back sides of objects from being pure black, because pure black looks artificial in most 3D scenes.
- The **specular** component is what makes an object look shiny or dull.

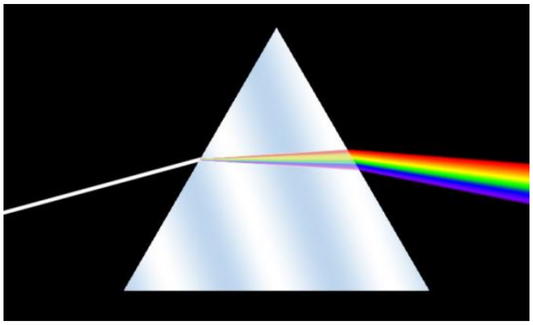
5.2 Diffuse Coefficient

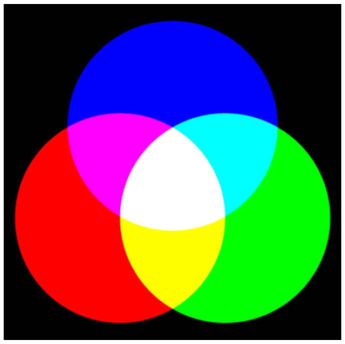
The Phong reflection model has three components, diffuse, ambient and specular. Here we present the diffuse component and how to calculate the diffuse coefficient.

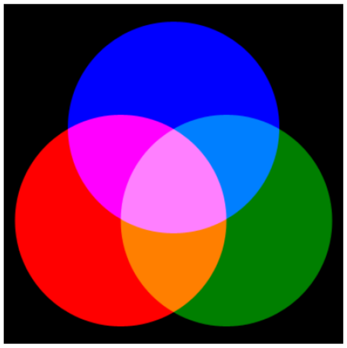
5.2.1 Light Intensities

The Phong reflection model is loosely based on the way that light behaves in the real world. So, in order to understand lighting in computer graphics, we have to understand a little bit about the physics of light – not a lot, but just enough to make our 3D scene look more realistic.

White light contains all the colors that our human eyes can see. This can be demonstrated by shining a white light into a prism, which makes the light split into a rainbow.
Another way to demonstrate this is by getting three different colors of light – red, green and blue – and shining them onto a white surface in a dark room. If you were to do this, you would see the image below.







We can draw some conclusions from this:

- White = red + green + blue
- Yellow = red + green
- Cyan (light blue) = blue + green
- Magenta (purplypink)
- = red + blue
- Black = none of the colors

Using only three colors of light, we can make eight different colors: red, green, blue, yellow, cyan, magenta, black and white.

But what about the other colors, like orange? Well, if you take the green light and make it half as bright as it used to be, you would see the image (to the right of the figure).

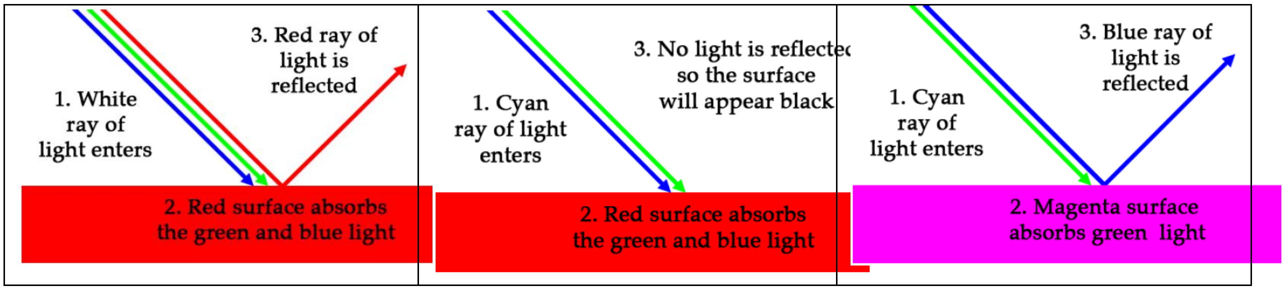
Lowering the intensity (a.k.a. brightness) of the green has made a few new colors: dark green, sky blue, orange, and pink.

As you can see, colors are combinations of different intensities of red, green and blue light. This is why the color of a light is called the *intensities* of the light. When we set the color of the light in the code, we will be using a vec3 to hold the red, green and blue intensities

5.2.2 Absorption & Reflection Of Color

Let's say you're looking at a red car. The sun emits a ray of white light. The ray bounces off the car, and goes into your eye. Your eye detects that the ray only contains red light, which is why you see a red car instead of a white car.

We know that white light contains all colors, so what happened to the green and blue? The green and blue light was absorbed by the surface, and the red light was reflected.


1. White ray of light enters
3. Red ray of light is reflected
2. Red surface absorbs the green and blue light
1. Cyan ray of light enters
3. No light is reflecte so the surface will appear black
2. Red surface absorbs the green and blue light
1. Cyan ray of light enters
3. Blue ray of light is reflected
2. Magenta surface absorbs green light

What if we were to shine a pure cyan (blue + green) light on the red car? If the car was pure red, it would look *black*, because it would absorb 100% of the light.

What about a cyan (blue + green) light on a magenta (red + blue) surface?

So if you shine a light-blue flashlight onto a purply-pink surface, the surface will appear to be dark blue. It's strange, but true.

If you look at the RGB value of each color, you will notice that the values represent reflectance. (0,0,0) is black, which means *reflect none of the light*. (1,1,1) is white, which means *reflect all of the light*. (1,0,0) is red, which means *only reflect the red*. Cyan is (0,1,1), which means *only reflect blue and green*. The RGB color of a surface represents how light is absorbed and reflected by that surface.

Calculating the reflected color is simple. The basic formula is:

***light intensities* × *surface color* = *reflected intensities* .**

For example:

cyan light × magenta surface = blue light

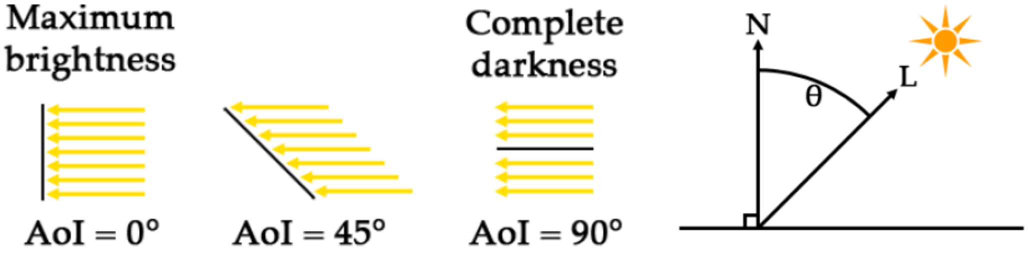
(0, 1, 1) × (1, 0, 1) = (0, 0, 1)

The multiplication is done by multiplying each of the RGB components individually, like so:

$$(X,Y,Z)(A,B,C) = (XA,YB,ZC)$$

5.2.3 Angle of Incidence

the *angle of incidence* (AoI) of the light affects the color of the surface. The surface will be the brightest when it faces directly the light, when the surface is far away it will be more darker.



If we represent the brightness as a single number, where 0.0 is completely dark and 1.0 is maximum brightness, then it's easy to calculate based on the cosine of the AoI. The formula is

brightness = cos(AoI)

It is possible to calculate the angle between two vectors using the *dot product* of the vectors $\vec{v_1} \bullet \vec{v_2}$. The result of the dot product is related to the angle between the two vectors. The exact relationship is:

$$\vec{v_1} \bullet \vec{v_2} = |\vec{v_1}||\vec{v_2}|cos(\theta)$$

$$\frac{\vec{v_1} \bullet \vec{v_2}}{|\vec{v_1}||\vec{v_2}|} = cos(\theta)$$

Where $\vec{v_1}$ and $\vec{v_2}$ are vectors, θ is the angle between the two vectors, and $|v|$ is the magnitude of v .

We don't actually need to know the AoI, we just need cos(AoI) , which represents brightness. Once we have a brightness value between 0 and 1, we can multiply it by the intensities of the reflected light to get the final color for the pixel. Here is an example with cyan light:

brightness × reflected intensities = final color for pixel

1.0 × (0, 1, 1) = (0, 1, 1) (cyan, unchanged)

0.5 × (0, 1, 1) = (0, 0.5, 0.5) (turquoise, which is darkened cyan)

0.0 × (0, 1, 1) = (0, 0, 0) (black)

This "brightness" value between 0 and 1 is sometimes called the "diffuse coefficient." The general formula for diffuse coefficient is:

diffuse coefficient = cos(AoI) × light intensities × surface color

5.3 Ambient Component

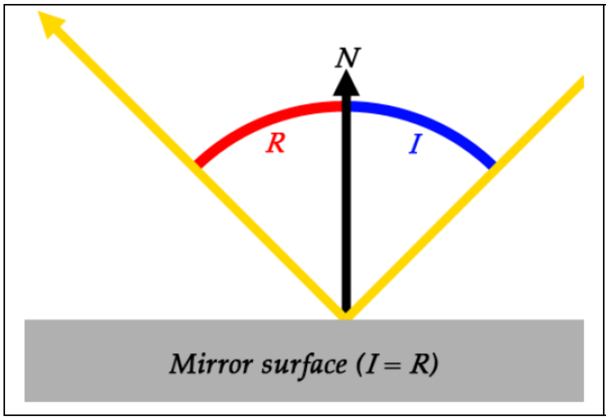
The ambient component of the Phong reflection model basically specifies a minimum brightness. Even if there is no light hitting a surface directly, the ambient component will light up the surface a little bit to stop it from being pure black. The ambient brightness is constant for all surfaces.

Ambient component is calculated using a percentage of the original intensities of the light source. This ambient percentage is stored as a float with a value between zero (0%) and one (100%), in a variable named ambient-value . For example if ambient-value is 0.05 (5%) and the reflected light intensities are $(1, 0, 0)$, which is pure red light, then the ambient component will be $(0.05, 0, 0)$, which is very dim red light.

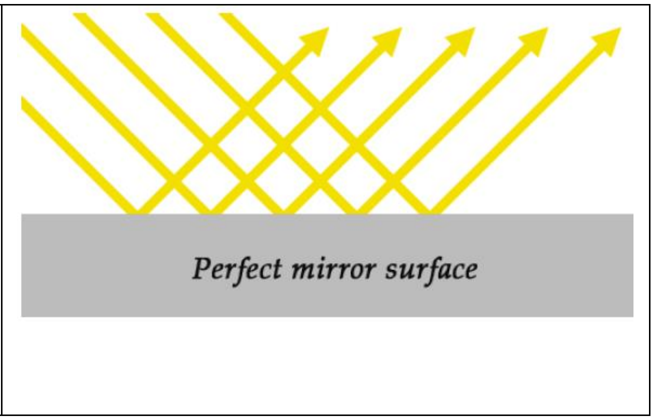
Ambient Component = ambient-value × ***reflected intensities***

5.4 Specular Component

The specular component is what makes a surface look shiny. The word "specular" means "like a mirror," and it is used here because the shiny patches (a.k.a. specular highlights) are fake reflections of light, like a mirror would reflect light. Let's start by looking at how a mirror reflects light.



Reflectance of light



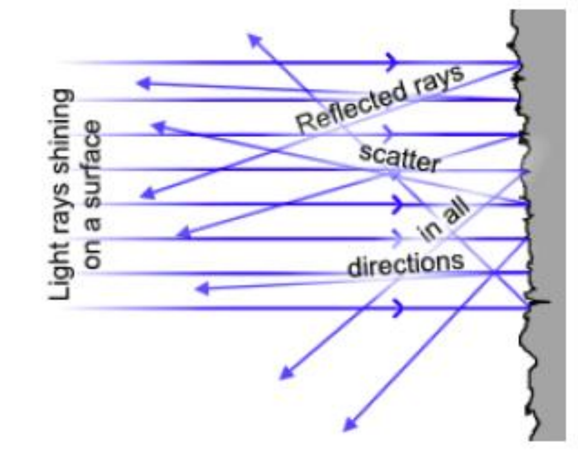
Perfect mirror

N and I are the normal and the angle of incidence (AoI), which we saw in the last section. R is new, and it represents *the angle of reflection* (AoR).

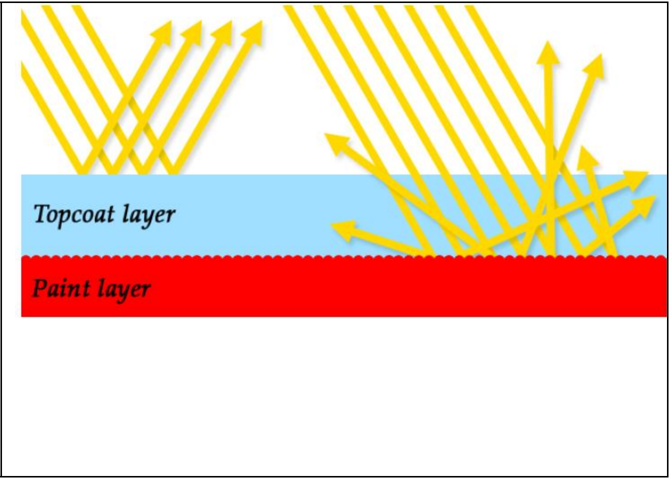
The angle of reflection is the angle between the reflected ray, and the surface normal. It is sort of the opposite of the angle of incidence When light hits a perfect mirror surface, the AoI and AoR are equal. That is, if light comes in at a 30° angle, it will be reflected at a 30° angle. Now let's look at surfaces that do no behave like a perfect mirror.

When light hits an irregular surface, like the one shown above, the light could be reflected in any direction. This is the difference between the diffuse and specular components: the diffuse component models irregular surfaces, and the specular component models mirrorlike surfaces.

Many surfaces have both specular *and* diffuse components to them. The outer surface of a car is a good example. It's not a perfect mirror, but it can be shiny enough to see your reflection in it. This is because the surface has a layer of paint underneath a layer of clear topcoat. The paint layer is diffuse, but the topcoat layer is specular. The topcoat is also clear, so some of the light is reflected, but some of the light goes straight through to hit the paint layer beneath.



Scattered light



Two layer surfaces

Notice how when the topcoat reflects light, the rays don't hit the paint layer. Normally the paint layer would change the color of the light by absorbing some of the intensities, but that can't happen if the light doesn't hit the paint. This means that the specular component is usually a different color to the diffuse component. Most specular surfaces don't absorb anything, they just reflect all of the light, which means the specular color would be white. This is why the shiny parts of a car are white, even though the paint is red.

To calculate the specular component, we basically solve this question: if the surface was a perfect mirror, would it be reflecting rays from the light source straight into the camera? To get the answer, we:

- Calculate the *incidence vector*, which is a vector from the light to the surface.
- Calculate the *reflection vector*, based on the surface normal and the incidence vector, using the AoI = AoR rule.
- Calculate a vector from the surface to the camera.
- Get the angle between the reflection vector and the surface-to-camera vector.

- If the angle is small, then we conclude that, yes, the light *is* being reflected straight into the camera.

Just like the diffuse component, we won't actually calculate the angle. We will just use cos(angle) instead, because it ranges from zero to one, which is more useful.

How small does the angle have to be? Well that depends on how shiny the surface is. We need a variable to represent shininess, and this variable is called the "specular exponent." The larger the specular exponent, the more shiny the surface will be. It's up to the artist to play around with this value, until it looks right.

To apply the specular exponent, we take cos(angle) and raise it to the power of the specular exponent. This produces the "specular coefficient", which is the brightness of the reflection.

$$x = cos(\theta)^s$$

x *is the specular coefficient*

θ *is the angle*

s *is the specular exponent*

5.5 Attenuation

if you move a candle away from a surface in real life, then the surface gets darker. To do this, we will implement attenuation. Attenuation is the loss of light intensity over distance. The greater the distance, the lower the intensity. We will represent attenuation as a percentage of remaining light, in a float with a value between zero and one. For example, an attenuation value of 0.2 means that 80% of the light intensity has been lost, and only 20% of the intensity remains.

In the real world, attenuation is proportional to the inverse of the distance squared:

$$i \propto \frac{1}{d^2}$$

i *is the intensity*

d *is the distance*

We'll use a modified version of this formula. Firstly, we want to avoid divide-by-zero errors if the distance is zero, so we modify the formula slightly to get this:

$$a = \frac{1}{1 + d^2}$$

a *is the attenuation*

d *is the distance*

Now, if d is zero, then a will be 1, which means that the light is at maximum intensity.

Secondly, we might want to control how fast the intensity decreases over distance. Maybe we want some lights to shine over very long distances without much attenuation, and other lights to only shine short distances with lots of attenuation. To control the attenuation we will add a variable, which we will just call K:

$$a = \frac{1}{1 + kd^2}$$

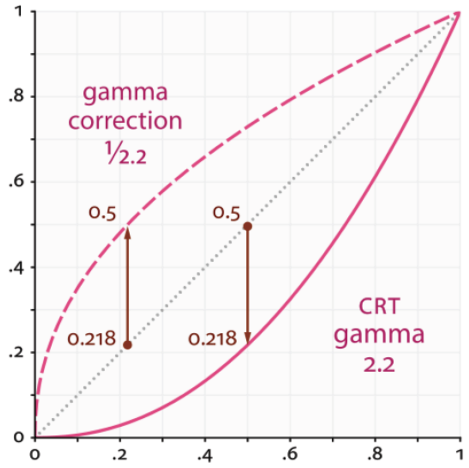
a is the attenuation
d is the distance
k is an arbitrary "attenuation factor"

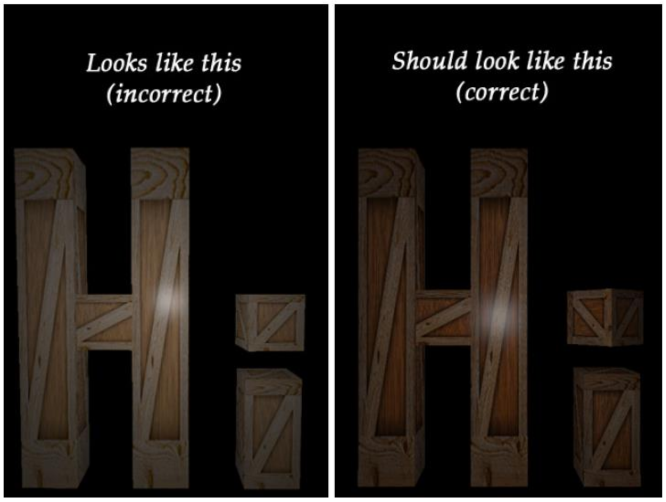
We will apply attenuation to the diffuse and specular components, but not the ambient component. Remember that the ambient component is a constant minimum brightness, so it doesn't make sense to lower the brightness below the minimum.

5.6 Gamma Correction

All of our lighting calculations so far have assumed that we are working a *linear color space*. In a linear color space, if you double one of the RGB color values, then the pixel on the screen should be twice as bright. For example, the 100% red color (1,0,0) should be twice as bright as the 50% red (0.5, 0, 0) color.

The problem is that computer screens do *not* display colors in a linear color space. The 100% red is actually about 4.5 times brighter than the 50% red, which makes the brightness in the 3D scene look wrong. This is because computer monitors mimic the way that old CRT monitors behaved. Gamma correction is also necessary because of the way that the human eye perceives brightness.




Looks like this (incorrect)

Should look like this (correct)

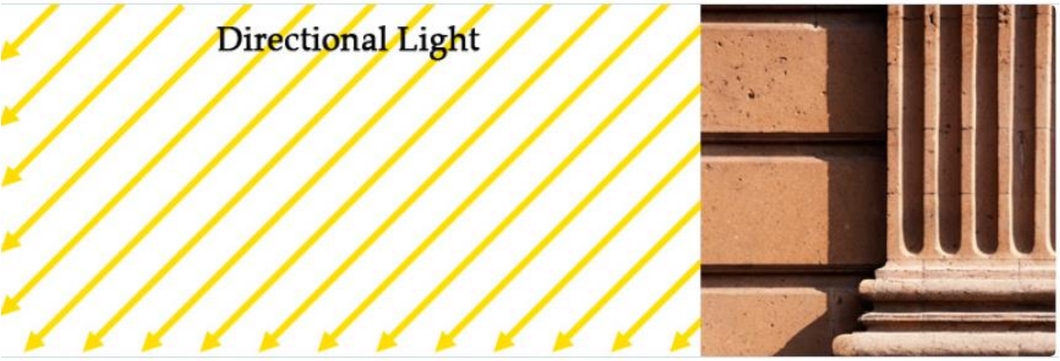
Th dotted line in the middle represents the linear color space we are working in. The solid line at the bottom represents the color space that computer monitors display. Notice how 0.5 on the dotted line matches up to 0.218 on the solid line. This means that if we calculate an RGB value of 0.5, it will actually look like 0.218 when it gets displayed on the monitor, so everything would look too dark. The dashed line at the top represents an RGB value after it has been gamma

corrected, which increases the brightness. The gamma corrected color is too bright, and the monitor is too dark, so when they are combined the result looks correct.
Also notice how the lines meet up at zero and one. This means that gamma correction doesn't affect maximum and minimum brightness – it affects all the shades of brightness in the middle.

Gamma correction is pretty simple to implement. You take each of the RGB values and raise them to the power of *gamma*. Some games give the user a brightness setting which allows them to change the gamma value, but we will just use the constant value $\frac{1}{2.2}$, which is the correct value for CRT monitors.

5.7 Directional Lights

Directional lights are lights that shine in a single, uniform direction. That is, all rays of light are parallel to each other. Pure directional lights do not exist (except maybe lasers?) but they are often used in computer graphics to imitate strong light sources that are very far away, such as the Sun.
The Sun radiates light in all directions, like a point light. Over an enormous distance, however, the tiny fraction of light rays that make it to earth appear to be almost parallel.



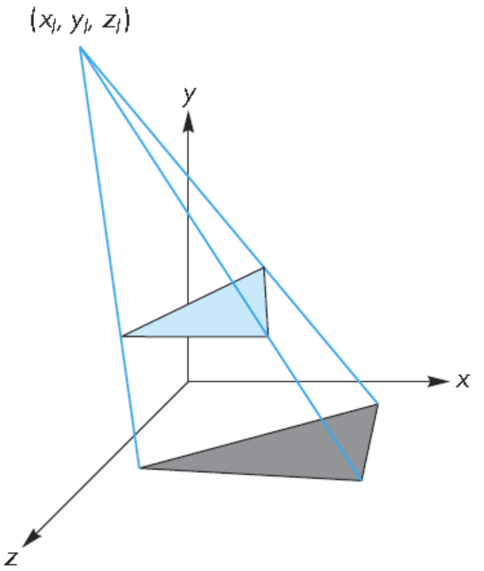
directional lights can be thought of as point lights that are infinitely far away. This causes an unfortunate interaction with attenuation. Attenuation is the reduction of light intensity over distance – the greater the distance, the dimmer the light is. If there is even the tiniest amount of attenuation, over an infinite distance the light becomes infinitely dim (i.e. invisible). For this reason, directional lights are implemented such that they ignore attenuation. This kind of makes sense if we're using directional lights to represent the Sun, because the sunlight we see here on Earth doesn't appear to be attenuated. That is, sunlight doesn't appear to get dimmer as it gets closer to the ground.

Unlike point lights, directional lights do not need a position coordinate. A directional light only needs a single 3D vector that represents the direction of all the rays of light. It has no actual position.

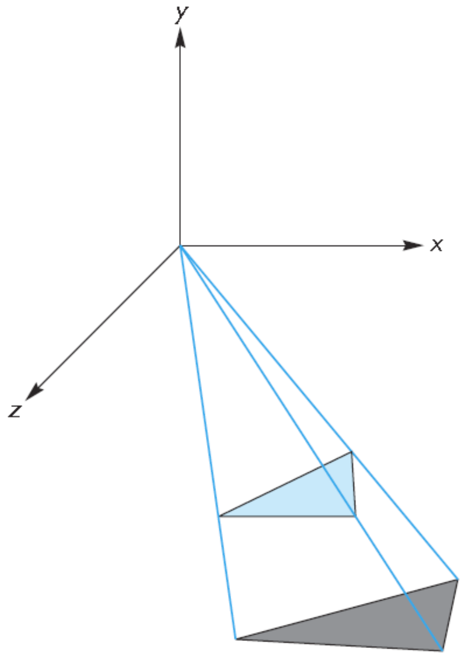
5.8 Shadows

Starting from a physical point of view, shadows require a light source to be present. A point is in shadow if it is not illuminated by any light source or, equivalently, if a viewer at that point cannot see any light sources.

We assume for simplicity that the shadow falls on the ground or the surface,
y = 0.



Shadow from a single polygon



Light source is moved to the origin

Not only is the shadow a flat polygon, called a **shadow polygon**, but it also is the projection of the original polygon onto the surface. Specifically, the shadow polygon is the projection of the polygon onto the surface with the center of projection at the light source.

Suppose that we start with a light source at *(xl , yl , zl)*, as shown in Figure 4.46(a). If we reorient the figure such that the light source is at the origin, as shown in Figure 4.46(b), by a translation matrix **T***(−xl , −yl , −zl)*, then we have a simple perspective projection through the origin. The projection matrix is

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{-y_l} & 0 & 0 \end{bmatrix}.$$

Finally, we translate everything back with **T***(xl , yl , zl)*. The concatenation of this matrix and the two translation matrices projects the vertex *(x, y, z)* to

$$x_p = x_l - \frac{x - x_l}{(y - y_l)/y_l},$$

$$y_p = 0,$$

$$z_p = z_l - \frac{z - z_l}{(y - y_l)/y_l}.$$

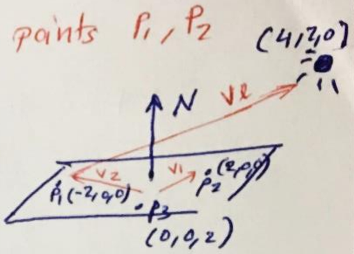
5.9 Summery

We have introduced lights and light types. How to calculate the change in color intensity of the object according to the light distance to the object surface and the angle it makes with the surface normal.

Example: Surface with 3 Points $P_1(-2,0,0)$, $P_2(2,0,0)$, $P_3(0,0,2)$. Its Color is red, the light source is located at $P_L(4,2,0)$.

Calculate the red Color Intensity at points P_1, P_2

Solution:



★ Find Nomal $N = V_1 \times V_2$

$$V_1 = P_2 - P_3 = (2,0,0) - (0,0,2) = (2,0,-2).$$

$$V_2 = P_1 - P_3 = (-2,0,0) - (0,0,2) = (-2,0,-2).$$

$$N = V_1 \times V_2 = \begin{vmatrix} u_x & u_y & u_z \\ 2 & 0 & -2 \\ -2 & 0 & -2 \end{vmatrix} = 0\,u_x + 8\,u_y + 0\,u_z$$

$$= (0,8,0)$$

Normalize N. $\|N\| = \sqrt{0^2+8^2+0^2} = \sqrt{8^2} = 8$

$$N = \frac{(0,8,0)}{\|N\|} = \frac{(0,8,0)}{8} = (0,1,0)$$

★ Find vector from Point P_1 to light source.

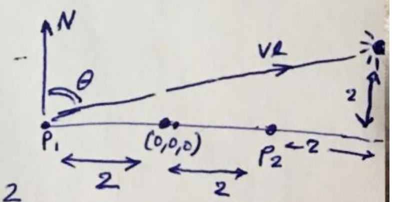
$$V_\ell = (4,2,0) - (-2,0,0) = (6,2,0)$$

$$\|V_\ell\| = \sqrt{6^2 + 2^2 + 0^2} = \sqrt{36+4} = \sqrt{40}$$

$$V_\ell = \left(\frac{6}{\sqrt{40}}, \frac{2}{\sqrt{40}}, 0\right)$$

★ Find ~~the an~~ cos the angle between Normal N and V_ℓ.

$$\cos(\theta) = \frac{N \cdot V_\ell}{\|N\| \, \|V_\ell\|}$$



$$= \frac{(0,1,0)\cdot\left(\frac{6}{\sqrt{40}}, \frac{2}{\sqrt{40}}, 0\right)}{\sqrt{0^2+1^2+0^2} \times \sqrt{\left(\frac{6}{\sqrt{40}}\right)^2 + \left(\frac{2}{\sqrt{40}}\right)^2 + 0^2}}$$

$$= \frac{\frac{2}{\sqrt{40}}}{\sqrt{\frac{36}{40} + \frac{4}{40}}} = \frac{\frac{2}{\sqrt{40}}}{\sqrt{\frac{40}{40}}} = \frac{2}{\sqrt{40}}$$

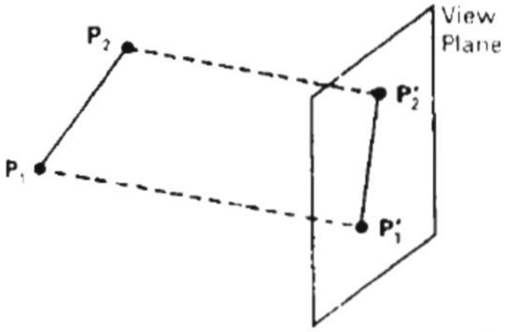
$$= 0.3162$$

★ Color Intensity $= (1,0,0) \times 0.3162 = (0.3162, 0, 0)$

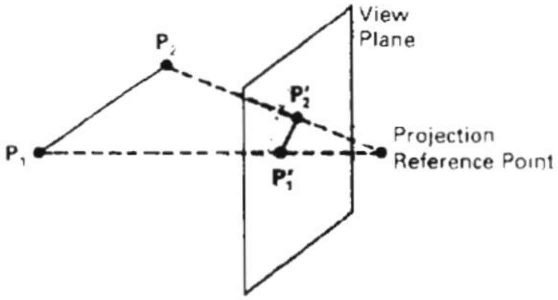
Chapter 6
Camera Transformations

Once world-coordinate descriptions of the objects in a scene are converted to viewing coordinates, we can project the three-dimensional objects onto the three dimensional view plane. (Hearn 2010 [12]) There are two basic projection methods.

- In a parallel projection, coordinate positions are transformed to the view plane along parallel lines
- For a perspective projection, object positions are transformed to the view plane along lines that converge to a point called the projection reference point (or center of projection COP). The projected view of an object is determined by calculating the intersection of the projection lines with the view plane.



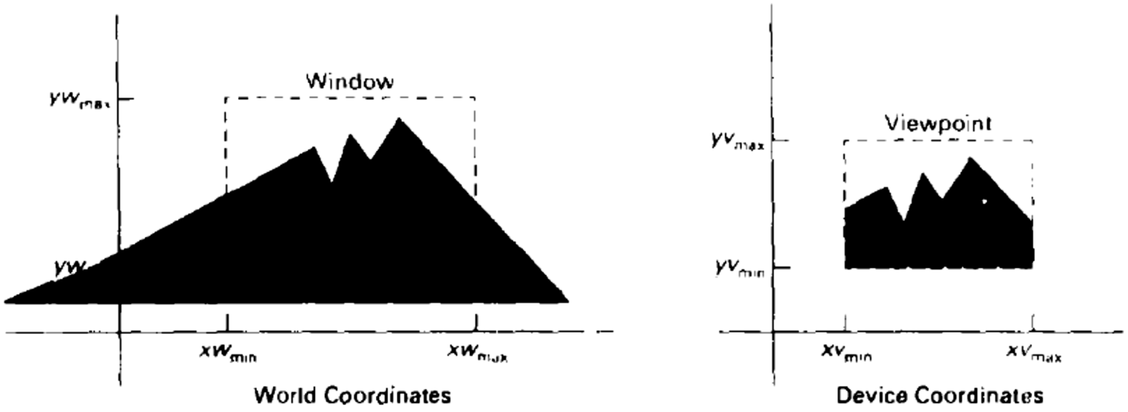
Parallel projection (example Orthographic projection)



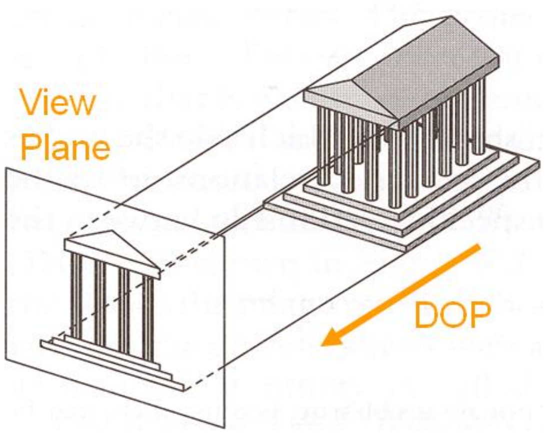
Perspective Projection

6.1 Orthogonal Projection

Window: A world-coordinate area selected for display is called a window.
Viewport: An area on a display device to which a window is mapped is called a viewport.
The window defines ***what*** is to be viewed;
the viewport defines ***where*** it is to be displayed
The dimensions ratios are kept.
Camera Transformation is the mapping of a part of a world-coordinate scene to device coordinates is referred to as a viewing transformation or camera transformations.



In orthographic projection**, s**imilar far objects and near objects are appearing with the same dimension ratios.



- *DOP perpendicular to view plane*

<u>Simple Orthographic Transformation</u> is to use the same values of real world coordinate as pixel coordinate the following matrix do the simple Ortho transformation

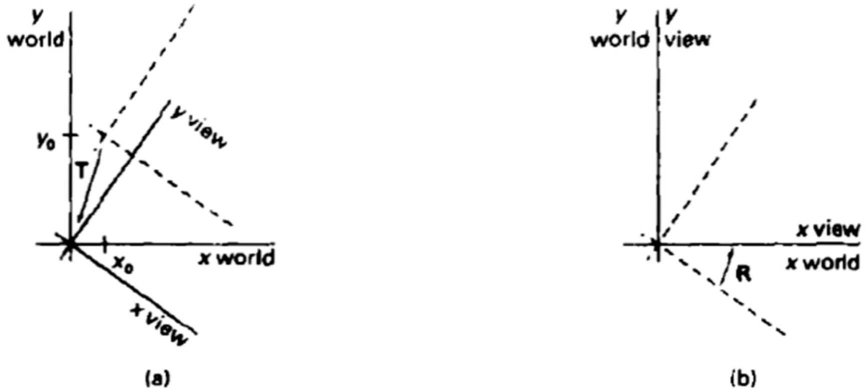
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

In Simple Ortho transformation, Original world units are preserved, this is not good solution because pixel units are preferred.

Orthographic Projection Procedure

To convert the real world vertices to device vertices (pixels) we do the following operations.

1. Translate the world so that the origin **(x0, yo)** of the **x'y'** system (the world) is moved to the origin of the **xy** system (the camera).
2. Rotate the **x'** axis onto the **x** axis.



A viewing-coordinate frame is moved into coincidence with the world frame in two steps: (a) translate the viewing origin to the world origin, then (b) rotate to align the axes of the two systems.

Translation of the coordinate origin is expressed with the matrix operation

$$T(-x_0, -y_0) = \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{bmatrix}$$

Simple camera rotation can be done with the R matrix if we know (θ)

$$R(-\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

In addition, the elements of any rotation matrix could ***be*** expressed as elements of a set of orthogonal unit vectors. Therefore, the matrix to rotate the **x' y'** system into coincidence with the *x y* system can be written as

$$R = \begin{bmatrix} u_x & u_y & 0 \\ v_x & v_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

We obtain the matrix for converting world coordinate positions to viewing coordinates as a two-step composite transformation, First step is to: use T to translate the viewing origin to the world origin, the Second step is to: use R to rotate and align the two coordinate reference frames.
The composite two-dimensional transformation to convert world coordinates to viewing coordinate is

$$\mathbf{M}_{WC,VC} = \mathbf{R} \cdot \mathbf{T}$$

where T is the translation matrix that takes the viewing origin point **Po** to the world origin, and R is the rotation matrix that aligns the axes of the two reference frames

- window-to-viewport coordinate transformation

A point at position (***xw,yw***) in the window is mapped into position ***(xv, yv***) in the associated viewport. To maintain the same relative placement in the viewport as in the window, we require that

$$\frac{xv - xv_{\min}}{xv_{\max} - xv_{\min}} = \frac{xw - xw_{\min}}{xw_{\max} - xw_{\min}}$$

$$\frac{yv - yv_{\min}}{yv_{\max} - yv_{\min}} = \frac{yw - yw_{\min}}{yw_{\max} - yw_{\min}}$$

Solving these expressions for the viewport position ***(xv, yv) ,*** we have

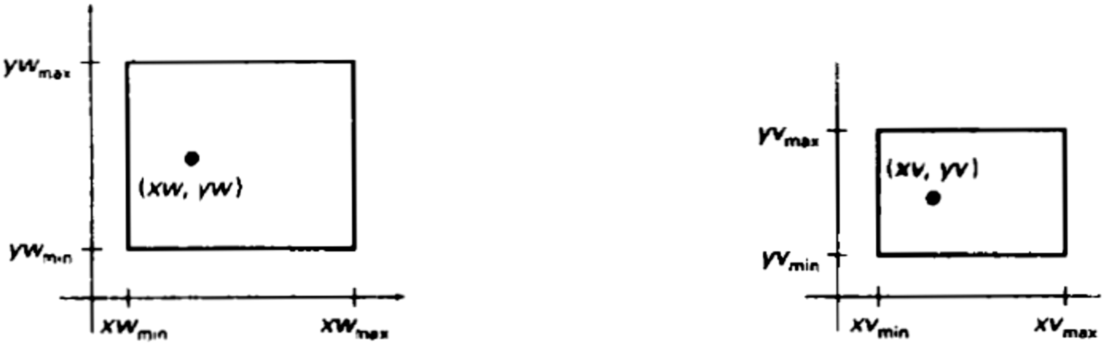
$$xv = xv_{\min} + (xw - xw_{\min})sx$$

$$yv = yv_{\min} + (yw - yw_{\min})sy$$

where the scaling factors are

$$sx = \frac{xv_{\max} - xv_{\min}}{xw_{\max} - xw_{\min}}$$

$$sy = \frac{yv_{\max} - yv_{\min}}{yw_{\max} - yw_{\min}}$$



A point at position ***(xw,yw)*** in a designated window is mapped to viewport coordinates ***(xu, yv)*** so that relative positions in the two areas are the same.

Example: A room with two persons in it , screen size are 600x800, world is 5 meters x 4 meters x 3 height meters. The two persons are identified by their bounding box in the world coordinates,

Person 1: p1(1,0,2), p2(2,2,2).

Person 2: p3(3,0,0), p2(4,2,0).

Camera at 2,1,7 and looking down z-direction, do Orthographic projection on the 5 meters wall.

Solution:

- Transfer the (0,0,0) world origin to the camera position (2,1,7), we ignore z component because this is an orthographic projection. We transfer by applying Translate transform with the following T matrix

$$T = \begin{vmatrix} 1 & 0 & -2 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{vmatrix}$$

- Rotate the world to align with camera coordinate. Since the camera is looking toward negative z, and no rotation in it. So the v vector is to the right and aligned with x axis. And the up vector u is aligned with y, so the rotation matrix will be

$$R = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Which is the identity matrix. This means that the world and camera coordinates are aligned together.

The resulting window transformation matrix is

$$M = R.T = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 0 & -2 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} 1 & 0 & -2 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{vmatrix}$$

After multiplying each point in the world coordinate with the matrix M, The resulting vertices are p1(-1,-1), p2(0,1), p3(1,-1), p4(2,1)

- Calculate the scaling factor for width and height of the device dimensions related to the world (window) width and height. It is known that after applying M to the vertices, the camera is located at (0,0) in the world, position of the camera on the screen should be defined.

$Sx = \frac{800-0}{5} = 160, Sy = \frac{600-0}{3} = 200,$

For camera position (0,0)

camX=0+(0-(-2) Sx =2Sx=2*160=320

camY=0+(0-(-1))Sy=1*Sy=200

For P1(-1,-1)

P1x=0+(-1-(-2) Sx =Sx=160

P1y=0+(-1-(-1))Sy=0*Sy=0

For P2(0,1)

P2x=0+(0-(-2) Sx =2Sx=2*160=320

P2y=0+(1-(-1))Sy=2*Sy=400

For P3(1,-1)

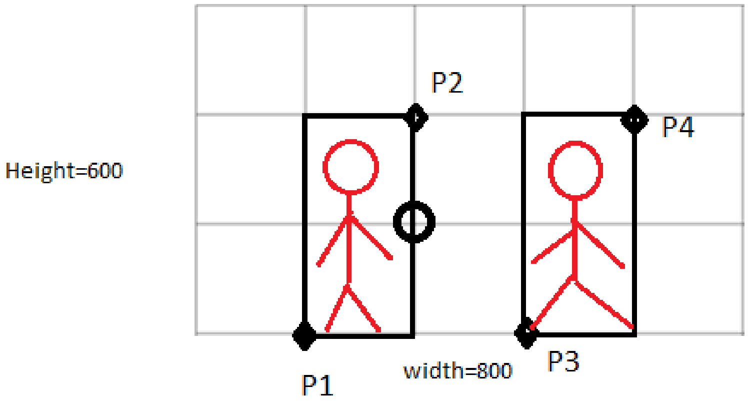
P3x=0+(1-(-2) Sx =3Sx=3*160=480

P3y=0+(-1-(-1))Sy=0*Sy=0

For P4(2,1)

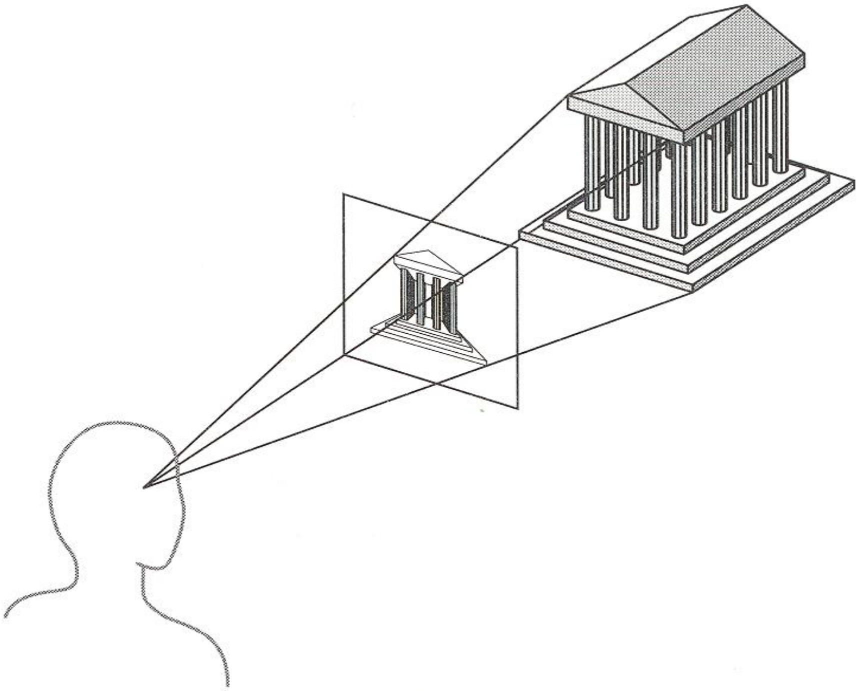
P4x=0+(2-(-2) Sx =4Sx=4*160=640

P4y=0+(1-(-1))Sy=2*Sy=2*200=400



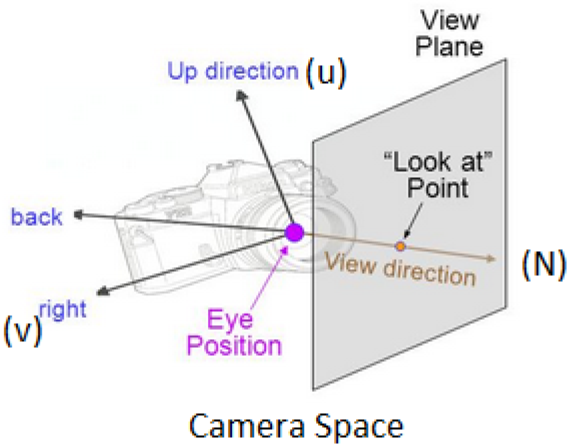
6.2 perspective Projection

First discovered by Donatello, Brunelleschi, and DaVinci during Renaissance
Objects closer to viewer look larger
Parallel lines appear to converge to single point called the vanishing point.
In the real world, objects exhibit perspective foreshortening: distant objects appear smaller

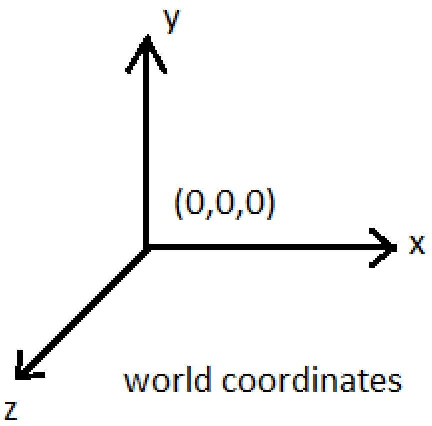


The camera has the following parameters

- The up direction (u), which is the direction up on the camera.
- The right direction (v), this direction points to the right, if we consider the camera is person’s eye.
- The normal direction N, which is the normal direction on the view plane.
- The look at point which is the point in the world scene that the camera is looking at.



Camera Space



world coordinates

To do perspective projection, we follow the following steps

1. Translate the view reference point to the origin of the world-coordinate system.
2. Apply rotations to align the **x,,** y,, and z,. axes with the world ***x,*** y,., and z,axes, respectively.

If the view reference point is specified at world position **(xo** yo, zo), this point is translated to the world origin with the matrix transformation

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Another method for generating the rotation-transformation matrix is to calculate unit uvn vectors and form the composite rotation matrix directly**.** given vectors N and V, these unit vectors are calculated as

$$\mathbf{n} = \frac{\mathbf{N}}{|\mathbf{N}|} = (n_1, n_2, n_3)$$

$$\mathbf{u} = \frac{\mathbf{V} \times \mathbf{N}}{|\mathbf{V} \times \mathbf{N}|} = (u_1, u_2, u\ )$$

$$\mathbf{v} = \mathbf{n} \times \mathbf{u} = (v_1, v_2, v_3)$$

This method also automatically adjusts the direction for V so that v is perpendicular to n. The composite rotation matrix for the viewing transformation is then

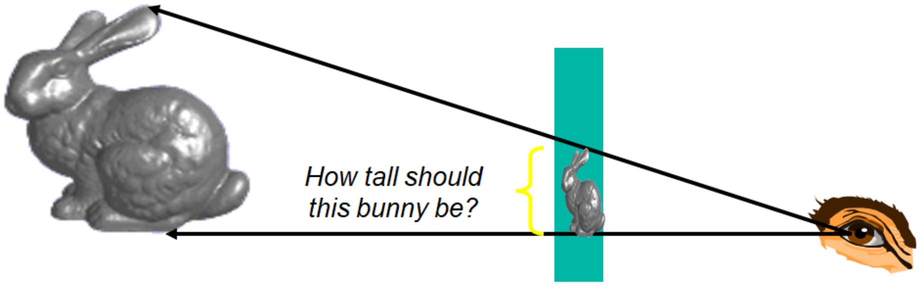
$$\mathbf{R} = \begin{bmatrix} v_1 & v_2 & v_3 & 0 \\ u_1 & u_2 & u_3 & 0 \\ n_1 & n_2 & n_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

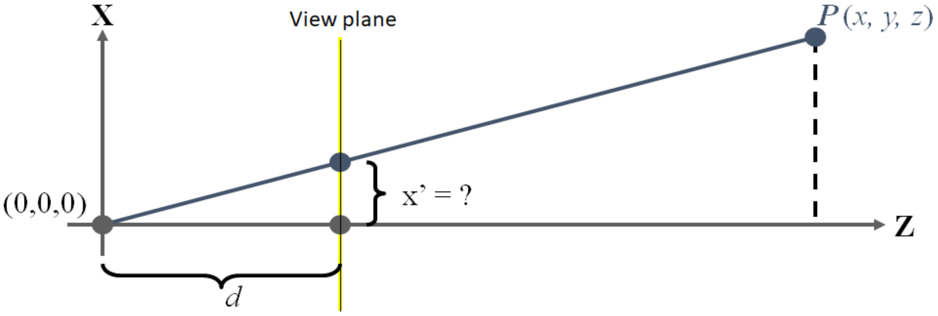
The complete world-to-viewing coordinate transformation matrix is obtained as the matrix product

$$\mathbf{M}_{WC,VC} = \mathbf{R} \cdot \mathbf{T}$$

3. Calculate the projected vertices in the camera coordinates. Vertices on the view plane not on the monitor screen.

When we do 3-D graphics, we think of the screen as a 2-D window onto the 3-D world: The geometry of the situation is that of similar triangles. View from above:





What is the value of x'?

Desired result for a point $[x, y, z, 1]^T$ projected onto the view plane:
What could a matrix look like to do this?

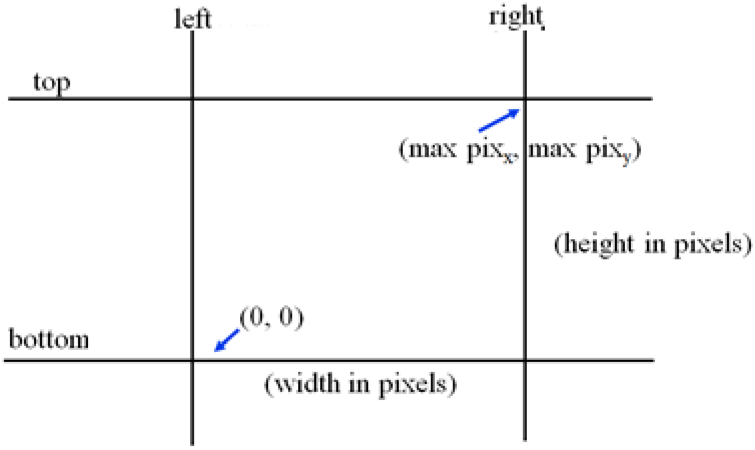
$$\frac{x'}{d} = \frac{x}{z}, \quad \frac{y'}{d} = \frac{y}{z}$$

$$M_{perspective} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

$$x' = \frac{d \cdot x}{z} = \frac{x}{z/d}, \quad y' = \frac{d \cdot y}{z} = \frac{y}{z/d}, \quad z = d$$

4. Calculate camera-to-viewport coordinate transformation: Use the following matrix to do the transformation

In the figure (0,0) is the device or the monitor coordinates.



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \frac{width}{right - left} & 0 \\ 0 & \frac{height}{top - bottom} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- left, right, top, bottom refer to the viewing frustum in modeling coordinates width and height are in pixel units
- This matrix scales and translates to accomplish the transition in units

Example: Do perspective projection for example 1. A room with two persons, screen size is 600x800, world is 5 meters x 4 meters x 3 height meters. Two persons are identified by their bounding box are in the world,
Line 1: p1(1,0,2), p2(2,2,2).
Line 2: p3(3,0,0), p2(4,2,0).
Camera at 2,1,7 and looking down z-direction, the projection plane is in front of the camera and separated by 2 meters (near), the far plane is 7 meters away of the camera (the x-y plane)
Solution: We need to project the two persons in the real-world room into the 600x800 pixels screen.
use this website for MATLAB calculations https://octave-online.net/

1. Translate the world origin to the camera position. The translate matrix is:

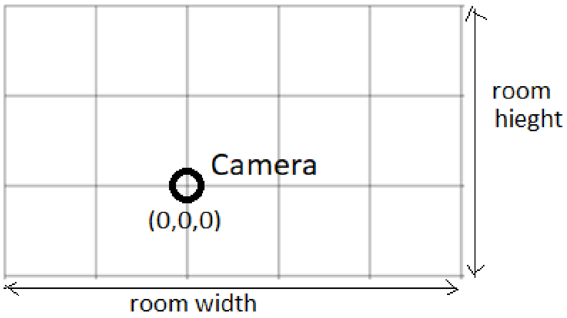
$$T = \begin{bmatrix} 1 & 0 & 0 & -2 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -7 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The resulting points are P1(-1,-1,-5), P2(0,1,-5), P3(1,-1,-7), P4(2,1,-7)

2. Rotate the world to align with camera axis. Calculate the camera vectors. Up camera vector is directed with y axis. So U=(0,1,0). Normal vector is directed toward negative z, so N=(0,0,-1). Right vector V is obtained with cross product of N and U, N x U = (1,0,0).

$$R = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The resulting points are P1(-1,-1,5), P2(0,1,5), P3(1,-1,7), P4(2,1,7), the camera is located at (0,0,0). left =-2, right =3, top=2, bottom= -1. The camera position is shown in the figure



3. Calculate the projections on the view plane (real-world dimensions)
The new dimensions are calculated by

$$\left(\frac{x}{z/d}, \frac{y}{z/d}, d \right)$$

d=2, so P1=(-1/(5/2), -1/(5/2), 2)=(-2/5,-2/5,2), similarly for P2, P3, P4 yielding P2(0,2/5, 2), P3(2/7,-2/7, 2). P4(4/7, 2/7,2).

4. Calculate the camera to viewport transformation, use the following matrix, ignoring z component (set it to zero)

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \frac{800}{3-(-2)} & 0 \\ 0 & \frac{600}{2-(-1)} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 160 & 0 \\ 0 & 200 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

The new points are P1(-64,-80), P2(0,80), P3(45,-57), P4(91,57)

Summery:

We have presented orthographic projection and perspective projections with two examples that show how to do that with real world examples.

References

[1] Tom Dalling Blog “Modern OpenGL Series“, Modern OpenGL 03 Matrices, Depth Buffering, Animation (/blog/modernopengl/03matricesdepthbufferinganimation/)

[28] Earth Sciences Sector of Natural Resources, Canada, Information available at their website http://ess.nrcan.gc.ca/index_e.php, Last updated 2006.

[11] F. Klawonn, *Introduction to Computer Graphics*, Undergraduate Topics in Computer Science, Chapter 2, Basic Principles of Two-Dimensional Graphics DOI 10.1007/978-1-4471-2733-8_2, © Springer-Verlag London Limited 2012

[12] Hearn, Donald D., M. Pauline Baker, and Warren Carithers. *Computer graphics with open GL*. Prentice Hall Press, 2010.

[13] Wolfe, Rosalee. "Bringing the introductory computer graphics course into the 21st century." *Computers & Graphics* 24.1 (2000): 151-155.

Similar Courses

[1] CS 425/625 Computer Graphics,

http://orca.st.usm.edu/~jchen/courses/graphics/lectures.htm

[2] CS 445: Introduction to Computer Graphics

https://www.cs.virginia.edu/~asb/teaching/cs445-fall06/

[3] Fundamental of Computer Graphics, Prof. Fabio Pellacini

http://pellacini.di.uniroma1.it/teaching/graphics13a/index.html